Linköping Studies in Science and Technology
Dissertation No. 1004

Admissible Heuristics for Automated Planning

by

Patrik Haslum

Linköpings universitet
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköpings universitet
SE-58183 Linköping, Sweden

Linköping 2006

Abstract

The problem of domain-independent automated planning has been a topic of research in Artificial Intelligence since the very beginnings of the field. Due to the desire not to rely on vast quantities of problem specific knowledge, the most widely adopted approach to automated planning is search. The topic of this thesis is the development of methods for achieving effective search control for domain-independent optimal planning through the construction of admissible heuristics. The particular planning problem considered is the so called "classical" AI planning problem, which makes several restricting assumptions. Optimality with respect to two measures of plan cost are considered: in planning with additive cost, the cost of a plan is the sum of the costs of the actions that make up the plan, which are assumed independent, while in planning with time, the cost of a plan is the total execution time – makespan – of the plan. The makespan optimization objective can not, in general, be formulated as a sum of independent action costs and therefore necessitates a problem model slightly different from the classical one. A further small extension to the classical model is made with the introduction of two forms of capacitated resources. Heuristics are developed mainly for regression planning, but based on principles general enough that heuristics for other planning search spaces can be derived on the same basis. The thesis describes a collection of methods, including the h^m, additive h^m and improved pattern database heuristics, and the relaxed search and boosting techniques for improving heuristics through limited search, and presents two extended experimental analyses of the developed methods, one comparing heuristics for planning with additive cost and the other concerning the relaxed search technique in the context of planning with time. Experiments aim at discovering the characteristics of problem domains that determine the relative effectiveness of the compared methods; results indicate that some plausible such characteristics have been found, but are not entirely conclusive.

Parts of the material in this thesis has previously appeared in the following papers:

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 140 – 149. AAAI Press.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proc. 6th European Conference on Planning (ECP'01)*, 121 – 132.

Haslum, P. 2004. Improving heuristics through search. In *Proc. European Conference on AI (ECAI'04)*, 1031 – 1032.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proc. 20th National Conference on AI (AAAI'05)*, 1163 – 1168.

Haslum, P. 2006. Improving heuristics through relaxed search – an analysis of TP4 and HSP$^*_a$ in the 2004 planning competition. *Journal of AI Research* 25.

Contents

1. Introduction

Planning is the art and practice of thinking before acting: of reviewing the courses of action one has available and predicting their expected (and unexpected) results to be able to choose the course of action most beneficial with respect to one's goals. Planning, alongside learning and language, is considered a hallmark of intelligence and, not surprisingly, automatic planning has been one of the key goals of research in Artificial Intelligence (AI) since the very beginnings of the field.

If planning is an archetypal AI problem, then search is the archetypal AI solution. This thesis is about the application of search to planning. Specifically, it is about the construction of admissible heuristics for the so called "classical" AI planning problem. Admissible heuristics are essential for effectively solving hard combinatorial problems, like the planning problem, optimally, *i.e.*, for finding plans with guaranteed minimal cost. Two measures of plan cost are considered: planning with additive cost (the cost of a plan is the sum of the costs of the actions that make up the plan; here called *sequential* planning) and planning with time (here called *temporal* planning; the cost of a plan is in this case the total execution time – *makespan* – of the plan).

The Classical AI Planning Problem

Planning is an activity as varied as it is ubiquitous, yet whatever form the problem takes, two facts remain. First, planning requires the ability to predict the results of a plan of action, without actually having to carry it out, and prediction requires a model of the dynamic system whose future state is to be predicted (in the literature on AI planning, this is often referred to as the *world model*). Second, to plan requires some notion of a *goal*.

To automate the process of planning requires a formal statement, or model, of the problem. By varying the form in which the world model and the goal are expressed and the properties they are assumed to have one can arrive at several different models of what the planning problem is. In the study of automated planning certain problem models have received particular interest: among them is the classical AI planning problem (so called because it was formulated in the course of some of the earliest efforts to construct automated reasoning systems, *e.g.*, by Green, 1969, and Fikes and Nilsson, 1971), but also other models of planning (such as motion or traffic planning; see *e.g.* Hwang & Ahuja, 1992, or Patriksson, 1994, respectively).

Underlying the classical AI planning model is a view of planning as a state transformation problem: the world is seen as a space of states and the task of the planner is to drive, through the application of actions, the world to a state satisfying the stated goal. Several limiting assumptions are made: that world states are discrete, that the

planner's model of the world is correct and complete, that the actions available to the planner affect the world in predictable and deterministic ways and that the world is static, in the sense that nothing besides the planned actions causes change to the world (the assumptions of correctness and determinism of the model should not be interpreted as an assumption that the world is modelled perfectly or that actions never fail, but rather that the model is correct at the modelled level of abstraction and that action failures, or other departures from the model, are infrequent enough that it is not worthwhile to take them into account when planning). There are, however, many variations of the classical planning model and the exact boundaries of what is "classical planning" are not easy to determine. A longer discussion of this topic is made in chapter 2 (section 2.3).

In spite of the limitations of the classical planning model there are numerous applications, *e.g.* in the control of mobile robots (Beetz 2002), other robotic systems (Jonsson *et al.* 2000) and industrial processes (Aylett *et al.* 1998; 2000) or in the planning of large-scale human activities such as rescue operations (Biundo & Schattenberg 2001), that fall, if not within the model, then close enough that the methods developed for this problem are applicable with some modification.

There is in AI research a tradition of seeking solutions that combine general algorithmic techniques with declarative formalized knowledge of the specific problem and in the area of planning this is manifested in the quest for *domain-independence*, *i.e.*, a separation of the planning engine from the world model, which is given to the automated planner as part of the problem, alongside the goal, in a suitable formalism. It is this ambition of generality that makes the classical AI planning problem hard, even with its many simplifying restrictions. In the complexity theoretic sense, it is as hard as the hardest problem that can be formulated in the planner's input language, and the great variety in the problems that a domain-independent automatic planner can be required to deal with is also a real and practical difficulty (consider, as an exercise, the multi-UAV mission, $(n^2 - 1)$-Puzzle and MRCPS example domains described in chapters 2, 4 and 5, respectively, and what the differences between domain-specific solution methods for each of them would be).

The world model and the goal are necessary inputs to the planning process, since they inform the planner of what it needs to achieve and what means are at its disposal, and in much of the work on classical planning it is assumed that this is the only knowledge of the specific problem available to the planner. In many instances of the planning problem, however, a lot of additional knowledge about the specific domain is available, which may be taken advantage of to simplify the solving of problems in the domain and thus to improve the performance of an automated planner. Thus, there exists for approaches to automated planning a scale from "knowledge-light" (relying only on the basic problem specification) to "knowledge-intensive" (requiring large amounts of additional problem knowledge to function efficiently). In most cases where automated planning has been applied to real-world problems, domain-specific knowledge has been incorporated in the model (see *e.g.* Jonsson *et al.*, 2000, or Wilkins & desJardins, 2000) but there are also examples of applications for which this has not been required (see *e.g.* Ruml, Do & Fromherz, 2005).

Admissible Heuristics for Automated Planning

The two desiderata of domain independence and of requiring no more problem knowledge than necessary make search the most widely adopted – if not the only – approach to automated planning (though there are many different ways in which this search can be made; again, see the discussion in section 2.3 of chapter 2). Search control is essential for making this approach efficient and *heuristics*, functions estimating distance in the search space, are an important search control technique. To ensure that plans found are optimal, *i.e.*, guaranteed to have minimum cost according to some measure of plan cost, heuristics are required to be *admissible*.

The topic of this thesis is the development of methods for automatic and domain-independent creation of admissible heuristics for planning search. The problem model considered is essentially the classical one, though the two measures of plan cost considered require somewhat different models. In sequential planning, or planning with additive cost, the objective is to minimize the sum of the costs of actions in the plan, which are assumed to be independent. In planning with time, each action has a specified duration and the objective is to minimize the total execution time (makespan) of the plan. This objective can not, in general, be formulated as a sum of independent action costs (because of this, planning with time is sometimes considered to be an extension of the classical model). Another small departure from the classical planning assumptions listed above is also made with the introduction of capacitated resources (described in chapter 5) as in the extended model, world states are no longer discrete. Heuristics are developed for the search space associated with a particular method of plan search, *viz.* regression, but most of the methods are based on principles general enough that methods for the creation of heuristics for other planning search spaces can be derived on the same basis.

Contributions

This thesis compiles work done over the course of several years, and most of the material has been published previously. This work is the result of a long term collaboration, primarily between myself, Héctor Geffner and Blai Bonet.

The main results are the development of two families of admissible heuristics and of several techniques for improving the basic heuristics. The two families of heuristics are the h^m relaxed reachability heuristics for sequential and temporal regression planning (introduced by Haslum & Geffner, 2000; 2001; the second paper also introduced the adaptation of regression search to planning with time and resources) and improved pattern database heuristics for sequential planning (introduced by Haslum, Bonet & Geffner, 2005, building on earlier work on pattern database heuristics for planning by Edelkamp, 2001). The improvements to the pattern database heuristics are the use of a more powerful abstraction (constrained abstraction) and automatic methods for pattern selection. The techniques for improving the basic heuristics have been applied to the h^m heuristics: For the case of sequential planning, an additive version of the heuristic is created through cost relaxation (a generalization of the

principle behind additivity exploited in the construction of pattern database heuristics; cost relaxation and the additive h^m heuristic was also introduced by Haslum, Bonet & Geffner 2005). The relaxed search and boosting methods (introduced in Haslum 2004; 2006) improve heuristics by means of limited search, and are applied to the h^m heuristics for both sequential and temporal regression planning.

The main contribution of this thesis is a coherent, and far more detailed, presentation of the material, not restricted by the chronology (or the, in some cases unfortunate, terminology) of earlier publications, but it also presents some new ideas and results. The formal proofs of a number of properties of the h^m heuristics (in chapter 3), including the equivalence of h^2 and the planning graph heuristic for parallel planning, are previously unpublished, as are the weighted bin-packing pattern selection method (in chapter 4), the use of composite resources for improving heuristics when planning with resources (in chapter 5) and a number of other minor improvements.

The experimental analyses presented here differ from those previously published in that they are, in addition to being more detailed and in some cases more extensive, aimed at discovering characteristics of planning problems that determine the relative effectiveness of the different heuristics. The results of these analyses (particularly those presented in chapters 6 and 7) are also an important contribution of this thesis (although an earlier version of the analysis in chapter 7 appears also in Haslum 2006).

Organization

The next chapter contains background material on the sequential and temporal planning models and regression planning methods that form the context for the main topic of the thesis. The extension of the planning models, and the corresponding regression methods, with certain kinds of resources is described separately in chapter 5.

Chapters 3 and 4 introduce the two basic classes of heuristics, *viz.* the h^m heuristics and pattern database heuristics. Chapter 6 presents the principle of cost relaxation and its application in the additive h^m heuristic for sequential planning, while the relaxed search and boosting techniques are covered in chapter 7.

Discussion and comparison with related work is distributed throughout the presentation. In particular, alternative classical and slightly non-classical planning models are discussed in chapter 2 (section 2.3), except for the treatment of resources in planning, which is instead discussed in chapter 5 (section 5.1).

Likewise, experimental analyses are presented in conjunction with the heuristic or method that is the topic of the analysis. As mentioned above, there are two experimental analyses of particular interest: In chapter 6 (page 105) the three main heuristics for sequential planning (the h^2, additive h^2 and improved pattern database heuristics) are compared, with results suggesting that the relative effectiveness of the heuristics depends on certain properties of the problem domain. In chapter 7 (page 124) the relaxed search technique for improving the h^2 heuristic is compared to the basic h^2 heuristic in the context of temporal planning, resulting in a partial characterization of the problem domains in which relaxed search is cost effective.

2. Planning as Search

To solve any problem algorithmically, a formal model of the problem must be given. To solve a problem by search, the formal model must be translated into a search space, a search algorithm to explore the space must be chosen, and, in most cases, a heuristic is needed to guide the search efficiently. The focus of this work is on the last of these: admissible heuristics for domain-independent planning. Yet, knowledge of the problem model and search space, and even some aspects of the search algorithm, are a necessary background for the main topic. This chapter introduces that background: it presents the first formal models of planning problems considered (which will be somewhat elaborated in chapter 5) and describes how the planning problem can be formulated as a search problem, via a technique known as regression (sections 2.1 and 2.2). Discussion of alternative planning models and ways of searching for plans is postponed to section 2.3 at the end of the chapter.

2.1 Sequential Planning

The classical view of planning in AI is as a state transformation problem: the world model consists of discrete states, and the task of the planner is to find a set of steps to take in order to transform a given initial state into a state satisfying the goal condition (Green 1969). The propositional STRIPS model of planning, introduced in this section, adheres to this view, but is simplified compared to the original STRIPS representation (Fikes & Nilsson 1971). It is essentially the model that was used by Bylander (1991) in his analysis of the computational complexity of the planning problem.

In the sequential case, the primary concern is the selection and ordering of actions in the plan. Exact action execution times (absolute or relative) are not considered. The cost model is additive: every possible action has a cost, and the cost of a plan is the sum of the costs of actions in the plan. In most works on classical planning unit costs are assumed for all actions, so that minimizing plan cost is equivalent to minimizing the length of the plan.

The STRIPS Planning Model

A (propositional) STRIPS planning problem (P) consists of a set of atoms, a set of action descriptions, an initial state description (I) and a goal description (G). Each action a is described by three sets of atoms: the preconditions ($pre(a)$) and the positive ($add(a)$) and negative ($del(a)$) effects. Associated with each action is also a fixed cost ($cost(a)$), a real number greater than zero.

A state of the planning problem (*world state*) is a complete assignment of truth values to the atoms of the problem. World states are normally described by the sets of atoms true in each state. For action a to be *executable* in world state w, all atoms in $pre(a)$ must be true in w (*i.e.*, $pre(a) \subseteq w$). If the action is executable, then in the world state w' resulting from its execution all atoms in $add(a)$ are true, all atoms in $del(a)$ are false, and all remaining atoms keep the truth value they had in s (*i.e.*, $w' = (w - del(a)) \cup add(a)$). The initial state description specifies a single world state. Thus, in set form, I is a set of atoms and the initial state is one which assigns the value true to atoms in I and false to atoms not in I. The goal description is allowed to be partial, *i.e.*, to specify a set of states. It is also a set of atoms, but with the interpretation that the atoms in the set are required to be true in any goal state, while remaining atoms are "don't cares". Thus, if w is a set of atoms describing a world state, w is a goal state if $G \subseteq w$.

A sequence of actions, $a_1, \ldots, a_n$, is executable in a world state w if each a_i is executable in the state resulting from execution of the preceding sequence $a_1, \ldots, a_{i-1}$. A sequence of actions executable in the initial state of the problem and such that the result of executing the entire sequence is a goal state (a state containing G) is a plan. The cost of a plan is the sum of the costs of the actions in it.

In general, not all actions in a plan need to be a strictly ordered. A plan (or rather, a set of alternative plans) may be represented as a partially ordered set of actions (or action occurrences, since the same action can occur more than once in the plan). The partially ordered set is a plan if every linearization consistent with the ordering is a plan, according to the above definition.

Example: Blocksworld

The Blocksworld domain is a standard example in AI planning. It is a "toy example", in the sense that it does not model any conceivable real application, but it is a useful example nonetheless, as it illustrates many different aspects of the classical planning problem. (Note also that although it is a toy example, that does not mean it is *easy*: solving it optimally is NP-hard, see Gupta & Nau, 1992.)

The Blocksworld domain involves a number of distinctly labeled blocks, set on a table. Besides being on the table, blocks can be stacked on top of other blocks, but there is room for at most one block to be on top of each block. A block with no other block on it is said to be clear, and a block must be clear for it to be moved. The table is large enough for all the blocks to be placed side by side on it. The problem is to transform a given initial arrangement of the blocks into one satisfying a given goal description, using a minimal number of moves. An example problem is illustrated in figure 2.1.

The STRIPS model of the problem consists of atoms[1] `(on-table ?b)`, meaning

[1]We will use a PDDL-like syntax for examples throughout the thesis. Atom and action (and later also resource) names are written in a schematic form `(<SCHEMA-NAME> <ARG>*)`, where the arguments are constants (*i.e.*, the whole expression is the name of the atom or action). When

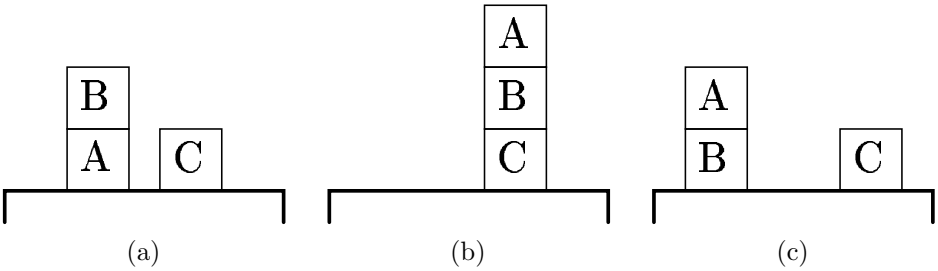


Figure 2.1: Example Blocksworld problem: (a) arrangement of blocks corresponding to initial state description {`(on-table A), (on B A) (clear B), (on-table C), (clear C)`}; (b) – (c) two arrangements satisfying the goal description {`(on A B)`}.

block `?b` is on the table, and `(clear ?b)`, meaning `?b` is clear, for each block `?b`, and atoms `(on ?b ?c)`, meaning block `?b` is stacked on block `?c`, for each pair of blocks. There are three separate actions for moving a block from the table to another block, from being stacked on a block to the table, and from on top of one block to another, respectively. As an example, the action `(move ?b ?from ?to)` has preconditions `(on ?b ?from)`, `(clear ?b)` and `(clear ?to)`, adds atoms `(on ?b ?to)` and `(clear ?from)` and deletes atoms `(on ?b ?from)` and `(clear ?to)`. Since every action moves a single block, the optimization criterion is modelled by assigning every action unit cost.

Properties of the STRIPS Planning Model

The plan existence problem, *i.e.*, the problem of deciding if there exists a valid plan for an arbitrary problem instance, in the propositional STRIPS planning model is decidable. It is in fact PSPACE-complete, as shown by Bylander (1991). The same also holds for the optimization problem of finding a valid plan of minimal cost for an arbitrary problem instance.

If there exists a plan for a given problem instance, there exists a plan containing at most 2^n actions, where n is the number of atoms in the problem description. Since actions are deterministic, an initial world state and a sequence of actions executable in that state correspond to a unique sequence (of equal length) of world states resulting from the execution of the action sequence. With n atoms there are only 2^n distinct world states so any longer plan must visit some world state at least twice. Since the executability of an action, and therefore a sequence of actions, depends only on the present world state (*i.e.*, actions are Markovian) the plan remains valid if the cycle (the part of the plan between the first and second visit to the world state) is removed.

describing example domains, we sometimes use parameterized schemata: symbols beginning with a question mark ("`?`") are parameters.

Invariants

The actions in a planning problem typically impose a certain amount of structure on the world states that can be reached from the initial world state by execution of a sequence of actions. A property of world states that is preserved by all actions in the problem, and therefore holds in every reachable world state if it holds in the initial world state, is called an *invariant* of the problem.

Invariants of a particular kind will be important to us, namely sets of atoms with the property that at most one, or in some cases exactly one, of the atoms is true in every reachable world state. For example, in the Blocksworld domain, at most one block can be on top of another block, so for each block `?b` at most one of the atoms `(on ?c ?b)` (for each `?c` $\neq$ `?b`) should be true in any world state. The actions of the problem do indeed maintain this property, so it is an example of an "at-most-one" invariant. An example of an "exactly-one" invariant is the set of atoms `(on-table ?b)` and `(on ?b ?c)` (for each `?c` $\neq$ `?b`) which corresponds to the property that block `?b` is always in exactly one place (on the table or on one of the other blocks).

Invariants of this kind are very common in the problems that are frequently used as examples or benchmarks in AI planning. An important special case is at-most-one invariants consisting of only two atoms (this is often referred to as a *mutex*, short for "mutually exclusive"; see section 3.1 in the next chapter). If a set C of atoms is an exactly-one or at-most-one invariant, any (non-empty) subset of C is also an at-most-one invariant. Hence, an at-most-one invariant can also be characterised as a set of pairwise mutex atoms. The automatic extraction of invariants (and of other properties of the planning problem) from the problem description is called *domain analysis*, and there is a substantial amount of work on algorithms for such analysis (*e.g.* Gerevini & Schubert, 1998; Fox & Long, 1998; Scholz, 2000; see Haslum & Scholz, 2003 for more references).

The definition is easily generalized to "at-most-k" or "exactly-k" invariants, for arbitrary $k \geqslant 1$, but such invariants are rarely, if ever, found in common example problems (in fact, Fox & Long (2000) devised a suite of example planning problems specifically rich in "two out of three" invariants to demonstrate a domain analysis technique).

Planning via Regression Search

Regression, in AI parlance, means "reasoning backwards". In the context of planning, regressing a condition (on a world state) through an action yields a new condition which ensures that the original condition holds in the state resulting from execution of the action in any state satisfying the new condition (thus, the new condition must imply the executability condition of the action).

Reasoning by regression leads to a planning method in which the search for a plan is made in the space of "plan tails": partial plans that end in a goal state, provided certain conditions on the state in which they start are met. The condition is obtained

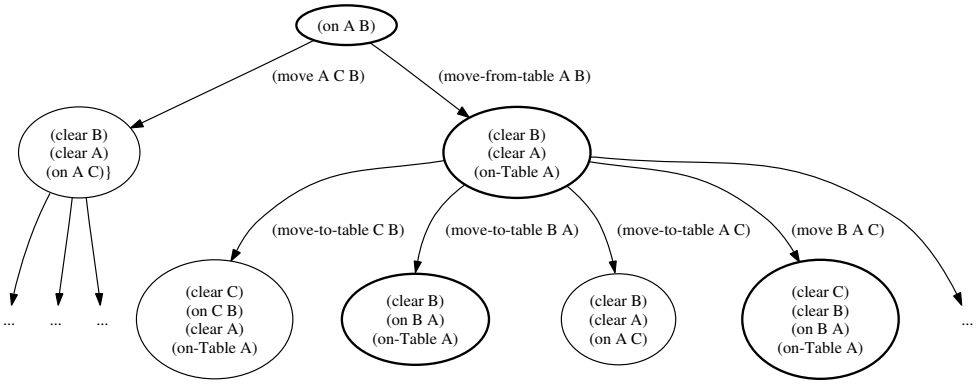


Figure 2.2: Part of the regression search tree for the Blocksworld problem in figure 2.1. The leaf nodes drawn in bold are final states (sets of subgoals that are true in the initial world state).

by regressing the goal condition G through the action sequence corresponding to the tail end of the plan under construction, in reverse order. Search starts with an empty plan, whose regressed condition equals G, the goals of the planning problem. Successors to a plan tail are constructed by prepending to it a new action, such that the action does not make the regressed goal of the partial plan unachievable. Search ends when a partial plan whose regressed condition is satisfied in the initial state is found.

In fact, it is not necessary to take the partial plan itself to be the search state: the regressed goal condition provides a sufficient summary of what needs to be known about it. Although different partial plans may result in the same regressed condition, they are in this case equivalent, in the sense that their successors will also have identical regressed goals, and either all or none of them are solutions. (Two partial plans with identical regressed goal conditions may have different costs, but cost is a property of the search space *path*, not the search state.) Thus, a sequential regression search state is a set, s, of atoms, representing subgoals to be achieved. An action a is applicable to a search state s (*i.e.*, can be used to regress s) iff $del(a) \cap s = \emptyset$, *i.e.*, if the action does not destroy any current subgoal. If action a is applicable, the result of regressing s through a is $s' = (s - add(a)) \cup pre(a)$, *i.e.*, s' contains the preconditions of a and any subgoals in s that were not made true by a. Search starts from the set of goals G and ends when a state $s \subseteq I$ is reached.

An example of (part of) the search tree for the Blocksworld problem in figure 2.1 is shown in figure 2.2. There are two final states, *i.e.*, sets of subgoals that hold in the initial world state: {`(clear B), (on B A), (on-table A)`} and {`(clear C), (clear B), (on B A), (on-table A)`} (these are the leaf nodes drawn in bold in the figure). The corresponding plans can be found by tracing the paths from these states back to the initial search state (the root of the tree) and reading off the

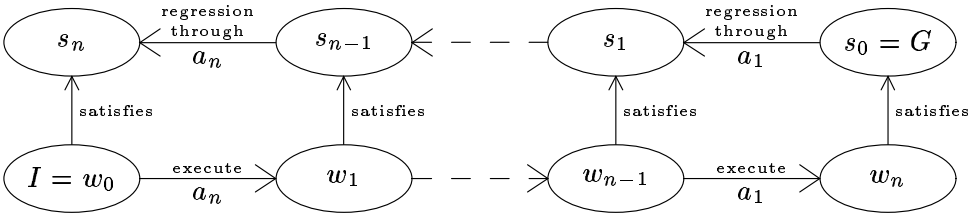


Figure 2.3: Relation between regression search states on a solution path and the world states resulting from the execution of the corresponding (reversed) action sequence.

associated actions: they are `(move-to-table B A)`, `(move-from-table A B)` and `(move B A C)`, `(move-from-table A B)`, respectively. The world states resulting from the execution of these plans are the ones depicted in figures 2.1(b) and 2.1(c).

Regression and Invariants

A solution path in the sequential regression space is a sequence of sets of atoms $s_0, s_1, \ldots, s_n$, such that $s_0 = G$, *i.e.*, the goals of the planning problem, s_i is obtained by regressing s_{i-1} through some action a_i, and $s_n \subseteq I$, *i.e.*, the final state on the path is a set of goals satisfied by the initial world state of the planning problem. The sequence of actions used to regress states along the path, reversed, is executable from the initial world state and its execution results in a world state in which all atoms in s_0 are true, *i.e.*, a state satisfying the goal of the planning problem. In fact, the sequence of world states generated by the execution of the reversed action sequence is such that each world state satisfies the set of goals in the corresponding regression state (this property, which implies correctness of the regression method, can be shown formally by a straightforward induction on the length of the solution path). The correspondence is illustrated in figure 2.3. Note also that any suffix $s_i, \ldots, s_n$ of the solution path is a solution path for s_i: thus, solving a regression state s is equivalent to solving a planning problem with s as the problem goal.

Recall from the last section that an invariant of a planning problem is a property on world states that is preserved by actions and that therefore holds in every reachable world state if it holds in the initial world state. The correspondence between regression states on a solution path and world states reachable by an executable action sequence implies that the goal sets on a solution path must be consistent with all invariants of the planning problem that are true of the initial world state. Hence regression states that are inconsistent with some invariant can never lead to a solution and can therefore be pruned from the search tree. A regression state s is inconsistent with an at-most-one or exactly-one invariant C iff $|s \cap C| > 1$, since in a world state satisfying s all atoms in s must be true, while in any reachable world state at most one atom in C can be true. (Note that s is not inconsistent with C if $|s \cap C| = 0$, even if C is an exactly-one invariant, since to satisfy s in a world state it is only

required that the atoms belonging to s are true, and atoms not belonging to s can be either true or false.)

In many planning problems, pruning search states inconsistent with invariants is important for the efficiency of the search. However, a good heuristic should also detect the insolubility of such states and in this case invariants need not be checked explicitly. This is demonstrated in the next chapter (section 3.2, page 46).

Commutativity

As noted above, it is often the case that a group of actions in a plan do not need to be strictly ordered for the plan to work. This is sometimes due to actions being *commutative*: two actions, a and a' are commutative iff the sequences a, a' and a', a are executable in exactly the same world states and also lead to the same resulting state when executed in the same state. A set of actions is commutative iff every pair in the set is. Under the assumptions of the sequential STRIPS planning model, the condition for commutativity is that $pre(a) \cap add(a') = \emptyset$, $pre(a) \cap del(a') = \emptyset$, $add(a) \cap del(a') = \emptyset$, and vice versa, *i.e.*, that neither action adds or deletes an atom that is a precondition of the other, or deletes an atom added by the other.

In the same way as execution of two different sequences comprised of the same set of commutative actions in the same world state leads to the same resulting world state, regression of the same goal condition backwards through the two sequences result in identical regressed conditions. Thus, the presence of commutative actions introduces *transpositions*, alternate paths to the same state, in the regression planning search space. (An example of a transposition, although one not caused by commutative actions, can be seen in figure 2.2: the state {`(clear B)`,`(clear A)`,`(on A C)`} is reached both by regression through action `(move A C B)` and by regression through first `(move-from-table A B)`, then `(move-to-table A C)`.) Transpositions are a source of inefficiency for linear-space search algorithms, such as IDA*, which do not detect them and therefore may explore the same part of the search space several times. General techniques for detecting (and thus avoiding) transpositions are based on the use of additional memory (*e.g.* transposition tables; see Reinfeld & Marsland, 1994). Transpositions in the sequential regression planning search space that are due to commutative subsequences of actions can be avoided by applying the following rule: When expanding a state s, which was reached by regression through an action a, an action a' that is commutative with a can only be applied if, in addition to the condition that $del(a') \cap s = \emptyset$, it follows a in a fixed arbitrary total ordering of the set of actions (since the set of actions in a planning problem is finite, such an ordering can always be found). This ensures that of the possible permutations of a subsequence of commutative actions in a developing plan, only one is considered.

Enforcing such "commutativity cuts" is not unconditionally beneficial: the rule makes the set of successors to a state dependent on the last action in the path by which the state was reached, effectively making this action a part of the search state. Thus, the rule eliminates *paths* from the search space, but at the cost of increasing the number of distinct *states*. Hence, it is useful for linear-space search algorithms, such as IDA*

or DFS Branch-and-Bound, that suffer from transpositions, but actually harmful for full-memory algorithms like A*.

2.2 Planning with Time

In most planning problems, the actual execution times of actions are not all equal – in fact, they can vary a great deal. Also in many problems, it is desirable to synthesize plans whose total execution time (known as the *makespan* of the plan) is minimal. This objective can not, in general, be formulated as a sum of independent action costs, and therefore requires an extended (temporal) model of planning problems. Obviously, action durations must be specified. Also, when minimizing makespan it is advantageous to plan actions that do not need to follow one another in time concurrently, so the planning model must describe when this is possible. Actions that are commutative in the sense defined above, and thus can be executed in arbitrary order, may nevertheless interfere *during* their execution, so that they can not be executed in parallel.

The temporal STRIPS planning model, introduced in this section, extends the STRIPS model for classical planning only enough to enable makespan minimization. Except for the addition of temporary delete effects, it is the model introduced with the Temporal Graphplan (TGP) planning system by Smith & Weld (1999)

The Temporal STRIPS Planning Model

A (propositional) temporal STRIPS planning problem contains the same elements as its non-temporal counterpart (a set of atoms, a set of action descriptions, and initial state and goal descriptions). Each action a has preconditions ($pre(a)$), positive ($add(a)$) and negative ($del(a)$) effects, but, in addition to these, also a set $lock(a)$ of atoms that are "temporarily deleted", or "locked" by the action. Atoms in $lock(a)$ are destroyed by the action during its execution, but restored to their original truth values before the action finishes. Locked atoms are a tool to model non-concurrency restrictions in the planning problem. Each action also has a duration ($dur(a)$) greater than zero. We assume that time is modelled by the extended rational numbers (*i.e.*, the rational numbers together with $+\infty$ and $-\infty$), though extended integers or reals could be used as well.

A schedule is a collection of action instances with specified start times, $S = \{(t_1, a_1), \ldots, (t_n, a_n)\}$. The makespan of the schedule is the distance from the start time of the earliest action to the end time of the latest. For simplicity, start times of actions in the schedule are considered relative to the start time of the schedule, *i.e.*, the start time of the earliest action is assumed to be 0 (this means the makespan is $\max_{(t,a)\in S} t + dur(a)$). A schedule is a plan iff every action in the schedule is executable at its start time and the world state at the end time of the schedule satisfies the problem goal description. For an action a to be executable over a time interval $[t, t + dur(a)]$, atoms in $pre(a)$ must be true in the world state at t, and preconditions and positive

effects of the action (atoms in $pre(a) \cup add(a)$) must not be destroyed (permanently or temporarily) by any other action throughout the interval. This implies that persistent preconditions (atoms in $per(a) = pre(a) - (del(a) \cup lock(a))$) remain true over the entire interval. Effects of the action take place at some (unspecified) point in the interior of the interval, so (non-temporary) effects can be relied on to hold at the end point.

Two actions, a and a', are said to be *compatible* iff they can be safely executed in overlapping time intervals. The assumptions listed above lead to the following condition for compatibility: a and a' are compatible iff $(pre(a) \cup add(a)) \cap (del(a') \cup lock(a')) = \emptyset$ and vice versa, *i.e.*, neither actions deletes (temporarily or permanently) an atom required or added by the other action. Note that compatibility does not imply commutativity (which requires also that neither action adds a precondition of the other), nor does commutativity imply compatibility (since the former does not take temporary deletes into account).

Example: Planning Multi-UAV Missions

The Multi-UAV Mission Planning domain models a fleet of autonomous unmanned air vehicles (UAVs) whose joint task is to carry out a set of observations (of stationary objects), as quickly as they can. The domain is inspired by the WITAS UAV project (Doherty *et al.*, 2004; the domain was originally designed by Per Nyblom). The planning problem abstracts away many aspects of the real world. For example, making an observation may involve complex image processing and reasoning, but from a planning point of view it is simply a matter of moving a UAV to the correct position for that observation and staying there for a short time. All UAVs are assumed to carry the same sensor equipment, so any one of them can be used. The intended UAVs are helicopters, but this fact is significant for the planning problem only in that it implies that the UAVs are able to hover, *i.e.*, to remain airborne at a fixed position. The main complication is to ensure the UAVs are at all times kept safely separated, in the air as well as on the ground.

To fit the problem into the temporal STRIPS model a finite set of "interesting" positions is assumed given, together with collision-free paths between those positions. The position of each UAV is represented by atoms `(at ?uav ?p)`, for `?p` a position, and atoms `(airborne ?uav)` and `(on-ground ?uav)` to indicate if the UAV is hovering in the air or landed on the ground at the position. Atoms `(free ?p)`, for each position or path `?p` are used to indicate that no UAV is currently on or dangerously near the position or path. The action for flying a UAV along a path between two positions, `(fly ?uav ?from ?to ?path)`, has preconditions `(at ?uav ?from)` and `(free ?p)` for every position and path that lies or passes within the safety distance of the path `?path`, except for those that are near the starting point `?from` since these will be occupied by the UAV itself and thus not free. The action deletes `(at ?uav ?from)` and adds `(at ?uav ?to)`, and also deletes `(free ?p)` for any position or path in the vicinity of the end point `?to` and adds `(free ?p)` for corresponding positions or paths in the vicinity of the starting point. In addition, it locks (temporarily

deletes) `(free ?p)` for any position or path `?p` that lies too close to the path flown but not near the start or end positions. The duration of the action is calculated from the path (this, and the generation of collision-free paths, is done by a specialized path planner).

A small part of the map from an example problem is shown in figure 2.4. As can be seen in the figure, the path between positions `p0` and `p42` crosses several other paths, for example the path between `p20` and `p23`, so the action of flying a UAV from `p0` to `p42` locks (temporarily deletes) the corresponding atom, `(free p20 p23)`. Because the path passes within the safety distance of point `p43`, the action also locks atom `(free p43)`, as well as the `free` atoms corresponding to all paths starting/ending in this point. It deletes (not locks) `free` atoms of all paths starting/ending in point `p42`, since this position will be occupied by the UAV flying there from `p0` at the end of the action, but adds `free` atoms of paths starting/ending in point `p0` since this position was occupied by this UAV before the action (and therefore not by any other UAV) but will not be after it is done.

Some important aspects of the mission planning problem are ignored in this model. For example, UAVs can only stay in the air for a limited time (due to a limited supply of fuel, among other things). This restriction is not captured by the planning domain above, nor can it be using the temporal STRIPS model (except by imposing a limit on the overall makespan of the plan). It is one example of why models of planning problems often need to include *resources*, which will be dealt with in chapter 5.

Properties of the Temporal STRIPS Planning Model

The temporal STRIPS planning model has the property that all valid plans, indeed all executable schedules, remain valid when sequenced, *i.e.*, when all actions in the schedule are placed in any sequence consistent with their ordering in the schedule. This is because the model never requires actions to execute concurrently; concurrent execution is only a means for reducing the makespan of plans. As a consequence, the complexity of the plan existence problem for the propositional temporal STRIPS planning model is the same as in the sequential case, *i.e.*, PSPACE-complete (Bylander 1991).

Temporal Regression Planning

Temporal regression, like regression in the sequential setting, is a search in the space of plan tails, partial plans that achieve the goals provided that the preconditions of the partial plan are met. In sequential planning, the atoms representing the collected precondition of the plan tail provide a sufficient summary of the partial plan, so search states are sets of (subgoal) atoms. In the temporal setting however, actions may execute concurrently: in particular, at the starting time of an action other actions may be on-going. Because of this, a set of precondition atoms is no longer sufficient to summarize a plan tail. States have to include actions concurrent

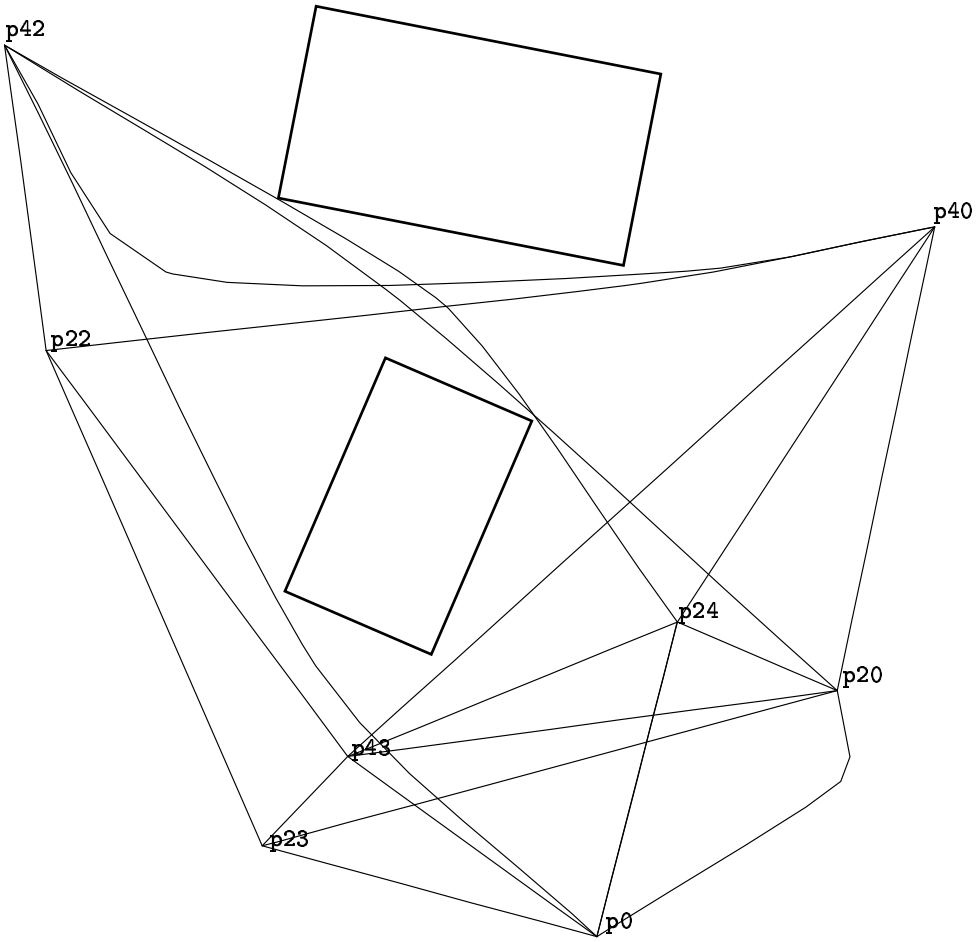


Figure 2.4: Part of the map for a UAV mission planning problem. The rectangles represent bounding boxes around obstacles (buildings) rather than the actual geometry of the obstacles. These include a safety margin, so paths that touch (but do not enter) a bounding box are collision-free. Normally, paths may cross obstacle bounding boxes if at a high enough altitude, but in this example the path planner has been explicitly instructed not to do so. Point `p0` is a UAV take-off/landing position, the rest are observation positions.

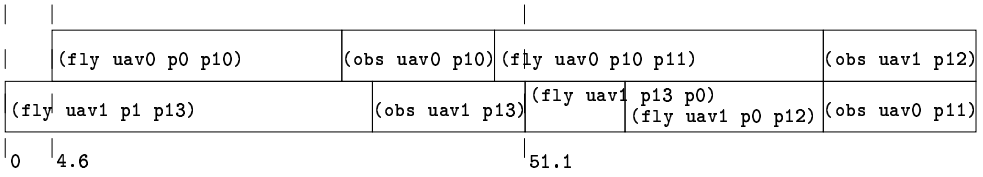


Figure 2.5: Example of a schedule for a UAV mission planning problem with two UAVs. (The corresponding planning problem is not the same as that shown in figure 2.4.)

with the subgoals, and the timing of those actions relative to the subgoals. Consider the example plan in figure 2.5, and the world state at time 51.1 (marked in the schedule by a vertical line). Since this is the starting point of action `(fly uav1 p13 p0)`, the preconditions of this action must be subgoals to be achieved at this point. But the action that achieves those conditions must be compatible with the action `(fly uav0 p10 p11)`, which starts 3 units of time earlier and whose execution spans across this point.

A temporal regression search state is a pair $s = (E, F)$, where E is a set of atoms and $F = \{(a_1, \delta_1), \ldots, (a_n, \delta_n)\}$ is a set of actions a_i with time increments δ_i. This represents a partial plan (tail) where the atoms in E must hold and each action (a_i, δ_i) in F has been started δ_i time units earlier. Put another way, an executable schedule *achieves* state $s = (E, F)$ at time t iff the plan makes all atoms in E true at t and schedules action a_i at time $t - \delta_i$ for each $(a_i, \delta_i) \in F$.

When expanding a state $s = (E, F)$, successor states $s' = (E', F')$ are constructed by chosing (non-deterministically) for each atom $p \in E$ an establishing action (a regular action a with $p \in add(a)$, or a special "no-op" action with p as both precondition and effect), such that the chosen actions are compatible with each other and with all actions in F, and advancing time to the next point where an action starts (since this is a regression search, "advancing" and "next" are in the direction of the beginning of the developing plan). The collected preconditions of actions starting at this point become E' while remaining actions (with their time increments adjusted) become F'. More formally, let $s = (E, F)$ be the state to be expanded. Let A be the set of (non-deterministically) chosen establishers, a non-empty set of actions such that for each $a \in A$, there exists some atom $p \in E$ such that $p \in add(a)$ and for no atom $p \in E$ does $p \in del(a)$, and such that all actions in $A \cup F$ are pairwise compatible. Let $N = \{p \in E \mid \neg\exists a \in A : p \in add(a)\}$, *i.e.*, the set N contains those atoms in E that are not added by any of the actions chosen, and therefore must be supported by no-ops. The successor state $s' = (E', F')$ is now constructed by moving to the "next interesting point" in time: this is the starting time of the action (or actions, if more than one starts at the same time) in $F \cup \{(a, dur(a)) \mid a \in A\}$ that is closest, *i.e.*, has the smallest δ value. Formally, the distance is

$$\delta_{\text{adv}} = \min\{\delta \mid (a, \delta) \in F \cup \{(a, dur(a)) \mid a \in A\}\} \tag{1}$$

Note that the no-ops supporting atoms in N are not included in the calculation: an action with no effects other than the persistence of an atom can of course last for any amount of time, and the selected no-ops are "stretched" just to the next interesting time point. The two components of the successor state become

$$\begin{aligned} E' &= \{pre(a) \,|\, (a, \delta_{adv}) \in F \cup \{(a, dur(a)) \,|\, a \in A\}\} \cup N \\ F' &= \{(a, \delta - \delta_{adv}) \,|\, (a, \delta) \in F \cup \{(a, dur(a)) \,|\, a \in A\} \wedge \delta > \delta_{adv}\} \end{aligned}$$

Any actions starting at the new time point (whether they were in the parent state s or added as new establishers) are removed from the successor state, and their preconditions, together with any atoms that were supported by no-ops, become subgoals. Remaining actions have their δ values decreased by the same amount, δ_{adv}. The part of the plan tail that lies between the old and the new "current time point" is effectively "forgotten": it no longer matters what lies there, at least for the purpose of the continued construction of the earlier part of the plan. (Once a final state has been reached, the "forgotten" actions can be recovered from the solution path in the search space by comparing the F component of each state along the path with that of its predecessor.)

An example of successor construction is depicted in figure 2.6: 2.6(a) depicts the state $s = (\{p_1, p_2, p_3\}, \{(a_1, 1), (a_2, 2)\})$, *i.e.*, a state with subgoals p_1, p_2 and p_3, and actions a_1 and a_2 scheduled at $\delta = 1$ and $\delta = 2$ time units earlier, respectively. Suppose action a_3, with duration 3, is selected to support atom p_1, action a_4, with duration 1, is selected to support atom p_2, and no action is selected for atom p_3, which is supported by a no-op. 2.6(b) shows the result of adding the selected actions to the "front" of the plan tail. The smallest δ among all scheduled actions in the resulting state is 1, so this is the value of δ_{adv}. The construction of the successor is finalized by advancing the "current time point" by this amount, as shown in 2.6(c). Actions a_1 and a_4, which start at this point, are forgotten and their preconditions together with atom p_3, which was supported by a no-op, become the subgoals of the successor state. The complete successor state is $s' = (\{p_3\} \cup pre(a_1) \cup pre(a_4), \{(a_2, 1), (a_3, 2)\})$.

The initial search state is $s_0 = (G, \emptyset)$, *i.e.*, the state containing the goals of the planning problem and no scheduled actions. A state $s = (E, F)$ is final if $F = \emptyset$ and $E \subseteq I$. The time increment δ_{adv} defined by equation (1) above is the amount that the makespan of the plan tail increases between state s and its successor s', and therefore it can be considered the "cost" of the transition from s to s'. When a final state is reached, the sum of the transition costs along the solution path (the path from the root to the final state) equals the cost (makespan) of the corresponding plan.

The search space defined above is sound, in the sense that any solution path found corresponds to a valid plan, but not *all* valid plans have a corresponding solution path. In a temporal plan there is often some "slack", *i.e.*, some actions can be shifted forward or backward in time without changing the plans structure or makespan, and because time is represented by the dense rational numbers, the set of such small variations of plans is infinite. The plans found in the temporal regression space are such that a regular action (not a no-op) is executing at all times and no-ops start

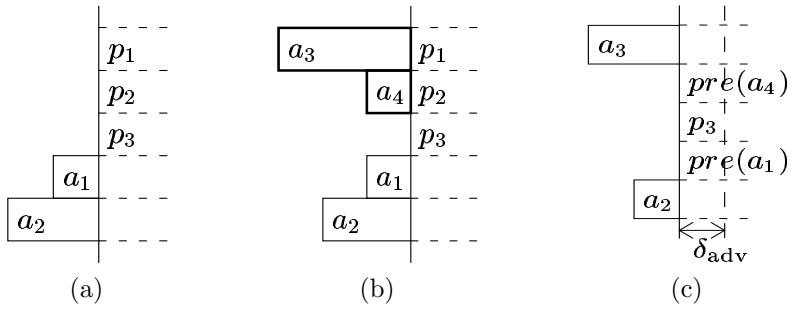


Figure 2.6: Illustration of the temporal regression process: (a) the state to be expanded; (b) the selected establishers added; (c) the resulting successor state, after the "current time point" has been moved.

only at the times where some regular action starts. For the present planning model, however, this is enough to ensure that *some* optimal plan will be found (assuming any plan exists at all).

Incremental Successor Construction

Heuristic search algorithms that find optimal solutions use an admissible heuristic to estimate the cost of states encountered during the search. For any non-final state, the admissible heuristic yields a lower bound on the cost of any solution reachable through that state. Different search algorithms use this information in different ways: in algorithms that perform cost-bounded search, such as IDA* or DFS branch-and-bound, it is of vital importance for the efficiency of the search that if the estimated cost of a successor state found during state expansion exceeds the current cost bound, this is detected as early as possible.

Therefore, in practice, temporal regression is done incrementally. The set of establishers is not selected simultaneously, but rather for one subgoal at a time, and the partially constructed successor state is evaluated after each selection to determine if its estimated cost is still within the bound. A recursive version of this incremental regression procedure is sketched in figure 2.7. The procedure is called initially with the E and F components of the state being expanded and empty sets of selected actions and no-ops. The procedure for evaluating (partial) states takes as arguments the set of actions and no-ops that will go into the final successor construction (including both those inherited from the expanded state and those newly selected), as well as the set of subgoals that remain to be dealt with, and returns an estimate of the cost of the successor state (actually an estimate of the estimated cost of the successor state plus the transition cost from the current state). It is described in more detail in the next chapter.

Another unspecified step of the procedure is how to pick the next subgoal atom to

```
TemporalRegress(E, F, selected, noops)
{
  if (E is empty) {
    finalize construction of successor state s';
    ContinueSearch(s');
  }
  g = pick next atom in E;
  // if g is a precondition of an already selected action, it
  // is not necessary to consider any establisher for g
  if (g in pre(a) for some a in selected) {
    TemporalRegress(E - {g}, F, selected, noops + {g});
  }
  // try supporting g with a no-op
  est. cost = eval(F + selected, noops + {g}, E - {g});
  if (est. cost <= bound) {
    TemporalRegress(E - {g}, F, selected, noops + {g});
  }
  // try each possible supporting action for g in turn
  for (each action a such that g in add(a))
    if (a compatible with F and selected) {
      est. cost = eval(F + selected + {a}, noops, E - {g});
      if (est. cost <= bound)
        TemporalRegress(E - {g}, F, selected + {a}, noops);
    }
}
```

Figure 2.7: Recursive algorithm for incremental temporal regression.

regress (`g` in the procedure in figure 2.7). The order in which this is done makes no difference for the completeness of the procedure (since all atoms must be regressed eventually), but can affect efficiency. Since the aim is to discover cost bound violations as early as possible, a sensible strategy is to choose first the atom expected to be the most "difficult" (costly) and the same heuristic function that is used by the state evaluation can be used also to estimate this.

Complementary to the early detection of cost bound violations is early detection of final states. Recall that a state is considered final only if it consists of a set of subgoal atoms that are true in the initial world state and the set of scheduled actions is empty. In practice, this condition can be relaxed to include also states containing some scheduled actions, if the preconditions of those actions all hold in the initial world state. An example can be seen in the plan in figure 2.5, at the time marked 4.6: the state at this point consists of $E = pre(\texttt{(fly uav0 p0 p10)})$ and $F = \{(\texttt{(fly uav1 p1 p13)}, 4.6)\}$, and both sets of preconditions are initially true. In such a state, there is no reason to consider any way of establishing current subgoals (or preconditions of actions with a δ value smaller than the greatest) other than by persistence from the initial world state, since this will yield a valid plan and there is no way to obtain one with a smaller makespan as an expansion of this state (the current time point must be advanced by an amount at least equal to the greatest δ among the already scheduled actions to include the starting time for all of these).

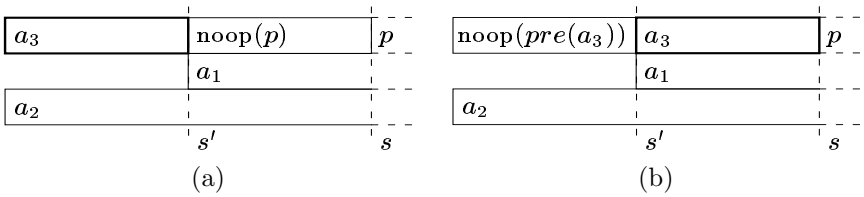


Figure 2.8: Illustration of the right-shift cut rule: Suppose action a_3 adds p, and is compatible with actions a_1 and a_2. In figure (a) action a_3 is positioned as early as possible (left-shifted) and p is supported by a no-op from the end of a_3 to the point where it is required to hold. In figure (b) action a_3 is positioned as late as possible (right-shifted) and its preconditions are supported by no-ops until its starting point. The right-shift cut rule eliminates the first possibility, by excluding a_3 as a possible establisher for p in state s' since p was supported by a no-op in the predecessor state s and a_3 is compatible with all actions in the F component of s.

Right-Shift Cuts

Even though no-ops in plans corresponding to paths in the temporal regression space are restricted to start only at times where some regular action starts, it may still be possible to shift some actions forward or backward in time. Again, an example can be found in the schedule in figure 2.5, where *e.g.* the action `(fly uav0 p0 p10)`, which starts at time 4.6 and is preceded by a no-op supporting the precondition of the action, could start at time 0 instead and followed by a no-op supporting the atom `(at uav0 p10)` which is added by this action and needed by the action `(observe uav0 p10)` that follows.

A right-shifted plan is one in which all such movable actions are scheduled as late as possible. Non-right-shifted plans can be excluded from consideration without endangering optimality. Like the commutativity cuts in sequential planning, this eliminates redundant paths in the search space. Right-shifting is ensured by applying the following rule: When expanding a state $s' = (E', F')$ whose predecessor is $s = (E, F)$, an action a that is compatible with all actions in F may *not* be used to establish an atom in s' when all the atoms in E' that a adds have been obtained from s by no-ops. The reason is that a could have been used to support the same atoms in E, and thus could have been shifted to the right (delayed). The application of the rule is illustrated in figure 2.8. Note that the right-shift cut rule, like commutativity cuts in sequential planning, introduces a dependency on the predecessor state, and therefore also effectively increases the number of distinct states.

From an execution point of view, it may be preferable to place actions whose execution time in the plan is not precisely constrained as early as possible (left-shifted) rather than at the latest possible time. From a search point of view what matters is that of the many possible, but structurally equivalent, positions in time for an action, only one is considered. The reason why right-shifting is used instead of left-shifting

is that in a regression search, a left-shift rule will "trigger" later (deeper in the search tree) and thus be less efficient.

Parallel Planning

An important special case of temporal planning is so called *parallel* planning, in which all actions have unit duration. This may be because action durations really are all equal (or near equal enough that the differences are of no practical concern). Also, for some domains, heuristic estimates of "parallel length" are more accurate than heuristics for sequential plan length (summed unit cost), so when the cost or makespan of the plan does not matter (the interest is simply in finding *a* valid plan) it is sometimes more efficient to search for parallel plans. When parallel planning was introduced, with the Graphplan planning system (Blum & Furst, 1997; see also next chapter), this was an important part of the motivation.

If the restriction to unit durations is made, temporal regression search may be simplified somewhat (generally resulting in improved performance). The most important consequence of the restriction to parallel planning is that the set of scheduled actions in a search state will always be empty. This is the case for the initial search state, which contains only the set of goals of the planning problems. When this state is expanded, the makespan increment between the state and its successors, δ_{adv}, which is defined as the minimum δ value of all scheduled actions in the state, will in this case equal the (unit) duration of any action just added, and since the durations of all the added actions are the same, they will all be "forgotten" so that the successor state again contains only atoms (the preconditions of the chosen actions, plus atoms supported by no-ops). This property of regression states in parallel planning simplifies the implementation of the search and the heuristic evaluation of such states, compared to the temporal case.

2.3 Discussion: Planning Models and Methods

Having introduced the underlying models of classical and temporal planning and the corresponding regression methods for translating planning problems into search problems, on which the remainder of this work is based, we discuss in this section some of the rationales for adopting these models and methods, and some alternatives. The discussion is not intended to be an overview (or history) of research in AI planning in general, but since the development of models for describing planning problems is closely related to the development of systems for solving such problems, a certain amount of systems name-dropping is unavoidable. Hendler, Tate & Drummond (1990) provide a good account of AI planning up to that time; Weld (1999) covers more recent developments. The textbook by Ghallab, Nau & Traverso (2004) gives a more thorough introduction to AI planning, and an up-to-date and widely scoped summary of variations on the planning problem, techniques and applications.

It is fair to say that the propositional STRIPS model is the simplest possible formal

model of classical planning problems. Likewise, the temporal STRIPS model is the simplest possible in the sense that it is the smallest extension of the classical STRIPS model that is necessary to allow for planning with makespan optimization (although the temporary delete effects constitute a slight complication which could be done away with for an even simpler model). This simplicity is precisely the reason why these models have been chosen as the basis for this work: our primary topic is admissible heuristics for planning and principles by which such heuristics can be derived, and both the discovery and presentation of those principles benefit from being made in a setting as simple as possible. But simplicity does not come without a price: the expressive power of the chosen models is less than that of some of the alternatives, in particular in the case of temporal planning. Applying the lessons learned in this most basic setting to more expressive models is a logical direction for the continuation of this research. As noted above, however, the plan existence problem in the propositional STRIPS representation is PSPACE-complete (Bylander 1991). Thus, the simplicity of the representation does not imply computational simplicity.

Models for Classical Planning

Underlying the STRIPS model of classical planning is a view of plan execution that can be coarsely summed up as "*actions*, when executed under specific *conditions*, cause change to the world *state*." This leads to the view of plans as paths in a state space, and of planning as a state transformation problem. The model rests on a a set of assumptions (collectively known as the STRIPS assumptions), *viz.* that the planners description of the world is correct and complete, and that the world is deterministic both *w.r.t.* to the actions available to the planner and in the sense that nothing besides actions in the plan change relevant parts of the world while the plan is executed. The model can be traced to some of the earliest AI planning systems (the QA3 system by Green (1969) and the STRIPS system by Fikes and Nilsson (1971) which lent the model its name; although the QA3 system employed general deductive reasoning, it was with a theory of actions similar to the STRIPS model) – hence the epithet "classical". However, the expression classical planning is normally understood to encompass more than the STRIPS model. The exact boundaries of classical planning are not easily determined, though typically any planning model that accepts the STRIPS assumptions is said to be classical. Hierarchical task decomposition models are an important borderline case, and will be discussed separately at the end of this section.

The original STRIPS representation is a finite first order, or "lifted", representation. This means predicates and parameterized action schemata are used in place of propositions and actions, but there is a finite set of constants with which predicates and action schemata can be instantiated (the restriction to a finite set of constants is essential: if the predicates and action schemata can be instantiated with arbitrarily nested function symbols, the corresponding plan existence problem is undecidable; see Chapman, 1987 or Ghallab, Nau, & Traverso, 2004, chapter 3). A problem in

lifted representation can be transformed into an equivalent propositional problem, equivalent in the sense that a valid plan exists for the transformed problem if and only if one exists for the problem as originally stated (this plan also has the same optimal cost), a process commonly called *grounding*. The resulting propositional problem, however, may become exponentially larger. This increase in size is unavoidable, since the plan existence problem for lifted STRIPS is EXPSPACE-hard (Erol, Nau, & Subrahmanian, 1991; Ghallab, Nau, & Traverso, 2004, chapter 3). In spite of this, grounding is done internally by many modern planning systems (see *e.g.* Hoffmann & Koehler, 2000). Planners that work with the lifted representation internally (*e.g.* UCPOP and its many descendants (Penberthy & Weld 1992); or IxTeT, Ghallab & Laruelle, 1994; Trinquart, 2003), can benefit from having a smaller number of options to consider (and thus a smaller branching factor for the search) by using partially instantiated action schemata and from being able to recognize symmetric options (a very important capability for some planning problems; see Fox & Long, 1999) but must deal with the additional complexity of managing constraints on the possible values of uninstantiated parameters. The potential benefit of using a lifted representation may also diminish with the introduction of cost-estimating heuristics, if the cost of an action is permitted to depend on the instantiation of the parameters of the corresponding action schema. Younes & Simmons (2002) compare the the use of lifted and grounded internal representations in the context of partial order planning and conclude that the advantages of grounding can for the most part be recaptured in a lifted representation through extended use of constraints and some modifications to the search strategy. In their empirical comparison, the improved lifted planner often demonstrated better efficiency, but with significant variation across different planning problems.

The STRIPS representation has been extended in many ways, while staying within the confines of the classical planning model. For example, the ADL representation (Pednault 1988) allows action schemata to have complex preconditions (essentially first-order formulas, including quantified formulas), to specify sets of effects by quantifying variables appearing in the effect description, and to specify for each individual effect of an action additional conditions that must hold for that effect to take place when the action is executed (so called conditional effects). These extensions can all be "compiled away", *i.e.*, a problem description using them can be transformed into an equivalent problem in basic STRIPS representation, although again at the cost of a potentially exponential increase in size (Gazen & Knoblock 1997) or changes to the structure of solution plans such that the compilation does not preserve optimal solution cost (Nebel 2000). The STRIPS representation and some extensions have, relatively recently, been unified in the *Planning Domain Definition Language* (PDDL; see McDermott *et al.*, 1998, and Bacchus, 2000). The primary purpose of PDDL has been to enable a series of planning system competitions, but due to successive additions of new features to the representation it has also been one of the forces pushing automated planning to deal with new challenging problems.

A feature found in the problem representations of many early planning systems (including the aforementioned QA3 and STRIPS), which were strongly influenced by

the use of logic to represent knowledge, is a so called background theory: a set of logical axioms about the problem domain that may, for example, define predicates in terms of other, more basic, relations, express invariants (of the kind discussed below, or others), or rules for determining indirect effects of domain actions. However, since entailment in first-order logic is only semi-decidable in general, the use of an unrestricted background theory makes the planning problem (at best) semi-decidable as well. As the planning problem later received a more formal treatment (*e.g.* by Chapman (1987), Bylander (1991) and others) the problem model was simplified and the background theories discarded. Practical planning systems maintained the use of background theories, but with strong restrictions on the inferences made (see *e.g.* Wilkins, 1983). Recently, background theories in a restricted form were reintroduced, in version 2.2 of PDDL (Edelkamp & Hoffmann 2004). PDDL 2.2 allows so called derived predicates, which are defined by a set of rules evaluated recursively (in PROLOG style), rather than part of the world state. This restricted form of theory allows, *e.g.*, the transitive closure of a relation to be expressed, but is still compilable into the basic representation (Thiebaux, Hoffmann, & Nebel 2003).

The state variable representation (Bäckström 1992; Sandewall & Rönnquist 1986) replaces the logical propositions of the STRIPS representation by variables with finite ranges of values, but embodies the same planning model: in fact, it is expressively equivalent with propositional STRIPS (for each variable v and value c, "$v = c$" is essentially a proposition; a formal proof of equivalence is given by Bäckström, 1992). However, describing the world state by non-binary variables makes some of the structure of a planning problem explicit, since a state variable expresses an exactly-one invariant (as described in section 2.1, page 8, above) over the set of values. In the Blocksworld domain, for example, each block is in each world state in exactly one place (on the table or on one of the other blocks) and has exactly one "thing" on top of it (either nothing, or one of the other blocks). These properties of the domain are reflected in the fact that atom sets {`(on-table ?b)`, `(on ?b ?c)` | ∀`?c` ≠ `?b`)} and {`(clear ?b)`, `(on ?c ?b)` | ∀`?c` ≠ `?b`)} are exactly-one invariants. In a state variable representation the position of block `?b` can be represented by a variable `pos(?b)` with values `on-table` and `on(?c)` (for each `?c` ≠ `?b`), making the invariant explicit. We make use of the state variable representation in the construction of pattern database heuristics, in chapter 4.

Models for Planning with Time

Planning problems including various temporal aspects have been addressed by several AI planning systems: actions with explicit duration, goals with deadlines and external events (events not under the control of the planner, but occurring at known times) were introduced by Vere (1983) and by Allen & Koomen (1983); the IxTeT planning system (Laborie & Ghallab 1995) combined durative actions with a model of resources, allowing finer control of concurrency and more; the planning model of the Zeno system (Penberthy & Weld 1994) described the world by continuous variables and actions causing continuous change to those variables; to name just a few.

Makespan minimization was made explicit in the TGP planning system (Smith & Weld 1999). However, it is closely related to planning with goal deadlines (which is essentially the decision version of the optimization problem). Due to the variation in the set of problem aspects addressed by each system and in the solution methods employed (and perhaps also in the philosophical standpoints of their creators, as is not uncommon in the field of AI) no common model for planning with time has been widely adopted, like the STRIPS model has for classical planning. Planning with time is also closely related to scheduling: it always includes an element of scheduling, in that actions in the plan are positioned in time, and any scheduling problem can be stated as a planning problem with very limited action choice (given a problem representation that is sufficiently expressive, in particular *w.r.t.* resources); on the other hand, complex real-world scheduling problems (*e.g.* in transportation, Becker & Smith, 2000 or manufacturing, Barták, 2004) usually involve a significant element of planning (in the sense of action selection) and probably most temporal planning problems can be stated as scheduling problems (given a problem representation with a sufficiently rich model of choice). For example, the multi-UAV planning problem could be (loosely) stated as a scheduling problem with sequence-dependent setup times: the observations are the tasks to be scheduled, the UAVs are resources required by those tasks, and positioning a UAV for an observation is the setup activity required to bring the resource into the correct state for the task (although this statement of the problem ignores the complication that the time required for repositioning a UAV can depend on the state, *i.e.*, position, of other UAVs, due to the safe separation constraint). A more developed example of translating a planning problem into a scheduling problem is presented by Bedrax-Weiss, Crawford & Smith (2004). So, in our review of models for planning with time we should also consider models of scheduling problems, which complicates the discussion since the two research fields have (until quite recently) been relatively separate, and have developed somewhat different basic concepts and terminology.

In coarse terms, two different kinds of models for planning with time can be distinguished: models descended from the classical planning view of planning as a state transformation problem typically describe actions by the (more or less) abstracted process of their execution, while so called "activity centered" models describe the requirements and constraints on the composition of actions (or activities), rather than the actions themselves. Like the classical view was summarized as "actions, when executed under specific conditions, cause change to the world state," the activity centered view can be summed up as "*activities*, subject to *resource* and other *constraints*, are executed to satisfy *requests*" (this is a very simplified rephrasing of the scheduling ontology presented by Smith & Becker, 1997). Activity centered models definitely dominate problem representations among scheduling systems and integrated planning/scheduling systems originating in a scheduling perspective (*e.g.* Muscettola, 1994), but can be seen also in hierarchical task decomposition models for classical planning and in models based on AI knowledge representation formalisms (Allen & Koomen 1983). It should be noted, however, that this distinction between "state transformation" and "activity centered" models is mostly conceptual, and

probably does not have much practical significance. Both views can be "instantiated" in a variety of different ways, resulting in models with different expressive power, and most known solution techniques can probably be applied to models of either kind.

In most applications of planning, the actual execution of an action in reality is a fairly complex process, involving concurrent processes that cause continuous change to the world state over time and events that cause those processes to begin, end, and follow in sequence. The descriptions of world and actions in the classical STRIPS model are *abstractions* in which all aspects of the execution process except its necessary conditions and lasting effects are ignored. Temporal planning models based on the classical view simply retain a little more of the original complexity, *i.e.*, describe the execution of actions in a slightly less abstract way. Depending on how, and to what level, the abstraction is made, it is possible to arrive at many different models, with different expressive power and different problem complexity. In most models the continuous variables of the world state are abstracted to logical properties and the continuous changes caused by actions are abstracted to events, corresponding to changes in the abstract world state (examples include the problem representations of planning systems such as Deviser (Vere 1983) and IxTeT (Ghallab & Laruelle, 1994; see also Ghallab, Nau, & Traverso, 2004, chapter 14), version 2.1 of PDDL (Fox & Long 2003), and the temporal STRIPS model), though this is not the case for all planning models (the aforementioned Zeno system (Penberthy & Weld 1994) and a more recent version of the IxTeT system (Trinquart & Ghallab 2001) both work with models that include continuous change).

For example, consider the action of flying a UAV along a path (discussed in section 2.2, page 13). The execution of this action actually consists of an initial turning phase, during which the UAV is still in hovering control mode, followed by a sequence of path segments flown in trajectory following mode and a final transition back to hovering control. At the end of the last segment, and at the end of any segment followed by a segment with sharper curvature, there is also a braking phase. During hovering and trajectory following, concurrent processes (feedback controllers) control the vertical and horizontal velocity of the UAV. Figure 2.9(a) shows a schematic illustration of this action. In the UAV mission planning problem, as it was described in the preceding section, the horizontal position of a UAV `?u` is abstracted to a set of propositions `(at ?u ?p)`, where `?p` ranges over a finite set of discrete points of interest, and the vertical position is even more abstracted, with only a proposition `(airborne ?u)` to distinguish when the UAV is in the air from when it is on the ground. The action of flying the UAV along the path from, *e.g.*, point `p0` to point `p42` (shown in the map in figure 2.4) makes `(at ?u p0)` false at a time point near the start of the action (from a few up to perhaps 15 seconds, depending on the length of the initial turning phase) and makes `(at ?u p42)` true near the end of the action (say within the last 10 seconds). Preconditions of the action are associated with time intervals during which they are required to hold, rather than with time points. For example, the precondition `(airborne ?u)` of the flying action is required to hold before and throughout the entire execution of the action while the precondition `(at`

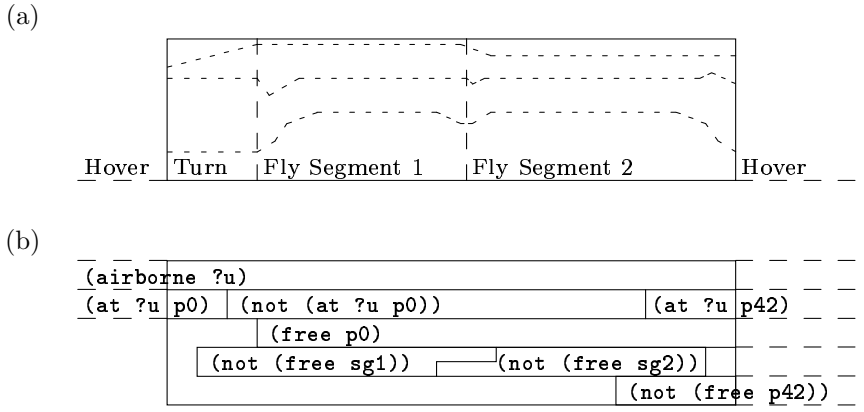


Figure 2.9: Schematic illustration of the action of flying UAV `?u` along the path between points `p0` and `p42`: (a) the control modes and continuous evolution of the world state (heading, altitude, velocity) during and surrounding the action; (b) abstracted view of the action (incomplete).

`?u p0)`, which is destroyed by the action, is only required to hold up to the point at which that effect takes place. Figure 2.9(b) shows part of the abstraction of the action.

Effect time points and precondition time intervals can be, and in fact normally are, related, as illustrated by the example action above. While effect time points can also be related exactly to the start and end of the action, they are usually only constrained to certain intervals within the duration of the action as the continuous processes underlying events are typically too complex to be predicted with perfect accuracy (indeed, even the duration of the action is usually only known to be within an interval, as the end of the action is typically also defined by an event). An action model that specifies the time of effects too precisely will often not be a correct description of the action's execution, and thus plans made on the assumption of such a model will run a high risk of failing when executed. As described above, the execution of the action of flying the UAV along the path to point `p42` consists of invoking a sequence of feedback controllers which each runs until a terminating condition is achieved. The time this takes depends of course on the path flown, but also on many other factors, *e.g.*, wind speed and direction, the current load on the UAV and the accuracy of its position estimate, all of which conspire to make estimating the exact duration of the action practically impossible. An inexact estimate is relatively easy to obtain, however, as the unpredictable factors rarely induce a deviation by more than a few percent of the time predicted by an idealized model.

The temporal STRIPS model is simply the least detailed abstract process model possible: the effects of an action, permanent as well as temporary, are minimally constrained, *i.e.*, only to occur at some point in the interior of the interval of the

actions execution, and the duration of the action is wide enough to ensure that it encompasses all effects (under normal circumstances). Because of this, the model is sometimes referred to as having a *conservative* action semantics. The high level of abstraction does limit the expressive power of the representation, compared to models that allow more precise specification of effect times (*e.g.* the IxTeT problem representation, or PDDL 2.1). For example, the action description depicted in figure 2.9(b), even though it is a conservative abstraction of the actual action execution, does not require that the condition `(free p42)` is true immediately at the start of the action, which the corresponding action in temporal STRIPS representation must do. Thus, a planning problem described using model of higher resolution could have solutions in which a second UAV arrives at point `p0` after the UAV en route to `p42` has vacated it, but well before the time at which it arrives at `p42`. However, expressive power can be increased in other ways as well, for example by allowing the STRIPS assumption that the truth of state propositions persist until affected by an action to be selectively relaxed (this was noted already by Vere, 1983). An extension of this kind to the model, in which state persistence can consume resources, is examined in chapter 5.

A final point to remark on regarding the temporal STRIPS model, as it has been defined here, is the use of rational numbers to model time, since it is more common to model time by the (positive) reals. This is mainly a matter of convenience. Modelling time by real numbers would give the model an increased expressivity, in that non-rational action durations could be specified exactly, but since action durations are generally conservative estimates, and thus not exact to begin with, the error induced by approximating such durations with a rational number (which can also be made arbitrarily small) is probably of no practical significance. Moreover, our techniques for temporal planning (temporal regression and associated heuristics) would work as well with the extended reals, or integers, since the operations required of the time domain are only addition, subtraction and comparisons. Note in particular that the existence of a smallest non-zero increment is not required. However, a complication which is of practical significance is that the temporal regression method requires the ability to reliably determine equality between time expressions (sums of constants). Standard floating point representations of real numbers do not offer this functionality, due to the possibility of round-off errors.

The constraint-based interval (CBI) temporal planning model was introduced with the HSTS planning/scheduling system (Muscettola, 1994, though the model of Allen & Koomen (1983) is very similar; the term "constraint-based interval" was introduced by Smith *et al.*, 2000). It is an example of a model based on the activity centered view, and probably the model of this kind that has had the most influence on developments in planning, outside of HTN planning, perhaps due to the fact that it has been used in some very high-profile applications (*e.g.* the NASA Remote Agent Experiment; Jonsson *et al.*, 2000). The fundamental concept in the CBI model is a *time line*, which is a function from time to values representing some aspect of the world state. A time line can also be seen as the history (or evolution) of a single state variable over the course of the plan. Such a variable can represent not only state, in the

conventional sense, but also on-going activity. Time line values are ascribed meaning by constraints, which describe the valid transitions between values on each time line and dependencies between values on different time lines (this use of constraints to describe the planning problem is not the same as the use of constraints and CSP techniques to represent and reason with search states, which has been applied also in planning systems using temporal STRIPS or similar models). A planning problem is specified by a partial description of the set of time lines, consisting of assertions about the values held by variables over intervals of time, and the task of the planner is to complement this description with additional assertions and further constrain the assertion intervals, until all constraints of the problem are satisfied.

For example, in the temporal STRIPS model of the UAV domain the fact that UAV `?u` is hovering at a position `?p` is represented as state (by the two propositions `(airborne ?u)` and `(at ?u ?p)`), while the fact that the UAV is flying along a path between two positions is represented by an action. In a CBI model of the UAV domain, the two modes of operation could be represented identically, by a variable `state(?u)` (representing the "state" of UAV `?u`) with values `hovering(?p)` and `flying(?p0,?p1)`. Examples of constraints in this domain could be that any interval in which the value of `state(?u)` is `flying(?p0,?p1)` must be immediately preceded by an interval in which the value is `hovering(?p0)` and immediately followed by an interval in which the value is `hovering(?p1)`, and that the length of the interval lies within the (predicted) minimal and maximal time required to fly the path from `?p0` to `?p1`. The requirement that UAVs are kept safely separate could be enforced by constraints specifying that any interval in which the value of `state(?u0)` is `hovering(?p0)` must be disjoint from all intervals in which the value of `state(?u1)` is `hovering(?p1)` or `flying(?p1,?p2)`, for every other UAV `?u1` and every position and path too close to `?p0`. An instance of the problem may specify that from $-\infty$ to t_1 and from t_2 to $+\infty$, `state(uav1)` = `on-ground(p0)` and that from t_3 to t_4 `state(uav1)` = `hovering(p42)`, together with the constraint that $t_4 \geqslant t_3 + 20$; *i.e.*, initially and at the end of the plan, the UAV is on the ground at point `p0`, but for some interval (of at least 20 seconds) in between, it should be hovering at the observation point `p42`. Completing this description into a consistent plan involves adding intervals during which the UAV is taking off, flying along the path to `p42`, flying back and landing, and constraining these intervals relative to those in the problem specification.

Hierarchical Task Decomposition Models

Hierarchical task decomposition models (or hierarchical task network (HTN) models, as they are usually called) were introduced early in the development of planning systems (the NOAH planning system, by Sacerdoti (1975) and the NONLIN system, by Tate (1977), are usually named as the original sources). The first (and until quite recently, only) planning systems to be used in applications (the SIPE system (Wilkins 1990) and the O-Plan system (Tate, Drabble, & Kirby 1994)) were based on HTN models, so clearly such models have some merit (Wilkins & desJardins

(2000) argue this point), but, like logical background theories, HTN models were not considered in the later more formal works on planning (presumably due to the somewhat "procedural" flavour of such models).

The HTN model is based on an activity centered view of planning (as outlined above): it permits the description of the planning problem to specify arbitrary abstract *tasks* (whereas in the STRIPS model, tasks are always related to the achievement of specific conditions on the world state) and to specify arbitrary *methods* of decomposing abstract tasks into networks of (less) abstract tasks or concrete actions (whereas in the STRIPS model, a state condition can be achieved by any sequence of actions with appropriate preconditions and effects). A solution plan is one in which all tasks have been reduced to an executable sequence (or partially ordered set) of concrete actions.

In the Blocksworld domain, for example, the abstract task of moving block `?b` from block `?b0` onto block `?b1` can be decomposed into the two abstract subtasks of making `?b` and `?b1` clear and the concrete action of performing the move, where the first two are ordered before the third, but not ordered *w.r.t.* each other. The abstract task of making a block clear can be accomplished in several ways: by doing nothing if the block is already clear, or by recursively clearing and moving away any blocks on top of the block. Note that in this case, the abstract tasks all correspond to the achievement of state conditions. The available methods, however, need not correspond exactly to all possible ways of achieving those conditions: for example, the methods for clearing a block may be specified so that if a block needs to be moved to make another block clear, it should only be moved to the table. In a STRIPS model, it is not possible to differentiate between moving a block in order to place it somewhere and moving it in order to clear a block below it, and therefore not possible to rule out the use of some actions for achieving one of those tasks. A typical example of an abstract task not corresponding to achieving a state condition, in the context of the UAV domain, may be, "take off, fly around a building, return and land" (a task that need not be as pointless as it seems, if, for example, the UAV is carrying a manually teleoperated camera, or other sensing equipment; it also tends to impress spectators).

Details of the problem representations used by different HTN planning systems vary a great deal (*e.g.* what types of conditions are associated with task decompositions, whether abstract tasks or only concrete actions can have state transforming effects, *etc.*). Compared to planning with the STRIPS model, there has been relatively little work on formal analysis of HTN planning (Erol, Nau & Hendler (1994) and Kambhampati (1994) are two exceptions). Nevertheless, it has been shown that the HTN model is, in the general case, more expressive than the STRIPS model (if the action sequences corresponding to plans are seen as a language, the HTN model is equivalent to a context-free grammar, while the STRIPS model is only regular; see Erol, Nau, & Hendler, 1994) and that the corresponding plan existence problem can be undecidable even if the world model is finite. On the other hand, the explicit specification of decompositions of tasks also makes it possible to restrict the search for a plan, *i.e.*, the HTN model (implicitly) provides a way of describing domain-specific search strategies as part of the problem statement. It has been argued that

this is the only purpose of the HTN methods (and consequently that HTN planning is only a less domain-independent form of STRIPS planning), while others argue that in many (real-world) domains, this flexible ability to restrict the set of plans considered valid solutions to a problem is central to accurately describe the problem.

HTN and CBI models, although interesting, will not be further discussed in this thesis. Investigating if effective admissible heuristics for HTN and CBI models can be derived by principles we develop is a topic for future research.

Searching for Plans

Nearly all approaches to automated planning rely on search, in one form or another. Enriching the description of a planning problem with knowledge beyond the basic problem statement can make it possible to reduce the amount of search needed, and in some specific domains eliminate the need for search altogether (Bacchus & Kabanza 1995; Kvarnström & Doherty 2000). Most planning systems used for real-world applications have relied on vast amounts of problem specific knowledge (see *e.g.* Jonsson *et al.*, 2000, or Wilkins & desJardins, 2000, but also Ruml, Do & Fromherz, 2005, for an example of an application problem solved without domain specific search control). Even in such knowledge-intensive planning systems, however, the planning problem is typically formulated as a search problem and problem specific knowledge added on to effectively guide the search. Early planning systems formulated the problem as one of logical deduction: the problem description is given as a set of axioms, in first order logic, and automated deduction is used to prove the existence of a plan (see *e.g.* Green, 1969; Manna & Waldinger, 1987). The STRIPS planning system was the first to formulate planning as a search problem directly (specifically, as the problem of searching for a path in the space of world states from the initial world state to one satisfying the goal condition, Fikes & Nilsson, 1971).

The search for a plan can be carried out in many different ways. Like the STRIPS and temporal STRIPS planning models, the regression method can be said to be the simplest possible, in the sense that it is comparatively easy to derive admissible heuristics for, and, again like with the choice of planning model, this is our reason for adopting it. There are two main reasons why deriving and applying admissible heuristics for regression search is easier than for other plan search methods. First, in directional search spaces, of which regression is an example, there is a close correspondence between plans (executable and goal achieving action sequences or schedules) and solution paths (paths in the search space leading to a final state) and therefore between the *cost* of plans and *distance* in the search space (recall figure 2.3), which is lacking in non-directional search spaces (a recent version of the IxTeT planning system uses a heuristic to estimate distance in a partial order planning search space, but this heuristic does not yield an admissible estimate of plan cost; Trinquart, 2003). Second, the STRIPS planning models are somewhat asymmetric in that they specify a unique initial world state but only a condition on the goal state, *i.e.*, there may be several world states satisfying the goal condition, which means that more information about the set of final search states is available when computing a regression

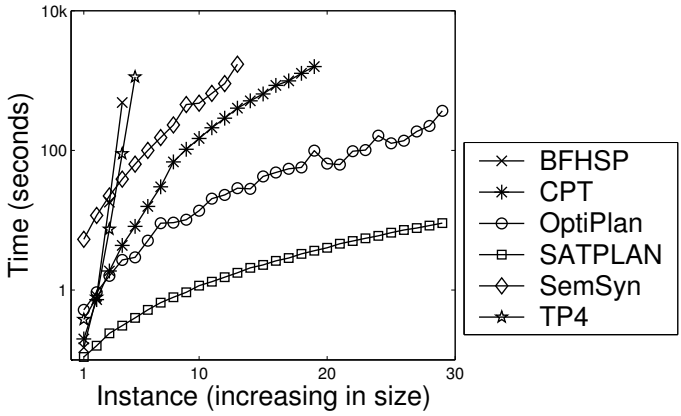


Figure 2.10: Time to find a solution for optimal planners on problems of the Promela domain ("philosophers" subset) from the 2004 International Planning Competition. Instances in this domain are regular, increasing monotonically in size and difficulty.

heuristic (planners that perform heuristic search using progression, such as *e.g.* FF, by Hoffmann, 2000, or the planner by McDermott, 1999, typically perform more computation at each search state evaluation).

Again, however, simplicity comes at a price. It is a fairly well established fact that non-directional plan search methods, such as partial order causal link planning (POCL; McAllester & Rosenblitt, 1991), SAT and CSP encodings (Kautz & Selman 1992; 1996) and others (Rintanen 1998; Hoffmann & Geffner 2003), are more efficient than directional search (see *e.g.* the experimental study by Barret & Weld, 1994). That planning systems using simple directional search, *e.g.*, Graphplan (Blum & Furst, 1995; 1997), HSP (Bonet, Loerincs, & Geffner 1997; Bonet & Geffner 1999) and FF (Hoffmann 2000), have been able to achieve far better performance than partial order planning systems (*e.g.* UCPOP, Penberthy & Weld, 1992; an experimental comparison is presented by Blum & Furst, 1997) is explainable by the fact that development of effective domain-independent search control heuristics for partial order planning has been slower than for the simpler methods. More recently POCL planning and planning methods based on SAT, IP and CSP encodings have been combined with effective heuristics, which has demonstrated that the advantage of non-directional search still exists.

As a case in point, figure 2.10 shows solution times for problems of the Promela domain ("philosophers" subset) from the 2004 International Planning Competition (see Hoffmann & Edelkamp, 2005, or `http://ipc.icaps-conference.org/` for the optimal planners participating in the competition. Among these planners, BFHSP and TP4 are based on directional search (BFHSP, by Zhou & Hansen, 2004, uses regression and progression, switching from one to the other if the first fails; TP4

uses temporal regression as described in this chapter) while CPT, OptiPlan and SATPLAN are based on non-directional search (CPT (Vidal & Geffner 2004) is a temporal partial order planner; OptiPlan (van den Briel & Kambhampati 2004) and SATPLAN (Kautz 2004) are based on encoding into IP and SAT, respectively). The difference in performance between the two directional search planners and the planners using non-directional search is evident, and it is also plausible to ascribe this difference in efficiency to the difference in plan search method since these planners all use essentially the same heuristic: BFHSP, CPT and TP4 use the h^m heuristic (described in chapter 3 of this thesis) and the encodings produced by OptiPlan and SATPLAN are based on the planning graph which encodes a heuristic estimate that is equivalent to the h^2 heuristic (the planning graph and its relation to this heuristic are described in section 3.1 of the next chapter). SemSyn, which uses a kind of bidirectional search (Parker 2004), falls somewhere in between. There is, however, also another difference that may impact the relative performance of the planners: BFHSP and SemSyn find optimal sequential plans (assuming unit action costs) while the other find plans of optimal makespan (though SATPLAN also assumes unit action durations, *i.e.*, performs only parallel planning, as described in section 2.2 above, page 21). Thus, the planning systems are actually solving somewhat different problems.

That the h^m and planning graph heuristics, which were both originally developed for regression planning, are applicable also in the context of other plan search methods is again due to the close correspondence between solution paths in the regression search space and plans. As noted earlier in this chapter (section 2.1, page 10) solving a regression state is equivalent to solving a planning problem with the set of atoms in the state as the goal and therefore an admissible regression heuristic yields a lower bound on the cost of the solution to a certain planning *problem*, which is valid regardless of how the search for this solution is made. This information, in the form of constraints, is incorporated in the pruning techniques used in the CPT planner, and in the IP and SAT encodings generated by OptiPlan and SATPLAN. In this way, admissible regression or progression heuristics can be incorporated even in local search planning (as in *e.g.* the LPG planner, Gerevini & Serina, 2002).

Finally, partial order causal link planning bears a great deal of similarity to planning methods for the constraint-based interval model. Planning methods of this kind are generally believed to be more suited to temporal planning, as it is easier to integrate external events, deadlines and other timing-related constraints.

3. Relaxed Reachability Heuristics

The classical planning problem can be seen as the problem of finding a path from the initial world state to a set of target states (those satisfying the goal condition) in a directed graph (whose nodes correspond to states and links to the possible state transitions, effected by actions). The hardness of the problem is due, quite simply, to the size of the graph, which may be exponential in the size of the description of the planning problem (in propositional form). This combinatorial blow-up is due to the fact that goals (and action preconditions) are conjunctions of atomic facts that need to be achieved simultaneously.

The standard approach to deriving admissible heuristics is to define a *relaxed* (simplified or abstracted) version of the problem, that can be solved optimally at reasonable computational cost, and use the optimal solution cost for the relaxed problem as the heuristic estimate (lower bound) on the solution cost for the original problem (see *e.g.* Gaschnig, 1979; Pearl, 1984). This chapter describes the h^m ($m = 1, 2, \ldots$) family of admissible heuristics, based on a particular relaxation which is to assume that the cost of any set (conjunction) of more than m goals equals the cost of the most costly subset of size m (it is popular to describe this relaxation as "ignoring delete effects", but this is accurate only for the special case $m = 1$). The parameter m offers a trade-off between the accuracy of the heuristic and its computational cost: the higher m the closer the heuristic is to the true cost of achieving all goals in a state, but the complexity of computing the heuristic is exponential in m. The principle by which the heuristic is derived is general and applied here to both sequential and temporal regression planning (sections 3.2 and 3.3, respectively). In chapter 5, the same approach is also used to derive estimators for resource consumption.

The h^m heuristic combines and generalizes ideas from two planning systems: The first is the Graphplan system (Blum & Furst, 1995; 1997) which uses a lower bound function equivalent to h^2 for parallel planning but defines and computes this function in a very different way, through constructing a "relaxed reachability graph" (known as the *planning graph*). The construction and use of the planning graph is briefly reviewed first in this chapter (section 3.1). The second is the HSP system (Bonet, Loerincs, & Geffner 1997; Bonet & Geffner 1999) which uses a different, inadmissible, heuristic but defines (and computes) this heuristic in a way much like that described here.

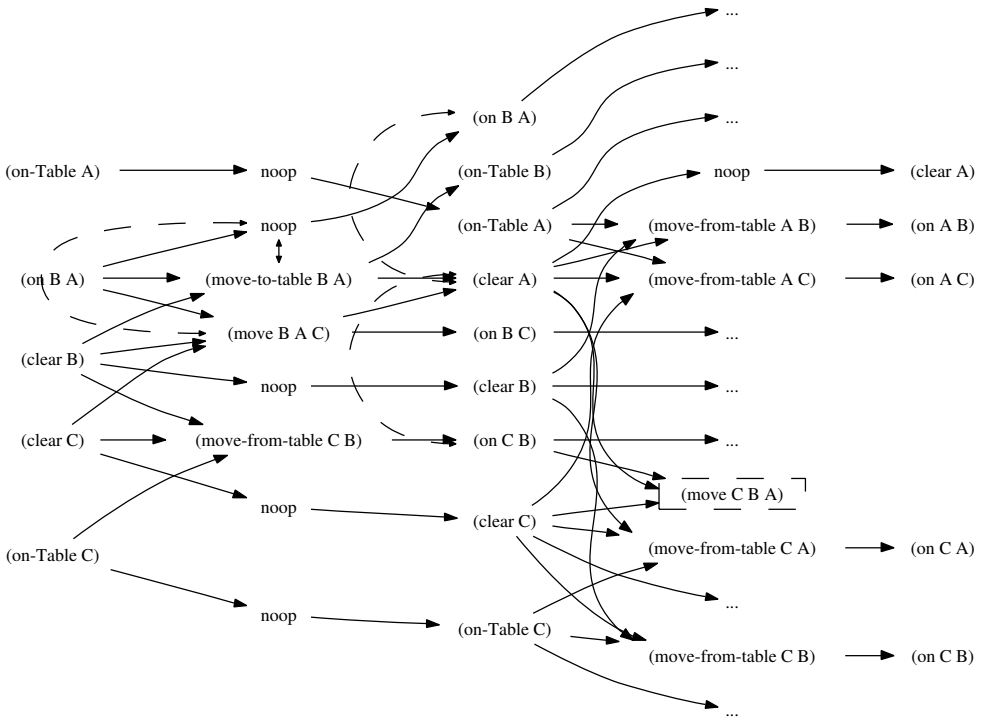


Figure 3.1: Part of the planning graph for the Blocksworld problem shown in figure 2.1. Only a few mutex relations are shown (the dashed arrows connecting atoms/actions in the same layer).

3.1 Relaxed Reachability in the Planning Graph

Admissible relaxed reachability heuristics were introduced into AI planning with the Graphplan system (Blum & Furst, 1995; 1997), which performs parallel regression planning using a heuristic that is equivalent to h^2 for this search space (a formal proof of equivalence is given at the end of this chapter; note, however, that the Graphplan planner was not explained in these terms when originally presented). The heuristic, however, is defined and computed by the construction of a graph, called the *planning graph*. Although the h^m definition is more general, and in a sense simpler, the planning graph provides a more intuitive and "graphical" view of the concept of relaxed reachability. Because of this, we begin this chapter by briefly explaining the planning graph heuristic.

The planning graph is a directed graph, consisting of alternating layers of propositions and actions. The first layer is a proposition layer, and contains all propositions that are true in the initial world state. Each action layer contains every action a, including

no-ops, such that every atom $p \in pre(a)$ can be found in the preceding proposition layer (with an edge from p to a) and each proposition layer following the first contains every atom p such that $p \in add(a)$ for some action a in the preceding action layer (with an edge from a to p). Figure 3.1 shows the first few layers of the planning graph for the example Blocksworld problem described in chapter 2. The initial world state of the problem (illustrated in figure 2.1(a), page 7) contains atoms `(on-table A)`, `(on B A)`, `(clear B)`, `(on-table C)`, and `(clear C)`. Actions in the first layer (besides no-ops) are `(move-to-table B A)`, `(move B A C)` and `(move-from-table C B)`, which add new atoms `(on-Table B)`, `(clear A)`, `(on B C)` and `(on C B)` to the second proposition layer.

Since the atoms in the first proposition layer are initially true, they can be said to be reachable in zero steps. Atoms in the second layer are added by actions that are executable in the initial world state, so they are reachable in one step. The relaxation lies in the reachability of conjunctions of atoms: the fact that two atoms are in the second layer does not imply that the conjunction of those atoms is reachable in one step, since the actions (or no-ops) that support the two atoms may be incompatible (not possible to execute in parallel). For example, even though atoms `(on C B)` and `(clear A)` are in the second proposition layer, the two of them can not be reached in a single step since the action `(move-from-table C B)` is incompatible with both `(move-to-table B A)` and `(move B A C)`. In proposition layers beyond the second, it is not even certain that single atoms appearing are actually reachable, since the preconditions of actions supporting them are conjunctive. However, if an atom is *not* found in the nth proposition layer it is certain that the atom can *not* be reached in $n-1$ steps. Thus, the index of the first proposition layer in which an atom appears is a lower bound on the cost (the required number of steps) of reaching the atom.

The planning graph used by the Graphplan system is actually more intricate: it adds to each layer a binary relation called the *mutual exclusion* relation (*mutex* for short). Two actions a and b are mutex iff they are incompatible, or if some precondition of a is mutex with a precondition of b in the preceding proposition layer. Two propositions p and q are mutex iff every action supporting p is mutex with every action supporting q in the preceding action layer. For example, atoms `(on B A)` and `(clear A)` are mutex in the second proposition layer of the graph in figure 3.1 since the only action supporting `(on B A)` (a no-op) is incompatible with both actions supporting `(clear A)`. In the same way, atoms `(on C B)` and `(clear A)` are also mutex in this layer. If the preconditions of an action are present in a proposition layer, but any two of them are mutex, the action is not included in the following action layer. An example of such an action is shown in figure 3.1: the preconditions of `(move C B A)` are all present in the second proposition layer, but as noted above `(on C B)` and `(clear A)` are mutex, so the action is excluded from the second action layer.

The addition of mutex relations refines the heuristic estimate obtained from the planning graph, in that the conjunction of two atoms is known to be unreachable in $n-1$ steps if the atoms appear *without mutex* for the first time in the nth proposition layer. It is still a relaxation though, since an action may require three (or more) preconditions, and it may be the case that even when any two of them are jointly

reachable the conjunction of all of them is still unreachable. Mutex relations are of two kinds: temporary and static. Temporary mutexes are those that disappear in some subsequent layer. For example, although atoms `(on C B)` and `(clear A)` are mutex in the second proposition layer of the graph in figure 3.1, they are both present and non-mutex in the third proposition layer (because supporting `(clear A)` with a no-op is compatible with supporting `(on C B)` with action `(move-from-table C B)` in the second action layer). Static mutex relations do not disappear: they are in fact a special case of the at-most-one invariants described in the preceding chapter (section 2.1, page 8) consisting only of two atoms.

3.2 h^m Heuristics for Sequential Planning

The definition of the h^m family of heuristics rests on a little bit of dynamic programming theory (see Bertsekas, 1995, ch. 2 of vol. 1 & ch. 1 of vol. 2, or Bellman, 1957). We begin this section by introducing this theory, before entering into its application to sequential regression planning.

The Optimal Cost Function and its Functional Equation

Consider an arbitrary search space, defined by an initial state (s_0), a set of final states and a basic transition relation ($R(s, s', c_{s,s'})$ iff there is a transition from s to s' with cost $c_{s,s'}$). Let $h^*(s)$ denote the optimal cost function, *i.e.*, the function that assigns to each state s in the search space the minimal cost of any path from s to a final state, and ∞ if no such path exists. The optimal cost function satisfies the equation

$$h^*(s) = \begin{cases} 0 & \text{if } s \text{ is final} \\ \min_{\{s' \mid R(s,s',c_{s,s'})\}} h^*(s') + c_{s,s'} & \end{cases} \quad (2)$$

known as *Bellman's equation* (though Bellman, 1957, calls it the functional equation). In fact, this equation uniquely determines $h^*(s)$ over the set of states s from which a final state is reachable. Combined with the condition

$$h^*(s) = \infty \text{ if no final state reachable from } s \quad (3)$$

it characterises the function completely. In principle the optimal cost function, in the form of an explicit table of values, can be computed from the system of equations (2) – (3) by dynamic programming, but this is as hard as solving the corresponding search problem since if the optimal cost function is known the optimal solution path can be trivially extracted (by selecting in every state s a successor state s' with minimum $c_{s,s'} + h^*(s')$).

h^m Heuristics for Sequential Regression Planning

In sequential regression, as described in chapter 2, search states are sets of atoms, representing subgoals to be achieved. Applying Bellman's equation to the sequential regression search space yields

$$h^*(s) = \begin{cases} 0 & \text{if } s \subseteq I \\ \min_{\{s' \mid R(s,a,s')\}} h^*(s') + cost(a) & \end{cases} \tag{4}$$

where $R(s, a, s')$ holds iff s' results from regressing s through action a (recall from chapter 2 that this is the case iff $del(a) \cap s = \emptyset$ and $s' = (s - add(a)) \cup pre(a)$), and I is the set of atoms true in the initial world state (because a set of subgoals s satisfied in the initial world state is a final state in the regression search space).

Because achieving a regression state (set of subgoals) s implies achieving all atoms in s, and therefore any subset of s, the optimal cost function satisfies the inequality

$$h^*(s) \geqslant \max_{s' \subseteq s, |s'| \leqslant m} h^*(s') \tag{5}$$

for any positive integer m. Assuming that this inequality is actually an equality is the relaxation that yields the h^m heuristics: rewriting equation (4) using (5) as an equality results in

$$h^m(s) = \begin{cases} 0 & \text{if } s \subseteq I \\ \min_{\{s' \mid R(s,a,s')\}} h^m(s') + cost(a) & \text{if } |s| \leqslant m \\ \max_{s' \subseteq s, |s'| \leqslant m} h^m(s') & \text{if } |s| > m \end{cases} \tag{6}$$

The solution to equation (6) is a function $h^m(s)$, which is a lower bound on $h^*(s)$ and thus an admissible heuristic for searching in the sequential regression space. The solution, in the form of an explicit table of $h^m(s)$ values for all sets with $|s| \leqslant m$, can be computed in various ways, in time polynomial in the number of atoms in the planning problem but exponential in m (one method, the generalized Bellman-Ford algorithm, is presented below). Admissibility of the h^m heuristic follows from the inequality (5) and the observation that for any $s' \subseteq s$ every action that is applicable in s (can be used to regress s) is also applicable in s' and the result of regressing s' through a is a subset of the state resulting from regression of s through a (a formal proof is given below).

The h^m relaxation can be explained as a change of search space, rather than in terms of solution cost: any state s with more than m atoms is a "max state", whose successors are the size m subsets of s and whose cost is the max of the successor costs, while states s with m or fewer atoms ("min states") are regressed as normal. A max (or "AND") state is solved only if all successors are solved, while a min (or "OR") state is solved if some regression leads to a solution (this view of the h^m relaxation is the basis for the relaxed search method, described in chapter 7). Due to the recursive focus on the most costly size m part of each state the heuristic can also be viewed as a generalization of the well known critical path length estimate. Figure 3.2 illustrates

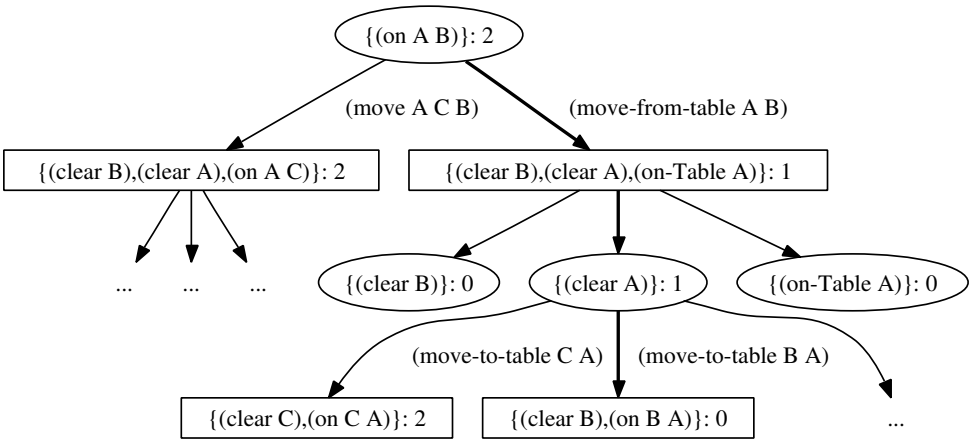


Figure 3.2: Illustration of the calculation of $h^1(\{\texttt{(on A B)}\})$ for the Blocksworld problem shown in figure 2.1. States with more than $m = 1$ atoms are drawn as rectangles: their child nodes correspond to subsets of size m (*i.e.*, with m atoms in the state) and the h^1 value of the state is the maximum of the values of its child nodes. States with $m = 1$ atoms are drawn as ellipses: their child nodes correspond to possible regressions of the state, and the h^1 value of the state is the minimum of the value of the child node plus the cost associated with the edge (in this example, 1 for all actions) over all its child nodes.

the calculation of $h^1(\{\texttt{(on A B)}\})$ for the running example Blocksworld problem (see figure 2.1, page 7), with the critical path that determines the h^1 value of the state indicated by bold arrows in the figure. In general, however, there need not be a single path that determines the value of a state: in a state with more than m atoms there can be several subsets of size m with maximum cost, and a state with m or fewer atoms can have several alternative minimum cost regressions. Thus, the h^m heuristic estimate is more accurately described as "critical tree height". Consider for example a different Blocksworld problem, in which blocks `A`, `B` and `C` are all on the table in the initial world state and the goal is $\{\texttt{(on A B)}, \texttt{(on B C)}\}$ (*i.e.*, to build a tower with `A`, `B` and `C` in order from top to bottom). Figure 3.3(a) shows the calculation of the h^1 value for this problem (which is 1), while figure 3.3(b) shows the corresponding calculation of the h^2 value (which is 2, and also the optimal solution cost). This illustrates how the accuracy of the heuristic improves with increasing m, but also that this improvement is in some cases not sufficient: the problem is easily generalized to constructing a tower of n blocks, for which the optimal solution requires $n-1$ moves, but the h^m heuristic value for this problem is m (for $m \leqslant n-1$).

The h^m value of single state can be calculated recursively, as illustrated in figures 3.2 – 3.3, but the *complete* solution to equation (6), tabulating h^m values for all states with at most m atoms, is more efficiently computed using a dynamic programing

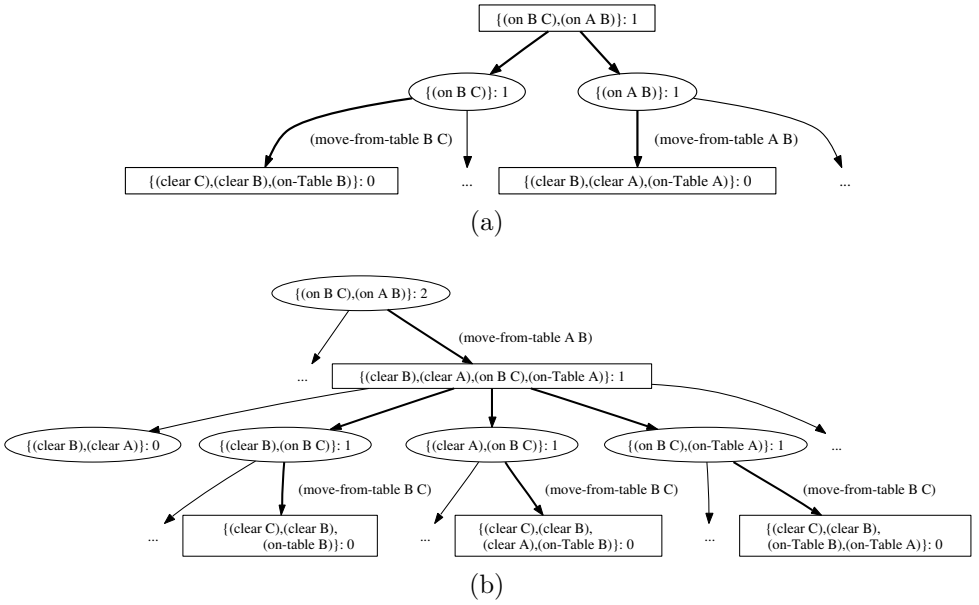


Figure 3.3: Calculation of (a) the h^1 value and (b) the h^2 value for the goal of the "tower construction" Blocksworld problem. The "critical tree" is indicated by bold arrows in each.

or generalized shortest path algorithm. A variation of the generalized Bellman-Ford algorithm is presented below (see Liu *et al* (2002) for some alternative methods).

The parameter m offers a trade-off between the accuracy of the heuristic and its computational cost. As m increases, the relaxation (last clause of equation (6)) plays a lesser role and the heuristic function more and more resembles the optimal cost function. At the same time, however, the computation of a complete h^m solution is polynomial in the number of atoms but exponential in m (since the number of subsets of size m or less grows exponentially with m). Also, the heuristic resulting from a complete solution to the h^m equation exhibits for many planning problems a "diminishing marginal gain": once m goes over a certain threshold (typically, $m = 2$) the improvement brought by the use of h^{m+1} over h^m becomes smaller for increasing m. This combines to make this method of computing the heuristic cost effective, in the sense that the heuristic reduces search time more than the time required to compute it, only for small values of m (typically $m \leqslant 2$; Zhou & Hansen (2004) report also using h^3). This is also supported by the experimental analysis presented later in this chapter (page 46). In chapter 7, we present methods for computing only *partial* solutions to the h^m equation (by two methods, one of which is similar to the recursive calculation illustrated in figures 3.2 – 3.3). The partial solution methods are sometimes more efficiently computable and therefore applicable for higher values

```
eval(s)    // s = {p1, ..., pn}
{
  v = 0;
  for (i = 1 ... size(s)) {
    v = max(v, T(pi));
    for (j = i+1 ... size(s))
      v = max(v, T(pi,pj));
  }
  return v;
}
```

Figure 3.4: State evaluation algorithm for the h^2 heuristic. $\mathtt{T}(p)$ and $\mathtt{T}(p,q)$ denote the h^2 values stored in the heuristic table for atom p and atom set $\{p,q\}$, respectively.

of m.

On-Line Evaluation and the Heuristic Table

The solution to equation (6) that is stored comprises only the values of $h^m(s)$ for sets s such that $|s| \leqslant m$. To obtain the heuristic value of an arbitrary state, the last clause of equation (6), *i.e.*,

$$h^m(s) = \max_{s' \subseteq s, |s'| \leqslant m} h^m(s')$$

is evaluated "on-line", and during this evaluation the value of $h^m(s')$ for any set s' such that $|s'| \leqslant m$ is obtained by looking it up in the table (the evaluation algorithm for the case when $m = 2$ is sketched in figure 3.4). This makes the complexity of the heuristic evaluation of a state $O(|s|^m)$.

A more general heuristic table and evaluation procedure can be implemented, although with a little overhead. This will be important in chapter 7, were several complete and partial solutions to equation (6) for different values of m are combined to form an improved heuristic. In this case, the heuristic table is a general mapping from sets of atoms to their associated value, and the heuristic value of a state s is the maximal value of any subset of s for which a value is stored in the table. In other words, if $(s, v) \in T$ denotes that s is stored with value v in the table, the heuristic value of a state s is given by

$$h(s) = \max_{(s',v) \in T, s' \subseteq s} v.$$

When all and only sets of size m or less are stored in the table (as is the case when h^m is computed completely), this coincides with evaluating the last clause of equation (6), as described above. However, the use of a general heuristic table and evaluation procedure implies that as soon as a value for *any* atom set s is stored in the table, it becomes immediately included in all subsequent evaluations of atoms sets containing s. In particular, by storing parts of the solution to $h^{m'}$, for some higher m', in the

form of updates of the values of some size m' atom sets, the heuristic evaluation implicitly computes the maximum of h^m and the partially computed $h^{m'}$.

The general heuristic table is implemented as a Trie (see *e.g.* Aho, Hopcroft, & Ullman, 1983). Using this structure, the evaluation of an atom set s can be done in time linear in the number of subsets of s for which values exist in the table. The Trie data structure stores mappings indexed by strings. In the implementation of the heuristic table, atom sets are treated as strings in which atoms appear in an arbitrary, but fixed, lexical order. However, when an atom set s is stored in the table, every set that is a prefix of s (viewed as a string with atoms in lexical order) must also be stored, with value 0 if no better value is available. With most methods of computing heuristic values this does not present a problem since whenever a set is stored, all its subsets (including subsets corresponding to lexical prefixes) have already been stored. Even so, there is some overhead compared to a table and evaluation procedure designed for a fixed maximal subset size.

Properties of the h^m Heuristic

Here, we formally show the admissibility and consistency of the h^m heuristic.

Theorem 1 h^m, for $m \geqslant 1$, is admissible.

Admissibility is shown by first showing that for any large enough m, $h^m = h^*$ (lemma 2 below) and second that the heuristic function is decreasing in m, *i.e.*, that if $m < m'$ then $h^m(s) \leqslant h^{m'}(s)$, for all s (lemma 3). Combined, this entails that for any m, $h^m(s) \leqslant h^*(s)$, for all s, or, in other words, that h^m is admissible.

Lemma 2 For every planning problem with n propositions, there is an $m \leqslant n$ such that $h^m(s) = h^*(s)$, for all s.
Proof: If m is larger than or equal to the size of every set of goals s in the regression search space, the last clause of equation (6) is never invoked and equation (6) reduces to equation (4) which characterises the optimal cost function h^*. Clearly no set of goals can contain more than the number propositions in the planning problem. □

Lemma 3 $h^m(s) \leqslant h^{m'}(s)$, for all s, whenever $m < m'$.
Proof: To show that $h^m(s) \leqslant h^{m'}(s)$ for $m < m'$, assume this is not the case, *i.e.*, that for some s, $h^m(s) = v > v' = h^{m'}(s)$. Since there are only finitely many states, it can be assumed that v is the smallest value for which this holds. If $|s| > m$, $h^m(s) = v = h^m(s')$ for some subset s' of s such that $|s'| = m$. (The case when $|s| \leqslant m$, is basically the same, except that the subset $s' = s$ and the minimizing action a below is the same for h^m as for $h^{m'}$.) It must be the case that $h^{m'}(s'') < v$ for every size m' subset s'' containing s', since otherwise $h^{m'}(s) \geqslant v$ (contrary to assumption). Therefore, it is enough to consider the simpler case when $|s| = m'$, and $h^m(s) = h^m(s')$ for some size m subset of s. By definition, $h^{m'}(s) = \min_{\{s'' \mid R(s,a,s'')\}} h^{m'}(s'') + cost(a)$: suppose a is an action that minimizes this expression, *i.e.*, $h^{m'}(s) = h^{m'}((s - add(a)) \cup pre(a)) + cost(a)$ (replacing s'' by the result of

regressing s through a). It follows that $h^{m'}((s-add(a))\cup pre(a)) = v'-cost(a)$. The condition for regression applicability is that $s\cap del(a) = \emptyset$, so if this holds for s it must also hold for s', since $s' \subset s$. Thus $h^m(s') \leqslant h^m((s' - add(a)) \cup pre(a)) + cost(a)$ ($\leqslant$ rather than $=$ since there may be actions with smaller cost applicable to s' but not to s) and thus $h^m((s'-add(a))\cup pre(a)) \geqslant v-cost(a) > v'-cost(a)$ (due to the assumption that $v > v'$ and the fact that $cost(a) > 0$). However, since $(s'-add(a))\cup pre(a) \subseteq (s-add(a))\cup pre(a)$, $h^m((s-add(a))\cup pre(a)) \geqslant h^m((s'-add(a))\cup pre(a))$, and thus $h^m((s - add(a)) \cup pre(a)) \geqslant v - cost(a) > v' - cost(a) = h^{m'}((s - add(a)) \cup pre(a))$, contradicting the assumption that v is the smallest value such that $h^m(s) = v > v' = h^{m'}(s)$. Since such a smallest value must exist if $h^m(s) > h^{m'}(s)$ for any s, this can not be the case.

Above, we've used the fact that the inequality (5) holds also for h^m, *i.e.*, that $h^m(s) \geqslant h^m(s')$ for any $s' \subset s$. This again follows from the fact that any actions that is (regression) applicable to s is also applicable to s'. □

Theorem 4 h^m, for $m \geqslant 1$, is consistent.
Proof: Consistency is shown by essentially the same argument. Assume that it does not hold, *i.e.*, that for some state s and action a applicable to s, $h^m(s) > h^m((s - del(a)) \cup pre(a)) + cost(a)$. $h^m(s) = h^m(s')$ for some $s' \subseteq s$ such that $|s'| \leqslant m$ (if $|s| \leqslant m$, $s' = s$). Since $s' \subseteq s$, a is applicable in s' and thus $h^m(s) \leqslant h^m((s'-del(a))\cup pre(a))+cost(a)$. Since $(s'-del(a))\cup pre(a) \subseteq (s-del(a))\cup pre(a)$, $h^m((s'-del(a))\cup pre(a))+cost(a) \leqslant h^m((s-del(a))\cup pre(a))+cost(a)$, contradicting the assumption. □

The Generalized Bellman-Ford Algorithm

The Generalized Bellman-Ford (GBF) algorithm is a "label correcting" algorithm: it assigns a crude initial cost estimate to every state and then iteratively applies local updates until costs converge. It can be viewed as an adaptation of the Bellman-Ford single-source shortest path algorithm (see *e.g.* Cormen, Leiserson, & Rivest, 1990) to graphs with directed multi-edges.

To describe the algorithm, it is convenient to rewrite equation (6) into a set of "update equations" for a fixed value of m. For $m = 2$, this results in

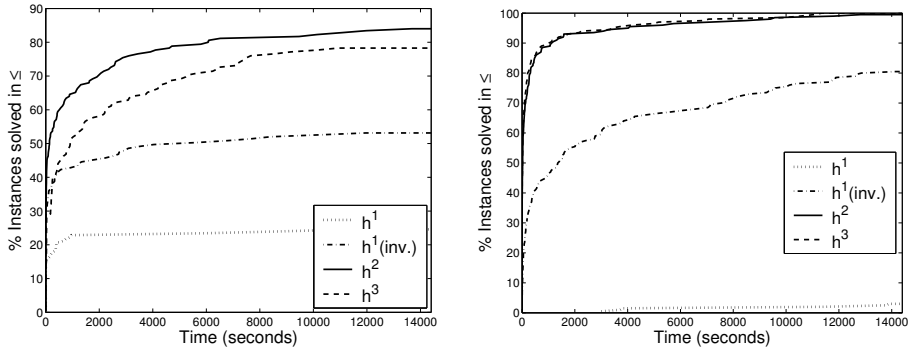
$$
\begin{aligned}
h^2(\{p\}) &= \min_{\{a \mid p \in add(a)\}} h^2(pre(a)) + cost(a) \\
h^2(\{p,q\}) &= \min\Bigg(\min_{\{a \mid p,q \in add(a)\}} \Big(h^2(pre(a)) + cost(a) \Big), \\
&\qquad \min_{\{a \mid p \in add(a), q \notin del(a)\}} \Big(h^2(pre(a) \cup \{q\}) + cost(a) \Big), \\
&\qquad \min_{\{a \mid q \in add(a), p \notin del(a)\}} \Big(h^2(pre(a) \cup \{p\}) + cost(a) \Big) \Bigg)
\end{aligned}
$$

```
GBF_H2()
{
  // initialization:
  for (each atom p) {
    if (p in initial world state) T({p}) = 0;
    else T({p}) = +INF;
  }
  for (each atom pair p,q) {
    if (p,q in initial world state) T({p,q}) = 0;
    else T({p,q}) = +INF;
  }

  // main loop (repeat until no change)
  repeat {
    changed = false;
    for (each action a) {
      c1 = eval(pre(a));
      for (each atom p in add(a)) {
        // update single atoms added by action a
        update({p}, c1 + cost(a));
        // update pairs of atoms added by action a
        for (each atom q in add(a), q != p)
          update({p,q}, c1 + cost(a));
        // update atom pairs with p added by a and r by persistence
        for (each atom r not in del(a), r != p) {
          // the already computed c1 = h(pre(a)) can be used to speed
          // up computation of c2
          c2 = eval(pre(a) union {q});
          update({p,r}, c2 + cost(a));
        }
      }
    }
  } until (not changed);
}

// update cost of atom set s (and note change) if the new value is smaller
// than the previously stored value
update(s, v)  // s is a set of one or two atoms
{
  if (T(s) > v) {
    T(s) = v;
    changed = true;
  }
}
```

Figure 3.5: The GBF algorithm for computing the h^2 heuristic for sequential planning (a complete solution to equation (6)). `T` denotes the heuristic table (indexed by sets of one or two atoms).



(a) Blocksworld (b) UAV Mission Planning

Figure 3.6: Runtime distribution for regression planning (IDA* search) with the h^m heuristics for $m = 1, 2, 3$ in (a) the Blocksworld domain and (b) the single UAV mission planning domain. The h^1 heuristic is used with and without explicit invariant pruning (the former is labeled "h^1 (inv.)").

These equations correspond to the possible regressions of a set of at most 2 subgoals: a set of only one atom $\{p\}$ can only be (usefully) regressed through an action that adds p; a set of two atoms $\{p, q\}$ can be regressed through either an action that adds both p and q, or an action that adds one and does not delete the other. It is straightforward to derive the corresponding set of equations other values of m.

The GBF algorithm initializes the heuristic table with cost 0 for every size m set of atoms that are true in the initial world state, and $+\infty$ for every other size m set. It then evaluates the right-hand side of each update equation in turn, and if the computed value is less than what is stored for the atom set to the left atom, updates the stored cost. This is repeated until no update equation yields a value lower than the value already stored in the heuristic table. Evaluating the right-hand sides of the above equations involves evaluating sets containing more than two atoms relative to the current contents of the heuristic table. As noted above, heuristic evaluation is somewhat expensive (quadratic for h^2) and hence efficiency can be improved by organizing updates to avoid duplicated effort: the resulting algorithm is sketched in figure 3.5.

Analysis: Accuracy/Cost Trade-Off in the h^m Heuristics

This section presents a small experimental analysis of the accuracy and computational cost of the h^m heuristics, for values $m = 1, 2, 3$. The main point demonstrated is the accuracy/cost trade-off and its variation over different planning problems. Experiments also demonstrate an important difference between h^1 and h^m for $m \geqslant 2$, *viz.* the ability of the latter to detect problem invariants.

The Experiment

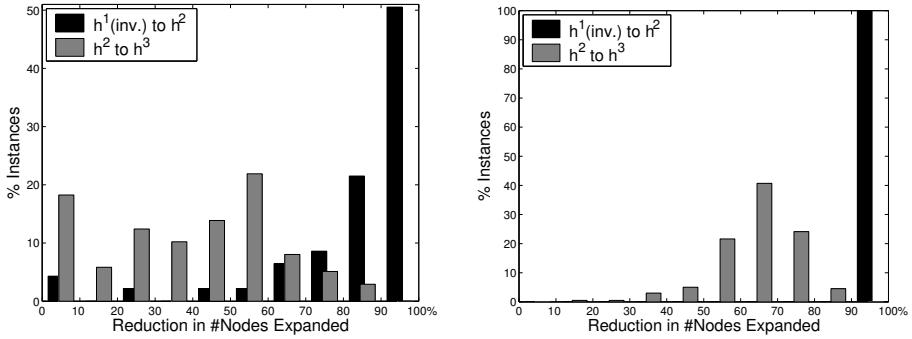
The h^1, h^2 and h^3 heuristics were compared in the context of sequential regression planning using IDA* search on planning problems from two different domains: the Blocksworld domain, described in the chapter 2, and the UAV mission planning domain but involving only a single UAV. (For reasons explained below, a fourth configuration was also included in the comparison *viz.* the h^1 heuristic combined with explicit invariant pruning, as described in chapter 2, page 10.) The search was made using commutativity cuts (described in chapter 2, page 11) and a standard transposition table (as described by Reinfeld & Marsland, 1994).

The UAV mission planning problem was introduced in chapter 2 as an example of a planning problem involving time and concurrent action. With only a single UAV, however, it reduces to a sequential planning problem since the UAV can only do one action at a time and the makespan of the plan equals the sum of the durations of the actions it performs. Also, with only one UAV there is no need to model the "safe separation" condition (so the sequential problem description does not have the `(free ?p)` atoms). The problem is still difficult, however, because it contains what is essentially a traveling salesman problem (see *e.g.* Garey & Johnson, 1979).

The test problems are all solvable so since regression planning is complete every problem would be solved eventually, with any heuristic. For practical reasons, a time limit of 4 hours CPU time per problem was imposed. Problem sizes (the number of blocks and the number of observations, respectively) were scaled from nearly trivial (5 blocks and 3 observations, respectively) to the the largest for which a reasonable percentage could still be solved within the time limit (13 blocks and 6 observations; with the h^2 heuristic, which generally achieves the best performance, roughly a third of the size 13 Blocksworld problems were solved). For the Blocksworld domain, 25 random problems for each size were generated using Slaney & Thiebaux's `bwstates` program (see Slaney & Thiebaux, 2001). For the UAV domain, 50 random problems for each size were generated by selecting from five different combinations of buildings the chosen number of observation points.

The Results

Results are summarized by runtime distributions in figure 3.6. A fact that is immediately apparent is the very large gap between the h^1 and h^2 heuristics: this is due to the presence of (at-most-one) invariants in the planning problems, which are detected by h^m for $m \geqslant 2$ but not by h^1. As described in chapter 2 (page 10) states (atom sets) inconsistent with some invariant are frequently encountered during a regression search, but can never be part of any solution path. Therefore, detecting such states (and pruning them from the search) is important for reducing the number of states searched. Any two atoms p and q that are part of the same at-most-one invariant are jointly unachievable even when both atoms are achievable separately (in planning graph terms, they constitute a static mutex). The h^1 heuristic, which only ever considers the cost of achieving individual atoms, can not detect



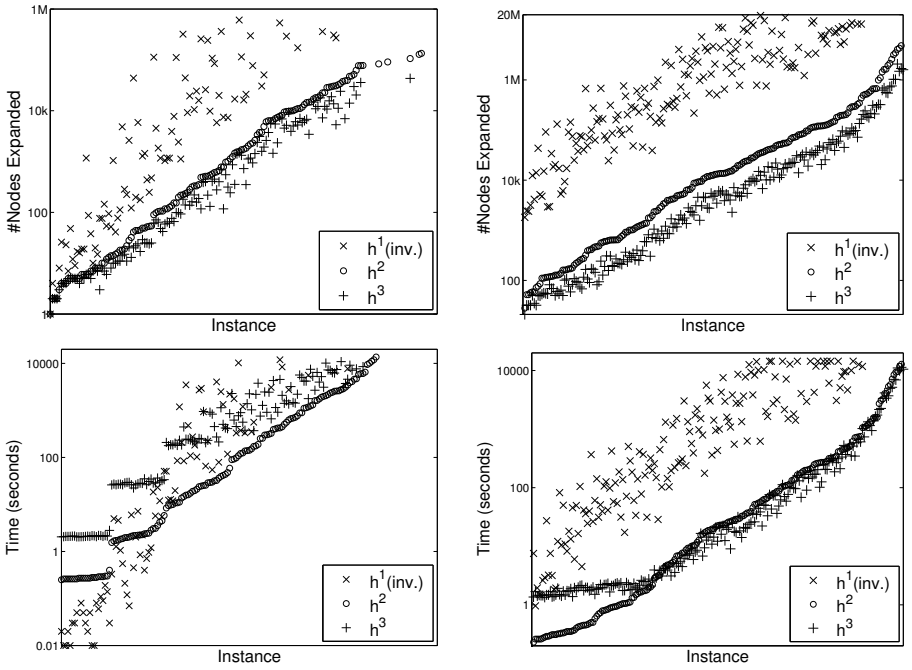
(a) Blocksworld (b) UAV Mission Planning

Figure 3.7: Reduction in the number of nodes expanded when the heuristic is changed from h^1 (with explicit invariant pruning) to h^2 and from h^2 to h^3, respectively. Percentages are relative to the number of problem instances solved by the weaker heuristic in each case.

this ($h^1(\{p,q\}) = \max(h^1(\{p\}), h^1(\{q\}))$ is finite since $h^*(\{p\})$ and $h^*(\{q\})$ both are, while in fact $h^*(\{p,q\}) = \infty$). Although there is in general no guarantee that the h^2 heuristic (or h^m for other $m \geqslant 2$) detects mutexes (*i.e.*, assigns infinite cost to such atom sets) it very frequently does in practice. In the two planning domains used in the experiment, h^2 detects all invariants. That the detection of invariants is responsible for a great part of the improvement in changing from h^1 to h^2 is demonstrated by the fact that h^1 with explicit invariant pruning (the curves labeled "h^1 (inv.)" in figure 3.6) achieves a performance significantly closer to that of h^2.

The second observation that can be made is that even when invariant detection is accounted for, the gain in changing from h^2 to h^3 is far smaller than the gain for the change from h^1 to h^2. In the Blocksworld domain, runtimes obtained using h^3 are actually worse than those obtained with h^2. Figures 3.7 and 3.8 show results in greater detail. Figure 3.7 shows how the reduction in the number of states explored by the search ("nodes expanded") in changing from h^1 (with explicit invariant pruning) to h^2 and from h^2 to h^3 distributes over the (solved) problem instances: the change from h^1 to h^2 reduces the number of nodes expanded by over 90% in half the Blocksworld instances and all the UAV problem instances, while the reduction caused by the change from h^2 to h^3 is smaller, and more varied, in both domains. This illustrates the "diminishing marginal gain" mentioned above.

That the number of nodes expanded is reduced at all demonstrates that the h^3 heuristic is more accurate than h^2 in both domains, but it is not *cost effective* in the Blocksworld domain. Recall that the time required to compute the complete h^m solution (*a.k.a.* the heuristic table) grows exponentially with m, as does the time to evaluate the heuristic over a search state. A more accurate heuristic is cost effective only when the time required to compute it is less than the search time saved due to



(a) Blocksworld (b) UAV Mission Planning

Figure 3.8: Comparison of the number of nodes expanded (first row) and runtime (second row) using heuristics h^1 (with explicit invariant pruning), h^2 and h^3. Instances are sorted by the displayed value for h^2 in each graph. Note that the Y-axis (nodes expanded/time) is logarithmic.

the smaller number of expanded search states compared to using a less accurate, but computationally cheaper, heuristic. In the Blocksworld domain, this is not the case for h^3. In the single UAV planning domain it is, but only by a very small margin.

Figure 3.8 provides a more detailed view of the accuracy and effectiveness of the compared heuristics. Here data (expanded nodes and runtime, respectively) is plotted by instance, with instances sorted in a rough order of increasing difficulty (note that the sort order is not the same in the four graphs). This illustrates the effect of a structural difference between the two domains, and why h^3 is cost effective for UAV planning but not for Blocksworld. The reduction in expanded nodes generally increases with increasing problem difficulty (note that the scale on the Y-axis is logarithmic). In the Blocksworld domain problem difficulty generally increases with the size of the problem, while the UAV planning problems used in the experiment are of constant size (in terms of the number of atoms and actions in the problem description) and the difficulty is instead related to the number of goal atoms. The effect of this is that in the Blocksworld domain, the increasing cost of using h^3 (although it is only polynomially related to problem size) "keeps pace" with the gain. Note that for the easiest problem instances, runtimes using h^2 and h^3 in the Blocksworld domain and h^3 in the UAV planning domain are almost constant since they are dominated by the time needed to compute the h^m solution.

Conclusions

Two principal conclusions can be drawn from the experiment described above. The first is the importance of detecting and pruning states inconsistent with problem invariants in regression search. This holds true for most planning problems. The second is that if the h^m heuristic is not effective enough to solve a particular planning problem, increasing m alone is not likely to improve performance (for $m \geqslant 2$ at least), although the "break even point", in terms of m and in terms of problem difficulty, varies with the structure of the problem. In chapters 6 and 7 we examine alternative ways to improve the h^m heuristics (computing several h^m heuristics, for low m, in such a way that estimates can be admissibly *added* and computing only *partial* h^m heuristics, for higher m, respectively).

3.3 h^m Heuristics for Planning with Time

The h^m heuristics for temporal planning are derived in basically the same way as in the sequential case. Bellman's equation for the optimal cost function has the same form as equation (4). Recall that a search state in the temporal regression space is a pair $s = (E, F)$, where E is a set of atoms (subgoals to be achieved) and F a set of concurrent actions with starting times relative to the time of achievement of the

atoms in E. The corresponding equation is[2]

$$h^*(E,F) = \begin{cases} 0 & \text{if } E \subseteq I \text{ and } F = \emptyset \\ \min_{\{s' \mid R(s,s')\}} h^*(s') + \delta_{\text{adv}}(s,s') & \end{cases} . \tag{7}$$

Here, $R(s,s')$ holds iff s' can be obtained from s by regression through a set of compatible actions (as described in chapter 2, page 16) and $\delta_{\text{adv}}(s,s')$ is the corresponding increase in plan makespan (defined by equation (1), page 16).

To apply the h^m relaxation, approximating the cost of a state by the "most costly size m subset", requires a definition of the size of a state (and of what constitutes a "subset" of a pair (E,F)) The obvious candidate is to define $|s = (E,F)| = |E| + |F|$, and this measure (with componentwise subsets) does indeed satisfy the relaxation inequality (5), and rewriting the optimal cost equation using this as an equality results in a characterisation of a lower bound function on the temporal regression space. In this case, however, due to the presence of a time increment δ in each $(\delta, a) \in F$, the set of states with $|s| \leqslant m$ is potentially infinite, which means a complete solution to the equation, in the form of an explicitly tabulated function, can not computed. (The number of distinct δ values appearing in the tabulated function can actually be bounded, by the longest action duration divided by the *gcd* of the durations of all actions, but since this can be a *very* large number the characterised heuristic is still impractical.)

To define a practically usable heuristic function, a further relaxation is applied. Since a plan that achieves the state $s = (E,F)$, for $F = \{(a_1,\delta_1),\ldots,(a_n,\delta_n)\}$, at time t must achieve the preconditions of each action a_i at time $t - \delta_i$, and these must remain true until t unless deleted by a_i, the optimal cost function satisfies

$$h^*(E,F) \;\geqslant\; \max_{(a_k,\delta_k)\in F} \left(h^* \left(\bigcup_{(a_i,\delta_i)\in F,\, \delta_i \geqslant \delta_k} pre(a_i), \emptyset \right) + \delta_k \right) \tag{8}$$

$$h^*(E,F) \;\geqslant\; h^* \left(E \cup \bigcup_{(a_i,\delta_i)\in F} pre(a_i), \emptyset \right). \tag{9}$$

An example may clarify the principle: Consider the state $s = (\{p\}, \{(a_1,1),(a_2,2)\})$, depicted in figure 3.9(a). A plan achieving this state at time t must achieve the preconditions of a_2 at $t-2$, so $h^*(s)$ must be at least $h^*(pre(a_2),\emptyset) + 2$. If action a_2 is "left out", as in figure 3.9(b), it can be seen that the same plan also achieves the joint preconditions of actions a_1 and a_2 at $t-1$, so $h^*(s)$ must be at least $h^*(pre(a_1) \cup pre(a_2),\emptyset) + 1$. Finally, if both actions are left out (figure 3.9(c)), it is clear that the plan also achieves simultaneously the preconditions of the two actions and atom p, so $h^*(s)$ must be at least $h^*(\{p\} \cup pre(a_1) \cup pre(a_2),\emptyset)$.

By treating inequalities (8) – (9) as equalities, a temporal regression state is relaxed to a set of states in which $F = \emptyset$, *i.e.*, states containing only goals and no concurrent

[2]Note that one set of parentheses has been simplified away from $h^*(E,F)$: since a state $s = (E,F)$ is a pair, it should in fact be written "$h^*((E,F))$". This simplified form, with only a single pair of parentheses, will be used throughout.

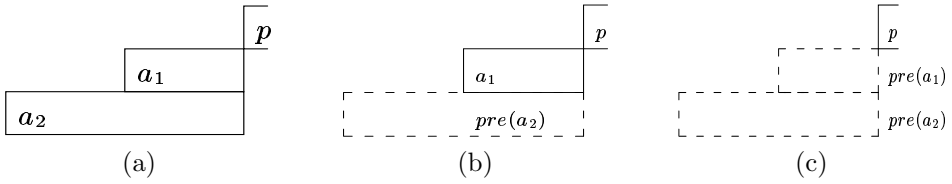


Figure 3.9: Relaxation of temporal regression states.

actions. Applying the relaxation (5) to each such state results in an equation similar to equation (6), defining the temporal h^m heuristic:

$$h^m(E,F) = \begin{cases} \max\left(\max_{(a_k,\delta_k)\in F} h^m(\bigcup_{\substack{(a_i,\delta_i)\in F,\\ \delta_i \geqslant \delta_k}} pre(a_i),\emptyset) + \delta_k\right), & \text{if } F \neq \emptyset \\ \quad h^m(E \cup \bigcup_{(a_i,\delta_i)\in F} pre(a_i),\emptyset) & \\ 0 & \text{if } F = \emptyset,\ E \subseteq I \\ \min_{\{(E',F')\,|\,R((E,F),(E',F'))\}} h^m(E',F') + \delta_{\text{adv}} & \text{if } F = \emptyset,\ |E| \leqslant m \\ \max_{E'\subseteq E, |E'|\leqslant m} h^m(E',\emptyset) & \text{if } F = \emptyset,\ |E| > m \end{cases} \tag{10}$$

This equation has a finite explicit solution, in the form of a table of the h^m values for states $s = (E,\emptyset)$ with $|E| \leqslant m$. Methods for computing the solution are the same as in the sequential case. The h^m heuristics for temporal planning are admissible and consistent, and the properties shown in lemmas 2 and 3 (page 43), that $h^m(s) = h^*(s)$ for any sufficiently large m (with the number of atoms in the planning problem as an upper bound) and that $h^m(s) \leqslant h^{m'}(s)$ for all s when $m < m'$, hold for the temporal case as well.

Evaluation of Temporal Regression States

Heuristic evaluation of a temporal regression $s = (E,F)$ is done analogously to the sequential case, by evaluating the relaxation equations (first and last clause of equation (10)) on-line and looking up the values of states $(E,\emptyset)$ with $|E| \leqslant m$ in the precomputed heuristic table. It is, however, possible to strengthen the relaxation of concurrently scheduled actions (first clause of equation (10)) somewhat, thus obtaining better heuristic values. Reconsider the relaxation of the state $s = (\{p\},\{(a_1,1),(a_2,2)\})$, illustrated in figure 3.9: if every action that can be used to establish atom p (*i.e.*, that adds p) is incompatible with action a_1, then atom p must be established by a no-op at least from the point where a_1 starts; in other words, $h^m(s)$ must be at least $h^m(pre(a_2) \cup pre(a_1) \cup \{p\},\emptyset) + 1$. If all possible establishers for p are incompatible with a_2, the lower bound $h^m(pre(a_2) \cup \{p\},\emptyset) + 2$ is similarly obtained, and the heuristic value is of course set to the maximum of all lower bounds, in accordance with the relaxation inequalities (8) – (9). The information about incompatible atom/action pairs can be computed in advance and cached.

```
eval(F, NOOP, E)  // F = {(a1,delta1), ..., (aN,deltaN)}
{
  sort F by decreasing delta values;
  A = {};  // empty set
  d = delta1;  // greatest delta value
  v = 0;
  for (i = 1 ... N) {
    add pre(ai) to A;
    for (each p in E or NOOP, p not compatible with ai) {
      add p to A;
      remove p from E or NOOP;
    }
    v = max(v, eval(A) + deltai);
  }
  min delta = min dur(a) over all a that add some p in E;
  if (min delta < deltaN) {
    add NOOPS to A;
    v = max(v, eval(A) + min delta;
  }
  add NOOPS and E to A;
  v = max(v, eval(A));
}
```

Figure 3.10: Procedure for heuristic evaluation of (partially constructed) temporal regression states. `F` is the set of scheduled actions (including both those of the parent state and the newly selected establishers), `NOOP` the set of atoms selected to be established by no-ops (empty for a completely constructed state), and `E` the set of subgoal atoms (atoms remaining to regress for a partially constructed state). `eval(A)` gives the heuristic estimate for a set of atoms (see figure 3.4).

This is particularly important for the early detection of cost bound violations during incremental successor state construction, as described in the previous chapter (page 18). The procedure for heuristic evaluation of a partially constructed successor state, sketched in figure 3.10, takes as input a set of scheduled actions, which includes both scheduled actions (the F component) in the parent state and the already selected new actions (there is no essential difference between them, as illustrated in figure 2.6(b), page 18), a set of selected no-ops and the set of atoms remaining to regress (a subset of the E component of the parent state), and returns an estimate of the h^m value of the completed successor state plus the makespan increment δ_{adv}. The inconsistent establishers optimization described above is applied both to atoms corresponding to already selected no-ops and atoms remaining to regress. Selected no-ops that are not "stretched" in this way have a minimum duration equal to the smallest possible value that δ_{adv} can take, which is the minimum of the δ values among (old and new) scheduled actions in the state and the shortest duration among all actions that may be selected as establisher for any of the atoms remaining to regress.

Evaluation of finished states is done by the same procedure, the only difference being that the set of selected no-ops is empty.

3.4 Discussion

The Graphplan and HSP planning systems outperformed the best comparable planning systems in existence at the time when they were introduced, in spite of the fact that they both use simple forms of directional search (regression and progression, respectively), due to the use of effective search control (Blum & Furst, 1995; Bonet, Loerincs, & Geffner, 1997). This issue had previously received relatively little attention, except in the context of knowledge-intensive planning systems, *e.g.* O-Plan (Tate, Drabble, & Dalton 1994), TLPlan (Bacchus & Kabanza 1995) or PRODIGY (Veloso *et al.* 1995). Heuristics for certain local decisions in partial-order planning had also been studied (*e.g.* by Peot & Smith, 1993).

The heuristics employed by Graphplan and HSP, however, estimate the required cost (number of actions or time steps) of a plan to achieve a goal which in a directional plan search this corresponds closely with the *distance* to a solution in the search space, since steps taken in the search space correspond to actions (or time steps) added to the plan. The definition of the HSP heuristic is very similar to h^1: the difference is that the cost of a set of more than one atom is estimated with the *sum* of the estimated cost of atoms in the set, instead of only the most costly atom. For sequential planning (where cost is additive) this is often closer to the true cost, but it is not admissible since there can be positive interactions between subgoals, *i.e.*, cases where some action contributes to the achievement of more than one subgoal, and is therefore counted by the heuristic more times than it actually appears in the plan. (In chapter 6 we describe a method that allows subgoal cost estimates to be added without loss of admissibility, resulting in a more accurate variant of the h^m heuristic for sequential planning.) The HSP heuristic is computed (tabulated) for single atoms and evaluated on-line for larger atom sets in essentially the same way as described for the h^m heuristics in this chapter.

Several efficient but non-optimal planning systems based on state space search guided by inadmissible variants of relaxed reachability heuristics have since been developed (*e.g.* GRT (Refanidis & Vlahavas 1999), FF (Hoffmann 2000) and FastDownward (Helmert 2004), to name a few). The planning graph has also inspired a great deal of innovation: it has been analyzed, improved, extended and put to many different uses. In particular, the planning graph has been used as a basis for devising a variety of very effective inadmissible heuristics, both for classical sequential planning (including the famous FF heuristic and others; see *e.g.* Hoffmann, 2000 and Nguyen, Kambhampati, & Nigenda, 2002) and for planning problems involving time/resource/cost trade-offs (Do & Kambhampati 2000).

The h^2 Heuristic and the Planning Graph: A Comparison

The development of admissible heuristics for optimal cost (or makespan) planning has been less explosive than the development of inadmissible heuristics. Besides the h^m heuristics and the planning graph, pattern databases is the only alternative approach to deriving admissible heuristics for planning to have been tried (Edelkamp 2001).

Pattern database heuristics are the subject of chapter 4 and their relation to the h^m heuristics is discussed there. As mentioned at the beginning of this chapter, the planning graph heuristic is equivalent to h^2 for parallel planning (temporal planning with unit action durations). A formal proof of this equivalence is given in the next section below. An equivalent of h^2 for sequential planning can also be easily obtained from the planning graph by making all non-noop actions in each action layer mutex (thus allowing at most one action in each time step). The planning graph has been extended to yield an admissible makespan heuristic for temporal planning, in the TGP (Smith & Weld 1999) and TPSys (Garrido, Onaindia, & Barber 2001) planning systems.

However, the planning graph has more uses than only as a tool for computing a heuristic value: it is also an explicit representation of the space of bounded-makespan plans. As such, it has been used for structuring and indexing information learned during the plan search (a simple form of learning, called memoization, was used in the Graphplan system; Kambhampati (2000) describes the application of more advanced learning methods). The main advantage of the explicit plan space representation, however, appears to be as a basis for efficient non-directional search: encoding the problem of finding a plan in the graph as a satisfiability or constraint satisfaction problem has been shown to be much more effective than encoding the (bounded) planning problem directly, but also more effective than the regression search through the graph done by the Graphplan system (Kautz & Selman 1999; Do & Kambhampati 2000). Hoffmann & Geffner (2003) show how such a non-directional search can be done on the planning graph directly, without the need for an intermediary encoding of the problem. In fact, the idea of a representation of the space of bounded-length plans by alternating layers of atoms and actions was used in the SATPLAN system (Kautz & Selman 1992). The planning graph adds reachability information, resulting from a particular admissible heuristic, to this representation (by removing unreachable atoms and actions, and by annotating unreachable, *i.e.*, mutex, pairs), which obviously could be done using any admissible heuristic.

Equivalence of the h^2 and Planning Graph Heuristics

We show the equivalence of the h^2 and planning graph heuristics only for the case of parallel planning (temporal planning with unit durations) as this is the most natural formulation of the planning graph. The temporal h^2 heuristic estimates the makespan required to reach search states of the form $s = (E, F)$, where E is a set of atoms representing subgoals to achieve and F a set of concurrently scheduled actions, but when all actions have unit durations only states with $F = \emptyset$ appear (see discussion in chapter 2, page 21). Thus, we identify a set of atoms s with the search state $(s, \emptyset)$.

The construction of the planning graph was described in section 3.1 above. Let $h^G(s)$ denote the planning graph estimate of the cost (number of parallel action steps) required to achieve the set of atoms s, *i.e.*, the index of the first proposition layer in the graph in which all atoms in s appear and there is no mutex between any of them.

Theorem 5 $h^2(s) = h^G(s)$, for all s.

Proof: We show that if $h^G(s)$ is finite then $h^2(s) \leqslant h^G(s)$ and, conversely, that if $h^2(s)$ is finite then $h^G(s) \leqslant h^2(s)$. Equivalence follows since this implies that if either estimate is finite they are equal, and if not then both must be infinite (and the atom set s thus unachievable).

Both proofs are by induction: we begin with showing that $h^G(s) \leqslant h^2(s)$ when $h^2(s)$ is finite. For the base case, suppose $h^2(s) = 0$, which it can be only if all atoms in s are true in the initial world state; if this is the case then all atoms in s also appear in the zeroth layer of the planning graph, without mutex since there are no mutexes in this layer, so $h^G(s) = 0$ also holds. For the induction step, assume that $h^G(s) \leqslant h^2(s)$ holds whenever $h^2(s) \leqslant k$, for all s. Suppose that $h^2(s) = k + 1$: then for each subset $s' \subseteq s$ such that $|s'| \leqslant 2$, and $h^2(s') \leqslant k + 1$, with equality for at least one such subset. By definition of the temporal h^2 heuristic (see equation (10) above) $h^2(s') = k + 1$ iff there exists some state s'' that results from regression of s', such that $k + 1 = h^2(s'') + \delta_{\text{adv}}$, where δ_{adv} is the corresponding increase in plan makespan (defined by equation (1), page 16). Because $|s'| \leqslant 2$, the regression leading to s'' is through a set of two compatible actions $\{a_1, a_2\}$ (one of which may be a no-op) and because actions have unit durations, $\delta_{\text{adv}} = 1$ and s'' consists only of the set of precondition atoms of those actions. Thus, $h^2(s'') = k$ and by the induction assumption $h^G(s'') \leqslant k$, *i.e.*, the joint set of preconditions of actions a_1 and a_2 appears without mutex in proposition layer k (or earlier) of the graph, which means that a_1 and a_2 appear in the following action layer. Since the actions are compatible there is no static mutex between them, so by the construction of the planning graph all atoms in $add(a_1) \cup add(a_2)$, which includes those in s', appear, without mutex, in the next proposition layer, and because this holds for every subset $s' \subseteq s$ of at most two atoms, $h^G(s) \leqslant k$.

The proof that $h^2(s) \leqslant h^G(s)$ when $h^G(s)$ is finite is very similar: the induction step is just the reverse of the above, and rests on the fact that if two atoms appear without mutex in proposition layer $k+1$ of the planning graph then there are two non-mutex (and thus compatible) actions a_1 and a_2 whose preconditions are present and non-mutex in the preceding proposition layer, and that if $h^2(pre(a_1) \cup pre(a_2)) \leqslant k$ then $h^2(add(a_1) \cup add(a_2)) \leqslant k + 1$. □

4. Pattern Database Heuristics

A Pattern Database (PDB) is a memory-based heuristic function obtained by abstracting away all but a part of the problem (the *pattern*) small enough to be solved optimally for every state by blind exhaustive search. The results are stored in a table in memory (the *pattern database*) and define an admissible heuristic function by mapping states into corresponding abstract states and reading their associated value from the table (Culberson & Schaeffer, 1996; 1998; Hernadvölgyi & Holte, 2000). Heuristic estimates from multiple abstractions can be combined by taking their maximum or, under certain conditions, their sum. Pattern database heuristics have been successfully applied to a number of search problems (Culberson & Schaeffer 1998; Felner, Korf, & Hanan 2004) and to sequential STRIPS planning (Edelkamp 2001).

The main problem with PDB heuristics is that since the time and memory required to compute and store the pattern database limits the size of the pattern, the quality of the heuristic depends crucially on the selection of an appropriate pattern (or patterns). In domain-independent planning, where the appropriate pattern can vary a lot between different planning problems and selection should ideally be automatic, this is particularly troublesome.

This chapter reviews the principles of pattern database heuristics and Edelkamp's application of PDB heuristics to planning (section 4.1), and introduces an improved abstraction, which enables better heuristic values to be obtained without increasing the size of the pattern (section 4.2). Finally, some strategies for automatically selecting patterns are presented, with an experimental analysis of the quality of the resulting heuristics (section 4.3).

4.1 PDB Heuristics for STRIPS Planning

Pattern database heuristics are defined by abstractions, which are mappings between search spaces with the property that optimal solution cost in the space mapped to is a lower bound on optimal solution cost in the space mapped from (Culberson & Schaeffer, 1996; Hernadvölgyi & Holte, 2000; the term "pattern" refers to states in the abstract space, in the sense that all states mapped to the same abstract state "match the pattern"). Recently, Edelkamp (2001) showed how such abstractions, and thereby PDB heuristics, can be constructed for domain-independent (sequential) STRIPS planning. In Edelkamp's formulation, a pattern consists of a subset of the propositions (or variables) of a planning problem and defines an abstraction mapping by ignoring propositions (or variables) not in the pattern.

Abstractions of Search Spaces

Consider a search space, S, defined by a basic transition relation ($R_S(s, s', c_{s,s'})$ iff there is a transition from s to s' with cost $c_{s,s'}$ in S), an initial state and a set of final states, and let $h^*_S(s)$ denote the optimal cost function, *i.e.*, the function that assigns to each state s in the search space the minimal cost of any path from s to a final state, and ∞ if no such path exists. An *abstraction* of S is a surjective mapping φ from S to a search space S' that preserves transitions and transition costs, *i.e.*, if $R_S(s, s', c_{s,s'})$ then $R_{S'}(\varphi(s), \varphi(s'), c_{s,s'})$, and that preserves final states, *i.e.*, if s is a final state in S then $\varphi(s)$ is final in S'. Any such mapping defines an admissible heuristic function h^φ on S, through $h^\varphi(s) = h^*_{S'}(\varphi(s))$. Admissibility follows from the fact that φ preserves solution paths: if $s, \ldots, s_n \in F$ is a solution path in S then $\varphi(s), \ldots, \varphi(s_n)$ is a solution path in S', with equal cost. Since h^* is defined as the minimum cost over all solution paths, $h^\varphi(s) = h^*_{S'}(\varphi(s)) \leqslant h^*_S(s)$ (less than or equal because there can exist solutions in S' that do not correspond to a solution in S).

If the abstract space S' is small enough the optimal cost function $h^*_{S'}(s)$ can be computed for all s by means of a breadth-first search from the final states "backwards" along the transition relation, and the results stored in a table (this is essentially equivalent to solving Bellman's equation for $h^*_{S'}$ by dynamic programming, as described in chapter 3). The table constitutes a pattern database (PDB) for the original search space S and allows the heuristic $h^\varphi(s)$ to be computed by computing only the abstraction $\varphi(s)$ and looking up the corresponding value in the table. The reverse search computes, of course, only the optimal costs of states from which a final state is reachable. For efficient indexing, the table must contain all states in the abstract space, so the values for unsolvable states (which are by definition all ∞) must be entered in an extra step.

Abstraction of STRIPS Planning Problems

Applied to STRIPS planning, abstractions can be defined by selecting a subset of the atoms in the planning problem and ignoring all other atoms. This subset, A, defines an abstracted planning problem in which the initial and goal states, and the preconditions, positive and negative effects of each action are simply intersected with A. The abstract problem can be viewed as a relaxation of the original planning problem where only the status of some propositions counts. Following Edelkamp, we refer to the atom set A as the *pattern*. The pattern A defines a mapping, φ^A, from the regression search space associated with the original planning problem to the corresponding search space associated with the abstract planning problem, also by intersection with A ($\varphi^A(s) = s \cap A$). This mapping is an abstraction, as previously defined, and thus the pattern defines an admissible heuristic $h^A(s)$ for the original search space as described above (in fact, the pattern defines an abstraction mapping also for the progression search space associated with the problem, as well as for some other planning search spaces, in the same way; a formal proof for the case of progression planning is given by Edelkamp, 2001).

Any subset A' of a pattern A defines an abstraction mapping from (the search space associated with) the abstract planning problem determined by A to (the search space associated with) the abstract planning problem determined by A' in exactly the same way, which implies that optimal solution cost in the latter, "more abstract", space is also a lower bound on optimal solution cost in the former, "less abstract", space. In other words,

$$h^{A'}(s) \leqslant h^{A}(s) \tag{11}$$

for all s when $A' \subseteq A$, and in particular

$$\max(h^{A}(s), h^{B}(s)) \leqslant h^{A \cup B}(s) \tag{12}$$

for all s and for any patterns A and B. Under certain conditions this can be strengthened to

$$\max(h^{A}(s), h^{B}(s)) \leqslant h^{A}(s) + h^{B}(s) \leqslant h^{A \cup B}(s) \tag{13}$$

in which case patterns A and B are said to be *additive*. In the case of sequential planning, where the cost of a plan is the sum of the costs of actions in the plan, additivity holds between atom sets A and B iff $add(a) \cap A = \emptyset$ or $add(a) \cap B = \emptyset$ for every action a, *i.e.*, no action adds atoms belonging to both sets.

To show that this is sufficient to ensure the right inequality in (13), consider a state s and its optimal solution path $s, \ldots, s_n$ in the abstract space defined by the pattern $A \cup B$. This path corresponds to a sequence of actions $a_1, \ldots, a_n$ (where s_i is obtained by regression of s_{i-1} through a_i). The actions can be divided in two sets: those that add some atom in A and those that add some atom in B, and by assumption these sets are disjoint (actions that do not add any atom in either A or B are redundant in the abstract space defined by $A \cup B$, and therefore can not appear on the optimal solution path). The subsequence of $a_1, \ldots, a_n$ consisting only of actions in the first set is a solution path in the abstract space defined by the pattern A (if it were not, this could only be because some precondition of one of the actions is not achieved, but this precondition either belongs to A, in which case the sequence would not be a solution in the $A \cup B$ abstract space, or does not belong to A, in which case it is ignored in the abstract space defined by A and so can not affect the validity of the solution path) and therefore $h^{A}(s)$ can not be greater than the sum of the costs of the actions in this subset. The same argument applies to the subset of actions that add atoms in B. Because the action subsets are disjoint it follows that the sum $h^{A}(s) + h^{B}(s)$ can not be greater than the cost of the solution path $s, \ldots, s_n$, which was optimal in the abstract space defined by the pattern $A \cup B$ and hence equal to $h^{A \cup B}(s)$.

Relation (11) indicates that the quality of a PDB heuristic, in general, improves with increasing pattern size. However, if the pattern A contains n atoms the corresponding abstract space has 2^n abstract states, and the corresponding PDB contains 2^n entries, so available memory limits the pattern size. Several approaches to reducing the memory required have been proposed: some exploit problem-specific properties, *e.g*,

symmetries (Culberson & Schaeffer 1998; Felner *et al.* 2005), while others are general (Felner *et al.* 2004; Edelkamp 2002). The time required to compute the PDB also grows linearly with the number of entries. Additive patterns allow PDB heuristics to be constructed using exponentially less memory, since the size of the PDB for pattern $A \cup B$ equals the product of the sizes of the PDBs for patterns A and B while the heuristic $h^A + h^B$ requires memory only equal to the sum of the PDBs for patterns A and B. However, the relation between $h^A + h^B$ and $h^{A \cup B}$ is an inequality, because there can be negative interactions between actions relevant to atoms in A and actions relevant to atoms in B that are not visible in the abstract spaces defined by patterns A or B, but that are detected by $h^{A \cup B}$. Selecting the collection of patterns yielding the best heuristic within given memory bounds is an intricate problem, which is discussed further in section 4.3 below.

Patterns based on State Variable Representations

Edelkamp (2001) based patterns on a (multi-valued) state variable representation of the planning problem rather than the propositional STRIPS representation (though the state variable representation is constructed from exactly-one invariants, which are automatically extracted from the STRIPS representation). As described in the previous section, any subset of the atoms in a planning problem defines a pattern and a corresponding PDB heuristic. The reason for basing patterns on state variables is to make more effective use of memory: whereas in general a pattern consisting of n atoms gives rise to 2^n abstract states, and thus to a PDB with 2^n entries, a pattern consisting of a state variable with n values (corresponding to a set of n atoms forming an exactly-one invariant) gives rise to only n abstract states, and a PDB with n entries (the size of the PDB for a pattern consisting of several state variables is of course still equal to the product of the ranges of the variables).

In this chapter, we will assume that the planning problem is available in state variable representation as well as propositional STRIPS representation, and that we can translate freely between the two, without worrying about how the representations and the mapping between them are obtained. As described in chapter 2 (page 24) a state variable corresponds to an exactly-one invariant containing the values of the variable. Invariants can be extracted automatically from the STRIPS representation by a variety of methods (*e.g.* Gerevini & Schubert, 1998; Fox & Long, 1998; Scholz, 2000). Two examples of state variable representations of planning problems are described below, mainly for illustrative use in the remainder of the chapter.

The correspondence between state variables and exactly-one invariants implies that the condition for additivity of two patterns based on state variables is that no action changes the value of variables in both patterns. When this is true for two patterns consisting of a single variable each, we simply say that the two variables are additive. Note that two patterns are additive if and only if every variable in one is additive with every variable in the other.

There is a slight complication with the use of patterns based on state variables to

create PDB heuristics for regression planning: since a regression search state consists only of a set of atoms required to be true (and does not require atoms not in the set to be false) the atom set may map to a partial assignment of values to state variables and therefore the range of each state variable has to be extended with a "don't care" value. (We say a variable is undefined in an assignment if it has this special value and defined if it has any of the normal values in the variables range; an assignment is complete if all variables (in the pattern of interest) are defined, and partial otherwise.) Note that a complete assignment to a set of state variables determines a unique world state (an assignment of truth values) relative to the set of propositions corresponding to the state variables. The cost associated with an abstract state in which one or more variables of the pattern are undefined equals the minimum over all completions of the abstract state, *i.e.*, complete assignments of the variables that agree with the partial state on all variables that are defined (because a regression state is equally achieved no matter what the truth value of any atom not in the state). Partial assignments can either be explicitly represented in the PDB (which increases the size of the PDB) or determined "on-line", by minimizing over all completions *w.r.t.* the pattern (which increases the computational cost of evaluating states). A procedure for computing the regression PDB (with explicit partial assignments) for a state variable pattern is presented below, after the two examples.

Example: Blocksworld

The Blocksworld planning problem was introduced in chapter 2. As described there, there are two exactly-one invariants for each block: one that describes where the block is and one that describes what is on top of the block. We name the corresponding state variables `pos(?b)` and `top(?b)`. The values of `pos(?b)` are `(on-table ?b)` and `(on ?b ?c)` (for each `?c` $\neq$ `?b`), and the values of `top(?b)` are `(clear ?b)` and `(on ?c ?b)` (for each `?c` $\neq$ `?b`). We use the names of the atoms in the propositional STRIPS representation of the problem as the names of state variable values as an easy way to describe the correspondence between the two representations. The world state depicted in figure 2.1(a) (chapter 2, page 7) maps to the complete assignment

```
pos(A)  =  (on-table A),  top(A)  =  (on B A),
pos(B)  =  (on B A),      top(B)  =  (clear B),
pos(C)  =  (on-table C),  top(C)  =  (clear C)
```

while the goal state description {`(on A B)`} maps to the partial assignment `pos(A) = (on A B)`, `top(B) = (on A B)`.

The `pos(?b)` variables are all additive, since every action moves only one block. The `top(?b)` variables are not additive, because an action that moves a block from being on `?b` to being on `?c` changes both `top(?b)` and `top(?c)`. For each block `?b`, `pos(?b)` and `top(?b)` are additive, since no action can move `?b` and move something to or from the top of `?b` at the same time, but `top(?b)` is not additive with `pos(?c)` for any block `?c` $\neq$ `?b`.

14	13	15	7
11	6	9	5
12		2	1
4	8	10	3

(a)

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(b)

Figure 4.1: Two configurations of tiles in the 15-Puzzle (the arrangement in (b) is the standard goal state). The sum of Manhattan distances heuristic for state (a) in relation to goal (b) is 37 while the optimal solution cost is 55.

For an example where a good PDB heuristic is easy to find, consider again the "tower construction" Blocksworld problem (also described in chapter 3, page 40): n blocks ($\mathtt{b}_1$ to $\mathtt{b}_n$) are on the table in the initial world state, and the goal is $\{\mathtt{(on\ b}_1\ \mathtt{b}_2\mathtt{)}, \ldots, \mathtt{(on\ b}_{n-1}\ \mathtt{b}_n\mathtt{)}\}$, *i.e.*, to have the blocks form an ordered tower with $\mathtt{b}_1$ at the top and $\mathtt{b}_n$ at the base. The PDB heuristic based on a pattern consisting of a single variable $\mathtt{pos(b}_i\mathtt{)}$ gives a value of 1 for this problem, since one action is required to change the value of this variable to the value specified by the goal state (except for block $\mathtt{b}_n$ which does not need to be moved), and since the patterns are all additive these values can be summed over all blocks resulting in a heuristic value of $n - 1$, which is also the optimal solution cost.

Example: The 15-Puzzle

The "$(n^2 - 1)$-Puzzle" (8-Puzzle, 15-Puzzle, 24-Puzzle, *etc.*) problem is a popular benchmark for evaluating search algorithms and related techniques in AI. The puzzle consists of a board with $n \times n$ squares occupied by $n^2 - 1$ numbered tiles. This leaves one square empty, and tiles (horizontally or vertically) adjacent to the empty square can be moved into it (the empty square is traditionally called the "blank"). The goal of the puzzle is to rearrange the tiles from the given initial placement to a specified goal configuration (usually with the tiles ordered by number, left-to-right and top-to-bottom). An example of initial and goal configurations (for $n = 4$, *i.e.*, the 15-Puzzle) is shown in figure 4.1. The 8-Puzzle is the smallest version, and is considered easy. Problem-specific search implementations solve the 15-Puzzle easily and can also solve most instances of the 24-Puzzle (Korf & Taylor 1996). Recently, a combination of pattern database heuristics and a suite of problem-specific enhancements have made it possible to solve some instances of the 35-Puzzle (Felner, Korf, & Hanan 2004). For a domain-independent planning system, however, solving the 15-Puzzle optimally is still a challenge.

A propositional STRIPS representation of the problem has atoms `(at ?t ?x ?y)`,

where `?t` = `T1`, ..., `T15` ranges over the 15 tiles and `?x`, `?y` = `1`, ..., `4` range over the 16 positions on the board, and atoms `(blank ?p)`, describing the places occupied by the tiles and the empty square. The action to move, for example, tile `?t` to the left has preconditions `(at ?t ?x ?y)` and `(blank ?x-1 ?y)`, deletes both of these and adds `(at ?t ?x-1 ?y)` and `(blank ?x ?y)` instead.

In the state variable representation there is a variable `pos(?t)` for each tile `?t`, with values `(at ?t ?x ?y)` (for `?x`, `?y` = `1`, ..., `4`), and a variable `blank` with values `(blank ?x ?y)` (also for `?x`, `?y` = `1`, ..., `4`), since every tile, and the blank, is in exactly one square in any world state. There are also variables `in(?x,?y)` for all coordinates `?x`, `?y`, with values `(at ?t ?x ?y)` for each tile `?t` and `(blank ?x ?y)`. These represent what is in the square at `?x`, `?y`, which is exactly one of the tiles or nothing (the blank).

There is a simple and well known admissible heuristic for the puzzle, based on summing over all tiles the "Manhattan distance", which is the horizontal and vertical distances added together, from the tiles current position to its position in the goal state. This heuristic can be explained as the optimal solution to a relaxed problem, in which the tiles are allowed to move into any adjacent square (whether empty or occupied by another tile), but it can also be reconstructed as the sum of a collection of additive PDBs, each for a pattern consisting of a single `pos(?t)` variable (Korf & Taylor 1996).

Computing Regression PDBs

The value stored for each abstract state s in a PDB is the minimum cost of any path from s to a final state in the abstract space, which is the same as the minimum cost over any path from a final state to s under the inverse transition relation. Thus, computing the PDB for a given pattern amounts to solving a shortest path problem (similar, but not identical, to computing the h^m solution as described in chapter 3). In a straightforward formulation of "inverse regression" in the abstraction defined by a pattern consisting of state variables, the starting (zero cost) states are all (complete and partial) assignments to variables in the pattern that agree with the initial world state of the planning problem. An action a is applicable to a (partial) assignment iff the assignment defines, and agrees with the preconditions of a, on all variables (in the pattern) on which a has a precondition, and when applied leads to all (partial) assignments that agree with the effects of a on at least one of the variables in the pattern and that for each variable (in the pattern) changed by a either agrees with the effects of a (on this variable) or is undefined.

A slightly different approach leads to a simpler (and perhaps also more effective) procedure: divide the computation of the PDB in two steps, where the first is to apply a standard single-source shortest path algorithm to compute the values of all complete assignments (*w.r.t.* the pattern) and the second is to fill in the values of partial assignments, using the fact that the PDB value of a partial assignment is the minimum over the values of all its completions. The two-step procedure is shown

```
ComputeRegressionPDB({V1,...,Vn}, PDB)
{
  // initialization:
  for (each assignment s to {V1,...,Vn}) {
    PDB[s] = +INF;
  }
  // step 1: compute complete assignments (Djikstra's algorithm)
  s = initial world state assignment to {V1,...,Vn};
  PDB[s] = 0;
  initialize queue to contain s with cost 0;
  while (queue not empty) {
    s = min cost assignment in queue;
    remove s from queue;
    for (each action a) {
      if (s agrees with pre(a) on {V1,...,Vn}) {
        s' = apply effects of a to s;
        PDB[s'] = PDB[s] + cost(a);
        insert s' with cost PDB[s'] in queue;
      }
    }
  }
  // step 2: compute partial assignments
  ComputePartial({V1 = undefined, ..., Vn = undefined});
}

ComputePartial(s)
{
  if (s is complete) {
    return PDB[s];
  }
  else {
    V = first undefined variable in s;
    min val = +INF;
    for (each value v of V) {
      s' = s completed with V = v;
      val = ComputePartial(s');
      min val = min(val, min val);
    }
    PDB[s] = min val;
    return min val;
  }
}
```

Figure 4.2: Procedure for computing the regression PDB for a pattern consisting of state variables `V1,...,Vn`. Function `ComputePartial` finds the value of a partial assignment `s` by recursively minimizing over all completions, and also stores the values of all "partial completions" of `s`, as well as the value found for `s`, in the PDB during the process.

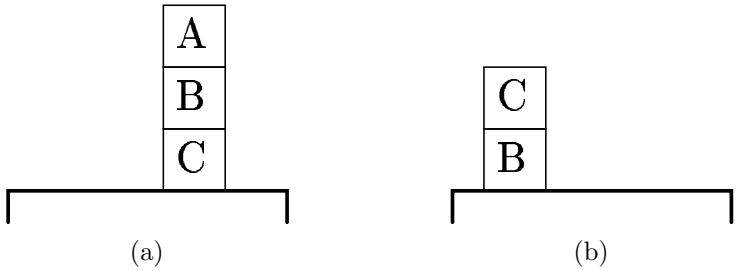


Figure 4.3: The "swap base blocks" Blocksworld problem, for three blocks: (a) initial world state; (b) goal state description. Block `A` does not figure in the goal state description, since the goal mentions only the two blocks at the base of the tower in the initial world state.

in figure 4.2. Alternatively, only the values for complete assignments can be stored in the PDB and values for partial assignments determined on-line by a function like `ComputePartial` in figure 4.2.

4.2 Constrained Abstraction

In the abstract planning problem defined by a subset of atoms (A) above, atoms not in A are ignored completely. A consequence of this is that interactions between actions caused by conflicting requirements on atoms not in A do not arise in the abstract problem, which often causes the PDB heuristic to underestimate the true cost of solving a state by more than what is necessary. To some extent this problem can be remedied by including the atoms causing conflicts in the pattern, but this is not really a satisfactory solution (since the size of patterns that can be used is typically limited). This section presents a better approach, which uses a stronger abstraction that carries certain invariants from the original planning problem into the abstract search space. The stronger abstraction retains the nice properties of the weaker abstraction, such as the condition under which patterns are additive.

For example, consider a Blocksworld problem in which there are n blocks ($\mathtt{b}_1$,...,$\mathtt{b}_n$) that in the initial world state form an ordered tower with $\mathtt{b}_n$ at the base and $\mathtt{b}_1$ at the top, and the goal is to swap the two bottom blocks, *i.e.*, the goal is `(on` $\mathtt{b}_n$ $\mathtt{b}_{n-1}$`)`. Figure 4.3 illustrates the initial and goal states for the case of three blocks (named `A`, `B` and `C`). A PDB heuristic based on a pattern consisting of only the variable `pos(`$\mathtt{b}_n$`)` gives an estimated cost of only 1 for this problem. This is because the single-step plan moving $\mathtt{b}_n$ directly to $\mathtt{b}_{n-1}$ is valid in the abstract problem, where everything except the atoms specifying the position of block $\mathtt{b}_n$, including the atoms `(clear` $\mathtt{b}_n$`)` and `(clear` $\mathtt{b}_{n-1}$`)`, has been abstracted away from the preconditions of this move action. For the same reason, the estimated cost is still 1 even if the position of the block immediately above $\mathtt{b}_n$, *i.e.*, variable `pos(`$\mathtt{b}_{n-1}$`)`, is also included in the pattern. The

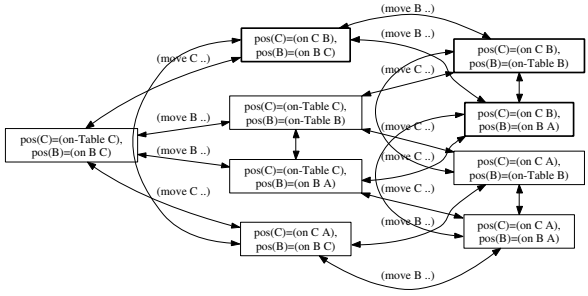
(world) state space of the abstract planning problem defined by this pattern (for the case of three blocks `A`, `B` and `C`) is shown in figure 4.4(a). This is not as reasonable, however, since the value of this variable in the initial state is `(on` $\mathtt{b}_{n-1}$ $\mathtt{b}_n$`)`, which means block $\mathtt{b}_n$ can not be clear. In fact, the two atoms `(on` $\mathtt{b}_{n-1}$ $\mathtt{b}_n$`)` and `(clear` $\mathtt{b}_n$`)` are *mutex* (form a two-atom at-most-one invariant) in the original problem but this fact is lost in the abstraction.

For another example, consider the 15-Puzzle problem shown in figure 4.1(a), specifically tiles 5 and 6: the sum of their Manhattan distances is 4, since both are 2 steps away from their goal positions, but the number of moves required to shift them both into their goal positions is at least 6. This is because they are in a so-called linear conflict: they are in the same row, but need to pass each other, and consequently at least one of them must move "out of the way" and back, adding (at least) 2 extra steps. The PDB heuristic $h^{\{\mathtt{pos(T5)},\mathtt{pos(T6)}\}}$, however, yields only a value of 4. The reason is again that the abstraction removes the preconditions involving the position of the blank, which prevent tiles from moving to an occupied square, from the actions. The problem could be remedied by including the variable `blank` in the pattern, but this increases the size of the PDB, and worse, since it would have to be included in the pattern for every pair of tiles, the corresponding PDBs would no longer be additive.

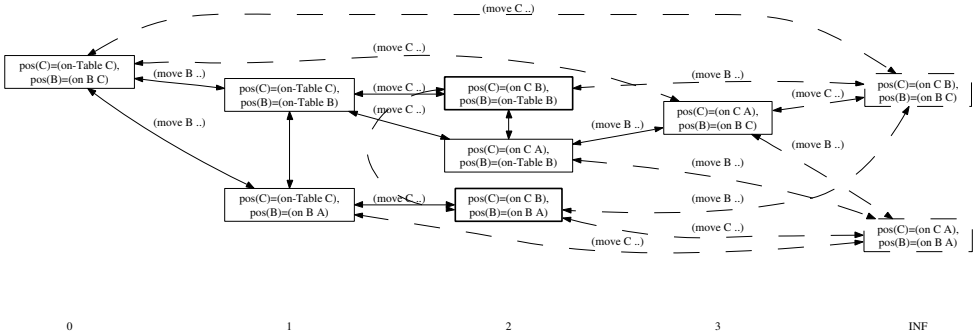
The general problem illustrated by the two examples is that even if two variables are in a pattern, interactions between actions relevant to those variables but caused by conflicting requirements on other variables are lost in the abstraction, unless those other variables are also in the pattern. This is due to the fact that the abstraction completely ignores everything not in the pattern.

Constrained abstraction enforces certain invariants (*viz.* invariants of the at-most-one kind) of the original planning problem in the abstract planning problem, even though they are not invariants of the abstract problem (because some action preconditions have been abstracted away). Simplifying somewhat, it can be described as applying invariant pruning (as described in chapter 2, page 10) to the abstract regression space but using the at-most-one invariants of the original planning problem.

Let $C = \{C_1, \ldots, C_k\}$ be a collection of at-most-one invariants of a planning problem, *i.e.*, sets of the atoms such that for every C_i at most one of the atoms in C_i is true in any reachable world state. Recall from section 4.1 that a pattern A defines an abstraction mapping from the regression search space associated with the original planning problem to an abstract search space (by intersection with A). The *constrained* abstraction (relative to C) is based on the same mapping, but excludes from the abstract search space any transition from a (abstract) search state s to a state s', where s' is the result of regressing s through action a, such that $s' \cup pre(a)$ violates some invariant in C. In other words, the constrained abstraction excludes transitions that are known to lead to unreachable states in the original search space given only the part of the problem kept by the abstraction. Figure 4.4(b) shows the constrained abstract (world) state space for the pattern consisting of variables `pos(B)` and `pos(C)` for the "swap base blocks" problem discussed above.



(a)



(b)

Figure 4.4: (a) State space of the abstract problem defined by the pattern {`pos(B),pos(C)`} for the "swap base blocks" Blocksworld problem (illustrated in figure 4.3). The states are grouped by the minimum cost to reach them (which is indicated at the bottom of the figure); the left-most state is the initial world state. States satisfying the goal condition (`pos(C)=(on C B)`) are drawn in bold. (b) State space of the constrained abstract problem for the same pattern. Excluded transitions are shown as dashed arrows. Note that some states are unreachable in this problem (these are also drawn with dashed lines).

Let $h^A_C(s)$ denote the optimal cost function associated with the constrained abstract space defined by pattern A and invariant set C.

Theorem 6 Optimal cost in the constrained abstract space is an admissible heuristic for the original problem and it is at least as strong as the heuristic obtained from unconstrained abstraction, *i.e.*,

$$h^A(s) \leqslant h^A_C(s) \leqslant h^*(s), \tag{14}$$

for all s.
Proof: A solution path in the regression space corresponds to an action sequence executable in the initial world state of the planning problem, so the states along the path must satisfy all problem invariants (see figure 2.3, page 10), and the solution path therefore also exists in the constrained abstract space, as well as in the unconstrained abstraction. □

Relations (11) and (12) hold also for constrained PDB heuristics (provided they are computed using the same set of invariants), as does relation (13) under the same condition for additivity (that no action adds atoms, *i.e.*, changes variables, in both patterns). The argument is essentially the same as in the unconstrained case.

4.3 Pattern Selection

The selection of appropriate patterns is crucial for the quality of PDB heuristics. In general there is no way to determine which of two patterns will yield the more accurate heuristic for a given problem, short of actually computing the respective PDBs and using them to solve the problem. Relation (11) implies that a larger pattern can never result in a less accurate heuristic, but the memory (and time) required for the PDB grows exponentially with the size of the pattern, which limits the size of patterns that can be used in practice. When patterns A and B are additive the sum $h^A + h^B$ (which requires only space equal to the sum of the PDBs for A and B) can be used in place of $h^{A\cup B}$ (which requires space equal to the product of the PDBs for A and B) but the heuristic for the combined pattern can be more accurate than the sum, in particular when constrained abstraction is used (as illustrated by the examples in the preceding section).

This section presents a number of methods for selecting patterns based on state variables, automatically and in a domain-independent way, and an experimental analysis of the quality of the resulting heuristics. All PDB heuristics discussed here are computed with constrained abstraction.

Bin-Packing Pattern Selection

The input to pattern selection is assumed to be the set $P = \{V_1, \ldots, V_n\}$ of state variables and a limit L on the allowed size of any single PDB. The result is a collection

of patterns, $A_1, \ldots, A_k$, which are subsets (not necessarily disjoint) of the set of variables, such that the size of the PDB for each pattern is no more than L. The size of the PDB for pattern A is $\prod_{V \in A} |V|$, where $|V|$ is the size of variable V, *i.e.*, the number of values it can take (+1 if the undefined value is explicitly represented).

Edelkamp (2001) suggests this can be viewed as a bin-packing problem: the variables are items to be packed in bins of limited size, with the objective of using the capacity of each bin as effectively as possible, *i.e.*, minimizing the number of bins needed to pack them all. This makes the collection of patterns a partitioning of the set of state variables, and the final heuristic $\max_{i=1\ldots k} h^{A_i}$. Optimal bin-packing is an NP-hard problem, but there are good and efficient approximation algorithms.

Additive Bin-Packing

The plain bin-packing method, however, fails to take into account additivity. If patterns are additive the final heuristic can be taken as the sum of the values from each of the PDBs instead of only the maximum, resulting in a far more accurate heuristic. In general, there need of course not be any partitioning of variables into patterns (of limited size) such that they are all additive: in this case the best that can be done is to find subsets (again, not necessarily disjoint) of additive patterns and maximizing over the resulting sums.

A way to do this is to find a collection of (not necessarily disjoint) maximal sets of additive variables, such that the collection covers all the variables, and to treat each such set separately. More precisely, find a collection $Q_1, \ldots, Q_l$ of subsets of the set of variables such that

(i) the variables in each Q_i are all mutually additive,

(ii) each Q_i is maximal, *i.e.*, there is no variable $V \notin Q_i$ such that V is additive with all variables in Q_i, and

(iii) $\bigcup_{i=1,\ldots,l} Q_i = P$, *i.e.*, the collection covers all variables.

Next, bin-packing is applied to each set Q_i separately, producing a collection of patterns $A_{i,1}, \ldots, A_{i,k_i}$ for each Q_i. These patterns are additive, so values from the resulting PDBs can be summed and the max of the sums taken as the final heuristic: $\max_{i=1\ldots l} \Sigma_{j=1}^{k_i} h^{A_{i,j}}$.

Finding sets $Q_1, \ldots, Q_l$ satisfying the stated properties is also an NP-hard problem (the *maximum independent set* problem), but again there are good approximation algorithms (see *e.g.* Boppana & Halldorsson, 1992).

In many planning problems large sets of additive variables can be found, *e.g.*, the variables representing the positions of blocks in Blocksworld, or the positions of tiles in the 15-Puzzle. However, there are often also small sets, *e.g.*, in the Blocksworld domain each of the variables `top(?b)` is only additive with `pos(?b)` for the same block `?b`. A small refinement to the procedure above is to merge the sets Q_i that are too small to require division into patterns and apply bin-packing to the union of their variables. This can not result in a worse heuristic, since the values from the

PDBs built for each of the small sets were only maximized in any case.

Weighted Additive Bin-Packing

The size of a PDB for pattern $A = \{V_1, \ldots, V_k\}$ equals the product of the sizes of PDBs for each of the variables V_i, so why group additive variables into patterns at all when the values from PDBs corresponding to each variable can be summed? The reason is that the heuristic for the larger pattern is in many cases more accurate, since it captures negative interactions between actions relevant to each of the variables, in particular when the PDB is computed using constrained abstraction. On the other hand, there are also many cases when the combination of several additive variables into one pattern does not produce better values than the sum of one PDB per variable. The problem is that there is no general domain-independent way of deciding which case one is dealing with.

Consider again the 15-Puzzle instance shown in figure 4.1(a). The variables {`pos(T1)`, ..., `pos(T15)`} are a (maximal) additive set and the sum of the corresponding PDBs, which is also the Manhattan heuristic, is 37. The sum of PDBs

$$h^{\{\texttt{pos(T1)},\texttt{pos(T2)}\}} + h^{\{\texttt{pos(T3)},\texttt{pos(T4)}\}} + \cdots + h^{\{\texttt{pos(T13)},\texttt{pos(T14)}\}} + h^{\{\texttt{pos(T15)}\}},$$

corresponding to one possible partitioning of the variables into patterns of at most two, yields a value of 39. This is because it captures the linear conflict between tiles `T5` and `T6`. The following collection of patterns,

$$h^{\{\texttt{pos(T1)},\ \texttt{pos(T2)}\}} + h^{\{\texttt{pos(T3)},\texttt{pos(T7)}\}} + h^{\{\texttt{pos(T5)},\texttt{pos(T6)}\}} + h^{\{\texttt{pos(T4)},\texttt{pos(T8)}\}} + h^{\{\texttt{pos(T9)},\texttt{pos(T10)}\}} + h^{\{\texttt{pos(T11)},\texttt{pos(T12)}\}} + h^{\{\texttt{pos(T13)},\texttt{pos(T14)}\}} + h^{\{\texttt{pos(T15)}\}},$$

which swaps `pos(T3)` and `pos(T7)` compared to the previous, gives a value of 41, because it captures also the linear conflict between tiles `T7` and `T3`. However, there are also many partitionings into sets of at most two tiles that give only the same value as the Manhattan heuristic (sum of single-variable PDBs).

Examples like this are frequent. Figure 4.5 shows the results of an experimental comparison of the PDB heuristics resulting from several different pattern selection strategies, on problems from the Blocksworld and 15-Puzzle domains. Among the strategies compared are random additive bin-packing selection, which works like the additive bin-packing procedure described above except that variables are placed in patterns at random (subject to the PDB size limit). In the experiment the random selection strategy was applied with 5 different random seeds and the curves labeled "min/max random" in the figure show the best and worst performance, *per problem*, thus obtained. For now, note only that there is a fairly large difference between the best and worst values obtained by different random variable placements.

This shows that there is room for improving the quality of PDB heuristics by a more careful partitioning of additive variable sets. In fact, minimizing the number of PDBs (as done by bin-packing) is a secondary concern: using more and smaller PDBs

consumes less memory, and the time to compute PDBs is in any case dominated by the largest PDB. Thus, we seek to maximize some measure of the "goodness" of the partitioning into patterns instead of blindly minimizing the number of partitions. This problem can be stated as follows: given a set of state variables, Q, find a partitioning of Q into patterns $A_1, \ldots, A_k$ that maximizes $\sum_{i=1,\ldots,k} F(A_i)$, subject to the PDB size constraint (that $\prod_{V \in A_i} |V| \leqslant L$, for each A_i). Again, this is done for each maximal set of additive variables Q, and the final heuristic sums the values from the PDBs constructed from each additive variable set and maximizes over the sums.

Note that the above optimization problem is slightly simplified, in that the value of the target function for the partitioning is assumed to be the sum of its values for each partition. Even so, maximizing an arbitrary function over partitions is also an NP-hard problem (if the target function is a negative constant the standard bin-packing problem results as a special case).

This method depends on having a target function F that accurately measures the "goodness" of patterns, and a general domain-independent such measure is difficult to define. The following function combines two measures of the degree of "interaction" between the variables in a partition:

$$F(A) = \left(\sum_{V,V' \in A} 1 - ISI(V,V') \right) + ICG^2(A) \Big),$$

where $ISI(V,V')$ is a measure of how "close" variables V and V' are in the initial world state and $ICG^2(A)$ measures the amount of inter-dependencies between variables in A. Intuitively, F favours grouping variables that are close in the initial world state and variables with common dependencies. This is of course only a heuristic approximation of pattern goodness. The closeness of two variables V and V' in the initial world state is measured by comparing the number of actions that are executable given the value of V with the number that are executable given the values of both variables:

$$ISI(V,V') = \frac{|\{a \,|\, V = p, V' = p' \text{ consistent with } pre(a)\}|}{|\{a \,|\, V = p \text{ consistent with } pre(a)\}|},$$

where p and p' are the initial values of variables V and V'. The amount of common dependencies between variables in A is given by

$$ICG^2(A) = \sum_{V \in A} \left(\frac{|CG^{-1}(V) \cap A|}{|CG^{-1}(V)|} \right)^2$$

where $V' \in CG^{-1}(V)$ iff some action that changes V' has a precondition or effect on V ($CG^{-1}(V)$ is the set of Vs immediate predecessors in the causal graph, hence the name).

The experimental analysis below shows that PDB heuristics resulting from weighted bin-packing using this function compare quite well against random packing in some

problem domains but not in all (see figure 4.5). The complex, and somewhat *ad hoc*, nature of the target function makes it difficult to characterize precisely the class of planning problems for which it is an accurate measure of pattern goodness.

Incremental Pattern Selection

The weighted additive bin-packing pattern selection method produces PDB heuristics of good quality for certain planning problems, but its behavior is difficult to explain and predict. Also, while the method tries to find the partitioning of a set of additive variables into patterns that will yield the best heuristic it is, like all bin-packing methods, "overly eager" in the sense that additive variables are joined into larger patterns even when no such grouping yields an improvement over building separate PDBs and summing their values, which may result in some waste of memory and computation time.

This section presents a different approach, which constructs patterns incrementally by first computing a PDB for each variable alone and then analyzing the abstract solutions corresponding to the values in the PDBs for possible conflicts and merging the patterns that appear to be most strongly interdependent. This is repeated until no conflicts are found or no more mergers can be made without making the resulting PDB too large. Like additive bin-packing the process is applied to each maximal set of additive variables separately to ensure that the collection of patterns is additive. The final heuristic sums the values from the PDBs constructed from each additive variable set and maximizes over the sums.

Recall that a regression PDB is computed by an exhaustive breadth-first exploration forwards from the initial world state. During this exploration a pointer can be stored with every abstract state, indicating which action was applied in which (abstract) state to reach this state and using these pointers the abstract plan corresponding to an entry in the PDB can be quickly extracted (this is the standard way of extracting the shortest path in any single-source shortest path algorithm). For a given pattern A, consisting of a set of variables, potential conflicts are found by extracting the abstract plan for the goal of the planning problem (projected onto A) from the corresponding PDB and simulating this plan forwards from the initial world state, using the original (unabstracted) actions. When a precondition of an action in the abstract plan is found to be inconsistent with the value of a variable V not in the pattern, either because the precondition is on the value of V and contradicts the current value or because the precondition and the value of V together violate an invariant, a conflict between pattern A and the pattern that V is part of is noted.

For each (maximal) set of additive variables, $\{V_1, \ldots, V_n\}$, the process starts with a corresponding collection of single-variable patterns, $A_1, \ldots, A_n$, and computes for each pattern a PDB and its conflicts with the other patterns. Each conflict is assigned a weight, given by the h^1 value of the inconsistent atom set. A pair of patterns A_i and A_j is a feasible merge iff the size of the PDB for $A_i \cup A_j$ is less than the given limit L. The feasible pair of patterns with the heaviest conflict are merged and a

new PDB for the joint pattern is computed, along with its conflicts with the other patterns. This repeats until there are no more conflicts or no feasible pair to merge. Values from the PDBs built from an additive variable set are added, and the final heuristic value is the maximum over the sums.

As an example, here is how the procedure is applied to the Blocksworld problem of swapping the two blocks at the base of a tower, described in section 4.2 above. The only large set of additive variables in this problem is {pos(b_1),...,pos(b_n)}, of which only pos(b_n) has a goal value, (on b_n b_{n-1}). The abstract plan extracted from its PDB consists of the single action (move-from-table b_n b_{n-1}). Simulating this action in the initial world state reveals that its preconditions conflict with variables pos(b_{n-1}) (whose value in the initial state is inconsistent with the precondition (clear b_n)) and pos(b_{n-2}) (whose value is inconsistent with the precondition (clear b_{n-1})). The first of these conflicts has a slightly higher weight, so pos(b_n) and pos(b_{n-1}) are merged into the pattern {pos(b_n),pos(b_{n-1})}, and a new PDB is computed. This pattern still has a goal value, with an associated abstract plan consisting of two steps: (move-to-table b_{n-1} b_n) and (move-from-table b_n b_{n-1}). This plan also conflicts with variable pos(b_{n-2}), so the pattern is extended to {pos(b_n),pos(b_{n-1}), pos(b_{n-2})}. The abstract plan extracted from this PDB will conflict with pos(b_{n-3}), so the merging process continues until the pattern has grown so large that no more mergers are feasible.

Analysis: Quality of PDB Heuristics Resulting from Different Pattern Selection Strategies

This section presents the results of an experimental comparison of the quality of PDB heuristics resulting from the pattern selection methods described above. The quality of a heuristic is measured by the number of nodes expanded in an A* search, using sequential regression (runtimes are quite closely related to this measure as well, since the time spent computing PDBs does not differ much between the different selection methods). The experimental results show that the "intelligent" pattern selection methods (weighted additive bin-packing and additive incremental selection) result in better heuristics than randomly selected patterns, even when additivity is taken into account, but also that there are weaknesses in both methods.

The Experiment

The pattern selection methods compared are a randomized version of the additive bin-packing method, weighted additive bin-packing using the goodness function F described above, and incremental pattern selection. All PDBs were computed using constrained abstraction, with the full set of invariants in the planning problem. The limit (L) on the size of any single PDB was set to 2 million entries, except in the random STRIPS domain where a limit of 20,000 entries was used.

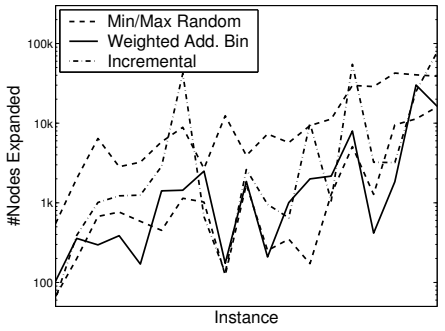
Randomized additive bin-packing selection works like the additive bin-packing proce-

dure described above except that variables are placed in patterns at random instead of arbitrarily (but still in the smallest number of patterns and respecting the PDB size limit). The random selection strategy is applied with 5 different random seeds: the curves labeled "min/max random" in figure 4.5 indicate the best and worst performance, *per problem*, over the 5 random trials. In other words, the performance of the "best of 5 random" is only available at five times the cost, in memory and computation time, compared to the other PDB heuristics.

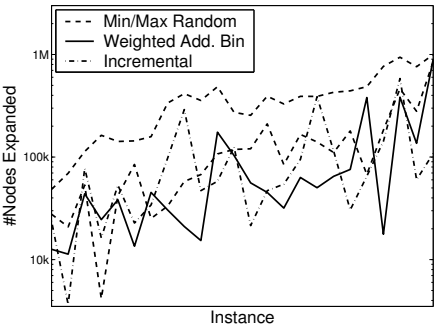
The planning problems used in the experiment are from three different domains: the Blocksworld and 15-Puzzle domains, described earlier in this chapter, and random (propositional) STRIPS problems. The random STRIPS problems differ significantly from the Blocksworld and 15-Puzzle problems in that they lack structure: there are very few invariants, and consequently the state variable representation of the problem is closer to the propositional representation. Typically, variables are binary with values corresponding to a proposition and its negation[3].

The Blocksworld problems are the same as in the experiment described in chapter 3 (page 46) but due to the volume of the experiment only half the problems (the smaller half) were used. The 15-Puzzle problems are the easiest quarter of the collection of random 15-Puzzle problems presented by Korf (1985). The random STRIPS problems were generated according to Bylander's variable model (1996), in which the preconditions and effects of each action are chosen independently, with equal probability for each proposition, as are the goal propositions. The parameters of the model (and values used in this experiment, respectively) are the number of propositions ($n = 60$), the number of actions ($o = 118$), the average number of preconditions and effects per action ($r = s = 3$) and the number of goals ($g = 6$). The hardness of deciding plan existence for random STRIPS problems depends mainly on the number of actions in relation to the number of atoms and goals, peaking around $o = O(n \ln g)$ (see Bylander, 1996). The chosen parameter values are in the "hard region" but even so a significant fraction of the generated problems are either trivial or trivially proved unsolvable. The problems used in the experiment were selected as follows: 200 problems were generated, an IDA* search with the h^2 heuristic run for each, and problems that were proved unsolvable (problems for which $h^2(G) = \infty$) or that were solved very quickly were removed. This left 55 problems. Because, however, the PDB heuristics generally work rather poorly for random STRIPS problem, the set was further reduced by excluding problems not solved within 4 CPU hours (by IDA* with the h^2 heuristic), leaving a final of 42 problems. Again, this was done to keep the volume of the experiment manageable (recall that each problem is solved seven times).

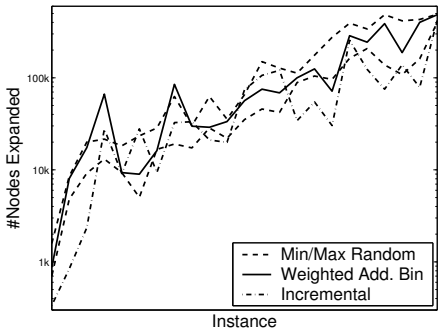
[3]Although the propositional STRIPS planning model, as defined in chapter 2, does not include negation the negation of an atom `p` can be represented by an atom `not-p`, if action and initial world state descriptions are written accordingly.



(a) Blocksworld



(b) 15-Puzzle



(c) Random STRIPS

Figure 4.5: Number of nodes expanded in A* search using heuristics resulting from random additive bin-packing, weighted additive bin-packing and incremental pattern selection. The "min" and "max" random show the best and worst performance, per problem, over 5 random trials. Instances are sorted by the median value over the 5 random additive bin-packing heuristics.

Results and Conclusions

Figure 4.5 shows the results. A first observation that can be made is that across all the domains there is a significant gap in performance between the best and worst PDBs obtained by random additive bin-packing. This demonstrates that considering only additivity is not enough: heuristic quality improves when additive variables are combined into larger patterns, but is also very sensitive to the exact partitioning.

Weighted additive bin-packing produces PDBs that compare quite well with the best of 5 random bin-packing selections in the Blocksworld and 15-Puzzle domains, but only around the median in the random STRIPS domain. This suggests that the target function is not quite domain-independent, *i.e.*, that it is only accurate for planning problems of a certain kind (in fact, it is not unlikely that the target function may be somewhat "overfitted" to the Blocksworld and 15-Puzzle domains specifically).

The results of incremental pattern selection, compared to the best random bin-packing selection, are more uneven within each domain, but are on the other hand more even across the three domains. This suggests that some of the choices that are currently made arbitrarily, *e.g.*, which of the possibly many different abstract plans is analyzed for conflicts or which pair to merge in case of ties, are significant. Investigating this, through further experiments with random variations, is an interesting topic for future research, since if this is the case it may be possible to refine the method by making these choices more informed.

It is also worth noting that neither weighted additive bin-packing nor incremental pattern selection produces PDBs that dominate those obtained by the "best of 5 random" bin-packing in any of the domains.

As noted above the performance of the PDB heuristics, regardless of pattern selection, is generally quite poor on the random STRIPS problems (the PDB and h^m heuristics, and an additive version of the h^m heuristics, are compared in chapter 6). In part this is probably due to the lack of structure in these problems, which results in a state variable representation with a large number of "small" variables. Another factor that may also be important, however, is that these problems typically have many large and overlapping sets of additive variables, whereas in problems of the Blocksworld and 15-Puzzle domains there is only one large set of additive variables. The bin-packing and incremental pattern selection methods both work on each set of additive variables separately (to ensure the additivity of the resulting PDBs) which in the random STRIPS problems results in a large number PDBs (this is why a smaller PDB size limit was used for the problems in this domain: a greater size limit would not significantly reduce the number of PDBs, but only increase the time required to compute them). Selecting patterns across additive sets may yield better PDB heuristics for problems in which this phenomenon occurs.

4.4 Discussion

Pattern database heuristics (introduced by Culberson & Schaeffer, 1996) have attracted a great amount of interest recently and have been used to solve a number of hard "puzzle" search problems, *e.g.*, the 15-, 24- and even 35-Puzzle, Rubik's Cube and others (see *e.g.* Hernadvölgyi & Holte, 2000; or Felner, Korf, & Hanan, 2004) and also, *e.g.*, sequence alignment problems. For many of the problems PDBs are the only effective admissible heuristics known. The first application of PDB heuristics to domain-independent (STRIPS) planning was by Edelkamp (2001).

Several improvements to the basic PDB method have been suggested, most concerning ways to reduce the amount of memory required to store the PDB (thus allowing larger patterns to be used). When applied to a specific problem, properties of that problem are of course exploited to improve performance or reduce the memory requirement. For example, symmetries in the abstract state space can be used to look up multiple values in the same PDB (Culberson & Schaeffer 1998; Felner *et al.* 2005), or a problem specific heuristic can be used and only improvements over the base heuristic stored in the PDB (Korf & Taylor 1996). PDB compression (Felner *et al.* 2004) is a general technique, in which the size of a PDB is reduced after it has been computed by "abstracting out" one or more pattern variables. This differs from not including the variables in the pattern in the first place in that interactions caused by those variables are taken into account, so heuristic values over the variables that remain after compression tend to be higher, although lower than in the uncompressed PDB (under certain conditions, the compression loss can be bounded, or reduced to nothing by the use of some additional memory). Another general memory reduction technique is the use of so called symbolic representations (*e.g.* ordered binary decision diagrams, OBDDs) to store the heuristic function instead of a plain lookup table (Edelkamp 2002).

For each of the search problems that have been solved with PDB heuristics different selections of patterns have been tried (see *e.g* Felner, Korf, & Hanan, 2004), but concerning pattern selection in general relatively little is known. Holte *et al.* (2004) found that the combination of several smaller PDBs (by maximum or sum, as possible) tends to yield better heuristics than one or a few larger PDBs, over a range of different search problems. Edelkamp (2001), in applying PDBs to STRIPS planning, discusses additivity (stating the condition "no action adds atoms in both patterns" described above and proving that it is sufficient to ensure admissibility) but does not describe how to exploit it in pattern selection, stating this only as a bin-packing problem. As the experimental analysis above demonstrates, plain bin-packing is unlikely to produce high quality heuristics, even if it is applied to sets of additive variables to produce additive PDBs.

Felner, Korf & Hanan (2004) describe dynamic additive PDBs (for the $(n^2 - 1)$-Puzzle) which are based on a different form of pattern selection: instead of partitioning the additive set of variables representing positions of tiles statically, a PDB is created for *every* pair of tiles and a subset of PDBs whose values are added is selected dynamically at every heuristic evaluation, in such a way that the heuristic value is

maximized. (The reason why this is done for pairs of tiles is that pairs correspond to edges in a graph with tiles as nodes: thus the problem of finding the maximizing partitioning corresponds to a weighted matching problem, which is solvable in polynomial time; see Papadimitriou & Steiglitz, 1982).

Pattern Database and h^m Heuristics: A Comparison

As briefly mentioned in chapter 3, the h^m relaxation can be understood as a change of search space, in which every state s with $|s| > m$ is a "max state", whose successors are the size m subsets of s and whose cost is the max of the successor costs, while states s with $|s| \leqslant m$ ("min states") are regressed as normal. Thus, there is a similarity with PDB heuristics in that h^m "abstracts" states with more than m atoms to states of size m (in all possible ways) but more importantly a difference in that the h^m relaxation *recursively* selects the most maximum cost "abstraction" (size m subset) while the abstraction induced by a pattern selects the same subset of variables in every state.

This makes the h^m heuristic stronger, in some cases. Recall the Blocksworld problem of swapping the two blocks at the base of a tower of n blocks (described in section 4.2 above). For the goal of this problem, even h^1 yields the optimal cost estimate of n, because it is able to "shift focus", and consider all blocks: to move $\mathtt{b}_n$ onto $\mathtt{b}_{n-1}$, $\mathtt{b}_{n-1}$ must be clear; to clear $\mathtt{b}_{n-1}$ $\mathtt{b}_{n-2}$ must be moved, so it must also be clear; and so on, while any PDB heuristic based on the $\mathtt{pos}(\mathtt{b}_i)$ variables can consider only a fixed subset of blocks (this applies to the dynamic additive PDBs of Felner, Korf & Hanan (2004) as well).

However, there are also advantages to using a fixed abstraction, like that induced by a pattern. Computing a complete h^m solution is only feasible for small values of m (typically, $m \leqslant 2$ or $m \leqslant 3$), while the patterns used in PDB heuristics are generally much larger and therefore may be able to capture more interactions. More important is that certain properties of the abstraction, like additivity, can be determined. Recall the Blocksworld problem of "tower construction", *i.e.*, building an ordered tower out of n blocks initially on the table: an additive PDB based on the `pos(?b)` variables yields for the goal of this problem the optimal value $n - 1$, while the h^m heuristic yields at most m (which is typically much smaller). This illustrates how additivity allows a collection of relatively simple heuristics to be combined into a very powerful heuristic.

Chapter 6 describes how h^m heuristics can be made additive, without loss of admissibility. An experimental comparison of the h^m, additive h^m and PDB heuristics is also presented there.

5. Planning with Resources

Resources are an important part of many planning problems, particularly planning problems involving time as they are related to scheduling problems (it is probably fair to say that scheduling would not be a problem if not for resource limitations). Resource is a very wide concept. Smith & Becker (1997) in their scheduling ontology define a resource as "an entity that supports or enables the execution of activities". The "entity" that is a resource can be material, *e.g.*, a tool needed or raw materials consumed, but it can also be logical, *e.g.*, a space being empty or a computer being powered on and idle.

Certain kinds of resources can be represented in the STRIPS planning model: in fact, the planning problems described in chapters 2 and 4 exhibit several examples of things that can be argued to be resources, *e.g.*, the UAVs in the UAV planning domain, the empty square in the 15-Puzzle or the "clearness" of blocks in the Blocksworld domain. This chapter deals with two classes of resources that can not be expressed in the STRIPS model (or the temporal STRIPS model), namely resources whose capacity is a numeric value. The two classes of resources are *reusable* resources (section 5.2), whose capacity limit what actions can be executed concurrently (and thus make sense only in temporal planning), and *consumable* resources (section 5.3), whose capacity limit what actions can be in the plan overall. The sequential and temporal STRIPS planning models are extended and the regression search space and heuristics are adapted to the extended model in each case (although in the case of consumable resources only under certain restrictions on the model which are necessary to ensure that the problem is decidable). The chapter begins, however, with a brief discussion of the different kinds of resources that appear in planning and scheduling problems.

5.1 Discussion: Resources in Planning and Scheduling

Resource is a wide concept, with somewhat fuzzy borders. Most planning and scheduling problems contain elements that can be argued to be resources. In this section we distinguish, and name, a few different classes of resources, following loosely the ontologies by Smith & Becker (1997) and Laborie (1995; 2001). Fadel *et al.* (1994) remark that "being a resource is not an innate property of an object, but is a property that is derived from the role an object plays with respect to an activity". Thus, sometimes the same entity can appear as a different kind of resource in relation to different actions, even at times not being considered a resource at all.

A first distinction made by Smith & Becker is between capacitated and discrete-state

resources. A discrete-state resource is a condition, or "resources whose availability is more a qualitative function of state" (Smith & Becker, 1997, page 5). This corresponds to a condition on the world state and is thus something that is expressible in, and indeed quite central to, the STRIPS planning model. A typical example of a discrete-state resource is the above mentioned "clearness" of a block in the Blocksworld domain, which is required by, for example, actions that move the block. A capacitated resource is characterized by the *amount* of the resource that is available. This corresponds perhaps more to the intuitive notion of a resource: typical examples include money, fuel, raw materials or parts, space, bandwidth, vehicles, tools, machines or workers *etc.* Characteristic of capacitated resources is that when part (or all) of the resource capacity is allocated to an action, the allocated capacity can not be used by another action, *i.e.*, use of the resource is *exclusive* (though there are exceptions to this principle, *e.g.*, batch capacity resources). Capacitated resources can be divided into reusable and consumable: with a reusable resource, the capacity allocated to an action becomes available again when the action ends, while with a consumable resource, the allocation of capacity is permanent, *i.e.*, the resource is consumed. A resource is also called renewable if there are actions that increase the amount available of the resource (though the term "renewable" is sometimes used for what is here called reusable resources, *e.g.*, by Laborie, 1995, and by Kolisch & Sprecher, 1996). By consumable resources we will generally mean resources that may or may not be renewable, and use the term "(monotonically) decreasing" to indicate that a consumable resource is not renewable.

Both reusable and consumable capacitated resources can also be classified according to the requirements that actions have on them. Laborie (1995) distinguishes between discrete resources, which are used only in "whole units", and continuous resources, of which actions can require or consume arbitrary amounts. Examples of discrete resources are transport vehicles, machines or tools (reusable) or parts (consumable) in a manufacturing problem, while fuel (consumable) or storage space (reusable) are examples of continuous resources. An important special case of discrete resources are resources with unit capacity, *i.e.*, resources that can only be used once (if consumable) or by one action at a time (if reusable). Smith & Becker (1997) do not distinguish between discrete and continuous resources but only between unit capacity and aggregate (multi-capacity) resources. Aggregate resources are further divided into simple (or homogeneous) resources, whose units are indistinguishable, and structured (or heterogeneous; called "individualized" by Laborie, 1995), whose units have individual state and/or attributes. An example of structured resources is transport vehicles, which can differ by being at different locations (thus having individual state) and by having different load carrying capacity (thus having individual attributes). Shin & Davis (2005) distinguish a class they call interval resources, which are continuous but also individualized in the sense that capacity is allocated in "segments" that can not overlap. An example of a reusable interval resource given by Shin & Davis (2005) is heap space (once a segment of memory is allocated, it can not be moved), while an example of a consumable interval resource is a piece of material, *e.g.*, wood or cardboard, out of which shapes are cut (obviously the pieces can not overlap).

Resources that have (individual) state can require a setup activity to be done before the resource is used by an action. The duration of the setup activity can be either a constant, or dependent on the current state of the resource and the state that the action requires the resource to be in (in the second case, the setup time is often called "sequence dependent", since the current state of the resource depends on what action used the resource most recently, and thus on how actions using the resource are sequenced). An example of resources with sequence dependent setup times, mentioned in chapter 2 (page 24) is the UAVs in the UAV mission planning domain.

Planning with Resources

As noted above, discrete-state resources correspond to conditions on world states, and are therefore naturally expressible in the STRIPS planning model. A few special cases of capacitated resources can also be represented in the STRIPS, or temporal STRIPS, model (some of them are described in more detail in the following sections). Most problems involving planning with capacitated resources, however, require some extension to the planning model. A number of AI planning systems have been extended to address problems involving resources of some kind but, as with planning problems involving time (discussed in section 2.3, page 24), there is some variation in the planning models assumed by different systems.

In most planning systems that deal with capacitated (consumable) resources, the resources play only the role of additional constraints on executable plans, and many systems, *e.g.*, RIPP (Koehler 1998), the LPSAT and TM-LPSAT planners (Wolfman & Weld 1999; Shin & Davis 2005), also require that action resource consumption is a linear function of action parameters. Some planning systems, *e.g.*, Pyrrhus (Williamson & Hanls 1994), ASPEN, (Fukunaga *et al.* 1997; Chien *et al.* 2000), Sapa (Do & Kambhampati 2001), Excalibur (Nareyek 2001) and MO-GRT (Refanidis & Vlahavas 2003), are able to include resource consumption in the optimization objective (though none of them offer an optimality guarantee). Version 2.1 of PDDL (Fox & Long 2003) introduced numeric state variables (allowing a fairly unrestricted use of them) which can be used to model consumable resources as well as more general metrics. In combination with the extensions for temporal planning also introduced in PDDL 2.1 it is also possible to model reusable resources (although with some restrictions, due to certain peculiarities of the PDDL 2.1 semantics; see Hoffmann *et al.*, 2004). In the area of scheduling, efficient algorithms have been developed for treating several special classes of resources (see *e.g.* Laborie, 2001), and some planning systems, *e.g.* O-Plan (Tate, Drabble, & Kirby 1994) and RealPlan (Srivastava & Kambhampati 1999), incorporate such specialized methods for dealing with resource constraints.

5.2 Planning with Reusable Resources

A *reusable resource* is a resource whose capacity constrains which, or how many, actions can take place concurrently, but does not impose any limit on actions executing in sequence. Thus, reusable resources are interesting only in temporal planning. From the perspective of actions, an action requiring a reusable resource "borrows" some (or all) of that resources capacity during its execution, returning the full amount when the action ends.

A special case of reusable resources, *viz.* discrete resources with unit capacity, is expressible in the temporal STRIPS model: an action that has as a precondition an atom p and that locks (temporarily deletes) p can be said to "require" p (exclusively), and hence p can be thought of as a unit capacity discrete reusable resource. In general, a resource can have any capacity and actions can require arbitrary amounts, something that can not be expressed in the temporal STRIPS model. In fact, even the only slightly more general case in which all actions requiring the resource have a requirement of 1 and the capacity of the resource is an integer (*i.e.*, the resource permits at most $k > 1$ actions to execute concurrently) can not be expressed in the basic model.

Extensions to the Planning Model

In the temporal STRIPS planning model extended with reusable resources, each resource r has a capacity ($cap(r)$) and each action a has for each resource r a requirement ($req(a, r) \geqslant 0$), equal to zero if the action does not require the resource. As with durations, we assume that resource capacities and requirements are rational numbers (though integers or reals could also be used).

Recall from chapter 2 (page 12) that a schedule is a collection of action instances with specified start times, $S = \{(t_1, a_1), \ldots, (t_n, a_n)\}$, and that the schedule is executable iff each action a_i is executable in the world state that results at t_i when the schedule is executed. In the presence of reusable resources, the condition for executability of a schedule S is amended to require that for every resource r and each time point t,

$$\left(\sum_{\{(t_i,a_i)\in S_t\}} req(a_i, r) \right) \leqslant cap(r)$$

where $S_t = \{(t_i, a_i) \in S \,|\, t_i < t < t_i + dur(a_i)\}$ is the set of actions *executing at* t, *i.e.*, that the sum of requirements of the set of actions executing concurrently at any time does not exceed the capacity, for any resource. Note that an actions resource requirements, like the effects of the action, are restricted to the interior of the execution interval.

As a consequence, a set of actions is compatible only if the sum of their requirements does not exceed the capacity for any resource (in addition to the pairwise condition for compatibility stated in chapter 2, page 13). Thus, in the presence of reusable

resources, pairwise compatibility is a necessary, but no longer sufficient, condition for a set of actions to be compatible.

Regression Planning with Reusable Resources

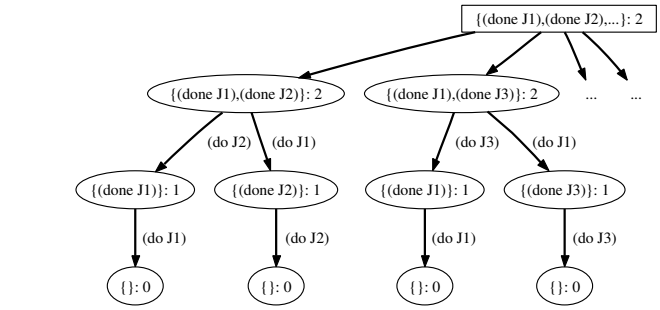
Apart from the above modification to the condition for compatibility of a set of actions, no further changes are required to the temporal regression method to plan with reusable resources. Nor are any changes to the definition of the h^m heuristic required, though the heuristic may become less accurate due to the limited way in which resources are taken into consideration.

Recall from chapter 2 (page 14) that a temporal regression search state is a pair $s = (E, F)$, where E is a set of atoms (subgoals to be achieved) and F is a set of actions scheduled relative to the goals in E. Expanding a state involves choosing (non-deterministically) an establishing action for each atom in E, such that the set consisting of the chosen actions and any actions in F is compatible (see figure 2.6(b), page 18). The set of actions scheduled concurrently at any time point in a plan found by temporal regression corresponds to the set of actions chosen in the construction of a successor state together with the set of scheduled actions (actions in F) in its predecessor state at some point along the search path. Therefore, ensuring compatibility in the construction of successor states in temporal regression is sufficient to ensure that the schedule corresponding to a solution path respects the capacity constraints of reusable resources.

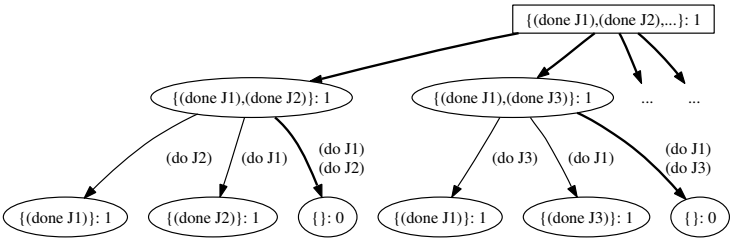
h^m Heuristics for Planning with Reusable Resources

The h^m relaxation (equation (5), page 39) and the additional relaxations applied to the temporal heuristic (inequalities (8) and (9), page 51) remain admissible in the presence of reusable resource constraints. In the definition of the temporal h^m heuristic, states with m or fewer atoms and no scheduled actions are regressed in the same way as in the temporal regression search (case $F = \emptyset$, $|E| \leqslant m$ of equation (10), page 52). In particular, the same condition for compatibility of the set of actions applies. Thus, the heuristic takes some account of resource limitations.

However, the heuristic for reasonable values of m can become much weaker. Since the h^m heuristic regresses states with at most m atoms, at most m concurrent actions are considered, but in the presence of reusable resources compatibility of each size m subset of a set of actions does not necessarily imply compatibility of the whole action set. As an illustrative example, consider a very simple scheduling problem involving n jobs $\mathtt{J}_1, \ldots, \mathtt{J}_n$ without ordering constraints, each requiring 1 unit of a reusable resource `R` (an example of a much more general scheduling problem, the multi-mode resource-constrained project scheduling problem, is given in the next section). The problem can be stated as a temporal STRIPS planning problem with atoms `(done ?j)`, for each job `?j`, and actions `(do ?j)` that have no preconditions, add `(done ?j)`, and require 1 unit of `R` ($req(\texttt{(do ?j)}, \texttt{R}) = 1$). For simplicity, assume that the



(a) $cap(\texttt{R}) = 1$



(b) $cap(\texttt{R}) = 2$

Figure 5.1: Illustration of how the resource capacity influences the calculation of $h^2(\{\texttt{(done J1)}, \texttt{(done J2)}, \ldots\})$ in the example scheduling problem (described on page 83). In figure (a) the capacity of the resource is 1 ($< m = 2$) and the h^2 value is 2, while in figure (b) the capacity of the resource is 2 ($\geqslant m = 2$) and the h^2 value is 1.

duration of each action `(do ?j)` is 1. The atoms `(done ?j)` are all goals (since all jobs must be done) and are all false initially (since no job has been done then). If the capacity of the resource allows all actions to be executed concurrently ($cap(\texttt{R}) \geqslant n$) the optimal makespan is 1. If the capacity of the resource is less the optimal makespan is $\frac{n}{cap(\texttt{R})}$ (rounded up), since the n actions have to be executed in groups of $cap(\texttt{R})$ at a time. The h^m heuristic, however, because it considers only sets of m concurrent actions yields an estimate of $\frac{m}{cap(\texttt{R})}$ (rounded up), which is 1 whenever $cap(\texttt{R}) \geqslant m$ (since then any set of m actions can be executed concurrently). Figures 5.1(a) and (b) illustrate the calculation of the h^2 value when the capacity of the resource is 1 and 2, respectively. As the cost of computing a complete h^m solution increases exponentially with m, the discrepancy between optimal and estimated cost can grow arbitrarily large as the problem size (n) grows. Chapter 7 presents two methods for improving the h^2 heuristic (by computing only partial h^m solutions for higher m) which are applicable to planning with reusable resources as well.

5.3 Planning with Consumable Resources

A *consumable resource* is a resource whose capacity constrains which, or how many, actions can be executed at all, concurrently or in sequence. A certain amount of the resource is initially available, and each action either reduces this amount (*consumes* the resource) or increases it (*produces* the resource). The resource has a minimum and a maximum capacity constraint and the amount available can never be reduced below the minimum or increased above the maximum. Concurrent consumption and/or production by several actions is assumed to be possible as long as capacity constraints are guaranteed to be respected.

Discrete consumable resources (resources that are consumed and produced only in whole units) with bounded minimum and maximum capacities can be represented in the basic STRIPS model, by treating the resource amount available as a proposition. For example, a resource r with minimum capacity $cap_{\min}(r)$ and maximum capacity $cap_{\max}(r)$, can be represented with a proposition `(avail r i)` for each $cap_{\min}(r) \leqslant i \leqslant cap_{\max}(r)$, where `(avail r i)` holds in a world state iff i units of the resource are available in that state. Each action consuming or producing the resource exists in a number of "copies" such that each is executable at a specific resource availability level, specified as a precondition of the action, and changes the state accordingly (multiple copies of actions can be avoided if conditional effects are used instead). Aside from the obvious limitations (and impracticality) of this way of describing consumable resources, it also does not work in temporal planning, where the effect of several actions concurrently consuming or producing the resource is cumulative. Thus, there are several incentives to extend the model with a more general treatment of consumable resources.

Extensions to the Planning Model

Consumable resources are useful in both sequential and temporal planning, and the required extensions to the corresponding STRIPS models are also very similar. Here, we introduce first the extension to the STRIPS model of sequential planning and then its adaptation to the temporal planning case.

Consumable Resources in Sequential Planning

In the STRIPS planning model extended with consumable resources, each resource r has a minimum and a maximum capacity ($cap_{\min}(r)$ and $cap_{\max}(r)$, respectively) and an amount available in the initial world state ($init(r)$). We assume that planning problems are well posed, in the sense that the initial amount available is within the capacity limits for all resources. Note that the minimum and maximum capacities can be infinite. Each action a has for each resource r an effect on that resource ($chg(a, r)$), which is positive if the action produces the resource, negative if the resource is consumed by the action, and zero if the action has no effect on the resource. Again, we assume that the resource capacities and effects are rational numbers.

Because the amount available of each consumable resource varies over the course of execution of a plan, it is a part of the world state. Thus, a world state w in the extended model consists of a complete assignment of truth values to atoms and an assignment of rational values to each of the resources in the planning problem (the amount available of resource r in state w will be denoted $w(r)$ and also referred to as the *level* of the resource in state w). An action a when executed changes the amount available of each resource r by $chg(a, r)$ (in addition to changing the truth values of atoms as described in chapter 2). The condition for the executability of an action in a world state is amended to require that the resource levels resulting from execution of the action do not violate the minimum or maximum capacity of any resource. Formally, action a when executed in world state w leads to a world state w' such that

$$w'(r) = w(r) + chg(a, r)$$

for every consumable resource r, and the action is executable in w only if

$$cap_{\min}(r) \leqslant w'(r) \leqslant cap_{\max}(r)$$

for every consumable resource r. (This can of course be translated into an equivalent condition on w.)

If a resource r is either consumed or not changed by every action ($chg(a, r) \leqslant 0$ for all a) the resource is said to be *decreasing*. Conversely, if a resource r is either produced or unchanged by every action ($chg(a, r) \geqslant 0$ for all a) the resource is said to be *increasing*. A resource that is either increasing or decreasing is *monotonic*. Note that an increasing resource can be converted into an equivalent decreasing resource

(and vice versa) by multiplying all effects on the resource by -1 and modifying capacities and initial level accordingly.

Consumable Resources in Temporal Planning

The treatment of consumable resources in temporal planning is very similar to the case of sequential planning, due to the assumption that the effect of concurrent consumption and/or production of a resource by several actions is simply cumulative. The same conservative semantics of actions are assumed for resource effects as for the propositional effects, *i.e.*, the change in resource level caused by an action takes place at some, unspecified, point in the interior of the actions interval of execution.

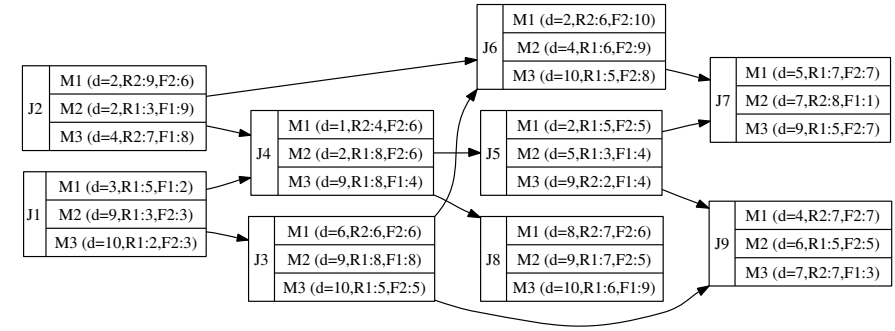
Thus, the condition for executability of a schedule must ensure that the capacity constraints of all resources are respected no matter what order the resource effects of actions executing concurrently take place in. For a schedule $S = \{(t_1, a_1), \ldots, (t_n, a_n)\}$ and a time point t, the set of actions *completed by* t is $S_t^< = \{(t_i, a_i) \in S \,|\, t_i + dur(a_i) \leqslant t\}$ and the set of actions *executing at* t is $S_t = \{(t_i, a_i) \in S \,|\, t_i < t < t_i + dur(a_i)\}$. The possible levels of a resource r at time point t are given by the initial level of r plus the total effects on r of all actions completed by t plus the total effects on r of any subset of the actions executing at t. Thus,

$$cap_{\min}(r) \leqslant init(r) + \left(\sum_{(t_i,a_i) \in S_t^<} chg(a_i, r) \right) + \left(\sum_{(t_i,a_i) \in S'} chg(a_i, r) \right) \leqslant cap_{\max}(r)$$

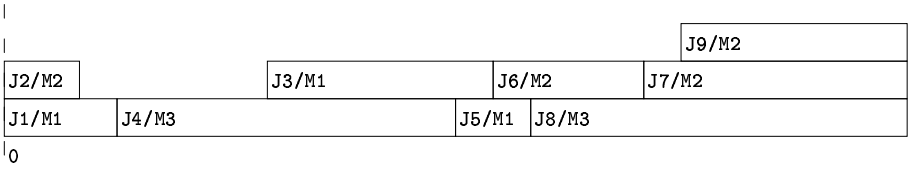
must hold for every resource r, time point t and $S' \subseteq S_t$ to ensure the executability of the schedule. This condition is fairly complex, and potentially computationally expensive to verify. In the special case when all resources are monotonic, however, it suffices to verify a much simpler condition: because all action effects on a resource in this case change the level of the resource in the same direction, and thus towards the same capacity limit, if the sum of all effects that can possibly have taken place at some time t (all actions in $S_t^<$ and S_t) does not violate the capacity constraint then no subset of those effects can violate the constraint either.

Example: Multi-Mode Resource-Constrained Project Scheduling

Resource-Constrained Project Scheduling (RCPS) is a general abstract scheduling problem that has been extensively studied in operations research, where both optimal and heuristic solution methods have been developed (see *e.g.* Kolisch & Sprecher, 1996, for an introduction to the problem, and Kolisch & Hartmann, 2005, for an overview and analysis of current solution methods). In the RCPS problem, the task is to schedule a set of n jobs, each with a specified duration, in a way that is consistent with a set of precedence constraints (in the form of a partial order on the execution of the jobs) and resource constraints. Resources can include both reusable resources, with arbitrary capacity, and decreasing consumable resources, and each job can have arbitrary resource requirements (for example, some jobs may require and/or consume



(a)



(b)

Figure 5.2: (a) Example MRCPS problem. Each node corresponds to a job, with the available modes of execution listed and duration and resource requirements indicated for each mode (`R1` and `R2` are reusable resources, while `F1` and `F2` are consumable). Arcs represent precedence constraints. (b) A possible mode selection and schedule with optimal makespan.

only some resources). The objective is typically to minimize makespan, but a variety of other optimization criteria can also be defined (*e.g.* given a maximal permitted makespan, or *deadline*, minimize the capacities of reusable resources needed to meet it).

The multi-mode version of the problem (MRCPS) offers an additional choice between different *modes* for the execution of each job, where each mode can have a different duration and different resource requirements and consumption (precedence constraints are the same for all modes, however). A small example problem is shown in figure 5.2(a). Typically, modes represent possible trade-offs between time and resource use, or the use of different resources. For example, modes `M1` and `M2` of job `J2` in the example problem in figure 5.2(a) have the same duration, but when executed in mode `M1` the job requires resource `R2` and consumes resource `F2` and when executed in mode `M1` the job requires resource `R1` and consumes resource `F1`. The duration of the job when executed in mode `M3` is twice what it is in the other two modes, but executing the job in this mode uses less resources than both the other modes.

The task in the MRCPS problem is for each job to select a mode and to schedule the job so as to minimize makespan, subject to precedence and resource constraints as in the single-mode case. The choice of modes introduces a planning aspect into the problem, although the structure of the planning problem is relatively simple. Still, choices of modes for different jobs interact through their resource requirements, which makes finding optimal schedules hard. Figure 5.2(b) shows an example of a schedule with optimal makespan for the problem shown in figure 5.2(a)

The MRCPS problem, under the makespan minimization criterion, is straightforwardly expressed in the temporal STRIPS model with reusable and consumable resources. An atom `(done ?j)`, for each job `?j`, represents the fact that the job has been done. For each job `?j` and mode `?m` that this job can be executed in there is an action `(do ?j ?m)` which adds `(done ?j)`, has as preconditions `(done ?i)` for each job `?i` constrained to precede `?j` and has the resource requirements and effects specified for the mode `?m`.

Decidability

An analysis of the decidability of the plan existence problem for a variety of sequential planning models involving the combination of propositional and numerical state variables is carried out by Helmert (2002). In the absence of further restrictions the STRIPS planning model with consumable resources, as defined here, corresponds to the class $(\mathcal{C}_\emptyset, \mathcal{C}_c, \mathcal{E}_{\pm c})$ in Helmert's taxonomy, meaning planning problems in which actions can compare the value of a numeric variable to a constant and modify variables by addition or subtraction of a constant, but in which there are no goal conditions on the numeric variables (a certain transformation is necessary since in our model an action can not have a precondition on the level of a resource without changing it). This class generalizes the class $(\mathcal{C}_\emptyset, \mathcal{C}_0, \mathcal{E}_{\pm 1})$, which is shown to be undecidable.

For (provably) optimal plan generation to be possible the planning model has to

be restricted so that the plan existence problem is decidable. The case of discrete consumable resources with bounded capacity described above, which is expressible in the basic propositional STRIPS model, is one example of such a restricted model. Another planning model for which the plan existence problem is decidable is when consumable resources are restricted to be monotonic. This is the model for which we describe an adaptation of the regression planning method and heuristics below. Helmert (2002) also presents several other restrictions that lead to decidable planning models.

Regression Planning with Monotonic Consumable Resources

This section describes how the regression planning method, for sequential and temporal planning, is adapted to work for the corresponding planning models extended with *monotonic* consumable resources. As the plan existence problem becomes undecidable in the presence of unrestricted consumable resources, the method can not be complete (and therefore not deliver guaranteed optimal solutions) unless some further restrictions on the use of resources are made. The restriction to monotonic resources is sufficient to ensure decidability (as, in fact, follows from the correctness of the method described below).

To simplify the presentation we assume that all resources are decreasing and have a minimum capacity of zero, but these assumptions are not restrictive. (As noted above, a monotonic increasing resource can be transformed into an equivalent decreasing resource. A decreasing resource with a minimum capacity other than zero can be "shifted" by adding a constant to the capacities and the initial level.) As usual we begin with the case of sequential planning, followed by the adaptation to the temporal planning case.

Sequential Regression with Monotonic Consumable Resources

Recall that regression is a search in the space of plan tails, partial plans leading to a goal-satisfying world state, and that the search state summarizes what needs to be known about the plan tail in order to complete it into a plan. In sequential regression planning the search state is the regressed goal condition: in any world state satisfying this condition the corresponding plan tail is known to be executable. With consumable resource constraints, however, the regressed goal condition alone is no longer enough to summarize a plan tail. To ensure that the completed plan meets resource capacity constraints (and is thus executable) a summary of the resource effects of the plan tail must also be included in the search state.

When consumable resources are restricted to be decreasing, the level of a resource during the execution of a valid plan can only change downwards from the level in the initial world state to the minimum capacity of the resource. Therefore, the total amount consumed by the actions in a plan tail, for each resource, is sufficient to summarize the resource effects of the plan tail. If this amount is less than the

amount initially available for all resources when the plan is complete, executing the plan will not reduce any resource below zero (the assumed minimum capacity of all resources). If, on the other hand, this amount exceeds the amount initially available for some resource, the plan tail can not be completed into a plan, as the level of a resource can not be increased.

Formally, a search state s is extended to include the total amount consumed of each resource r, denoted $s(r)$. $s(r) = 0$ for each resource r in the starting search state. An action a is applicable in search state s if $s(r) + -chg(a, r) \leqslant init(r)$, for each resource r, in addition to the condition that $del(a) \cap s = \emptyset$ (described in chapter 2, page 8). In the successor state s' resulting from application of a to s, $s'(r) = s(r) + -chg(a, r)$, for each resource r. (Recall that $chg(a, r) < 0$ if a consumes r: thus the amount of r *consumed* by a is $-chg(a, r)$.)

Temporal Regression with Monotonic Consumable Resources

The modification described for sequential regression search above also works for temporal regression planning with decreasing consumable resources. Recall from chapter 2 (page 16) that the construction of a successor state to temporal regression state s involves choosing (non-deterministically) a set of actions A, mutually compatible and compatible with scheduled actions in s. The chosen actions are all added to the plan tail, which increases the total amount consumed of each resource r by $\sum_{a \in A} -chg(a, r)$.

Resource Consumption and the Persistence of World State

A fundamental assumption in the STRIPS and temporal STRIPS planning models is that *world states persist*, *i.e.*, that an atom once made true in a world state remains true, without further effort, until explicitly made false by an action. This assumption limits the expressive power of the models, and needs to be relaxed in order to capture the essential constraints of some planning problems. As an example, consider the UAV mission planning problem described in chapter 2 (page 13). An important aspect of this problem, ignored in the problem model presented in chapter 2, is that the UAVs carry a limited supply of fuel and have to land before fuel runs out. In the extended planning model, it is straightforward to introduce a resource `(fuel ?u)`, for each UAV `?u`, and to add to each `(fly ?u ?p)` action a corresponding resource effect, $chg($`(fly ?u ?p)`$,$ `(fuel ?u)`$) = -v$ (for some appropriate value v), but this problem description still ignores the fact that a UAV (helicopter) consumes fuel also while hovering and not just when flying along a path. In the problem formulation, however, there is no action for hovering (as the activity of UAV `?u` hovering at position `?p` is represented by the persistence of the atoms `(airborne ?u)` and `(at ?u ?p)`) and therefore no way to state the associated fuel cost.

Lifting this limitation requires only a small extension to the planning model, and to the temporal regression method. For each resource r, each atom p has a *maintenance effect* on that resource (denoted $chg(p, r)$) which is negative if maintaining the truth

of the atom consumes the resource and zero otherwise (the case that maintaining the truth of an atom produces resources is less reasonable, so we do not consider it). While the resource effect of an action is interpreted as an absolute amount, the maintenance effect of an atom is interpreted as a *rate* of consumption. In other words, the amount of resource r consumed by maintaining atom p over an interval of duration t is $t * -chg(p, r)$. In the construction of a successor state s' to a temporal regression state $s = (E, F)$, any atom $p \in E$ not established by one of the chosen actions is supported by a no-op (*i.e.*, by persistence) from the next time point (see chapter 2, page 16). The duration of the interval of persistence is δ_{adv}, as defined by equation (1) on page 16. Thus, when the successor state s' is finalized the maintenance resource cost during the step from s to s' is known and can be added to the total amount consumed in s' (*cf.* figures 2.6(b) and (c), page 18).

The extension described above, although simple, increases the expressive power of the temporal STRIPS planning model drastically, as it makes it possible to force one action to follow another within a certain time (including to follow immediately after). This opens up several possibilities, *e.g.*, of splitting an action into a series of consecutive actions and thus to model the temporal relations between effects and conditions of an action more precisely (*cf.* figure 2.9 and the discussion in section 2.3, page 27). The increased expressive power comes at the cost of a computationally harder planning problem, however: in particular, the introduction of maintenance resource costs weakens the early detection of cost bound violations (described in section 2.2, page 18) and aggravates the problem of isolated resource consumption estimates, discussed in the next section.

Heuristics for Resource Consumption

In the regression search space for planning with decreasing consumable resources described above, a branch of the search tree is cut only when resource over-consumption is a fact, *i.e.*, when the total amounts consumed are so large that no relevant action is any longer applicable. This can cause a great waste of search effort: as it is in a cost bounded search important to detect cost bound violations early, so it is in a resource bounded search important to detect resource over-consumption as early as possible.

Let $h_r(s)$ denote an admissible heuristic function for the consumption of resource r, *i.e.*, $h_r(s)$ is a lower bound on the minimum amount of resource r consumed by any plan that achieves s. If $s(r) + h_r(s) > init(r)$, no plan that achieves s and respects resource constraints exists and thus the branch below s can be pruned from the search tree. Note that the search still minimizes plan cost (in the sequential case) or makespan (in the temporal case). The resource consumption estimator is used only to prune search states that can not lead to any solution.

The total amount consumed of a resource r by a plan equals the sum of the amounts consumed by each of the actions in the plan. Thus, resource consumption is an additive cost measure, same as the cost associated with actions in sequential planning

(except that the resource cost of an action may be zero for some resource, if the action does not use the resource, whereas action cost in sequential planning was assumed to be strictly positive). This means that admissible heuristics for estimating the amount of resources required to achieve a goal (or search state) can be derived in exactly the same way as admissible heuristics for sequential planning, *e.g.*, as described in chapters 3, 4 and 6, only with a different measure of cost.

Composite Resources

The approach to deriving admissible heuristics for resource consumption outlined above, although simple, suffers from a significant weakness: the required amount of each resource is estimated in isolation, independent of the use of other resources and independent of the plan cost or makespan. In many planning problems, there is a possibility of trading use of one resource for use of another, or of obtaining a plan with smaller resource use at the expense of increased cost or makespan.

Consider, for example, the MRCPS problem depicted in figure 5.2(a), and specifically job `J4`. If executed in mode `M1` or mode `M2` this job consumes resource `F2`, but if executed in mode `M3` it consumes resource `F1`. Thus, neither resource `F1` or `F2` is *necessarily* consumed by a plan for doing this job (for achieving the goal `(done J4)`). Heuristic estimates of the resources required to achieve `(done J4)` suffer from the same problem: $h^1_{\texttt{F1}}(\texttt{(done J4)}) = 0$ and $h^1_{\texttt{F2}}(\texttt{(done J4)}) = 0$, since the two jobs preceding `J4` can also both be done using either resource ($h^1_{\texttt{F1}}$ and $h^1_{\texttt{F2}}$ denote the (sequential) h^1 heuristic function computed with the consumption of `F1` and `F2` as cost measure, respectively). However, it is clear that `(done J4)` can not be achieved without consuming some resource.

This problem can be countered to some extent by introducing *composite resources*. A composite resource is an artificial entity representing the sum of several resources, but is treated like any other consumable resource. If r_1 and r_2 are consumable resources in a planning problem, the characteristics of the composite resource $r_1{+}r_2$ are obtained by adding the corresponding characteristics of r_1 and r_2, *i.e.*, $r_1{+}r_2$ has an initial level equal to $init(r_1) + init(r_2)$ and $chg(a, r_1{+}r_2) = chg(a, r_1) + chg(a, r_2)$ for each action a. Composite resources can of course also be defined as the sum of more than two resources.

Conjoining $r_1{+}r_2$ to the problem description does not change the set of valid plans: any plan that respects the capacity constraints of r_1 and r_2 also respects the constraints imposed by $r_1{+}r_2$. More importantly, any plan that overconsumes $r_1{+}r_2$ must also overconsume r_1 or r_2, so an admissible heuristic for $r_1{+}r_2$ cost can be safely used to prune unsolvable search states. Intuitively, $h_{r_1+r_2}$ can be viewed as a heuristic obtained by a relaxation of the problem which is to assume that r_1 and r_2 are completely interchangeable. Returning to the example of job `J4` of the MRCPS problem in figure 5.2(a), $h^1_{\texttt{F1+F2}}(\texttt{(done J4)}) = 10$ (since `J2` consumes a minimum of 6 units of resource `F2` and `J4` a minimum of 4 units of resource `F1`). An experimental analysis of the effectiveness of the use of composite resources, in combination with the h^1 and h^2 heuristics for resource cost, is presented in the next section.

The introduction of composite resources, however, can not solve the problem that the heuristic for the primary optimization objective (cost or makespan) is independent of the resource consumption heuristics. To address this issue properly requires a true multi-criteria optimization method, *e.g.*, along the lines suggested by Refanidis & Vlahavas (2003). This is an interesting topic for future research.

Analysis: Effectiveness of Resource Consumption Heuristics

This section presents an experimental analysis of the effectiveness, and cost effectiveness, of the use of the h^1 and h^2 heuristics for resource cost, with and without composite resources, in the MRCPS problem domain. The problems used in the experiment are taken from the PSPLib repository of benchmark problems (sets J16 and N3; see Kolisch & Sprecher, 1996, or `http://129.187.106.231/psplib/`). Each problem consists of 16 jobs with 3 possible modes of execution and has 2 reusable and $2-3$ consumable resources. The amount available of each resource in relation to the amounts required varies according to two parameters, called the reusable and consumable resource strength. At strength 1 a resource is available in sufficient quantity to execute each job in any mode, so the resource imposes practically no constraints on possible schedules, while at strength 0 the resource is limited to the minimum that allows each job to be executed in some mode, so that the resource alone does not render the problem unsolvable (though at low resource strengths, many problems are still unsolvable due to interactions between different resources). The strength of reusable and consumable resources in the problem set varies between 0.25 and 1.

Four different planner configurations are compared. The base configuration uses temporal regression, with extensions for reusable and consumable resources as described in this chapter, in an IDA* search (with standard enhancements, *i.e.*, right-shift cuts and a transposition table) with the h^2 heuristic for makespan (the optimization objective) and the h^1 heuristic to estimate the consumption of each consumable resource. The three alternative configurations differ from the base configuration by the addition of composite resources (all combinations of size 2), by replacing the h^1 resource consumption heuristic with h^2 for each resource and by the combination of both, respectively. A time limit (4 hours CPU time) for each problem was set for each of the planners.

Results are summarized in figures 5.3(a) and (b): figure 5.3(a) shows a runtime distribution (percentage of instances solved as a function of runtime), while figure 5.3(b) shows the distribution of the reduction, compared to the base configuration, in the number of nodes expanded for the three alternative planner configurations (only instances solved by both the base configuration and the alternative are included in this comparison).

Clearly, there is not a large difference between the four configurations: for the majority of problems the reduction in the number of expanded nodes is small (less than 10%) and although there is some runtime overhead associated with each of the alternative configurations (the base configuration is the fastest overall) this is also quite

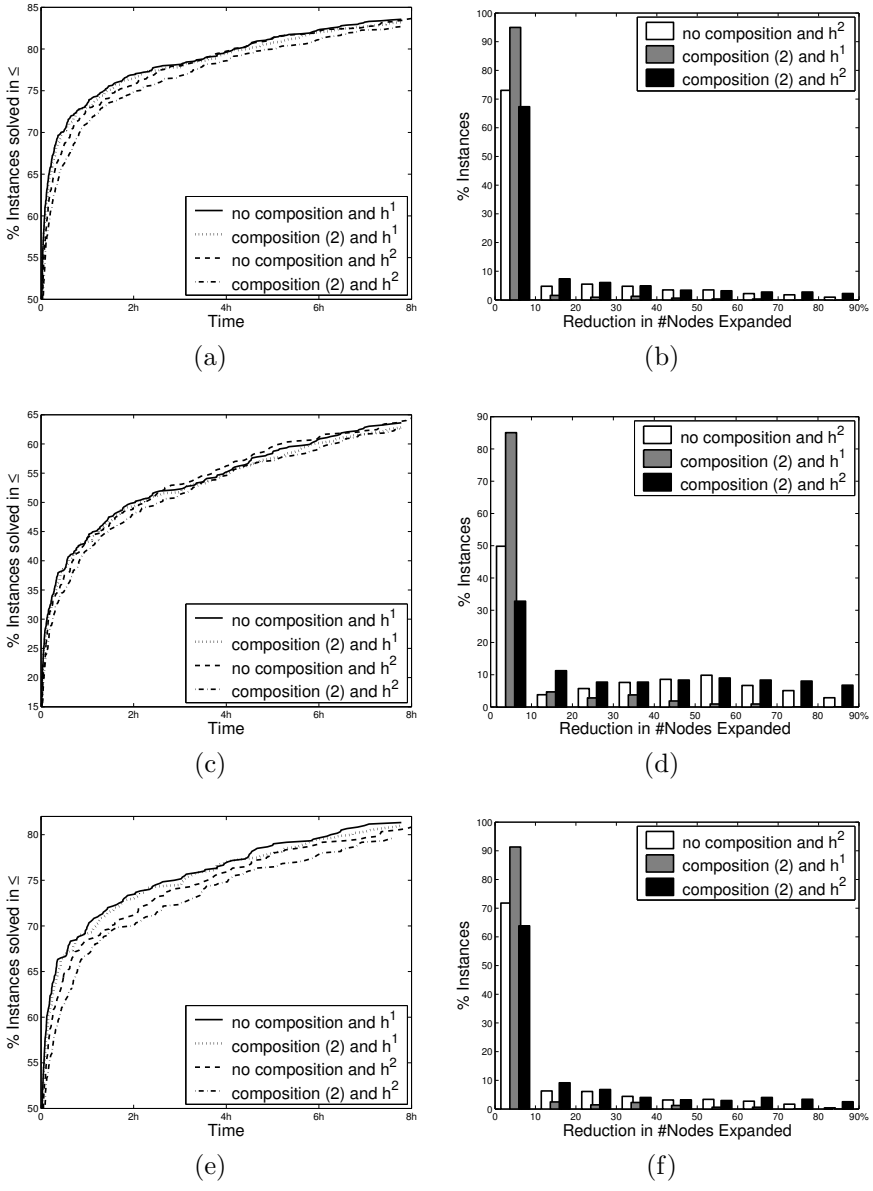


Figure 5.3: (a) Runtime distributions for the four variations of resource consumption estimators (using the h^1 and h^2 heuristics, with and without composite resources) on the MRCPS problem set. (b) Distribution of the reduction in the number of nodes expanded, compared to the base configuration (using the h^1 heuristic and no composite resources). Distributions are also shown for two subsets of problem instances: in (c) – (d) instances with tighter consumable resource constraints (instances with consumable resource strength $\leqslant 0.5$, roughly 45% of the problem set) and in (e) – (f) instances with 3 consumable resources.

small. Still, it can be observed that the introduction of composite resources only leads to any significant reduction in node expansions if combined with the use of the h^2 heuristic to estimate resource consumption.

The lack of difference can be explained by the fact that in a large fraction of the instances in the problem set it is not the consumable resource constraints that are the main source of difficulty (some problems are simply easy and in some it is reusable resource constraints that cause difficulty). Figures 5.3(c) and (d) show the same distributions for the subset of problem instances with a consumable resource strength of 0.5 or less, *i.e.*, problems in which consumable resource constraints are relatively tight. On this subset the reduction in node expansions is greater for all the alternative configurations, so much so that the configuration using the h^2 resource consumption heuristic without composite resources achieves runtimes comparable to those of the base configuration. Figures 5.3(e) and (f), finally, show the same distributions for another subset of instances, those with 3 consumable resource. Also here a slightly greater reduction in node expansions, compared to the full problem set, can be observed for all three alternative configurations. The runtime overhead, however, is also noticeably larger, in particular for the configurations using the h^2 heuristic to estimate resource consumption. This is reasonable since evaluating the h^2 heuristic is computationally more expensive than evaluating h^1 and this price is paid for each consumable resource in each state evaluated during the search.

In conclusion, the results suggest that it is only for problems with a small number of tightly constrained consumable resources that the introduction of composite resources in combination with the use of the h^2 heuristic to estimate resource consumption is cost effective. It should be noted, however, that in this experiment only a few of the many possible configurations of resource estimators were compared. Interesting alternatives to investigate in future research are the use of pattern database heuristics (described in chapter 4) and additive h^m heuristics (described in chapter 6) for resource consumption and more selective resource composition strategies.

6. Cost Relaxation and the Additive h^m Heuristics

The h^m family of admissible heuristics, introduced in chapter 3, is based on the relaxation of assuming that the cost of achieving any set of more than m atoms equals the cost of the most costly subset of size m. This recursive focus on the most costly part of each state means that the heuristic can be viewed as a generalization of the critical path length estimate, but because a state of size $|s| > m$ can have several subsets with maximum cost and a state of size $|s| \leqslant m$ can have several alternative minimum cost regressions the heuristic is more accurately described as measuring the (weighted) height of a "critical tree" rooted in the state evaluated. Illustrations of this critical tree (for two example Blocksworld problems) can be found in figures 3.2 and 3.3 in chapter 3 (page 40).

In sequential planning, however, the cost of a plan is the *sum* of the costs of the actions in the plan, so a heuristic estimate obtained by summing the costs of all actions in the critical tree, rather than only those along an arbitrary critical path, is often much more accurate. In fact, it is something like this that is done in the HSP heuristic, which adds up the estimated costs of subgoal atoms in any state with more than one atom, and the FF heuristic, which counts the number of actions in a relaxed plan, corresponding to the critical tree of the h^1 heuristic (Bonet & Geffner 1999; Hoffmann 2000). However, these heuristics are not admissible. The HSP heuristic fails to take into account the fact that some actions may contribute to the achievement of more than one subgoal and thus appear more than once in the critical tree. The FF heuristic does take this into account to some extent, as in the relaxed plan it extracts such duplicate actions are only counted once but it does not extract the *minimal* relaxed plan because doing this is NP-hard (it amounts to solving optimally a STRIPS problem with actions that have no negative effects; see Bylander, 1991).

This chapter introduces the principle of *cost relaxation*, by which the h^m heuristic for sequential planning can be made additive (and thus more accurate) while retaining admissibility. The idea is simple: The set of actions is partitioned into disjoint sets, $A_1, \ldots, A_n$, and the heuristic $h^m_{A_i}$ is defined (and computed) exactly like h^m except that the cost of any action *not* in A_i is considered to be zero. The sum $\sum_i h^m_{A_i}$ is then admissible, and often a more accurate estimate than h^m. This is in fact a general principle which can be applied to any admissible heuristic for sequential planning (provided the heuristic satisfies a certain condition), or indeed for any problem where the cost of a solution is the sum of individual costs of elements of the solution.

As with pattern database heuristics the accuracy of the additive h^m heuristics depends crucially on the selection of an appropriate partitioning of the set of actions. A method for automatically selecting the action partitioning is presented (in section

6.2) and the quality and cost effectiveness of the resulting heuristic is experimentally compared to the normal h^m heuristic and the pattern database heuristics developed in chapter 4.

6.1 The Additive h^m Heuristics

For any subset A of the actions in a planning problem, $h^m_A(s)$ denotes a heuristic function that is defined identically to h^m except that the cost of every action not in the set A is considered to be zero. We say the *costs* of those actions are *relaxed*.

More formally, the set A can be said to define a relaxed planning problem (P_A) that is identical to the original problem (P) except that in the relaxed problem $cost(a) = 0$ for every action $a \notin A$: h^m_A is then the h^m heuristic for the relaxed problem P_A. This means that any method for computing h^m can also be used to compute h^m_A, by changing only the cost function.

Any solution to the original problem is also a solution to the problem with relaxed action costs but may have a lower cost in the relaxed problem if the solution contains some actions whose costs have been relaxed. This implies that $h^*_A(s) \leqslant h^*(s)$ for all s (with equality for all s when A is the set of all actions in the problem) and since h^m_A is admissible for the relaxed problem (*i.e.*, $h^m_A(s) \leqslant h^*_A(s)$, for all s) it follows that $h^m_A(s)$ is admissible also for the original problem. Moreover, if $A_1, \ldots, A_n$ are disjoint sets of actions then the sum $h^m_{A_1}(s) + \ldots + h^m_{A_n}(s)$ is also an admissible heuristic, since the cost of no action is counted more than once (a formal proof is given below).

For an illustration of the additive h^m heuristic, consider again the Blocksworld problem of constructing a tower, represented by the goal $\{$`(on b1 b2)`$, \ldots,$ `(on bn-1 bn)`$\}$, from a collection of n blocks, `b1` to `bn`, initially on the table. This problem was first described in chapter 3 (page 40) and also discussed in chapter 4 (page 61 and page 78) where it was used to illustrate the advantage of additivity in pattern database heuristics. The shortest plan for this problem consists of $n - 1$ actions, since each block except the one at the base of the tower needs to be moved, but the h^1 value of the goal G of having all blocks in their goal positions is only 1, since only one move is needed to achieve each subgoal `(on bi bi+1)`.

If the set of actions is partitioned into n sets such that set A_i contains all actions that move block `bi` then $h^1_{A_i}($`(on bi bi+1)`$) = 1$. For each other subgoal `(on bj bj+1)` the value of $h^1_{A_i}$ is zero, because even though this goal also requires one action (moving block `bj`) the cost of this action is relaxed in $h^1_{A_i}$. Thus, $h^1_{A_i}(G) = 1$ (except for $h^1_{A_n}(G) = 0$, since block `bn` at the base of the tower does not have to be moved) and because each $h^1_{A_i}$ counts the cost of a different action the sum $h^1_{A_1}(G) + \ldots + h^1_{A_n}(G)$ is $n - 1$, the optimal plan length. Figure 6.1 illustrates for the case of three blocks (named `A`, `B` and `C`).

Next, consider the Blocksworld problem of swapping the two blocks at the base of a tower (introduced in chapter 4, page 65; initial and goal states for a small instance with three blocks are shown in figure 4.3, page 65). The shortest solution to this

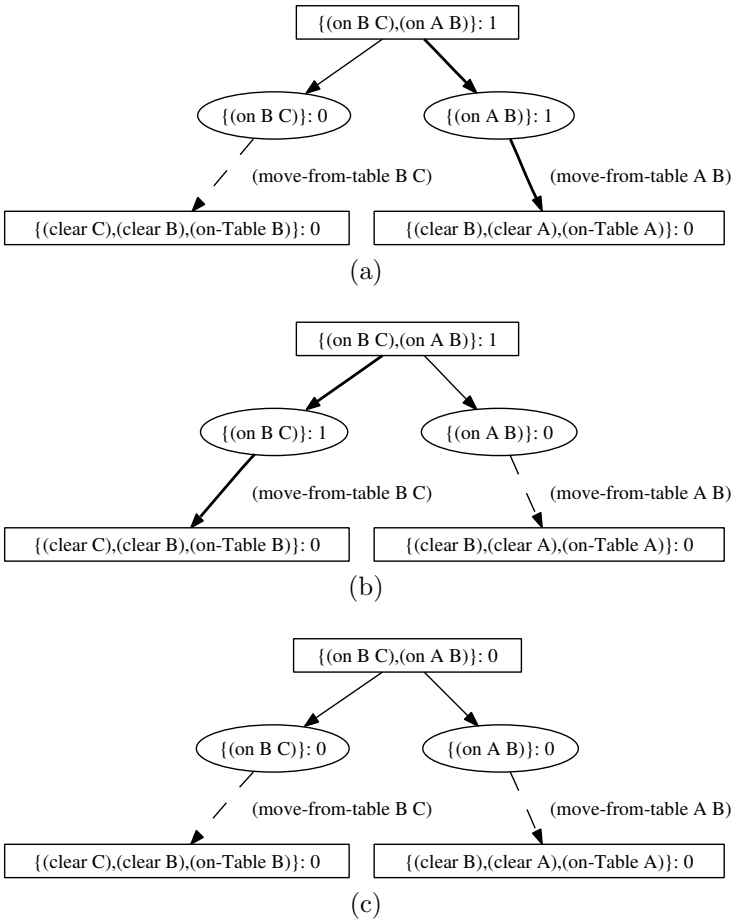
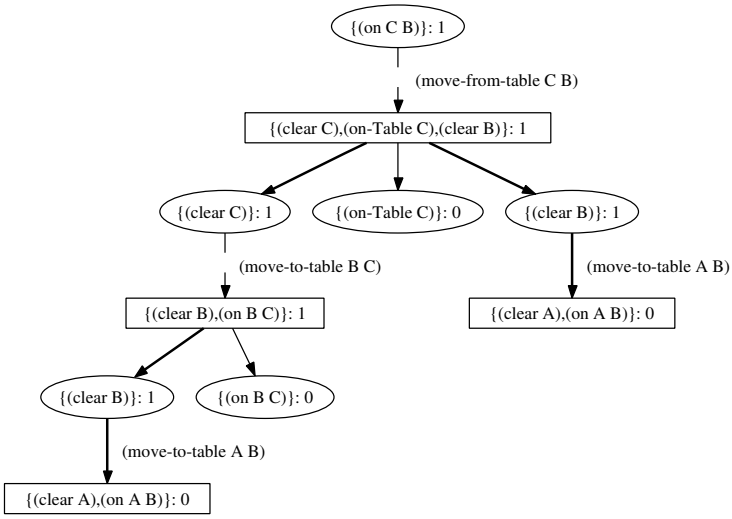
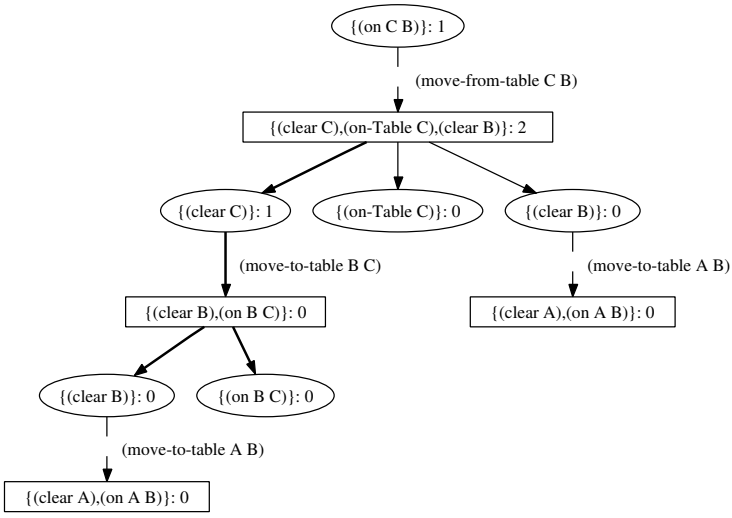


Figure 6.1: Calculation of the (a) $h^1_{\texttt{A}}$, (b) $h^1_{\texttt{B}}$ and (c) $h^1_{\texttt{C}}$ values for the goal $G = \{\texttt{(on A B)}, \texttt{(on B C)}\}$, where $h^1_{\texttt{?b}}$ counts only actions that move block `?b`. Cost relaxed actions are drawn as dashed arrows and the critical tree is indicated in bold. Note how $h^1_{\texttt{A}}(G) + h^1_{\texttt{B}}(G) + h^1_{\texttt{C}}(G) = 2$.

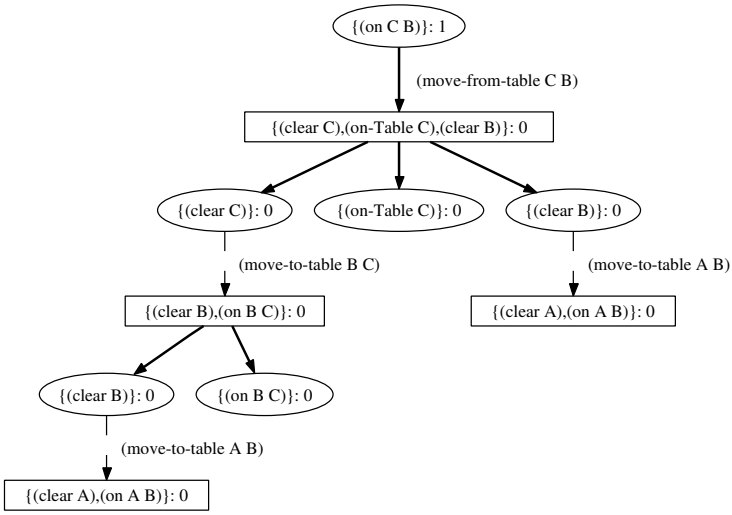


(a)

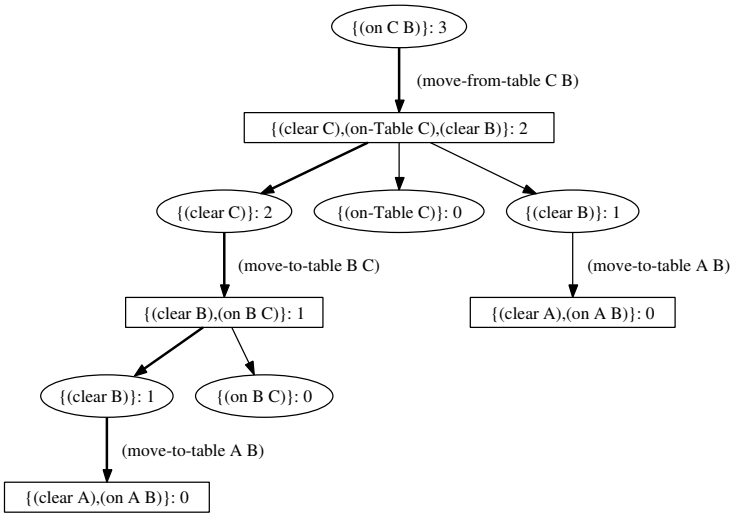


(b)

Figure 6.2: Calculation of the (a) $h^1_{\texttt{A}}$, (b) $h^1_{\texttt{B}}$ and (c) $h^1_{\texttt{C}}$ values for the goal of the "swap base blocks" Blocksworld problem, $G = \{\texttt{(on C B)}\}$. Figure (d) shows the calculation of the plain h^1 value, for reference. Cost relaxed actions are drawn as dashed arrows and the critical tree is indicated in bold (some of the relaxed actions appearing in each of the figures are also part of the critical tree). Note that $h^1_{\texttt{A}}(G) + h^1_{\texttt{B}}(G) + h^1_{\texttt{C}}(G) = 3$, which is also $h^1(G)$.



(c)



(d)

Figure 6.2 (continued).

problem needs to move every block once (the blocks above the one at the base to clear it and the block at the base because the goal is to have it on another block). As noted in the discussion of the relative merits of the pattern database and h^m heuristics in chapter 4 (page 78) even the h^1 heuristic yields the optimal cost (n moves for n blocks) for this problem. This is because h^1 considers the most costly subgoal recursively: to move $\mathtt{b}_n$ onto $\mathtt{b}_{n-1}$, $\mathtt{b}_{n-1}$ must be clear; to clear $\mathtt{b}_{n-1}$, $\mathtt{b}_{n-2}$ must be moved so it must also be clear; and so on. The sum of $h^1_{A_i}(G)$, for the same action partitioning $A_1, \ldots, A_n$ as in the previous example, yields the same result. This is because even though each $h^1_{A_i}$ only counts the cost of one action they all detect that the other actions are necessary, by the same reasoning as for the standard h^1 heuristic, and therefore all the necessary actions are counted by some component of the sum. Figure 6.2 illustrates for the case of three blocks.

Compared to the standard h^m heuristic the additive version is clearly computationally more expensive, to compute as well as to evaluate, as it involves computing and evaluating the standard h^m heuristic n times. For this to be cost effective the heuristic must be much more accurate which depends on selecting the sets of actions $A_1, \ldots, A_n$ appropriately, in the same way that the effectiveness of pattern database heuristics depends on the selection of appropriate patterns. In practice, the collection of subsets $A_1, \ldots, A_n$ will be a partitioning of the set of actions so that the cost of every action is counted by some $h^m_{A_i}$. A method for partitioning the set of actions automatically is presented in section 6.2 below.

Admissibility of Cost Relaxation

The modification done to the h^m heuristic above can be generalized to what can be called the *principle of cost relaxation*. Next, we show that this principle is applicable to any heuristic function satisfying a certain condition for a wide class of search spaces and that applying it preserves the admissibility of the heuristic. Admissibility of the additive h^m heuristic for sequential planning follows as a special case.

Consider a search space, defined as in chapter 3 (page 38) by an initial state, a set of final states and a basic transition relation ($R(s, s', c_{s,s'})$ iff there is a transition from s to s' with cost $c_{s,s'}$), with the property that the cost of a solution path is the sum of the costs of transitions along the path, *i.e.*, if $s_0, \ldots, s_l$ is a sequence of states such that s_0 is the initial state, s_l belongs to the set of final states and $R(s_{i-1}, s_i, c_{s_{i-1},s_i})$ holds for $i = 1, \ldots, l$, then $s_0, \ldots, s_l$ is a solution and its cost equals $\sum_{i=1,\ldots,l} c_{s_{i-1},s_i}$. (Recall that in the regression search space for sequential planning, $R(s, s', c_{s,s'})$ holds iff s' can be obtained by regressing s through some action a and in this case $c_{s,s'} = cost(a)$.) Also, let $h^*_R(s)$ denote the optimal cost function for the search space (the function that assigns to each state s the minimal cost of any path from s to a final state, or ∞ if no such path exists) and let h_R be an admissible heuristic for the search space (a function such that $h(s) \leqslant h^*(s)$ for all s). Here we assume that the function h is parameterized by R, *i.e.* that any transition relation R' over the same set of states defines a corresponding heuristic function $h_{R'}$ that is admissible with respect to R', *i.e.* such that $h_{R'}(s) \leqslant h^*_{R'}(s)$ for all s.

Let $R_1, \ldots, R_n$ be a collection of transition relations over the same set of states as R with the property that (1) $R_i(s, s', \cdot)$ iff $R(s, s', \cdot)$, for all i and (2) if $R_i(s, s', c)$ and $c > 0$ then $R(s, s', c)$ and $R_j(s, s', 0)$ for all $j \neq i$. In other words, each R_i has the same transitions as R but counts only the cost of some of them and each transition is counted only in one R_i. (In the case of sequential regression planning a division of R into $R_1, \ldots, R_n$ is obtained by dividing the set of actions in the planning problem into disjoint sets, $A_1, \ldots, A_n$, counting only the cost of actions in A_i.)
Finally, let $h_\Sigma(s) = \sum_{i=1,\ldots,n} h_{R_i}(s)$.

Theorem 7 h_Σ is an admissible heuristic for the search space R, *i.e.*, $h_\Sigma(s) \leqslant h^*(s)$, for all s.
Proof: To show this note that the cost of a solution path $s_0, \ldots, s_l$ can be rewritten as

$$\sum_{i=1,\ldots,l} c_{s_{i-1},s_i} =$$
$$\sum_{k=1,\ldots,n} \left(\sum_{i=1,\ldots,l} c_{s_{i-1},s_{i_k}} \mid R_k(s, s', c_{s_{i-1},s_{i_k}}) \right) +$$
$$\left(\sum_{i=1,\ldots,l} c_{s_{i-1},s_i} \mid R(s, s', c_{s_{i-1},s_i}), R_{k=1,\ldots,n}(s, s', 0) \right)$$

i.e., as the sum of the cost of the path according to each relation R_k, plus the cost of the transitions not counted in any R_k. Equality holds because of condition (2) above, that the sets of transitions counted in each of the relations R_k are disjoint. Since this holds for any solution path it holds also for the minimum cost paths and therefore

$$\sum_{k=1,\ldots,n} h^*_{R_i}(s) \leqslant h^*_R(s)$$

for all s, from which it follows that

$$\sum_{k=1,\ldots,n} h_{R_i}(s) \leqslant h^*_R(s)$$

since each h_{R_i} underestimates $h^*_{R_i}$. □

Admissibility of the additive h^m heuristic for sequential regression planning follows from this general theorem because solution cost in the sequential regression space is additive, the partitioning of actions induces a corresponding collection of transition relations R_i satisfying properties (1) and (2) stated above, and the h^m heuristic is admissible and parameterized by the transition relation (induced by the set of actions and their associated costs).

6.2 Partitioning the Set of Actions

While the sum $\sum_i h^m_{A_i}$ is admissible for any collection $A_1, \ldots, A_n$ of disjoint sets of actions, the quality of the heuristic depends very much on the selection of an appropriate partitioning. This section presents one method for partitioning the set of actions automatically and demonstrates experimentally that it generally results in a good heuristic.

The approach is to create one partition A_i for each atom g_i in the problem goal G and assign actions to the partition where they appear to contribute the most to the sum. To determine if an action a is relevant for goal atom g_i the heuristic resulting from relaxing the cost of a is compared to one that does not relax the cost of a: if the value of $h^m_{A-\{a\}}(g_i)$ is less than $h^m_A(g_i)$ then action a belongs to the critical tree of g_i and is clearly relevant. The "loss", $h^m_A(g_i) - h^m_{A-\{a\}}(g_i)$, is a (rough) measure of how relevant action a is to each of the different goals and the action can be assigned to the goal where relaxing its cost causes the greatest loss. Ties can be broken arbitrarily, randomly or using some secondary ranking (*e.g.* assigning actions to the smallest partition with the greatest loss balances the sizes of the partitions).

The process of assigning actions to goals is iterative. Initially, partition $A_1, \ldots, A_n$ contains all actions. When an action is assigned to one partition it is removed from all the other so that at the end, when all actions have been assigned, the partitions are disjoint (and their union contains all actions).

The test $h^m_{A-\{a\}}(g_i) < h^m_A(g_i)$, however, is sometimes not enough to detect relevance. This is because the h^m value of even a single atom is determined often by a critical tree, branching out on the preconditions of actions. Thus, relaxing the cost of a single action sometimes does not change the h^1 estimate of an atom, although simultaneously relaxing the costs of several actions does. The impact of this problem can be lessened by dividing the set of actions into sets of related actions and performing the test for each set of actions simultaneously relaxed. The preliminary division of actions into sets is based on state variables, specifically on finding a maximal set of additive state variables (as described in chapter 4, page 69): related actions are those that affect the same variable in this set. Because the variables are additive the sets of related actions are disjoint and because the set of additive variables is maximal all actions are in some set (except for actions that do not have any effect, but those can safely be ignored).

The partitioning method described above has some obvious shortcomings: First, if the goal of the planning problem consists of only one atom, it will not result in any partitioning at all. This problem can be overcome by basing partitions, in case the goal set of atoms is too small, on the set of atoms immediately relevant to the goal atoms, *i.e.*, the set of atoms that appear in search states reachable by one or a few regressions. Second, since the method involves computing the h^m heuristic numerous times it can be computationally very expensive. In the experimental analysis presented in the next section the test for relevance is done with the h^1 heuristic, even though the additive heuristic computed for the final partitioning is h^2, precisely for

this reason.

Analysis: Comparison of the h^m, Additive h^m and PDB Heuristics

This section presents an experimental comparison of the quality and cost effectiveness of the normal and additive h^2 heuristics and the pattern database heuristics discussed in chapter 4. The experiment involves planning problems from three domains, each of which demonstrates a very different possible relationship between the compared heuristics.

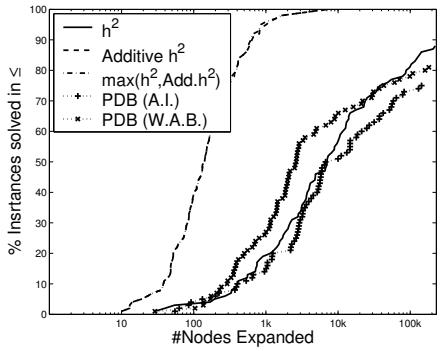
The combined heuristic $\max(h^2, \text{additive } h^2)$ is also included in the comparison. Intuitively, the reasons for considering this heuristic are that the h^2 and the additive h^2 heuristics often exhibit complementary strengths (see figure 6.6) and that the overhead of computing and maximizing with the h^2 heuristic is relatively small compared to the cost of computing and evaluating the additive h^2 heuristic. As it turns out, however, maximizing the h^2 and additive h^2 heuristics does not always result in a more effective heuristic: for some problems search with this heuristic actually expands more nodes than search with only the additive h^2 heuristic. The effect of maximizing is analyzed more in depth below (page 110).

The Experiment

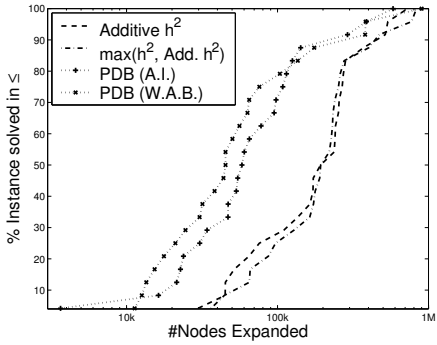
The comparison of the heuristics is made in the context of sequential regression, with A* search. The quality, or effectiveness, of each heuristic is measured by the number of nodes expanded during search using the heuristic, since this indicates how effectively it guides the search. The cost effectiveness of each heuristic is measured by the total runtime, including both the time required for computing the heuristic and for the search, since this shows if the time spent on computing the heuristic is compensated for by a reduction of the time spent on search. Each run was limited to a total of 1GB of memory, for heuristics and for the search, but no time limit was imposed.

The pattern database heuristics were computed by the weighted additive bin-packing and additive incremental pattern selection methods (described in section 4.3 of chapter 4). The PDB size limits were the same as in the experimental comparison of pattern selection methods in chapter 4. The additive h^2 heuristic was computed with the action partitioning method described above. As already mentioned the h^1 heuristic was used to test for loss in the partitioning of actions, since using h^2 for this purpose is too computationally expensive. In case of ties assigning actions to smaller partitions was preferred, since this tends to keep the sizes of the partitions more even.

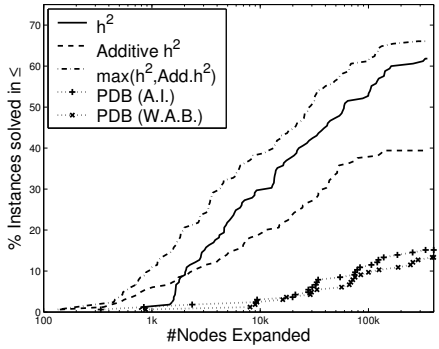
The planning problems used in the experiment are from the Blocksworld, 15-Puzzle and random STRIPS domains. The Blocksworld problems are the same as in the experiment described in chapter 3 (page 46; the smaller half of this problem set was



(a) Blocksworld

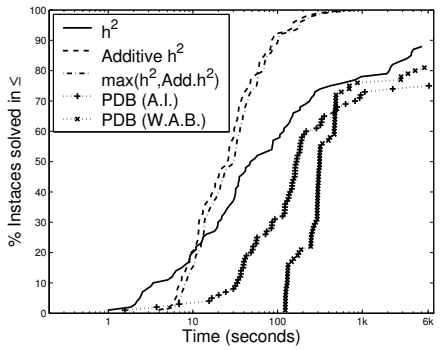


(b) 15-Puzzle

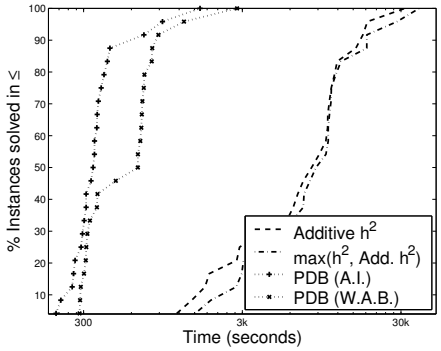


(c) Random STRIPS

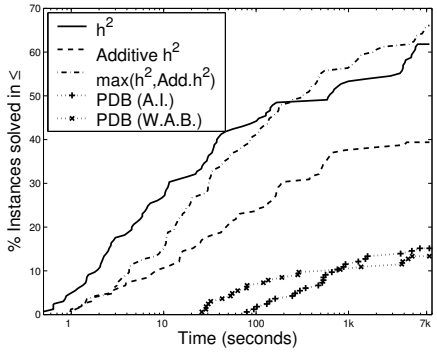
Figure 6.3: Cumulative distribution of the number of node expansions required to reach a solution by A* search with the h^2 and additive h^2 heuristics, the combination (by maximization) of the two (in figure (a) the curves for additive h^2 and the combined heuristic are so close as to be nearly indistinguishable), and the PDB heuristics resulting from additive incremental ("A.I.") and weighted additive bin-packing ("W.A.B.") pattern selection. Note that the scale on the X-axis (number of nodes expanded) is logarithmic.



(a) Blocksworld



(b) 15-Puzzle



(c) Random STRIPS

Figure 6.4: Distribution of the time required to find a solution with the h^2, additive h^2, combined and PDB heuristics. The time measured is total runtime, *i.e.*, including the time required to compute heuristics before the search. The scale on the X-axis (time) is logarithmic.

also used in the comparison of pattern selection methods in chapter 4, page 73). The 15-Puzzle problems are the easiest quarter of the collection of random 15-Puzzle problems presented by Korf (1985; also the same problems that were used in the experiment reported in chapter 4). The random STRIPS problem set included the 55 problems originally generated for the experiment reported in chapter 4 and an additional set of 110 larger problems (parameters $n = 90$, $o = 218$, $r = s = 3$ and $g = 9$) generated by the same method (described on page 74; note that the final filtering step was not applied here).

Results and Conclusions

Results of the experiment, in the form of the cumulative distribution of the node expansions and runtime required to find a solution, are shown in figures 6.3 and 6.4, respectively. As can be seen, the relative effectiveness, and cost effectiveness, of the compared heuristics is very different in the three problem domains.

In the Blocksworld domain (figures 6.3(a) and 6.4(a)) the additive h^2 heuristic is clearly more effective than the PDB and h^2 heuristics, which are rather similar. Maximizing the additive h^2 heuristic with h^2 reduces the number of nodes expanded only marginally but incurs a noticeable runtime overhead.

In the 15-Puzzle domain (figures 6.3(b) and 6.4(b)) the PDB heuristics are more effective than the additive h^2 heuristic and the difference is magnified even more when looking at runtime, since the computational cost of heuristic evaluation is greater for the additive h^2 heuristic. Search with only the plain h^2 heuristic fails to solve any instance in this domain and neither does maximizing h^2 with the additive h^2 improve the effectiveness of the latter. In fact, for roughly half of the instances search with the combined heuristic expands more nodes than with the additive h^2 heuristic alone. This, somewhat surprising, result is further analyzed below.

For random STRIPS problems (figures 6.3(c) and 6.4(c)) the h^2 heuristic outperforms the additive h^2 heuristic, though maximizing the two yields a heuristic clearly more effective than either. Looking at runtime, however, there is no such clear advantage of the combined heuristic: the greater number of nodes expanded with the h^2 heuristic is roughly compensated for by the lower cost of computing and evaluating it. The PDB heuristics are very ineffective in this domain, as was noted also in chapter 4.

Although a sample of three problem domains is really too small as a basis for a theory of the relative effectiveness of the studied heuristics, a few interesting observations can be made. First, and as already mentioned in the discussion concerning the results of the experiment reported in chapter 4, the state variable representations of Blocksworld and 15-Puzzle problems contain comparatively few and large variables (variables with many values) while the state variable representations of random STRIPS problems consist of many small (typically binary) variables.

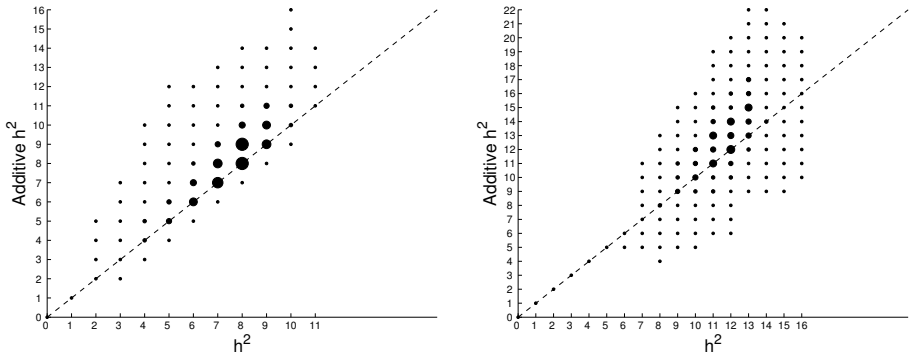
The second characteristic that differs significantly between the three domains is the length of optimal plans and the number of variable value changes during the course of plans. The Blocksworld and random STRIPS problems have relatively short plans

	Blocksworld	15-Puzzle	Random STRIPS
Average # of variables	23.0	32	74.1
...with goal value	16.0	30	7.6
...with secondary goal value(s)	5.8	2	17.5
Average plan length	13.5	47.4	9.5
Average # value changes/variable	1.4	~ 5.9	0.5

Figure 6.5: Number of variables and number of variables with goal values and secondary goal values in the state variable representation, plan length and number of value changes per variable, averaged over the set of all solved instances in each of the three problem domains. The number of value changes per variable for 15-Puzzle problems is an estimate, based on the fact that each action changes four variables.

(averaging 13.5 and 9.5 actions, respectively) and each variable, of those that change at all, changes value only a few times during the course of the plan (the average number of value changes in Blocksworld problems is 1.4, and no variable changes value more than twice, while for the random STRIPS problems the average is 0.5, although some variables change value 2 – 4 times). Also, many of the variables that change do so not because the goal of the planning problem specifies a value for them but because a different value of the variable is required by a precondition of some action in the plan. The number of variables with goal values averages 16.0 of 23.0 for the Blocksworld problems and only 7.6 of 74.1 for the random STRIPS problems. The number of variables with "secondary" goal values (variables that do not have a value specified in the goal of the planning problem but for which some particular value is required by the precondition of some action in the plan) averages 5.8 and 17.5, respectively, for the two domains. Plans for 15-Puzzle problems, by comparison, are much longer (averaging 47.4 actions over the set of instances solved in the experiment) with each variable changing value numerous times during the course of the plan (the exact figure was not determined but given that there are only 32 variables and each action changes 4 of them an estimate is 5.9, although this is probably a bit too high) and 30 of the 32 variables have goal values (the last 2, representing the empty square, have secondary goal values).

In summary, 15-Puzzle problems can be described as having "deep" solutions (each variable changes value several times to reach its goal value) while random STRIPS problems have "wide" solutions (variables change directly to required values, but many variables are required to change values to enable the few that have goal values to change) and Blocksworld problems fall somewhere in between (the numbers mentioned above are summarized in table 6.5). This goes some way towards explaining why PDB heuristics are more effective for 15-Puzzle problems and less effective than (additive) h^2 for Blocksworld and random STRIPS problems, since PDB heuristics take into account all required value changes of variables in the pattern but ignore any required changes outside the pattern while the h^m heuristics are able to capture required changes across all problem variables (*cf.* the discussion of the example



(a) Blocksworld (b) 8-Puzzle

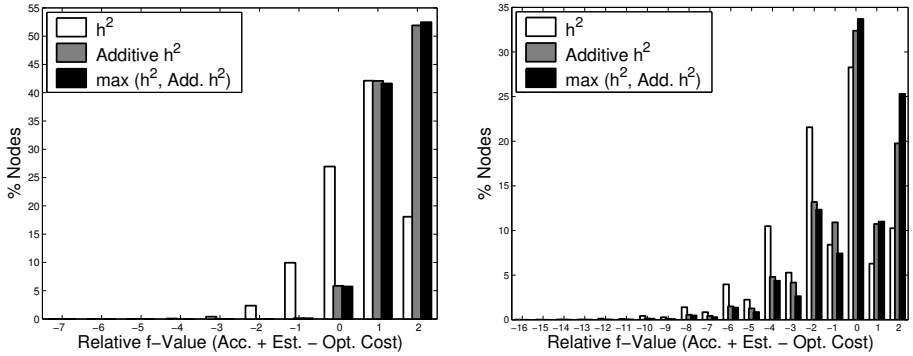
Figure 6.6: Distribution of combinations of h^2 and additive h^2 heuristic values over the set of nodes encountered during search. The size of each point in the diagrams indicates (approximately) the relative frequency of values at that point.

Blocksworld problem of swapping two blocks at the base of a tower in section 4.4 of chapter 4, page 78).

The Effect of Maximizing the h^2 and Additive h^2 Heuristics

As observed above, maximizing the h^2 and additive h^2 heuristics does not always result in a heuristic that is more effective at guiding the search than the additive h^2 heuristic alone, in spite of the fact that the combined heuristic is never less accurate. In fact, it is in general more accurate as the h^2 and additive h^2 heuristics tend to have somewhat complementary strengths. This is illustrated in figure 6.6 which shows the distribution of combinations of values for the two heuristics over the set of states encountered during searches, with the combined heuristic, made in a collection of Blocksworld and 8-Puzzle problems (due to the volume of data, only smaller Blocksworld problems and 8-Puzzle instead of 15-Puzzle problems were used in this experiment).

In the comparative experiment presented above, however, search with the combined heuristic was found to expand more nodes than search with the additive h^2 heuristic alone for about half the 15-Puzzle problems. To explain this apparent contradiction requires a closer examination of how the improved accuracy of the combined heuristic affects the search. Because the A* algorithm expands states strictly in order of increasing total cost (*i.e.*, the sum of accumulated and estimated cost, the so called f-value) and stops when the next state to be expanded is a final state, it expands *every* state with an f-value less than the optimal solution cost but only *some* of the states with f-values equal to the optimal solution cost (no state with an f-value greater than the optimal cost is expanded). Which states with f-values equal to



(a) Blocksworld (b) 8-Puzzle

Figure 6.7: Distribution of the relative f-value (the sum of accumulated and estimated cost, minus optimal solution cost) over the set of nodes encountered during search.

the optimal cost are expanded and which are not is determined by the order of expansion of states with equal f-value, which in principle can change arbitrarily as a result of any change to the relative heuristic values of the successor states of any expanded state. Thus, if the number of states with f-value equal to the optimal cost that are expanded is considered random it is not at all strange that in half the cases it is greater and in half the cases it is less, given that maximizing with the h^2 heuristic does not significantly change the number of such possibly expanded states (which is reasonable considering the weakness of the h^2 heuristic in this domain). The question is rather why in the Blocksworld domain, where maximizing with h^2 also does not significantly reduce the number of node expansions compared to the additive h^2 heuristic alone, the same does not happen.

Part of the reason lies in the fact that maximizing with the h^2 heuristic has very little effect in the Blocksworld domain: on the set of smaller problems used in this analysis it improves the heuristic value of less than 0.02% of the states encountered during search and consequently it also changes the relative heuristic values of the successor states of a very small fraction of the expanded states, while in the 8-Puzzle problems maximizing with h^2 increases the heuristic value of approximately 0.7% of the encountered states and changes the relative values of the successors of slightly less than 0.1% of the expanded states. Because the heuristic guides the exploration of the search space, the greater effect that maximization has in the 8-Puzzle domain quite simply implies a greater degree of rearrangement in the part of the search space that is explored. Another part of the explanation lies in which states have their heuristic values improved by maximization and thus how this improvement translates into improved f-values. Figure 6.7 shows the distributions of the relative f-values (f-value minus optimal cost) calculated with the h^2, additive h^2 and combined heuristics, of states encountered during search (including both states expanded and

states created but not expanded). As can be seen, maximizing the additive h^2 and h^2 heuristics in the 8-Puzzle domain shifts some fraction of states from negative relative f-values (necessarily expanded states) to zero (possibly expanded states) while in the Blocksworld domain both the fractions of states with negative and zero f-values decrease (although the effect is very small). However, even in the 8-Puzzle domain the total fraction of necessarily and possibly expanded states is less when maximization with h^2 is used, which is confirmed by the fact that when A* is modified to search all nodes with f-value equal to the optimal, *i.e.*, to find all optimal solutions, maximization with h^2 does reduce the number of nodes expanded.

7. Improving Heuristics through Search

The h^m family of heuristics, introduced in chapter 3, offers a trade-off between accuracy and computational cost, by varying the parameter m: the higher the value of m the closer the heuristic estimate is to the true cost, but the cost of computing the heuristic grows exponentially with m. For most planning problems, the h^m heuristics exhibit a diminishing marginal gain, in the sense that the higher m already is the smaller is the improvement in heuristic accuracy brought by raising it further, a fact that was also experimentally demonstrated in chapter 3 (page 46). This combines to make the heuristic cost effective, in the sense that it reduces search time more than the time taken to compute it, only for small values of m (typically $m \leqslant 2$). Yet, for many planning problems the h^2 heuristic is too weak.

However, the exponentially increasing computational cost and the diminishing improvement are not absolutely inherent in the definition of the h^m heuristic but are both to some extent due to the method used to compute the heuristic, specifically the fact that it computes a *complete* solution to the set of equations that define the heuristic (comprising h^m values for every search state of size m or less). First, the size of the complete solution grows exponentially with m, which explains the exponentially increasing computational cost. Second, the h^m value of a search state s (of size greater than m) is the maximum of the values of any size m subset of s and therefore the computed values of small sets contribute to the evaluation of a greater number of search states than the heuristic values of large states (which are more specific). As m increases the complete solution includes an increasing fraction of larger atom sets, and a smaller fraction of the solution is thus relevant to evaluating states actually encountered in the search.

This chapter presents two alternative methods, both based on search, for computing h^m heuristics: *relaxed search*, which computes a *partial* solution to the h^m equation, and *boosting*, which improves the accuracy of an h^m solution "pointwise". Intuitively, the idea behind both methods is to focus effort on computing the part of the heuristic that is relevant to the planning problem at hand, and thus to avoid the two problems described above. Relaxed search (described in the next section) exploits the fact that the h^m heuristic is also the optimal cost in a relaxed search space and computes partial solutions by searching this space and recording information discovered during the search. As computing a partial solution is (sometimes) less computationally expensive the method can be applied for higher values of m, resulting in a more accurate heuristic. Boosting (described in section 7.2) instead makes use of the fact that each size m atom set is itself a search state and improves the estimated cost of selected atom sets by performing a limited search starting from the corresponding

state. Results of the three methods (in the form of complete or partial h^m solutions, or isolated improved heuristic values) are combined by storing them all in the heuristic table. As described in chapter 3 (page 42) the heuristic table, which stores the computed h^m solution and which is used to perform the on-line evaluation of search states, can be implemented as a general mapping from states (sets of atoms) to associated heuristic values. Because the relaxed search and boosting methods are both based on search, a vital concern is to make use of any heuristic information already computed to reduce the cost of this search.

Heuristics derived by searching in an abstraction of the search space have been studied extensively in AI (*e.g.* by Gaschnig, 1979, Pearl, 1984, Prieditis, 1993, and in the works on pattern database heuristics mentioned in chapter 4) and the chapter concludes with a discussion of related ideas (section 7.3).

7.1 Relaxed Search

This section introduces the relaxed search method of computing partial h^m heuristics, based on the idea of viewing the h^m relaxation as a relaxation of the search space rather than the equations characterizing the cost function. Arguably, relaxed search is most interesting in the case of temporal planning where the cost relaxation method from the last chapter can not be applied, but since there are some complications in applying the method to temporal regression planning it is first described for the sequential case, followed by the modifications required in the case of temporal planning (on page 123). The section concludes with a more extensive analysis of the cost effectiveness of relaxed search for temporal planning (page 124).

The m-Regression Search Space and its Relation to the h^m Heuristic

As explained in chapter 3 the h^m relaxation (equation (5) on page 39) can be viewed as a change of search space. The relaxed search space, which we will call the *m-regression space*, is an AND/OR (or "min/max") space: any state s with more than m atoms is a "max state", whose successors are the size m subsets of s and whose cost is the max of the successor costs, while states s with m or fewer atoms are "min states" and are regressed as normal. A max (or AND) state is solved only if all successors are solved, while a min (or OR) state is solved if some regression leads to a solution. Figures 7.2 – 7.2 show (part of) the 2-relaxed search tree for an example planning problem (the example is described in detail on page 119). As can be seen in the examples, the search space is not strictly layered in the sense that OR-nodes may have successors that are also OR-nodes.

The h^m heuristic is the optimal cost function for the m-regression space, *i.e.*, the h^m value of any search state equals the optimal m-relaxed solution cost, which can be found by searching in the m-regression space. In fact, the optimal m-relaxed solution to a search state s, which in an AND/OR search space has the form of a tree rather than a path, gives the optimal cost of many more states than s (the same holds in

a normal "OR-space": an optimal solution path gives the optimal cost of all states along the path).

The relaxed search method exploits this correspondence to compute a partial h^m solution by searching in the m-regression space, starting from the state consisting of the goals of the planning problem. This makes the part of the h^m solution that is computed likely to be the part most relevant for solving the planning problem. The search is carried out using a heuristic search algorithm called IDAO*, which, as the name suggests, is an adaptation of the IDA* search algorithm to AND/OR spaces (Haslum, 2004; the algorithm is described on page 117 below). For any $m' \leqslant m$, $h^{m'}$ is an admissible heuristic also for search in the m-regression space. This follows from the fact that $h^{m'}(s) \leqslant h^m(s)$, for all s, when $m' < m$, which was shown in chapter 3 (page 43). The IDAO* algorithm has the important property that it discovers the optimal cost of every state that is solved (this includes the states appearing in the solution tree, but may also include other states) and a lower bound greater than the cost estimate given by the heuristic used in the search for every state that is expanded but not solved (this property is not unique to IDAO*: alternative algorithms are discussed in section 7.3).

The h^m values (and lower bounds) discovered during the search are combined, by maximization, with the heuristic used to guide the search, resulting in a more accurate heuristic. This can be done using the same generalized heuristic table both to store improved values and to evaluate states in the relaxed search (as described in chapter 3, page 42). Thus, the heuristic used to guide the search in the m-regression space is actually the maximum of several (complete and partial) $h^{m'}$ solutions for different values of m' (but all smaller than, or equal to, m). The maximization is automatic, since the heuristic value of a state is simply the maximum value of any subset of the state that is stored in the table. In particular, heuristic values discovered and stored during the relaxed search become immediately available for use in subsequent state evaluations so that the part of the h^m solution computed by the part of the search that is already done is also used for the part of the search that remains. An example of this can be seen in the search tree in figure 7.3: the search state {`(at p2)`, `(at p0)`}, with an initial heuristic estimate of 3, is expanded in the left-most branch of the tree and as a result its estimated cost is increased to 4. When the same state appears in a later branch its heuristic estimate is the new value of 4 and since this exceeds the cost bound at this point in the search the state is not expanded as it would otherwise have been.

A final important point to note is that transposition elimination rules for regression planning described in chapter 2 (commutativity cuts for sequential planning, on page 11, and right-shift cuts for temporal planning, on page 20) can not be applied in the relaxed search. Doing so would endanger admissibility, because the elimination rules make the set of successor states to a search state (and therefore the optimal cost of the state) dependent on the path by which the state was reached, and thus a lower bound on the cost of a state reached through one path is not necessarily a lower bound on the cost of the same state reached through a different path. Unless the path (or the relevant features of the path) is not used along with the state to

retrieve values stored in the heuristic table it then becomes possible to overestimate cost. The fact that transposition elimination rules are not used when searching the m-regression space in some cases has a significant impact on the efficiency of the relaxed search, as shown in the experimental evaluation later in this chapter (page 124).

Properties of the Resulting Heuristic

Properties of the heuristic resulting from relaxed search depend, of course, on how the search is carried out. For the heuristic to be admissible, the initial heuristic (the one being improved by the relaxed search) must be admissible and the algorithm used to search the m-regression space must store only values that are optimal, or lower bounds on optimal, costs. The IDAO* algorithm satisfies this requirement.

Theorem 8 If the initial heuristic is admissible and the values discovered and stored during the relaxed search are (lower bounds on) optimal solution costs then the resulting heuristic is admissible.
Proof: This follows directly from the fact that optimal solution cost in the m-regression space equals the h^m heuristic estimate, which is admissible, and that the maximum of any two admissible heuristics is also admissible. □

It is also easily seen that when m is sufficiently large there will not be any AND-nodes in the m-regression space, since AND-nodes are defined as states of size greater than m, and in this case the m-regression space is identical to the normal regression space (analogous to the fact that $h^m = h^*$ for any sufficiently large m, shown in chapter 3, page 43).

Consistency of the resulting heuristic is more complicated to show, since it depends on the workings of the IDAO* algorithm. It also holds only for sequential planning, because of certain simplifications made in the temporal case (described on page 123).

Theorem 9 For sequential planning, provided that the heuristic used to guide the relaxed search is consistent, the heuristic resulting from values discovered by IDAO* is consistent.
Proof: Let s be a regression search state and s' a successor state obtained by regressing s through action a, and let h and h' denote the heuristic (defined as the maximum value of any subset stored in the heuristic table: see chapter 3, page 42) before and after the relaxed search, respectively. Note that $h'(s) \geqslant h(s)$, since any updates made in the relaxed search can only increase the heuristic value. First, if s has not been expanded in the relaxed search, $h(s) = h'(s)$ and consistency of h' follows from the consistency of h and the fact that $h'(s') \geqslant h(s')$. Assume s has been expanded during the relaxed search. If $|s| \leqslant m$, s is expanded by normal regression in the relaxed search and $h'(s)$ equals the minimum $h'(s'') + c_{s,s''}$ over all successor states s'' of s, which include s', so $h'(s') + c_{s,s'}$ can not be less than $h'(s)$ (recall that $c_{s,s''}$ is the transition cost from s to s'', *i.e.*, the cost of the action used to regress s to s'' in the case of sequential planning). If $|s| > m$, s is an

AND-node in the m-regression space and $h'(s) = h'(s'')$ for some size m $s'' \subset s$, which implies that s'' was also expanded during the relaxed search (see the IDAO* algorithm in figure 7.1). Since s' is obtained by regressing s through a, there are two possibilities: either a adds some atom in s'', or it does not. In the first case, a is applicable also to s'', which means $h'(s'') \leqslant h'((s'' - add(a)) \cup pre(a)) + cost(a)$, since $h'(s'')$ has been updated to the minimum over all regressions of s'', and since $s'' \subset s$, $(s'' - add(a)) \cup pre(a) \subset (s - add(a)) \cup pre(a) = s'$ so $h'((s'' - add(a)) \cup pre(a)) \leqslant h'(s')$, from which follows that $h'(s) = h'(s'') \leqslant h'(s') + cost(a)$. In the second case, $s'' \subset s'$ which implies that $h'(s') \geqslant h'(s'') = h'(s)$ and thus that $h'(s) \leqslant h'(s') + cost(a)$, which shows the consistency of h'. □

IDAO*

IDAO* is an adaptation of IDA* to searching AND/OR graphs (Haslum 2004). Like IDA*, it carries out a series of cost-bounded depth-first searches with increasing cost bound. IDAO* is admissible, in the sense that if guided by an admissible heuristic, it returns the optimal solution cost of the starting state. In fact, it finds the optimal cost of every OR-node that is solved in the course of the search. However, it does not keep enough information for the optimal solution itself to be extracted so it can not be used to find solutions to AND/OR search problems. It works for the purpose of improving the heuristic since for this only the optimal *cost* needs to be known.

The algorithm is sketched in figure 7.1. The main difference from IDA* is in the DFS subroutine: when expanding an AND-node it recursively invokes the main procedure IDAO*, instead of the DFS function. Thus, for each successor to an AND-node the algorithm performs a series of searches with increasing cost bound, starting from the heuristic estimate of the successor state (which for some successors may be smaller than that of the AND-node itself) and finishing when a solution is found or the cost bound of the predecessor AND-node is exceeded. This ensures that IDAO-DFS is never called with a state and a cost bound exceeding the optimal solution cost of that state and thus that the cost returned is always a lower bound on the optimal cost of the expanded state, equal to the optimal cost if the state is solved. As described above, the updated costs of OR-nodes are stored in the heuristic table, and the same table is used to evaluate states in the search. In this way the partial h^m solution that is discovered by the search is combined with the previously computed heuristic and the result becomes immediately available for use in subsequent heuristic evaluations. IDAO* stops searching the successors of an AND-node as soon as one is found to have a cost greater than the current bound, since this implies the cost of the AND-node is also greater than the bound. However, since the algorithm performs repeated depth-first searches with increasing bounds, remaining successors of the AND-node will eventually also be solved. When an m-solution has been found, all successors to every AND-node appearing in the solution tree have been searched, and their updated costs stored.

Because the successor nodes of AND-nodes are subsets, IDAO* frequently encounters

```
IDAO*(s, b)  // search root and cost bound
{
  solved = false;
  current = h(s);
  while (current < b and not solved) {
    current = IDAO_DFS(s, current);
  }
  return current;
}

IDAO_DFS(s, b)
{
  if final(s) {
    solved = true;
    return 0;
  }
  if (s stored in SolvedTable) {  // short-cut search if s already solved
    solved = true;
    return stored solution cost;
  }
  if (|s| > m) {  // AND-node
    for (each subset s' of s such that |s'| <= m) {
      new cost of s' = IDAO*(s', b);  // call IDAO* with cost bound b
      if (new cost of s' > b) {  // s' not solved
        return new cost of s';
      }
    }
    solved = (all subsets solved);
    new cost of s = max over all s' [new cost of s'];
    if (solved) {
      store (s, new cost of s) in SolvedTable;
    }
    return new cost of s;
  }
  else {  // OR-node
    for (each s' such that R(s,s',c(s,s'))) {
      if (c(s,s') + h(s') <= b) {
        new cost through s' = c(s,s') + IDAO_DFS(s', b - c(s,s'));
        if (solved) {
          new cost of s = new cost through s';
          store (s, new cost of s) in SolvedTable;
          return new cost of s;
        }
      }
      else {
        new cost through s' = c(s,s') + h(s');
      }
    }
    new cost of s = min over all s' [new cost through s'];
    T(s) = new cost of s;  // store new cost in the heuristic table;
    return new cost of s;
  }
}
```

Figure 7.1: The IDAO* algorithm (with solved table). `R` denotes the transition relation of the search space, *i.e.*, `R(s,s',c(s,s'))` iff there is a transition from `s` to `s'` with cost `c(s,s')`. `h` is the heuristic defined by current contents of the heuristic table, `T`.

the same state (set of goals) more than once during search. The algorithm can be sped up, significantly, by storing solved nodes (both AND-nodes and OR-nodes) together with their optimal cost and short-cutting the search when it reaches a node that has already been solved. This is done in the "solved table" referred to in figure 7.1. In difference to the lower bounds stored in the heuristic table, which are valid also in the m'-regression search space for any $m' > m$ as well as in the original search space, the information in the solved table is valid only for the current m-regression search (since states of size m', for $m' > m$ are relaxed in the m-regression space but not in m'-regression). An example can be found in figure 7.3: the state {`(at p2)`, `(airborne)`} is solved with a cost of 3 (at the root the subtree drawn in bold) and appears again in a different branch of the tree. If the fact that the node is solved has been recorded, the same solution does not have to be found again.

Note that a standard transposition table (Reinfeld & Marsland 1994) which records updated cost estimates of unsolved nodes is of no use in IDAO* since this role is played by the heuristic table: estimated cost of OR-nodes are stored in the heuristic table and the heuristic estimate of an AND-node is always given by the maximum of its size m subsets. The solved table complements the information in the heuristic table by indicating when the heuristic value of a state is equal to its optimal cost and not only a lower bound. Since the solved table is only a speed-up technique it is not necessary to store every solved node, so the table can be limited in size and implemented efficiently (*e.g.* as a closed hash table).

An Example

For an illustration of how relaxed search using the IDAO* algorithm proceeds, consider a simplified instance of the single UAV mission planning problem (this problem was introduced in chapter 3, page 47, for the experimental analysis of the accuracy/cost trade-off in the h^m heuristics; here, we simplify the problem by assuming all actions have unit cost). The UAV is initially on the ground (atom `(on_ground)`) at point `p0`, and the goal is to have images from two points (`(img p2)` and `(img p8)`), and to have the UAV back on the ground. To take an image, the UAV needs to be at the point (represented by atom `(at ?p)`), and to fly to a point it must be airborne (atom `(airborne)`). To land, the UAV must be airborne and at a landing point (`p0` and `p1` in this example).

To keep the size of the example manageable, assume a complete solution has been computed only for h^1 and that relaxed search is used to compute a partial h^2 solution. Figures 7.2 – 7.4 show (part of) the 2-regression space explored by the first, second and third iteration, respectively, of an IDAO* search starting from the problem goals.

In the first iteration (figure 7.2) IDAO-DFS is called with a cost bound of 4, as this is the estimated cost of the starting state given by the precomputed h^1 heuristic. The root node is an AND-node, so when it is expanded IDAO* is called for each size 2 subset. This recursive call to IDAO* is made with a cost bound equal to the current IDAO-DFS bound (4). The first such subset to be generated is {`(img p2)`, `(on_ground)`}. This state also has an estimated cost of 4, so IDAO-DFS

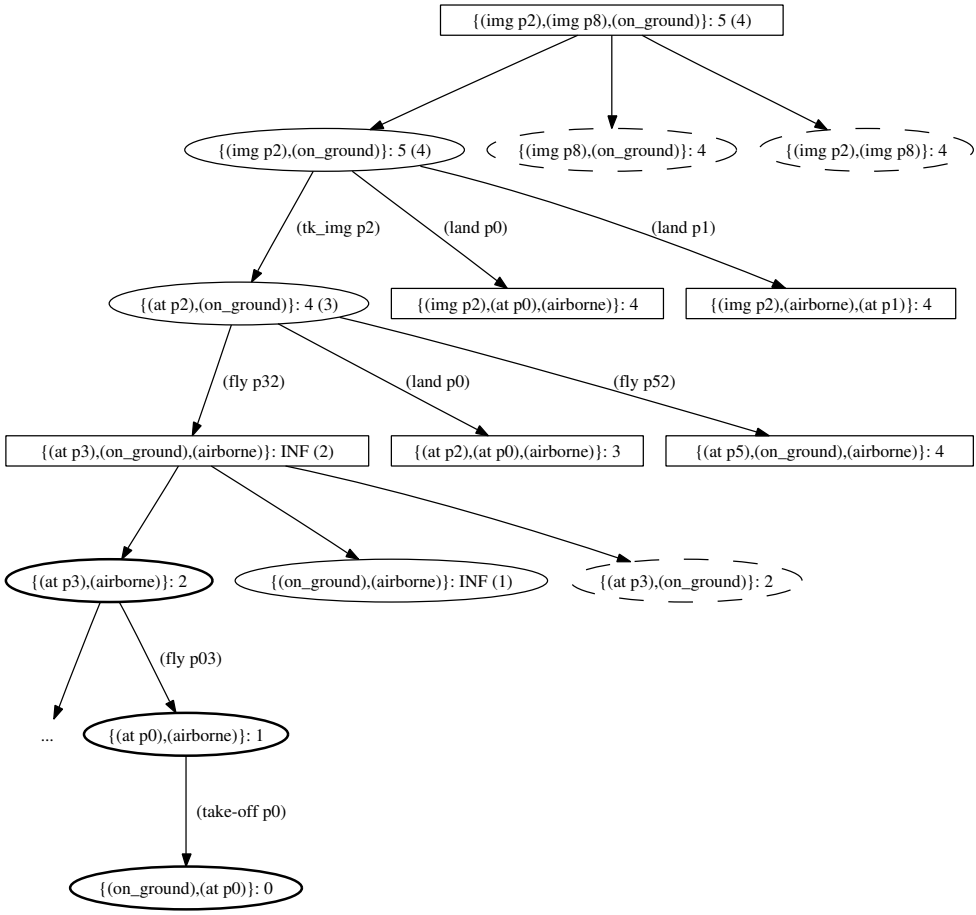


Figure 7.2: Part of the 2-Regression tree for the example UAV planning problem expanded in the first IDAO* DFS iteration (with a cost bound of 4). AND-nodes are depicted by rectangles, OR-nodes by ellipses. Nodes drawn in bold are solved, while nodes drawn with a dashed outline are not expanded in this IDAO* DFS iteration due to the updates caused by their siblings (the expansion of child nodes proceeds left to right). For nodes whose estimated cost has been updated after expansion, the (h^1) estimate before expansion is given in parentheses.

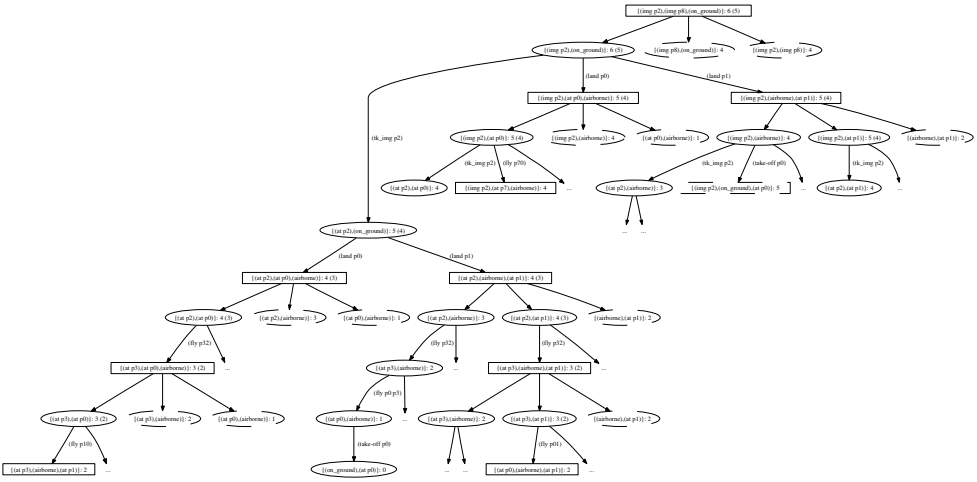


Figure 7.3: Part of the tree expanded in the second IDAO* DFS iteration (cost bound of 5). Note that the estimate before expansion (the value in parentheses) here includes the updates made in the previous iteration.

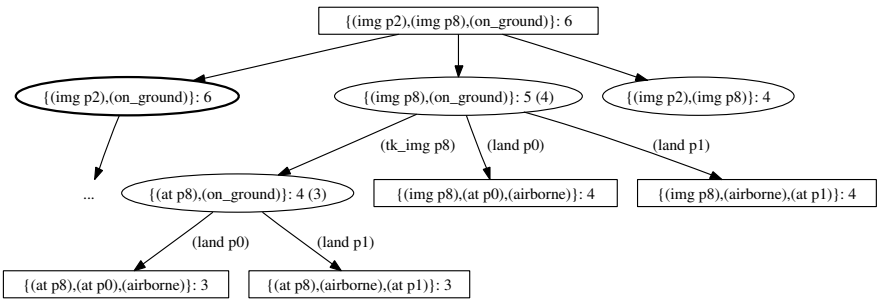


Figure 7.4: Part of the tree expanded in the third IDAO* DFS iteration (cost bound of 6), but showing only the first iteration of the recursive IDAO* call to the second node below the root.

is called with this bound in the first iteration (since 4 also happens to be the cost bound of the recursive IDAO* call, this is the only iteration that will be made). From this node search proceeds like a normal IDA* search, expanding nodes within the cost bound by regression, until it reaches the state {`(at p3)`, `(on_ground)`, `(airborne)`}, which is an AND-node with an estimated cost of 2. This AND-node is again expanded by calling IDAO* for each size 2 subset, with the current cost bound (2). The first subset, {`(at p3)`, `(airborne)`}, has an estimated cost of 2 and is solved within that bound. The second subset, {`(on_ground)`, `(airborne)`}, has an estimated cost of 1, but when it is expanded it is discovered that there are no possible regressions of this state so its cost is updated to $+\infty$ (the new cost is stored in the heuristic table). Since this exceeds the cost bound of the recursive IDAO* call this call returns, and since the estimated cost of the parent AND-node is now above the bound with which IDAO-DFS was called (2) this call also returns, without searching the last subset. The remaining regressions of the two expanded OR-nodes lead to states with estimated costs above their respective bounds, so these are not searched. The costs of the two OR-nodes are updated to the minimum cost of their child nodes (action cost plus the estimated cost of the child node). Since this is enough to raise the estimated cost of the root AND-node over the cost bound of the initial call to IDAO-DFS (4) the remaining subsets of this node are not searched.

In the second iteration (figure 7.3) IDAO-DFS is called with a cost bound of 5. This iteration proceeds like the first, but this time the branch leading to the state {`(at p3)`, `(airborne)`, `(on_ground)`} is not examined, since its already known to be unsolvable. Instead, other regressions of the two OR-nodes expanded but unsolved in the last iteration are now within the cost bound, and therefore expanded. These lead to AND-nodes, and the recursive invocation of IDAO* on the size 2 subsets as before. The cost of the state {`(at p2)`, `(at p0)`}, in the left-most branch is updated from 3 to 4 which increases the cost of the parent AND-node above the bound so that remaining subsets are not searched. Note that when the state {`(at p2)`, `(at p0)`} reappears (to the left in the second branch of the first OR-node below the root) its estimated cost is the new value 4 and the state is therefore not expanded again. The search proceeds in a similar fashion and when the iteration finishes the estimated cost of the first subset of the root AND-node has increased to 6.

In the third iteration, the first subset of the root AND-node is solved, with a cost of 6, and search proceeds to the next subset, {`(img p2)`, `(on_ground)`}. This state has an estimated cost of 4 so the first iteration of the recursive IDAO* call is made with cost bound 4, but note that the cost bound of the recursive IDAO* call is 6, the estimated cost of the parent AND-node. The first iteration of the recursive IDAO* call is shown in figure 7.4: during this iteration, the cost of the state is updated to 5. Because this is still below the cost bound of the IDAO* call (6) more iterations are made until either the state is solved, in which case the last subset of the root is also searched, or the cost increases above the bound of the recursive IDAO* call.

The process continues until all the size 2 subsets of the root state have been solved, at a cost of 7. At this point, improved heuristic estimates of roughly one quarter of the size 2 atom sets have been stored in the heuristic table.

Relaxed Temporal Regression

As with the h^m heuristic itself, adapting relaxed search to temporal regression planning, although straightforward in principle, is somewhat complicated in its details. The crux is how to define which states are AND-nodes in the temporal m-regression space and what their successor states are. There are at least two ways that this can be done.

Recall from chapter 3 that an additional relaxation, defined by equations (8) – (9) on page 51, was applied and that the cost of a temporal regression state $s = (E, F)$ is estimated by the maximum of a set of states without scheduled actions, *i.e.*, of the form $(E', \emptyset)$ where $E' \subseteq E \cup \bigcup_{(a,\delta)\in F} pre(a)$ (and $|E'| \leqslant m$). The first alternative is to define only states of this form to be OR-nodes in the temporal m-regression space (and the successors of AND-nodes correspondingly as all $(E', \emptyset)$ as above). This definition, however, leads to a higher density of AND-nodes, each with a large number of successor states, in the m-regression space. These factors both contribute to make the relaxed search computationally more expensive, as shown in the experimental analysis in the next section.

The second alternative is slightly less restrictive, in the sense that some states with a non-empty F component are regarded as OR-nodes. Define the size of a state $s = (E, F)$ as

$$|(E, F)| = \left| E \cup \bigcup_{(a,\delta)\in F} pre(a) \right| \tag{15}$$

and s as an AND-node in the temporal m-regression space iff $|s| > m$, with successor states being all size m subsets of the atom set on right hand side of equation (15). This corresponds to relaxing temporal regression states using only equations (9) and (5). This, however, gives rise to another difficulty: a state with a non-empty F component can not be stored in the heuristic table (which maps only atom sets to associated costs) and neither can the optimal cost or lower bound discovered for such a state be stored as the cost of the corresponding atom set (right hand side of equation (15)) since the optimal cost of achieving this atom set can be lower (recall that equation (9), page 51, is in fact an inequality). However, a plan that achieves $E \cup \bigcup_{(a,\delta)\in F} pre(a)$ at time t also achieves the state (E, F) at most $\max_{(a,\delta)\in F} \delta$ time units later, through inertia, *i.e.*,

$$h^*(E, F) \leqslant h^*\left(E \cup \bigcup_{(a,\delta)\in F} pre(a), \emptyset \right) - \left(\max_{(a,\delta)\in F} \delta \right) \tag{16}$$

Thus, to maintain the admissibility of the heuristic defined by the values stored during relaxed search the largest δ among all actions in F is subtracted from the discovered cost of the state before it is stored (this modification does not affect the states that are immediate successors of AND-nodes since they have, by definition, an empty F component).

While this definition leads to fewer AND-nodes, and AND-nodes with fewer successor states, in the temporal m-regression space it also has some drawbacks: subtracting the largest δ value weakens the heuristic values discovered by relaxed search and, what is worse, since a cost under-approximation is applied when storing states containing scheduled actions, but not during the search, the heuristic resulting after relaxed search can be inconsistent. Also, as mentioned earlier, right-shift cuts can not be used in the relaxed search, since the discovered cost values could then be inadmissible.

Analysis: Cost Effectiveness of Relaxed Search

This section presents an experimental analysis of the cost effectiveness of the use of relaxed search as a method of enhancing h^m heuristics. The main result is a characterization, in terms of certain features of the planning problem, of the conditions under which relaxed search can be expected to be cost effective. These features are related to the frequency of AND-nodes in the m-regression search space and the relative computational cost of expanding OR-nodes in the m-regression space as compared to the normal regression space. The characterization is somewhat weak: it can not quantitatively predict the performance of a planner using relaxed search relative to one that does not, and in at least one problem domain there are other, as yet undetermined, factors involved. The analysis does however provide some insight into the factors that can potentially make relaxed search ineffective, leading to some ideas for improvement. These are discussed more at length in section 7.3.

Before describing the experiment and its results, in the first section below, we discuss some theoretical results on the cost effectiveness of heuristics based on search and provide an intuitive picture of how relaxed search is intended to work. The next section describes how relaxed search is integrated in the planning system used in the experiment (page 126). The following section presents the problem domains on which the experiment was carried out and an overview of the results (page 127), and after this the domains where relaxed search fails to be effective are examined in more depth. The findings are summarized in the last section (page 137).

Can Relaxed Search be Cost Effective?

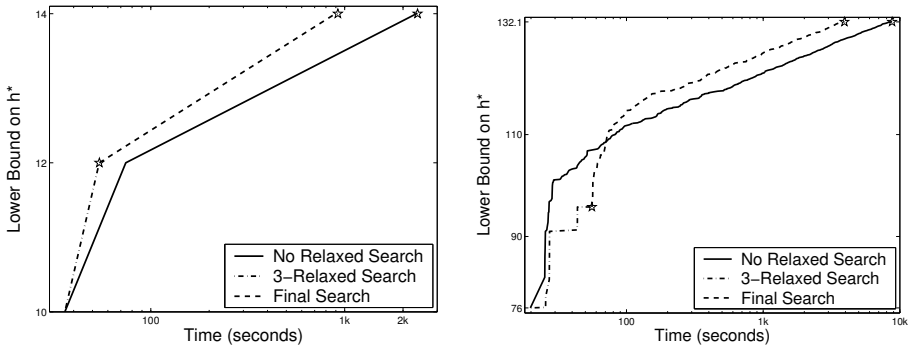
For small values of m it is normally more efficient to compute a complete h^m solution using the GBF algorithm, as described in chapter 3. The experimental analysis in chapter 3 indicated that the breakpoint typically lies at $m = 2$ and also demonstrated that an advantage of the complete h^2 heuristic is that it often detects binary at-most-one invariants (mutexes). While relaxed search can also find such invariants (as illustrated in the example above by, for example, the atom set {`(on_ground)`, `(airborne)`} in figure 7.2) it is less likely to find all of them. For example, in the 2-regression trees shown in figures 7.2 – 7.4 there are many atom sets of the form {`(at ?p0)`, `(at ?p1)`} that are mutexes but that are not discovered by the 2-relaxed search until the tree has been expanded to such a depth that cycles appear along every branch, something that typically only happens at a cost bound

much greater than the cost of the relaxed solution.

For these reasons, relaxed search is interesting mainly as a method of improving on the h^2 heuristic, by computing partial h^m solutions for $m = 3, \ldots$ and maximizing the results. Thus, a first condition for relaxed search to be cost effective is that the heuristic improvement results in a reduction of search effort (in other words, that the resulting heuristic is more accurate than h^2 *and* that h^2 is not good enough for the problem at hand). Second, the computational cost of the relaxed search, relative to this reduction, must not be too large.

The generalized theorem of Valtorta, by Holte *et al* (1996), states that in the course of an A* search guided by a heuristic derived by searching blindly in some relaxation of the search space, every state that would be expanded by a blind search in the original search space must be expanded either in the relaxed space or by the A* search in the original. In particular, if the relaxation is an embedding (the set of states in the relaxed space is the same as in the original search space) such a heuristic can not be cost effective (Valtorta 1984). Although the theorem has not been shown for the case of IDA* search, the fact that the m-relaxation of the regression planning search space is an embedding (since every state in the normal regression space corresponds to exactly one state, containing the same set of subgoal atoms, in the m-regression space) may invite some scepticism as to the potential cost effectiveness of the relaxed search method. However, relaxed search can be cost effective (as demonstrated by the two examples below, and the following experiment), which can be explained by the following facts: First, the algorithm used to search the m-regression space discovers (and stores in the heuristic table) the true h^m value, or a lower bound on this value greater than that given by the current heuristic, for every OR-node expanded during the course of the relaxed search. The AND/OR structure of the m-regression space and the fact that the general heuristic evaluation procedure makes use of all relevant information present in the heuristic table together imply that an improvement of the estimated cost of an OR-node may yield immediately an improved estimate of the cost of many states (all states that are supersets of the improved state) in both the relaxed and the normal regression space, without any additional search effort. Second, each OR-node expansion in the m-regression space is likely to be computationally cheaper than the average in the normal regression space. This is because the number of successors generated when regressing a state generally increases with the number of subgoal atoms in the state and OR-nodes in the m-regression space are states of limited size.

The experiment described below compares a planning system that uses relaxed search to improve the h^2 heuristic to one that does not (the two planning systems are described in detail in the next section). Both systems use iterative deepening search, both for the relaxed search (the IDAO* algorithm) and search in the normal regression space (the IDA* algorithm), so that there is a best known lower bound on the optimal solution cost which gradually increases during the course of the search (until it reaches the actual solution cost and a solution is found). Thus, the lower bound on solution cost can be seen as a function of the search effort invested (or search time) and the shape of this function indicates the effectiveness of the search. Figure



(a) Pipesworld (b) Multi-UAV Planning

Figure 7.5: Evolution of the lower bound on solution cost during relaxed and final search in two example problems: (a) problem #8 from the Pipesworld domain and (b) a multi-UAV mission planning problem with two UAVs. Stars indicate where (relaxed and real) solutions are found. Note that the time scale is logarithmic.

7.5 illustrates with two example problems for which relaxed search is cost effective. In the first example (problem #8 from the Pipesworld domain) the 3-relaxed search reaches a solution (with cost 12) faster than the normal regression search discovers that there is no solution within the same cost bound and the final (non-relaxed) regression search with the resulting heuristic is also faster than the search with only h^2 (as indicated by the slope of the curve). In the second example (a multi-UAV problem) it takes more time to find a 3-relaxed solution than to reach the corresponding lower bound in normal search, but the final search with the improved heuristic is so much faster that the total time (for the relaxed and final search) is still less than that for searching with only the h^2 heuristic.

Integrating Relaxed Search in a Heuristic Search Planner

The heuristic search planner incorporating relaxed search consists of three main stages: the first computes the h^2 heuristic, completely, using the GBF algorithm, the second performs a series of m-relaxed searches, for $m = 3, \ldots$, in order to improve the heuristic, and the last is a search in the normal (temporal) regression space guided by the computed heuristic, which is then the combination of the complete h^2 solution and one or more partial h^m solutions, for $m = 3, \ldots$ (the last stage is referred to as the "final search" in the following descriptions). The m-relaxed searches are done by IDAO* and the final search is done using the standard IDA* algorithm (with standard enhancements, *i.e.*, right-shift cuts and a transposition table).

In the procedure outlined above, relaxed searches are carried out for increasing m until some stopping condition is satisfied. If the last m-regression search finishes without encountering any AND-node, so that the relaxed solution is in fact a solution

to the original problem, there is obviously no reason to continue with further relaxed searches, but it is in most cases much too computationally expensive to pursue relaxed searches up to this point. The basic configuration of the planning system used in this experiment is to perform only 3-relaxed search. In one problem domain, an alternative configuration is also tested: this configuration performs relaxed searches for increasing m until the cost of the m-solution found is the same as that of the $(m-1)$-solution (or h^2 heuristic estimate, for $m = 3$). This is a reasonable, but by no means perfect, indicator that relaxed searches for higher m will not yield additional heuristic improvement. Another alternative, though one not used in this experiment, is of course to limit relaxed searches to a certain amount of time, number of expanded nodes, or some other measure of search effort.

To measure the impact that relaxed search has on the performance of this planning system, with a minimum of noise, it is compared to a planner that is identical except that it does not use relaxed search, *i.e.*, that consists simply of only the first and last stages in the above procedure: computing h^2 by GBF and searching by regression using IDA* (with the same enhancements as in the final search). The two systems share implementations of all common parts.

Experiment and Results Summary

The two planners are compared on temporal planning problems from seven different domains (the Promela and PSR domains are actually parallel rather than temporal problems, *cf.* chapter 2, page 21). Six of these (the Airport, Pipesworld, Promela, PSR and UMTS domains) are the problem domains of the most recent International Planning Competition. and the last is the multi-UAV mission planning problem introduced in chapter 2 (page 13). The Airport domain models movements of aircraft on the ground at an airport and the goal is to guide arriving aircraft to parking positions and departing aircraft to suitable runways. The Pipesworld domain is also a transportation problem, concerning batches of petroleum products in a pipeline network. The Promela domain is a translation of a model checking (deadlock detection) problem into STRIPS (the problems used in the experiment are from the "dining philosophers" subset). The PSR domain models a power network reconfiguration problem. The Satellite domain is about scheduling astronomical observation tasks on a collection of satellites. The UMTS domain models the call set-up procedure for data applications in mobile telephones, and is also a scheduling problem. Detailed descriptions of the IPC domains are given by Hoffmann, Edelkamp *et al.* (2004; the problem collection can also be found at `http://ipc.icaps-conference.org/`).

For the Satellite domain, a set of additional problems were generated for this experiment, with the random problem generator used to create the problems in the IPC collection (the problem generator is also available at `http://ipc.icaps-conference.org/`). The set comprises 80 additional problems, 10 random variations for each of the controllable parameter settings of the 8 smallest instances in the IPC set. This was done in an attempt to control the very large impact that the random elements in the problem generation procedure have on the difficulty of the resulting problems.

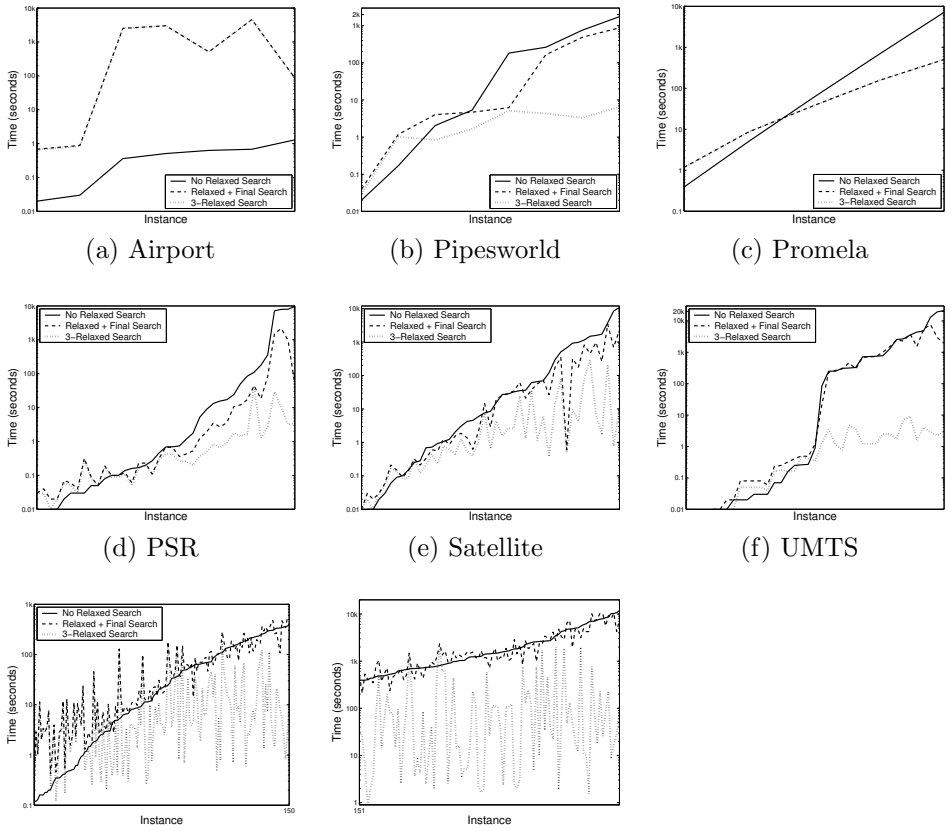
It could not be done for the other IPC domains because problem generators were not available.

When using IDA* with temporal regression, the cost bound tends to increase by the gcd (greatest common divisor) of action durations in each iteration, except for the first few iterations[4]. In problems of the Satellite domain, action durations differ by very large amounts and are at the same time specified with a very high resolution (for example, in problem #2 one action has a duration of 1.204 and another a duration of 82.99) and as a result the gcd of action durations is typically very small (on the order of $\frac{1}{100}$). Combined with the fact that the h^2 heuristic is quite weak in this domain, which means the difference between the initial heuristic estimate of the solution cost of a problem and the actual optimal cost is often large, this results in an almost astronomical number of IDA* iterations being required before a solution is found. To avoid this (somewhat artificial) problem, action durations in this domain were rounded up to the nearest integer. This increases the makespan of the plans found, but not very much – on average by 2.9%, and at most by 5.9% (comparison made on the problems that could be solved with original durations). Optimality can be restored by a two-stage optimization scheme, in which the makespan of the non-optimal solution is taken as the initial upper bound in a branch-and-bound search, using the original action durations.

The set of multi-UAV mission planning problems comprises 350 problems. In all problems there are two UAVs and two buildings. The buildings and the number and selection of observation goals vary and in roughly half the problem the goal also requires both UAVs to be back on the ground. The two planners were run on each problem with a time limit of $4 - 8$ hours of CPU time, varying with domain. This limit exists mainly for practical reasons: since both planners are complete and all problems are solvable, both planners will eventually find a (optimal) solution to every problem instance. The limit was chosen so as to generate sufficient data for each of the domains considered.

Results are summarized by problem domain in figure 7.6. Each graph shows the solution time, per problem instance, with no relaxed search (*i.e.*, search with the plain h^2 heuristic) and solution time with the h^2 heuristic improved by 3-relaxed search, including the time required for the 3-relaxed search. The time for the 3-relaxed search alone is also shown (in figures (a) and (c) it is not visible, since in these two domains the time in the final search is so small that they practically coincide). Instances in each domain are sorted by the solution time with no relaxed search, which gives a rough order of problem difficulty. Only instances solved by both planners within the time limit are shown. In the Pipesworld, Promela, PSR

[4]The planner implementations treat action durations as rationals: by the gcd of two rationals a and b is meant the greatest rational c such that $a = mc$ and $b = nc$ for integers m and n. Note that the planners do *not* compute the gcd of action durations and use this to increment the cost bound (as is done in *e.g.* the TGP planner; Smith & Weld, 1999). The bound is in each iteration increased to the cost of the least costly node that was not expanded due to having a cost above the bound in the previous iteration (as this is the standard IDA* algorithm). That this frequently happens to be (on the order of) the gcd of action durations is an undesirable effect of the branching rule used to generate the search space.



(a) Airport (b) Pipesworld (c) Promela

(d) PSR (e) Satellite (f) UMTS

(g) Multi-UAV Mission Planning

Figure 7.6: Solution time, per problem instance, in each of the problem domains, with no relaxed search (*i.e.*, with the plain h^2 heuristic) compared to solution time with the h^2 heuristic improved by 3-relaxed search plus the time required for the 3-relaxed search, and the time for the 3-relaxed search alone. Note that the time scale (vertical axis) is logarithmic. Instances in each domain are sorted by the solution time with no relaxed search (as a rough order of difficulty). Results for the multi-UAV mission planning domain are split over two graphs, due to the large number of instances in this domain and the wide span in solution times. Only instances solved by both planners within the time limit (4 – 8 hours of CPU time, varying with the domain) are included: in the Pipesworld, Promela, PSR and Satellite domains, the planner using relaxed search solves more problems than the planner that does not.

and Satellite domains, the planner using relaxed search solves more problems than the planner that does not, while in the Airport domain it is the other way around. In the multi-UAV mission planning domain, both planners solve a few instances that the other fails to solve.

Across all domains solution times on the easiest instances (to the left in the graphs) for the planner using relaxed search are dominated by the relaxed search itself and exceed solution times with only the h^2 heuristic. In three of the domains (Pipesworld, Promela and PSR) there clearly exists a point beyond which the total time taken by the relaxed and final searches is less than that taken by search with h^2, *i.e.*, in these domains relaxed search is clearly cost effective for hard problem instances, while in the Satellite domain this holds for a majority of instances but it is not as clearly related to problem hardness (as measured by the time taken by search with h^2). In the remaining three domains (Airport, UMTS and multi-UAV mission planning) the relative performance of the planner using relaxed search does not as clearly improve on that of the planner without relaxed search (in the Airport domain it is clearly much worse).

Some Observations in the Airport Domain

The Airport domain offers an example of problems on which relaxed search is clearly very computationally expensive, as can be seen in figure 7.6(a). The problems solved (within the time limit) by the planner not using relaxed search in this domain are relatively easy, as evidenced by the fact that the number of nodes expanded in search is about the same as the solution depth. This implies that, for these particular instances, the h^2 heuristic is very accurate, so the fact that the heuristic improvement due to relaxed search is small (close to non-existent) is not surprising. However, the solved problems are only 13 out of the 50 in the problem set, which indicates that a more accurate heuristic is needed to solve the hard instances.

The question, then, is why the relaxed search is so time consuming. The apparent reason is that in this domain, search in the 3-regression space is more expensive than search in the normal regression space. This is contrary to the assumption stated above, that the cost of expanding a state (OR-node) in the relaxed regression space should be smaller than in the normal regression space due to a smaller branching factor.

For problems in the Airport domain states in the normal regression space contain a fairly large number of subgoal atoms, as much as 88.7 on average, while in the 3-regression space states corresponding to OR-nodes are by definition limited to 3 atoms. Consequently, the branching factor of OR-nodes in 3-regression is smaller (since the choice of establisher for each subgoal is a potential branch point) but not by much: 1.09, compared to 1.37 in the normal regression space. The many subgoals in the normal regression states interact, resulting in relatively few consistent choices. Also, the right-shift cut rule, which eliminates some redundant branches, is used in the normal regression space, but not when expanding OR-nodes in 3-regression. However, regression tends to make states "grow", *i.e.*, successor states generally

contain more subgoals than their predecessors, and while this effect is quite moderate in normal regression, where successors have, on average, 3% more subgoals, it is much more pronounced for the smaller states corresponding to OR-nodes in the 3-regression space, whose successors are on average 2.79 *times* larger. As a result, successors to OR-nodes are all AND-nodes, with an average of about 8.3 atoms and 70.8 successors (subsets of size 3). In summary, each expanded OR-node in 3-regression results (via an intermediate "layer" of AND-nodes) in an average of 77.2 new OR-nodes. Even though most of them (74.2%) are found in the IDAO* solved table, and therefore don't have to be searched, those that remain yield an effective "OR-to-OR" branching factor of 19.9 (25.8% of 77.2), to be compared with the branching factor of 1.37 for normal regression.

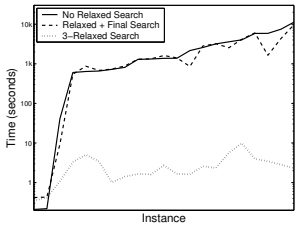
The problem is not the high branching factor in itself, but that the branching factor in the relaxed search space is far *higher* than it is for normal regression and that search in the 3-regression space is consequently more expensive than search in the normal regression space, rather than less.

Some Observations in the UMTS Domain

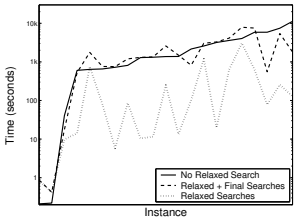
In the UMTS domain 3-regression is not very expensive (taking no more than 10 seconds, even on the hardest instances) but does on the other hand not lead to any significant improvement of the efficiency of the final search. Unlike in the Airport domain, this is not because solved instances are easy, in the sense that the h^2 heuristic is accurate enough: the hardest instances are solved just short of the 8 hour CPU time limit, requiring in excess of 10 million node expansions. The final search using the improved heuristic resulting from 3-relaxed search expands fewer nodes than search using only the h^2 heuristic (see figure 7.7(d)) but for a majority of the instances the reduction is not enough to have a significant effect on the total time, as shown by figure 7.6(f).

Problems in the UMTS domain are actually scheduling, rather than planning, problems and feature a number of reusable resources, *i.e.*, resources whose capacity constrain which, or how many, actions can take place concurrently but do not impose any limit on actions executing in sequence (the integration of reusable resources in temporal regression planning and in the h^m heuristics was described in chapter 5, page 82). Reusable resource constraints are the main reason why optimizing makespan in this domain is a hard problem, although a sizable fraction of the instances in the problem set are actually trivial in the sense that there are no resource conflicts (these instances are of course solved easily even with only the h^2 heuristic, as can be seen in the left part of the graph in figure 7.6(f)).

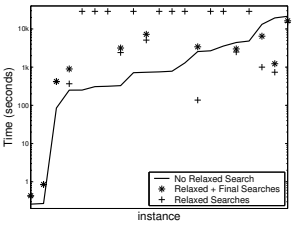
Reusable resources frequently cause non-binary mutual exclusions between actions, as the capacity of a resource may be sufficient to execute two actions concurrently but not three, or three but not four, *etc.*, and such exclusions are not detected by the h^2 heuristic, which considers at most two subgoal atoms and therefore at most two concurrent actions (as illustrated by the example in chapter 5, page 83). A possible explanation for the weak improvement yielded by 3-relaxed search is that a large



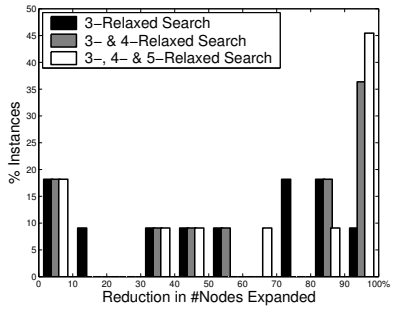
(a) 3-Relaxed Search



(b) 3- & 4-Relaxed Search



(c) 3-, 4- & 5-Relaxed Search



(d)

Figure 7.7: (a) – (c): Solution time, per problem instance, in the UMTS domain, comparing search with only the h^2 heuristic to search with the h^2 heuristic improved by relaxed searches plus the relaxed search time. Note that the time scale (vertical axis) is logarithmic. Instances are sorted by the solution time with no relaxed search. Unsolved and trivial instances are not included. (d) Distribution of the reduction of the number of nodes expanded in final search with the heuristic resulting from relaxed searches as compared to the number of nodes expanded in search with the h^2 heuristic.

number of these constraints involve more than three actions and are therefore not detected by the (partial) h^3 heuristic resulting from the relaxed search either. If this is the case then the heuristic resulting from m-relaxed search for higher values of m should be more effective.

Figure 7.7(a) – (c) shows the solution times for search with the h^2 heuristic compared to solution times with the heuristic resulting from relaxed search for $m = 3$ up to $m = 3, \ldots, 5$ (as before, the final search heuristic is the maximum of the complete h^2 and partial h^m solutions). Only solved instances are shown, except in figure 7.7(c) where instances not solved by the planner using relaxed search are also indicated. The trivial instances (with no resource conflicts) are also excluded. Figure 7.7(d) shows the distribution of the reduction in the number of nodes expanded in the final search, compared to search with h^2 only. As can be seen, the more accurate heuristics

do reduce the number of nodes expanded in the final search somewhat but still not enough to significantly reduce the total solution time. Also, the 4- and 5-relaxed searches are clearly much more computationally expensive – on several instances the 5-relaxed search is not even completed within the time limit.

In this domain, again unlike the Airport domain, the high computational cost of relaxed search is not due to states growing (in fact, due to the simple structure of solutions, states tend to decrease in size when regressed) but due to OR-nodes in the m-regression space having a higher branching factor than states in the normal regression states. The most likely explanation for this is that right-shift cuts, which reduce the branching factor, are applied in the normal regression search but not in the relaxed search. This is supported by the fact that the average branching factor in normal regression without right-shift cuts is 2.7 (higher than the branching factor of OR-nodes in 3-regression).

The Computational Cost of Relaxed Search

The h^m relaxation of a planning problem is in a sense a simpler problem: part of the idea underlying the relaxed search method of improving the heuristic is that the h^m relaxation also simplifies solving the problem by search. The assumption behind this idea is that small states (states consisting of a small number of subgoal atoms) are computationally cheaper to regress than larger states, due to having a smaller branching factor, and that regressing small states also gives rise to small successor states. As observed in the experiment, this assumption does not hold in some of the problem domains.

Figure 7.8 summarizes some characteristics of the normal and relaxed regression spaces for each the considered domains. These are the average state size ($|s|$), the average ratio of successor state size to the size of the predecessor state ($|s'|/|s|$) and the average branching factor (the number of successor states generated, and evaluated, per state expanded). The values are averages over the set of solved instances in each domain, except in the UMTS domain where trivial instances are not included and the Airport domain where they are from the smallest unsolved instance (since only trivial instances were solved in this domain).

In the domains where relaxed search is expensive, states tend to grow when regressed, *i.e.*, $|s'|/|s|$ is large and as a result there are many AND-nodes with many successors (a typical example being the Airport domain) or because OR-nodes in the relaxed regression space have a higher branching factor than in the normal regression space and are therefore computationally more expensive to expand (as for *e.g.* 4- and 5-regression in the UMTS domain). In the domains where relaxed search is cost effective, on the other hand, $|s'|/|s|$ is typically close to 1, *i.e.*, small states stay small when regressed, and regression of OR-nodes in the relaxed space is computationally cheaper than node expansion in the normal regression space, as indicated by a lower (or roughly equal) branching factor. The averaged numbers are of course not perfect predictors of performance: in the Promela domain, for example, the $|s'|/|s|$ ratio is quite large but relaxed search is quite effective anyway (as shown in figure 7.6(c)).

	Normal Regression	3-Regression OR	3-Regression AND	4-Regression OR	4-Regression AND	5-Regression OR	5-Regression AND
Airport							
$\|s\|$	88.7	3.0					
$\|s'\|/\|s\|$	1.03	2.79					
branching factor	1.37	1.09	70.8				
Pipesworld							
$\|s\|$	6.76	2.99					
$\|s'\|/\|s\|$	1.21	1.35					
branching factor	15.1	5.13	5.2				
Promela							
$\|s\|$	14.9	2.99					
$\|s'\|/\|s\|$	1.17	2.17					
branching factor	21.2	3.30	30.6				
PSR							
$\|s\|$	9.05	2.99					
$\|s'\|/\|s\|$	1.08	1.42					
branching factor	24.5	15.1	7.75				
Satellite							
$\|s\|$	6.88	2.99					
$\|s'\|/\|s\|$	1.04	1.06					
branching factor	6.98	5.01	7.33				
UMTS							
$\|s\|$	8.10	2.52		3.61		4.49	
$\|s'\|/\|s\|$	0.97	0.91		0.87		0.89	
branching factor	1.85	2.40	6.36	4.98	17.9	8.27	48.1
Multi-UAV Planning							
$\|s\|$	11.7	2.98					
$\|s'\|/\|s\|$	1.26	1.7					
branching factor	6.93	1.75	71.4				

Figure 7.8: Some characteristics of the normal regression and m-regression search spaces in the domains considered: the average state size ($|s|$), the average ratio of successor state size to the size of the predecessor state ($|s'|/|s|$) and the branching factor. In general, the values are averages over the set of solved instances, except in the UMTS domain where trivial instances are not included and the Airport domain where the values are from the smallest unsolved instance.

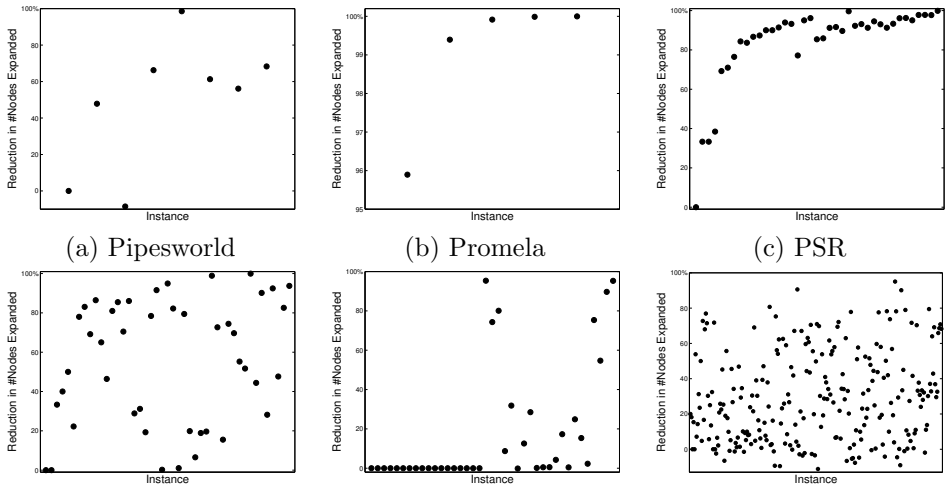
In the multi-UAV planning domain the sizes of AND-nodes in the 3-regression space show great variance, typically with many small states and a few large. As a result, the average $|s'|/|s|$ ratio is low but the average branching factor of AND-nodes quite large. This is because the number of successor states of an AND-node (subsets of size 3) is $\binom{|s|}{3}$ which grows exponentially with the state size so that the (very) high branching factor of the (relatively) few large states tends to dominate the average.

It is instructive to look more closely at why states grow when regressed in the Airport domain. The domain models movements of aircraft on the ground at an airport and thus part of the world state describes the position and status, for each aircraft, by three state variables (corresponding to exactly-one invariants, as described in chapter 2, page 24, and chapter 4, page 60): one tells where the aircraft is in the network of airport runways and taxiways, one which direction it is facing and one whether it is parked, being pushed or moving under its own power. Almost every action that changes one of these has a precondition on the other two as well. For example, actions that change the position of an aircraft require the aircraft to be moving and facing a particular direction and may also change the facing. Because of this, regressing a state containing only an atom from one of the state variables results, in most cases, in a state containing an atom from each of the three. A conclusion that can be made from this observation is that splitting large states (AND-nodes) into smaller states (OR-nodes) based only on the number of atoms is not always the right choice. An alternative would be to divide atoms in the planning problem into groups of "related" atoms and take the number of groups represented in a state to be its size.

Another observation is that the bulk of time spent in relaxed search is spent in the final search iteration, when a solution exists within the current cost bound. This iteration is not only the most expensive but also tends to be the least useful, since relatively few heuristic improvements are discovered during it. This also relates to the branching factor, specifically the fact that AND-nodes have many more successors than OR-nodes: for an AND-node to be solved all its successors must be solved and thus in the final iteration all successors of every AND-node are searched. However, the purpose of relaxed search is not to find a solution in the m-regression space but to improve the heuristic by finding size m states (OR-nodes) whose cost is underestimated and more accurate cost estimates for these. Therefore it may not actually be necessary to search all the successors to every AND-node: an alternative would be a (m, k)-regression search in which only the k most "promising" successors to every AND-node are considered, giving another dimension for iteratively refining the relaxed search.

The Value of the Heuristic Improvement

Another, even more fundamental, assumption on the basis of which the idea of the relaxed search method was conceived is that on problems for which the h^2 heuristic is not accurate enough, maximizing h^2 with one or more partial h^m solutions, for higher values of m, results in a better heuristic. This is, of course, mostly true, in the sense that the number of nodes expanded in the final search with the heuristic



(a) Pipesworld (b) Promela (c) PSR

(d) Satellite (e) UMTS (f) Multi-UAV Planning

Figure 7.9: Reduction in the number of nodes expanded in search with the h^2 heuristic improved by 3-relaxed search compared to the number of nodes expanded in a search using only the h^2 heuristic, per problem instance. Instances in each domain are selected and sorted as in figure 7.6.

resulting from relaxed search is less than the number of nodes expanded in the search using h^2 alone (although on a few problems in the Pipesworld, UMTS and multi-UAV planning domains the final search with the improved heuristic actually expands more nodes: this can be explained by the fact that the searches are done using IDA* and that raising some heuristic values can sometimes cause IDA* to perform more iterations, which can lead to more nodes being expanded if the heuristic improvement does not reduce the number of nodes expanded in each iteration enough).

It is, however, not always the case that the improvement is proportional to the weakness of the h^2 heuristic, or even shows any tendency to correlate with it. Figure 7.9 shows the reduction of the number of nodes expanded in the final search with the heuristic resulting from relaxed search compared to the number of nodes expanded in search with the h^2 heuristic, per problem instance in each of the domains (the Airport domain is not included because only very easy instances were solved in this domain). Instances are sorted in the same order as in figure 7.6, *i.e.*, by the time required to find the solution when searching with the h^2 heuristic. This shows a clear difference between on the one hand the Pipesworld, Promela and PSR domains, in which the reduction, *i.e.*, the gain due to the improved heuristic, increases with the hardness of the problem, as measured by the difficulty of solving it with only h^2, and the Satellite, UMTS and multi-UAV planning domains on the other, in which there is no such correspondence (except that in the UMTS domain the reduction is zero for the trivial instances which are also the easiest to solve).

Another point to consider is that the computational cost of the heuristic evaluation of a state, using the generalized heuristic table, is proportional to the number of subsets of the state for which there are values in the table. Thus, storing values of more atom sets in the heuristic table increases the cost of heuristic state evaluation and thus the search time. This effect varies between domains, from reducing the number of state expansions per second by only 1% in the UMTS domain to reducing it by $30 - 40\%$ in *e.g.* the Pipesworld and PSR domains.

Conclusions

The experiment and analysis described above reveals some characteristics of the problem domains for which the use of relaxed search for improving the h^2 heuristic is likely to be cost effective, at least for harder problem instances: in such domains, states corresponding to OR-nodes in the m-regression space are computationally cheaper to regress than states in the normal regression space (due to having a smaller branching factor) and do not give rise to many or large successor AND-nodes, and the relative effectiveness of the resulting improved heuristic at controlling the search compared to that of the underlying (unimproved) heuristic increases with the hardness of the problem, as measured by the weakness of the underlying heuristic. Conversely, if the computational cost of searching in the relaxed regression space is higher than that of normal regression (due to OR-nodes having a higher branching or giving rise to large successor states that become AND-nodes with a high branching factor) or if the resulting heuristic is not significantly more effective than the underlying heuristic, the relaxed search method is not cost effective.

The experiment, however, also demonstrates that several problem domains fall in the grey zone between these extremes: the Satellite, UMTS and multi-UAV planning domains are the main examples to be found among the domains considered. Yet, there is clearly a difference between *e.g.* the Satellite domain, in which the total time taken by relaxed search and the final search with the resulting heuristic is less than the time taken by search with only h^2 for 73% of the solved instances, and the multi-UAV planning domain in which this is the case for only 34% of the solved instances. Explaining this difference is an important open research question.

As a final note, an experiment comparing the cost effectiveness of the boosting technique (described in the next section) to that of relaxed search in the multi-UAV domain shows some interesting facts. The boosting technique exhibits greater variability, reducing the total time, compared to search with the basic h^2 heuristic, by as much as a factor 50 on some problem instances and increasing it by as much on some problem instances (to be compared with an increase/decrease of at most a factor 5 for relaxed search). The instances on which boosting is cost effective are relatively few, but those instances are also ones for which relaxed search is not cost effective. Thus, there seems to be some complementarity between the two techniques.

7.2 Boosting

The h^m heuristic estimates the cost of a set of atoms s by the cost of the most costly subset of size m but because this relaxation is applied recursively the heuristic can also underestimate the cost of a state s of size $|s| \leqslant m$. Thus, the heuristic can be improved in a different way by computing more accurate cost estimates for states of size at most m, but still using the approximation $\max_{s' \subset s, |s| \leqslant m} h(s')$ for states s with more than m atoms. We will refer to this as *boosting* the heuristic.

The optimal cost of a state can be found by simply performing a search starting from s. Since the heuristic that is being boosted is admissible it can be used to guide the search and if the search algorithm is admissible (guaranteed to find optimal solutions when guided by an admissible heuristic) boosting can not make the heuristic inadmissible. Even if the search is interrupted before a solution is found it may still result in an improved estimate of the cost of s (if done by a search algorithm that approaches the optimal cost from below, such as IDA* or A*).

For this to be cost effective the search for improved heuristic values must, obviously, be strictly limited. The idea of the scheme presented in this section is to keep the required search depth low by boosting heuristic estimates of states in increasing order, starting with states that have a low estimated cost and proceeding to states with higher heuristic values. In this way, the already boosted heuristic values contribute to reducing the effort needed for later boosting searches. Even so, searches may need to be further limited and focused.

Boosting in the Planning Graph

The boosting scheme presented here is more easily explained by applying it to the planning graph (described in chapter 3, page 36). Recall that the planning graph is a directed graph of alternating proposition and action layers, where each action layer contains the actions (including no-ops) whose preconditions are present (and non-mutex) in the preceding proposition layer and each proposition layer contains all atoms added by actions (or no-ops) in the preceding action layer. The graph encodes a relaxed notion of atom reachability, in the sense that if an atom p appears for the first time in the nth layer it is not achievable in less than n steps. Furthermore, in each layer a binary mutex relation marks pairs of atoms whose conjunction is not achievable in the corresponding number of steps (in action layers, the mutex relation marks actions that are not concurrently executable). The graph thus encodes a heuristic that estimates the cost of a pair of atoms by the index of the first proposition layer in which they both appear and are not mutex, and the cost of a larger set of atoms by the highest cost of any pair in the set. This heuristic is admissible for parallel planning (and, in fact, equivalent to h^2).

In the context of the planning graph, boosting a heuristic value means taking a pair of atoms, p and q, that appear without mutex for the first time in the nth proposition layer and performing a search to solve the goal $\{p, q\}$ in at most n steps (single atoms

appearing for the first time in the nth layer are also eligible for boosting). If no such solution exists, the conjunction of atoms p and q can not be achieved in n steps and the two atoms can therefore be marked as mutex in the nth layer of the planning graph (in the case of a single atom the atom is removed from the nth layer, as it is in fact not reachable in n steps).

In Graphplan, the planning system that introduced the planning graph, the search for a solution is done by parallel regression (a special case of temporal regression, as described in chapter 2, page 14) but organized by the layers of the graph (though as noted in section 3.4 of chapter 3, later planning systems have organized search around the planning graph in different ways). A search state is a set of atoms (representing subgoals to be achieved) in a specific layer. The state is regressed by choosing, non-deterministically, a set of compatible establishing actions from the preceding action layer, and the preconditions of those actions, in the preceding proposition layer, become the next search state. Whenever a search state contains a pair of atoms that are mutex the state is known to be unsolvable and that branch of the search can be cut.

By performing the boosting searches for pairs of atoms in the lower layers of the planning graph first the mutex relations added to the graph as a result of boosting help cut search branches, thereby reducing the computational cost, in the following boosting searches.

Boosting the h^m Heuristic

Applying the scheme outlined above to boosting the h^m heuristic works, in principle, in the same way, but instead of proving additional mutex relations the boosting searches prove higher (admissible) heuristic estimates for search states of size at most m.

As noted at the beginning of this chapter we assume that a generalized heuristic table is used to store the heuristic and to perform on-line state evaluations (as described in chapter 3, page 42) and denote the table by T and the corresponding heuristic simply by h. Exactly what is stored in the table is not important: it can be a complete or partial h^m solution or even the combination of several (complete or partial) solutions (like the heuristic resulting from relaxed search). For the purpose of boosting the heuristic it is sufficient that a list of entries $(s, v) \in T$ can be efficiently extracted. We also make use of the fact that additional information (in particular, a number of binary variables or "marks") can easily be associated with each entry in the table. This affects the memory requirements of the general heuristic table only marginally and the cost of performing heuristic evaluations not at all.

A possible boosting procedure is sketched in figure 7.10 and described in more detail in the following. Note, however, that there are a number of choices and alternatives in applying the boosting idea to the h^m heuristics, some of which are also discussed below.

Entries in the heuristic table whose cost is already known to be infinite (*i.e.*, unreach-

```
boost(T)  // heuristic table to be boosted
{
  // extract and sort the list of entries
  L = sort {(s,v) in T | v < +INF and s not solved} by increasing v;

  // main loop
  repeat {
    (s,v) = first entry in L;
    lim = v' of second entry (s',v') in L, +INF if |L| = 1;
    new v = IDA*(s, lim);  // search with cost bound lim
    if (new v > v) {
      T(s) = new v;
    }
    if (s solved) {
      mark s as solved;
      remove (s,v) from L;
    }
    else if (new v = +INF) {
      remove (s,v) from L;
    }
    else if (new v > v) {
      replace (s,v) by (s,new v) in L, at appropriate position;
    }
    else {  // no cost improvement and not solved
      remove (s,v) from L;
    }
  } until (v > h(G), for (s,v) = first entry in L, or L empty)
}
```

Figure 7.10: Boosting procedure, using IDA* for boosting searches. The procedure uses the early stop condition (estimated cost of cheapest boostable entry greater than $h(G)$) and can also be used with a limit on the amount of work allowed per boosting search (only with such a limit can the last case, where s is not solved but the new cost is not higher, occur).

able states) obviously do not need to be boosted, so any such states can be removed from the list of entries to boost, as can states whose cost is known to be optimal (*i.e.*, states for which a solution has been found within the current estimated cost). Initially, only states with an estimated cost of zero, *i.e.*, sets of atoms that hold in the initial world state, are known to be optimal but as boosting searches are carried out more states become solved, since some searches end with a solution found. Entries in the heuristic table whose cost is known to be optimal are marked as such. Among the remaining entries many are not relevant, in the sense that they do not contribute to the evaluation of any state encountered in the search for a solution to the planning problem at hand, but deciding which entries are relevant in this sense is hard.

Throughout the boosting process the list of selected entries is maintained sorted in order of increasing estimated cost, and boosting searches are made in this order. To achieve the behaviour described above for boosting in the planning graph, however, boosting searches are made with a cost limit equal to the estimated cost of the next

(higher cost) entry in the list. If a state is solved within this limit the corresponding entry is marked as solved and removed from the list, otherwise it is repositioned in the list to reflect its new (higher) estimated cost, unless the search proved that the state is unreachable. Boosting of the state may be resumed if it again becomes the least costly entry on the list. This assumes that the boosting search is carried out using a search algorithm that approaches the optimal solution cost from below, such as IDA* or A*, and that the searches are guided by the heuristic that is being boosted so that effective use is made of the already boosted heuristic values in subsequent boosting searches. If necessary, boosting searches can be further limited to a fixed time, number of nodes expanded or some other measure of search effort. In this case entries for which the search is interrupted without any improvement in cost are also removed from the list.

As entries are solved or proved unreachable (or reach some search effort limit without cost improvement) the list of entries to boost shortens and eventually becomes empty, at which point the boosting process comes to a natural end. Continuing up to this point, however, is typically too computationally expensive to be cost efficient. If there exists a solution for the set of goals of the planning problem (G) within the current estimated cost $h(G)$ there is clearly nothing to be gained from boosting the heuristic estimates of states s that already have a value $h(s) > h(G)$: thus, a reasonable earlier stopping point is when the estimated cost of the next (least costly) entry in the list is greater than $h(G)$. Note that $h(G)$ may increase during the process as a result of the cost estimates of subsets of G being boosted.

If boosting is performed in a separate step before the search for a plan (like the relaxed searches in the planning system described on page 126 in the previous section) the cost of the problem goals may be underestimated, so stopping at $h(G)$ some relevant entries may be missed. An alternative is to interleave the main search for a plan with the boosting searches. This is described in the section following the next (page 143).

Conflict Detection

Boosting can only improve the h^m heuristic to a certain point. This is because boosting only improves the heuristic estimates of states already stored in the heuristic table, *i.e.*, states with $|s| \leqslant m$. In this respect boosting differs from relaxed search which if performed for increasing m approaches h^*, albeit only for part of the search space (but a part that includes the problem solution). To achieve this with boosting increasingly larger atom sets have to considered for boosting.

As long as a state s is not solved there is no reason to boost any superset of s since further improving the cost estimate for s also improves on all its supersets. Thus, it is only when a set s becomes solved that new states $s \cup \{p\}$, for atoms $p \notin s$, are considered for boosting. Many of these are not interesting, however, in the sense that their cost equals that of the most costly subset: this is of course discovered when the new entry is boosted, but because the boosting search that finds a solution is typically the most expensive search (and does not yield any improvement to the

heuristic) the strategy for adding new states to the list of entries to boost must be more selective for this method to be cost effective. Several criteria, with different degrees of selectivity, for choosing potentially interesting supersets of s can be devised based on analyzing the solution plan found for s for conflicts with atoms not in s. The criteria are slightly different for sequential and for temporal planning.

Conflict Criteria for Sequential Planning

If the plan for s also achieves p the set $s \cup \{p\}$ has the same optimal cost as s and is thus uninteresting in the sense above. This is true of both sequential and temporal planning. In the sequential planning case, only if *no* plan that achieves s also makes p true must the set $s \cup \{p\}$ necessarily have a higher optimal cost since then at least one more action must be taken to achieve p (to call the failure of the plan for s to achieve p a "conflict" may seem like stretching the word: the conflict lies in the fact that different plans are required to achieve s and p, and in sequential planning this causes a conflict in the sense that the two plans can not both be executed at the same cost).

A criterion based on the set of all optimal solutions for s is typically too computationally expensive (although it is possible to infer some properties that hold for every valid plan for a given problem without actually generating all such plans, *e.g.*, using *landmarks* as described by Hoffmann, Porteous, & Sebastia, 2004). A computationally cheaper alternative is to consider only the (arbitrary) plan found for s. This is more restrictive, as even if this plan does not achieve p there can be an alternative plan for s that does. In practice, however, using a more restrictive condition is preferable, since this helps keep the list of entries to boost from growing too much.

Conflict Criteria for Temporal Planning

Temporal planning differs from sequential planning in that separate plans required to achieve separate goals can sometimes be executed concurrently, and thus at a cost (makespan) not necessarily greater than that of the separate goals. Thus, conflict criteria for temporal planning aim to detect if the found plan (or every optimal plan) for s is (or is likely to be) incompatible with any (or some) plan to achieve p. None of the following criteria are sufficient to ensure that the makespan of an optimal plan for $s \cup \{p\}$ is necessarily greater than that for achieving s only, they only suggest that this is more likely to be the case: the plan for s contains an action that deletes p; the plan for s contains an action that deletes a precondition of some (or every) action that adds p; the reusable resource requirements of the plan (at some point, or throughout the whole of the plan) are high enough to prevent some (or every) action that adds p from being executed concurrently (this also applies to unit capacity resources modelled by temporary delete effects). As in the sequential case, the different criteria can be made more or less restrictive by considering the set of all optimal plans for s (in addition to the alternatives outlined above), though this is typically too computationally costly.

Interleaved Search and Boosting

The main problem with the boosting method, as presented above, is that it lacks goal direction in the sense that many of the boosted heuristic table entries are not relevant to the search for a plan for the problem goal, G. By interleaving the main search for a plan with (possibly limited) boosting, information learned in the search can be used to better direct boosting efforts. If the main search is done with IDA*, the natural point to insert boosting is between iterations, when the known lower bound on the cost of the root search state has increased, and a similar point can also be found in the A* algorithm (whenever the estimated cost of the first node on the open queue increases).

This can be done in three ways: First, if the boosting process is terminated early (when the estimated cost of the least costly entry in the list of entries to boost exceeds the estimated cost of the problem goals, $h(G)$) it can be resumed when the main search has progressed enough to prove that there is no solution within $h(G)$. Again, this assumes that the search algorithm is one that approaches the optimal cost from below, like IDA* or A*. Second, boosting can be limited to only those entries that have actually been used in the evaluation of some search state during the main search so far (in addition to the conditions specified above), thus directing boosting efforts to the entries that appear to be relevant. Marking the heuristic table entries that are used can be done during the heuristic evaluation procedure, without significant overhead.

Third, when interleaving boosting with the main search it is possible to impose a limit on the amount of effort invested in boosting searches and to adjust this limit dynamically, based on the effort spent in the main search. For example, the time per boosting search can be limited so that the total time consumed by the next round of boosting is no more than a constant times the time consumed by last segment of the main search (since the number of entries to be boosted is known). Limiting the boosting effort can of course also be done when boosting is done as a separate step before the main search, but the advantage in the case of interleaved boosting and search is that the limit can be determined automatically and adjusted to the hardness of the problem as indicated by the time consumed by search.

7.3 Discussion: Related Ideas

The idea of using search to derive or improve heuristics is not new. This section reviews a selection of related methods and discusses how these (or similar methods) can be adapted and exploited to improve the relaxed search and boosting methods presented in this chapter.

Search-Based Heuristics

Deriving heuristics by solving an abstracted, or relaxed, version of the search problem is not a new idea, and neither is the idea of using search to solve the abstract problem (see *e.g.* Gaschnig, 1979, Pearl, 1984, or Prieditis, 1993).

The recently most successful variant on this theme is pattern database heuristics, which were discussed in chapter 4. Pattern database heuristics are cost effective because the relaxation defined by a pattern is an *abstraction*, mapping many states in the original search space to each state in the abstract space (*cf.* the discussion about cost effectiveness and Valtortas theorem earlier in this chapter, page 124). As noted in chapter 4 (page 78) the h^m and pattern relaxations are actually very different, in that the h^m relaxation recursively selects the most expensive of all possible size m subsets of any state with size greater than m while the abstraction induced by a pattern selects the same subset of variables in every state. Thus, the methods presented in this chapter have little in common with pattern database heuristics beyond the fact that they both use search to solve relaxed problems.

More closely related is the idea of *pattern searches*, developed in the context of the Sokoban puzzle (Junghanns & Schaeffer 2001), which are more dynamic. Like a pattern database heuristic, a pattern search abstracts away part of the problem and solves the remaining (small) problem to obtain an improved lower bound but the pattern (*i.e.*, the part of the problem that is kept by the abstraction) is selected whenever a particular state expansion ("move") is considered. Patterns that have been searched are stored, along with their updated cost, and taken into account in the heuristic evaluation (by maximization) of any new state that *contains* the same pattern encountered in the search. This is similar to the way that boosting the h^m heuristic improves the cost estimates for selected relaxed states in a way so that heuristic estimates of several search states are improved. The patterns explored by pattern searches are found through an incremental process: The first pattern consists of only the part of the problem ("stone") that is directly affected by the move under consideration. The next pattern extends the previous with stones that in the current state conflict with the solution found in the preceding pattern search, and this is repeated until no more conflicts are found. The method of expanding the list of entries to boost through conflict detection described above (page 141) is inspired by the way conflicts are used in pattern searches. In fact, the same idea can be seen in the incremental pattern selection method presented in chapter 4 (page 72) and a similar use of conflicts to direct search efforts in the context of relaxed search is discussed below.

An observation made in the experimental analysis of relaxed search (section 7.1, page 135) is that one factor contributing to the sometimes high computational cost of the m-regression search is the fact that AND-nodes have many successors. As also noted it is most of the time not actually necessary to search all successors for every AND-node: often, many of them are solved at a cost lower than that of the most costly size m subset and thus do not contribute to raising the estimated cost of the parent AND-node. An alternative is to search only the most "promising"

successors, where a promising successor is an OR-node whose cost is likely to be underestimated by the current heuristic and therefore likely to increase when the node is expanded. Limiting the number of successors searched for every AND-node uniformly to at most k results in an (m, k)-regression space, and a series of (m, k)-regression searches with increasing m and k can be organized in different ways: for example, the planner could perform $(m, 1)$-regression searches for $m = 3, \ldots$ until some stopping condition is met, then $(m, 2)$-regression searches for $m = 3, \ldots$, *etc.* This is very similar to the iterative broadening search described by Ginsberg & Harvey (1992). Incremental solving of subproblems is also found in the Russian doll search algorithm (by Verfaillie, Lemaitre, & Schiex, 1996) for constraint optimization, the problem of finding an assignment of discrete values to a collection of n variables that minimizes a specified target function, subject to a set of hard constraints. The algorithm proceeds by solving subproblems of increasing size, starting with the nth variable only, then the two last variables, and so on, similar to an $(m, 1)$-relaxed search. The solution cost for the i last variables is a useful lower bound in the search for a solution to the problem involving the $i + 1$ last variables because variables are assigned in a fixed order (from 1 to n).

Another alternative is to limit the expansion of AND-nodes non-uniformly, *e.g.*, to search all successors satisfying some criterion for being promising, similar to iterative widening used in the context of game-tree search (Cazenave 2001). Promising successors can be identified by looking at conflicts, similar to the strategy for selecting entries to boost presented here and the process for finding patterns in pattern searches. Consider sequential planning where an AND-node is simply a set of more than m atoms and the successors are all size m subsets of this set: subsets more likely to have a higher cost than the estimate given by h^{m-1} can be identified by solving each size $m-1$ subset and examining the solutions for conflicts with remaining atoms in the state (using various conflict criteria, as discussed in section 7.2 above). Some care must be taken to ensure that the searches needed to find the promising sets are not more expensive than searching every set but if the h^{m-1} value was also computed by relaxed search, the size $m - 1$ subsets (or at least some of them) have already been searched and conflicts found during previous searches can be saved.

Another method that uses limited search to improve heuristic estimates is *perimeter search* (Dillenburg & Nelson, 1994; Manzini, 1995) This algorithm first performs a depth-limited search from the target node, following the transitions of the search space in reverse, to generate a set of nodes (the *perimeter*) from which the target node is reachable with known cost. In the main search, which starts from the source node, the heuristic estimate is calculated as the estimated cost of reaching a node n on the perimeter plus the known cost of reaching the target node from n, minimized over all nodes on the perimeter. Since the error in heuristic estimates tends to increase with the distance in the search space, this estimate is often more accurate than the raw heuristic estimate of the cost of reaching the target node. The main problem with the method is that heuristic evaluations become very expensive as the set of perimeter nodes grows (Kaindl & Kainz, 1997, provide a detailed analysis of perimeter search and other bidirectional search algorithms, and suggest remedies for this problem).

The relation between perimeter search and boosting is easiest to explain in the context of the planning graph: Recall that the planning graph encodes information about relaxed, or *possible*, reachability. By performing boosting searches for atom pairs that appear reachable in a layer and extending the mutex relation to include those pairs proved unreachable, the relaxed measure of reachability is brought closer to *actual* reachability. The construction of a perimeter in perimeter search does the same thing, but limited in depth rather than in the size of atom sets considered. Thus, in the context of the planning graph, constructing a perimeter up to a depth of n corresponds to boosting every atom set (regardless of size) but in the first n proposition layers of the graph only. By contrast, the boosting scheme presented here boosts sets in every proposition layer of the graph, but only sets of at most two atoms.

AND/OR Search Algorithms

Algorithms for searching AND/OR spaces (or trees) have been mostly investigated in the AI area of game playing.

The SCOUT AND/OR tree search algorithm, developed by Pearl (1984), tries to reduce the number of nodes evaluated by first testing for each node if it can affect the value of its parent before evaluating the node exactly. The test is performed by a procedure, called simply "Test", which takes as arguments a node and a threshold value and determines if the value of the node is greater (or equal) than the threshold by recursively testing the node's successors (to a specified depth) but only until the inequality is proved. The procedure can easily be modified to return a greater value for the node when such a value is found (though this may still be less than the nodes actual value) and it has been shown that the Test procedure, enhanced with memory in the form of a transposition table, can be used iteratively to give an efficient algorithm that determines the exact value of a node (Plaat *et al.* 1996).

The DFS subroutine of IDAO* is very similar to a depth-unbounded version of the Test procedure and thus the IDAO* algorithm is similar to such an iterative application of Test. The main difference lies in that IDAO-DFS applies iterative deepening (by calling IDAO*) to the successors of AND-nodes, whereas Test calls itself recursively with the same cost bound. As a result, IDAO* finds the optimal cost of any solved OR-node which Test does not (though the higher cost returned by the modified Test procedure when the cost of a node is proved to exceed the threshold is still a lower bound on the nodes optimal cost). Recently, Bonet & Geffner (2005) presented a general depth-first search algorithm for AND/OR spaces, called LDFS, which is also similar to IDAO*. Like IDAO*, it finds the optimal cost of every solved node and improved lower bounds on nodes that are explored but not solved. LDFS, however, stops searching the successors of an AND-node as soon as one of them is found to have a cost greater than the current estimate for that node, even if the new cost estimate is still less than that of the parent AND-node (and the cost estimate of the AND-node thus unaffected by the improvement). IDAO*, on the other hand, performs iterative deepening searches until the node is solved or shown to have a cost

greater than the current estimated cost of the predecessor AND-node. Experiments with an Iterative Test algorithm for m-regression search have shown that it is not more efficient than IDAO*. An experimental comparison between IDAO* and the LDFS algorithm remains to be done.

The GBF algorithm described in chapter 3, like most other algorithms for computing complete solutions to the h^m equation, can be seen a "bottom-up" labeling of the nodes in the m-regression space, starting from nodes with cost zero and propagating costs to parent nodes according to the min/max principle. The propagation is complete, *i.e.*, it proceeds until every (solvable) node has been labeled with its optimal cost (although only the costs of OR-nodes are actually stored). IDAO*, Iterative Test and LDFS, on the other hand, all perform top-down, depth-first iterative deepening searches. None of these characteristics of the algorithms, however, are essential for the their use in computing an improved heuristic. Any AND/OR search algorithm can be used to carry out the relaxed search, as long as it discovers the optimal cost (or a greater lower bound) of every expanded OR-node. For example, the standard AO* algorithm (Nilsson 1968) and the Generalized Djikstra algorithm by Martelli and Montanari (1973) both do this, and both offer a possibility of trading greater memory requirements for (hopefully) less search time (though the results of Bonet & Geffner, 2005, indicate that this may not be the case).

The AND/OR (or Min-Max) search spaces representing two-player games are somewhat different from the m-regression space: there is no concept of solution cost, other than the won/lost distinction, and for most games it is infeasible to search for a complete solution, rather the search aims to improve the accuracy of a heuristic estimate of the usefulness of a move or position in the game. Thus, game-tree searches are depth-bounded, rather than cost-bounded, and values at the leaf nodes of the tree are given by a static heuristic function. m-regression can be formulated in this way, by taking the sum of accumulated and estimated cost as the static heuristic function, and a depth-bounded search with a standard game-tree search algorithm used to improve the accuracy of the estimated cost of the root node. This, however, fails to achieve the main objective of relaxed search, which is to discover (and store) improved cost estimates for the size m states encountered during the search. Thus, this method would have to be used in a different way, *e.g.*, as a depth-bounded look-ahead to improve the accuracy of heuristic evaluations of states in the normal regression search.

8. Conclusions

The development of efficient and capable domain-independent automated planners is a many-faceted problem: the problem model must be expressive enough to allow application problems to be accurately stated and the efficiency of a search-based planner depends both on the formulation of planning as a search problem and on effective search control. This thesis addresses only one aspect of this broad problem, that of search control for optimal planning through the automatic creation of admissible heuristics, specifically, heuristics for the regression search space associated with the classical sequential and temporal STRIPS planning models.

As noted in chapter 2, the sequential and temporal STRIPS models are not the most expressive and the regression method of plan search is not the most efficient, but the developed methods are based on principles general enough that methods for the creation of heuristics for other, more efficient, planning search spaces and other, more expressive, planning models can be derived on the same basis. For example, the definition of the h^m heuristic depends only on the existence of a "subset" relation and a measure of size on states of the search space and the admissibility of the heuristic rests only on the requirement that any expansion operation applicable to a search state is applicable also to any of its subsets. Pattern database heuristics can be applied to any search space where a mapping to a finite abstract space can be defined from patterns of some suitable form, and the principle of enforcing global constraints in the pattern abstraction is also generally applicable.

An indication of this is the fact that some of our methods, in particular the h^m heuristics, have been adopted by researchers focusing on other aspects of the development of automated planning systems: *e.g.*, the CPT temporal planning system employs the h^m heuristic in a constraint-based partial order planning search (Vidal & Geffner 2004), while the BFHSP sequential planner uses it in progression and regression search together with a novel search algorithm (Zhou & Hansen 2004).

The thesis collects material previously published in a series of papers: Haslum & Geffner (2000) introduced the h^m heuristics for sequential and parallel planning. Haslum & Geffner (2001) introduced temporal regression, including planning with resources, and the adaption of the h^m heuristics to this case. Haslum (2004; 2006) introduced the relaxed search and boosting methods (the second paper also contains an earlier version of the experimental analysis presented in chapter 7). Haslum, Bonet & Geffner (2005) introduced the constrained abstraction for pattern database heuristics, the iterative pattern selection method and the additive h^m heuristics. The main addition to this material that has been made in the thesis is a considerable amount of detail: complete algorithm descriptions, formal proofs, more extensive analyses of experimental results and discussions of alternative problem models.

Final Analysis: Admissible Heuristics for Regression Planning

Not that many different admissible heuristics for classical planning are known: there are heuristics based on variations of the planning graph, the h^m heuristics, the additive h^m heuristics, and pattern database heuristics. An important part of this work is the analysis of the relative strength and cost effectiveness of different heuristics.

As shown in chapter 3, for the case of sequential and parallel planning with unit costs the h^m and planning graph heuristics are equivalent, in the sense that they yield the same heuristic estimates, though they are different in other respects, such as *e.g.* their respective methods of computation. The planning graph has also been extended to yield admissible makespan heuristics for temporal planning (Smith & Weld 1999; Garrido, Onaindia, & Barber 2001), which are not obviously equivalent to the temporal h^2 heuristic. Proving or disproving this equivalence is an open problem.

Heuristics for Sequential Planning

For the case of sequential planning, the h^2, additive h^2 and the improved pattern database heuristics have been experimentally compared (in chapter 6, page 105). Although the experiment covers only three different problem domains, this is sufficient to demonstrate that none of the heuristics dominates any of the other. Results of the experiment suggest that pattern database heuristics are more effective in problem domains that have "deep" solutions (involving many changes of value for each state variable) while the h^2 heuristic is more effective in domains where solutions are "wide" (each state variable changes value only a few times, but many variables are required to change values to enable the few that have goal values to change) and that the additive h^2 heuristic combines both some strengths and some weaknesses of the other two (being the most effective in the "intermediate" problem domain). However, this can only be considered a conjecture since the scope of the comparison is quite small. Also, it should be emphasized that only one particular action partition method for the additive h^2 heuristic is considered in this experiment. A different partitioning method may yield very different results.

Cost Effectiveness of Improving h^m through Search

The second large experimental analysis (presented in chapter 7, page 124) concerns the cost effectiveness of the use of the relaxed search technique for improving the h^m heuristics for temporal planning. Results demonstrate some characteristics of the problem domains in which this use of relaxed search is likely to be cost effective, at least for harder problem instances: in such domains, states corresponding to OR-nodes in the m-regression space are computationally cheaper to regress than states in the normal regression space (due to having a smaller branching factor) and do

not give rise to many or large successor AND-nodes, and the relative effectiveness of the resulting improved heuristic at controlling the search compared to that of the underlying (unimproved) heuristic increases with the hardness of the problem, as measured by the weakness of the underlying heuristic. Although this analysis is performed only for the case of temporal planning, it is a reasonable conjecture that similar criteria hold for the sequential case. For sequential planning, however, a wider range of methods is available, so that the heuristic resulting from applying relaxed search to improve a complete h^m solution may be outperformed by, *e.g.*, an additive h^m or pattern database heuristic even in domains where the use of relaxed search is cost effective compared to the use of only the basic h^m heuristic.

For the second technique, boosting, only a few initial experiments have been made, but results suggest that in certain cases where relaxed search is not cost effective, compared to the basic h^m heuristic, this technique may be.

Directions for Future Research

As mentioned, above and in chapter 2, the classical and temporal planning models for which our methods of heuristic construction have been developed are quite simple, so one direction for the continuation of this work is adapting the methods to more expressive models of planning problems. Particularly, the more scheduling-like CBI and HTN models (described in section 2.3 of chapter 2, page 28) are interesting for planning with time. A small step towards increasing the expressivity of the planning models was made in chapter 5, with the introduction of resources. The methods of constructing heuristics for the extended models are quite crude, however: for the case of consumable resources in particular, estimates of plan cost or makespan and estimates of the resources required are not integrated and thus heuristics frequently fail to detect cost/resource trade-offs, which are typical of planning problems involving resources. A more elaborate solution would likely involve some kind of multi-criteria optimization method (such as *e.g.* that suggested by Refanidis & Vlahavas, 2003).

Concerning the methods themselves, several design options remain unexplored. For example, the analysis of the quality of the PDB heuristics resulting from different pattern selection methods (in section 4.3 of chapter 4, page 72) suggests that there is room for improving the incremental selection method by making more informed some of the choices that are currently made arbitrarily. For the problem of partitioning the set of actions in the construction of the additive h^m heuristics only a single method was developed, though there are many more imaginable. Finally, the boosting technique has barely been sketched and may also be applicable to heuristics other than h^m.

The experimental analyses presented in this thesis are aimed at discovering characteristics of planning problems that determine the relative effectiveness of the different heuristics. The resulting characterizations, however, are yet far from exact, and need to be further refined. Although our results indicate that no single method of admissible heuristic construction will yield results superior to all others across the full

spectrum of planning problems, the eventual development of a suite of methods along with a precise characterization of the problem domains best fit to each method is a conceivable prospect.

Acknowledgements

I am deeply grateful to a great many people: my advisor, Patrick Doherty; my collaborators, Héctor Geffner and Blai Bonet; everyone else I've worked with, in and out of the WITAS project (Peter, Marcus, Uli, Jonas, Lars, Silvia, Tommy, Björn, Klas, Johan, Fredrik, Per-Olof, Per, Erik, Simone, Torsten, Gianpaolo, Mariusz, Piotr); the department administrators (Janette, Jenny, Anna-Maria, Anki, Lillemor, Britt-Inger); friends and family; and probably many more.

The constructive criticisms of Patrick, Jonas, and Per-Olof on this thesis, and of numerous anonymous reviewers on the papers that preceded it, have helped shape it into its present, hopefully readable, form.

All faults are, of course, mine.

This work has been funded in part by the Knut and Alice Wallenberg Foundation under the WITAS UAV Project and by the ECSEL/ENSYM graduate school.

References

Aho, A.; Hopcroft, J.; and Ullman, J. 1983. *Data Structures and Algorithms.* Addison-Wesley.

Allen, J., and Koomen, J. 1983. Planning using a temporal world model. In *Proc. 8th International Joint Conference on Artificial Intelligence (IJCAI'83)*, 741 – 747.

Aylett, R.; Soutter, J.; Petley, G.; Chung, P.; and Rushton, A. 1998. AI planning in a chemical plant domain. In *Proc. 13th European Conference on Artificial Intelligence (ECAI'98)*, 622 – 626.

Aylett, R.; Petley, G.; Chung, P.; Chen, B.; and Edwards, D. 2000. AI planning: Solutions for real world problems. *Knowledge-Based Systems* 13:61 – 69.

Bacchus, F., and Kabanza, F. 1995. Using temporal logic to control search in a forward chaining planner. In *Proc. 3rd European Workshop on Planning (EWSP'95).*

Bacchus, F. 2000. Subset of PDDL for the AIPS 2000 planning competition. `http://www.cs.toronto.edu/~aips2000/pddl-subset.ps`.

Bäckström, C. 1992. *Computational Complexity of Reasoning about Plans.* Ph.D. Dissertation, Linköpings Universitet.

Barret, A., and Weld, D. 1994. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence* 67(1):71 – 112.

Barták, R. 2004. Integrating planning into production scheduling: A formal view. In *Proc. ICAPS'04 Workshop on Integrating Planning into Scheduling*, 1 – 8. `http://pst.istc.cnr.it/wipis-at-icaps-04/WIPIS-ICAPS04-Notes.pdf`.

Becker, M., and Smith, S. 2000. Mixed-initiative resource management: The AMC barrel allocator. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 32 – 41.

Bedrax-Weiss, T.; Crawford, J.; and Smith, D. 2004. Compiling planning into scheduling: A sketch. In *Proc. ICAPS'04 Workshop on Integrating Planning into Scheduling*, 9 – 16. `http://pst.istc.cnr.it/wipis-at-icaps-04/WIPIS-ICAPS04-Notes.pdf`.

Beetz, M. 2002. Plan representation for robotic agents. In *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02).*

Bellman, R. 1957. *Dynamic Programming.* Princeton University Press.

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control*, volume 1 & 2. Athena Scientific.

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief: A preliminary report on combining state abstraction and HTN planning. In *Proc. 6th European Conference on Planning (ECP'01)*, 157 – 168.

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, 1636 – 1642.

Blum, A., and Furst, M. 1997. Fast planning through graph analysis. *Artificial Intelligence* 90(1-2):281 – 300.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proc. 5th European Conference on Planning (ECP'99)*, 360 – 372.

Bonet, B., and Geffner, H. 2005. An algorithm better than AO*? In *Proc. 20th National Conference on AI (AAAI'05)*, 1343 – 1347.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *Proc. 14th National Conference on Artificial Intelligence*.

Boppana, R., and Halldorsson, M. 1992. Approximating maximum independent sets by excluding subgraphs. *BIT* 32(2).

Bylander, T. 1991. Complexity results for planning. In *Proc. 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, 274 – 279.

Bylander, T. 1996. A probabilistic analysis of proposition STRIPS planning. *Artificial Intelligence* 81(1-2):241 – 271.

Cazenave, T. 2001. Iterative widening. In *Proc. 17th International Conference on Artificial Intelligence (IJCAI'01)*, 523 – 528.

Chapman, D. 1987. Planning for conjunctive goals. *Artificial Intelligence* 32:333 – 377.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; Stebbins, G.; and Tran, D. 2000. ASPEN – automating space mission operations using automated planning and scheduling. In *Proc. 6th International Symposium on Technical Interchange for Space Mission Operations*.

Cormen, T.; Leiserson, C.; and Rivest, R. 1990. *Introduction to Algorithms*. MIT Press.

Culberson, J., and Schaeffer, J. 1996. Searching with pattern databases. In *Canadian Conference on AI*, volume 1081 of *LNCS*, 402 – 416. Springer.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318 – 334.

Dillenburg, J., and Nelson, P. 1994. Perimeter search. *Artificial Intelligence* 65(1):165 – 178.

Do, M., and Kambhampati, S. 2000. Solving planning-graph by compiling it into CSP. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 82 – 91. AAAI Press.

Do, M., and Kambhampati, S. 2001. Sapa: A domain-independent heuristic metric temporal planner. In *Proc. 6th European Conference on Planning (ECP'01)*, 109 – 120.

Doherty, P.; Haslum, P.; Heintz, F.; Merz, T.; Persson, T.; and Wingman, B. 2004. A distributed architecture for intelligent unmanned aerial vehicle experimentation. In *Proc. 7th International Symposium on Distributed Autonomous Robotic Systems*.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: the language for the classical part of IPC-4. In *4th International Planning Competition Booklet*, 2 – 6. Available at `http://ipc.icaps-conference.org/`.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. 6th European Conference on Planning (ECP'01)*, 13 – 24.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, 274 – 283.

Erol, K.; Nau, D.; and Hendler, J. 1994. HTN planning: Complexity and expressivity. In *Proc. National Conference on Artificial Intelligence (AAAI'94)*, 1123 – 1128.

Erol, K.; Nau, D.; and Subrahmanian, V. 1991. Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. Technical Report CS-TR-2797, Computer Science Department, University of Maryland.

Fadel, F. G.; Fox, M. S.; and Gruninger, M. 1994. A generic enterprise resource ontology. In *Proc. of the 3rd IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*.

Felner, A.; Meshulam, R.; Holte, R.; and Korf, R. 2004. Compressing pattern databases. In *Proc. 19th National Conference on AI (AAAI'04)*, 638 – 643.

Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. 2005. Dual lookups in pattern databases. In *Proc. 19th International Conference on Artificial Intelligence (IJCAI'05)*.

Felner, A.; Korf, R.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of AI Research* 22:279 – 318.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189 – 208.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367 – 421.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning domains. In *Proc. 16th International Conference on Artificial Intelligence (IJCAI'99)*, 956 – 961.

Fox, M., and Long, D. 2000. Utilizing automatically inferred invariants in graph construction and search. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 102 – 111. AAAI Press.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of AI Research* 20:61 – 124.

Fukunaga, A.; Rabideau, G.; Chien, S.; and Yan, D. 1997. ASPEN: A framework for automated planning and scheduling of spacecraft control and operations. In *Proc. International Symposium on AI, Robotics and Automation in Space.*

Garey, M., and Johnson, D. 1979. *Computers and intractability: A guide to the theory of NP-completeness.* Freeman.

Garrido, A.; Onaindia, E.; and Barber, F. 2001. Time-optimal planning in temporal problems. In *Proc. 6th European Conference on Planning (ECP'01)*, 397 – 402.

Gaschnig, J. 1979. A problem similarity approach to devising heuristics: First results. In *Proc. 6th International Joint Conference on Artificial Intelligence (IJCAI'79)*, 301 – 307.

Gazen, B., and Knoblock, C. 1997. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *Proc. 4th European Conference on Planning (ECP'97)*, 223 – 235.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proc. 15th National Conference on Artificial Intelligence (AAAI'98)*, 905 – 912.

Gerevini, A., and Serina, I. 2002. LPG: A planner based on local search for planning graphs with action costs. In *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, 281 – 290.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. 2nd International Conference on AI Planning Systems (AIPS'94)*, 61 – 67.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice.* Morgan Kaufmann Publishers. ISBN: 1-55860-856-7.

Ginsberg, M., and Harvey, W. 1992. Iterative broadening. *Artificial Intelligence* 55(2-3):367 – 383.

Green, C. 1969. Applications of theorem proving to problem solving. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI'69)*, 219 – 240.

Gupta, N., and Nau, D. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56:223 – 254.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 140 – 149. AAAI Press.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proc. 6th European Conference on Planning (ECP'01)*, 121 – 132.

Haslum, P., and Scholz, U. 2003. Domain knowledge in planning: Representation and use. In *Proc. ICAPS 2003 workshop on PDDL.*

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proc. 20th National Conference on AI (AAAI'05)*, 1163 – 1168.

Haslum, P. 2004. Improving heuristics through search. In *Proc. European Conference on AI (ECAI'04)*, 1031 – 1032.

Haslum, P. 2006. Improving heuristics through relaxed search – an analysis of TP4 and HSP$_a^*$ in the 2004 planning competition. *Journal of AI Research* 25.

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, 303 – 312.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. 14th International Conference on Automated Planning & Scheduling (ICAPS'04)*, 161 – 170.

Hendler, J.; Tate, A.; and Drummond, M. 1990. AI planning: Systems and techniques. *AI Magazine* 11(2):61 – 77.

Hernadvölgyi, I., and Holte, R. 2000. Experiments with automatically created memory-based heuristics. In *In Proc. Abstraction, Reformulation, and Approximation, 4th International Symposium (SARA 2000)*, 281 – 290.

Hoffmann, J., and Edelkamp, S. 2005. The classical part of IPC-4: An overview. To appear in the *Journal of AI Research* (this Special Track).

Hoffmann, J., and Geffner, H. 2003. Branching matters: Alternative branching in Graphplan. In *Proc. 13th International Conference on Automated Planning & Scheduling (ICAPS'03)*, 22 – 31.

Hoffmann, J., and Koehler, J. 2000. Handling of inertia in a planning system. Technical Report 122, Institute for Computer Science, Albert Ludwigs University, Freiburg.

Hoffmann, J.; Edelkamp, S.; Englert, R.; Liporace, F. Thiébaux, S.; and Trüg, S. 2004. Towards realistic benchmarks for planning: The domains used in the classical part of IPC-4. In *4th International Planning Competition Booklet*, 7 – 14. Available at `http://ipc.icaps-conference.org/`.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of AI Research* 22:215 – 278.

Hoffmann, J. 2000. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proc. 12th International Symposium on Methodologies for Intelligent Systems (ISMIS'00)*, 216 – 227.

Holte, R.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proc. 13th National Conference on Artificial Intelligence (AAAI'96)*, 530 – 535.

Holte, R.; Newton, J.; Felner, A.; Meshulam, R.; and Furcy, D. 2004. Multiple pattern databases. In *14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, 122 – 131.

Hwang, Y. K., and Ahuja, N. 1992. Gross motion planning – a survey. *ACM Computing Surveys* 24(3):219 – 291.

Jonsson, A.; Morris, P.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in interplanetary space: Theory and practice. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 177 – 186.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence* 129(1-2):219 – 251.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of AI Research* 7:283 – 317.

Kambhampati, S. 1994. Comparing partial order planning and task reduction planning: A preliminary report. In *Working notes of the AAAI'94 Workshop on Comparative Analysis of Planning Systems.* Also available as Technical Report CSE 94-001, Arizona State University.

Kambhampati, S. 2000. Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan. *Journal of AI Research* 12:1 – 34.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proc. 10th European Conference on Artificial Intelligence (ECAI'92)*, 359 – 363.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proc. 13th National Conference on Artifical Intelligence.*

Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In *Proc. 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, 318 – 325.

Kautz, H. 2004. SATPLAN04: Planning as satisfiability. In *4th International Planning Competition Booklet*, 44 – 45. Available at `http://ipc.icaps-conference.org/`.

Koehler, J. 1998. Planning under resource constraints. In *Proc. 13th European Conference on Artificial Intelligence (ECAI'98)*, 489 – 493.

Kolisch, R., and Hartmann, S. 2005. Exprerimental investigation of heuristics for resource-constrained project scheduling: An update. To appear in *European Journal of Operational Research.*

Kolisch, R., and Sprecher, A. 1996. PSPLIB – a project scheduling problem library. *European Journal of Operational Research* 96:205 – 216.

Korf, R., and Taylor, L. 1996. Finding optimal solutions to the twenty-four puzzle. In *Proc. 13th National Conference on Artificial Intelligence (AAAI'96)*, 1202 – 1207.

Korf, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97 – 109.

Kvarnström, J., and Doherty, P. 2000. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30(1):119 – 169.

Laborie, P., and Ghallab, M. 1995. Planning with sharable resource constraints. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, 1643 – 1651.

Laborie, P. 1995. *IxTeT: Une Approche Intégrée pour la Gestion de Ressources et la Synthèse de Plans.* Ph.D. Dissertation, École Nationale Supérieure des Télécommunications. LAAS-CNRS Rapport No 95526.

Laborie, P. 2001. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. In *Proc. 6th European Conference on Planning (ECP'01)*, 205 – 216.

Liu, Y.; Koenig, S.; and Furcy, D. 2002. Speeding up the calculation of heuristics for heuristic search-based planning. In *Proc. 18th National Conference on Artificial Intelligence (AAAI'02)*, 484 – 491.

Manna, Z., and Waldinger, R. 1987. How to clear a block: A theory of plans. *Journal of Automated Reasoning* 3:343 – 377.

Manzini, G. 1995. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence* 75(2):347 – 360.

Martelli, A., and Montanari, U. 1973. Additive AND/OR graphs. In *Proc. 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, 1 – 11.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. 9th National Conference on Artificial Intelligence.*

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C. A.; Ram, A.; M., V.; Weld, D.; and Wikins, D. 1998. PDDL – the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control. `ftp://ftp.cs.yale.edu/pub/mcdermott/software/ pddl.tar.gz`.

McDermott, D. 1999. Using regression-match graphs to control search in planning. *Artificial Intelligence* 109:111 – 159.

Muscettola, N. 1994. Integrating planning and scheduling. In Zweben and Fox (1994).

Nareyek, A. 2001. Beyond the plan-length criterion. In *Local Search for Planning and Scheduling*, volume 2148 of *LNAI*. Springer Verlag. 55 – 78.

Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *Journal of AI Research* 12:271 – 315.

Nguyen, X.; Kambhampati, S.; and Nigenda, R. 2002. Planning graph as the basis for deriving heuristics for plan synthesis by state space and csp search. *Artificial Intelligence* 135(1-2):73 – 123.

Nilsson, N. J. 1968. Searching problem-solving and game-playing trees for minimal cost solutions. In *Proc. IFIP Congress*, 125 – 130.

Papadimitriou, C., and Steiglitz, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall.

Parker, E. 2004. Combining backward-chaining with forward-chaining AI search. In *4th International Planning Competition Booklet*, 51 – 52. Available at `http://ipc.icaps-conference.org/`.

Patriksson, M. 1994. *The Traffic Assignment Problem: Models and Methods*. Utrecht, The Netherlands: VSP.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pednault, E. 1988. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence* 4:356 – 372.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. International Confrence on Knowledge Representation and Reasoning (KR'92)*.

Penberthy, J., and Weld, D. 1994. Temporal planning with continous change. In *Proc. 12th National Conference on Artificial Intelligence (AAAI'94)*, 1010 – 1015.

Peot, M., and Smith, D. 1993. Threat-removal strategies for partial-order planning. In *Proc. 11th National Confrence on Artificial Intelligence*.

Plaat, A.; Schaeffer, J.; Pijls, W.; and de Bruin, A. 1996. Best-first fixed-depth minimax algorithms. *Artificial Intelligence* 87(1-2):255 – 293.

Prieditis, A. E. 1993. Machine discovery of effective admissible heuristics. *Machine Learning* 12:117 – 141.

Refanidis, I., and Vlahavas, I. 1999. A domain-independent heuristic for STRIPS worlds based on greedy regression tables. In *Proc. 5th European Conference on Planning (ECP'99)*, 347 – 359.

Refanidis, I., and Vlahavas, I. 2003. Mulitobjective heuristic state-space planning. *Artificial Intelligence* 145:1 – 32.

Reinfeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701 – 710.

Rintanen, J. 1998. A planning algorithm not based on directional search. In *Principles of Knowledge Representation and Reasoning: Proc. 6th International Conference (KR'96)*.

Ruml, W.; Do, M.; and Fromherz, M. 2005. On-line planning and scheduling for high-speed manufacturing. In *Proc. 15th International Conference on Automated Planning & Scheduling (ICAPS'05)*.

Sacerdoti, E. 1975. The nonlinear nature of plans. In *Proc. 4th International Joint Conference on Artificial Intelligence (IJCAI'75)*, 206 – 214.

Sandewall, E., and Rönnquist, R. 1986. A representation of action structures. In *Proc. National Conference on Artificial Intelligence (AAAI'86)*, 89 – 97.

Scholz, U. 2000. Extracting state constraints from PDDL-like planning domains. In *Proc. AIPS 2000 Workshop on Analyzing and Exploiting Domain Knowledge for Efficient Planning*, 43 – 48.

Shin, J., and Davis, E. 2005. Processes and continuous change in a SAT-based planner. *Artificial Intelligence* 166:195 – 254.

Slaney, J., and Thiebaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125. `http://arp.anu.edu.au:80/~jks/bw.html`.

Smith, S., and Becker, M. 1997. An ontology for constructing scheduling systems. In *Working Notes of the 1997 AAAI Symposium on Ontological Engineering*. AAAI Press.

Smith, D., and Weld, D. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, 326 – 333.

Smith, D.; Frank, J.; and Jonsson, A. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15(1).

Srivastava, B., and Kambhampati, S. 1999. Scaling up planning by teasing out resource scheduling. In *Proc. 5th European Conference on Planning (ECP'99)*, 172 – 186.

Tate, A.; Drabble, B.; and Dalton, J. 1994. The use of condition types to restrict search in an AI planner. In *Proc. 12th National Conference on Artificial Intelligence (AAAI'94)*, 1129 – 1134.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: An open architecture for command, planning and control. In Zweben and Fox (1994). 213 – 239.

Tate, A. 1977. Generating project networks. In *Proc. 5th International Joint Conference on Artificial Intelligence (IJCAI'77)*, 88 – 93.

Thiebaux, S.; Hoffmann, J.; and Nebel, B. 2003. In defense of PDDL axioms. In *Proc. 18th International Conference on Artificial Intelligence (IJCAI'03)*, 961 – 968.

Trinquart, R., and Ghallab, M. 2001. An extended functional representation in temporal planning: Towards continuous change. In *Proc. 6th European Conference on Planning (ECP'01)*, 217 – 228.

Trinquart, R. 2003. Analyzing reachability within plan space. In *Proc. of the ICAPS'03 Doctoral Consortium*, 122 – 126.

Valtorta, M. 1984. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences* 34:48 – 59.

van den Briel, M., and Kambhampati, S. 2004. Optiplan: Unifying IP-based and graph-based planning. In *4th International Planning Competition Booklet*, 18 – 20. Available at `http://ipc.icaps-conference.org/`.

Veloso, M.; Carbonell, J.; Perez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental AI* 7(1):81–120.

Vere, S. 1983. Planning in time: Windows and durations for activities and goals. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 5:246 – 267.

Verfaillie, G.; Lemaitre, M.; and Schiex, T. 1996. Russian doll search for solving constraint optimization problems. In *Proc. 13th National Conference on Artifical Intelligence (AAAI'96)*, 181 – 187.

Vidal, V., and Geffner, H. 2004. Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proc. 19th National Conference on Artificial Intelligence (AAAI'04)*, 570 – 577.

Weld, D. S. 1999. Recent advances in AI planning. *AI Magazine* 20(2):93 – 123.

Wilkins, D., and desJardins, M. 2000. A call for knowledge-based planning. In *Proc. AIPS Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning.*

Wilkins, D. 1983. Representation in a domain-independent planner. In *Proc. 8th International Joint Conference on Artificial Intelligence (IJCAI'83)*, 733 – 740.

Wilkins, D. 1990. Can AI planners solve practical problems? *Computational Intelligence* 6(4):232 – 246.

Williamson, M., and Hanls, S. 1994. Optimal planning with a goal-directed utility model. In *Proc. 2nd International Conference on Artificial Intelligence Planning Systems (AIPS'94)*, 176 – 181.

Wolfman, S., and Weld, D. 1999. The LPSAT engine & its application to resource planning. In *Proc. 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, 310 – 317.

Younes, H., and Simmons, R. 2002. On the role of ground actions in refinement planning. In *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, 54 – 61.

Zhou, R., and Hansen, E. 2004. BFHSP: A breadth-first heuristic search planner. In *4th International Planning Competition Booklet*, 61 – 63. Available at `http://ipc.icaps-conference.org/`.

Zweben, M., and Fox, M., eds. 1994. *Intelligent Scheduling.* Morgan-Kaufmann.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl**: Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea / ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin**: Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm**: Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson**: Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald**: Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel**: Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin**: Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder**: Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson**: Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson**: An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson**: Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre**: A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg**: Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund**: Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl**: Development of IT-supported Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm**: Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid**: Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin**: Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund**: Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linquistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson**: An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma**: Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache**: Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao**: Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum**: Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

Linköping Studies in Information Science

No 1 **Karin Axelsson**: Metodisk systemstrukturering- att skapa samstämmighet mellan informa-tionssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm**: Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic**: Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson**: Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind**: Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin**: Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk**: Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth**: Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström**: Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf**: Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson**: Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström**: Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson**: Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.