Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Corso di Laurea in Informatica

AN EDUCATIONAL TOOL FOR AGENT PLANNING IN AN AI COURSE

Relatore: Prof. Mario ORNAGHI

Autore:
Riccardo Cantoni
Matricola: 820800

Anno Accademico 2015-2016

Preface

This work tries to be a useful tool for all those students who start measuring themselves with programming simple intelligent agents, in the context of an AI course, with ProLog.
I became interested in the field of Artificial Intelligence after attending the course myself, and although I found both logic programming and agent planning to be intricate concepts at first, I came then to realize how powerful and interesting tools they can be, once you start to understand how they work.
In particular I found the sheer concept of being able to produce a piece of code that is capable of exhibiting intelligent, although simple, behaviour to be quite an exciting challenge.
What I am trying to achieve with this is to produce a piece of software that can be used to introduce students to said challenge. Providing enough power and flexibility to allow interesting and stimulating applications, while retaining sufficient simplicity and clarity to be comprehensible and useful for the education of students with little or no experience in the field of Artificial Intelligence.
I would also like to thank professor Ornaghi for giving me this opportunity, and for all the help and support he provided me with during these months.

Contents

Preface **ii**

1 Introduction **1**
1.1 Intelligent agents 1
1.2 Problem solving 2
1.3 State spaces and state space problems 3
1.4 State space graphs and search trees 5
1.5 Search strategies 7

2 The Development **17**
2.1 Features and specifications 17
2.2 Preexisting code 19
2.3 Development 20

3 User Manual **25**
3.1 Installation and loading 25
3.2 Problem specification 26
3.3 Solving problems 29
3.4 Interactive mode 32
3.5 Saving data 34
3.6 Exporting results 35
3.7 A simple use case 37

4 Goal-oriented Action Planning **50**
4.1 Goal-oriented behaviour 50
4.2 Goal-oriented Action Planning 54
4.3 A GOAP model 55
4.4 Experimentation 60

Chapter 1

Introduction

1.1 Intelligent agents

An agent is any kind of autonomous entity that acts in a certain environment. The interaction is carried out by the means of sensors that provide data about the environment, and actuators that modify the world.
The structure of the agent consists of an architecture, and a program. The architecture acts as an interface between the perception-actuation apparatus and the program, transmitting the collected data to it, and feeding the subsequent controls issued by the program to the actuators.
The job of Artificial Intelligence programmers is to design and produce the agent program. Like any other program, the agent program can be described as a function: in this case it maps from the agent's percepts to its actions.
A minimal example of a (barely) intelligent agent can be as simple as a thermostat controlled by a reflex function: if the numerical input produced by the sensor, being in this case the measured temperature of the room, is greater than a certain threshold, a reflex response will trigger a command, and the actuator will turn off the heating. This simple approach can, in some cases, be used in more complicated tasks, like a self driving car braking when the red lights of the vehicle in front of it turn on.
The structure of the program in a reflex agent is basically just a static table of reactions to specific conditions. when the percepts match one of these conditions, the appropriate response will be triggered.
The behaviour of this kind of agent only depends on its perception at the current instant of time. This is insufficient for most applications: even in the case of the aforementioned thermostat, the sudden and brief change in the air flow in the room caused by someone going through a door could produce a temperature variation that could in turn trigger an unwanted reaction in the agent program.
An internal representation of the state of the environment, on top of sensor data, is

therefore necessary. In this case it can be something as simple as the average measured temperature in the last five minutes.
For a driverless car we will need something with a totally different degree of complexity: the position, size and speed of all the vehicles in the area, pedestrians, traffic lights and so on.
Although a good knowledge of the environment and its evolution is a huge leap forward in the construction of an intelligent agent, for it to decide what to do is often necessary to define some sort of goal: a desirable condition representing an acceptable end to the chain of actions that our agent will take.
The program will then be able to combine the definition of its goal with its knowledge about the outcome of all the available actions to decide what to do in order to achieve said condition.

1.2 Problem solving

Having an internal representation of the outside world, coupled with knowledge about the effects that various actions will have on it, allows agents to plan ahead before actually executing their actions, thus being able to devise the best possible course of action for their current status.
Intelligent agents are supposed to undertake various actions that will change the state of the world to a point where the goal is satisfied. Actions are therefore the operators that produce this modification.
This said, the solution of the problem of inducing a desired state in the environment can be reduced to a list of actions, a plan that can then be carried out to fulfill the goal.
The quality of the agent program is therefore heavily dependant on how well we define the internal representation of the environment, the actions with their effects, and the goal.
First of all it is of primary importance to exclude from the computation all those characteristics of the environment that are of no interest to the task the agent will be faced with.
Again in the case of a driverless car, the cameras that act as sensors for the agent driving the vehicle will record a large quantity of useless data that are of no concern to the virtual driver, even data about objects that are usually very, such as other vehicles, can and should be discarded when they are moving on a road that does not intersect the one where the car driven by the agent is.
If not discarded, this large volume of unnecessary information will increase the complexity of the state representation, to a point where the computation required to plan its evolution might be impossible.

An other very important parameter that should be carefully considered is the level of abstraction the agent will work at: the task of planning the route between city A to city B can not take into consideration the response to traffic lights or to the movement of other vehicles, while the task of avoiding pedestrians has no connection with the general route to follow.
Usually the problem of integrating all these different levels of abstraction is solved by using a hierarchical structure of different agents, each of them working at a different level.
Differences in the amount of information about the environment accessible to the agent causes differences in how the agent can reason: this produces three classes of problems [2].
If the agent has complete knowledge about the state of the world and about the transitions between states that will be caused by its actions we have what is called a single state problem. If instead the amount of information provided by the sensors is insufficient to fully know the state of the environment, the agent will not always know exactly what the current state is, or what will be the resulting state after applying a certain transformation.
Depending on the level of incompleteness the agent could be able to define a set of states among which the current one is certainly found. These are called multi-state problems.
Finally, an agent could have no knowledge about the effects of its actions, and it is therefore forced to experiment in order to gain such knowledge.
This experimentation can not be carried out on the internal representation of the world and can only take place in the actual environment itself, with all the risks that this produces. These are referred to as exploration problems.

1.3 State spaces and state space problems

We introduced the basic pieces that make up the definition of a problem: states, actions, and goals. We shall now analyse this structure in a more precise and formal fashion.

- The set of all possible states is called the state space. It contains all the states that can be reached by the agent.

- Among these states there is one in which the agent knows to be at the beginning of the solution process. This is referred to as the initial state.

- A set of actions, also called operators, that are available to the agent. Not all of them are always possible at any given time. Therefore they must be accompanied by additional information to distinguish which ones are available to the agent in a specific state.

- A function that, given a state, returns the set of states generated by applying every possible action to it. This allows an agent to simulate the results of its choices.

- A goal function, that given a state, determines if said state is a goal, or final. We assume the information contained in whatever structure was chosen to define a state to be sufficient to allow this calculation. This function must necessarely be boolean, although its structure and degree of complexity can widely vary: sometimes it is simply defined as a collection of acceptable states, sometimes as an abstract property relative to the agent, or to the world in which it acts.

- In many cases, it is important to distinguish between solutions with a different degree of quality. This value can then be used to filter out unacceptable solutions or to choose the optimal one.
  As explained in the previous section, a solution is a list of actions, more specifically a path from an initial to a final state. We can then calculate the abovementioned quality value by the use of a path cost function that assigns a cost value to the path that defines a certain solution. Usually, although not necessarily, the cost is calculated as the sum of the costs of the actions along the path. Any other approach can be used, as long as the function associates a precise cost value to any path.

More formally, a state space is defined as the t-uple
$\mathbf{S} :< S, A, successor(s), goal(s), cost(s, a) >$ where:

- S is the nonempty set of possible states.

- A is the nonempty set of possible actions.

- $successor : S \to \mathcal{P}(S)$ is a function that, given a state $s \in S$, returns the set containing all the states obtainable by applying currently available actions to

s.

- $goal : S \rightarrow \{0, 1\}$ is a boolean function that returns 1 if $s \in S$ is a final state or 0 if it is not.

- $cost : (S, A) \rightarrow \mathbb{R}$ returns the cost of applying a to s, given $s \in S$ and $a \in A$. the return value is typically positive.

A formally defined state space **S** can then be considered as the frame, or environment, in which we define a state space problem to be solved by an intelligent agent.
$\mathbf{P} :< \mathbf{S}, s_0, G >$ where:

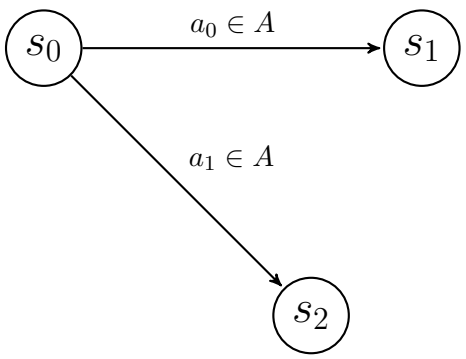
- $\mathbf{S} :< S, A, successor(s), goal(s), cost(s, a) >$ is a state space.

- $s_0 \in S$ is the initial state.

- $G \subseteq S$ is the set of all final states $\{s \in S \mid goal(s)\}$. It contains all states that verify the goal function.

1.4 State space graphs and search trees

Any state space **S** can be treated as a graph $G :< V, E >$ where V is a set of vertices, also called nodes, and E is a set of oriented edges connecting couples of nodes.
In the case of a graph G_s representing a state space
$\mathbf{S} :< S, A, successor(s), goal(s), cost(s, a) >$, we have $G_s :< S, E_s >$
where $E_s : \{(s_0, s_1) \mid s_0 \in S, s_1 \in successor(s_0)\}$.
For instance if we have a state s_0, for which $successor(s_0) = \{s_1, s_2\}$, it is possible to represent the corresponding graph as follows.



Representing a state space or a state space problem as a graph allows to treat the process of solving this problem as the relatively simple task of finding an uninterrupted path from the initial state to any final state contained in the goal set.
However, the state space graph is usually extremely large, even infinite, or simply unknown in its entirety. This makes keeping it in memory unfeasible.
The graph is therefore generated and stored during the exploration process: at each step, if a goal state is not yet found, new nodes have to be generated from already known ones in order to push forward the boundaries of the agent's knowledge. Further away from the initial state and, hopefully, closer to the goal.
This operation of generating new nodes is called node expansion. It is carried out by applying to a known node $s \in S$ the *successor* function. The resulting set of nodes is added to the graph with the corresponding set of edges outgoing from s. These nodes can in turn be expanded.
The process can be viewed as the construction of a search tree, that covers the nodes of state space graph over multiple steps. This tree has its root in the initial state, while the leaves are nodes that can not be expanded for which $successor(s) = \emptyset$, or nodes that have yet to be expanded.
At each step, a leaf is picked for expansion and a number of new nodes can be generated. The selection criterion defines the search strategy, which is an extremely important component of the solution process and will be discussed in the next section.
The structure that holds the nodes to be expanded is called fringe, or frontier. Different implementations can have impacts on the node selection process, thus having different effects on the resulting search strategy.
If every explored path leads to a dead end, eventually the fringe will be empty. In this case the task of solving the problem is terminated, and no solution can be found.
It is also possible to find multiple, different solutions. Either because they are found at the same step, or because the computation is continued after the first one is found. In this case the *cost* function can be used to evaluate the quality of these results, finding the optimal solution, or discarding unacceptable ones.

Finding the best possible solution might require more expensive calculations, or the analysis of many possible candidates. For this reason it might be necessary to trade a high or optimal solution quality for a shorter computation time. This is particularly common in real-time environments, where time requirements are more strict.
Although it is probably the most common case, nodes in the graph do not necessarily represent states of the world and solutions are not necessarily paths from the initial state to a final one.
In some cases it is useful to model the nodes to represent partial solutions or sub-problems, viewing the search as a process of specialization and refining from the vague to the specific, rather than the evolution of the initial situation into the desired one. The space described is not the space of the possible states, and it's rather called problem space or solution space.
This approach does not require any change to the general search procedure, while allows the solving of a wider set of problems, for instance in the field of combinatorial optimisation, where a notorious example is the Branch and Bound search strategy explained in section 1.5.

1.5 Search strategies

With the above explained structures and functions, it is possible to define an algorithm for the solution of a generic search problem.
the function $select : F \to S$, where $F \subseteq S$ is the fringe, chooses a certain node to be processed next.

```
Fringe = {S0}
LOOP
     IF (Fringe == {})
          return(search failed)
     END IF
     N = select(Fringe)
     Fringe = Fringe - {N}
     IF goal(N)
          return(solution)
     ELSE Fringe = Fringe U successor(N)
    END IF
END LOOP
```

This algorithm terminates after reaching a first solution. It is possible to change it easily so that it keeps searching until it finds a specific number of solutions.
However, the state space S could be infinite. In this case it is not guaranteed that the algorithm will ever terminate, since a solution could be impossible to find, and the fringe would never be emptied out.
Properties of the algorithm are determined by the search strategy, carried out by the means of the *select* function. These properties are useful to measure the quality of a particular strategy, thus allowing to determine which strategy is to be used under certain circumstances.

- Completeness: is the strategy guaranteed to find a solution where there is a reachable one?

- Optimality: provided that there are multiple solutions, will the strategy find the one of highest quality?

- Complexity in space: how much memory will the strategy need?

- Complexity in time: how much computation time will be required?

Search strategies are usually classified under two groups: uninformed, or blind, strategies have no information about what lies between the current state and the goal.
The only knowledge at their disposal is whether a certain node satisfies the *goal* function or not. Here are shown some of these strategies.

- Breadth-First search: the root node is obviously expanded during the first iteration of the loop. In general, the *select* function picks the nodes that have the lowest depth in the search tree.
  This behaviour is usually produced by implementing the fringe as a "First in-First out" (FIFO) queue. This way, it is guaranteed that nodes at depth d will always be expanded before nodes at depth $d + 1$.
  This strategy is complete, because it is guaranteed to find a solution where one exists. And if there are several, it will find the solution with the shortest (in terms of the number of nodes) path.
  This means that for the strategy to be optimal, the function that calculates the cost of a path, must be a non decreasing function of the depth value of the node. Usually this only happens when the cost is equal for every edge, so that

$cost(s, a) = k, \forall s \in S, \forall a \in A$.
Then, being c_s the cost of the solution path connecting the root node to the node s, and d_s its depth in the tree $c_s = d_s * k$.
In order to calculate the requirements of this strategy in memory space and time, we need to introduce the concept of Branching Factor. The branching factor b is the number of successors each node can have. In most cases b is not the same for every node in the tree, and an average value is used.
It is immediately clear that the number of nodes at a give depth d in a tree with branching factor b is equal to b^d. This means that if a goal state exists at depth d, the maximum number of nodes that have to be processed before reaching that node is equal to $b + b^2 + b^3 + \cdots + b^d$, meaning $O(b^d)$ in the worst case. The asymptotic complexity in space results to be the same, since all the leaf nodes must be maintained in memory at any given time.

- Depth-First search: the *select* function chooses the node with the highest depth value to be expanded. this is usually implemented by the use of a "First in-Last out" (FILO) stack as the fringe, so that the first node to be expanded, is the last one to have been generated. Therefore, since the last node to be expanded was the deepest at depth d, the next in line will be one of his successors, with depth $d + 1$, being again the deepest.
  A progressively longer and deeper path will be explored, and only if this reveals to be a dead end, terminating with a non-expandable leaf, will the search explore a new, shallower path.
  This strategy is very space-efficient: at any given time it is only necessary to store a single, root-to-leaf, path, along with diverging paths that could be necessary if the current one proved to be unsuccessful: meaning bm nodes, in a state space with maximum depth m.
  In the worst case, still, every single node with $d \leq m$ will have to be explored, if the first goal state happens to be in the last explored path, at depth m. Therefore the complexity in time remains the same as Breadth-First, being $O(b^m)$.
  The problem with this strategy is that it is nor complete, nor optimal: in the case of an infinite tree, for instance, the search will continue to explore the first available path deeper and deeper, even if a shallow solution exists, thus finding a solution with a longer path, or potentially never terminating.

- Iterative Deepening Search is really the evolution of an other search strategy called Limited Depth Search. Since it is clear that complexity in both space and time increases with the number of nodes, and the latter grows exponentially with deeper branches, it is possible to limit the growth of the resources needed

by the algorithm limiting the maximum depth of expandable nodes.
This is used in conjunction with Depth First search to avoid the scenarios where the algorithm is stuck following a very long or infinite path.
The Iterative Deepening search strategy tries to solve the problem with every possible depth limit, starting from 1, then 2, 3... until termination.
This combines the useful properties of a Breadth First search, completeness and optimality, with the efficient memory usage of Depth First.
The number of states expanded, however, is greater than that of Breadth First, since nodes are expanded several times. A node with depth d, is first expanded when the limit is equal to d, and then again when it is $d+1$,$d+2$ and so on.
In fact, the total number of expansions in a tree with branching factor b and depth limit d, is $(d+1)+(d)b+(d-1)b^2+\cdots+2*b^{d-1}+b^d$.
The weight of the overhead depends on the branching factor b. A high branching factor means a smaller overhead, since most nodes will be at higher depths, and will be expanded less times.
The Time complexity is then again $O(b^d)$, while the efficiency in memory required from Depth First is retained, with $O(b*d)$. For these reasons Iterative Deepening is the most widely used uninformed strategy.

Although uninformed search strategies provide the means of solving many problems, they are severely limited by their inherent inefficiency: they blindly generate new nodes and test the goal function on them without any knowledge about the actual structure of the problem.
Informed strategies, instead, use problem-specific knowledge to produce a great increase in efficiency. The *select* function evaluates the nodes contained in the fringe, assigning a "desirability" value to each one of them. The selection itself is then carries out based on this value, and the most desirable node is picked. For this reason these strategies are also called Best First.

- Greedy Search: this strategy relies on the simple idea of finding the goal by minimising the cost to reach it. The node selected is always the closest to the goal.
  The problem is that accurately measuring the cost of reaching a goal from a specific state is rarely possible. It is then necessary to estimate such cost in a less precise way.
  A function that estimates this value is called a heuristic function, $h(s)$, where $s \in S$ is the state we are currently processing. $h(s)$ therefore estimates the cost of the shortest path from s to any final state.
  The behaviour of Greedy Search is in a way similar to the one of Depth First Search: both strategies prefer to explore and follow a long and deep path all

the way to the end, where the goal hopefully is. Both strategies will go back on their steps and follow again an other path if the previous one proves to be a dead end.
Like Depth First, Greedy Search is nor optimal nor complete, since an infinitely deep path will produce an infinite computation, even when a shallower solution is present.
The asymptotic complexity is therefore $O(b^d)$, where d is the maximum depth of the search tree, but a good heuristic function can prove to be very effective in reducing the number of nodes expanded. In fact, in some cases a very precise heuristic function can allow the *select* function to always choose nodes that are along the path to the goal.
The reduction in the number of nodes expanded can widely vary depending on the particular problem and the quality of the $h(s)$ function.

- A*: the *select* function uses again an assigned value to choose which node is to be expanded first.
  The calculation of this value for a given node s uses both the estimated distance from it to a final state $h(s)$, and the cost $g(s)$ of the shortest known path to reach s from the root node.
  The value used is then $f(s) = g(s) + h(s)$.
  Since $g(s)$ gives the exact path cost from the start node to node s, and $h(s)$ is the estimated cost of the cheapest path from n to the goal, we have that $f(s)$ is the estimated cost of the shortest path from the root node to a goal node passing through s, being $s_0, \ldots, s, \ldots, s_f$ where $goal(s_f)$ is true.This in turn is the cheapest solution to the problem that passes through s.
  The *select* function then chooses the node with the lowest $f(s)$ value, being the state that belongs to the shortest solution.
  Properties of the heuristic function $h(s)$ have a strong impact on properties of the overall search strategy.
  In fact, optimality A* has been proved to apply when the $h(s)$ function never overestimates the distance between s and the goal [2].
  $h(s)$ is then said to be admissible. The property formally applies if
  $h(s) < \mathbf{h}(s) \forall s \in S$, where $\mathbf{h}(s)$ is the true distance between s and the goal.
  Completeness of the strategy depends on the structure of the search space. A* expands nodes in order of increasing f and will eventually reach a goal state s_g for which $f(s_g)$ is certainly finite, unless there are infinite nodes s_i for which $f(s_i) < f(s_g)$.
  This can only happen if the graph is not locally finite, thus there are nodes with infinite branching factor, or if there is some path with infinite depth but finite cost.

A* is therefore complete in the case of locally finite graphs, where all actions have positive costs.
The main strength of A* is its efficiency. No other strategy that relies on expanding nodes from the initial state is guaranteed to expand fewer nodes than A*, while remaining complete [3].
Still, for most problems, the number of nodes with an $f(s)$ value less than or equal to the goal node grows exponentially in the length of the path to the solution. It has been proved that this exponential growth will occur unless the error in the heuristic function does not grow faster than the cost of the effective path cost, so that $\mid h(s) - \mathbf{h(s)} \mid \leq O(log(\mathbf{h}(s)))$.
Most heuristics, while greatly improving the efficiency of the search, retain an error that is at least proportional to the path cost, still producing an exponential growth in the number of nodes.
The main problem with this strategy, however, is the complexity in space. In fact, all of the generated nodes are stored in memory, leading to an exponentially growing need of memory that soon overtakes any machine.

A system that can be employed to increase the efficiency of all the above mentioned strategies is to prevent the *successor* function from repeatedly generating already processed nodes that would lower the overall efficiency of the search.
Not all problems introduce this issue: in some cases, states can only be reached one way and will therefore be only generated once.
In most cases, however, states do occur multiple times and can make the search tree infinite. Pruning some or all of them can restrict the tree to the size of the actual problem space, reducing the total number of nodes and the resulting need for resources.
Even when the search tree is already finite, avoiding repeated states can lead to an exponential reduction in the cost of the search.
There are three ways to deal with the problem:

- Do not generate nodes that are the same as the parent of the node currently being expanded.

- Do not generate successors to nodes that are the same as any of the ancestors of the node currently being expanded. This will prevent cycles from being created.

- Do not generate nodes that have already been generated before.

The three options are here displayed in increasing order of effectiveness and computational overhead. The trade-off between generating more nodes during the search and storing and checking them to avoid repeating them depends on the problem.
Some problems, as previously stated, never produce repeating states and would therefore making the whole idea redundant, while in other cases the amount of unnecessary nodes that would be generated makes the extra cost worthwhile.
In particular the last option requires all previously generated nodes to be maintained in memory, resulting in a potentially dangerous exponential space complexity of $O(b^d)$.
This additional functionality can then be used to obtain a more efficient version of the generic algorithm with a function $prune : S \to S$ that removes duplicates before adding new nodes to the fringe.
A search strategy that explicitly takes advantage of the increase in effectiveness obtainable with an accurate pruning is the Branch and Bound Algorithm[6].
Unlike the strategies previously explained, Branch and Bound does not work by generating states as nodes of a graph until it finds an acceptable one. Instead, it partitions the problem into sub-problems, while discarding paths that are certainly not going to lead to the optimal solution.
In the case of the pathfinding problem of going from the point X to Y, rather than states that represent the positions a, b, c, nodes would represent the sub problems of going from a to Y,from b to Y,from c to Y.
While the branching procedure could eventually lead to the generation of all possible solutions (with an exponential complexity), it is the bounding technique that can greatly increase the total efficiency.
The bounding works by computing, for any generated node n, the maximum and the minimum quality of any solution belonging to the sub-tree which has n as root. Then, these bounds are evaluated against the quality of a known solution, and if they are worse than the quality of that solution, or better than the best possible quality, the corresponding branches are pruned, and if they are better .
Continuing with the pathfinding example, if a measure of the quality of a solution is the length of the road that has to be covered, so that the shortest way is the best, a simple way of calculating a bound on a node is to consider the length of the road up to that point.
If, starting from X, it takes k kilometers to get to a, it is certain that no solution that involves passing through a is going to be shorter than k, and it is therefore safe to consider k as a bound for the node representing the problem of going from a to Y.
If then a solution is found with a total length of K, every time a new node is generated with a bound greater than (hence a quality lower than) K, it can safely be discarded.
As a worst case scenario, the algorithm could find the lowest-quality solution first, then the second lowest, and so on until the optimal one is found last. In this case no pruning happens and the algorithm has to consider an exponential number of nodes.

If the function that calculates bounds is accurate, the efficiency of the algorithm depends on how good are the first solutions that are encountered: in the best possible case, the optimal solution is found immediately, and every other branch is soon pruned, resulting in a highly efficient computation.
Additional data need to be stored in memory to allow for the pruning, along with all those that are needed by any other function taking part in the search process: data pertaining to the nodes involved, structures relative to the heuristic, or to the cost function, even to the search strategy itself like the bounds for Branch and Bound.
All this has to be carried along during the search process, not unlike the fringe. Without changing the overall functioning of the generic search algorithm, a new data structure is introduced, called Generalised Fringe (**GF**), which holds the actual **Fringe** set, as well as all the auxiliary data structures that may be required during the computation, including the ones that support the pruning. The structure is also accompanied by the relative operations necessary to the management of the data it contains.
With pruning and the appropriate additions, the general search algorithm becomes:

```
GF = init(S0)
LOOP
     IF (fringe_empty(GF))
          return (search failed)
     END IF
     <N,PurgedFringe>= select(GF)
     IF (goal(N,GF))
          return(N)
     ELSE
          S = successor(N)
          PrunedS = prune(S,GF)
          GF = add(GF,PrunedS)
     END IF
END LOOP
```

Where:

- **init(S0)** initialises **GF** with the initial node **S0**.
- **fringe_empty(GF)** checks if **Firinge** in **GF** is empty.

- **PurgedFringe** is **Fringe** $- \{N\}$.
- **prune(S,GF)** discards the nodes contained in the set **S** according to the pruning technique and the data contained in **GF**.
- **add(GF,S)** adds to **Fringe** in **GF** the nodes contained in the set **S**.

With this algorithm as framework it is possible to create a variety of specific strategies. For example if a **GF** structure is designed to be holding a set of already visited nodes (also called "closed" nodes) as well as the fringe, the A* strategy can be implemented, with the pruning technique that discards all nodes that have already been generated:

```
type GF := <Closed, Fringe>

GF = <{},{S0}>
LOOP
     IF (GF.Fringe == {})
          return (search failed)
     END IF
     <N, PurgedFringe> = select(GF)
     GF = <GF.Closed U N, PurgedFringe>
     IF (goal(N,GF))
          return N
     ELSE
          Successors = successor(N)
          Pruned = prune(Successors, GF.Closed)
          GF= add(GF, Pruned)
     END IF
END LOOP
```

An other example could be the implementation of a Branch and Bound strategy for a minimisation problem.
In this case **GF** would be required to store the best solution already encountered as the variable **Best**, as well as **Fringe**, being a set of subproblems rather than a set of states. The pruning is operated by discarding subproblems which have a calculated upper bound that is already worse than the one of **Best**.

```
type GF := <Best, BestUpperBound, Fringe>

GF = <{}, +inf, {P0}>
LOOP
     IF (GF.Fringe == {})
          IF (GF.Best == {})
               return (search failed)
          ELSE
               return (GF.Best)
          END IF
     END IF
     <S, PurgedFringe> = select(GF)
     IF (goal(S))
          GF.Best = S
          GF.BestUpperBound = upperBound(S)
     ELSE
          Subproblems = successor(N)
            Pruned = prune(Subproblems, BestUpperBound)
          GF = add(GF, Pruned)
     END IF
END LOOP
```

Chapter 2

The Development

2.1 Features and specifications

This tool was designed to meet the specific needs of students measuring themselves with the planning of intelligent agents of various complexity.
A complex problem was to achieve balance between widely adaptable, extremely flexible implementations, better suited for a more experienced user but capable of a more complex usage and standardised, highly structured functions with a simple syntax that could be more effective from an educational point of view.
For this reason, some of the features were implemented with multiple syntaxes to meet the different needs of different users.
A set of features and characteristics were immediately specified to be implemented, and to be the backbone of the tool:

- The primary function of the tool is to allow the user to launch a search procedure on the specified problem. Additional parameters are necessary to fully define the search strategy such as the search algorithm or the pruning technique, and the problem, with its initial and final states. The output of this function will be the set containing the solutions that were found.

- The tool should work like a library, to be imported in the problem specification file written by the user.

- The problem will be defined with a structured set of predicates: initial states, final states, actions, edge cost and a heuristic function to be used for an informed search job.

- The tool should be able to operate in an interactive, step-by-step mode, where the user can directly inspect every step of the search process as well as extract and export data relative to the current operations.

- The tool should keep track of statistics concerning the search tree and the computation relative to a specific problem-solving task.
  These statistics should be retrievable by the user so that they can be used to evaluate the properties of a specific implementation.

- The tool should give the means of formatting and exporting outputs in a human-readable format, as well as in a format readable by third-party, widely available, mathematical analysis softwares such as MATLAB, Mathematica or Microsoft Excel. The formats chosen were ".txt" and ".csv".

Along with these core features, during the development, an array of quality-of-life improvements, additional functions and support features were designed ad implemented:

- Support for multiple search procedures on the same problem instance with different parameters. Aggregation of the resulting output data to allow its exporting.

- Logging and journaling functions to keep track of past activities and results in a human-readable format.

- The possibility of defining initial and final states as functions rather than actual states, to allow for initialization of the problem instance via parameters and the definition of the goal as a function rather than just as a state.

- A database system for storing problem instances and states that would otherwise be long and tedious to write every time a search is launched.

- Additional optional parameters to the search definition to allow for an "oracle-given" search bound to be specified, or to limit the depth of the search tree.

- An "exploration" mode, in which the agent explores the problem space according to the search strategy, whithout stopping when solutions are found, with a

limit in depth. Useful to analyse the general behaviour of an agent in a certain environment.

- The possibility of selecting different action definition systems: non structured, with pre and post conditions, and following the STRIPS standard.

2.2 Preexisting code

I was allowed to use code from a pre-existing work [4], which had different objectives but some rather useful functions. A system was already in place to compose and launch a search procedure from given parameters, such as the search strategy to employ, the pruning technique, and the heuristic function to be used.
In order to compose a modular search strategy, the system consults specific files which contain the implementation of the functions that define a specific strategy in the context of the generic algorithm exposed at the end of chapter 1.
For instance, in order to load a specific search algorithm, this procedure is used:

1. The user requests the launch of a search with the parameter **astar**.

2. The parameters are read and the name of the requested strategy is asserted as a generic dynamic predicate with the **currentJob** and **search** identifiers, and is then treated as a global variable.

3. After all the parameters have been read and processed, the system begins the composition of the general strategy by extracting the necessary global predicates with the correct identifiers. The corresponding module files are located through predicates **dp/2** which couple a module identifier with its definition file, in this case **'strategies/astar.pl'**.

4. The necessary files are then loaded via **consult/1** providing the predicates that will be used in the generic algorithm. In this case the file contains the predicate

```
priority(n(State,_,Cost,_),Z):-
        h(State,H), Z is Cost + H,!;
        fail.
```

which contains the definition of the $f(s)$ function, described in section 1.5, as $cost(s) + h(s)$.

5. Once the loading of all the necessary predicates is completed, the actual search procedure is launched, which will use the properly defined $f(s)$ value to select nodes for expansion.

In the same way the problem to work on was treated like a parameter: a problem file had to be consulted in order to have the problem specification predicates.
This system, although quite flexible, is not really fit to be used by students with little or no experience with ProLog and is hardly expandable, since the addition of any new strategy, heuristic, or even problem would require changes in both the parameter-processing code, and in the set of predicates coupling module identifiers with their corresponding files, forcing users to modify parts of the code that should really remain transparent to them.
Moreover, the system was implemented with a series of assumptions that prevented from adding any problems besides the specific ones for which it had been designed.
For example, in the case of iterative strategies, the depth limit was initialised as the heuristic value of the initial node: this only makes sense if there is a specific relationship between the heuristic value of a node and its depth.
For these reasons the code that was kept is only what manages the loading of strategy-specific files and the composition of the pruning technique.
The strategy-defining files were also kept, but some changes were necessary in order to allow the system to work with less assumptions and restrictions, on a wider array of problems.
This preexisting system, properly corrected and adapted, proved to be a solid basis on which to build an effectively usable tool.
An other functionality that was kept is the part of code that keeps track of computational statistics about search procedures. Values are updated during every step of the search cycle and stored in generic predicates with identifiers used like global variables.

2.3 Development

A bottom-up approach was used to develop the features, starting from the most simple functionalities, and adding them up hierarchically composing more complex procedures.
Bottom-up development was preferred to top-down (starting from the big picture,

then refining its components recursively) for two reasons: first of all logic programming doesn't support any standardised "mocking" technique that would be necessary to test unrefined functionalities, and second ProLog lacks high-level code inspection tools that would be needed to find and tweak sub-modules during the inevitable bug fixing phase.
The bottom up approach, instead, allows thorough testing on individual, finished sub-modules and predicates, and in theory, if this is done correctly, these sub modules will not need to be touched in the bugfixing phase of the finished (or hierarchically superior) function.
The First step in the development of the framework was to remove all the unnecessary code. Basically all that was kept is the predicate **multiheur/0** in the file **multiheur.pl**, along with all the auxiliary files and folders containing predicates necessary to its functioning.
multiheur/0 initiates and launches the generic algorithm. In order to do this, many different input parameters have to be collected, so a new input/output system was implemented.
The predicate had to collect inputs from a variety of different files, containing user written predicates. To improve the usability and the clarity of the framework, all the input system was redesigned to pass through a single channel: an interface file, initially called **toolInterface.pl**, holding all the predicates which allow for any user interaction.
This file also served the purpose of facilitating the installation and launch procedures: instead of writing several predicates in several specific files, and then launching the solution predicate from an other one, users only need to write one file (containing the entire problem specification), that consults the interface that can be considered as a library.
The variety of predicates that were needed to specify a search strategy was also replaced with a single one, with multiple parameters.
This way, all it is required to effectively design an agent and start its activity is to write the predicates that design it in a file and then call the single predicate **solve_single/5**.
The next step was to make the system capable of handling a wider variety of problems and agents: in particular a system that allows the user to define multiple heuristics for the same problem was designed, so that the desired one can be specified as a parameter to the **solve_single/5** predicate.
In order to implement some of the features, it became necessary to slightly modify the code that implements the generic algorithm. In particular additional checks were added before the calculation of the $successor(n)$ function to ensure that the node is not expanded under certain conditions:

- If the search is an exploration search, called from the predicate **explore_space**, and the relative depth limit has been reached.

- If the user defined a depth limit as a search parameter and it has been reached.

- If the user defined a static, "oracle-given", bound and the cost of the current node already exceeds it.

The general loop structure of the algorithm also required an addition: in order to allow for user interactions between individual iterations of the search loop, a breakpoint was inserted at the beginning of the iteration.
At that point it is checked if the execution of the current iteration should be delayed to allow for user interaction. If that is the case and the user asks for the search to continue, the normal flow of execution remains unaltered.
If instead the user issues a command that causes the search to be paused, the global predicate **pause(n)** will be false, causing a change in the execution flow.
The predicate that implements an iteration for the process is then called again with the same parameters that were used at the last step, causing the current iteration to start anew.
Otherwise, if the search can continue, The **search/4** predicate, that implements a single iteration of the search loop, is called at the end of the loop iteration in order to start the next one, with new parameters calculated during the iteration.

```
search(P):-
        breakpoint,
        pause(n)->(
                ...
               %normal loop iteration
                ...
                search(P1)
                ); search(P).
```

The various other features were implemented in separated, library-like files, but can only be accessed via predicates located in the main interface:

- **db_lib.pl** contains the predicates used to support the functioning of the transparent database.

- **export_lib.pl** contains the functions that allow the formatting and exporting of data to output files or to the log file.

- **interactive_lib.pl** contains functions and predicates relative o the interactive mode.

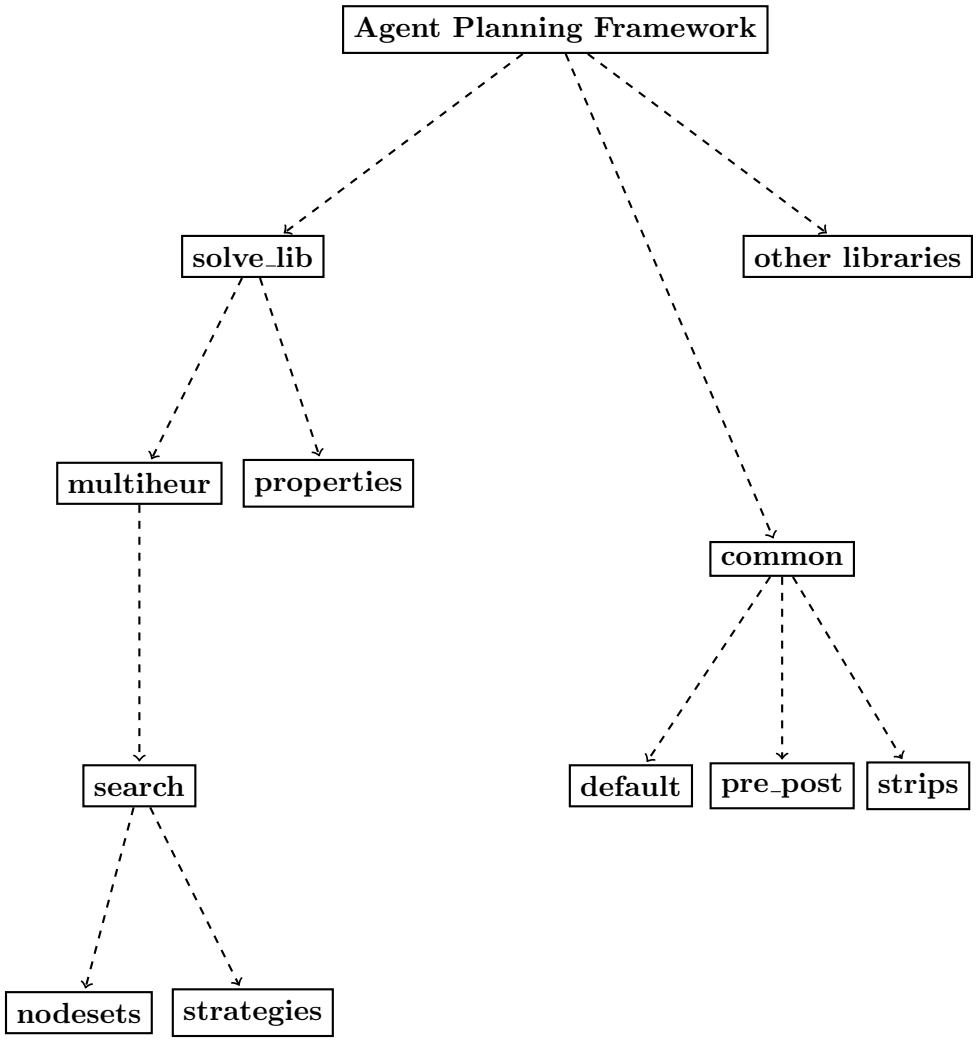
- **msg_lib.pl** contains the functions that display messages to the user, mainly the various help predicates.

- **solve_lib.pl** contains predicates that ensure two-way communication between the user-visible predicates in the interface file and the rest of the framework.

The implementation of the three action definition languages was kept separated from the rest of the code. The three languages are defined in separate files called **default.pl**, **pre_post.pl**, and **strips.pl**, the file **common.pl** holds the parts of code that are shared between the three as well as the predicates that manage the loading and unloading of any of them.
As last part of the development process, a general reworking of the architecture of the framework was done, to make it into a prolog module.
The interface file **toolInterface.pl** was renamed **Agent Planning Framework.pl** made into the ProLog module **apf**. This ensures the hiding of all predicates that are used inside the framework from any external accidental use, and also makes for a clearer and cleaner interface to the used, by presenting only the predicates explicitly exported through **module(apf,Predicates)**, being **start/2**,**goal/2**,**cost/3**,**heuristic/3** that are crucial to the definition of the agent, as well as those predicates that are used to define actions (see chapter 3).
Any predicate that is not exported explicitly will have to be renamed with the addition of the preamble **apf:**. This way any accidental use of predicates that are hidden in the module is prevented.
In order to ensure the hiding of the predicates coming from files that are only loaded after the loading of the module, and would therefore remain unhidden, a refactoring of the files defining action languages was done, keeping only one of them outside the main module. This predicate, **do/3**, was renamed **apf:do/3**, effectively including it

manually in the main module.

The final architecture of the framework can be described by the following graph. nodes represent files or groups of files similar to each other, while "UML-like" dependencies are represented by a discontinuous line.



Chapter 3

User Manual

3.1 Installation and loading

An installation of SWI-ProLog is required: the recommended version, for which the software was realised and on which the code has been tested, is the 7.2.1.
The software is distributed in the form of a compressed folder. Once Decompressed, the main directory willl contain two ProLog files, namely **Agent Planning Tool.pl** and **log.pl**, as well as two folders: **Problems**, which contains a few examples of how the framework can be used, and **framework**, which contains the framework code.
No actual installation or setup is required: once the package has been decompressed, it is sufficient to load the module file **Agent Planning Tool.pl** with the directive **use_module/1**, passing the path of the file as parameter.The name of the module is **apf**, and the predicates that are to be used as commands to the module will require the preamble**apf:**.
The loading of the module will show a series of errors and warnings regarding the mandatory predicates that have to be defined by the user, even once they are in fact defined. This is a consequence of how ProLog handles modules, and they can be safely ignored.
The warnings can actually be suppressed by setting the ProLog flag **warn_override_implicit_im** to **false**, while the errors cannot be disabled.
At any moment **apf:help_it/0** and **apf:help_eng/0** can be used to view a short text that introduces the main features of the tool in italian or english respectively.

3.2 Problem specification

In the file containing the problem definition five elements are mandatory, these predicates are explicitly exported by the module and will therefore not require the **apf:** preamble:

1. A **start(+InitParameter,-S0)** predicate, which is to be used for the initialization of the problem space, and to allow the user to dynamically define the specific instance of the problem. **InitParameter** is to be used as a parameter for initialization, while **S0** is to be the result initial state for the search procedure.

2. A **goal(+GoalParameter,+S)** predicate, which evaluates if **S** is or is not a final state, using **GoalParameter** as parameter.

3. Heuristic functions, in the form of **heuristic(+Name, +Node, -H)** predicates. **Name** is to be used as an identifier, and it is necessary to request the employment of a specific heuristic function in the search procedure. **Node** is the search node whose heuristic value is to be calculated, while **H** is the corresponding result.
   Several heuristic functions can be defined for the same problem, and will be distinguished through their identifier.

4. A cost function, with the predicate **cost(+N0,+N1,-C)**. where **C** is the cost of the action-edge from the node **N0** to **N1**.
   For the purpose of this tool the assumption is made that for every couple of nodes $< N0, N1 >$, where $N1 \in successor(N0)$, here can be only one edge going from $N0$ to $N1$.
   If two edges going from $N0$ to $N1$ existed, they should correspond to two actions with different costs, or they would essentially be the same. But in that case one of the two edges would never be used, since the one with the lowest associated cost would be always preferable.
   Therefore we have that every edge going from $N0$ to $N1$ other than the lowest cost one would never take part in the optimal solution, and can be discarded.
   For this reason in the implementation of this tool, an assumption is made that the edge connecting two adjacent nodes is always unique.

5. Predicates defining the possible actions available to the agent. The tool offers

three different ways of defining such actions, with three different action definition modules.
To use a specific action language the corresponding module must first be loaded with the predicate **apf:action_language(Module)**, where **Module** can assume the values **default**, **pre_post**, and **strips**.

- The **default** language is very simple yet very flexible. It revolves around the predicates **act(+S0,-S1)** and **act(+Action,+S0,-S1)** where **S0** is a node, **S1** is a successor node to **S0**, and **Action** is the action that would produce the transition. Since everything is left to the user no structural restraint is in place, leaving complete freedom in the definition of the actions and in the structure of the state nodes.
  The two versions of the predicate **act** can be used in the same way and even at the same time, although using only one of the two is advised for clarity and readability reasons.

- The **pre_post** language introduces some degree of restriction forcing the student to use a more structured approach in the definition of the available actions. The structure remains quite simple and open, while its relative simplicity facilitates usage and comprehension.
  Every action is to be defined through its pre and post-conditions with the predicates **pre(+Action,+State)**, which checks if the pre-conditions for the action **Action** are met in the state **State**, and **post(+Action,+State1,+State2** which applies its effects on **State1** producing **State2**.
  It is shown here how, in the case of an agent that has to perform a certain activity on a graph-like map, the action of simply moving from a position to an adjacent other can be described.
  The state definition structure is not relevant. For this reason auxiliary, black-box predicates are used in the following example to extract and update values pertaining to it: **state_position(+State,?Position)** couples a state structure with a specific position on the graph, while **adjacent(?Position1,?Position2)** is true when there is an edge between **Position1** and **Position2**.
  update_position(+State1,+Position,-State2) is used to calculate the state resulting from changing the position of **State1** into **Position**.

```
pre(move(From,To),StateCurrent):-
        state_position(StateCurrent,From),
```

```
        adjacent(From,To)

post(move(From,To),StateCurrent,StateNext):-
        state_position(StateCurrent,From),
        update_position(StateCurrent,To,StateNext).
```

- The **strips** module uses the STRIPS standard [5] introducing a strict constraint in they way states are defined. The standard requires the state to be implemented as a list of fluents that describe the state of the world, and actions add and remove characteristics of the world simply through this list, adding and removing fluents.
  Actions are therefore to be defined again by their pre-conditions with **pre(+Action,+State)**, and their effects with **add_rem(+Action,+ToAdd,+ToR**
  Since states are defined as lists, the state S_1, resulting from the application of **Action** to S_0 will be S_0 plus the fluents in the list **ToAdd**, and whithout the fluents in the list **ToRem**.
  The power of this approach rests in the fixed state structure which greatly enhances the usability and clarity for the less experienced users, while retaining power and flexibility.
  In the following example the problem is the same used previously. In this case the state definition is fixed, being a list of fluents: **latitude** and **longitude** define a bi-dimensional position, while other fluents can be used to describe any other characteristic of the environment.

```
pre(move([Lat,Long],[Lat2,Long2]),StateCurrent):-
        member(StateCurrent,latitude:Lat),
        member(StateCurrent,longitude:Long),
        adjacent([Lat,Long],[Lat2,Long2]).

add-rem(move([Lat,Long],[Lat2,Long2]),
        [latitude:Lat2,longitude:Long2],
        [latitude:Lat,longitude:Long]
).
```

A built-in predicate **actions_from_path(+Path,-Actions)** can be used to extract from a sequence of states the corresponding sequence of actions that where undertaken to produce it.
This feature is particularly useful when used to define costs of actions within the predicate **cost/3**, since it makes it possible to incorporate the action that was applied along that edge into the calculation, or when applied to a solution path to extract the chain of actions that lead to the goal.
The predicate will return an empty path if the actions were defined using the predicate **act/2**, since it only describes a relation between two nodes, without giving any information about the edge in between them. It will work as usual with **act/3**, or with any other module.

```
cost(State1,State2,C):-
        actions_from_path([State1,State2],[Action]),
        cost_of_action(Action,C).
```

3.3 Solving problems

Once a problem has been properly specified, procedures can be launched to search for solutions. The tool offers two of these functions: one for attempting to solve a single problem instance, and one capable of dealing with several tasks in sequence.

- **apf:solve_single(start(+Start),goal(+Goal),+Job,-Solution,-SubOpt).** will try to solve a single problem instance. Parameters **Start** and **Goal** will be used as first parameter for the predicates **start/2** and **goal/2** in the problem specification file, described in the previous section and are meant to be used to allow for the launch of multiple search procedures without the need to modify the problem description.
  Job holds the information that define the search strategy and has to be a list of parameters, some of which are mandatory and some others are optional, in whatever order:
  - **s:Strategy**, mandatory: defines the search strategy to be be used.

- **p:Pruning**, mandatory: selects the pruning technique.

- **h:Heuristic**, mandatory: identifies what heuristic function is to be used. **Heuristic** has to be a valid identifier as part of a **heuristic/3** predicate in the problem specification file, as described in section 3.2.

- **i:Increment**, optional: if an iterative strategy is selected, **Incremement** specifies by what value the maximum search depth is going to be increased.

- **l:N**, optional: if present, forces the system to continue searching until **N** solutions are found.

- **n:NodeLimit**, optional: the search is halted when **NodeLimit** nodes have been expanded.

- **b:Bound**, optional: any node whose cost value is higher than **Bound** is pruned, particularly useful if represents the cost of a known solution.

- **d:Depth**, optional: sets a depth limit. Nodes with a depth value greater or equal to **Depth** are never expanded.

The possible values for **Strategy** are:

- **bestfirst**, nodes are selected using the value of the heuristic function.

- **astar**, A*, where the nodes are selected using the sum of path costs and heuristic values.

- **idastar**, IDA*, nodes are ordered by the sum of path cost and heuristic value.

- **idbestfirst**, IDA*, using the heuristic value only.

- **branchbound** Branch and Bound algorithm [6], with nodes ordered by heuristic value.

- **branchboundwithdepth** Branch and Bound. nodes ordered by the sum of path cost and heuristic value.

- **iterativedeepening** uses the Iterative Deepening algorithm.

If no pruning technique is selected, duplicates will always be considered for expansion, therefore highly impacting computational performances.
However, in some cases the problem itself will not allow the generation of duplicate states and pruning will introduce an unnecessary overhead. In some other cases, even, the agent considering duplicate nodes could be a desirable behavior.
Pruning can assume the values:

- **none**:no pruning is done.
- **cutcycle**: nodes that are already part of the path between the current node and the root are pruned.

- **closed**: all generated nodes are stored. When an already explored one is generated again, it is pruned (as described in section 1.5).

If the search terminates and no solution is found, the predicate fails. Otherwise **Solution** and **SubOpt** hold the solutions that were reached.
The format in which a solution is returned is **sol(FinalState,TreeDepth,Cost,Path)**, where **FinalState** is the goal state that corresponds to the solution, **TreeDepth** is its depth in the search tree, **Path** is the path that leads to said goal node, starting from the root node and descending towards it, and **Cost** is the cost value associated with **Path**.
Statistics about the last search can be retrieved even if the search fails using the predicate **stats/5**. The five parameters will hold, in order, the size of the fringe when the computation ended, the maximum size reached by the fringe during the process, the total number of processed nodes, the maximum depth reached by the search tree, and the average branching factor.
This predicate, however, will only return data about the last search, since statistics are overwritten when a new one is launched.
The same functionalities offered by **apf:solve_single/5** are accessed via **apf:solve_single/** the last, extra parameter has to be a list of options, regarding the post-computation exporting of the results. This is the subject of section 3.6.

- **apf:solve_multiple(start(+Start),goal(+Goal),+JobList,-SolutionList,-StatsList).** works similarly to **apf:solve_single/5**, with the difference that it executes a series of search procedures based on the same problem.
  Start and **Goal** are used exactly like in **apf:solve_single/5**. However **JobList**, instead of being a single one, holds a list of different search specifications that will allow the user to define multiple different search procedures to be launched in sequence.
  SolutionList and **StatsList** will return the list of solutions resulting from the series of search processes, and the corresponding computational statistics. If one of the procedures has terminated with a failure, its corresponding return value will be **sol(not_found)**.
  Again this predicate can also be used with an additional parameter in order to export the results: **apf:solve_multiple/6**.
  apf:solve_multiple/5 and **apf:solve_multiple/6** do not return suboptimal solutions. This is because the focus with these two predicate is to compare the computational efficiency of search procedures with different strategies and heuristic functions, rather than finding the actual solutions. If the aim is to find solutions perhaps it is more appropriate to use **apf:solve_single/5**.

An other function is present with a slightly different intent: **apf:explore_space(start(+Start),g**
Fringe) is useful to analyse the behaviour of an agent by seeing how it reasons.
In particular, **Start**,**Goal** and **Job** define a search procedure on a problem instance like in the case of **apf:solve_single/5** or **apf:solve_multiple/5**, but the agent will not stop after finding any number of solutions.
Instead, it will continue exploring the search space until the depth limit defined by **Limit** is reached, and then return the last configuration of the fringe as **Fringe**.
Basically the return value will hold all the reachable states at depth **Limit** or less: this is particularly effective in showing the result of sequences of possible action, even more so if inspected in interactive mode (see section 3.4).

3.4 Interactive mode

One of the main features of the tool is the possibility of inspecting the search process step by step, allowing an user to watch closely on the reasoning activity of the agent in order to better understand or improve it.
Interactive mode can be enabled by calling **apf:interactive.**, and disabled with **apf:nointeractive.**.
Once in interactive mode, when a search procedure is launched via **solve_single**,

apf:solve_multiple or **apf:explore_space**, the cycle will halt at every iteration (including the very first one) of the search loop and await for commands. Accepted commands are:

- **next**: one iteration of the loop is executed, then the system halts and awaits instructions.

- **skip N**: **N** iterations are executed, then the system awaits for input.**N** must be a non negative integer.

- **skip all**: the search continues until it terminates, without further user interaction.

- **abort**: the current search procedure is aborted.

- **help**: a brief help text about interactive mode commands is displayed.

- **show**: the current fringe configuration is displayed, along with computational statistics about the current search.

- **show withpath**: analogue to **show**, but the path from the root to every node in the fringe is also shown.

- **print**: calls the user defined predicate **myprint(Fringe,Stats)**. if no such predicate is defined it does nothing.

- **tolog**: an entry in the log file is created, which contains the current fringe along with computational statistics about the current search.

- **tolog N**: analogue to **tolog**, but only the first **N** nodes of the fringe are written. This greatly improves the readability of the subsequent log entry, especially with a large fringe. Moreover, the nodes that are not displayed are the least desirable ones, since the fringe is ordered as defined by the search strategy. **N** must be a non negative integer.

Once the search is terminated, the system will remain in interactive mode until it is disabled by user input. The interactive mode will have no repercussions on any other functionality other than the ones previously described on the search cycle.
Due to how ProLog handles its own termination and the user input, a measure is in place to prevent infinite loops when the program is terminated in interactive mode. For this reason if the user inserts 10 empty commands during a single search, the search process is aborted.

3.5 Saving data

Especially in the case where states are defined by a complex structure, typing them in very time a new search procedure is issued becomes easily and increasingly problematic. For this reason a quality of life feature was introduced, that allows a user to save and load any term in a built-in non-volatile database system in a fully transparent way.
This database is non volatile, and will remain saved between sessions.
The database is automatically loaded when necessary. **apf:db_show.** will display all the currently saved terms:

```
?- apf:db_show.
id:identifier1    term:some_atom
id:identifier2    term:[a,b,c]
id:identifier3    term:some_composite_term(term1,term2)
true.
```

In order to save a new term, **apf:db_save(+Identifier,+Term)** can be used to store **Term** with the corresponding **Identifier**. Since the identifiers have to be unique for obvious reasons, trying to save a term with an identifier already in use will not succeed.

```
?- apf:db_show.
id:identifier1    term:some_term
true.

?-apf:db\_save(identifier1,some_other_term).
name already in use
```

```
false.

?-apf:db\_save(identifier2,some_other_term).
true.

?- apf:db_show.
id:identifier1    term:some_term
id:identifier2    term:some_other_term
true.
```

To remove saved terms there is the predicate **apf:db_delete(+Identifier)**, that deletes the term with id **Identifier**. The predicate succeeds even if no such id is found, although with no effect on the database.
Stored terms can be retrieved with **apf:db_read(+Identifier,-Term)**, which unifies **-Term** with the unique term corresponding to **+Identifier**.
The database is unique. Therefore terms saved while working from a certain problem file will be visible from any other problem file that imports the same instance of the tool. For this reason it is recommended to use identifiers that differentiate between terms used for different problems and agents.

3.6 Exporting results

The system supports the exporting of search results to .txt and .csv files, which can in turn be used for further analysis or processing with various mathematical tools and softwares. This is particularly useful to draw a comparison between computational statistics regarding different search strategies used on the same problem.
Also, it is possible to export results to a log file, **log.pl**, located in the main installation directory.
Both functionalities can be accessed via the last, optional, parameter of **apf:solve_single/6**, and **apf:solve_multiple/6** which has t be a list holding atoms that identify the operation that is being requested.
Accepted atoms are:

- **txt**: the results of the search are exported as a table to the file **output.txt** located in the desired directory. the data are formatted in a way that allows the file to be readily imported in most data processing softwares.

A row of the table holds data about a single search job, the columns contain (in order) the values regarding:

1. Search parameters.
2. Number of nodes in the fringe when the algorithm terminated.
3. Maximum size reached by the fringe during the computation.
4. Total number of nodes processed.
5. Maximum depth of the search tree.
6. Average branching factor.
7. Cost of the best solution that was found. Blank if the search terminated without success.
8. Depth of the best solutions that was found. Blank if the search terminated without success.

Search	inFringe	MaxFringe	Processed	MaxDepth	Branching	Cost	Depth
search1	...	...	...	...	...	...	...
search2	...	...	...	...	...	...	...

- **csv**: analogue to **txt**, but produces a .csv file instead.
- **log**: a journaling entry is appended to the log file. Entries will contain the following information: date and time, start parameter, goal parameter, search definition parameters, solutions and statistics.

```
\%----------DATE: 2016/8/28 TIME: 17:49.
search results:
start parameters: ...
goal parameters:...
job: [s: ...,p: ...,h: ...]
```

```
solutions:
[state: ..., depth: ..., cost: ..., path: ...]
[state: ..., depth: ..., cost: ..., path: ...]
statistics: [inFringe: ...,maxFringe: ...,processed: ...,maxDept
```

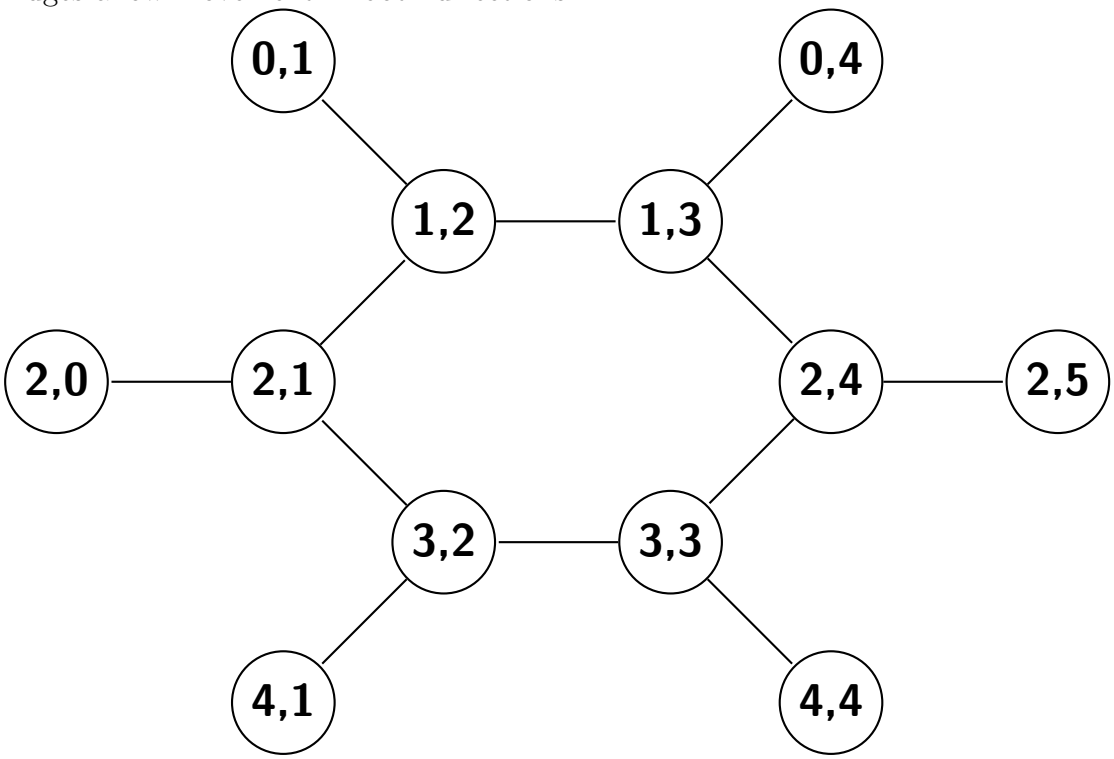
Partial results can be written in the log file during a search procedure, while in interactive mode.
the interactive mode **tolog** will generate a log entry containing the current content of the fringe, as well as the computational statistics about the search.
The statistics are partial and shouldn't be treated as a final result, since they only regard part of the total search process.
The format in which nodes in the fringe are displayed is the following:
node(s:State,d:Depth,c:Cost,h:Heur), where **State** is the state corresponding to the node, **Depth** is its depth in the search tree, while **Cost** and **Heur** are the cost value and the heuristic value of the node, respectively.
Note that the path leading to each node is not displayed, for readability reasons.
If an additional parameter **N** (a non negative integer) is added to **tolog**, the resulting command **tolog N** will generate a log entry only containing the first **N** nodes of the fringe.
Although some information is lost in this case, the fringe is ordered according to the search strategy. For this reason the nodes that are displayed are the most desirable ones, and the ones who are more likely to be explored by the agent in the next iterations.

```
\%----------DATE: 2016/8/29 TIME: 10:55.
Max Fringe size: ...      Processed nodes: ...
Max Tree depth: ...  Branching Factor: ..
FRINGE:  [node(s:...,d:...,c:...,h:...),
,node(s:...,d:...,c:...,h:...),
,node(s:...,d:...,c:...,h:...),
...,].
```

3.7 A simple use case

In order to clarify most of the features and their use, an example is usually more effective than plain written text.

This is not intended to be a full walkthrough of all the features and functions, but rather a display of how a simple agent can be modeled in a way that makes it capable of finding solutions to a simple problem. In this case we want to build an agent that can move on a graph-like topography. Its goal is to reach and pick up an item, and bring it to a designated drop-off point, then drop it.
The graph is the one shown below. Each node is labeled with its bi-dimensional coordinates, like if it was laying on a grid.
Edges allow movement in both directions.



Information about the graph structure has to be represented in the problem file.

In this case, a set of edges is sufficient to describe the structure:

```
edge(node(1,2),node(0,1)).
```

With a series of **edge(node(X,Y),node(X2,Y2))**, one for every edge in the graph, we are only representing one of the two directions in which the edge can be crossed. For this reason, when checking for adjacency between two nodes, checking the opposite direction as well becomes necessary.

Alternatively, one could double the size of the set of **edge** predicates including the opposite direction of each one.
Note that a using generic predicates in the problem definition will allow the agent to solve this problem on any graph, provided that it is properly described.

Before beginning to plan the agent, it is necessary to load the **Agent Planning Framework** module, with the directive**use_module/1**.
First of all, the problem space has to be defined. The characteristics of the world that matter to us are just a few:

- the node in which the agent is at a given moment in time.
- the node in which the item is.
- the drop-off point.
- whether the agent is still trying to reach and pick up the item, or is carrying it to the drop-off point.

For the purpose of this example, the STRIPS standard was chosen for the definition of actions and state representation, since its fixed structure makes it particularly clear and readable.
With STRIPS, the state of the world is to be represented as a list of terms, each describing a particular aspect of the situation. Given the characteristic we need to model, a natural choice of fluents could be:

- **agentpos(X,Y)**, describing the coordinates of the node where the agent currently is.
- **itempos(X,Y)**, regarding the position of the item.
- **targetpos(X,Y)**, the drop-off point.

- **phase(Ph)**, to describe what the agent is doing: **P** can assume the value **collecting** if the agent is trying to reach the item, **dropping** if the agent is carrying the item towards the drop-off point, and **dropped** if the item was unloaded.

Once the state representation is chosen, it is possible to begin writing the predicates that compose the problem definition.

1. **start(P,S0)**: this predicate should assemble the initial state **S0** from the initialization parameter **P**.
   The initial positions of the agent, of the item, and the position of the drop-off points can be left to the user. For this reason, they will have to be inserted in the solution predicate as **P**.
   The initial **phase(Ph)**, however, has to be **collecting** in every instance of the problem, and should be set by the initialization predicate.
   For safety reasons, we should check if the initialization parameter **P** is what it should be. In this case we need it to be a list containing the three fluents describing the position of the agent, the position of the item, and the position of the drop-off point.
   Therefore:

```
start(P,S0):-
        member(agentpos(_,_),P),
        member(itempos(_,_),P),
        member(targetpos(_,_),P),
        append(P,[phase(collecting)],S0).
```

2. **goal(P,S)** should be true when **S** is a final state. In our case the goal is to bring the item and drop it at the designated position. This does not require any initialization parameter, since all the corresponding information is already part of the state description.
   For this reason, **P** is not necessary and can be left as a "match anything" underscore, so that the predicate will work regardless of the input given by the user.
   The predicate checks if the item is in the designated position, and if it has been dropped.

```
goal(_,S):-
        member(itempos(X,Y),S),
        member(targetpos(X,Y),S),
        member(phase(dropped),S).
```

3. The the possible actions have to be described. since we are using the STRIPS standard it is useful to add at the top of the file the directive:

```
:-apf:action_language(strips).
```

This will automatically load the STRIPS action definition module when the problem file is opened, so that it won't be necessary to do it manually every time. the standard demands the actions to be defined through **pre(Action,State)**, and **add_rem(Action,ToAdd,ToRem)** predicates, for each one of them.
The actions available to the agent are:

- Moving from one position on the graph to an adjacent one. Actually it can move in two different ways: towards the item in order to pick it up, and towards the drop-off position,while carrying the item, in order to deliver it.
  For readability reason it is better to define the two actions separately. In both cases the preconditions will require to move between adjacent nodes, and the agent to be in the one where the movement starts from.
  During the delivering phase the position of the item will have to be update along with the one of the agent that is carrying it.

```
pre(move(from(X,Y),to(X2,Y2)),S):-
        member(agentpos(X,Y),S),
        member(phase(collecting),S),
        (edge(node(X,Y),node(X2,Y2));
        edge(node(X2,Y2),node(X,Y))).

add_rem(move(from(X,Y),to(X2,Y2)),
                [agentpos(X2,Y2)],
```

```
                    [agentpos(X,Y)]).

pre(movecarry(from(X,Y),to(X2,Y2)),S):-
        member(agentpos(X,Y),S),
        member(itempos(X,Y),S),
        member(phase(dropping),S),
        (edge(node(X,Y),node(X2,Y2));
        edge(node(X2,Y2),node(X,Y))).

add_rem(movecarry(from(X,Y),to(X2,Y2)),
            [agentpos(X2,Y2),itempos(X2,Y2)],
            [agentpos(X,Y),itempos(X,Y)]).
```

- The agent will need to pick up the item once it reaches it. The corresponding action will mark the transition between the **collecting** and the **dropping** phase.
  With the **pre/2** predicate, checking if the agent and the item have the same position will ensure that this action is performed only when it should be.

```
pre(pickup,S):-
        member(agentpos(X,Y),S),
        member(itempos(X,Y),S),
        member(phase(collecting),S).
add_rem(pickup,
            [phase(dropping)],
            [phase(collecting)]).
```

- Finally, an action is required for the agent to drop the item at the designated position. We must check if both the agent and the item are in the right place, and change the phase to **dropped**.

```
pre(drop,S):-
        member(agentpos(X,Y),S),
        member(itempos(X,Y),S),
        member(targetpos(X,Y),S),
```

```
        member(phase(dropping),S).
add_rem(drop,[phase(dropped)],[phase(dropping)]).
```

4. A cost function is required for the search strategy. In this very simple scenario we can consider the distance between two points to be the cost of moving between them.
   Picking up the item and dropping it, however, do not require any actual movement while still needing an associated cost. The structure of the graph implies that the minimum distance between two adjacent positions, being the minimum cost of a moving action, is 1. Since they require no movement, we can assign to the other actions a fixed cost of 0.5, that will make them more desirable.
   In order to distinguish between the different actions in the cost definition, we can use the predicate **actions_from_path/2** from the STRIPS module.
   The predicate calculating the distance is not displayed here, and can naturally be implemented in a number of ways.

```
cost(S0,S1,D):-
        (actions_from_path([S0,S1],[move(_,_)]);
         actions_from_path([S0,S1],[movecarry(_,_)])),
        member(agentpos(X,Y),S0),
        member(agentpos(X2,Y2),S1),
        euclidean_distance(from(X,Y),to(X2,Y2),D),!.

cost(_,_,0.5).
```

5. A heuristic function is also mandatory. We want the agent to prioritize actions that will lead it to the goal, so it should move towards the item at first, then pick it up, then move towards the target position where it will drop the item and reach a final state, ideally with $h = 0$.
   The most widely adopted heuristic in pathfinding problems is a function of the distance between the agent and his target position. In this case either the euclidean distance or the manhattan distance will work.
   In order to properly guide the movement of the agent it is necessary to change its target position once the collecting phase is terminated.
   If the heuristic was left only as a pathfinding guide, the agent would have no idea of the fact that progressing through the three phases (**collecting**,**dropping** and

dropped) is a good thing.
A simple way to make states belonging to a certain phase more desirable than the ones belonging to the previous phase is to displace their heuristic values with a constant.
During the last phase, the value of h is always 0. Therefore if any quantity greater than 0 is added in the calculation of h relative to nodes in the other phases, states with the **phase(dropped** fluent will always be considered closer to the goal than anything else (and they are).
With the same approach we can ensure than the agent will always try to reach the dropping from the retrieving phase, by additionally adding any value greater than 5 to the heuristic value of nodes in the dropping phase, since 5 is the maximum euclidean distance between two nodes of the graph, hence the maximum value of the heuristic function.
Since the heuristic function uses the euclidean distance, a natural identifier for its definition predicate would be **euclidean**.

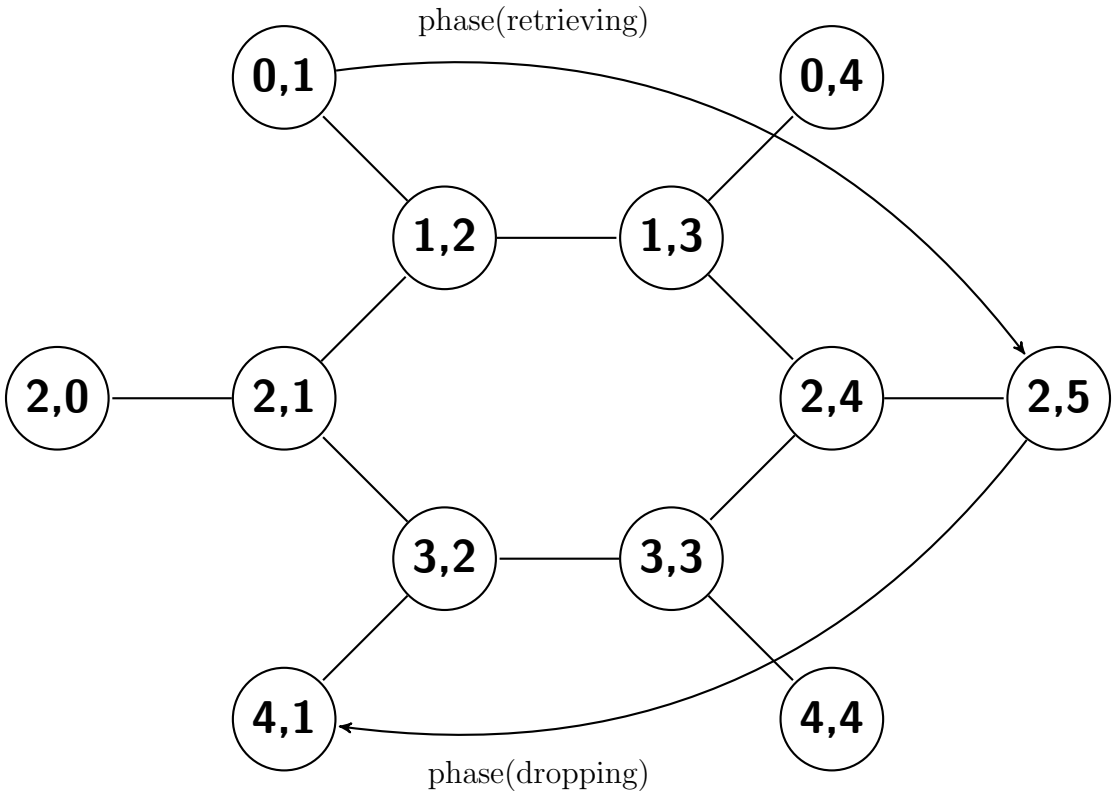
```
heuristic(euclidean,S,H1):-
        member(phase(collecting),S),
        member(agentpos(X,Y),S),
        member(itempos(X2,Y2),S),
        euclidean_distance(from(X,Y),to(X2,Y2),H),
        H1 is H + 10,!.

heuristic(euclidean,S,H1):-
        member(phase(dropping),S),
        member(agentpos(X,Y),S),
        member(targetpos(X2,Y2),S),
        euclidean_distance(from(X,Y),to(X2,Y2),H),
        H1 is H+1,!.

heuristic(euclidean,_,0).
```

Once the problem definition is in place, it is possible to launch the procedure that will attempt to find a solution.
In order to use the **apf:solve_single/5** predicate, it is necessary to decide the initialization parameters that define the specific instance of the problem.
the parameter [**agentpos(0,1),targetpos(4,1),itempos(2,5)**] will result in the initial state [**agentpos(0,1),targetpos(4,1),itempos(2,5),phase(retrieving)**] that

defines an instance of the problem where the agent starts in the node labeled **0,1**, has to move to the node **2,5** where the item is, pick it up and carry it to **4,1**, where it can finally drop it.



Regarding the goal parameter, we implemented the goal function in a way that doesn't consider said parameter in the computation, therefore we can use **goal(_)**. It is also necessary to choose a search strategy: in the case we can use A*, since it is probably the most widely used for this kind of application, with our **euclidean** heuristic.
Since we know that the problem can generate repeated states (if the agent moves back and forward for instance), it is advisable to select some pruning technique. Having a small graph means that the memory overhead produced by the **closed** approach will not be problematic.
Since no additional processing of the return parameters holding the solutions is going to be done, we can discard them with underscores while exporting the solution to the log file where it will be in a more readable format.

```
?- apf:solve_single(
        start([agentpos(0,1),targetpos(4,1),itempos(2,5)]),
        goal([]),
        [s:astar,p:closed,h:euclidean],
        _,_,[log]).

---------- Problem Instance ----------
[agentpos(0,1),targetpos(4,1),itempos(2,5),phase(collecting)]
--------> Search: astar  Prune: closed
Heuristic: euclidean
outputting results to log.pl
true.
```

And as the last entry in the log file:

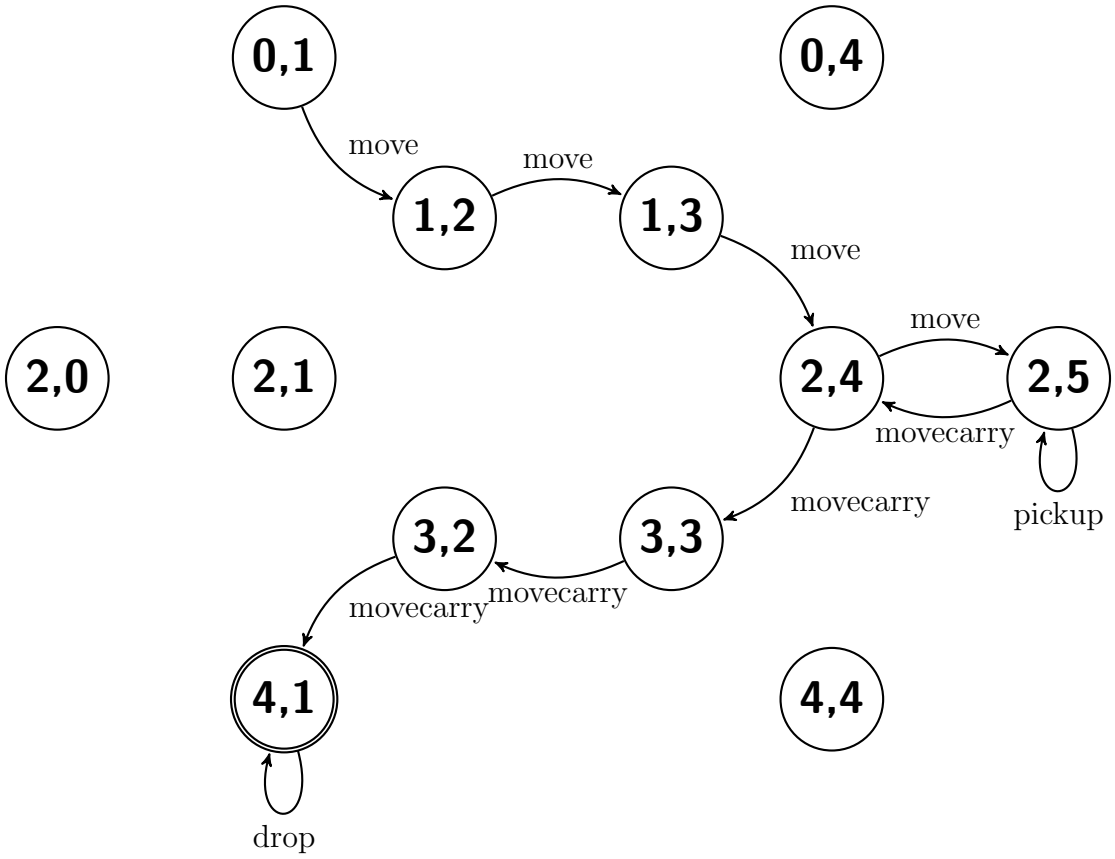
```
%----------------------------DATE: 2016/8/31 TIME: 11:9.
search results:
start parameters: [agentpos(0,1),targetpos(4,1),itempos(2,5)]
goal parameters: []
job: [s:astar,p:cutcycle,h:euclidean]
solutions:
[state:
[phase(dropped),agentpos(4,1),itempos(4,1),targetpos(4,1)],
depth:10, cost:10.65685424949238,
path:
[
[agentpos(0,1),targetpos(4,1),itempos(2,5),phase(collecting)],
[agentpos(1,2),targetpos(4,1),itempos(2,5),phase(collecting)],
[agentpos(1,3),targetpos(4,1),itempos(2,5),phase(collecting)],
[agentpos(2,4),targetpos(4,1),itempos(2,5),phase(collecting)],
[agentpos(2,5),targetpos(4,1),itempos(2,5),phase(collecting)],
[phase(dropping),agentpos(2,5),targetpos(4,1),itempos(2,5)],
[phase(dropping),agentpos(2,4),itempos(2,4),targetpos(4,1)],
[phase(dropping),agentpos(3,3),itempos(3,3),targetpos(4,1)],
[phase(dropping),agentpos(3,2),itempos(3,2),targetpos(4,1)],
[phase(dropping),agentpos(4,1),itempos(4,1),targetpos(4,1)]]
]
statistics: [inFringe:7,maxFringe:8,processed:18,
maxDepth:10,branching:1.335141362540313]
```

We can immediately see that a solution was found, with final state

```
[phase(dropped),agentpos(4,1),itempos(4,1),targetpos(4,1)]
```

This corresponds to the desired situation: the agent has moved the item to the target position, and then dropped it, changing its internal phase status to **dropped**. Analysing the path leading to said final state, we can reconstruct how the agent acted and moved on the graph by comparing **agentpos** fluents.

We have that it moved from its initial position, the node **(0,1)** to the item position **(2,5)**, through **(1,2)**, **(1,3)** and **(2,4)**.
In **(2,5)** it changed its **phase** from **retrieving** to **dropping**, without moving.
Then it moved all the way through **(2,4)**, **(3,3)**, **(3,2)**, to **(4,1)** bringing the item along, before changing state again and dropping it.



In order to more easily analyse the actions that led the agent to solve the problem, it is possible to use the built-in predicate **actions_from_path/2** to extract the list of actions corresponding to the solution path.
The predicate is available since the module **strips** is being used. it could not be used with the **default** action definition language.
The complete path is composed by the final state and the partial path leading to its parent node. For this reason it is necessary to append the final state at the end of the incomplete path before processing it.
The command

```
?- solve_single(
        start([agentpos(0,1),targetpos(4,1),itempos(2,5)]),
        goal([]),
        [s:astar,p:closed,h:euclidean],
        [sol(Final,_,_,PathPartial)]
        ,_),
    append(PathPartial,[Final],PathComplete)
    actions_from_path(PathComplete,Actions).
```

Will result in the term **Action** being the list of actions that the agent performed along the path to the goal state.

```
[move(from(0,1),to(1,2)),
move(from(1,2),to(1,3)),
move(from(1,3),to(2,4)),
move(from(2,4),to(2,5)),
pickup,
movecarry(from(2,5),to(2,4)),
movecarry(from(2,4),to(3,3)),
movecarry(from(3,3),to(3,2)),
movecarry(from(3,2),to(4,1)),
drop]
```

Chapter 4

Goal-oriented Action Planning

4.1 Goal-oriented behaviour

In many real time AI applications, for reasons ranging from implementation simplicity to computational efficiency, agents are implemented simply as mere sets of reaction to given inputs: the input is received and processed, and the appropriate, pre-made response is selected.
Even with such a simple approach, it is possible to make an agent act like it has desires and goals: for instance, a predatory animal can have a very basic scripted input-response map that can produce a behaviour seemingly intelligent as follows

Input	Reaction
hungry	attack nearby prey
no prey nearby	move away
threat nearby	move away
competing creature nearby	attack competing creature

This system can hardly be described as intelligent. Still the agent would exhibit a somewhat organic and natural behaviour, such as that of an animal that desires to survive hunger or threats, and to preserve its territory from intrusions. Clearly no actual desire for survival is implemented.
Techniques can be employed to make virtual characters and intelligent agents less static in their acting, and this can lead towards a higher degree of adaptability, flexibility and complexity in their behaviour.
Simulations of the intelligent behaviour of characters that try to reach and fulfill various aims and goals has always been subject of study in the field of videogame development.
From this area of application the concept of goal-oriented behaviour (GOB) emerged[7],

as a slightly vague definition of AI techniques that explicitly consider goals and desires in the implementation of an agent.
A few different approaches are described in Ian Millington and John Funge's *Artifial Intelligence for Games*, to the implementation of GOB in a character. At first, a decision making system that can decide what to do next based on a set of goals.
The two components that are used by the system are goals and actions:

- Goals: a character may have a number of them, that can be active or inactive at a certain point in time. Each goal has a level of importance, that describes how pressing it is to the character.
  Goals with a high importance, or insistence, level will have a stronger impact on the behaviour of the character, since their fulfillment will be prioritised over the fulfillment of others.
  In some cases, goals can be satisfied completely and they won't matter anymore. Goals regarding activities that have to be performed regularly, such as avoiding hunger, will only see their insistence reduced when some actions are undertaken, like eating, and will remain part of the simulation. In this case, their insistence will be likely to increase again as a result of other events or simply over time.
  A zero value of insistence refers to a goal that has been completely satisfied, but not necessarily to one that will not be pressing again.
  The insistence of goals is usually represented with non negative reals. Booleans can also be used in cases where the number of goals requires a certain simplification of the simulation.

- Actions: they define what the agent can do, and they have an impact on its goals.
  Not all of them will be available, or active, at any given time. The activation or deactivation of actions can be the result of other actions: an animal will not be able to perform the "eat" action unless it hunts a prey first. Or it can be a consequence of events that don't depend on the character: a prey wandering into the territory of the animal will trigger the activation of the action "hunt".
  Active actions compose a set from which the character can choose, evaluating the effect of each action against its own goals. A single action can have both positive and negative effects on any number of goals: for instance, in the case of the animal, the action of resting would have the beneficial effect of reducing tiredness at the expense of an increase in the need for food.
  Since actions can in fact activate other action, the beneficial effect on a goal could be a certain number of individual actions away: the positive effect of eating a prey can only be achieved after finding it and killing it. This can also be obtained by defining an action as a sequence of other actions: hunting could

be composed of finding, chasing and striking a prey.

A first implementation of GOB would involve a simple selection mechanism to choose the next action. the underlying data structure on which the selection is performed could be a list of actions, each one coupled with the quantitative effects (positive or negative) on the insistence value of the goal involved.
For example (a negative change value represents a positive effect, decreasing the insistence of the corresponding goal):

Action	Goal	Effect
eat	avoid_hunger	-5
drink	avoid_thirst	-5

On the basis of a table like this, a selection is then made to choose what action is to be performed next.
One way is to find the most pressing goal, being the one with the highest insistence value, and then pick the action which has the strongest positive effect on it.
This approach just involves finding the maximum (or minimum) value in two sets: resulting in a time complexity of $O(n)$ for the operation that finds the most pressing goal, where n is the number of goals, and $O(m)$, where m is the number of actions, under the assumption that finding the effect of an action is a constant time operation. The total time complexity is then $O(n + m)$ with a space complexity of $O(1)$, since the only memory required is for the storage of temporary maximum(or minimum) values.
This system, although very cheap and simple to implement, manages to be quite fast and reliable. Its main drawback is that it can not take into consideration the side effects that an action might have on other goals.
Also, this approach doesn't plan anything in advance, since it only considers the effect of an action on the goals and not the fact that it could activate or deactivate other actions. For this reason it fails to work with chains of individually separated actions that might offer great reward at the end, but present very little, or no appeal at the beginning: the action of chasing a prey has no real benefit in itself and would therefore be never considered, even if it would subsequently allow the animal to perform the highly beneficial action action of eating the prey.
A slightly more advanced approach can take account of the side effects that actions have on other goals: chasing a prey will make the character tired, drinking from a pond could lead to an attack by other animals, or sleeping will decrease the need for rest but will probably increase hunger.
This requires a change in the basic action structure:

Action	Effect
hunt	rest: +3
eat	avoid_hunger: -5
drink	avoid_thirst: -5, be_safe: +4
sleep	rest: -5, avoid_hunger: +3

In order to take account of the multiple effects of an action, it is possible to encode all the insistence levels into a single value representing the discontentment of the character. The agent should then try to reduce the total discontentment rather than focusing on the single goals.
Simply adding the various insistence values together tends to produce rather unnatural behaviors (see section 4.4), since the agent will not necessarily try to tackle extremely pressing issues if other actions have a greater quantitative effects on other goals, even if these are not pressing at all.
A character that has drunk recently but is starving should prefer an action that reduces by 1 its need of food, over one that lowers its thirst by 2. In this case it would not, since the greater positive effect on the total discontentment would make it choose the second action, even if that leads to death by starvation.
More natural results are obtained by squaring the insistence values before adding them, this giving more importance to the goals that have a higher insistence over the satisfied, or nearly satisfied, ones.
A more flexible approach would involve giving different weights to different goals: the goal of staying alive could for example have a greater impact on the total discontentment by being cubed instead of squared, or increased by a constant before being squared. The relative importance of goals could even change on the basis of events that are part of the simulated environment: the appearance of a threat could increase the importance of the goal of being safe, or the coming of the dry season could give more weight to thirst.
The agent would then consider all actions, and choose the one that produces the lowest discontentment value.
Looking at the actual implementation, this approach would require finding the minimum discontentment resulting from every possible action. This, having n goals and m actions, would result in a time complexity of $O(n * m)$. The increase in computational costs is due to the fact that the calculation of the discontentment value has to be done for all goals and for all actions.
Something that is not taken into account up to this point is the notion of time: carrying out a certain action obviously takes some time, and considering this in the selection process would produce a more sensible behaviour.
One simple way is to use time as a secondary parameter with which to select the next action: in cases where multiple actions have the same effect on the discontentment, the one that requires less time is chosen.

A more sophisticated approach would involve incorporating the passing of time into the effect of actions on the insistence values: for instance, the need for food will increase at an approximately constant rate during the day, on top of the increase deriving from physical activity or energy recovery. The same can be said for goals such as rest or drink or the need for entertainment or social activity.
Giving the rate of change of a goal insistence over time, and given the time required to carry out an action, it is possible to incorporate in the calculation of the discontentment value the effect that the passing of time will have on that goal.
This calculation requires additional data structures to store the information about insistence changes over time. as the example below shows, it is possible to have goals whose insistence decreases over time: in this case an animal will feel the need to hide or run away only if a threat is detected, and this need will fade away over time when no such thing happens.

Goal	Change/Hour
eat	+1
avoid_threats	-2
drink	+1

Calculating the discontentment would require multiplying the duration in time of the action by the change over time of the goals involved, and add the result to the insistence values. Then the usual total discontentment calculation can take place.
This addition would not change the asymptotic complexity neither in space or time, since the only different from the previous approach was the constant time operation of multiplying the time an action requires by the passive change in insistence of the goals involved.

4.2 Goal-oriented Action Planning

All the previously described selection techniques can produce a sensible, sufficiently natural behaviour in most cases, but they share a common weakness: they don't take into consideration the fact that actions could very well enable or disable several others.
By only considering one step at a time the agent would then miss the opportunity of gaining high rewards if a previous non rewarding action is necessary.
Similarly the agent would not be able to avoid extremely problematic or even fatal situation if an apparently rewarding action leads to them: a piece of food at the bottom of a pit would in fact trap a hungry animal that reasons this way.

To have characters capable of taking advantage of sequences of actions and avoiding traps, some degree of forward planning must be introduced.
Goal-Oriented Action Planning (GOAP) allows the agent to plan in advance in order to reach the optimal fulfillment of its goals, making it capable of considering the effects of multiple actions in sequence.
To support this activity, it is necessary to take into consideration all the effects of an action, not only the change in the agent's goals, but also the effects on other actions.
GOAP can be implemented as a search problem: nodes of the graph represent states of the world, including information about the goals of the agent and the availability of actions. The problem of finding a suitable sequence of actions that satisfies the goals is then the problem of finding a path to a state where the goals are sufficiently satisfied.
If not restrained, a planning algorithm would never terminate and would continue generating infinite sequences of actions in exponential quantity. For this reason a limitation in the depth of the search tree used for the planning is usually implemented. However, any system that prevent the algorithm from producing an infinite tree is suitable. An example of a different approach is given in section 4.4.

4.3 A GOAP model

The tool was used to implement a model of GOAP, and to experiment with the problem definition.
The model simulates the behaviour of a virtual person, that should act on the basis of its current goals. Goals are treated as needs: need for food, for using the bathroom, for rest and for entertainment.
The agent will try to fulfill its needs, lowering their insistence, by the means of a set of given actions. actions can affect positively and negatively any number of needs, and can also enable or disable any number of other actions.
The needs are defined in the range [0-24]. Any insistence value that would go below 0 is set to 0, and any value greater than 24 is set to 24.
All needs passively increase by 1 unit per hour of the simulation, this change over time is coded in the effects that actions have depending on their duration: an action that lasts 2 hours will, on top of any beneficial effect, increase all needs by 2.
Actions that last less than one hour have a negative effect that doesn't follow any specific scheme, and is left to what seems the most natural value.
The focus of this implementation is, rather than being a high-quality simulation, experimenting on how the definitions of different parameters of the search affect the resulting behaviour of the agent.
In order to properly observe the behaviour of the agent and at the same time prevent

the algorithm from not terminating, the *goal* function was made so that the simulation lasts a certain amount of virtual time, rather than having a depth limit that would effectively set the amount of actions that can be planned in advance, or having a target discontentment value that would stop the simulation as soon as the agent is sufficiently satisfied.
Having the simulation continue for a given amount of time allows the observation of the agent's behaviour without any restriction other than computational resources (time and space).
The **default** module was chosen for the problem definition for two reasons: it allows for a more compact code with a single action definition predicate, which in turn makes adding new actions quicker and easier, at the same time the educational benefits of the other module were not really important in this case.
Not having the restrictions of the STRIPS standard, a state was defined as a list holding, at fixed positions, the different values that take part in defining a specific situation.
Any state of the problem takes then the form
[**LastAction,Discontentment,Needs,InactiveActions,Time**], where:

- **LastAction** identifies the action whose result is the current state. It is used for readability reasons since the predicate **actions_from_path/2** is not available with the **default** module.
- **Discontentment** is the overall discontentment value resulting from the current m needs: it is calculated as $\sum_{i=0}^{m} x_i^2$, where $x_0, ..., x_i, ..., x_m$ are the insistence values relative to the needs of the agent.
- **Needs** is a list which holds the insistence value of every need. the position of a given need is fixed.
- **InactiveActions** is the list that holds all the actions that are currently not available. This approach is equivalent to having a list that holds all the currently active ones, but was preferred because in the model most actions remain always active, and this results in a shorter list.
- **Time** is an integer value that represents the time in minutes since the beginning of the simulation. It is necessary in order to stop the search process once the time limit is reached.

Each action was defined through a specific predicate **action(Name,Time,Effects,Activates,Deactivates)**, where:

- **Name** is the identifier.
- **Time** is the time required to carry out the action.
- **Effects** is a list of values that represent the effect of the action on the various insistence values. A need that is not affected by the action will require a value of 0 at its corresponding index in the list. The resulting insistence values are defined and calculated by adding the values of the insistence values at the previous state to the effects of the action.
- **Activates** is a list that holds the identifiers of all the actions that are activated by this one.
- **Deactivates** holds the identifiers of all the actions that are deactivated.

For example, the action of eating a snack will be defined as:

```
action(eat_snack,15,[-1,1,1,0],[],[]).
```

which means that the action will take 15 minutes to be completed, and will have a small beneficial effect to the insistence of the need for food (position 0 in the **Effects** list), while having a small negative effect on both the need for the bathroom and the need for rest (positions 1 and 2, respectively) resulting from a passive change overtime, and no effect on the need for entertainment.
This action will not enable or disable any other.
With a set of such predicates, a set of 13 simple actions is described from which the agent will be able to choose. With this implementation, adding a new action just requires writing a new **action/5**.

action	time	effects	enables	disables
start	0	0,0,0,0		
sleep_bed	480	8,8,-20,8		
sleep_couch	60	1,1,-2,1		
eat_snack	15	-1,1,1,1		
eat_cooked	30	-10,8,1,1		eat_cooked
cook	30	1,1,1,4	eat_cooked	cook
do_dishes	15	0,0,0,2	cook	
shower	15	1,1,-1,0	sleep_bed	
bathroom	15	1,-16,0,1		sleep_bed
watch_TV	60	1,1,0,-6		
read_book	60	1,1,-3,-6		
choose_book	5	0,0,0,1	read_book	
gardening	60	1,1,2,-8		sleep_bed

These actions are specifically implemented to test how the agent is capable of planning in advance: the are multiple actions that provide a benefit to the same need, but while one is always available, the other has to be enabled by something else and is more effective.
There are a few actions that have the sole purpose of enabling others: **cook**, **choose_book**, **shower** only have small negative effects on the overall discontentment but enable highly rewarding actions such as **eat_cooked**, **read_book**, **sleep_bed**. **do_dishes** enables a second **cook-eat_cooked** sequence after a first one has been performed.
Once the support structure for the actions has been defined, the actual agent can be designed through the necessary predicates.
The predicate **start/2** simply assembles the initial state with the parameters **Needs** and **InactiveActions** provided by the user. The initial discontentment value is calculated and the time parameter is set to 0.

```
start([Needs,InactiveActions],[start,D,Needs,InactiveActions,0]):-
        calculate_discontentment(Needs,D).
```

goal/2 just checks if the time limit has been reached.This will stop the search once a sequence of actions reaches the end of the simulated time.
This means that the *goal* function will not have any relationship with the the general problem the agent is working on, being the satisfaction of its needs. This is in contrast

with what is the usual definition of a *goal* function (see chapter 1), that should be true only when the purpose of the agent is fulfilled.
However, this approach was necessary to implement the time limit without modifying the search procedure

```
goal(P,[_,_,_,_,T]):-
        T>=P.
```

Since the **default** action language is in use, the definition of actions is carried out through the means of the sole predicate **act(S0,S1)**, which returns as **S1** all the actions available at the state **S0**.
In this case the predicate should check whether an action is actually enabled, before computing its effects on the current state: changing the agent's needs, producing a new discontentment value, enabling/disabling actions and increasing the time counter by the appropriate amount.
In order to achieve this with **act/2**, a choicepoint is generated that unifies with all actions: after either succeeding in generating a new state or failing due to the action being disabled, the predicate will backtrack to the next action.

```
act([_,_,Needs1,Inactive1,T1],[ActionName,Discontentment,Needs2,Inac
        action(ActionName, Time, Effects, ToActivate, ToDeactivate),
        not(member(ActionName, Inactive1)),
        apply(Needs1,Effects,Needs2),
        calculate_discontentment(Needs2,Discontentment),
        append(Inactive1,ToDeactivate, In0),
        sort(In0,In1),
        subtract(In1,ToActivate,Inactive2),
        T2 is T1+Time.
```

apply/3 is an auxiliary predicate that applies the effects of an action to the list of needs of the agent, generating the needs relative to the next state.
The heuristic function and the cost function were subjects of experimentation, with the purpose of exploring the relationships between their definition and the behaviour of the agent, as well as their effects on the overall computation process.

This process of experimentation is explained in the next section.

4.4 Experimentation

The main parameter used to evaluate results is a weighted average discontentment value, measuring the discontentment of the agent weighted on its duration in time. The formula is based on the assumption that the effects of an action take effect only when the action was completed, and in a sudden and complete way.
For a sequence of n actions, where d_i is the discontentment at the step i, and t_i is the duration in time of the state i, being the duration in time of the action that was applied to i to produce $i+1$ (0 for the final state):
$weighted\_avg = \frac{\sum_{i=0}^{n} d_i t_i}{\sum_{i=0}^{n} t_i}$ Other parameters used are the maximum/minimum discontentment value reached during the simulation and the rate of discontentment change, measured as $\frac{d_0 - d_n}{\sum_{i=0}^{n} t_i}$. For the evaluation of computational statistics the duration in time of the computation was considered, as measured by the ProLog built-in predicate **time/1**, along with the total number of nodes expanded, measured by the tool. The two parameters are evaluated together because they are certainly related but not necessarily proportional: a very precise yet time intensive heuristic function would decrease substantially the amount of nodes processed by the search procedure, but could ultimately cause an increase in the time required if the overhead it introduces is great enough. The same could happen with an expensive pruning technique as explained in section 1.5.
The same model was used through all the experimentation process. Two different instance of the problem were chosen to be the initial situations the agent finds itself in.
The two scenarios try to describe the state of a person in the morning after waking up, and in the evening after returning home from work:

	need:food	need:bathroom	need:rest	need:entertainment
scenario 1	14	16	6	6
scenario 2	12	6	12	15

Simulations were made with a time limit of 6 hours (360 minutes), which can be handled by all the tested search strategies without exceeding the maximum stack size, while remaining long enough to provide sufficient observable choices.

First of all, a test was devised to observe how the agent would work without any planning, with a simple GOB approach.
GOB was implemented by using the predicate **act/2** to obtain all the possible successive states, then selecting the one with the lowest discontentment value and repeating until the time limit was reached.
These are the results with the scenario 1. The "last action" field displays the action that brought to that state, while the "time" field shows the time in minutes since the beginning of the simulation.

needs	discontentment	last action	time
14,16,6,6	524	start	0
15,0,6,7	310	use_bathroom	15
16,1,6,1	294	watch_tv	75
15,2,7,2	282	eat_snack	90
14,3,8,3	278	eat_snack	105
13,4,9,4	282	eat_snack	120
13,4,9,5	291	choose_book	125
14,5,6,0	257	read_book	185
13,6,7,1	255	eat_snack	200
14,0,7,2	247	use_bathroom	215
15,1,4,0	242	read_book	275
14,2,5,1	226	eat_snack	290
13,3,6,2	218	eat_snack	305
12,4,7,3	218	eat_snack	320
13,5,4,0	218	read_book	380

Analising the result, one can observe that the agent decided to have a snack 7 times in 6 hours to reduce its hunger. This shows how the inability to plan in advance can produce unnatural behaviours. in this case the agent was not able to eat cooked food, that would have been much more efficient in its effect on hunger, because the action of cooking has, in itself, no beneficial effects.
However, the agent managed to slowly reduce its discontentment, mainly through entertaining activities (the need for entertainment was reduced from 6 to 0), while keeping hunger at bay with frequent snacks (hunger was only decreased by 3 units over the course of the entire simulation). The most beneficial move in the first one, with which the agent decreases its discontentment value by 214.
From an initial discontentment of 524, in 380 minutes, a discontentment of 218 was reached, with a rate of improvement of 48 per hour, and an average value of 248.
The scenario 2 has a starting discontentment value of 549, with three pressing needs: food (12), rest (12) and entertainment (15).

Over 410 minutes, the discontentment was reduced to 374, with an average of 419 and a rate of 25.6/h.
Observing the needs, the main benefit came from the complete satisfaction of the need for entertainment (15 to 0). The need for rest, however, not only was not reduced, but actually increased by a significant amount (12 to 17).
This was caused by using the bathroom at the beginning of the simulation, therefore disabling the **sleep_bed** action. Without being capable of planning, the agent was unable to sleep effectively and could only rely to the relatively inefficient action of sleeping on the couch for one hour at a time, action that was performed twice.
In general, we can observe that this implementation produced an unacceptably unnatural behaviour: in the "morning" scenario, we have an agent continuously consuming large quantities of snacks without really being able to satisfy its hunger, in the "evening" scenario we obtain a character that will never be able to go to bed and is becoming increasingly tired.
Designing a GOAP implementation requires the selection of a suitable search strategy, starting with heuristic and cost definitions.
Given that the agent is trying to minimise its discontentment level D, the heuristic function should direct towards the lowest possible value for it: since we have that $D = \sum_{i=0}^{4} x_i^2$, where x_i is an insistence value in the range $[0, 24]$, the lowest possible value is $\sum_{i=0}^{4} 0 = 0$.
This means that the heuristic function on the node n can be as simple as an extremely natural $h(n) = D(n)$.
The cost function offers more variety in its implementation: It is possible to incorporate the length in time that the action would require, or take the natural choice of using the resulting discontentment similarly to the heuristic function, or even put aside the usual conception of using the sum of the individual costs of all edges from the root to the node in question, using the maximum or minimum discontentment value instead.
Choosing the right search strategy is of primary importance. All the available strategies were analised in order to find the most suitable one:

- Best-First search was discarded because it would have similar results as the already observed GOB without planning. Although computationally cheap, it would not allow for actual long-term planning.

- Iterative Deepening was also discarded, because it is guaranteed to find the solution with the lowest depth: in this case the depth of a solution has no correlation with how good it is and such search would simply find the shortest

possible sequence of actions with a total duration greater than 360 minutes.
When tested, the agent decides to immediately go to bed (duration:480 minutes) without any regard to its current state.

- Branch and Bound's efficiency depends on how good the bounding system is. Given the inherent unpredictability of the discontentment value, no bound calculation can be effectively implemented and the algorithm would end up exploring the whole space.
  During testing, the bounding was therefore done on th cost of nodes, losing the ability to plan in advance, but discarding a large amount of non desirable nodes.
  The result is similar to what was produced by Best First search, but with a great increase in efficiency.

IDA* and A* remained as candidates: the memory requirements of A* pose problems in the case of long simulations that would be resolved by IDA*, that is more space-efficient and would therefore allow for the simulation of longer periods of time. However, IDA* is slower than A* and shares the issue encountered with Iterative Deepening: if the limit is too short, it will find shallow solutions in a context where the depth of a solution has no relationship with its quality.
In order to properly test the effects of A* and IDA*, the pruning technique remains to be defined. During the testing, computational time values are the average of 10 computations on the same machine:

strategy	scenario	pruning	time(seconds)	nodes expanded
IDA*	scenario1	none	180+	
IDA*	scenario1	cutcycle	180+	
IDA*	scenario1	closed	180+	
IDA*	scenario2	none	129	
IDA*	scenario2	cutcycle	152	
IDA*	scenario2	closed	131	
A*	scenario1	none	35	115005
A*	scenario1	cutcycle	49	115005
A*	scenario1	closed	144	98001
A*	scenario2	none	1.6	6951
A*	scenario2	cutcycle	2.5	6951
A*	scenario2	closed	3.3	6932

From this testing it emerges clearly that the increased space efficiency of IDA* is not

sufficient to produce a corresponding increase in time efficiency, instead the computation results much longer than the one with A*, to the point where it had to be stopped after 3 minutes without termination.
discarded IDA*, more attention was given to how pruning interacts with the search with A*:it emerges that in fact pruning is not really effective: the **cutcycle** technique even fails to prune any node at all.
The reason for this resides in the structure of nodes: every state carries a time value, that increases along the path. This means that under no circumstances a node will be produced that is the same as a node already present along its path, and therefore no nodes will ever be pruned.
The **closed** pruning technique actually manages to reduce the number of nodes, but this reduction is so small that the resulting computation is significantly slower than without any pruning.
Again this is an effect of the node structure: the array of needs has 4 elements in a range [0-24], this means that the odds of having two nodes where it is the same are extremely small.
Even if two nodes had the same list of needs, the increasing time value would probably make them different, and even if that was the same they could be made different by the "last action" parameter (necessary to the evaluation of the quality of a solution) if their paths had just converged.
An extreme drop in both the duration of the computation and the number of nodes expanded can be noted when the agent starts in the scenario number 2: analysing the solution found by that search it is clear that something is not working properly. Here the result, printed in readable format:

```
search results:
start parameters: scenario2
goal parameters: 360
job: [s:astar,p:none,h:h(1)]
solutions:
state:
[sleep_bed,1125,[20,14,0,23],[eat_cooked,read_book],480],
        depth:1, cost:1125,
path:[
        start,549,[12,6,12,15],[eat_cooked,read_book],0]
]
statistics:
        [inFringe:6246,maxFringe:6247,processed:6951,
```

```
maxDepth:6, branching:4.368589270164376]
```

The agent started from a situation where its needs for food and rest were very high (12 and 12, respectively), and its need for entertainment was even higher (15), for a total discontentment of 549.
The result would suggest that the best way to reduce the discontentment value in 6 hours would be for the character to immediately go to bed, effectively doubling it from 549 to 1125.
This can be explained by observing the computational statistics: the fringe had a maximum size of 6247, and a size of 6246 when the process terminated. This means that at that moment the fringe contained 6247 nodes, the first of which had a $f(n)$ value of $h(n) + c(n) = D(n) + c(n) = 11251 + 1252 = 2250$, while all the others had even higher f values. The first node was expanded and since the action of sleeping in the bed is longer than 6 hours, it immediately satisfied the goal function and was returned as a solution.
The very high D value of this solution is mainly caused by the needs for food and entertainment (20 and 23). Higher D values would require an even greater insistence for those needs.
The action that was finally selected has a strong negative effect on the needs of the character. In order to find a greater impact it is necessary to construct a chain of multiple actions that consistently ignore or even increase at least one need to the point where it grows out of control.
This is exactly like what was happening when GOB was tested: the agent lost control of its need for food, which was eventually going to grow indefinitely causing increasing amounts of discontentment. In this case, the agent was still unable to understand that cooking would have had a great positive impact after the initial negative effect.

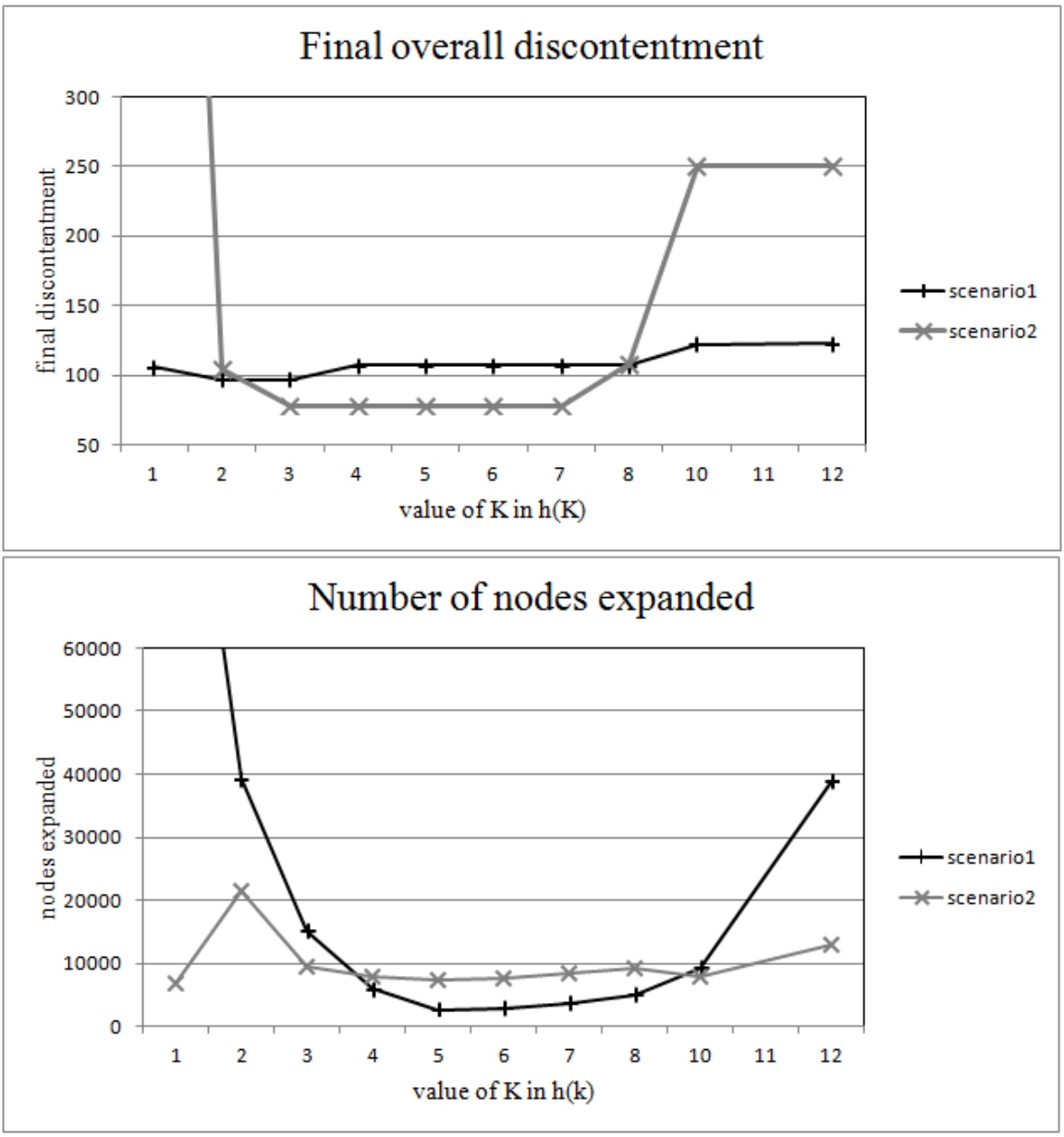
```
action(cook,30,[1,1,1,4],[eat_cooked],[cook]).
```

The negative impact of the **cook** action is mainly on the need for entertainment. In this scenario the initial needs of the character are **[12,6,12,15]**, and the need for entertainment has the highest insistence of all.
For this reason the action would have had an even greater impact on the total discontentment and the agent never attempted to perform it. Instead, it explored a number

of sequences that never involved cooking. Sequences that produced higher and higher levels of hunger with the corresponding extreme discontentment, to the point where simply going to bed promised to be better than performing long chains of actions that produced worse effects over a shorter period of time.
In order to prevent this from happening, it is necessary to reduce the impact of the discontentment on the short term, if a beneficial action can be performed later.
The algorithm uses the evaluation function $t(n) = h(n) + c(n)$, and selects for expansion the node with the lowest $t(n)$. In order to have an agent that "cares less" if an action has a negative effect, as long as it offers a greater reward later, it is possible to lower the relative impact of $c(n)$ in favor of $h(n)$.
To do this, the heuristic function $h(n) = D(n)$ was changed into $h(n) = d(n) * K$: operating on the K coefficient it is possible to increase or decrease the impact of the heuristic in the evaluation functions, that would therefore become $t(n) = D(n) * K + c(n)$.
The testing of multiple K values was done by making the heuristic definition parametric:

```
heuristic(discontentment(K),[_,D,_,_,_],D2):-
        D2=D*K.
```

This way K can be set from the **apt:solve_single/5** predicate, by having **h:h(K)** in the job definition, to request the use of the heuristic **discontentment** with the parameter **K**.
The testing was done using **apt:solve_multiple/5**, with a list of job specifications containing the parameter **h(K)**. the results were observed with particular attention to the final discontentment reached, and the number of nodes expanded:



As shown by the charts, the two scenarios share a somewhat common trend for both parameters, excluding the simulation of the scenario number 2 with $K = 1$, which produced a solution of extreme low quality in a very short time due to the particular interaction between the action of cooking food and the scenario setting.
Regarding the final D value reached, both scenarios show a noticeable decrease down to a minimum for $K = 3$, where the best solutions were found (97 for the first scenario, 79 for the second). The values then slowly increase following the increase of K: this

is caused by the decreasing efficiency of the selection process, in which the cost value progressively loses weight in favour of the heuristic. This causes the degradation of the overall search strategy towards a simpler Best First (that would only select nodes on the basis of the heuristic function, with no associated cost).
In fact, the results produced by A* and Best First eventually converge if A* is used with a sufficiently large K: Best First, applied to the scenario number 1, produces a solution with a final discontentment of 210. A* returns the same solution with $K = 27$, and keeps returning it with $K = 10^2, 10^3, 10^4, 10^5$).
Similarly, in the case of the scenario number 2, the results converge for $K \geq 166$.
As shown by the chart displaying the number of nodes processed, the two scenarios share again a similar trend: in fact they both have their overall low point at $K = 5$, after which they produce a slow but continuous increase, as the heuristic loses efficiency.
As a consequence of these tests, $h(3)$ was chosen as an improved version of the heuristic, since it appears to produce qualitatively superior results with a noticeable increase in computational speed: slightly higher values showed to be more effective in reducing the number of nodes processed, but produced slightly inferior solutions.
Lastly an attempt was made to improve the cost definition, by testing two other approaches, that involved using as cost function the maximum and the minimum discontentment reached during the simulation.
The maximum (or minimum) value has to be updated during the expansion of a node, and has to be stored in the definition of each state. this operation is carried out inside the **act/2** predicate:

```
(Discontentment>OldMax->
      NewMax is Discontentment;
      newMax is OldMax)
```

A* selects nodes on the basis of the function ($t(n) = h(n) + cost(n)$. If the predicate **cost/3** was designed to return the maximum discontentment value encountered, $cost(n)$, as used by the $t(n)$ function, would actually be the sum of the maximum discontentment recorded at every node, because $cost(n)$ is in fact the cost of the path from the root to n, and therefore is equal to the sum of the individual costs of all edges.
To overcome this, the cost returned by the predicate **cost/3** has to be set to 0, so that the sum of all edge costs is 0, and the maximum (or minimum) discontentment is added to the heuristic function, so that $t(n) = h(n) + cost(n)$ becomes

$t(n) = h(n) + maxD(n) + 0.$

```
cost(_,_,0).

heuristic(h(K),[_,D,_,_,_,Max],H):-
        DK=D*K,
        H is DK+Max.
```

A* with the new cost definition (implemented with the new heuristic **hMax(K)**) produces the same results as Best First with **h(K)** with both scenarios, as shown (for scenario1) by

```
apt:solve_single(start(scenario1),
                         goal([360]),
                        [s:astar,p:none,h:hMax(3)],
                        [sol(S,_,_,_)],_),
apt:solve_single(start(scenario1),
                         goal([360]),
                        [s:bestfirst,p:none,h:h(3)],
                        [sol(S,_,_,_)],_).

---------- Problem Instance ----------
[start,524,[14,16,6,6],[eat_cooked,read_book],0,524]
---------> Search:astar Prune:none Heuristic:hMax(3)
---------- Problem Instance ----------
[start,524,[14,16,6,6],[eat_cooked,read_book],0,524]
---------> Search:bestfirst Prune:none Heuristic:h(3)

S=[read_book, 210,[13, 5, 4, 0],[eat_cooked, sleep_bed],380,524].
```

This is a consequence of the fact that, with this model, once the agent uses an appropriate heuristic function, it can generally decrease its discontentment from the initial value.

This means that the maximum discontentment encountered is always the initial one, and is never updated. This causes a constant $maxD(n) = M$ value for every node, making the cost evaluation completely useless.
The agent then uses the heuristic alone to select the next node to be expanded, exactly like what happens with Best First.
The same happened when using the minimum D value: the agent is again completely incapable of choosing an action that will increase D, even if it allows for greater rewards on the long term.
Both these two additional ways of calculating the cost function proved to be ineffective.
In conclusion the best strategy found used A*, with no pruning, the heuristic $h(3)$ which is defined as $D(n)*3$, and the **cost(n1,n2,C)** predicate, defined as $C = D(n_2)$.
This results in these solutions, for both scenarios:

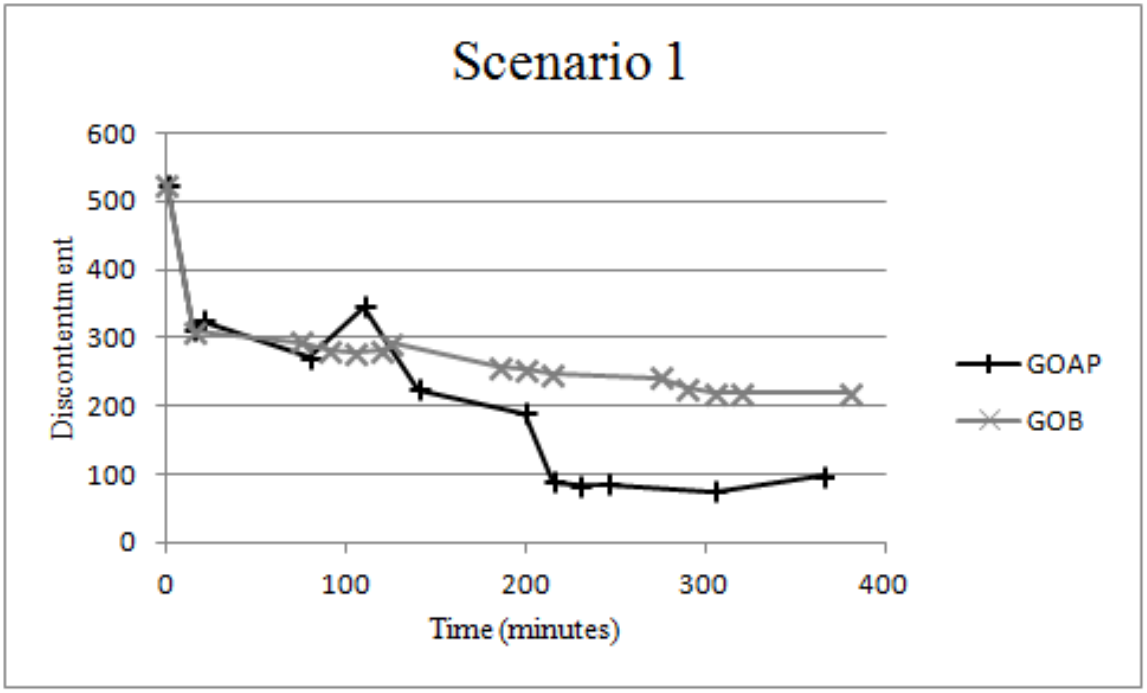
scenario1

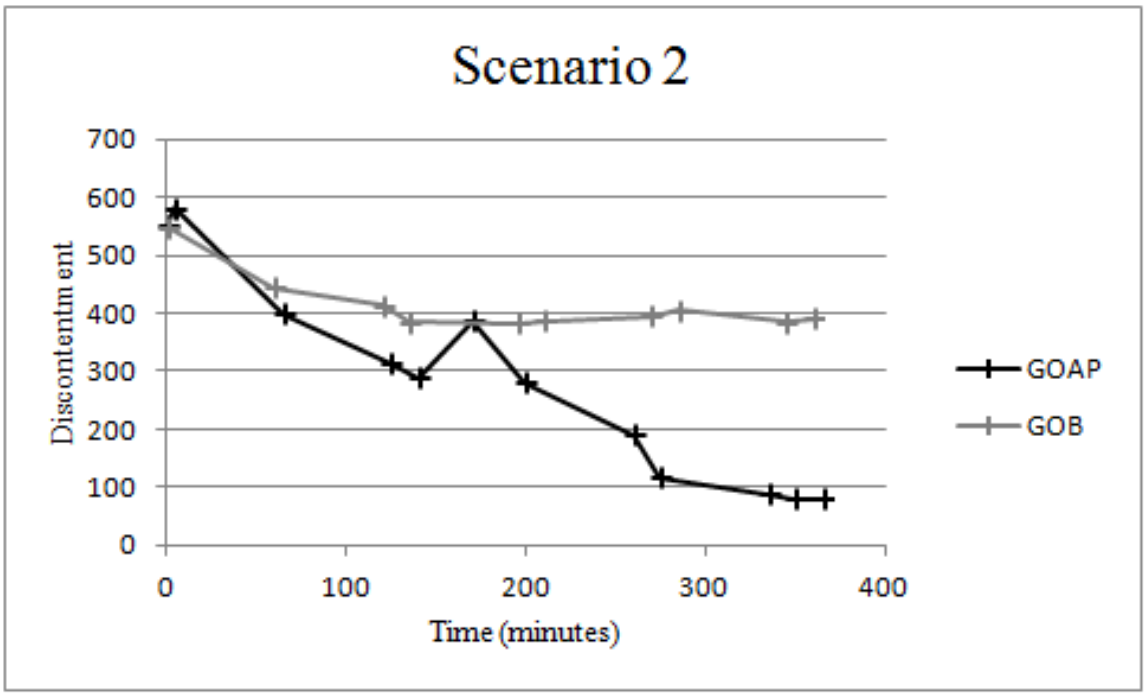
last action	discontentment
start	524
bathroom	310
choose_book	325
read_book	270
cook	345
eat_cooked	223
read_book	190
bathroom	89
eat_snack	83
eat_snack	85
read_book	74
read_book	97

scenario2

last action	discontentment
start	549
choose_book	580
read_book	399
read_book	312
bathroom	286
cook	387
eat_cooked	281
read_book	190
bathroom	114
read_book	86
eat_snack	78
eat_snack	78

This, from the starting point of GOB, produced a great improvement in both the empirical "naturalness" of the solution and the quantitative quality as expressed by the metrics. As well as a decisive decrease in computational time if compared to the earlier implementations using $h(1)$.
These charts provides a clear graphical comparison between GOB and this final implementation of GOAP, showing how the agent is much more effective in reducing its discontentment value overtime.
It is also shown how the ability of accepting a discontentment increase on the short term can cause a stronger decrease on the long term.


Scenario 1
Discontentm ent
600
500
400
300
200
100
0
0
100
200
300
400
Time (minutes)
GOAP
GOB


Scenario 2
Discontentm ent
700
600
500
400
300
200
100
0
0
100
200
300
400
Time (minutes)
GOAP
GOB

Bibliography

[1] Henry Farreny: *Completeness and Admissibility for General Heuristic Search Algorithms.*
Journal of Heurstics, Boston, 1999.

[2] Stuart Russel, Peter Norvig: *Artificial intelligence a modern approach.*
Prentice Hall, Englewood Cliffs, New Jersey 07632, 1995.

[3] Rina Dechter, Judea Perl: *Generalized best-first search strategies and the optimality of A.*
Journal of the ACM (JACM) , 1985.

[4] Tommaso Ceresini: *A unified framework for search and optimization algorithms.*
Università degli studi di Milano, Milano, 2015.

[5] Richard Fikes, Nils Nilsson: *STRIPS: A new approach to the application of theorem proving to problem solving, in Artificial Intelligence.*
Stanford Research Institute, 1971.

[6] A. H. Land, A. G. Doig:*An automatic method of solving discrete programming problems.*
Econometrica, 1960.

[7] Ian Millington, John Funge:*Artificial Intelligence for Games - 2nd edition.*
CRC Press, 2009