KRF

Knowledge Representation Framework Project
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Resume of a Course on Artificial Intelligence and Lisp

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).

The present report, PM-krf-020, can persistently be accessed as follows:

Project Memo URL:	http://www.ida.liu.se/ext/caisor/pm-archive/krf/020/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2010/019/
Date of manuscript:	2010-12-07

Related information can also be obtained through the following www sites:

KRFwebsite:	http://www.ida.liu.se/ext/krf/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/∼erisa/

This lecture note contains material that was presented towards the end of the course on Artificial Intelligence and Lisp (TDDC65) at Linköping university in the autumn semester of 2010, and that is not otherwise covered in the lecture notes for the course. This includes:

- An overview of topics in the field of artificial intelligence, in order to provide a fuller view of what the field contains, including topics that have not been covered in the course. This was presented in Lecture 13.
- A description of the concluding lab exercise, called lab 5b.
- A brief introduction of the topic of satisfiability solver methods, which was also presented in Lecture 13.

Please take notice also of the following materials that are found in the course webpage, lecture materials for Lecture 13, slides number 2 to 11 in the e-slide sequence (= 'powerpoints') for "Knowledge Acquisition and Ontologies." These slides illustrate some general issues in contemporary knowledge acquisition, in particular, the requirement to handle very large volumes of simple facts as well as more complex information, and the problem of insufficient formal quality in available knowledge sources. We have not converted these slides to lecture note format since they are purely illustrative, and more easily read in their present form.

Please notice as well the lecture-note materials about ontologies. This occurs as the final chapter in Compendium III for this course, i.e. the one with the subtitle "Intelligent Autonomous Agents."

Chapter 1

Overview of Topics in Artificial Intelligence

Different people may have more or less different opinions about what are the subareas of Artificial Intelligence, and about which of these subareas are the most important ones or the most basic ones. The following represents my point of view, which I believe can safely be said to represent a strong majority opinion.

1.1 A list of sub-areas

As a starting point for the discussion I shall take the list of research areas in the call for papers of the 2011 AAAI conference (Association for the Advancement of Artificial Intelligence). It goes as follows.

```
A  Agent-based and multiagent systems
C  Cognitive modeling and human interaction
A  Commonsense reasoning
C  Computer vision
B  Constraint satisfaction, search, and optimization
C  Evolutionary computation
C  Game playing and interactive entertainment
C  Information retrieval, integration, and extraction
B  Knowledge acquisition and ontologies
A  Knowledge representation and reasoning
C  Machine learning and data mining
A  Model-based systems
C  Natural Language Processing
A  Planning and Scheduling
B  Probabilistic Reasoning
C  Robotics
C  Web and information systems
```

The list of topics for the International Joint Conference on Artificial Intelligence (IJCAI) is similar. - In this list I have marked the various areas with the letters A, B, and C, with the following meanings. The A and B groups include work with a similar methodology, based on core computer science

(algorithms, etc), the use of formal logic, and applying these tools to the qualitative modelling of phenomena in the real world and reasoning based on the information in such models, as well as to the design of autonomous software agents that can make use of those models and that can perform the said type of reasoning.

The difference between the A and B groups is that the A-listed topics tend to be more basic, so that one should study them first before proceeding to the B-listed topic. [1] This borderline is not strict but it can serve as a guide.

The C-listed topics are by and large interdisciplinary, in the sense that each of them has strong connections to some other scientific discipline besides artificial intelligence, and in most cases outside computer science. The field of robotics contains a lot of work in mechanical engineering and in control engineering, for example; the field of natural language processing as seen from A.I. integrates seamlessly with work in computational linguistics and with the cognitive sciences in general, and so forth.

The cases of evolutionary computation and machine learning should also be mentioned separately. The actual literature that goes by the name of machine learning has a large component that makes heavy use of probability theory. This work is only remotely connected to the work in the A and B categories above. There are also some niche areas, such as case-based learning and inductive logic programming, that is well connected to the A and B categories, but these areas seem to be marginal in the contemporary research literature on machine learning.

With respect to evolutionary computation, there is a range of work that is represented at one end by the approach of 'artificial life', where one tries to develop computational life-forms using a simulated evolution process, and at the other end by work on generate-and-test techniques (i.e. systematic search techniques) that are applied to moderately complex objects that have a dynamic interpretation, for example simple scripts in a specialized language, or decision trees. In the former case the methodology is remote from what we have in the A and B listed areas; in the latter case there is a quite close connection.

This discussion implies that there is coherent point of view about artificial intelligence where, with simplification, the A listed areas are considered as the basic ones, the B listed areas are also included but are best studied in a second step, and where the C listed areas integrate work that is close to the A and B areas, with work that has an entirely different character.

Specifically, the C listed areas can be characterized in terms of three major groups, as follows

[1] One may ask why 'search' and 'optimization' are not listed in the most basic category, since these are mathematical techniques that are widely used in A.I. The answer is that from an A.I. point of view it is natural to first study the A listed topics in order to understand what the field is about, and then to proceed to the relatively more technical details of the algorithms being used. From a mathematical mindset one may prefer the reverse ordering.

Bridge Areas

Most of the topics in the C category have the character of a bridge that combines Artificial Intelligence with one or more other disciplines. They form three natural groups, as follows.

Bridge to Robotics

This includes the following ones of the topics listed above:

```
A  Agent-based and multiagent systems
C  Computer vision
A  Knowledge representation and reasoning
C  Robotics
```

Bridge to Human-Machine Interaction

This includes:

```
A  Agent-based and multiagent systems
C  Cognitive modeling and human interaction
A  Commonsense reasoning
C  Game playing and interactive entertainment
A  Knowledge representation and reasoning
C  Natural Language Processing
```

Bridge to Large Knowledgebases and their Use

This includes:

```
C  Information retrieval, integration, and extraction
B  Knowledge acquisition and ontologies
A  Knowledge representation and reasoning
C  Web and information systems
```

1.2 Alternative Views of Artificial Intelligence

Our view of what Artificial Intelligence is about was described in the memo "The Goals of Artificial Intelligence Research. A Brief Introduction" which is part of the course literature. This memo begins with a discussion of concrete examples of the use of intelligence in small children and in adults, and takes it from there to an identification of what are the major problems and issues in A.I. This results in a view of A.I. where *agent intelligence* is emphasized, that is, one is concerned with the design of software systems that are able to *act intelligently* in some naturally occurring environment. This is the classical view of Artificial Intelligence, and it was expressed, for example, in the proposal for the design of an 'advice taker' by one of the field's pioneers, John McCarthy at Stanford University.

1.2.1 Systems with Specialized Intelligence

This classical view may be contrasted with approaches that work towards the design of quite *specialized varieties of intelligence.* One example of this is the work on programs that play specific games, for example chess-playing programs. These are sometimes seen as examples of artificial intelligence,

in the sense that intelligence is considered as essential for being able to play chess well by humans. Today there exist advanced programs that play chess extremely well and that are able to match even the world's best human chess players. These programs are quite advanced and shall by no means be thought of as mere brute-force search programs. However, there are few connections between the techniques that are used in these programs and techniques that are of use A.I. in general: chess-playing program techniques are not used for A.I. in general, and vice versa, except for a few basic principles such as the importance of combinatorial search.

Another example of specialized intelligence is for automatic translation of natural language. It was previously thought that general-purpose artificial intelligence would be required for good-quality language translation, since in order for a person to translate a text he or she must understand its contents. However, at the present stage of development it turns out that better translation results are obtained using specialized, 'stupid' translation programs that use statistical methods and large databases of known translation patterns, and knowledge-based language translation systems are not able to compete with them at present. This is another example of a specialized task that seemingly requires general-purpose intelligence in humans, and where a specialized approach has been more successful so far in computers.

Given these observations one may wonder whether the most promising long-term strategy for intelligence in computers will be to build a number of specialized 'intelligences', like the ones just described, and to integrate them gradually. Some researchers have argued this position, sometimes under the name of the 'big switch' theory of intelligence. Time will tell, but it is fair to assume that the 'agent intelligence' approach that has been used for the present course will *at least* be one of the participating 'intelligences'. Whether it will just be one out of several, or whether it will also be the cohesive force that makes it possible to integrate the others, or whether it will eventually be the universal principle that includes the others, this remains to be seen.

1.2.2 Alternative Computational Infrastructures

The topics in the A and B categories of the topic list above are based on conventional computational technology: algorithms, datastructures, programming languages, software architectures, and the use of formal logic. This technology has been developed for, and is adapted to the use of the von Neumann computer design with a small number of processors that work with almost entirely passive memories.

This computational infrastructure is of course very different from the structure of the human brain with its extremely large number of concurrently active brain cells. Some branches of research propose that this is a serious handicap, and that a better strategy for the development of machine intelligence is to build computational systems that are more similar to the neural or neural-based system in people and in animals. This is the approach of *neural computation.*

A somewhat related view questions the precedence of the knowledge-based techniques in the A category above, and propose that it is better to first implement the counterparts of more elementary human behavior, including

sensorimotoric behavior and other stimulus-response behavior. The proposal is that intelligence of the kinds discussed above – agent intelligence, chess-playing and language-translation capabilities, and so on – are based on these simpler behaviors and that they *emerge* from them, i.e. they are evolved from them. These simpler behaviors can then conceivably be implemented either using neural networks or other alternative hardware architectures, or using conventional programming-language technology.

Approaches using alternative computational infrastructures have produced good results concerning the implementation of lower-level behavior, below the level that is normally called 'intelligence'. However, their usefulness for the construction of agent-level intelligence or specialized intelligence of the kinds discussed above has not been demonstrated and there is no concrete indication that it will ever be a viable approach.

1.3 Representation of subareas in the present course

The contents of the present course can now be related to the structure of the field that was shown above. The following topics, among those listed above, are the ones that have been represented to some reasonable extent in the present course.

```
A  Agent-based and multiagent systems
A  Commonsense reasoning
B  Constraint satisfaction, search, and optimization
B  Knowledge acquisition and ontologies
A  Knowledge representation and reasoning
A  Planning and Scheduling
```

A few additional areas have been touched upon very marginally, in particular:

```
C  Cognitive modeling and human interaction
A  Model-based systems
C  Web and information systems
```

This means in summary that the present course has concentrated the topics that are of basic importance, in particular for the design of agent intelligence, but also arguably for artificial intelligence in general.

Chapter 2

A Simple Example of a BDI Agent

The previous chapter described the structure of the field of Artificial Intelligence in terms of its subareas, and emphasized the central character of those aspects of the field that are relevant for agent intelligence. It also confirmed that the selection of contents for the present course has placed the most weight on exactly those aspects.

One of the first topics that was addressed during the course was the topic of the overall software architectures for an intelligent agent, where in particular we described the Belief-Desire-Intention (BDI) architecture and Hierarchical Task Networks (HTN). At this point towards the end of the course there is therefore a need to indicate how the various contents of the course can fit into such an architecture. This is the topic of the present chapter.

We shall use the BDI architecture for this exercise since it is the more general one and since it can be seen as a top-level framework within which it is possible to accomodate a variety of A.I. methods, including an HTN subsystem for current plans as well as specific methods for planning, scheduling, decision-making, and so forth.

Lab materials have been developed for a lab 5b in this course, where the idea is that the student will make test runs with an autonomous agent whose behavior is defined using a simple version of the BDI system definitions. This should be a concrete example of how different methods fit together. The lab does not involve defining additional behaviors for this agent, but merely making some simple changes in the microworld where it operates, and then letting the agent run and see what happens.

The present chapter contains a comprehensive description of this lab, including a description of the microworld where the agent operates, as well as a description of typical scenarios in this microworld.

With respect to the course requirements, we notice that already the earlier labs have required substantial work by the students, and therefore we have decided to let the actual running of the lab be optional for those students who wish to try it, but not making it obligatory. The careful reading of the present chapter should instead be sufficient (and probably more efficient) for understanding the whole.

2.1 World Model for the Zoo Microworld

Lab 5b operates in a microworld consisting of a simple zoo that is 'populated' with animals, a warden and other personnel, a number of vehicles, and physical structures that provide locations for, and resources for the people and the animals in the zoo. The warden is the sole agent in the zoo, so there are definitions for a number of actions that the warden can perform. The warden is goal-oriented in the sense that he has some standing goals that ought to be always satisfied, and also in the sense that events and conditions in the microworld may lead the agent to adopt specific goals. These standing goals are what is called 'desires' in the BDI model, whereas the specific goals and the instantiations of the desires are simply called 'goals'. Each goal in this sense induces the agent to choose or to make a plan for achieving that goal, and then to try to execute it.

This Zoo Microworld has been implemented in the Leonardo system and based on the materials from labs 2b, 2c and 5a. The materials for decision trees and causal nets in labs 3a and 3b have not been included in the actual lab materials, but we know already how lab 3b is based on lab 2 materials.

The following structure is assumed for the Zoo Microworld.

2.1.1 Types and Classes

The Zoo Microworld contains entities of the following *types:*

```
animal
personnel
building
route
sprinkler
food
medicine
tool
vehicle
bag
```

The type `route` is used for roads and footpaths within the premises of the Zoo. The type `tool` is used for scissors, spades, scales and the like. The type `building` is used for stables, cages etc where the animals are kept, as well as for the administration building, the restaurant and other buildings for the personnel. (Visitors are not included in the Zoo model). The types `sprinkler` and `bag` are used for specific purposes in some of the scenarios.

In addition the following types of entities are used for classification or characterization of entities of the above types:

```
species
occupation
ailment
```

Each animal belongs to a particular species; each personnel has a particular occupation; ailments include diseases, wounds and other problems that may afflict animals and personnel.

Besides types, the model of the Zoo Microworld also contains a number of *classes.* Each class pertains to one specific type, and has a number of instances of that type as its members. For example, if `chimpanzee` is an entity of type `species,` and `Rollo` and `Lollo` are specific animals whose `type` is `animal` and whose `in-species` attribute is `chimpanzee,` then there may be a class `our-chimpanzees` that *pertains to* the type `animal` and that has `Rollo` and `Lollo` as its *members.* If `Rollo` and `Lollo` are the only chimpanzees in the microworld then this class can also be written using the following expression in Description Logic [[1]]

```
(those animal that in-species all {chimpanzee})
```

2.1.2 Static Structure

The Zoo Microworld has a *static structure* that stays the same as time passes in this simulated world, and a *dynamic structure* that changes due to spontaneous changes in the world and due to actions that are performed by the warden. The static structure consists in turn of a *conceptual structure* and a *physical structure.* The conceptual structure consists of analytical statements such as "giraffes are hoofed animals" whereas the physical structure consists of empirical statements such as "The chimpanzee court is located along route-4."

Some statements about class membership are clearly empirical, for example, "Rollo is in the class of animals having brown fur." For some other kinds of class membership statements it is debatable whether they are conceptual or empirical, for example "Rollo is a chimpanzee" or "Rollo is a male," but for simplicity we include all statements about classes and their members among the empirical ones.

2.1.3 Dynamic Structure

The dynamic structure is represented using *features* that are assigned a value at each point in time using the `H` predicate as usual. The most common way of constructing features is using the `the` function, like in

```
(the fur-color of Rollo)
```

for example. The basic lab does not contain any other features and feature-formers, but it is designed in such a way that more can be added if needed for some extension of the basic lab.

2.1.4 Actions

Actions are written using the usual CEL notation. Most actions have a parameter with the tag `by` that indicates the *subject* of the action, for exammple

```
[eat :obj banana-4 :by Rollo]
```

[1] This uses the CEL variant of the Description-Logic notation, with the additional proviso that the first argument of the `those` operation shall be a type and not a class.

or

```
[pass-overhead :by airplane-4]
```

There are a few examples of actions that do not have a `by` parameter ("natural actions"), such as

```
[raining]
```

Actions that are performed by the warden of the Zoo (or the primary warden if there are several of them) have the entity `TheWarden` as the value of the `by` parameter. In this case the `by` parameter may be omitted.

It may be argued that natural actions and actions having an inanimate subject (for example "the stone falls") ought to be called events rather than actions. However we stay with the term "action" since the term "events" has another meaning in the BDI terminology, and since that meaning is also widely used.

2.2 Motivational Structure

The warden is the primary actor in the Zoo Microworld and is represented by the entity `TheWarden` in the implementation. The implementation represents the warden in the sense that at successive timepoints it decides what actions the warden is going to perform in the world. The implementation also simulates the execution of these actions. A more developed system might use two separate computational processes, one for simulating the world and one for simulating the warden's cognitive state and decisions, but in this case we combine these functions into one single computational session. This also means that there is no difference between "knowledge" and "belief" in this implementation: the conditions that hold in the simulated world are also known to the warden agent.

2.2.1 Desires, Goals and Intentions

The *motivational subsystem* is the system that determines what actions the agent is going to perform. It is organized as a simplified BDI model; simplified in the sense that there is no distinction between facts and beliefs.

Desires are expressed as logic formulas typically using the predicate `Hc` which means that they refer to the current timepoint in the execution of the system, or to a specific, anticipated future timepoint. The following is an example of a desire formula.

```
[Hc (the hunger of TheWarden) none]
```

This formula assumes that the entity for the warden has an attribute `hunger` indicating his level of hungryness on a scale from 0 and up, and it expresses that the warden desires not to be hungry. Here is an example of a more general desire formula.

```
(all .p (imp [equal (get .p type) personnel]
             [Hc (the hunger of .p) none] ))
```

saying that for every entity `.p` of type `personnel` it is desired that this entity shall not be hungry.

The purpose of the motivational subsystem is to try to arrange that the desires are true as much as possible. It can do so by choosing actions that the warden performs. These actions will have immediate effects in the world, but they can also affect what actions are done by some of the animals.

As always in such systems there is an issue concerning whether the agent (the warden, in our case) shall reconsider all desires at each point in time, or whether it shall stick to a plan that it has chosen, and only verify the desires sometimes. Different systems may address this question in different ways. The Zoo Microworld system does it as follows. Desires are specified on two levels, for *high* and *medium urgency.* When the agent recognizes that a desire is violated then this constitutes a goal, and the agent may choose a *method* for achieving the goal. A method is typically defined as a sequence of actions that are to be performed by the agent, but it may also have another character, for example, adding or modifying a method. Action sequences that are used as methods for medium-urgency desires are called *scripts.*

Each time an elementary action has been completed during the execution of a script, the agent checks whether all the high-urgency desires are satisfied. If one is not, then the execution of the script is interrupted, a method for the violated high-urgency desire is applied, and then the agent returns to the interrupted script.

High-urgency desires also have methods, but such methods are usually not defined as action sequences. In some cases they may be like rapid stimulus-response behaviors in people; in other cases they may have a cognitive definition, such as making a change in the agent's current plan.

Medium-urgency desires are not checked between successive steps in a script, but only when a script has been completed. At such a point all the desire violations are identified and a feasible subset of them is identified. The desires in that subset are called *goals.* In this case there are three things that distinguish the set of goals from the set of desires: (1) goals only contain those desires that are not presently satisfied; a desire that is already satisfied does not go into the goals; (2) goals are usually instantiated desires, i.e. some of the variables in the desire have been replaced by constants; (3) some candidate goals may be omitted from the goal set if this is necessary in order to be able to make a plan.

For example, if one of the desires is "all the animals shall be healthy" and at a particular time two of the animals are not well, namely `Lollo` and `Granny` then this will contribute two goals to the goal set: *Lollo shall be healthy* and *Granny shall be healthy.* At another time if all the animals are well, then the goal set will not contain any goal with respect to animal health.

The term "high urgency" shall not be taken as meaning that high urgency desires are very important; it only means that they may be attended to during the execution of a script towards some other goal. For example, if the simulated warden walks from one place to another in the Zoo and he notices a piece of litter along the path, then he may stop by to pick it up and put it in the nearest waste basket. Doing so is presumably not more important than other duties, but the fact that it can be done in the middle of another activity qualifies it as "high urgency."

The term "low urgency" is reserved for desires that are only attended to if the agent does not have anything else to do, but such desires are not used

in the Zoo Microworld.

The selected goal set will correspond to intentions and plans for achieving the goals in that set. If there is only one goal in the set then the current intentions consists of the current plan and the current goal, i.e. what the agent intends to do and what it intends to achieve with that. If there are several goals then there may be several separate plans, for example one for each goal, and the term "intentions" refers to the aggregate of these plans, the relations between them, and the goals that they are supposed to achieve.

The distinction between plan and intentions is more salient if more than one agent is involved, or if causality is involved, so that the intentions may include those results that are indirect effects of performing the plan.

2.2.2 Method Checking

Each time a medium-urgency goal is identified, the system will look for methods that are associated with that goal, using a *method generator*. The answers from the method generator are however not definite; they shall be understood as suggestions that work often, but not always. Therefore, before one of these plans is adopted by the system, it is an advantage if the method kan first be *checked* using a process that predicts the effects of using the method (usually, executing a script) in the situation at hand. Checking by prediction is not a necessary facility, since it is certainly possible to have agents that go ahead and use proposed methods without first checking them by prediction, but doing prediction can add robustness to the system and help avoiding the use of methods that turn out not to have the intended results in a particular situation.

One way of doing such a prediction is using a simulator, but in the current system it is done by logical deduction. Each verb is associated with effect rules that specify the effects on the state of the world when the action is performed.

If the method proposes several methods then the system must choose between them. Without prediction, it may use a *priority level* that is associated with the desire, or with the method. With prediction, it may predict the outcomes using each of the methods, evaluate the merits of each of the outcomes, and make its choice accordingly.

If the method generator is not able to propose any method at all then the system simply decides to remove that goal from the goal set – it becomes an example of a desire that can not be satisfied. (Some systems, such as those using the SOAR architecture, will then consider the task of finding an adequate plan as a meta-level problem and address it using the same techniques as for object-level problem solving). Similarly, if the action generator produces one or more methods but none of them passes the check by prediction, then again the present system gives up the goal. Some other systems go into a mode where they try to *modify* some of the proposed methods so that they will achieve the selected goal.

2.2.3 Reportoire of actions and verbs

We distinguish between actions that are performed by Zoo personnel (and therefore by people) and actions that are performed by animals. The following is a catalog of verbs that may be appropriate in the Zoo Microworld. Each scenario in the world uses its own subset of these verbs.

Verbs for actions by personnel without dealing with animals: The following are operations done to the physical premises of animals

```
go to a location
enter/leave a building
pick up/ put down an object, e.g. a bag
take out/put back an object in a bag
open up/fold down an umbrella
```

Verbs for actions by personnel done to the physical premises of animals:

```
open/close door (e.g. door of a cage)
clean up
wash floor
turn on/ turn off light
```

Verbs for actions by personnel, with an animal as the object:

```
pick up/put down an animal
cause animal to do (some of the above)
assist animal in doing (some of the above)
move
wash
delice              (= remove lice from)
wrap in sheets      (for warmth)
separate            (two or more animals, in case of fight)
take body temperature
clean wound
cut nails/claws
inoculate           (= vaccinate)
chain
give name
take picture of
kill
```

Verbs for actions by animals:

```
eat
drink
bark/ cackle/ ...
sleep
bathe
swim
point at
give birth
breastfeed
begin or end doing some of the above (including wake up, fall asleep)
```

2.3 The Rainy Day Scenario

The structure shown above is a general framework within which a number of scenarios can be defined. Each scenario uses its own subset of the general framework and adds some minor components of its own. For the purpose of lab 5b we use the *Rainy Day Scenario* which is defined as follows. It involves a single personnel, called `TheWarden` that can move around the physical premises of the Zoo. As he does so he may get wet, either because it starts raining, or because he passes a point where a lawn sprinkler is operating. His desire is not to get wet, or at least to minimize his level of wetness - quickly passing by a sprinkler may not be too bad, in particular if it is a sunny day so he will get dry quickly. A few other phenomena are also defined in this scenario in order to add some complexity.

2.3.1 Physical Structure

In the Rainy Day Scenario the Zoo Microworld contains a rectangular road network. There are roads in the east-west direction called *paths* and in the north-south direction called *trails.* Each path and trail has an integer number between 1 and 6. Each intersection between path and trail is a *crossing* and is described in the obvious cartesian way as (trailnr, pathnr), so that the first component represents the x axis.

Each crossing is a roadpoint. In addition there are roadpoints halfway between the crossings but on the paths and trails, for example (3+, 2) and (3, 2+), but not (3+,2+) since that would not be on a road. These are all the roadpoints there are. The following diagram of roadpoints illustrates this structure:

```
    |             |
-- 3,4 -- 3+,4 -- 4,4 -- 4+,4 --  etc
    |             |
-- 3,3+          4,3+
    |             |
-- 3,3 -- 3+,3 -- 4,3 -- 4+,3 -- etc
    |             |
```

It follows that when the warden is at a crossing he has four possible moves, namely north, south, east and west, and each of these will take him to a roadpoint that is not a crossing. Non-crossing roadpoints only allow two possible moves, in the obvious way.

The implementation of this scenario contains a simple route planner that will construct a sequence of moves that will take the warden from his present position to a given new position. The route plan is a sequence where the elements are north, south, east, or west.

2.3.2 Dynamical Structure

The warden may move from one roadpoint to an adjacent one. While doing so he may or may not carry a particular bag, called `TheBag.` There are actions whereby he puts the bag on the ground beside him, or picks it up.

If he has picked up the bag then it comes with him as he moves to a new roadpoint, but if it is on the ground then it stays there.

The dramatical component of this microworld is obtained by situations where the warden can get wet, namely, if it is raining, or if he passes by a roadpoint where a grass sprinkler is running. Raining is a feature of the Nature object, so it may start and stop raining at specific timepoints. The sprinklers do not change, so at some roadpoints there is a sprinkler and it runs all the time.

One more artifact is included in the scenario, namely an umbrella that may be either in the bag, or held in the warden's hand. If the warden gets in the way of a sprinkler or of rain then he will get wet, except if he has taken out his umbrella. The warden does not like to get wet, so he may choose to take out his umbrella when rain occurs or he gets in the way of sprinkling.

However, to complicate matters, in order to take out the umbrella from the bag, he must first put down the bag on the ground, then take out the umbrella, and then pick up the bag again if he wishes to continue carrying it. While performing these operations he will get increasingly wet. Therefore, if he is already in the way of a sprinkler, it is best to continue walking through it and not stop to take out the umbrella. If it is raining then the reverse is true, unless the agent is already soaked.

Another way for the warden to avoid getting wet is to go inside a building. However, then he must be at, or go to a roadpoint where there is a building. Well inside a building he may wait a while so as to become dry again.

Being wet is not an absolute concept, therefore; the warden's wetness is defined on a scale from 0 to 6, where 0 means entirely dry. Each timestep where the warden is exposed to rain or sprinkling his wetness increases by 2; every timestep where it is not raining or he is inside a building the wetness decreases by 1, all within the interval from 0 to 6. For these purposes, a timestep is when the warden moves from one roadpoint to the next, or when he puts down the bag, picks up the bag, gets out the umbrella, or puts it back. Moreover there is a 'wait' action where nothing happens except that the warden's wetness changes according to the rules.

For the purpose of this scenario only one desire is considered, namely the desire to be dry, or at least not too wet. In the formal sense, the warden distinguishes between 'dry' (level 0), 'soaked' (level 6) and 'wet' (intermediate levels), and he has two desires: not to be soaked (highest priority) and not to be wet (lower priority, but still a desire). The reason for defining the desires in this way is that he will anyway become somewhat wet when he reacts, for example by putting down the bag in order to take out the umbrella, or when he walks to a nearby building.

2.3.3 Implementing the Motivational Subsystem

In this section we shall describe the formal definitions that are needed in order to implement the motivational structure described above. The purpose of this section is not to be like a software documentation, but merely to give the reader an idea of the size, complexity and character of the definitions that are needed.

The following are the definitions of the warden's desires and their associated methods:

```
------------------------------------------------------------
-- desire-1

[: type bdi-desire]
[: has-urgency medium]
[: has-methods <open-umbrella dry-in-building>]
[: latest-rearchived nil]

@Desidef
[-Hc (the howmuch-wet of TheWarden) 6]

------------------------------------------------------------
-- desire-2

[: type bdi-desire]
[: has-urgency medium]
[: has-methods <open-umbrella dry-in-building>]
[: latest-rearchived nil]

@Desidef
[Hc (the howmuch-wet of TheWarden) 0]

------------------------------------------------------------
-- open-umbrella

[: type goalmethod]
[: has-priority 6]
[: latest-rearchived nil]

@Requires
(and [Hc (the open-umbrella of TheWarden) no]
     [Hc (the carries-bag of TheWarden) TheBag] )

@Method
[soact [putdown :b TheBag][takeout][pickup :b TheBag] ]

@Comment
This method for avoiding wetness is for the warden to put down the bag,
take out the umbrella, and pick up the bag again. It requires that the
warden carries the bag and that he has not already taken out the
umbrella.

------------------------------------------------------
-- dry-in-building

[: type goalmethod]
[: has-priority 6]
[: latest-rearchived nil]

@Requires
(and [Hc (the is-inside of TheWarden) no]
```

```
    [-attrib-is (get TheWarden location) has-building nil] )

@Method
[soact [go-in][wait-until-dry][go-out]]

@Comment
This method is for the warden to go into a building and wait there
until he gets dry. It requires for him to be beside a building.
```

The following attributes have been used here:

`howmuch-wet`	`number indicating the wetness of the warden`
`open-umbrella`	`indicates whether the warden has opened up his umbrella`
`carries-bag`	`indicates whether the warden carries the designated bag`
`is-inside`	`indicates whether the warden is inside a building`
`location`	`indicates the location of the warden as a composite entity, e.g. (crossing: 2 3)`

In these definitions there are two desires, called `desire-1` and `desire-2` which represent the two levels of desire for the warden not to get wet. Each of these is specified by a logical expression for the desire, in the `Desidef` property, and by a sequence of methods that may be applicable. Each method is specified by a `Requires` formula that tells when the method is applicable, and a `Method` formula containing a script for what actions to perform in order to use the method.

Notice that the two properties of the methods are just the same as are used for precondition-repair methods which were used in lab 2b and 2c. These definitions are found in the entityfile `motivdefs` in the downloadable lab materials for lab 5b.

This is basically what is needed in order to define such desires and methods for achieving the instantiated goals. The following other things are also needed:

- Definitions of the verbs that are used for performing actions and achieving goals. These are in the entityfile `lab5b-verbs.`
- Definitions for the machinery for instantiating desires and administrating the resulting goals. This is in the entityfile `motivsys` and requires around 250 lines of code in the present lab materials, which means it can be done quite compactly.
- Definitions for how to do prediction of the results of proposed scripts. This uses the techniques that were shown in lab 5a of this course, and very little additional definitions are needed.

The following are the important points that can be seen in this simple example and that we wish to illustrate with it:

- The simple desire-driven and goal-directed behavior can be implemented using a very moderate amount of code, combined with a straightforward use of standard first-order logic and of the formalism for reasoning about actions.
- The behavior that results from the definitions in the example is quite simple, but considerably more sophisticated behavior can be imple-

mented by adding more 'knowledge' in the form of logic formulas, and without the need for a lot of additional programming.

2.3.4 Episodes in the Rainy Day Scenario

The implementation of the Rainy Day Scenario contains definitions for elementary move actions in the Zoo's road network. It also contains a composite `goto` verb that makes successive single moves from one roadpoint to the next until it reaches the specified definition. This verb is defined in such a way that at each step during the promenade it will check the medium-urgency desires and see if they are satisfied; if not it introduces a goal to achieve the desire, and selects a method for achieving the goal.

The test and demo of the BDI machinery is therefore best done in the context of 'promenades', that is, in the context of `goto` actions where the warden passes by successive roadpoints, and where there will be occasions for getting wet due to occasional rain or passing by sprinklers. Every such promenade can constitute an episode just like we have used in earlier labs.

Consider now at first the case of doing this without prediction, which means that the agent selects the most promising one, according to a numerical priority, among those methods that are found to be applicable. The agent executes that method. However, since one may not be sure that the method has the intended effect, the agent must re-evaluate the desire after the method has been used in order to check. If the goal was achieved then all is fine, but otherwise the agent will add this goal to its set of unachieved goals, and proceed.

Several design decisions are possible at this point: one may choose to let the agent re-execute the script one more time, or several more times, in the hope that this will help. However, for actions such as 'take out the umbrella' this obviously will not help, and in fact this method is not even defined as applicable if the umbrella is already out.

Another possibility is to try another method, if the knowledgebase contains several methods that are appropriate for the situation at hand.

The classical BDI approach would say that in this case the agent should just drop the goal as being unachievable. In our implementation we arrange instead to let the agent keep the unachieved goal on a separate list, so that they are retried at later points. For example, the method of going inside a building in order to dry out will be applicable if the agent passes by a building during its continued walk.

However, this example also shows the importance of look-ahead. Given the world model of the present scenario, suppose the simulated warden is making his promenade and one step ahead there is a building where he can get inside, and at that point it starts to rain in the simulated world. One alternative is to take out the umbrella, but he will get more wet while doing that; a better approach will be to just continue walking and to slip into the building in the next step, and then either wait for the rain to stop, or take out the umbrella while being inside the building. The systematic way of getting that behavior is to arrange that when the agent notices a violation of a desire, it shall check the currently available methods and foresee the effects of using each of them, but it shall also check the foreseen effects of

continuing its base activity one or a few steps and maybe take remedial action a little bit later.

In fact, an even more effective method is to arrange the agent so that it does not merely check its present state for desire violation, but also, in each step of its behavior, it anticipates a few steps ahead what is going to happen, checks whether there is a desire violation in the predicted future, and reacts to it. In this case, if the agent sees a sprinkler up its path, or sees the indications that it is starting to rain, it will be able to perform the remedial action of taking out the umbrella even before it starts to get wet. In lab 5a we showed how the reasoning system could predict effects in alternative, immediate futures, and this is the technique that shall be used here.

Chapter 3

Computational Engines

The computational methods of Artificial Intelligence include a number of 'engines' which are essentially configurable algorithms. This means that they perform a certain computation in a systematic way, but the way that the computation is performed is subject to adaptations and adjustments that are specific to the application at hand. The resolution method for logical deduction, which has been presented earlier in the course, is one example of such a computational engine. The present chapter contains an introductory description of SAT solvers, which are also such engines, and a very brief mention of a third engine. There are others, of course, but we restrict the course contents to these.

3.1 Satisfiability Solvers

The Boolean Satisfiability Problem (SAT) is defined as follows in its basic form.

Given: a set of propositional clauses

Question: does there exist an assignment of truthvalues to the proposition symbols whereby all the clauses are true?

In other words, does there exist an assignment whereby at least one of the literals in each clause is true?

There are two major engines for addressing the SAT problem: (1) The DPLL Engine: search the space of truthvalue assignments in a systematic, depth-first way, and (2) Stochastic Local Search: Pick one assignment randomly, then change the value of one proposition symbol at a time. Each of these have the character of computational engines.

3.1.1 The DPLL Engine

The DPLL Engine is based on a particular algorithm whose full name is the the Davis-Putnam-Logemann-Loveland algorithm, after its inventors. It obtains its strengths by the various plug-in methods that have been developed for it, and it is use of these additional methods that give it the character of a computational engine rather than just an algorithm.

The basic algorithm is as follows. Given a set A of clauses:

- Pick one of the proposition symbols in these clauses, e.g. p, and construct one modification of A for each of the two truth-values. Obtain A(p) by removing all clauses containing p and by removing -p in all clauses where they occur. Similarly for A(-p).
- Repeat the same operation with another prop symbol, obtaining e.g. A(p,-q), and proceed recursively obtaining a search tree.
- If a branch obtains a descendant of A containing both {a} and {-a} for some literal a, then close that branch, i.e. do not expand the tree further from that point. This is called a conflict.
- If you can find a branch where all the prop symbols have a value, then you have found an assignment. If all branches become closed then no assignment can exist whereby all clauses are true.

For an example of this algorithm, please see the e-slides ('powerpoint') for Lecture 13.

The crucial issue in executing this procedure is to decide which proposition symbol to use next, during the search in a particular direction of the search tree. The following are some major *decision strategies,* i.e. rules for how to make this choice.

- Maximum Occurrence in clauses of Minimum Size (MOMS) as a goodness measure for selecting prop symbol.
- Dynamic Largest Individual Sum (DLIS): choose the literal occurring the most frequently in the clauses at hand.
- Variable State Independent Decaying Sum (VSIDS): keep track of the 'weight' of each literal, allow it to 'decay' i.e. it is gradually reduced over time, but if a literal is used for closing a branch then it is 'boosted' (its value is increased) for use elsewhere in the search tree.

There are also several other decision strategies, some of them fairly complex. One interesting method is *Clause Learning and Randomized Restart* which works as follows. The basic algorithm is modified so that if you arrive to a conflict, then analyze the situation and identify what clauses contributed to the conflict. Extract one or more additional "learned" clauses that are added to the given ones. Also, identify the level in the search tree that one has to return to. Proceed from there.

Then, from time to time you let the search process do a randomized restart, i.e. it restarts the search process from the root of the search tree, but retains the learned clauses. The purpose of these techniques is to let the process "learn" more direct ways of arriving to the desired result in the sense of the closing of a branch in the search tree.

In addition there are the following important implementation considerations that contribute to keeping down the search.

- Do a (modified) depth-first search, not a breadth-first search of the tree of possible assignments.

- Implement iteratively rather than using recursion.
- Literals in unit clauses are immediately set to true (as a preprocessing step and during the computation), except if you have a conflict (in which case close that branch).
- Proposition symbols that only occur positively in all the given clauses are immediately set to true and those clauses are removed. Conversely for those prop symbols that only occur negatively.

3.1.2 Statistical Local Search Techniques

The basic method in this approach to pick an initial assignment randomly, and then to change the value of one proposition symbol at a time, in such a way as to gradually approach a solution to the given SAT problem.

A number of techniques of this kind exist. We only consider one, called GSAT (G for Greedy). The basic idea in GSAT is: Start with an randomly chosen assignment. Calculate, for each proposition symbol, the increase or decrease in the number of clauses that become true if the value of that prop symbol is reversed. Pick the one that gives the best increase. Repeat this process until a satisfying assignment has been found or a maximum number (max-flips) has been reached. If max-flips has been reached, then try another randomly chosen assignment. Repeat until success or until a maximum number (max-tries) has been reached.

3.1.3 State of the Art for SAT Solvers

SAT solvers have been strikingly successful, both within Artificial Intelligence and in other areas. In principle, they provide a method for combinatorial reasoning and search which is able to handle very large sets of clauses. In comparison with the use resolution theorem-proving, SAT solvers use a more primitive representation, but they have the advantage of a number of very efficient implementation techniques.

3.2 Constraint Programming

Constraint programming is an additional and important software technique that exhibits some similarities with SAT solving, but also many differences.

In general, a constraint programming problem specifies:

- A set of “variables”
- A domain of possible values for each variable
- A set of constraints (“relations”) on these variables

An assignment of values to the variables that satisfies the restrictions is a solution to the constraint programming problem.

This is similar to SAT solving in the sense that one searches for an assignment of values to variables, but SAT solving is concerned with assignment of truth-values whereas constraint programming can be applied to any kinds

of values, for example integers. Another difference is that constraint programming requires the facilities of a programming language, so that it is realized by extending a programming language with facilities for stating and solving constraint programming problems. Logic programming was the original host language for constraint programming. In this case we talk of constraint logic programming.

A tight integration of constraint programming in its host language requires that it should use as much as possible the data structures, declarations, and operators on data that are provided by that language. Therefore, constraint programming is the most easily hosted by languages with an interpretive character, e.g. functional programming languages (including Lisp and Scheme) and even Java, besides logic programming languages. However, constraint programming packages do exist even for C++.

The Wikipedia article on Constraint Programming is a useful source of examples and additional information.