# Fast Detection of Unsolvable Planning Instances Using Local Consistency

**Christer Bäckström, Peter Jonsson** and **Simon Ståhlberg**
Department of Computer Science, Linköping University
SE-581 83 Linköping, Sweden
christer.backstrom@liu.se peter.jonsson@liu.se simon.stahlberg@liu.se

**Abstract**

There has been a tremendous advance in domain-independent planning over the past decades, and planners have become increasingly efficient at finding plans. However, this has not been paired by any corresponding improvement in detecting unsolvable instances. Such instances are obviously important but largely neglected in planning. In other areas, such as constraint solving and model checking, much effort has been spent on devising methods for detecting unsolvability. We introduce a method for detecting unsolvable planning instances that is loosely based on consistency checking in constraint programming. Our method balances completeness against efficiency through a parameter $k$: the algorithm identifies more unsolvable instances but takes more time for increasing values of $k$. We present empirical data for our algorithm and some standard planners on a number of unsolvable instances, demonstrating that our method can be very efficient where the planners fail to detect unsolvability within reasonable resource bounds. We observe that planners based on the $h^m$ heuristic or pattern databases are better than other planners for detecting unsolvability. This is not a coincidence since there are similarities (but also significant differences) between our algorithm and these two heuristic methods.

## 1 Introduction

There has been an impressive advancement in domain-independent planning over the past decades. New and efficient planners have entered the scene, for instance, FAST FORWARD (Hoffmann and Nebel 2001), FAST DOWNWARD (Helmert 2006b), LAMA (Richter and Westphal 2010), and planners based on compilation into SAT instances (Rintanen, Heljanko, and Niemelä 2006). Various ways to exploit the structure of planning instances have been proposed and integrated into existing planners. Perhaps most importantly, a rich flora of heuristics has appeared, making many previously infeasible instances solvable with reasonable resources. See, for instance, Helmert and Domshlak (2009) for a survey and comparison of the most popular heuristics. This development has largely been driven by the international planning competitions (IPC). However, in the long run one might question how healthy it is for a research area to focus so much effort on solving the particular problems choosen for the competitions. This is a very natural question from a methodological point of view, and although it has not been raised as often as one might expect, it is not a new one, cf. Helmert (2006a).

One problem with the planning competitions so far is that all planning instances have a solution, and the effect of this is that the planners and methods used are getting increasingly faster at finding the solutions that we already know exist. It is important to note that other similar competitions, such as the SAT competition (`http://www.satcompetition.org`) and the Constraint Solver Competition (`http://cpai.ucc.ie`), consider unsolvable instances together with solvable instances. It is no secret that few, if any, planners exhibit any similar speed improvement for instances that have no solution. For many applications we can know, e.g. by design, that there is a solution, but this cannot be a general assumption. Obvious examples are planning for error recovery and planning for systems that were not designed to always have a solution (old industrial plants that have evolved over time is an archetypical example). Another example is support systems where a human operator takes the planning decisions and the planner is used to decide if there is a plan or not, or to attempt finding errors in the human-made plan (Goldman, Kuter, and Schneider 2012). System verification has an overlap with planning (Edelkamp, Leue, and Visser 2007), and proving that a forbidden state is unreachable corresponds to proving that there is no plan. Similarly, planning is used for penetration testing in computer security (Boddy et al. 2005; Sarraute, Buffet, and Hoffmann 2012; Futoransky et al. 2010; Harp et al. 2005), which is an area of emerging commercial importance (Sarraute and Pickering 2012). In penetration testing, a system is considered safe if there is no plan for intrusion. We finally note that *oversubscribed planning* (i.e. where not all goals can be satisfied simultaneously and the objective is to maximize the number of satisfied goals) have been considered in the literature (Smith 2004). Clearly, this is a source of relevant unsolvable instances.

This paper is concerned with unsolvable planning instances and how to efficiently detect them. We present an algorithm for this problem which is loosely based on consistency checking in CSPs. Our method is often very efficient, but at the cost of being incomplete, that is, it is infeasible

to detect all unsolvable instances. To the best of our knowledge, this is the first time such a method is analysed and tested for this pupose in planning (although it is, of course, implicitly used when planning instances are reformulated into SAT or CSP instances and solved by such techniques). Like most such methods, our algorithm also runs in polynomial time for any fixed value of a parameter $k$, which can thus be used to trade off between efficiency and completeness. We also present empirical data for a number of unsolvable planning instances, matching our algorithm against some of the most commonly used planners. There are also some heuristic methods, such as iPDB (Haslum et al. 2007) and $h^m$ (Haslum and Geffner 2000), that have certain similarities with our method, and that can also be used for consistency checking. Hence, we include also these methods in the empirical analysis. The overall result is that our algorithm had the best coverage, detecting all instances as unsolvable, although iPDB was a faster method when it worked. Only one instance could be detected as unsolvable by the planners, requiring much more time than our method. There are of course unsolvable instances that our method may fail to capture given reasonable resource bounds and we give concrete examples of when and why this may happen in Sections 3.3 and 4.1. Yet, our results are encouraging: consistency checking is very cheap for moderate values of $k$, so it seems obvious to recommend using such methods routinely, either integrated into the planner or as a preprocessing step.

The rest of the paper is structured as follows. Sec. 2 presents our consistency concept, the algorithm which is based on it, and the computational complexity of this algorithm. Sec. 3 presents the unsolvable test instances we use as well as the planners and other methods that we compare our algorithm against. It also presents and discusses the empirical data from our experiments. Sec. 4 deepens the previous complexity analysis of the consistency checking problem and discusses methods for improving its performance. The paper ends (in Sec. 5) with an discussion that mostly focuses on future research directions.

## 2 Local Consistency Checking

We assume the reader is familiar with the SAS$^+$ planning formalism. Briefly, a *SAS$^+$ instance* is a tuple $\Pi = \langle V, A, I, G\rangle$ where $V$ is a set of multi-valued variables, $A$ is a set of *actions*, $I$ is the *initial state* and $G$ is the *goal*. Given a variable $v \in V$, we let $D(v)$ denote its corresponding domain. Each action $a \in A$ has a *precondition* $\mathrm{pre}(a)$ and an *effect* $\mathrm{eff}(a)$. A solution plan for $\Pi$ is a sequence of actions that can be executed in the initial state $I$ and leads to a state that satisfies the goal $G$. We sometimes write $V(\Pi)$ for the set of variables of $\Pi$, $A(\Pi)$ for the actions of $\Pi$ and so on.

We further define *variable projection* in the usual way (Helmert 2004). Let $\Pi = \langle V, A, I, G\rangle$ be a SAS$^+$ instance and let $V' \subseteq V$. Then the *variable projection* of $\Pi$ onto $V'$ is $\Pi|_{V'} = \langle V', A|_{V'}, I|_{V'}, G|_{V'}\rangle$, where $I|_{V'}$ is the restriction of $I$ to the variables in $V'$ etc. For actions, $a|_{V'}$ restricts $\mathrm{pre}(a)$ and $\mathrm{eff}(a)$ to $V'$.

It is well known (Helmert 2004) that if the projection of a SAS$^+$ instance onto a subset of its variables is unsolvable, then also the original instance is unsolvable. To find that an instance $\Pi$ is unsolvable it is thus sufficient to demonstrate one single subset $V'$ of the variables $V(\Pi)$ for $\Pi$ such that the projection $\Pi|_{V'}$ is unsolvable. This is a very powerful fact, but one that comes at a cost; deciding if such a subset exists is obviously as hard as deciding if $\Pi$ is solvable. In order to gain any efficiency at all from this idea, we must thus necessarily settle for an incomplete method. Based on the assumption that there will often exist a small subset of variables that suffices as a witness for unsolvability, the basic principle of our method is to check all subsets of variables up to some predefined size $k$. An obvious use of this method is as a preprocessing stage, attempting to decide if the instance is unsolvable before even invoking the planner to search for a plan. The parameter $k$ can be used to balance time against coverage; a higher value of $k$ requires more time but also catches more unsolvable instances.

**Definition 1.** A SAS$^+$ instance $\Pi = \langle V, A, I, G\rangle$ is *variable $k$-consistent*, for an integer $k$ such that $0 < k \leq |V|$, if $\Pi|_{V'}$ has a solution for every variable set $V' \subseteq V$ of size $k$.

We define the *consistency checking problem* as taking a SAS$^+$ instance $\Pi = \langle V, A, I, G\rangle$ and a positive integer $k \leq |V|$ as input and asking the question whether $\Pi$ is variable $k$-consistent? Clearly, the number of subsets to check grows quickly with the value of $k$. If there are $n$ variables in total, then there are $\binom{n}{k}$ subsets of size $k$, so checking consistency is usually only feasible for small values of $k$. If an instance is consistent for a particular $k$, then it is also consistent for all $k' < k$ by definition, so if we have decided on a fixed $k$ to check consistency for, it is sufficient to check all subsets exactly of size $k$. In practice, however, it may be desirable to make an iterative computation that checks also all smaller sets in order of increasing size, since the algorithm may often terminate early, for some set of smaller size than $k$. Furthermore, if we do not fix $k$ in advance, then we can check consistency for increasing values of $i$ until the algorithm is stopped or hits a resource limit. The longer it runs, the greater the chance that it will detect that an unsolvable instance is unsolvable. This iterative algorithm can thus be used as an *anytime algorithm*.

The straightforward iterative algorithm appears in Figure 1 (note that 'succeed' means that $\Pi$ is consistent for $k$). The test in line 4 can be performed by any algorithm that checks whether two vertices are connected in a directed graph or not: suitable standard algorithms include Dijkstra's algorithm and depth-first search. Given the instance $\Pi|_{V'}$, first construct the corresponding state-transition graph. This graph can be constructed in polynomial time and it contains at most $d^k$ vertices where $d$ is the size of the largest variable domain. Secondly, check whether there exists a path from $I(\Pi)|_{V'}$ to some state consistent with $G(\Pi)|_{V'}$ or not. It may be advisable, for increased efficiency, to introduce a dummy goal vertex $g$ into the graph and add an edge from each goal state to $g$. By doing so, repeated application of the search algorithm is avoided. One may also observe that every sound and complete planner can be used for the test in line 4 and this may be a good idea when $k$ is large since the explicit construction of the state-transition graph is avoided.

We now consider the time complexity of line 4 of the al-

```
1   ConsistencyCheck(Π,k)
2     for i from 1 to k do
3       for all V' ⊆ V(Π) s.t. |V'| = i do
4         if Π|_{V'} is unsolvable then fail
5     succeed
```

Figure 1: The brute-force iterative consistency algorithm.

gorithm. Constructing the state-transition graph can be done in time $O((d^k+1)^2 \cdot |A|)$ as follows. First introduce $d^k+1$ vertices; the $d^k$ vertices denote the states of the projected instances and the additional vertex is the special vertex for handling goal states. Now, for each distinct pair of vertices $s, t$, check whether there is a projected action in $A$ that transforms state $s$ into state $t$ or not, and if this is the case, add a directed edge from $s$ to $t$. Also add an edge from each vertex consistent with the goal state to the goal vertex. After having constructed this graph, we can check whether there is a path from the initial state to the goal state by, for example, using Dijkstra's $O(|V|^2)$ algorithm. Hence, the total time for line 4 is $O((d^k+1)^2 \cdot |A|) + (d^k+1)^2) = O((d^k+1)^2 \cdot |A|)$. The total time for checking consistency for one value of $k$, i.e. one iteration of the algorithm, for an instance $\Pi$ with $n$ variables and $m$ actions is thus $O(\binom{n}{k} \cdot (d^k+1)^2 \cdot m) = O(\binom{n}{k} \cdot d^{2k} \cdot m)$ which is polynomial in the size of $\Pi$ for fixed $k$ since $\binom{n}{k} \in O(n^k)$.

The algorithm in Figure 1 is reasonably efficient for small values of $k$ but it can easily be improved by using well-known and obvious techniques. We will consider the following two straightforward methods for filtering which subsets to check.

(1) A projection without any goal variables is solved by the empty plan, so it is sufficient to check projections containing at least one goal variable.

(2) A variable projection induces a subgraph of the causal graph which need not be weakly connected even if the causal graph is. The components of the subgraph define independent subinstances, so if a projection to $k$ variables induces more than one component, it suffices to check consistency for each component separately. As a further improvement, our implementation traverses the causal graph to generate only those sets that are connected, thus avoiding to even consider the non-connected ones.

# 3 Experiments

Our consistency-checking method is novel to the area of planning, so the primary purpose of our experiments is a proof of concept. We want to demonstrate that our method can be very effective in cases where standard methods fail. Since we are not aware of any standard sets of benchmarks with unsolvable instances, we use a mix of IPC instances that are modified to be unsolvable and new instances designed from scratch, intended to display a diversity in why they are unsolvable. We acknowledge that this is a small and probably biased set of test instances but we hope that the experiments are illuminating anyway. The lack of unsolvable test examples and the resulting difficulties in planner evaluation are discussed in Section 5.

This section is divided into four parts: we present the test instances, we discuss the methods tested, we present and discuss the empirical data and, finally, we discuss some similarities and differences between methods.

## 3.1 Test Instances

We used the following test instances.

**Trucks:** An unsolvable variant of the IPC Trucks instances, with two independent parts (see Figure 2). The left

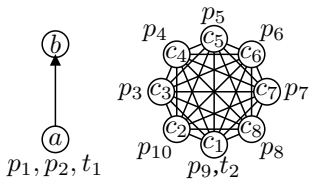
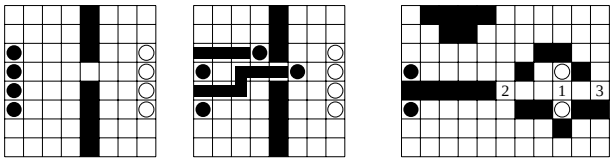



Figure 2: The Trucks instance.

part has two locations, $a$ and $b$, and a one-way path from $a$ to $b$. There is one truck and two packages at $a$ and the packages must both be delievered at $b$. The truck can only carry one package at a time so it cannot satisfy its goal. The right part has 8 locations interconnected to form a clique. There is one truck and 8 packages. The truck must pick up every package and deliver it at another location, but this is easy due to the structure of the road graph.

**Tiles:** A grid-based path planning problem with mutual-exclusion constraints (see Figure 3). Agents move from their initial positions to their individual goal positions, one tile at a time, moving up, down, left or right. An agent can only move to a white tile, and every tile it visits turns black and becomes impassable for the other agents. The positions of




a) Tiles I (initial state) b) Tiles I (a deadlock state) c) Tiles II (initial state)

Figure 3: The Tiles instances.

the agents are marked with filled circles and their goal positions with unfilled circles. In the initial state of instance Tiles I (Figure 3a) the initially black tiles form a wall with a one-tile wide passage. Only one agent can pass the wall since the passage will also be black afterwards, meaning that only one agent can reach its goal. Figure 3b illustrates this. Instance Tiles II (Figure 3c) has two constraining tiles, marked 1 and 2. Number 1 always causes a deadlock, while number 2 can cause a deadlock or not, depending on whether the upper agents goes via tile 2 or 3. The number of agents is not important for consistency checking, as long as there are enough of them. Two agents are sufficient in a projection of Tiles I to spot that it is inconsistent, since only one agent can pass the wall.

**Grid:** An unsolvable variant of the IPC Grid problem, with a grid of rooms and a robot that can move around and move things. In particular, the robot may fetch keys that are required to open doors to certain rooms. Two rooms, A and B, both require a key to enter, but the key to room A is in room B and vice versa, so the robot cannot get any of these keys. There is also another key to room A, but if the robot goes to fetch it, then it cannot get back to room A.

**3SAT:** PDDL encodings of unsolvable 3SAT instances generated using code from an existing project (Muise 2010). Instance 3SAT I has 15 variables and 17 clauses, and it is on the form $x_1 \wedge \overline{x_1} \wedge c_1 \wedge \ldots \wedge c_{15}$ where $c_1 \wedge \ldots \wedge c_{15}$ is a satisfiable 3CNF formula. Instance 3SAT II is a formula with 5 variables and 25 clauses that is less obviously unsatisfiable.

### 3.2 Methods Tested

Our consistency checking algorithm was implemented in C#. This was primarily a pilot study of the method, so we did not attempt much optimization of the algorithm. The actual consistency checking of a particular projection (line 4 of the algorithm in Figure 1) is done using a basic depth-first search algorithm. Our algorithm works best with multi-valued variables but all test instances were coded in PDDL so we had to translate them. To this end we used the translator module of the Fast Downward planner (Helmert 2006b), which also uses multi-valued variables internally.

For comparison, we tried a number of standard planners: FAST FORWARD (FF) (Hoffmann and Nebel 2001), FAST DOWNWARD (FD) (Helmert 2006b) and LAMA (Richter and Westphal 2010). Fast Downward was tested in two configurations, seq-sat-fd-autotune-1 and seq-sat-fd-autotune-2, referred to as FD1 and FD2, respectively. It is reasonable to assume that the heuristics of the planners will often do more harm than good on unsolvable instances. Hence, we also tested breadth-first search by using the A$^*$ algorithm in Fast Downward equipped with a blind heuristic.

There are some methods that are primarily intended for planning that are reasonable to use also for detecting unsolvable instances. An admissible heuristic function underestimates the cost to the goal and is used to guide the search algorithm to find a solution faster. The cost to the goal is infinite when there is no solution, so the planner can terminate if the heuristic estimate is infinite. However, this is also an incomplete method since the heuristic estimate will often be lower than the true cost and need not be infinite. We have, thus, also tested two heuristic methods that can be used for the same purpose as our method, pattern databases and $h^m$.

The variable projection of an instance $\Pi$ to a subset of its variables is a new instance that is a state abstraction of $\Pi$. It is common to exploit this by choosing a subset of variables and compute the heuristic estimate between two states as the true cost (or length) of a plan between their corresponding abstract states in the abstract instance. This is an admissible heuristic. It is also common to compute the costs between all abstract states and store them in a so called *pattern database (PDB)* (Culberson and Schaeffer 1998). In practice, one often builds a number of different PDBs, using different variable subsets, and compute a single heuristic function from these that is better than what each of the PDBs alone can provide. One such method is iPDB (Haslum et al. 2007), which we have chosen for our experiments.

The $h^m$ heuristic (Haslum and Geffner 2000), where $m \geq 1$, is a family of admissible heuristics. For a given $m$, $h^m$ is the maximum cost of achieving any subset of $m$ goal atoms by regression, applying the same criterion also to all subgoals. We used the built-in $h^m$ module of Fast Downward and applied it to the inital state of the given instance. If this state is identified as a dead-end, then the instance is clearly unsolvable. The implementation is considered very inefficient so we are slightly surprised that this approach is competitive when it succeeds.

### 3.3 Experiments and Results

All tests were run a computer with an Intel Core i5 3570K 3.4 GHz CPU and 4 GB available RAM. Both the implemented algorithm and all test examples are available upon request. All methods were given a time limit of 30 minutes on each instance, and were considered to fail if they either ran out of time or memory. The empirical data for all methods is listed in Table 1. The figures in the table denote the time for detecting that the instance is unsolvable, while failure to detect unsolvability is marked with either – M – or – T –, depending on whether the memory or the time limit was hit first. Our consistency algorithm was run iteratively with both filters until it detected unsolvability, and the figures denote total time for all iterations until it terminated. The iPDB method needs to set a parameter num_samples, which controls how many states it samples when comparing two projections. The figures in the table use manually tuned values for each instance, but using the default value of 1000 was only marginally slower. We tried values higher than $10^6$ for those instances that were not detected as unsolvable, which did not help. There is also a parameter min_improvement to control how much improvement a new candidate must give for the algorithm to continue. This parameter was set to 1, to force the algorithm not to give up too early. For $h^m$, we tried $m$ values of 1 to 5, since 5 was high enough to run out of time or memory for all instances.

Our consistency checking algorithm detected all 6 instances as unsolvable, iterating to $k = 7$ for one instance. The iPDB method could only detect 3 of the 6 instances as unsolvable, but, on the other hand, it was considerably faster than our method on those instances. The $h^m$ family of methods together detected 4 of the instances as unsolvable, but it failed on one instance that iPDB could handle. Furthermore, the $h^m$ approach was slower than our method in all but one case. We finally note that Grid was the only instance that any of planners could detect as unsolvable, and that all of them but FF managed this. We also note that blind search was considerably faster than all other planners, which supports the assumption that the heuristics of the planners are often more of a burden than an asset for unsolvable instances.

More detailed data for our algorithm is provided in Table 2. The table lists two cases for each instance, the brute-force variant with no filters and the variant where both filters 1 and 2 are used. The data is also broken down to show the time and number of subsets to explore for each iteration, up

| Method | 3SAT I | 3SAT II | Tiles I | Tiles II | Trucks | Grid |
|---|---|---|---|---|---|---|
| cons. check. | 15 ms | 10.1 s | 61.5 s | 288 s | 106 ms | 32.1 s |
| iPDB | – M – | – M – | 360 ms | 690 ms | – M – | 1.96 s |
| $h^1$ | – T – | – T – | – T – | – T – | – T – | – T – |
| $h^2$ | 219 ms | – T – | 100 s | – T – | 440 ms | 8.38 s |
| $h^3$ | 12.6 s | – T – | – T – | – T – | 35.8 s | 24:30 min |
| $h^4$ | 13:58 min | – T – | – M – | – M – | – T – | – T – |
| $h^5$ | – M – | – T – | – M – | – M – | – T – | – T – |
| FF | – M – | – T – | – T – | – M – | – T – | – T – |
| FD1 | – M – | – M – | – M – | – M – | – M – | 18:41 min |
| FD2 | – M – | – M – | – M – | – M – | – M – | 16:49 min |
| LAMA | – M – | – M – | – M – | – M – | – M – | 25:33 min |
| Blind | – M – | – M – | – M – | – M – | – M – | 10 s |

Table 1: Empirical data for the methods tested. Figures denote time for detecting unsolvability, while – M – and – T – denotes failure due to memory and time limit, respectively.

| $k$ | 3SAT I | | 3SAT II | | Tiles I | | Tiles II | | Trucks | | Grid | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 8 ms | 47 | 4 ms | 21 | 123 ms | 67 | 42 ms | 70 | 7 ms | 14 | 42 ms | 13 |
| 2 | 176 ms | 1081 | 26 ms | 210 | 6.24 s | 2211 | 2.01 s | 2415 | 53 ms | 91 | 365 ms | 78 |
| 3 | 3.58 s | 16215 | 233 ms | 1330 | 178 s | 47905 | 63.1 s | 54740 | 378 ms | 364 | 1.92 s | 286 |
| 4 | 535 ms | 178365 | 1.48 s | 5985 | | | 1241 s | 916895 | 173 ms | 1001 | 6.75 s | 715 |
| 5 | | | 6.96 s | 20349 | | | | | | | 18.7 s | 1287 |
| 6 | | | 24.9 s | 54264 | | | | | | | 29.5 s | 1716 |
| 7 | | | 69.1 s | 116280 | | | | | | | | |
| 1 | 1 ms | 17 | 1 ms | 15 | 16 ms | 5 | 3 ms | 2 | 2 ms | 10 | 0 ms | 1 |
| 2 | 3 ms | 44 | 3 ms | 37 | 1.17 s | 310 | 277 ms | 136 | 12 ms | 20 | 49 ms | 10 |
| 3 | 10 ms | 105 | 16 ms | 166 | 60.3 s | 10075 | 12.4 s | 4624 | 89 ms | 68 | 429 ms | 49 |
| 4 | 1 ms | 235 | 84 ms | 633 | | | 275 s | 102510 | 3 ms | 141 | 1.82 s | 155 |
| 5 | | | 413 ms | 2254 | | | | | | | 7.76 s | 348 |
| 6 | | | 1.86 s | 7012 | | | | | | | 22.1 s | 575 |
| 7 | | | 7.73 s | 18414 | | | | | | | | |

Table 2: Detailed data for consistency checking, without filters (top box) and with both filters (bottom box). The figures for each $k$ value denote time and number of subsets to explore for this value.

to the $k$ value where inconsistency was detected. For example, the Trucks instance is consistent for all $k \leq 3$ but is inconsistent for $k = 4$, so the algorithm was never run for higher values of $k$. While the running time of the algorithm on each instance increases monotonically with the value of $k$, as would be expected, it does sometimes drop dramatically for the highest $k$ value. This is also to be expected; if the instance is consistent for $k$, then it must check all (possibly after filtering) variable subsets of size $k$. On the other hand, if the instance is inconsistent for $k$, then the algorithm only has to run until it finds a subset of size $k$ that is inconsistent. Hence, the time needed depends on the order in which the subsets are checked. The number of subsets given in the table is the total number of sets to check, i.e. not all of these sets were checked for the highest $k$ value.

Finally, Table 3 gives some examples of how the effect of the two filtering methods can differ between instances. For example, method 1 has negligible effect on 3SAT II, so all filtering effects are due to filter 2. On Tiles II, however, the number of subsets is the same for all filter combinations, meaning that the two filters choose exactly the same subsets. Finally, Grid is an example where both filters contribute to a smaller number of subsets than either filter alone does.

| filters | 3SAT II (k=6) | Tiles II (k=3) | Grid (k=5) |
|---|---|---|---|
| none | 54264 | 54740 | 1287 |
| 1 | 54263 | 4624 | 495 |
| 2 | 7012 | 4624 | 857 |
| 1+2 | 7012 | 4624 | 348 |

Table 3: Examples of the effect of the filters.

At this point, it is important to remind the reader that local consistency checking is an incomplete method. The incompleteness is entirely due to the fact that some unsolvable instances are still consistent for large values of $k$. To illustrate this, consider an instance with binary variables $v_0, \ldots, v_{n-1}$ and actions $a_0, \ldots, a_{n-1}$ where $a_i$ sets variable $v_{((i+1) \mod n)}$ if $v_i$ is set, i.e. the causal graph is a cycle over the variables. Suppose all variables are initially false and the goal is to set one variable, which has no solution. Any projection to fewer than $n$ variables has some

action with empty precondition, making the projection solvable. This instance is, thus, consistent for all $k < n$ but not for $k = n$ so consistency checking is not very helpful in this case.

### 3.4 Similarities between Methods

The only methods tested that could compete with our method were iPDB and $h^m$. This is, perhaps, not very surprising since there are similarities between these methods and ours.

The iPDB method is similar to ours in that it considers a number of different variable projections and computes the true plan length in each, but there are also differences. Our method systematically checks subsets of a certain size $k$, so it is guaranteed to detect that a $k$-inconsistent instance is unsolvable. The iPDB approach chooses a smaller number of subsets according to some method that is intended to improve the resulting heuristic function. Given the same amount of time, it will thus check fewer but larger subsets; it may find that an instance is $m$-inconsistent but $(m-1)$-consistent for some $m$ larger than $k$, but at the same time miss that an instance is inconsistent already for some value smaller than $k$. This method is thus more opportunistic and less systematic than consistency checking.

Some kind of consistency checking is inherent also in the $h^m$ method but it is less straightforward to compare this approach to ours. Consequently, we leave such a comparison for future work.

## 4 Improving Consistency Checking

Since the consistency checking algorithm is highly systematic (at least without the filtering methods activated) and basically only performs an exhaustive enumeration of subinstances, there is no reason to expect any surprises concerning its scaling properties: its behaviour on instances that are $k-1$-consistent but not $k$-consistent can be closely bounded from above by the worst-case time complexity analysis in Section 2. This actually holds also when using the filtering methods: the methods we have used are so computationally efficient that it is very rare that they slow down the consistency checking algorithm. With this in mind, it is not hard to see that consistency checking may take a considerable amount of time (in the worst case) when the constant $k$ is large, and that this may very well happen when applied to real-world examples. It is consequently interesting to think about ways of improving consistency checking. Hence, we will now consider the possibility of improving consistency checking from both positive and negative angles.

(1) When $k$ is large, then the time needed for performing consistency checks increases rapidly (albeit polynomially) with the size of instances. One way to improve the situation is thus to use methods that can identify unsolvability at lower values of $k$. In Section 4.1, we present such a method based on *mutexes*. Mutexes have earlier been used in various ways (cf. Haslum, Bonet, and Geffner (2005) and Zilles and Holte (2010)) and they are explicitly constructed and used by the Fast Downward planner. We show that this is a viable idea and we demonstrate it on a blocks world example.

(2) Although variable $k$ consistency can be checked in polynomial time for every fixed $k$, it is problematic that the degree of the polynomial grows with the constant $k$. However, it may be the case that we could solve the problem in a way that does not scale as badly with the value of $k$ by using some other algorithm. By exploiting *parameterised complexity theory*, we show that there is no consistency checking algorithm running in $O(f(k)\cdot n^c)$ time for some function $f$ and constant $c$, unless an unlikely collapse of complexity classes occur. In other words, we can expect every given consistency checking algorithm to run in $O(n^{g(k)})$ time for some function $g$ that is not bounded from above and, consequently, have scaling properties comparable to those of our algorithm. This result is presented in Section 4.2.

### 4.1 Mutex Groups

Given a planning instance $\Pi = (V, A, I, G)$, we say that a *proposition* is a pair $(v, d)$ where $v \in V$ and $d \in D(V)$. A *mutex group* $M$ is a set of propositions such that for any state $s$ reachable from $I$, there is at most one proposition $(v, d) \in M$ such that $v$ equals $d$ in $s$. Consider a projection $\Pi|_{V'}$ for some $V' \subseteq V$. If some state in this instance violates the mutex condition (with respect to $M$), then we can safely remove it from further consideration. In particular, we can remove it from the state-transition graph that we consider in line 4 of the consistency checking algorithm. Note that similar ideas have been used elsewhere, cf. Haslum, Bonet, and Geffner (2005). This makes the state-transition graph smaller (which may be beneficial) but, more importantly, it may disconnect the initial state from the goal states and allow us to conclude that there is no solution. In fact, we will exhibit unsolvable instances that are $k$-consistent but not $k+1$-consistent having the following property: mutex-enhanced consistency checking discovers inconsistency for subsets of size $k'$ where $k' < k+1$. That is, there is a potential for significant improvements.

The implementation of the mutex-enhanced consistency algorithm is based on the previously presented consistency algorithm, i.e. we use a simple depth-first search algorithm and we have both filters activated. We check in every state considered by the search algorithm (i.e. line 4 in Figure 1) whether any of the mutex groups has two or more propositions true at the same time. If this is the case, then we backtrack and, thus, avoid considering plans which violate mutex groups. Obviously, this is equivalent with first constructing the state-transition graph and then removing certain vertices as described above; the chosen method has proven to be more computationally efficient, though. The mutex groups we use in our experiments are automatically provided by the translator module in the Fast Downward planner.

We tested this idea on a number of unsolvable blocks world instances. The test instances are unsolvable because the goal is a physically impossible configuration, namely, we have that block $A$ has to be on top of block $B$ and $B$ has to be on top of $A$, but the rest of the blocks have solvable goals. These instances are 5-consistent but not 6-consistent and they are interesting since consistency checking is not particularly useful for identifying that they are unsolvable. This can be seen in Table 4 where we apply a selection of

| | BW 4 | BW 8 | BW 16 | BW 32 | BW 64 |
|---|---|---|---|---|---|
| cons. check. with mutex groups | 9 ms | 100 ms | 1.2 s | 18.5 s | 5:05 min |
| cons. check. without mutex groups | 493 ms | – M – | – M – | – M – | – M – |
| iPDB | 180 ms | 2.2 s | – T – | – T – | – T – |
| $h^2$ | 97 ms | 691 ms | 28.5 s | – T – | – T – |
| Blind | 71 ms | 1.8 s | – M – | – M – | – M – |

Table 4: How different methods performs given a number of similar blocks world instances of different size.

| | $k=1$ | $k=2$ | $k=3$ | $k=4$ | $k=5$ | $k=6$ |
|---|---|---|---|---|---|---|
| with mutex groups | 0 ms | 4 ms | 5 ms | | | |
| without mutex groups | 0 ms | 4 ms | 10 ms | 26 ms | 73 ms | 380 ms |

Table 5: How consistency checking performs with and without mutex groups for the blocks world example with 4 blocks.

methods to instances containing 4, 8, 16, 32, and 64 blocks. Obviously, consistency checking is inferior when compared to the other methods. However, when consistency checking is enhanced with mutex groups, then it undoubtedly shows the best performance among these methods. The heuristic $h^2$ performs the best out of all the other methods, but is only able to identify the 16 block instance as unsolvable, while consistency checking with mutex groups is able to identify instances up to 4 times this size as unsolvable. iPDB and blind search are only able to identify instances of 8 blocks as unsolvable before running out of time and memory, respectively. Table 5 gives additional information concerning the difference between using mutex groups and not using them. The difference is huge and this is mainly due to the fact that inconsistency is found at a lower $k$ value when using mutex groups, which is very promising.

### 4.2 Parameterised Complexity Analysis

We begin by defining the basic notions of parameterised complexity theory and refer to other sources (Downey and Fellows 1999; Flum and Grohe 2006) for an in-depth treatment. A *parameterized problem* is a set of *instances* $\langle \boldsymbol{I}, k\rangle$, where $\boldsymbol{I}$ is the main part and $k$ the *parameter* (usually a non-negative integer). The problem is *fixed-parameter tractable (FPT)* if there is an algorithm that solves any instance $\langle \boldsymbol{I}, k\rangle$ of size $n$ in time $f(k) \cdot n^c$, where $f$ is an arbitrary computable function and $c$ is a constant independent of both $n$ and $k$. Note that this expression is separable in the sense that $f(k)$ must not depend on $n$ and $n^c$ must not depend on $k$. **FPT** is the class of all fixed-parameter tractable decision problems. There are also various classes used to prove that problems are hard, eg. the class **W**[1] which is often considered as a parameterised analogue of **NP**. A parameterized problem $L$ reduces to a parameterized problem $L'$ if there is a mapping $R$ from instances of $L$ to instances of $L'$ such that (1) $\langle \boldsymbol{I}, k\rangle$ is a YES-instance of $L$ if and only if $\langle \boldsymbol{I}', k'\rangle = R(\boldsymbol{I}, k)$ is a YES-instance of $L'$, (2) there is a computable function $f$ and a constant $c$ such that $R$ can be computed in time $O(f(k) \cdot n^c)$, where $n$ denotes the size of $\langle \boldsymbol{I}, k\rangle$ and (3) there is a computable function $g$ such that $k' \leq g(k)$. If variable $k$-consistency is in **FPT**, then there is an algorithm that might scale badly in $k$ but not in the combination of $k$ and $n$, as the polynomial $n^k$ does. Unfortunately, the problem is **W**[1]-hard when using $k$ as the parameter, and thus unlikely to be in **FPT**.

**Theorem 2.** *Variable $k$-Consistency is **W**[1]-hard when parameterised by $k$.*

*Proof.* We present a parameterised reduction from Independent Set, which is **W**[1]-hard (Downey and Fellows 1999), to Variable $k$-Consistency. Let $\langle V, E\rangle$ and $k \geq 0$ denote an arbitrary instance of Independent Set and assume $V = \{v_1, \ldots, v_n\}$. Construct a planning instance $\Pi = \langle W, A, I, G\rangle$ as follows. Define the actions $b_i^j : \{w_i = s\} \Rightarrow \{w_i = v_j\}$ and $c_{ij}^{lm} : \{e_{ij} = 0, w_i = v_l, w_j = v_m\} \Rightarrow \{e_{ij} = 1\}$, and let

- $W = \{w_1, \ldots, w_k\} \cup \{e_{ij} \mid 1 \leq i < j \leq k\}$
- $D(w_i) = \{s, v_1, \ldots, v_n\}, 1 \leq i \leq k$
- $D(e_{ij}) = \{0, 1\}, 1 \leq i < j \leq k$
- $A = \{b_i^j \mid 1 \leq i \leq k,\ 1 \leq j \leq n\} \cup \{c_{ij}^{lm} \mid 1 \leq i < j \leq k,\ 1 \leq l, m \leq n, l \neq m,\ (v_l, v_m) \notin E\}$,
- $I = \{w_i = s \mid 1 \leq i \leq k\} \cup \{e_{ij} = 0 \mid 1 \leq i < j \leq k\}$
- $G = \{e_{ij} = 1 \mid 1 \leq i < j \leq k\}$

To prove that this is indeed a parameterised reduction, we first prove that $\Pi$ has a solution if and only if $G$ has an independent set of size $k$. Assume $G$ contains an independent set $\{v_1, \ldots, v_k\}$. Then, the following is a valid plan for $\Pi$ for some function $f : [1, k] \to [1, n]$ that tells how the $b_i^j$ actions select the $k$ independent vertices from $V$:

$$b_1^{f(1)}, \ldots, b_k^{f(k)}, c_{11}^{f(1)f(1)}, c_{12}^{f(1)f(2)}, \ldots, c_{(k-1)k}^{f(k-1)f(k)}.$$

Assume instead that $\Pi$ has a valid plan $\omega$. Since all variables $e_{ij}$ are set to 1 after the execution of $\omega$, it follows that all variables $w_1, \ldots, w_k$ are given distinct values. Additionally, these values (if interpreted as vertices in the original graph) form an independent set. Now note that $|W| = k + k(k-1) = k^2$ so $\Pi$ has a solution if and only if $\Pi$ is variable $k^2$-consistent. Thus, it satisfies condition (1) and (3). To see that it satisfies condition (2), it is sufficient to note that the reduction can be performed in $O(\text{poly}(k) \cdot \text{poly}(|V| + |E|))$ time: we introduce $k + k(k-1)$ variables where each variables has a domain of size $\leq n + 1$ and we introduce less than $k \cdot n + k^2 \cdot n^2$ actions. □

## 5 Discussion

At this point, the message of this paper should be perfectly clear: traditional planners have severe difficulties in identifying unsolvable instances but, fortunately, there are methods (such as consistency checking) with the potential of significantly improving their behaviour. We see several directions for continuing this work which we describe in the next two sections.

### 5.1 Research Directions

(1) There is a need for more test examples and we view the construction of such examples as an urgent task. Constructing a portfolio of interesting unsolvable planning instances is highly non-trivial and calls for a cooperation between different parts of the planning community: obtaining unsolvable benchmarks matching the breadth and diversity of the IPC benchmarks will be a major undertaking. Interesting unsolvable instances appear in industrial applications, and we have already discussed some of these in the introduction. However, one should keep in mind that industrial examples typically come as large individual instances and this makes them hard for studying scalability. Thus, one may also consider theoretical examples based on, for instance, unsatisfiable SAT instances Chvátal and Szemerédi (1988) and Friedgut (1998).

(2) It is vital to study the behaviour of additional planners on unsolvable instances and pinpoint their strengths and weaknesses. We have considered a fair number of different planners and heuristics but there are others that may provide us with additional insights. Immediate examples that come to mind are planners that compile instances into other problems such as SAT, CEGAR-based methods (Seipp and Helmert 2013), and heuristics such as *implicit abstraction heuristics* (Katz and Domshlak 2010) that have interesting connections with PDBs. An exciting way of performing parts of this research would be to launch an IPC track devoted to both solvable and unsolvable instances.

(3) It is obvious that consistency checking need to be generalized in order to cope with certain real-world problems; some concrete ideas are collected in Section 5.2. When it comes to radically new methods for detecting unsolvable instances, we have the following open-ended idea. Bäckström and Jonsson (2012) have shown that there is not much to gain (in terms of computational complexity) by reformulating the plan existence problem as some other problem and then utilizing algorithms for the second problem (unless the polynomial-time hierarchy collapses). However, for the purpose of detecting unsolvable instances, one may use 'one-sided reformulations' where solvable planning instances are mapped (by a function $\rho$) to 'yes'-instances of some problem $X$ but we allow unsolvable instances to be mapped into both 'no'- and 'yes'-instances. Given a (preferably highly efficient) algorithm $A$ for detecting 'no'-instances of $X$, we can now convert a planning instance $I$ into $\rho(I)$ and then apply algorithm $A$. If $A$ replies that $\rho(I)$ is a 'no'-instance, then we know for sure that $I$ has no solution. Thus, we circumvent the barrier implied by their result and the process may be computationally advantageous.

### 5.2 Improving Consistency Checking

We expect that our method can be improved in a number of ways. A very straightforward way is to parallelize the algorithm. Consistency checking is an archetypical example of so called *embarassingly parallel* problems, since all projections can be checked in isolation from each other. A more interesting way is to exploit mutex groups (or other types of implicit information) in more powerful ways. Admittedly, the use of this kind of information in Section 4.1 is fairly primitive and it is highly likely that it can be used for speeding-up consistency checking even further. This is not trivial, though, and we have noted in preliminary experiments that mutex groups can both increase and decrease the time for consistency checking. The time needed may increase due to the overhead induced by mutex group testing; this is typical when the mutex groups are very large but they do not disconnect the state-transition graph during search. In other cases, the time needed is virtually unchanged: the Trucks scenario is a case where we basically do not yield or lose anything since Fast Downward fails to generate any mutex groups. In Trucks, every package has a location variable and every truck has a flag variable, to tell if it is loaded or empty. Unloading a package changes its location and flags the truck as empty. If a projection contains the truck flag but not the location of some package, we get an action that flags the truck as empty without unloading anything, i.e. the truck gets unlimited capacity. Mutex groups can obviously replace such flags by constraining that every package cannot be in the truck at the same time. The Fast Downward planner does not give these mutex groups, however, so it is an interesting question if there are better methods for computing mutex groups, or perhaps even identifying when and how we can replace flag variables with mutex groups. Other ways forward could be to consider other ways of detecting and removing 'anomalous' states and actions in projections, to require certain groups of variables to always appear together or not at all in projections, or to find particular ways to do the encodings to avoid certain problems.

We have further noted that our method is similar to iPDB. One way to view the two methods is as a search in the space of variables subsets. Then our method is a blind complete enumeration while iPDB is some kind of heuristic search in the space of variable subsets. This gives a more uniform view of the two methods which opens up for a better understanding of how to combine them or design new methods with a flavour of both. One might also envision a variant of iPDB that explores many more subsets, for the purpose of checking consistency, without necessarily keeping more projections in the database.

## Acknowledgements

We thank the reviewers for their feedback on this paper. In particular, we would like to thank one of the reviewers for pointing out that heuristics other than iPDB implicitly utilize some kind of consistency checking. Simon Ståhlberg is partially supported by the *National Graduate School in Computer Science* (CUGS), Sweden.

## References


Bäckström, C., and Jonsson, P. 2012. Algorithms and limits for compact plan representations. *Journal of Artifical Intelligence Research* 44:141–177.

Boddy, M. S.; Gohde, J.; Haigh, T.; and Harp, S. A. 2005. Course of action generation for cyber security using classical planning. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-2005)*, 12–21.

Chvátal, V., and Szemerédi, E. 1988. Many hard examples for resolution. *Journal of the ACM* 35(4):759–768.

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

Downey, R. G., and Fellows, M. R. 1999. *Parameterized Complexity*. Monographs in Computer Science. Springer, New York.

Edelkamp, S.; Leue, S.; and Visser, W. 2007. Summary of Dagstuhl seminar 06172 on directed model checking. In *Directed Model Checking*, number 06172 in Dagstuhl Seminar Proceedings. Dagstuhl, Germany.

Flum, J., and Grohe, M. 2006. *Parameterized Complexity Theory*, volume XIV of *Texts in Theoretical Computer Science. An EATCS Series*. Springer, Berlin.

Friedgut, E. 1998. Sharp thresholds of graph properties, and the k-sat problem. *Journal of the American Mathematical Society* 12:1017–1054.

Futoransky, A.; Notarfrancesco, L.; Richarte, G.; and Sarraute, C. 2010. Building computer network attacks. *ArXiv* abs/1006.1916.

Goldman, R. P.; Kuter, U.; and Schneider, T. 2012. Using classical planners for plan verification and counterexample generation. In *Proceedings of AAAI workshop Problem Solving Using Classical Planners, Toronto, ON, Canada*.

Harp, S.; Gohde, J.; Haigh, T.; and Boddy, M. 2005. Automated vulnerability analysis using AI planning. In *Proceedings of AAAI Spring Symposium on AI Technologies for Homeland Security, Stanford, CA, USA*, 52–62.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*, 140–149. AAAI Press.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the 22th AAAI Conference on Artifical Intelligence (AAAI-2007)*, 1007–1012. AAAI Press.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proceedings of the 20th AAAI Conference on Artifical Intelligence (AAAI-2005)*, 1163–1168.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-2004)*, 161–170.

Helmert, M. 2006a. New complexity results for classical planning benchmarks. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-2006)*, 52–62.

Helmert, M. 2006b. The fast downward planning system. *Journal of Artifical Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Katz, M., and Domshlak, C. 2010. Implicit abstraction heuristics. *Journal of Artifical Intelligence Research* 39:51–126.

Muise, C. 2010. The planning domain repository. `https://bitbucket.org/haz/planning-domain-repository`.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artifical Intelligence Research* 39:127–177.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artifical Intelligence* 170(12-13):1031–1080.

Sarraute, C., and Pickering, K. 2012. Encounters of the third kind between pentesting and automated planning. In *Invited talk at AAAI workshop Problem Solving Using Classical Planners, Toronto, ON, Canada*.

Sarraute, C.; Buffet, O.; and Hoffmann, J. 2012. POMDPs make better hackers: Accounting for uncertainty in penetration testing. In *Proceedings of the 26th AAAI Conference on Artifical Intelligence (AAAI-2012)*.

Seipp, J., and Helmert, M. 2013. Counterexample-guided cartesian abstraction refinement. In *Proceedings of the 23th International Conference on Automated Planning and Scheduling (ICAPS-2013)*, To appear.

Smith, D. E. 2004. Choosing objectives in over-subscription planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-2004)*, 393–401.

Zilles, S., and Holte, R. 2010. The computational complexity of avoiding spurious states in state space abstraction. *Artifical Intelligence* 174(14):1072–1092.