Foundations of Artificial Intelligence

Niccoló Didoni

September 2021

Contents

Part I

Introduction to Artificial Intelligence

Chapter 1

Intelligence

Artificial intelligence is a discipline between science and engineering that tries to build an intelligent system. This definition is far from complete in fact it doesn't specifies what an intelligent system is. AI researchers have formulated four types of intelligence systems.

systems that **think** as a **human**	systems that **think rationally**
systems that **act** as a **human**	systems that **act rationally**

Table 1.1: Definitions of artificial intelligence

All the definition given in Table 1.1 are reasonable definitions. Each definition is different (even if it seems not) and brings different problems when designing of an AI.

1.1 Systems thinking as humans

To develop systems that think as humans we first have to understand how humans think. In other words we should understand and model the internal behaviour of our brain and of our reasoning process and try to emulate such behaviour. The model that resembles our thinking process is called **cognitive model**.

For instance in the sixties researchers thought that backward planning (i.e. start from the goal and work backwards) was our reasoning model and they applied it to machines.

1.2 Systems acting as humans

The second definition replaces the thinking process with the acting process. In this case the focus switches from the mechanism used to think to the outer performance of the system. In other words we aren't interested anymore in how the system thinks but we only care about what the system does, without caring how internally the system decided what to do.

To put it in another way the system is intelligent because what it does is intelligent (i.e. it performs some actions in a predetermined way, which is considered intelligent). Namely if one system is showing some performance and we have no idea on how it is thinking but we can confront it's behaviour with a set of characteristics that define a system as intelligent, then the system can be defined as intelligent.

1.2.1 The imitation game

Alan Turing came up with a test, known as the imitation game (evolved in the **Turing test**), to verify if a system behaves (performs) in an intelligent way. The test consists of two separated parts.

Part 1

In the first part of the test an interrogator I, a male M and a female F are divided in three separated rooms. The interrogator can communicate with M and F but the male and the female cannot communicate. Communication is only based on text. The goal of the male is to fool the interrogator into believing he's a female, while the female has to convince I that she is actually a woman. The interrogator can ask questions to both and finally has to decide who of the two is the female and who is the male.

Part 2

In the second part of the experiment the male is replaced with a computer that has the same goal of the male. The goals of the interrogator and of the female are the same.

Final result

The test is repeated many times and if the number of times the interrogator is wrong in the first part is comparable with the number of times the interrogator is wrong in the second one (i.e. if the machine can trick the interrogator a number of times comparable with the number of times the human has fooled the interrogator), than the machine can be considered intelligent.

Problems of the Turing test

The main problem of this test is that intelligence isn't recognised in a general way. For instance if the interrogator isn't that smart, even a system that doesn't behave intelligently, is considered to be intelligent.

1.3 Systems thinking rationally

This definition switches the focus from a real world reference (the human being) to a theoretical and ideal one. Namely, it focuses on the process of thinking that follows logical deduction.

In fact, accordingly to this definition, a system is intelligent if it thinks in a formally correct way using logic and laws of thought. In other words a system is intelligent if it can do the right thing (defined using logic).

This definition allows to define a standard for intelligence because rationality can be defined formally.

1.4 Systems acting rationally

This definition is similar to the previous one but it focuses on the behaviour of the system without paying attention to the reasoning process. In other words this definition focuses on finding a standard to define what a good behaviour is.

1.4.1 Application in real life

This definition is the easier to apply in real life application because it does not puts constraints on the internals of the reasoning process and it uses logic rules (which can be expressed in a formal way).

Chapter 2

Rational agent

An agent is an entity that interacts with an environment that can be

- **Virtual** like a video-game, the internet or a file-system.
- **Real**.

Examples of agents are software and robots.

Definition 1 (Agent)**.** *An agent can **perceive** the environment an act on it. In particular an agent does (in a loop) three main actions*

1. *Perceives the environment.*
2. *Decides how to behave.*
3. *Acts.*

Artificial intelligence has to design an agent that fits Definition 1 of agent. This behaviour defines what an agent is but many different agents fits to this description, for example

- A human.
- A thermostat (reads the ambient temperature, decides if it has to cool or heat the air, activates or regulates the system).

What we are interested in is something in between these two extreme examples.

2.1 Agent

Definition 2 (Agent)**.** *An agent is defined as a function f that maps a sequence of perceptions to an action (executed at time t).*

$$f(p_1(t), ..., p_n(y)) \to a(t)$$

It's important to underline that Definition 2 is true if time and perceptions are discrete (fine or coarse grained), otherwise a computer could not perceive the environment and act on it.

In principle there are no constraints on the definition of the function f.

2.2 Rational agent

Definition 3 (Rational agent)**.** *An agent is rational when for every possible sequence of perceptions, it chooses the action that maximises the expected value of its performance measure, given its knowledge action up to that moment.*

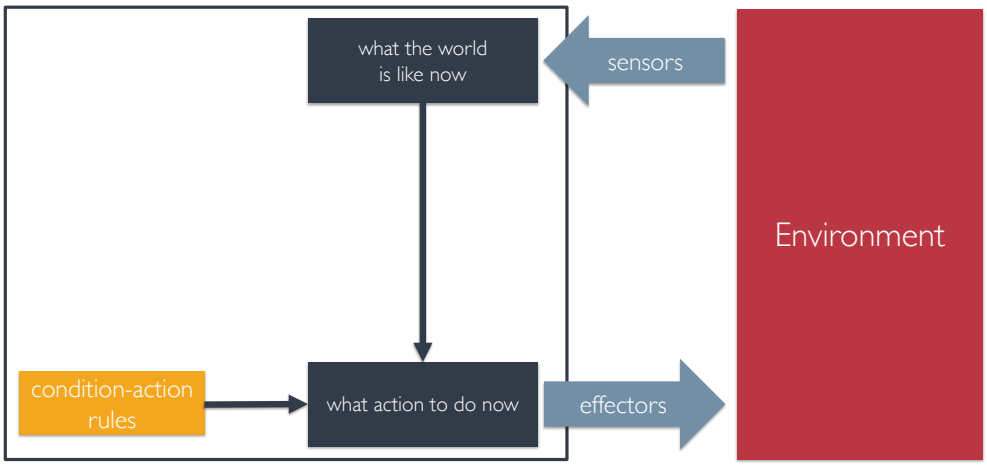

Figure 2.1: A rational agent

It is important to understand that rational agents do not have to be omniscient, in fact Definition 3 states that the agent has to maximises the performance given only the perceptions and the knowledge it possesses up that moment, therefore the agent may not have a complete knowledge of the environment. Furthermore rational agents can also learn from the environment by storing the perceptions.

From Definition 3 we can obtain three important results

1. A rational agent has to possess **knowledge**.
2. A rational agent has to **act** in the environment.
3. A rational agent has to maximise some value that represents the **performance** (how it acts in the environment).

Let's analyse each of these characteristics.

2.2.1 Knowledge

A rational agent has two sources of knowledge

- **Perceptions** from the environment (that can also be stored to learn).
- **Previous knowledge** provided by the designer.

2.2.2 Actions

An agent can change the state of the environment through actions. In this case it's also important to consider the various types of environment the agent can interact with.

2.2.3 Performance

The performance and its evaluation are usually defined by the designer. An example of performance measure is the utility function that states the result the designer would like the agent to reach.

2.3 Environment

As seen before the environment is the main source of information of the agent. For this reason it is important to classify the environments to design agents that can better learn from the environment.

2.3.1 Classification of environments

Observability

An environment can be

- **Completely observable** if the agent can completely perceive the state of the environment.
- **Partially observable** if the agent can only observe part of the state of the environment.

Dynamism

An environment can be

- **Static** if it does not change if not for the hand of the agent.
- **Dynamic** if it does change even if the agent is not acting.

This difference is important because the agent has to behave differently, for instance if the environment keeps changing the agent has to know that he might do some action based on data that might be obsolete.

Continuity

An environment can be

- **Discrete** if it is described by discrete variables (likes moves in a game).
- **Continuous** if it is described by continuous variables (like trajectories).

Even if the environment is continuous the agent keeps working on discrete perceptions.

Future states

The environment can be

- **Deterministic** if the next state is fully determined by the action of the agent and the previous state of the environment.
- **Stochastic** if it is only possible to give a distribution of probability of the possible next events.
- **Non deterministic** if it is only possible to determine the next possible states but without an associated distribution of probability.

2.4 Architectures

Building intelligent (rational) agents is a difficult task so it is useful to start from some models (i.e. architectures) developed by AI researchers.

Choosing the right architecture is an important task because each one fits better to certain kinds of problems. Furthermore there isn't a perfect model for each type of model, in fact every model could have its advantages and disadvantages for a certain kind of problem.

Agents are divided in 4 different types with increasing levels on complexity

1. **Simple reflex agents**.
2. **Reflex agents with state**.
3. **Goal-based agents**.
4. **Utility-based agents**.

These architectures are presented as static agents but it is important to underline that they can actually learn by exploiting the past experience to improve their performance.

2.4.1 Simple reflex agents

Simple reflex agents

- Perceive the environment through **sensors**. In other words sensors allow the agent to know what the environment is like in the current instant.
- Processes the perceptions obtained via its sensors and decides **what to do next** according to a set of condition-action rules (basically if-then rules). For instance the agent could be a robot that scans the environment and processes each frame. For each frame if the robot sees an obstacle then it moves away from it. If multiple rules could be activated at the same time, the agent needs an arbitrage mechanism that allows it to select the action to perform. For example we could give a priority to each rule so that only the rule (among all the possible rules that can be activated) with the highest priority is activated.
- Acts on the environment using **effectors**.

The decision of this type of agent is based only on the current perception, so this kind of agent performs well in a **completely observable environment**.

If this type of agent faces the same situation multiple times, it will in always make the same decision.

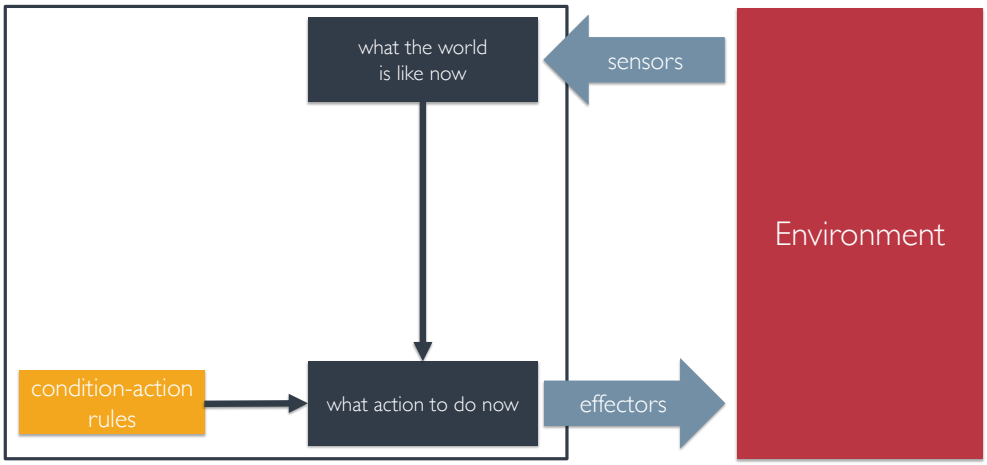

Figure 2.2: A simple reflex agent

Real life example

The first versions of Roomba used this model, in fact it worked in a simple way: every time it found an obstacle (like a corner or a wall) it changed direction. A Roomba did not follow any fix path but it just roomed around in a random path decided by the bumps against the obstacles.

2.4.2 Reflex agents with state

The main limitation of the simple reflex agent is that it works well in a completely observable environment.

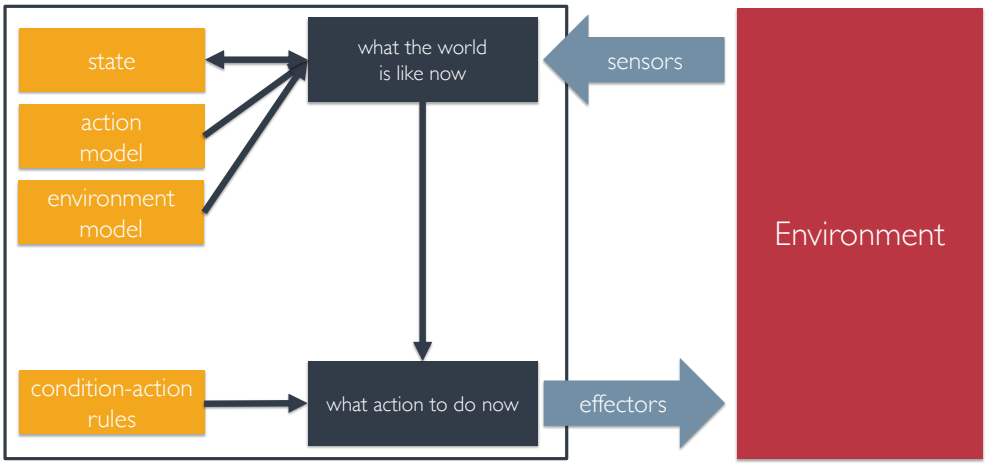

Figure 2.3: The model of a reflex agent with state

To solve this problem we can add a state to the agent. When the agent perceives the environment

1. The agent updates the state.
2. The agent uses the previous state to assess the part of the environment that is not visible.

To obtain this result the agent also needs to have two different models

- The **action model**.
- The **environment model**.

The decision-making part of the model remains unchanged (the agent takes decisions using condition-action rules). The only difference is that the conditions are matched against more complex environment states (i.e. the combination of state and perceptions).

This type of agent, like the simple reflex agent, behaves identically when it is faced against the same situation because the decision-making part of the agent still relies on if-then rules.

Real life example

The newer versions of Roomba keep track of the parts of the houses that have already been cleaned to reduce the time needed to clean the whole house.

2.4.3 Goal-based agents

In goal based agents, the perception part of the agent is the same as for a reflex agent with state, in fact it has a state, an action model and an environment model. The decision-making part instead is based on the concept of goal. A goal is a state of the environment that the agent would like to reach. For instance if the agent has to play chess the goal is a checkmate, if the agent is an assembly robot the goal is to have the object completely assembled.

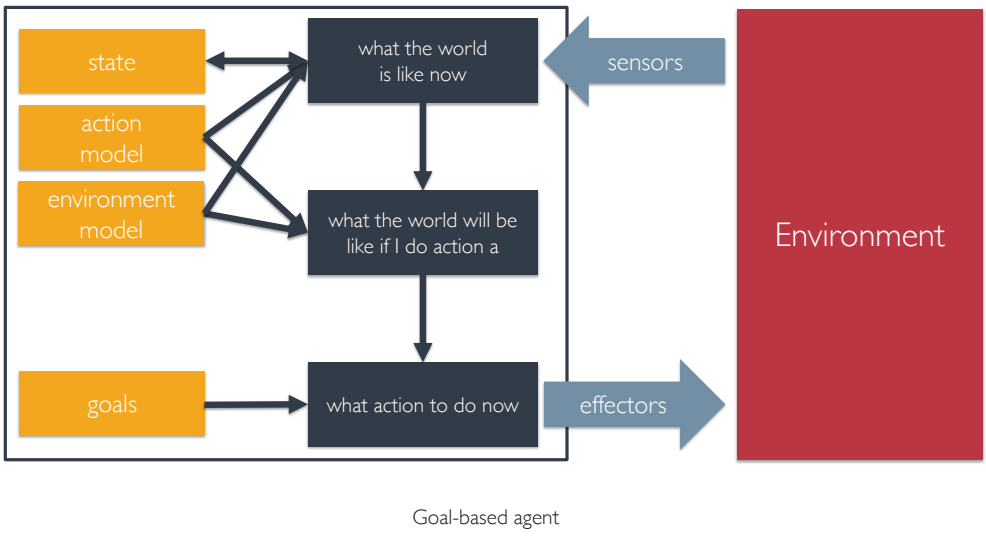

Figure 2.4: The model of a goal-based agent

In some sense the other two previous models had an implicit goal expressed using the rules.

This type of agent, to decide how to act, initially doesn't do any action but tries to evaluate all the possible results out of a set of possible actions and chooses the action that generate a result that satisfies a certain property (usually the action that brings the agent closer to the goal). For instance an agent that has to play chess tries every possible move (without actually moving the pieces) and chooses the one that is closer to a checkmate position.

In other words this type of agent doesn't act unless it has a plan to reach the goal.

Real life example

Tic-tac-toe solvers build a tree that consider every possible move and use it to follow the path that bring to a win situation (i.e. the goal).

2.4.4 Utility-based agents

In goal-based agents a goal is a binary property. In other words a state of the environment is either a goal or not. This property, depending on the type of problem, can be an advantage or a disadvantage. In some cases we need more expressivity.

Utility-based agents share the same model as goal-based agents but the goal is replaced by the concept of **utility**. The utility is a real number attached to a state of the environment. Therefore the concept of utility is the generalisation of the concept of goal, in fact we can think a goal as a utility that can only hold the values 1 or 0.

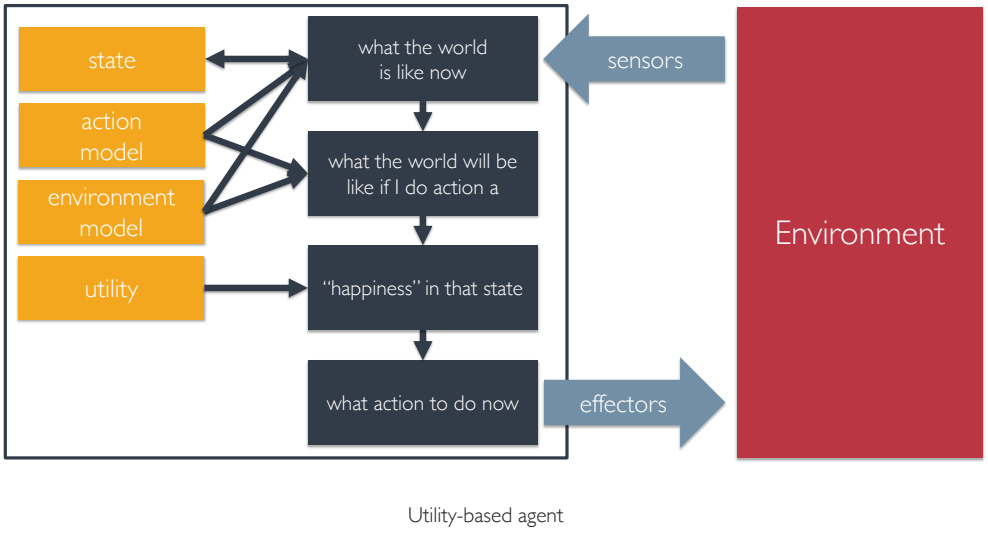

Figure 2.5: The model of a utility-based agent.

Thanks to the concept of utility it is also possible to confront the states of an environment. Utility values can also be attached to the paths that lead to a certain state so that we can also put emphasis on the path to reach the goal. Utility can also be used to balance different qualities of a state. For these reasons this agents can be used in many different environments.

Chapter 3

Use of artificial intelligence

3.1 The objective of AI

Artificial intelligence can be divided in three categories

- **Weak Artificial Intelligence**.
- **Strong Artificial Intelligence**.
- **General Artificial Intelligence**.

3.1.1 Weak Artificial Intelligence

Definition 4 (Weak Artificial Intelligence)**.** *Weak Artificial Intelligence tries to build an intelligent system that can perform like a human, without caring about how this is achieved.*

In other words the focus of this AI is on performance. This form of AI is equivalent to the definition of AI as a system that acts like a human or rationally.

3.1.2 Strong Artificial Intelligence

Definition 5 (Strong Artificial Intelligence)**.** *Strong Artificial Intelligence tries to build an intelligent system that can exactly replicate how human think.*

In other words the agent is consciously thinking. This form of AI is equivalent to the definition of AI as a system that thinks like a human.

3.1.3 General Artificial Intelligence

Definition 6 (General Artificial Intelligence)**.** *General Artificial Intelligence tries to build a system that can solve an arbitrary wide variety of tasks, including novel ones, as well as a human.*

In other words the system has to be able to reach human-like performances on a wide variety of tasks.

For instance nowadays we have programs that can beat a chess world champion at chess and can think many moves ahead. But if we use the same program on another different task it does not perform well, for example it cannot plan the production of a company. These programs are not general AI.

3.2 Types of Artificial Intelligent Systems

AI systems can be divided in three classes depending on their application

- Milestone applications.
- Stand alone systems.
- Ubiquitous or embedded systems.

3.2.1 Milestone applications

Milestone application are application that aren't very useful (i.e. they do not fulfil an actual need) but they are build to show what a company can do. They have some sort of advertisement purpose.

3.2.2 Standalone systems

Standalone systems are systems that can be recognised as independent AI systems. Some examples of standalone systems are warehouse robots and autonomous cars.

3.2.3 Ubiquitous or embedded systems

Embedded systems are AI systems that work inside a non-AI environment. For instance voice assistants like Siri or Alexa on smartphones and speakers are embedded systems because they operate in an environment that is not an AI system and cooperate with other non-AI components.

Part II

Search

Chapter 4

Search problem

Many real world problems can be formulated as search problems that can be solved by a **goal agent**.

To discuss search problems it is useful to introduce the 8-puzzle game. We will design an agent that can solve this puzzle.

The 8-puzzle game is played on a 3×3 board with 8 tiles numbered from 1 to 8. Each cell of the board contains a tile and one cell is left blank (8 tiles on 9 cells leave a blank cell). The goal of the game is to move the tiles adjacent to the blank cell to order the tiles from 1 to 8 starting from the upper-left corner

8	2	
3	4	7
5	1	6

Table 4.1: An initial configuration for the 8-puzzle game.

1	2	3
4	5	6
7	8	

Table 4.2: The winning configuration for the 8-puzzle game.

A search problem is defined by

- A **set of states** $S = \{s_0, ..., s_n\}$ and an initial state $s_0 \in S$.
- An **actions function**.
- A **result function**.
- A function that verifies if a **state is a goal** state.
- A **cost function** that returns the cost of applying a certain action to go from one state to another.

4.1 Definition

4.1.1 States

A state is a valid configuration of the environment. In our example both configurations 4.1 and 4.2 are valid states. Alternatively, states are abstract representations of a possible physical situation with the same fundamental properties of the physical situation.

When an agent solves a search problem it goes from an initial state to a goal trough a series of intermediate states. For this reason, the solution to a search problem can be represented as a sequence of states.

4.1.2 Actions function

The `actions(s)` function returns all the possible actions that we can do from a state `s`. An action is a transformation of the environment (i.e. a function that transforms a state `s` in a new state `s'`).

In the 8-puzzle game we can model two equivalent types of action

- We can say that an action moves a tile to the blank adjacent cell. This model is more intuitive for a human.
- We can also say that an action swaps the blank cell with an adjacent tile. This action cannot be replicated in the real world but it much simpler to do by a computer.

4.1.3 Result function

The `result(s, a)` function returns the state `r` the environment reaches when the agent applies the action `a` to the state `s`. Obviously the action `a` has to be contained in the set of actions returned by the function `actions` applied to `s`.

4.1.4 Goal function

To define a search problem we also have to define a function that says if a state `s` is goal state.

Goal states are defined by a goal function and not by a set of states because we might have too many final states to enumerate (like all the possible checkmate states in chess) and furthermore it is easier in a programming language to check if a state is a goal using a function.

4.1.5 Cost function

Every time the agent does an action we have to assign a cost to such action. For smaller problems, like the 8-puzzle game, every action has the same cost (typically 1).

If the cost function is a combination of factors we should switch from goal-based agents to utility function-based agents.

4.2 Solution

States can be represented with a directed graph in which

- Nodes are the possible states.

- Weighted arcs connect node `s` to node `s'` if and only if `s'` is a successor state of `s`. Namely `s` and `s'` are connected if and only if there exists an action `a` that applied to `s` leads to `s'` (i.e. `result(s, a)=s'`). The weight of the arc represents the cost of getting from `s` to `s'`.

A solution is therefore a path from the node that represents the starting state to one of the nodes that represent a goal state. The optimal solution is the path with minimum cost. If there's no path from an initial state to a goal state then the problem has no solutions.

4.2.1 Environment

The environments in which search problems agents operate are

- **Fully observable**. Namely the agent knows the complete state of the environment.
- **Static**. Namely the environment changes only due to an action of the agent.
- **Discrete**. Namely the environment has a finite (although arbitrarily large) set of states.
- **Deterministic**. Namely if an agent applies an action `a` to state `s`, it always get the same result state `s'`.

4.2.2 Greedy solution

The solve a search problem an agent simply has to find the best path from one node to another. Dijkstra algorithm, for instance, should do the job. The only problem with classical graph algorithms is that such algorithms have to memorise and generate all the possible states (i.e. nodes) and for some problems it isn't feasible to generate them all (it would take too much time). For instance the 8-puzzle game has 362880 possible states. If the game is expanded to 15 tiles the number of possible states grows to $1.3 \cdot 10^{12}$.

4.2.3 Tree solution

Search space as a graph

The search space that the agent has to explore can be represented as a graph. In particular,

- A node represents a state of the environment.
- An arc from node n_1 to node n_2 represent an action that brings the environment from n_1's state to n_2's state.

Tree

To search the optimal path in a graph we can use a search tree in which

- The root of the tree is the initial state s_0 of the graph.
- The children nodes of a node `s` are obtained applying the `result` function to `s` using every action obtained from the `actions` function.

In other words to obtain a child node from a state `s` an agent has to

1. Call the `actions` function on `s`.

2. Choose an action obtained at step 1 (say `a1`) and call the `result` function passing `s` and `a1` as arguments.

A tree built this way can have the same state as child of two different parents because starting from two states `s1` and `s2` we can apply to different actions `a1`, `a2` and get the same result state `s`.

Tree search algorithm

The algorithm that searches a goal state starting from a initial state is the following

```
LOOP DO
    IF no more nodes for expansion
        RETURN FAIL

    select a node not yet expanded (initially the root)

    IF node IS GOAL
        RETURN SOLUTION
    ELSE
        expand the node adding the resulting nodes to the tree
END LOOP
```

The nodes that have still to be expanded are called frontier. Usually a frontier is implemented as a priority queue in which the priority to each element depends on how much promising a state is. If a state is promising (i.e. brings us closer to a goal state) it is inserted with high priority. For instance a check state in chess is a promising state while a state in which the queen gets eaten is a low priority one. To expand a note the agent has to

1. Apply the `actions` function.
2. For each action returned by the `actions` function compute the `result` function to generate the next node.

Chapter 5

Search strategies

5.1 Uninformed search strategies

When facing a search problem we have to explore a graph to find a goal state. The exploration can be done in different ways and using different algorithms. The main focus of these algorithms is the frontier (i.e. the list of nodes that have to be expanded to find a goal state). When a new node is expanded all the new nodes (i.e. the nodes returned by `result`) have to be inserted in the frontier in some order that depends on the algorithm. To ensure fast insertions and extractions and, if required, to keep the data structure ordered a frontier is usually implemented with a **queue** (is some cases prioritised). An array would be too slow to ensure ordering and fast insertions.

Uninformed algorithms Uninformed algorithms use only the cost of a move (i.e. the cost needed to go from a node to another node in the graph) to determine how to sort the queue.

It is important to underline that when an element is added to a queue, such queue isn't reordered. In other words the node has to be inserted in the right place to ensure that the frontier is ordered. This allows to use a queue in a more object-oriented way (it only has to be implemented once). To put it in another way, the algorithms always expands the first element of the queue and what changes between the different algorithms is in which order such elements are inserted in the queue.

Informed algorithms On the other hand informed algorithms use other information (like the number of pieces on a board in chess) relative to a certain state to determine the next node to explore. Uninformed algorithms are more general (i.e. can always be applied) but are less powerful than informed algorithms.

General model In general an uninformed algorithm can be described as follows

```
LOOP DO
    IF there are no more nodes candidate for expansion THEN RETURN failure

    select a node not yet expanded

    IF the node corresponds to a goal state THEN RETURN the corresponding solution
    ELSE expand the chosen node, adding the resulting nodes to the tree
```

The difference between the algorithms that implement uniformed search is simply how nodes are inserted in the frontier.

Ingredients To build an uninformed search algorithm we need

- The **states** of the problem.
- The `actions(s)` function that returns all the actions that can be done in state `s`.
- The `result(s, a)` function that returns the state in which the agent ends after applying action `a` to state `s`.
- The function that says if a certain state is a **goal state**.
- The `cost(a)` function that returns the cost of action `a`.

Evaluation criteria To decide how good an algorithm is we can use three criteria

1. **Completeness**. An algorithm is complete if it finds the solution (i.e. finds a goal state). An algorithm has to guarantee that the solution is always found.
2. **Optimality**. An algorithm is optimal if it can find the path with the minimum cost to a goal state.
3. **Complexity**. The complexity of an algorithm is measured in terms of time and space complexity in the worst case scenario.

To evaluate the complexity of an algorithm we can use

- The **branching factor** b that represents the maximum number of successors for a node. The branching factor also represents the possible actions that can be done starting from a node. An high value of b indicates a big number of actions, hence a large number of possible decisions that increases the complexity and the difficulty of the problem.
- The **depth of the shallowest goal node** d. In some cases the value of d is fixed (e.g. in the 8-queens problem).

These parameters are a property of the problem and are used to determine how much time and memory we need to find a solution.

5.1.1 Breadth first search

The breadth first search (BFS) algorithm expands all the nodes of a certain level before expanding the nodes at the next level. The nodes are added level by level to the frontier. In other words all the nodes at level k have to be expanded before expanding the nodes at level $k + 1$.

To put it in another way the root is expanded first, then all the successors of the root are expanded, then their successors, and so on. The nodes are added with increasing value of depth so that the shallowest node is always extracted. If all the nodes have the same depth then a tie-breaker is used (e.g. the cost to reach every node).

Goal testing In this algorithm goal testing on the nodes can be done before adding a node to the frontier, or before expanding the node (i.e. when the node is popped from the queue).

Queue implementation In breadth first search every node has the same cost so we can implement the queue with a simple FIFO non-prioritised queue.

Performance The breadth first search algorithm

- Is **complete**, in fact if the maximum number of branches is finite we will eventually (it may take a lot of time) find a solution.
- Is **optimal** when the costs are all equal (at least for nodes at the same level) or when the cost of the path of any node is not a decreasing function of the depth of the node.
- Has an **exponential** worst time and space **complexity**

  $$\mathcal{O}(b^{d-1})$$

  Such complexity is obtained because in the worst case (every node has b children) every node at level l has to generate b children. In total

  $$\sum_{l=1}^{d} b^l + (b^{d+1} - b)$$

  nodes are generated so the complexity

  $$\mathcal{O}(b^{d+1})$$

  Such nodes have to be stored in memory, thus the spatial complexity is the same as the time complexity. The last member $(b^{d+1} - b)$ is justified by the fact that before finding the goal state at level d all the nodes at level d are generated. If all the nodes of level d would have been generated the term would have been b^{d+1}, but from such nodes we have to remove the nodes generated by the node that is the goal (that are b).

5.1.2 Uniform cost search

The uniform cost search (UCS) algorithm is a generalisation of the breadth first search algorithm. In particular the UCS algorithm sorts the nodes in the frontier according to the cost of the path from the root. The node popped from the frontier is always the one with the smallest cost.

Test goal It's important to highlight that the node popped from the queue isn't always the shallowest one because a sequence of moves can be cheaper than a single move. For this reason it's fundamental to **test if a node is a final state only before the expansion of a node** (i.e. when the node is popped, not when the node is inserted), in fact a deeper node not added yet could be a cheaper goal state.

Generalisation of breadth first This algorithm is a generalisation of BFS, in fact if we consider identical costs for all moves we obtain the behaviour of the BFS algorithm. This means that UCS is relevant when the actions have different costs.

Queue implementation In this case the queue can't be implemented as a FIFO queue, in fact we should use a priority queue that allows fast insertions and keeps the nodes ordered using the full path cost as priority.

Performance The uniform cost search algorithm

- Is **complete** if the branching factor b is finite and the step cost is bigger than a small positive value ε.
- Is **optimal**.
- Has a time and space complexity of

$$\mathcal{O}(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$$

where C^* is the cost of the optimal solution. In general, the complexity of this algorithm is greater than the complexity of BFS. When all costs are equal, the term $1 + \lfloor \frac{C^*}{\varepsilon} \rfloor$ is equal to d, thus UCS has the same complexity of BFS.

5.1.3 Depth first search

The depth first search (DFS) algorithm has the opposite approach with respect to BFS, in fact in this case the algorithm visits a branch of the search tree until it reaches a leaf, then it goes back to a parent (i.e. in the direction of the root) until it finds a branch not visited yet and visits it. In other words after visiting a branch the algorithm goes back to the nearest node that has a branch not visited yet.

In the tree in Figure 5.1, leaf C is visited first ($R \to A \to C$). Then the algorithm comes back to A and visits D. After that the algorithm goes back to the root and visits the right branch applying the same procedure ($B \to E \to B \to F$.)

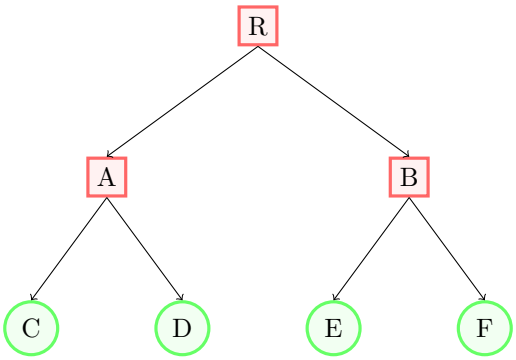

Figure 5.1: A search tree.

Queue implementation The queue used in DFS can be implemented using a LIFO queue (i.e. a stack) in fact the first node visited has to be expanded immediately unless it is a leaf. In this case the nearest node is expanded. When expanding a node, depth-first search always selects the node at maximum depth (i.e. the deepest one) in the queue. Consider the tree in Figure 5.1, in this case initially B and A are added to the stack (but we could have added A before B).

```
S = [A, B]
```

The last element added is A so we expand A and remove it from the stack. After expanding A we can add D and C to the stack.

```
S = [C, D, B]
```

After popping C (here we can see that we have expanded the node at maximum depth) and expanding it we can go back to D and after expanding it we go back to B (both C and D are leaves so the expansion doesn't add nodes to the queue).

```
Q = [B]
```

From now on the process continues as seen in the previous steps. As seen in this example the queue works like a stack, as expected.

Performance The depth first search algorithm

- Is **not complete**, in fact if a branch has infinite length the algorithm never stops and it never finds the solution. Infinite branches are generated when the states' graph is cyclic (i.e. a set of moves allows us to return to a certain configuration).
- Is **non optimal**, in fact it stops when it finds a solution, but if it analyses a branch until the end, it might happen that a shallower solution exists on another branch not visited yet. In other words it's not optimal because it visits the deepest leaves first.
- Has a **linear space complexity**

  $$\mathcal{O}(bm)$$

  and an **exponential time complexity**

  $$\mathcal{O}(b^m)$$

  where m is the maximum depth of the tree. The linear time complexity is justified by the fact that the we don't have to keep in memory every node, but just the nodes of the branch that we are currently analysing. This happens because if in a branch we can't find a goal state, than we don't need anymore the nodes of such branch (because we have explored it all). Generally m (that can also be infinite) is grater than d and most of the times it is much bigger than d, in fact d is the shallowest solution while m is the deepest leaf.

5.1.4 Depth limited search

The depth limited search (DLS) algorithm is a variant of the depth first algorithm. The only difference is that it considers the nodes at the level L as leaves (even if they have children).

Performance The depth limited search algorithm

- Is **complete** if the maximum depth L is not smaller than the shallowest goal depth d.
- Is **not optimal**, in fact the algorithm still checks the deepest leaves first.
- Has linear space complexity

  $$\mathcal{O}(bL)$$

  and exponential time complexity

  $$\mathcal{O}(b^L)$$

5.1.5 Iterative depth limited search

The iterative depth limited search (IDLS) algorithm repeats the depth limited search algorithm using increasing values of L. In this case a new tree has to be built for each iteration.

Performance The iterative depth limited search algorithm

- Is **complete**, in fact it repeats the DLS algorithm until $L = d$.
- Is **optimal** if the costs are the same for each move, in fact it stops when $L = d$.
- Has linear space complexity
$$\mathcal{O}(bd)$$
and exponential time complexity
$$\mathcal{O}(b^d)$$
These two results combine the complexity of the DFS and BFS algorithms and are obtained summing all the complexities until $L = d$, in fact for space complexity
$$\mathcal{O}\left(\sum_{L=1}^{d} Lb\right) = \mathcal{O}(b + 2b + \cdots + db) = \mathcal{O}(bd)$$
and for time complexity
$$\mathcal{O}\left(\sum_{L=1}^{d} b^L\right) = \mathcal{O}(b + b^2 + \cdots + b^d) = \mathcal{O}(b^d)$$

5.1.6 Bidirectional search

The bidirectional search algorithm applies two parallel searches

1. One search starts from the initial state and expands the nodes.
2. One search starts from the goal state and expands the nodes backwards.

The search ends when the two searches find two nodes that share the same state.

Issues This algorithm can be very fast but can't be used when there are too many goal states (e.g. in chess).

Furthermore finding the previous state (i.e. moving backward from the goal state) can be difficult and can generate a large number of previous states.

5.1.7 Repeated state problem

In all of the previous algorithms we haven't considered the fact that in many problems it's possible to return to a state previously visited. In these cases the search tree becomes very big (cycles create infinite branches because the algorithm keeps adding nodes even if it has returned to an already visited node) and some algorithms could even fail to find a solution (e.g. the DFS algorithm, because cycles can create infinite branches).

Graph search algorithms To solve the repeated states problem we can create a list, called **closed list**, with the nodes already visited. This way we can verify if a node is already in the closed list and in such case we can simply ignore it and expand the next node of the frontier. This operation can be done before putting the node in the list or before expanding it (i.e. after removing it from the list).

The algorithms that uses this strategy to solve the repeated states problem are called **graph search algorithms**.

General model The general model of a graph search algorithm is the following

```
initialise the root of the tree with the initial state of problem
initialise the closed list to the empty set
LOOP DO
    IF there are no more nodes candidate for expansion THEN
        RETURN failure

    choose a node not yet expanded

    IF the node corresponds to a goal state THEN
        RETURN the corresponding solution
    ELSE IF the state corresponding to the node is not in the closed list THEN
        add the corresponding state to the closed list
        expand the chosen node, adding the resulting nodes to the tree
```

5.2 Informed search strategies

Informed search strategies add knowledge about a problem. Such knowledge is specific to a problem or to a class of problems and allows us to optimise the search of the goal state.

In particular informed search strategies

- Exploit knowledge that isn't in the definition of the problem.
- Use the same algorithms used in uninformed search strategies but the frontier is ordered using a function f.

Ordering function The main difference between informed and uninformed search strategies is that the former uses a function $f(n)$ that calculates how promising a node n is. In particular the agent wants to minimise the function f, meaning that the frontier (i.e. the priority queue) is ordered in increasing order of $f(n)$ and the element popped out from the queue is the one that has the minimum value of $f(n)$.

```
frontier[0] = min([f(node) for node in frontier])
```

Example For instance consider the 8-puzzle game (the one in which 8 tiles numbered from 1 to 8 have to be reordered in a 3×3 matrix with one empty slot). In this case the ordering function f could be the number of tiles that are misplaced. This function satisfies the definition because if almost all the tiles are in order than we find ourselves in a good state and the ordering function has a small value.

5.2.1 Best-first greedy search

Heuristic function

The best-first greedy search uses an heuristic function $h(n)$ to determine the order of the nodes in the frontier

$$f(n) = h(n)$$

Characteristics The heuristic function

- Has to be known and represents an **estimate cost** of the shortest path from a node n to a goal node. If the function h was the exact cost, the problem could have been solved using uninformed search strategies.
- Can be hard to build.

The heuristic function can be computed or stored in every node. In the latter case we can use a dictionary in which a value of h (i.e. the value of an entry) is associated to each node (i.e. the key of the entry).

Relaxed problem The heuristic function represents the solution to a relaxed problem, that is a simpler problem that has relaxed rules and constraints. For instance if we consider a robot that has to navigate from a point A to a point B in an environment with obstacles, the relaxed problem could ignore the existence of the obstacles. In this case the heuristic function could be the Pythagorean distance or the Hamiltonian distance (distance calculated only using up, down, left, right movements) between the current position to the goal position.

Algorithm

Algorithm The algorithm, until the frontier is empty

1. Pops the node with minimum f from the frontier.
2. If the node is a goal state then it is returned and the algorithm stops.
3. Expands the node.
4. Adds the newly generated nodes to the frontier only if such nodes do not generate cycles.

The pseudo-code for the best-first algorithm is shown below.

```
frontier = PriorityQueue(sorting_function=f)

WHILE frontier is not empty:
    node = frontier.pop()
    IF is_goal(node) THEN
        RETURN node
    FOR child IN expand(node) DO
        IF NOT is_cycle(child) THEN
            frontier.add(child)

RETURN FAILURE
```

Performance The best-first search strategy

- Is **not complete**, in fact, the algorithm can get stuck in a local optimum.
- Is, in general, **not optimal** for the same reason of incompleteness.
- Has an exponential time and space complexity

$$\mathcal{O}(b^m)$$

where m is the maximum depth of the search space. This complexity can be reduced using better heuristic functions.

Local optimum

The best-first search strategy can only find the local optimum, in fact the algorithm expands the node with the minimum value of h without considering all the previous moves. In particular there could exist a path that in general is less costly but that initially is not convenient. This happens because the h function considers a relaxed problem. For instance, in the example of the robot that has to find a way from a point A to a point B we consider the Pythagorean distance as heuristic function. In this situation if we follow the closest points to the goal we might have to travel around an obstacle thus lengthen the path.

5.2.2 A* search

Function

The A^* algorithm solves the problem of the local optimum. In particular in this algorithm the function f is the sum of

- The cost so far $g(n)$.
- The heuristic function (as defined for best-first greedy search) $h(s)$.

$$f(n) = g(n) + h(n)$$

Cost function The cost function $g(n)$ is known (i.e. it's exact), in fact it's computed using the part of the tree that we have already visited. For instance $g(n)$ can be the length (or the cost) of the path from the root to the current node n.

Heuristic function The heuristic function $h(n)$ is (as for the best-first greedy search) an estimate of the cost to reach the goal considering a relaxed problem.

Admissible heuristic function The heuristic function $h(n)$ is admissible if and only if

- It's **optimistic**, that is it rounds down the actual cost h^*. In other words the function $h(n)$ has values between 0 and $h^*(n)$ for all nodes n

$$0 \leq h(n) \leq h^*(n) \quad \forall n$$

- It is 0 only if n is a goal state.

$$h(n) = 0 \iff n \text{ goal}$$

In other words the heuristic function is admissible when it considers the cost of the optimal solution in a relaxed problem, in fact if the problem has less constraints, the cost of the solution is less than the actual cost of the solution.

Consistent heuristic function The heuristic function $h(n)$ is consistent if and only if for all possible nodes n, $h(n)$ it's smaller than the sum of

- the cost to get to a subsequent state n' using action a

$$c(n, a, n')$$

and

- the heuristic function of n'

$$h(n')$$

$$h(n) \text{ consistent } \iff h(n) \leq c(n, a, n') + h(n') \quad \forall n$$

If a function is consistent than it is also admissible. In other words consistency is stronger than and implies admissibility

$$h(n) \text{ consistent} \Rightarrow h(n) \text{ admissible}$$

Algorithm

The algorithm works exactly as best-first. The only difference is in the function used to sort the priority queue. In this case we use $f(n) = g(n) + h(n)$.

```
f = ( lambda node: g(node) + h(node) )
frontier = PriorityQueue( sorting_function=f )

WHILE frontier is not empty:
    node = frontier.pop()
    IF is_goal(node) THEN
        RETURN node
    FOR child IN expand(node) DO
        IF NOT is_cycle(child) THEN
            frontier.add(child)

RETURN FAILURE
```

Performance The A^* algorithm

- Is **complete** and **optimal** for tree-search when the heuristic function $h(n)$ is **admissible**.
- Is **complete** and **optimal** for graph-search when the heuristic function $h(n)$ is **consistent**.
- Has a time and space **exponential complexity**.

Tree-search and admissible functions The A^* tree-search algorithm is complete and optimal for admissible heuristic functions $h(n)$ and cost of the path larger of a small positive value ε.

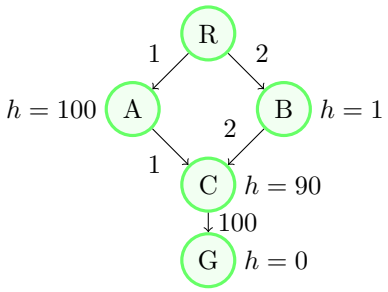

Figure 5.2: Graph-serach with non consistent heuristic function.

Graph-search and consistent functions The A^* graph-search algorithm is complete and optimal for consistent heuristic functions $h(n)$ and cost of the path larger of a small positive value ε.

If the function h isn't consistent we can't ensure optimality, in fact when we arrive in a node it's not possible to see other paths (eventually cheaper) that bring to such node and when we can see them we might already have evaluated the node. In other words the first path that arrives on a node might remove it from the frontier, put it in the close list and expand it so that no other path can expand it. This behaviour is, in general, ok in tree-search but in some cases the tree can expand too much and the algorithm might become slow.

To better explain the problem with non consistent functions consider Figure 5.2. Initially the algorithm follows the path on the right ($R \to B \to C$) because every node always has minimum f. When we arrive at C, we decide to expand it because the value of $f(C) = 90 + 2 + 2 = 94$ is less than the value of $f(A) = 100 + 1 = 101$. When we expand C we also add it to the closed list so that it cannot visited again. After visiting C we should expand A but its successor is in the closed list thus A is simply removed from the frontier. Finally G is visited and because it is a goal state, the algorithm stops. The path obtained ($A \to B \to C \to G$) has cost 104 which is not optimal because the path that passes through A has a cost of 102. This happened because $h(A) > c(A, C) + h(C)$.

Properties

The A^* search algorithm

- Expands all nodes with $f(n)$ smaller than the optimal cost C^*.
- Expands some nodes with $f(n)$ that equal the optimal cost C^*.
- Expands no nodes with $f(n)$ bigger than the optimal cost C^*.

Notice that the function $h(n) = 0$ (i.e. the function that returns 0 for every node) is consistent (thus admissible) but completely useless.

Accuracy

Given two heuristic functions h_1 and h_2 we say that h_2 dominates h_1 if and only if $h_2(n)$ is greater than $h_1(n)$ for all nodes n

$$h_2 \text{ dominates } h_1 \iff h_2(n) \geq h_1(n) \quad \forall n$$

If a function h_2 dominates h_1 it means that h_2 is more accurate and better approximates the optimal cost (i.e. the cost of the non-relaxed problem).

Non admissible functions in practice Sometimes in practice it's better to use heuristic functions that aren't admissible or consistent but that can solve a problem better or faster.

5.2.3 Weighted A*

Weighted A^* works exactly like A^* with the only difference that a weight w is added to each value of the heuristic function. In other words the function f is

$$f(n) = g(n) + wh(n)$$

Performance The weighted A^* algorithm is faster and visits less nodes than A^*.

5.2.4 Iterative deepening A*

Iterative deepening A^* (IDA*) applies the same concept already seen for the iterative depth limited search to A^*. In particular we put a limit to the value of h and repeat multiple iteration of the same algorithm with increasing limit values of h.

Heuristic function This algorithm can be applied only if **the heuristic function is consistent**.

Algorithm The iterative deepening A^* algorithm works as follows

```
cutoff = f(root-node)
REPEAT
    perform depth-first search by expanding all nodes such that
         the cost is less than the cutoff.
    cutoff = min value of f of all the non expanded nodes
UNTIL the goal node is found OR there are no more nodes to expand
```

Performance The iterative deepening A^* algorithm

- Is **complete** if h is admissible.
- Is **optimal** if h is admissible.
- Is more memory efficient than A^* but less time efficient because some states are visited multiple times.

The IDA* reduces the memory requirement because it explores one branch of the tree at a time and when the branch is finished the nodes don't have to be kept in memory.

Chapter 6

Adversarial search

Adversarial search has a different approach with respect to other search strategies, in particular we consider an environment in which there are two agents that compete against each other to reach a goal.

6.1 Model

Environments in which two agents (or in general many agents) compete can be modelled with war games.

Game In particular we will consider games that are

- **Turn based**. The players take turns one after each other and a player can't take two consecutive turns.
- **Zero sum games**. This means that either a player wins the game and the other loses it or both tie. To better understand this property we can assign to each result a value (win=1, lost=-1, tie=0). The sum of the value of each players' result must be 0.
- **Completely visible**. Both agents have perfect information about the state of the environment. In other words the entire state is visible. Every agent also know the state of the other player. For instance chess is a completely visible game, in fact every agent can see the entire environment while poker is not because an agent can't see the opponent's card nor the cards in the deck.
- **Deterministic**. This means that given a state and an action we know the next state.

Players To describe adversarial search we will consider two players MAX and MIN. We will consider MAX's point of view.

MAX always moves first and the goal of both players is to win the game.

Games as search Games can be formulated as search problems, in fact if we consider a certain state s as a node in a search tree we can define the sons of such node as the possible states that can be obtained with an action starting from state s.

The problem with search trees is that when an agent builds the tree it doesn't know what move the opponent will play. To solve this problem we'll have to consider the best move at the current time for the agent that is playing.

6.1.1 Functions

To describe adversarial search problems we need the following functions

- `initial_state()` that returns the initial state of the game.
- `actions(s)` that given a state `s` returns the possible actions that can be done.
- `result(s, a)` that returns the state reached applying action `a` to state `s`.
- `is_goal(s)` that says if the state `s` is a goal state, i.e. a state that ends the execution (either the agent wins, loses or ties).
- `utility(s, p)` that returns the utility value of the final state `s` for player `p`. The utility value represent if a goal state is a winning or losing state for a player.

Utility The utility value `utility(s, p)` represents if a goal state is a winning (e.g. utility=1) or losing (utility=-1) state for a player.

MAX's goal is to maximise the utility (e.g. reach utility +1) while MIN's goal is to minimise the utility (e.g. reach utility -1). Let us better explain why MIN tries to reach -1. Initially we said that the goal of both actors is to win the game but now we say that MIN wants to reach -1 which represents a loss. This happens because both agents operate on the same tree that represents the point of view of MAX, thus MIN wants to reach a state with utility is -1 because it means that the utility for MAX is -1 and MAX has lost the game.

6.1.2 Finding a solution

Goal The goal of each agent is to find the best possible action for every position. This means that every time an agent has to decide what move to do, it builds a tree (actually this process can be optimised) to find the best possible move in the current situation.

Solved games For some games the set of moves to win a game is always known (like in tic-tac-toe). Such games are called **solved games**.

6.2 Minimax

Minimax is a possible implementation of adversarial search. To analyse minimax we'll consider MAX's point of view.

6.2.1 Tree

Every time an agent has to take a decision it has to build a tree. The tree has tree type of nodes

- **MAX nodes**, represented with a triangle. This type of node represents a state from which MAX has to take a decision. In other words its a state after MIN's turn.

- **MIN nodes**, represented with an upside-down triangle. This type of node represents a state in which MIN has to take a decision. In other words its a state after MAX's turn.
- **Leaves**, represented with a square. This nodes represent a goal state, i.e. a state in which the game ends.

Evaluation of a node

For each leaf in the tree we can evaluate it, i.e. decide if such node represents a win or loss state. In particular we have two possible function to evaluate a leaf

- `utility(l)` that precisely defines if a game has been won or lost.
- `evaluation(l)` that returns an estimate of the utility function. This function is used when we can't explore the tree until a leaf node and we have to stop closer to the root. This function can't tell if a certain state is a winning state or not but simply returns how close the state is close to a winning state. For instance the evaluation function is used in chess, in fact it's not possible to to explore all the state to a leaf.

Minimax values The value returned by the `utility` or `evaluation` function is called **minimax**. A minimax can be

- **Positive**. A positive value is good for MAX. If the minimax is obtained using the utility function then MAX has won the game.
- **Negative**. A negative value is good for MIN. If the minimax is obtained using the utility function then MIN has won the game.
- **Zero**. A null value represents a tie.

Back propagation After building the tree, MAX computes the `utility` for each leaf node and it chooses the moves that maximises the minimax value. To maximise the minimax value we need to back-propagate the values from the leaves. In particular

- A MIN node chooses the node with the minimum value among its children. This is because MIN wants to reach the leaf with the minimum minimax value.
- A MAX node chooses the node with the maximum value among its children. This is because MAX wants to reach the leaf with the maximum minimax value.

For instance in Figure 6.1

1. Initially we compute the minimax values of the leaves.
2. The leftmost MIN node chooses the value 0 (the minimum between 1 and 0) while the rightmost node chooses the value -1 ($\min\{-1, 1\} = -1$).
3. The MAX node (i.e. the node that represents the state the agent currently finds itself) chooses the value 0 because its the maximum value between the values of its children. MAX's next action is the action that leads to the node on the left because it's the node that maximises MAX's chances to win.

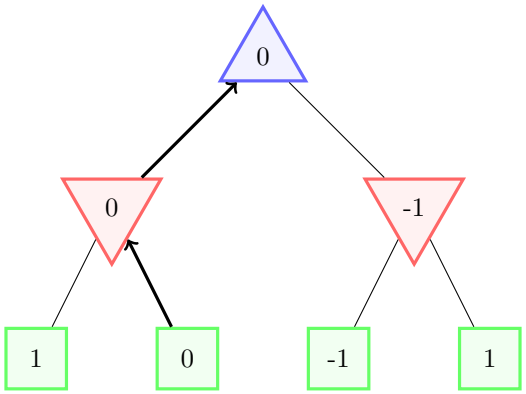

Figure 6.1: A minimax tree.

Exploration The game tree is explored using a depth-first search algorithm to improve space performance, in fact after exploring a branch we can eliminate from memory the nodes already explored.

6.2.2 Performance

Optimality Minimax is optimal when we can use the utility function. Optimality isn't guarantee when we use the evaluation function.

Complexity Minimax uses depth first search thus the complexity of this algorithm is the same of DFS, in particular it can find the next move in

- Linear space
$$\mathcal{O}(bm)$$
- Exponential time
$$\mathcal{O}(b^m)$$

where

- b is the branching factor (i.e. maximum number of branches).
- m is the maximum depth of the tree.

6.2.3 Problems

Minimax allows to play a game but has some problems

- The nodes in the game tree grow exponentially with the maximum depth of the tree, thus even for small games it's not feasible to use this algorithm.
- The algorithm checks every node leaf even if some nodes clearly can't improve the minimax value chosen by a MIN or MAX node.

6.3 Alpha-beta pruning

Alpha-beta pruning (α-β pruning) is a variation of minimax that allows to avoid checking useless nodes (i.e. nodes with a minimax values that can't change the minimax value of a node).

It's important to highlight that this algorithm isn't an approximation of the minimax algorithm, it simply reduces the number of nodes to check.

6.3.1 Alpha and beta

Alpha-beta pruning keeps track of the maximum and minimum values available using two variables α and β. Each node has a couple of α and β values that are propagated to its successors.

Alpha Alpha is the best value found for MAX, i.e. the highest value found for MAX. For instance consider the MAX node in figure 6.1 and consider that all the nodes at the left of MAX have already been visited. In other words consider that the current value of MAX is 0 and the right part hasn't been explored. When the node with value -1 is explored the MIN node on the right chooses -1 (without visiting the other node yet). Because MAX has already chosen 0 and the current value of rightmost MIN is -1, it's not possible to upgrade MAX, in fact MIN can only be upgraded with a value that is smaller of the current value of MIN. Thus the leftmost leaf is not visited. Alpha is initialised with $-\infty$.

Beta Beta is the best value found for MIN, i.e. the lowest value found for MIN. Beta is initialised with $+\infty$.

Search

Node generation When a node is expanded, its children get the α and β values of the parent.

MAX updating A MAX node can only increment its α value and pass both its alpha and beta values to its children when they are created. In particular a MAX node updates the value of α with the value of its successor if the value is bigger than the current α.

```
alpha = max(succ.minimax, alpha)
```

MAX pruning A MAX node prunes all successors not already visited when the minimax of the successor is greater than the value of β.

```
if succ.minimax >= beta then prune
```

This is equivalent to saying that we have to prune when

$$\alpha \geq \beta$$

Intuitively, we prune all other branches if the child's minimax is greater than the smallest minimax (i.e. β) because for sure we can't find a smaller minimax than the one already found.

To better understand why this happens let us consider Figure 6.2. In this case the MAX node on the left at level 2 initially has $\alpha = -\infty$ and $\beta = \infty$ (inherited by MIN at level 1). It explores both its children and discovers that 1 is the best value. This value is passed to the MIN node above that updates its β with 1. The MIN node can now generate the right-hand side MAX and pass the

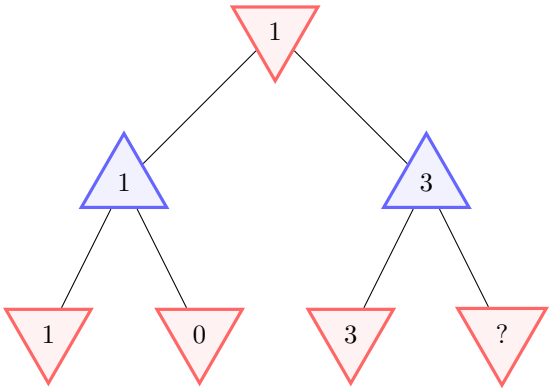

Figure 6.2: A minimax tree.

new values of α and β. MAX explores the left children and updates the value of α with 3. At this point MAX stops because 3 is bigger than β (i.e. the minimum value found by its MIN parent), in fact

- If the right successor was bigger than MAX's current value (e.g. $5 > 3$) then MAX would update its value but MAX's parent wouldn't update because it's looking for the minimum value (and $\beta = 1$ is already smaller than $\alpha = 3$).
- If the right successor was smaller than MAX's current value, then MAX wouldn't update it's value because MAX is looking for the best value.

In both cases exploring the right child is useless because we can't update the parent of MAX.

MIN updating A MIN node can only **decrement** its β value and pass both its alpha and beta values to its children when they are created. In particular a MIN node updates the value of β with the value of its successor if the value is smaller than the current β.

```
beta = min(succ.minimax, beta)
```

MIN pruning A MIN node prunes all successors not already visited when the minimax of the successor is smaller than the value of α.

```
if succ.minimax <= alpha then prune
```

This is equivalent to saying that we have to prune when

$$\beta \leq \alpha$$

Intuitively, we prune all other branches if a child's minimax is smaller than the greater minimax (i.e. α) because for sure we can't find a bigger minimax than the one already found.

6.3.2 Improvement

Alpha-beta pruning allows us to avoid checking some states. Let us analyse how much the minimax algorithm improves when using alpha-beta pruning. The improvement depends on how the minimax values are distributed in the game tree.

Worst case In the worst case scenario no node can be pruned so the algorithm has the same complexity of minimax, in particular

- Linear space complexity
$$\mathcal{O}(bm)$$
- Exponential time complexity
$$\mathcal{O}(b^m)$$

Best case scenario In the best case scenario we have to visit all leaves of one branch and only one leaf for the other branches, thus the time complexity is reduced to

$$\mathcal{O}(b^{\frac{m}{2}})$$

Average case In average the time complexity is reduced to

$$\mathcal{O}(b^{\frac{3}{4}m})$$

6.4 Stochastic games

Initially we considered only deterministic games. Actually it's possible to extend the minimax algorithm to stochastic games, i.e. games in which we relate a probability to each action.

6.4.1 Chance node

To model probability we can use a special node called chance node (represented as a round node). The children of a chance node can be reached with the probability written on the arcs to the children.

Utility The utility of a chance node is the linear combination of the utility of the children and the probability to reach the children. For instance in Figure 6.3 the utility of the rightmost chance node is

$$(0.3 \cdot -1) + (0.7 \cdot 1) = 0.4$$

6.4.2 Alpha-beta pruning

We can use alpha-beta pruning to reduce the number of states visited only if we know the range of values of the utility function, in fact if the range is unknown we cannot know if the utility of a chance node is smaller than α or bigger than β.

6.5 Monte Carlo tree search

The main problem with minimax and alpha-beta pruning is the exponential complexity, furthermore it's difficult to obtain a lower complexity using an evaluation function.

Game tree Monte Carlo tree search (MCTS) solves minimax's problem creating a sparse tree with the most promising nodes. Moreover, MCTS doesn't need a good evaluation function, in fact the algorithm plays (i.e. simulates) the game at random many times and computes the evaluation function using the result of such games.

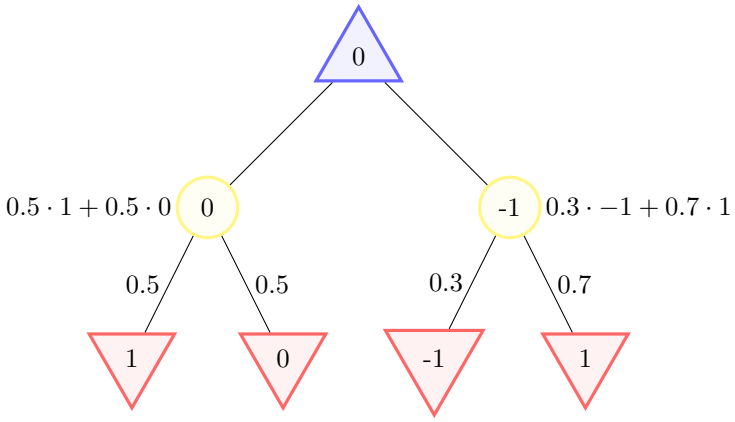

Figure 6.3: A piece of a stochastic minimax tree.

Nodes Every node keeps track of the total games played and the number of times a game has been won. Notice that a node considers only the games that start from its sub-tree.

Algorithm

In particular, Monte Carlo tree search is based on the following steps

1. **Selection**. The algorithm selects the next action to expand. MCTS has to find a balance between new moves and most promising moves already discovered (the same problem of reinforcement learning). To evaluate the next action we can compute the function $UCB1$ (Upper Confidence Bound) for every node n

$$UCB1(n) = \frac{U(n)}{N(n)} + C \cdot \sqrt{\frac{\log(N(Parent(n)))}{N(n)}}$$

   where

   - U is the utility function, i.e. the total number of wins.
   - N is the total number of games played.
   - C is an exploration term.
   - $N(Parent(n))$ is the number of simulations involving the parent of n.

   The node with the highest value of $UCB1$ is selected for expansion. Intuitively, the $UCB1$ function sums the performance of a node n and an exploration term that is bigger if the parent of n was involved in many simulations. Basically, we are encouraging exploration if the parent of a node has been involved in many simulations.

2. **Expansion**. The algorithm generates the node selected and adds it to the game tree.

3. **Simulation**. The algorithm simulates the game starting from the selected state. The simulation doesn't build a tree, in fact it's just a random execution of the game.

4. **Back-propagation**. The result of the simulation is back-propagated to all the nodes in the tree (the numbers U and N are updated). The result of the simulation is called reward and it's used to evaluate a state (a node of the tree).

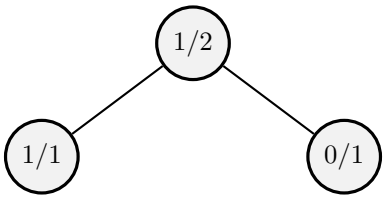

Figure 6.4: The starting search tree used to show a MCTS example.

Anytime method This steps are executed continuously and the process can be stopped whenever we want, for this reason we call this process an **anytime method**. Whenever the MCTS stops it can give an answer.

Advantages

Anytime methods are very powerful and adaptable, in fact we can stop them after an arbitrary number of iterations (even after just one) and it still returns the next move to do. Obviously, the more iterations the better the next move is.

Mobile games This behaviour is very useful in mobile games, in fact in a mobile environment we can decide to run the algorithm for a limited amount of seconds. Whereas, using minimax we should build the entire tree (which may be expensive).

Little known fields This algorithm needs little knowledge about the game (just the basic rules, but no strategies), in fact the evaluation function is derived simulating the game multiple times, thus it can be applied to many fields and to little known games.

Example

To better understand how Monte Carlo tree search works let us show one execution. Say we start from the tree in Figure 6.4.

1. In the **selection** phase the algorithm decides to expand the node on the left of the root.
2. In the **expansion** phase the node is expanded and added to the tree. We obtain the tree in Figure 6.5.
3. In the **simulation** phase a game is simulated multiple times starting from the newly added node. Once the simulation ends the number of wins and the number of games played is saved in the node. The simulation is shown in Figure 6.6 with a triangle.
4. In the **back-propagation** phase the result of the simulations is saved in the node and it's propagated to the root. The final result is shown in Figure 6.7 (say we did one simulation that resulted in a loss).

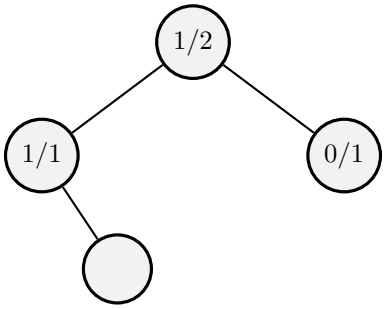

Figure 6.5: The search tree after expanding a node.

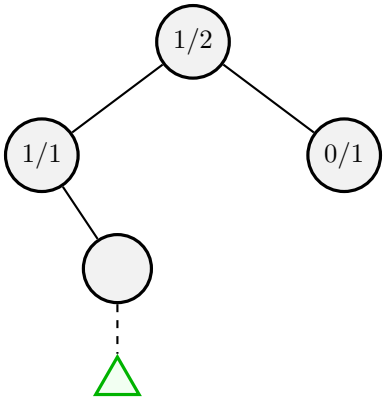

Figure 6.6: The search tree during the simulation phase.

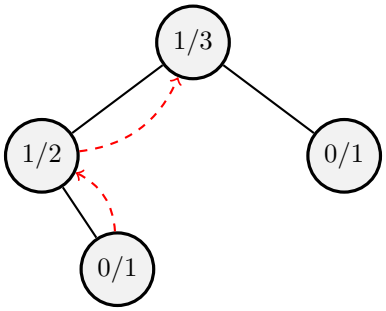

Figure 6.7: The final result of the Monte Carlo tree search execution.

Part III

Learning

Chapter 7

Learning agents

Every agent presented in chapter 2 can learn from experience to improve its performance (which is the way we measure how good an agent is).

Learning allows an agent to operate in unknown environments and helps us design an agent when it's not possible to perfectly model a problem.

When designing an agent it's important to define the measure of performance the agent will use to evaluate its actions once it learned how to behave rather than focusing on how to get the agent to learn how to behave.

7.1 Structure

The structure of a learning agent is an expansion of the one of an agent. In particular a learning agent is made of

- A **critic** element that receives perceptions from the environment and provides feedback on how the agent is doing and determines how the performance element should be modified to improve future performance. The critic element sends feedback to the learning element.
- A **learning element** that improves the performance of the agent using the feedback returned by the critic element. This element exchanges information with the performance element and sends learning goals to the problem generator.
- A **problem generator** that that suggests exploratory actions that will lead to new experiences that could bring knowledge. Such suggestions are sent to the performance element so that the agent can decide to act and follow such suggestions.
- A **performance element** that is the agent structure seen in chapter 3.

The structure of a learning agent is shown in Figure 7.1.

7.2 Machine learning

Definition 7 (Machine learning)**.** *Machine learning is an area of Artificial Intelligence that provides algorithms capable of learning and extracting knowledge from experience.*

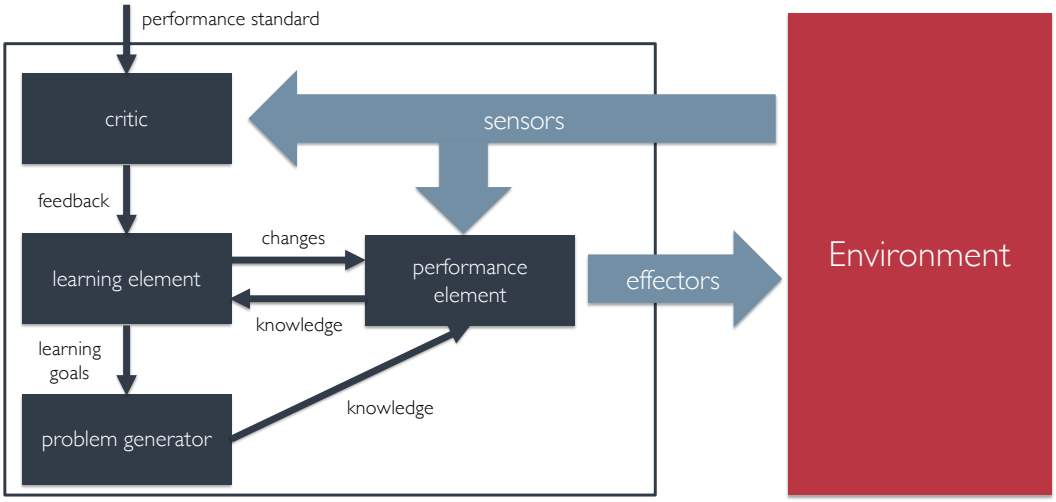

Figure 7.1: The structure of a learning agent.

Difference between Machine Learning and traditional programming In traditional programming we give a machine a set of inputs and a program and the it uses the program to generate some outputs from the given inputs.

```
inputs + program -> outputs
```

In Machine Learning we give a machine a set of inputs and the expected outputs and it generates a program that can transform the inputs in outputs. Such program should perform well with other unknown inputs.

```
inputs + outputs -> program
```

Machine Learning provides three different approaches that differ on how the machine learns

- **Supervised learning** in which we give the machine a set of labelled input the machine uses to learn how to label new inputs.
- **Unsupervised learning** that exploits regularities in the input data to find an hidden patter.
- **Reinforcement learning** that allows a machine to learn given a feedback (reward) on an experience. The machine learns to maximise the long-term reward.

7.2.1 Reinforcement learning

Reinforcement learning allows a machine to learn given a feedback on an experience. If an experience is positive the machine gets rewarded, otherwise it gets penalised (just like we train animals). The machine learns to maximise its long-term reward.

An agent

1. Perceives the environment that is in the state s_t (t being the instant in which the agent perceives the state).
2. Uses the state s_t and what it has previously learned to perform an action a_t on the environment.

3. Receives a reward r_{t+1} and executes the first point to perceive state s_{t+1}. The state s_{t+1} is the result of the action a_t. Basically from now on when receiving the reward we are also sensing the state for the next iteration.

Goal

The goal of the agent is to learn a policy π that maximises the total reward obtained (i.e. not the reward on the single action).

Policy A policy is a function that returns the action that has to be played at each state.

Problems with the reward

Rewarding is a good way to learn but sometimes an agent cannot associate a reward with a certain action or a certain action a_t can result in a negative reward in the short-term but in a very positive reward in the long-term. For instance when we get a medicine it might taste bad (negative reward) but it the long run it heals us (very positive reward). The same concept applies to machines.

The goal of the agent is to maximise the total amount of reward received.

Environment

An agent that learns with reinforcement learning has to operate in a Markov environment.

Definition 8 (Markov environment)**.** *A Markov environment is an environment in which the state of the environment at time $t+1$ (i.e. s_{t+1}) and the reward r_{t+1} only depend on the state at time t, (i.e. s_t) and the action at time t (i.e. a_t).*

This definition doesn't consider history. If we need to consider history we can include the states from $t-1$ backwards in the state s_t.

Markov decision process A Markov environment can be described as a Markov decision process (MDP). A Markov decision process is defined as

$$MDP =< S, A, P, R, \gamma, \mu >$$

in which

- S is the **states space** (i.e. the set of all possible states the environment can find itself).
- A is the **actions space** (i.e. the set of the possible actions that can be done).
- P is the **transition model**. The transition model can be stochastic, this means that given a state s and an action a, the environment can transition to a state s' with a probability p returned by the function P. In particular P is a function

  $$P : S \times A \times S \to [0, 1]$$

  For instance

  $$P(s_1, a_1, a_3) = 0.7$$

  means that if we apply action a_1 to state s_1 we get to state s_3 with probability 0.7.

- R is the **reward** obtained when applying an action a to a state s. In particular R is a function

$$R : S \times A \to \mathbb{R}$$

  Some examples of reward functions are

  - $R = 1$ if the goal is reached, $R = 0$ otherwise.
  - $R = 0$ if the goal is reached, $R = -1$ otherwise.

- γ is the **discount factor** that represents how much we care about the rewards that we will collect in the future with respect to the previous rewards. In particular the bigger this number is, the more we care about future rewards. The discount factor is a number between 0 and 1.

- μ is the function that returns the probability that a state s is **initial**.

$$\mu : S \to [0, 1]$$

Agent

An agent is modelled with a policy π that allows to select the next action to perform.

Deterministic policy A deterministic policy returns the action a that has to be executed in state s

$$\pi : S \to A$$

Notice that, being deterministic, this policy selects always the same action a when in a state s.

Stochastic policy Given a state s and an action a, a stochastic policy

$$\pi : S \times A \to [0, 1]$$

returns the probability p to perform action a in state s. That is

$$\pi(a|s) = p$$

Knowledge of the environment An agent doesn't know the transition model P and the reward function R of the environment, in fact the agent has to interact with the environment to learn how to play (i.e. to learn the function π).

Maximising the reward

The goal of a learning agent is to maximise the total amount of reward received.

Value function For each possible actions the agent calculates how much future reward will it get when it performs such action and then continues to do its best from there on. In other words the agents tries to calculate what is the expected payoff (reward) from state s_t and action a_t.

For each action a_t that can be done from state s_t the agent computes the function

$$Q(s_t, a_t) \to \text{payoff}$$

	a_1	a_2	a_3
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$Q(s_1, a_3)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$Q(s_2, a_3)$
s_3	$Q(s_3, a_1)$	$Q(s_3, a_2)$	$Q(s_3, a_3)$
s_4	$Q(s_4, a_1)$	$Q(s_4, a_2)$	$Q(s_4, a_3)$
s_5	$Q(s_5, a_1)$	$Q(s_5, a_2)$	$Q(s_5, a_3)$

Table 7.1: A Q-table.

that maps a couple state-action to the reward obtained (i.e. Q estimates the reward when the agent performs action a from state s), or the function

$$V(s_t) \rightarrow \text{payoff}$$

that maps a state to the expected payoff. It is important to underline that the functions Q and V are equivalent.

The function Q should be a table with states on the rows and actions on the columns that keeps being upgraded by the agent.

In practice the agent works in complex environments with many variables so it's not feasible to represent Q in a tabular way. Instead we can use an function approximation of Q (created using deep learning).

Discounted reward The total amount of reward G_t received at time t (that's the reason of the t subscript in G_t) is the sum of all the rewards considering the discount factor (the bigger, the more important future rewards are)

$$G_t = R(s_{t+1}, a_{t+1}) + \gamma R(s_{t+2}, a_{t+2}) + \gamma^2 R(s_{t+3}, a_{t+3}) + \cdots = \sum_{i=0}^{\infty} \gamma^i R(s_{t+i+1}, a_{t+i+1}) < \infty$$

The discount factor is introduced because otherwise G would sum to infinity. Let us simplify a bit the equation writing $R(s_{t+i+1}, a_{t+i+1})$ as R_{t+i+1}

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1} < \infty \tag{7.1}$$

The expected reward Q the agent has to maximise at time t is the expected value of the total reward G_t.

$$Q = \mathrm{E}[G_t] = \mathrm{E}\left[\sum_{i=0}^{\infty} \gamma^i R_{t+i+1}\right] \tag{7.2}$$

Bellman Expectation Equation As we can see the reward G_t can written differentiating the immediate reward and the future rewards. In this case the total reward G_t can be written as

$$G_t = R_{t+1} + \sum_{i=1}^{\infty} \gamma^i R_{t+i+1} \tag{7.3}$$

thus the expected reward is

$$Q(s, a) = \mathrm{E}\left[R_{t+1} + \sum_{i=1}^{\infty} \gamma^i R_{t+i+1}\right] \tag{7.4}$$

More explicitly the expected reward $Q(s, a)$ after executing action a in state s is

$$Q(s,a) = \mathrm{E}\Big[R(s_t, a_t) + \sum_{i=1}^{\infty} \gamma^i R_{t+i+1} | a_t = a \wedge s_t = s\Big] \tag{7.5}$$

Equivalently we can use the V function and write

$$V(s) = \mathrm{E}\Big[R(s_t) + \sum_{i=1}^{\infty} \gamma^i R_{t+i+1} | s_t = s\Big] \tag{7.6}$$

Notice that since $\sum_{i=1}^{\infty} \gamma^i R_{t+i+1}$ is the reward expected from the next state (performing the best action possible) we can write $Q(s, a)$ as

$$Q(s,a) = \mathrm{E}\Big[R(s_t, a_t) + \gamma V(s_{t+1}) | a_t = a \wedge s_t = s\Big] \tag{7.7}$$

and $V(s)$ as

$$V(s) = \mathrm{E}\Big[R(s_t) + \gamma V(s_{t+1}) | s_t = s\Big] \tag{7.8}$$

For the final version of Q in Equation 7.7 we can derive the general form of the Bellman equation that considers the probability of all next moves

$$Q(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s,a,s') \cdot \max_{a' \in A} Q(s,a') \tag{7.9}$$

where

- $R(s, a)$ is the **immediate reward**.
- $\sum_{s' \in S} P(s, a, s')$ is the **expectation over the next state**.
- $\max_{a' \in A} Q(s, a')$ is the **best possible action**.

This equation can be approximated for practical uses. In particular the Q function (i.e. the table) is randomly initialised and at every instant t the table is upgraded with the following formula

$$Q(s_t, a_t) \leftarrow (1 - \beta) \cdot Q(s_t, a_t) + \beta \cdot (R(s_t, a_t) + \gamma \max_{a \in A} Q(s_{t+1}, a))$$

in which

- γ is the **discount factor**.
- β is the **learning rate**. Sometimes the learning rate is referred as α.
- $R(s_t, a_t)$ is the **immediate reward**. Notice that the reward considers s_t and a_t even if we have moved to s_{t+1} and the original expected value (Equation 7.3) considered the reward R_{t+1}. For this reason we might expected to use $R(s_{t+1})$ but we would be wrong. To understand why we use s_t we have to remember that we are working in a Markov environment in which the next state and the reward only depend on the current state and action (i.e. s_t and a_t), thus R_{t+1} is a function of s_t and a_t.
- $\max_{a \in A} Q(s_{t+1}, a)$ is the **best possible action** that the actor can do. Choosing the best action means to choose the action with the biggest value considering the row s_{t+1} in the Q-table.

Action selection

At each time step t the agent has to decide what action to take based on the policy function $\pi(s_t)$ that selects an action based on the current state s_t. The policy can be

- **Deterministic** if selects the action with the largest expected payoff.
- **Stochastic** if every action is associated with the probability of being selected.

Exploration-exploitation dilemma

When an agent explores the environment it has two choices

- Keep doing the action that currently maximise the reward.
- Trying to do something new to test if such action may return a bigger reward.

Choosing between these options is important to improve learning. For instance an agent that keeps doing a single action may not consider other actions that might give a bigger reward. On the other hand if the agent only explores the environment it might do actions that give a bad reward. For this reason the agent has find a trade-off between these options.

An agent can use

- A **greedy policy**. For each state the agent deterministically selects an action with the maximal value.
- A **ε-greedy policy**. The agent performs a random action with probability ε and the action with highest payoff with probability $1 - \varepsilon$.

7.2.2 Supervised learning

In supervised learning the machine is given a set of input an the associated desired outputs. The machine learns to predict the correct output given new, unseen inputs.

For instance if the inputs are photos we show the machine which one are photos of apples and which one are photos of oranges and the machine learns how to classify new photos of apples or oranges. In other words we give the machine desired output to allow the machine to predict the future output on different data.

In supervised learning the target values (i.e. the labels) can be

- **Real values**, in this case we have a **regression problem**. For instance we could have a set of (x, y) points and the agent has to assign a value to y given a new unseen value of x. In this case the label (i.e. y) is a real number.
- **Discrete values**, in this case we have a **classification problem**. For instance the apple-oranges photos is a classification model because the labels are apple-photo and orange-photo.

7.2.3 Unsupervised learning

In unsupervised learning the machine exploits regularities in the input data to find an hidden patter. In this case the machine has no desired output to predict and is given no feedback.

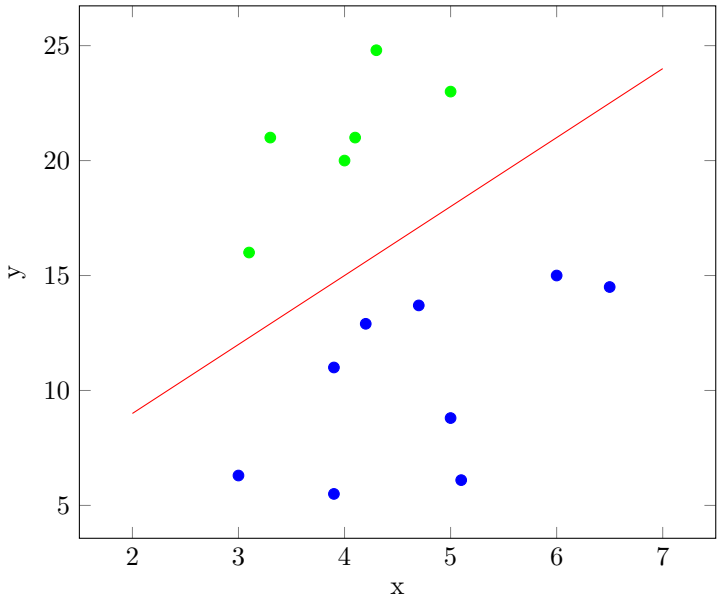

Figure 7.2: An example of classification problem.

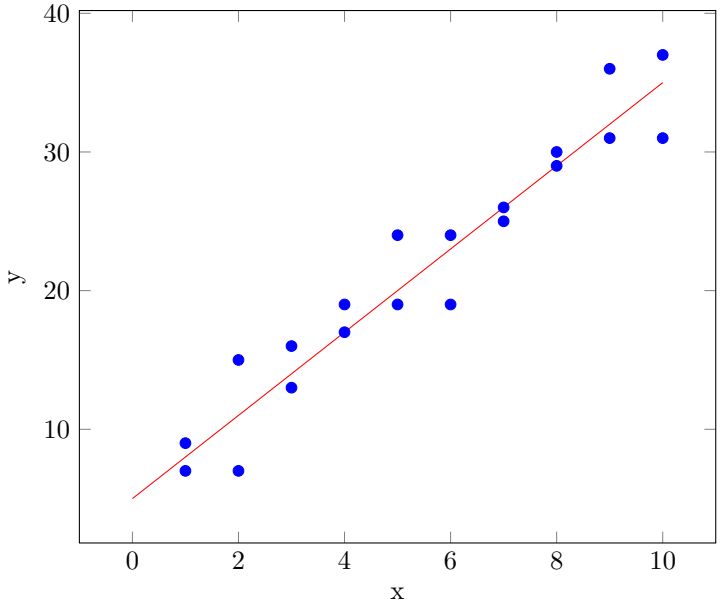

Figure 7.3: An example of regression problem.

Part IV

Constraint satisfaction problems

Chapter 8

Factored representation

Until now we have said that each state corresponds to a node in the search tree, but we haven't specified how such state should be. In fact we haven't specified neither the structure nor the constraints on the state.

This means that all search algorithm considered so far are **agnostic**, that is, don't have any information on the structure of the state. In particular, the search algorithms studied so far only needed to know if any two states were equal to check if a state is in the closed list or if it's a goal state.

Specialisation of the state Agnostic algorithms are rather general but slow. Defining the structure of a state allows us to **improve performance** with respect to agnostic algorithms because we can exploit the structure of the state. This performance improvement comes at a cost, in particular we **reduce the expressiveness** of the algorithms.

8.1 Representation of the problem

A state can be described using a factored representation. In particular a state is represented using a collection of pairs variable-value and a set of constraints.

8.1.1 Constraint satisfaction problem

When an action is performed, it has an effect on the next possible moves, thus it propagates the constraints of a state to the next states. For instance consider the 8 queens puzzle in which we have to place 8 queens on a chess board so that the queens do not attack themselves. In this puzzle, after placing a queen, we reduce the number of cells on which it's possible to place the next queen, thus we have propagated the constraints of a state to the next state.

A problem in which the solution has to satisfy a set of constraints is called constraint satisfaction problem CSP.

Representation of the problem A constraint satisfaction problem can be modelled using

- A set of variables $x_1, ..., x_n$. A variable can also be a vector.

- A set of domains $D_1, ..., D_n$, one for each variable. In particular the domain D_i specifies all the possible values the variable x_i can have. Usually a domain is finite and discrete.
- A set of constraints $C_1, ..., C_m$. Each constraint can be represented as a couple

$$C_i = (S_i, R_i)$$

in which

  - S_i is the scope, i.e. a list of variables that are involved in the constraint.
  - $R_i \subseteq D_i \times \cdots \times D_m$ is the relation, i.e. how the values of the variables in the scope can be combined.

A constraint represents which combinations of variables' values are accepted.

Example To better understand how to model a problem, consider the problem of colouring the regions of Australia (Western Australia, Northern Territory, South Australia, Queensland, New South Wales, Victoria, Tasmania is omitted because it's an island and can be coloured whatever colour) with three different colours (red, green, blue) so that no two adjacent regions have the same colour. In this example the variables are the colour of each region region. Each variable can be coloured in red, green or blue, thus the domain of each variable is $\{R, G, B\}$. Finally we have to define 9 constraints to specify that no two regions can be of the same colour. In particular, we are specifying, given two regions, what colour they can be. For instance consider the first constraint $C(x_{wa}, x_{nt}) = \{...\}$. In this case we are saying that

- If Western Australia is red then Northern Territory has to be blue or green.
- If Western Australia is blue then Northern Territory has to be green or red.
- If Western Australia is green then Northern Territory has to be red or blue.

$$\begin{aligned}
&X = \{x_{wa}, x_{nt}, x_{sa}, x_q, x_{nsw}, x_v\} \\
&D_i = \{R, G, B\} \\
&C(x_{wa}, x_{nt}) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\} \\
&C(x_{wa}, x_{sa}) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\} \\
&C(x_{sa}, x_{nt}) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\} \\
&C(x_{sa}, x_q) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\} \\
&C(x_{sa}, x_{nsw}) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\} \\
&C(x_{sa}, x_v) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\} \\
&C(x_{nsw}, x_v) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\} \\
&C(x_{nsw}, x_q) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\} \\
&C(x_{nt}, x_q) = \{(R,B),(R,G),(B,G),(B,R),(G,R),(G,B)\}
\end{aligned} \tag{8.1}$$

Goal The goal of a CSP is to assign a value to each variable so that all constraints are satisfied. When we assign a value to a variable we use the notation

$$x \leftarrow v$$

or

$$x = v$$

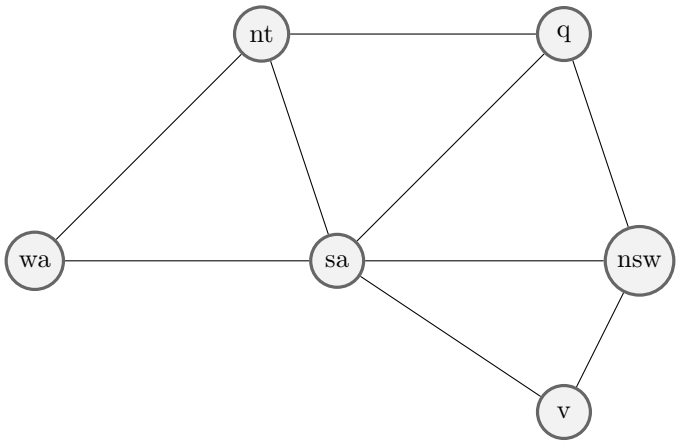

Figure 8.1: A graph representation of Australia and its regions.

Assignment An assignment is

- **Consistent** if all constraints that involve the variable assigned are satisfied.
- **Complete** if all the variables have been assigned.

8.1.2 Variables

Node consistency A variable x_i is node consistent (or 1 consistent) if and only if all values v_i in D_i satisfy all unary constraints.

Arc consistency A variable x_i is arc consistent with respect to variable x_j if and only if all values v_i in D_i satisfy all binary constraints between x_i and x_j.

8.1.3 Domains

8.1.4 Constraints

Binary constraints To describe constraints satisfaction problems we will consider only binary constraints, that is, constraints that involve only two variables.

This simplification can be done without loss of generality, in fact every CSP can be converted to a CSP with binary constraints only.

8.2 Constraints graph

Having considered only binary constraints, we can represent a CSP as a graph in which

- The nodes represent the variables.
- The arcs represent the constraints. In particular an arc (i, j) from node i to node j represents a constraint between x_i and x_j.

Chapter 9

Constraints satisfaction problems as search problems

A constraint satisfaction problem can be seen as a search problem, in fact we can identify each assignment with a state in the search tree. The tree starts with the empty assignment {} and at each action a new variable is assigned so that all the constraints are satisfied. If an assignment can't satisfy any constraint then we have reached a dead end (like a loss situation in adversarial search) and we have to go the the last node in which it's possible to assign another value to a variable.

Example Consider the Australia's map colouring example (the map to colour is shown as a graph in Figure 8.1). The search is the following (the search tree is shown in Figure 9.1)

1. We start with no coloured regions (root node with empty assignment).
2. We colour Western Australia in red (node 2, $\{x_{wa} = R\}$).
3. We colour Northern Territory in green (node 3, $\{x_{wa} = R, x_{nt} = G\}$).
4. We colour Southern Australia in red. This assignment breaks the constraints, in fact the couple $(x_{wa} = R, x_{sa} = R)$ can't be found in the constraints between x_{sa} and x_{wa}.

$$(x_{wa} = R, x_{sa} = R) \notin C(x_{wa}, x_{sa})$$

5. We go back to node 3 and choose another colour for Southern Australia (say blue). This colour satisfies the constraints (node 4, $\{x_{wa} = R, x_{nt} = G, x_{sa} = B\}$).
6. We colour Queensland in green (node 6, $\{x_{wa} = R, x_{nt} = G, x_{sa} = B, x_q = R\}$).
7. We colour New South Wales in green (node 7, $\{x_{wa} = R, x_{nt} = G, x_{sa} = B, x_q = R, x_{nsw} = G\}$).
8. We colour Victoria in green (node 8, $\{x_{wa} = R, x_{nt} = G, x_{sa} = B, x_q = R, , x_{nsw} = G, x_v = R\}$). All variables have been assigned and the assignment satisfies the constraints (i.e. we have obtained a complete and consistent assignment), thus we can stop the algorithm and return the result.

The result of the algorithm (i.e. the solution to the problem) is shown in Figure 9.2.

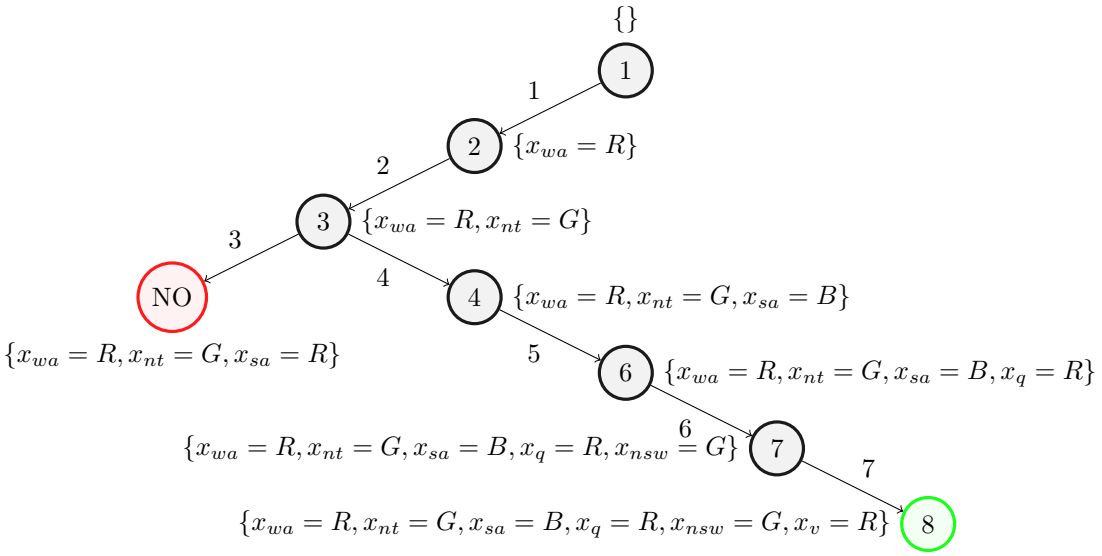

Figure 9.1: Search tree for problem 8.1.

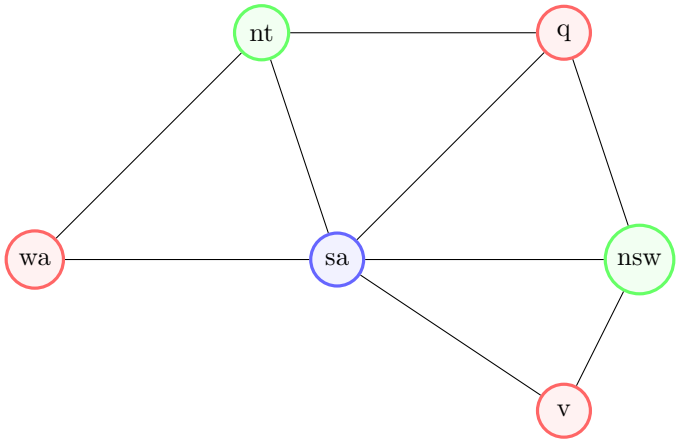

Figure 9.2: A graph representation of Australia and its regions coloured.

9.1 Formal model

To formally define a CSP as a search problem we have to define the main functions of the search problem model

- The **states** S are the consistent assignments.
- The `actions(s)` function returns all the possible assignments that can be done. This means that the assignments returned by `actions(s)` are made of `s` (i.e. the current assignment) plus one other variable not assigned yet.
- The `result(s, a)` function returns the assignment obtained when performing action `a` on assignment `s`.
- The **goal function** checks if all variables have been assignment. In other words if k is the cardinality of an assignment and n is the number of variables of the problem, then a state is a goal when $k = n$.
- The **cost** of each action is unitary.

9.1.1 Solutions

Depth of the tree The maximum depth of the search tree is $n = |X|$, in fact we find a solution when every variable has been assigned. In particular the tree has always a depth of n, in fact we find a solution only when the assignment is complete, thus all solutions are at depth n.

Possible solutions Depending on the order in which we consider the variables and in which decide to assign the values to the variables we can find multiple solutions (i.e. multiple consistent and complete assignments) to the same problem. For instance in the Australia example both

$$\{x_{wa} \leftarrow R, x_{nt} \leftarrow G, x_{sa} \leftarrow B, x_q \leftarrow R, x_{nsw} \leftarrow G, x_v \leftarrow R\}$$

and

$$\{x_{wa} \leftarrow G, x_{nt} \leftarrow R, x_{sa} \leftarrow B, x_q \leftarrow G, x_{nsw} \leftarrow R, x_v \leftarrow G\}$$

are complete and consistent assignments.

In other words the order doesn't impact on finding a solution but only on the specific solution (i.e. the assignment) found.

Commutativity The fact that the order in which variables and values are considered doesn't impact on finding a solution is called commutativity.

9.2 Basic implementation

Building the tree When we build the search tree to solve a CSP we have to consider two actions

- **Choose which variable to assign**. Choosing a variable is represented using a variable (square) node in the search tree.
- **Choose the value to assign to the variable**. Choosing a value is represented using a value (round) node in the search tree.

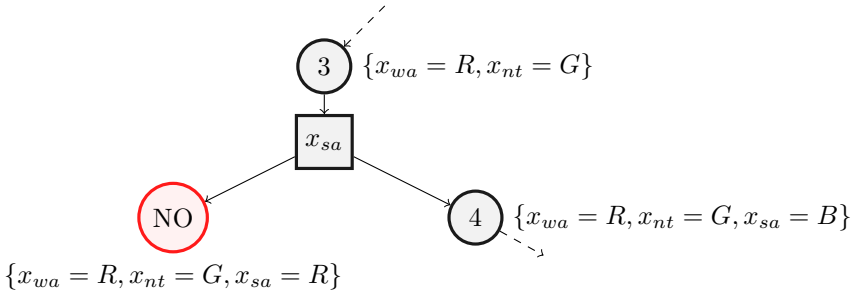

Figure 9.3: Variable and value assignment in a CSP search tree.

To model such actions we can represent the graph using the notation shown in Figure 9.3. The figure shows a sub-tree of the search tree in Figure 9.1. Variable nodes can be omitted if we assign at each level the same variable, in fact in tree 9.1 we can easily understand what variables we are considering for each transition.

Depth first search Because the maximum depth of the search tree is n and no loops can exist then we can implement the search using depth first search with backtracking. This search algorithm has a linear space complexity thus it behaves better that BFS and it always ends its execution because of the maximum depth.

The only difference with DFS is that in this case we use backtracking. This means that when we choose a variable x_i, we don't create all the nodes (i.e. we don't assign the values to x_i) but we generate one node at a time. This is possible because we don't care to reach a specific solution, we just want to reach a solution. When a path reaches a dead end (i.e. the assignment can't satisfy any constraint) than we have to go back to the nearest expandable variable node and try to assign another value (i.e. create a value node).

In our example (Figure 9.1), at level 2, we have chosen variable x_{sa}. When we assign $x_{sa} = R$ we recognise that the assignment isn't consistent, thus we have to go back to the nearest variable node that can be expanded. In our example we go back to x_{sa} and we assign blue.

Pseudocode The pseudocode to solve a CSP using search is shown in Listing 9.1. The algorithm is recursive and takes as only parameter an assignment `A`.

```
1     CSP_backtracking(A):
2         if A is complete: return A
3         x <- choose a variable not in A
4         d <- choose an order of values in the domain of x
5         foreach v in d:
6             add(x <- v) to A
7             if A is consisten:
8                 result <- CSP_backtracking(A)
9                 if result != FAILURE: return result
10            remove(x <- v) from A
11            return FAILURE
12        return FAILURE
```

Listing 9.1: CSP solving algorithm.

9.3 Forward checking

The search algorithm seen so far allow us to find a solution but aren't optimised. The first optimisation we are going to analyse is called forward checking and allows us to eliminate from the domains the values that surely lead to some dead end (i.e. the values that will lead to an inconsistent assignment that can't satisfy any constraint).

Eliminating inconsistent values When a value v is assigned to a variable x we have to consider all the variables y_i not in the assignment and remove the values of y_i's domain that don't satisfy any constraint that involves x and y_i. In other words we have to

1. Assign a value v to x.
2. Consider a variable y not in the assignment.
3. Remove the values of y's domain that don't satisfy any constraint that involves x and y.
4. Repeat point 2 if we haven't already checked all variables.

Empty domain If we keep eliminating some values we might end up with an empty domain. When a domain is empty, we have reached a dead end because there we can't assign a value that satisfies the constraints, thus we have to backtrack.

Parallel algorithm This algorithm works in parallel with respect to the search algorithm.

Pseudocode The presudocode for the forward checking algorithm is the following

```
forward_check(d, x, v, A):
    foreach y not in A:
        foreach constraint c that involves y and x:
            eliminate from d[y] all the values that don't satisfy c
```

The pseudocode to solve a CSP using search and forward checking is shown in Listing 9.2. The new algorithm needs parameter more: the domains `D` of the variables.

```
1   CSP_backtracking(A, D):
2       if A is complete: return A
3       x <- choose a variable not in A
4       d <- choose an order of values in the domain of x
5       foreach v in d:
6           add(x <- v) to A
7           D <- forward_check(d, x, v, A)
8           if no variable has an empty domain:
9               result <- CSP_backtracking(A, d)
10              if result != FAILURE: return result
11          remove(x <- v) from A
12          return FAILURE
13      return FAILURE
```

Listing 9.2: CSP solving algorithm with forward checking.

9.4 Next variable optimisation

An important part of the CSP solving algorithm is choosing the next variable to expand. Until now we have chosen a variable at random but there exist methods that allow us to choose a variable to reduce the number of nodes expanded. Such methods are heuristics. In particular we are going to analyse

- The **minimum remaining value** heuristic.
- The **degree** heuristic.

Solution These methods affect only the performance of the algorithm in fact, as we have already said, optimisations have no impact on finding the solution because of commutativity.

9.4.1 Minimum Remaining Value

The Minimum Remaining Value (MRV) heuristic suggests to select the variable with the smallest number of consistent values remaining in the domain.

Example To better understand MRV consider the following CSP

$$
\begin{aligned}
&X = \{x_1, x_2\} \\
&D_1 = \{1, 2, 3, 4, 5\} \\
&D_2 = \{\alpha, \beta\} \\
&C(x_1, x_2) = \{\cdots\}
\end{aligned}
$$

In this case we should expand x_2 because the domain D_2 (i.e. the domain of x_2) has less elements than the domain of x_1. Namely x_2 has less consistent values.

9.4.2 Degree heuristic

The degree heuristic suggests to choose the variable involved in the largest number of constraints with unassigned variables. This heuristic is less powerful than MRV, thus it can be used to break ties among variables with same Minimum Remaining Value.

Example To better understand the degree heuristic consider the following CSP

$$
\begin{aligned}
&X = \{x_1, x_2, x_3, x_4\} \\
&D_1 = \{1, 2\} \\
&D_2 = \{\alpha, \beta\} \\
&D_3 = \{a, b\} \\
&D_4 = \{(,), [\} \\
&C(x_1, x_2) = \{\cdots\} \\
&C(x_1, x_3) = \{\cdots\} \\
&A = \{(x_4 \leftarrow [)\}
\end{aligned}
$$

In this example the variables not assigned are x_1, x_2 and x_3. Variable x_1 is involved in 2 constraints (with x_2 and x_3) but x_2 and x_3 only appear in only one constraint (both with x_1), thus we have to choose variable x_1 because it appears in more constraints than the other two.

9.5 Least Constraining Value

Another important decision that has to be made is the order in which values are assigned.

```
d <- choose an order of values in the domain of x
```

To decide such order we can use the Least Constraining Value (LCV) heuristic. This method suggests to choose the value that leaves more freedom to other unassigned variables. In other words we have to **choose the values that removes the fewer number of values from the domains**.

Solution Like for variable selection heuristics, these methods affect only the performance of the algorithm in fact optimisations have no impact on finding the solution because of commutativity.

Example To better understand LCV consider the Australia's map example (Figure 8.1) with the following assignment

$$A = \{(x_{wa} \leftarrow R), (x_{nt} \leftarrow G)\}$$

In this case if we assign blue to x_q then the domain of x_{sa} is empty $D_{sa} = \{\}$, but if we assign red to x_{sa} then we obtain a domain $D_{sa} = \{B\}$, thus it's better to choose red over blue.

9.6 Arc consistency

Arc consistency allows us to further improve performance and discover dead ends earlier.

Assumptions The arc consistency method works under two assumptions

- The CSP only has binary constraints. This is not a problem because every CSP can be reduced to a CSP with binary constraints.
- The constraints graph is directed. This is not a problem because an undirected graph can be transformed in a directed graph (transform an arc that connects i and j in two arcs i, j and j, i).

Algorithm Arc consistency reduces the domains as follows.

1. Consider arc (y, x).
2. For every value v in D_y if there exists a consistent value in D_x then keep v. Else delete v from D_y because v is inconsistent.
3. Repeat instruction 1 for the next arc.

In other words we delete all values of the source node that are not consistent with the domain of the destination node. The same procedure can be written in a more algorithmic way as follows

```
arc_consistency(D):
    foreach (y, x) in arcs:
        foreach v in D[y]:
            if v not consistent with D[x]:
                D[y] = D[y] - v
```

Arc consistency in forward checking Arc consistency can be used to implement forward checking, in fact forward checking aims to remove the inconsistent values in the domains. Notice that, given a constraint $C(x, y)$,

- Arc consistency removes the values from D_x.
- Forward checking removes the values from D_y.

Pseudocode The algorithm is executed before all instructions of the CSP solving algorithm. The final version of the CSP solving algorithm is shown in listing 9.3.

```
1   CSP_backtracking(A, D):
2       if A is complete: return A
3       arc_consistency(D)
4       if a variable has an empty domain: return FAILURE
5       x <- choose a variable not in A
6       d <- choose an order of values in the domain of x
7       foreach v in d:
8           add(x <- v) to A
9           D <- forward_check(d, x, v, A)
10          if no variable has an empty domain:
11              result <- CSP_backtracking(A, d)
12              if result != FAILURE: return result
13          remove(x <- v) from A
14          return FAILURE
15      return FAILURE
```

Listing 9.3: Complete CSP solving algorithm.

9.6.1 AC-3

Arc consistency can also be used to find a solution to a CSP without searching algorithm. In particular we are going to consider AC-3, an algorithm that iterates arc consistency multiple times.

Inference procedure AC-3 is an inference procedure, in fact no search is required.

Algorithm AC-3 works as follows

1. A queue Q is initialised with all the constraints of the problem. The constraints are considered in both directions, this means that both (x, y) and (y, x) are inserted. Namely we are considering the directed arcs of the constraints graph.
2. If the queue Q is empty, end execution.
3. Pop a couple (x, y) from the queue.
4. Apply arc consistency to (x, y). This means that we have to remove all the values v_x of x that are not consistent with the domain of y.
5. If some elements have been removed
   - If the domain of x is empty then stop execution with an error.
   - If the domain of x is not empty then insert (if not already present) all constraints $\{(z, x) : z \in X, z \neq y, x\}$ in the queue.
6. Repeat from instruction 2.

Pseudocode AC-3 needs the constraints `C` and the domains `D`.

```
// removes from the domain of x the values that are inconsistent  with the values
 in the domain of y
arc_consistency(x, y):
    removed <- false
    foreach v in D[x]:
            if v not consistent with any value of D[x]:
                D[y] = D[y] - v
                removed <- true
    return removed

AC-3(C, D):
    Q = set_queue()
    while Q not empty:
        (x, y) <- pop(Q)
        if arc_consistency(x, y):
            if D[y] is empty: return FAILURE
            forall (z, x) in C: Q = Q + (z, x)
```

Part V

Logical agents

Chapter 10

Introduction

Logical agents are agents that reason in a logical way. A logical agent represents a state in which it can be using

- **Objects**.
- **Statements** that hold in the world.

In particular logical statements

- Represent the knowledge in states and actions.
- Exploit the knowledge to decide the actions to do.

Logical sentence A logical sentence is a phrase expressed in a formal language (like propositional logic or first order logic).

10.1 Structure

The knowledge of a logical agent is stored in a structure called **knowledge base** (KB). In particular a KB is a list of logical sentences. The list has to be interpreted as a logical conjunction (AND), this means that all the sentences in the KB have to be true at the same time. This approach is also called **declarative approach**.

10.1.1 Operations

We can execute two operations on a knowledge base

- **Tell**. The tell operation allows to insert knowledge (i.e. some sentences) in the KB.
- **Ask**. The ask operation (also query operation) allows to question the KB and obtain some answers that imply reasoning.

Incremental knowledge As we can see, there is no operation to remove some knowledge, in fact we can only add sentences. This means that the new knowledge added has to derive from the knowledge already present in the KB.

Time One may think that incremental knowledge doesn't allow statements to change. In particular, we could have added a statement that at one point in time was true but after some time became false. This problem is solved adding the time in which a statement is true so that a sentence A can be true at time t but false at time $t + 1$.

This is very expensive because the KB consumes a lot of memory thus it isn't used in practice.

Search tree Sentences in a knowledge base can be organised in a tree in which

- Every node represents a sentence
- An arc from a parent node p and its child c means that sentence c can be derived from p.

Using a tree allow us to use search when we have to answer a question using reasoning.

10.1.2 State

A state of a logical agent is represented as

- A **knowledge base**.
- An **inference engine**, i.e. an algorithm that implements the ask operation. The inference engine is a generic algorithm, in fact it isn't specific to the domain of application of the agent.

10.1.3 Next action

A logical agent chooses the next action to do using reasoning, in particular it performs an ask operation on its knowledge base. The reasoning process of a logical agent can be described as follows

1. The agent makes a perception P.
2. The agent updates the knowledge base with P (i.e. it executes a tell operation).
3. The agent asks the knowledge base what action to perform via the ask operation.
4. The agent tells the knowledge base the performed action.

10.2 Advantages

This model has many advantages with respect to other models. In particular

- It is useful to have agents that are represented using what they know. If we consider search, the knowledge is embedded in the `actions()` function, thus the search model is less portable. In particular, an implementation of the `actions()` function can't be reused for another application because different applications require different knowledge.
- The logical model is more economic, in fact logical agents can use a small set of sentences from which it's possible to derive other sentences. Furthermore parent sentences are implicit assumption of child sentences. Such assumptions can be seen through the ask operation.
- Given the available knowledge, logical agents can answer all answerable questions.

Chapter 11

Logic

11.1 Propositional logic

Before digging into propositional logic let's have a look at some definitions.

Fact A fact is something that holds in a certain time in the world. A fact is detected by the agent's sensors.

Rule A rule is a statement given by the designers or learned by the machine.

11.1.1 Rules

Propositional logic associate a symbol to each simple fact (i.e. to each statement). The symbols can be combined using logic operators.

For instance the statement *if the semaphore is green and the car is on then the car crosses the crossroad* can be represented using the symbols

- G = the semaphore is green.
- O = the car's engine is on.
- C = the car crosses the crossroad.

and the statement can be written as

$$G \wedge O \Rightarrow C$$

Objects Propositional logic doesn't allow us to represent an object but only statements about an object that can be either true or false. In particular there is no state in between true and false.

11.1.2 Syntax

Propositional logic's syntax can be defined recursively

- Propositional symbols P are sentences.
- If α is a sentence then $\neg\alpha$ is a sentence.

- If α and β are sentences then
  - $\alpha \wedge \beta$
  - $\alpha \vee \beta$
  - $\alpha \Rightarrow \beta$
  - $\alpha \iff \beta$

  are sentences.
- Nothing else is a sentence.

11.1.3 Semantics

Model Semantics in propositional logic is defined by a model. In particular, a model is a truth assignment (true or false) for each propositional symbol.

11.1.4 Representation

A sentence in propositional logic can be represented using a tree. In particular

- Every non-leaf node represents a logical operator.
- Every leaf represents a propositional symbol.

For instance the statement $A \wedge B \Rightarrow C$ is represented in Figure 11.1

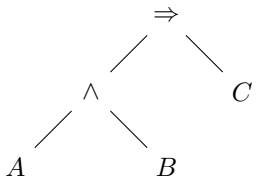

Figure 11.1: The tree representing $A \wedge B \Rightarrow C$

11.1.5 Advantages and disadvantages

Pros

Propositional logic has many advantages

- It's declarative, in fact it explicitly represents a piece of knowledge.
- It allow partial, disjunctive and negated information.
- It is compositional, in fact the meaning of a composition of symbols is derived from the meaning of the single symbols. In other words the meaning of α depends on the components of α. For instance the meaning of $P \wedge Q$ is defined by the meaning of the symbols P, Q.
- The meaning of a sentence is independent from the context.

Cons

The main disadvantage of propositional logic is that it has a very limited expressive power, in fact it's not possible to express sentences with quantifiers using a single sentence. For instance to express the sentence *For each cell in a grid, it has 8 neighbours* we have to write a different sentence (i.e. cell x, y has 8 neighbours) for each cell. Ideally we would like to write a single sentence.

11.2 First order logic

Objects In first order logic, opposite to propositional logic, it's possible to define objects directly or using functions (that return objects).

11.2.1 Syntax

In first order logic we can use

- Constant symbols $A, B, C, ...$
- Variables $a, b, c, ...$
- Predicate symbols $\mathcal{A}, \mathcal{B}, \mathcal{C}, ...$
- Functions that take predicate symbols in input $f(), \phi(), ...$.
- Connectives $\wedge, \vee, \neg$.
- An equality symbol $=$.
- Universal and existential quantifiers $\forall$, $\exists$.

Definition 9 (Term)**.** *A term identifies an object and can be defined in the following way*

- *A* ***constant*** *is a term.*
- *A* ***variable*** *is a term.*
- *If $t_1, ..., t_n$ are terms then a* ***function*** *$f(t_1, ..., t_n)$ is a term.*

Atomic sentence Let $\mathcal{A}$ be a predicative symbol and $t_1, ..., t_n$ be terms then

$$\mathcal{A}(t_1, ..., t_n)$$

is an atomic sentence.

Sentence A sentence can be defined recursively

- An atomic sentence is a sentence.
- If α is a sentence then
  - $\neg\alpha$
  - $\forall\alpha$
  - $\exists\alpha$

are sentences.

- If α and β are sentences, then
  - $\alpha \wedge \beta$
  - $\alpha \vee \beta$
  - $\alpha \Rightarrow \beta$
  - $\alpha \iff \beta$

  are sentences.

Quantifiers

Universal quantifier The universal quantifier $\forall$ is typically used in the following form

$$\forall x \quad A(x) \Rightarrow B(x)$$

Beware that the sentence

$$\forall x \quad A(x) \wedge B(x)$$

isn't equivalent to the former one.

Existence quantifier The existence quantifier $\exists$ is typically used in the following form

$$\exists x \quad A(x) \wedge B(x)$$

Beware that the sentence

$$\exists x \quad A(x) \Rightarrow B(x)$$

isn't equivalent to the former one.

Properties Here's a list of quantifiers' properties

- Quantifiers of the same type can be swapped. In particular

$$\forall y \forall x \equiv \forall x \forall y \qquad \exists y \exists x \equiv \exists x \exists y$$

- Different quantifiers can't be swapped. In particular

$$\forall y \exists x \not\equiv \exists x \forall y$$

Transformations It's possible to transform a sentence with a quantifier using the following rules

- $\forall x \;\; A(x) \equiv \nexists x \;\; \neg A(x)$
- $\exists x \;\; A(x) \equiv \neg(\forall x \;\; \neg A(x))$

Equality

First order logic introduces an equality symbol. A term t_1 is equal to another term t_2 if both terms refer to the same object in the world.

11.2.2 Semantic

The semantic of sentence in first order logic is more complex, in fact we have to define the concept of

- **Model**. A model is made of the actual entities of the world.
- **Interpretation**. An interpretation connects the symbols in a sentence to the concepts of the real worlds. In particular
  - Constant symbols are associated to objects in the real world.
  - Predicate symbols are associated to relations in the real world.
  - Function symbols are associated to functions in the real world.

A sentence $predicate(term_1, \ldots, term_n)$ is true when the objects in the real world referred by $term_1, \ldots, term_n$ are in the relation referred by the predicate.

Chapter 12

Inference engines

After describing how to define the knowledge base of a rational agent using propositional logic or first order logic we should focus on the inference engine. An inference engine is an algorithm that, independently from the domain of application, allows to question the knowledge base and obtain some answers that imply reasoning.

Inference engines will be analysed considering propositional logic (i.e. the sentences of the knowledge base are written in PL).

12.1 Inference

Before describing the algorithms of an inference engine we need some definitions.

Entailment

Given two sentences α and β we say that α entails β (or that β logically follows α) and we write

$$\alpha \models \beta$$

if in every world in which α is true, β is also true. This means that β is true in more worlds (i.e. in more models) than α.

$$\text{worlds}(\alpha) \subseteq \text{worlds}(\beta)$$

Intuitively if β is a consequence of α (i.e. β follows α) then every time α is true, β also have to be true.

Proof

A proof is a demonstration that we can derive a sentence β from another sentence α using an algorithm A. If we can obtain β from α using A, then we write

$$\alpha \vdash_A \beta$$

Proofs are built by algorithms using symbolic manipulation.

A proof can be obtained using

- **Model checking**. Model checking verifies for every model that if α is true than also β is true.
- **Theorem proving**. Theorem proving uses a sequence of proof steps.

Properties Here's a list of algorithm's properties

- **Soundness**. If an algorithm proves β starting from α then α has to entail β. In other words everything the algorithm claims to prove is in fact entailed.

$$\alpha \vdash_A \beta \Rightarrow \alpha \models \beta$$

- **Completeness**. Everything entailed can be proven.

$$\alpha \models \beta \Rightarrow \alpha \vdash_A \beta$$

Between these two properties the most important is the first one (i.e. we have to ensure soundness and we can relax completeness).

Validity

A sentence α is valid if it's true in all models.

Deduction theorem The deduction theorem states that a sentence α follows the knowledge base if and only if when the sentences in the KB are true then α is true.

$$KB \models \alpha \iff KB \Rightarrow \alpha \text{ is valid}$$

Satisfiability

A sentence α is **satisfiable** if it's true in some models (i.e. in at least one model). On the other hand a sentence α is **unsatisfiable** if there is no model in which it's true.

Refutation Refutation states that a sentence α follows the knowledge base KB if and only if the conjunction of the KB and the negation of α is unsatisfiable.

$$KB \models \alpha \iff KB \wedge \neg\alpha \text{ unsatisfiable}$$

Intuitively $A \models B \iff A \wedge B$ unsatisfiable because if we can't find a model that satisfies both A and $\neg B$, then there must exist a model that satisfies both A and B, thus B is true when A is true.

12.1.1 Inference procedures

An inference procedure is an algorithm A that can demonstrate the entailment between a formula α and a formula β (i.e. A demonstrate $\alpha \models \beta$).

Inference procedures can be divided in two big families

- **Model checking** procedures. Given α and β, model checkers try to verify that when α is true, β is also true. In other words these procedures try to understand if the entailment holds or not.
- **Theorem proving** procedures. Theorem provers try to build a logical proof that demonstrates that we can obtain β from α. In particular these procedures try to transform α in β (i.e. try to build β given α).

12.2 Model-checking inference procedures

12.2.1 Truth tables

The first model-checking procedure we are going to describe is based on truth tables. This method simply verifies the definition of entailment, in fact it checks that every true model for α is also true for β. This means that we have to enumerate every model and for each model verify that if the model is true for α, it is also true for β.

Since every propositional symbol can take two values (true and false), we can generate 2^n models, where n is the number of different propositional symbols. All the models can be organised in a table where each row represents a model. The column of the table contain the truth values for the statements α and β and eventual intermediate statements (if $\alpha = a \wedge b \Rightarrow c$ we can have two columns $a \wedge b$ and α to better organise the algorithm). An example of truth table is shown in Table 12.1.

	α	β
model_1	true	true
model_2	false	false
model_3	false	false
model_4	true	true
model_5	false	false
model_6	false	false
model_7	false	false
model_8	true	true

Table 12.1: A truth table for which $\alpha \models \beta$.

Example

To better understand how truth tables work we can consider the following statements

a. When it rains and it's windy, Alice wears a coat.

b. It rains but Alice doesn't wear a coat.

c. It's not windy

We can use the following propositional symbols to translate the aforementioned statements

- R to say that it's raining.
- W to say that it's windy.
- RCA to say that Alice wears a raincoat.

The statements in natural language can be translated in propositional logic as follows

a. $R \wedge W \Rightarrow RCA$

b. $R \wedge \neg RCA$

c. $\neg W$

Given the statements just translated we want to prove that a and b entails c

$$\{a, b\} \models c$$

The truth table used to prove the entailment is shown in Table 12.2. In the table the models are shown in red, the statements a and b are shown in green and statement c is in light blue. As we can see in the table, there exists only a model in which a and b are true (i.e. $R = 1, W = 0, RCA = 0$). For this model, statement c is true, thus we have proven that for each true model of $\{a, b\}$, also c is true and $\{a, b\} \models c$

R	W	RCA	$R \wedge W$	$R \wedge W \Rightarrow RCA$	$\neg RCA$	$R \wedge \neg RCA$	$\neg W$
0	0	0	0	1	1	0	1
0	0	1	0	1	0	0	1
0	1	0	0	1	1	0	0
0	1	1	0	1	0	0	0
1	0	0	0	1	1	1	1
1	0	1	0	1	0	0	1
1	1	0	1	0	1	1	0
1	1	1	1	1	0	0	0

Table 12.2: The truth table for the example.

Advantages and disadvantages

The main disadvantage of this procedure is that it doesn't scale up, in fact if we consider a problem that requires 100 propositional variables, we would need a table with 2^{100} rows, which is too big to instantiate.

Another disadvantage of this procedure is that it produces an hard to interpret answer by humans. This happens because the reasoning process behind this method is very far from how humans think. Currently this is becoming a problem because we would like agents to provide an explanation and justify the reasoning process.

The main advantage of this method is that it's very simple to implement and to understand.

Properties

The truth table procedure is **sound** and **complete**, in fact it applies the definition of entailment. Furthermore, if the problem is described by a finite number of propositional symbols, then the table has a finite number of rows and the algorithm always terminates (even if it might take a very long time because the number of rows might be very large). Notice that the fact that this algorithm always terminates implies that propositional logic is **decidable**.

12.2.2 Propositional satisfiability - DPLL

Satisfiability problem

Propositional satisfiability solves the main issues of truth tables. Propositional satisfiability is based on the problem of satisfiability, that is, given a complex formula ϕ we have to say if ϕ is satisfiable

(i.e. can we find a model for ϕ). This problem seems quite trivial but many real world problems can be translated and reduced to this problem, thus finding efficient algorithms to solve it is very important and useful. In general the problem of satisfiability is applied to a finite set ϕ of formulas ϕ_i in conjunction

$$\phi = \{\phi_1, \phi_2, \ldots, \phi_n\} \text{ true} \iff \phi_1 \wedge \phi_2 \wedge \ldots, \wedge \phi_n \text{ true}$$

The problem of satisfiability is also called **SATProblem**. Notice that a trivial way to solve the SATProblem is to use truth tables.

DPLL

DPLL (Davis-Putnam-Logemann-Loveland) is an efficient algorithm to solve the satisfiability problem (i.e. the goal of the algorithm is to find a model for a formula α or to answer that no model for α exists). DPLL works in two steps

1. In the first step, which is a preprocessing step, the formula α is converted in Conjunctive Normal Form (CNF for short) which is a special representation of α.
2. In the second step the actual algorithm is executed. The algorithm tries to build a model for α in an incremental way starting from the empty assignment. If the algorithm can build a model then α is satisfiable, otherwise if it ends without having generated a model α is unsatisfiable. In both cases the algorithm terminates.

DPLL can be applied only to Propositional Logic, thus First Order Logic predicates have to be translated in Propositional Logic.

Conjunctive normal form

CNF is a representation of a sentence. In particular CNF allows to write a formula α as a conjunction (i.e. $\wedge$) of clauses. A clause can be defined recursively, in fact

1. A **clause** is a disjunction of literals. A clause that is composed of just one literal is called **unit clause**.
2. A **literal** is a propositional symbol or the negation of a propositional symbol. Two literals are **complementary** if they refer to the same propositional symbol S but it is negated in one literal and not negated in the other (e.g. C and $\neg C$ are complementary).

An example of clause is shown in Equation 12.1.

$$\neg A \vee B \vee C \tag{12.1}$$

A clause can also be represented using sets (and it's implicit that the literals in the set are in disjunction)

$$\{\neg A, B, C\} \tag{12.2}$$

The formula ϕ

$$\phi : (A \Rightarrow B \vee C) \wedge (C \Rightarrow \neg D) \tag{12.3}$$

can be transformed in CNF as

$$\phi : (\neg A \vee B \vee C) \wedge (\neg C \vee \neg D) \tag{12.4}$$

and can be represented as

$$\phi : \{\neg A, B, C\}, \{\neg C, \neg D\} \tag{12.5}$$

Transforming a formula in CNF A formula can be converted in CNF form using the following steps (consider $A \iff (B \vee C)$ as an example).

1. Eliminate the biconditional $\iff$ using the following transformation

   $$\alpha \iff \beta \text{ becomes } (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$$

   Our example becomes

   $$(A \Rightarrow (B \vee C)) \wedge ((B \vee C) \Rightarrow A)$$

2. Eliminate the implications $\Rightarrow$ using the following transformation

   $$\alpha \Rightarrow \beta \text{ becomes } \neg\alpha \vee \beta$$

   Applying this transformation to our example becomes we obtain

   $$(\neg A \vee B \vee C) \wedge (\neg(B \vee C) \vee A)$$

3. Move negations $\neg$ inwards using De Morgan rules, in particular

   $$\neg(\alpha \vee \beta) \text{ is equivalent to } \neg\alpha \wedge \neg\beta$$

   and

   $$\neg(\alpha \wedge \beta) \text{ is equivalent to } \neg\alpha \vee \neg\beta$$

   If we apply De Morgan's rules to our example we obtain

   $$(\neg A \vee B \vee C) \wedge ((\neg B \wedge \neg C) \vee A)$$

4. Apply the distributivity law, namely we have to distribute the and $\wedge$ over the or $\vee$. To help understand how to apply this rule we can think at the and like a multiplication and the or like a sum. Applying the distributivity law allows us to obtain

   $$(\neg A \vee B \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A)$$

   Notice that both $\wedge$ and $\vee$ can be distributed. In particular

   - $A \wedge (B \vee C)$ becomes $(A \wedge B) \vee (A \wedge C)$.
   - $A \vee (B \wedge C)$ becomes $(A \vee B) \wedge (A \vee C)$.

After applying this steps we obtain a formula in Conjunctive Normal Form.

Algorithm

The actual procedure that tries to find a model for a formula is a **backtracking depth-first** algorithm that searches over the models (i.e. the models are the nodes of the tree). The algorithm adds some concept to classical depth-first search, in particular we can include the following concepts

- **Early termination** allows the algorithm to stop when
  - It finds a **partial model** for which all the clauses are true. For instance if we are trying to find a model for $(A \vee C) \wedge (A \vee \neg B)$ and we have already found the partial model $\{A = 1\}$, then we can stop because if we replace $A = 1$ both clauses are surely true independently from the values of B and C.

  - It finds a **partial model** for which at least one clause is false. This is because the clauses are in conjunction, thus if one is false the conjunction is false. For instance if we are trying to find a model for $(A \vee C) \wedge (A \vee \neg B)$ and we have already found the partial model $\{A = 0, C = 0\}$, then we can stop because if we replace $A = 0$ and $C = 0$ the first clause is surely false independently from the value of B, thus the whole formula is surely false.

- **Pure literals**. If a literal appears in a formula always with the same sign then it is called **pure**. If a literal A is pure and positive then there surely exist a model in which $A = 1$, in fact if $A = 1$ the clauses in which it appears are surely true because a clause is a disjunction of literals, thus we only need one true literal to consider the clause true. Notice that A doesn't necessarily need to appear in all clauses. For instance in $(A \vee B) \wedge (A \vee \neg C) \wedge (C \vee \neg B)$, the literal A is pure and positive but C is not pure (because it is negative in the second clause and positive in the third).

- **Unit clauses**. If a clause is made by a single literal then we must build a model to satisfy such clause. We have to satisfy this clause because otherwise the formula wouldn't be true, in fact clauses are in conjunction and if one clause if false then the whole conjunction is false. For instance if we have a clause $\neg C$ we must add $C = 0$ to the model.

- **Splitting**. If no other rule is applicable we have to guess a value (true or false) for a literal. If the assigned value doesn't allow us to build a model then we should try the other value. This operation is a branch in the search tree (and it's explored using depth-first). If we have to prove unsatisfiability (instead of satisfiability), all branches must be analysed and must be unsatisfiable.

This rules can be applied iteratively until no rule can be applied anymore, in fact after applying a rule it might happen that another rule might be applicable. For instance if we consider the following example

$$(A \vee B) \wedge (A \vee \neg C)$$

we can

1. Identify B as pure literal an apply rule 2 to obtain a model $\{B = 1\}$. The formula can be rewritten as $(A \vee \neg C)$.
2. A is a pure literal thus we can apply the second rule again and obtain $\{A = 1, B = 1\}$.
3. All clauses are true, thus we can stop early and return the model $A = 1, B = 1, C = any$.

Code PDLL can be written as in Listing 12.1

```
function DPLL(clauses, symbols, model) returns true or false
    if ( every clause in clauses is true in model ) then return true
    if ( some clause in clauses is false in model ) then return false
    P, value <- FIND_PURE_SYMBOL(symbols, clauses, model)
    if ( P is non_null ) then return DPLL(clauses, symbols - P, model + {P=value})
    P, value <- FIND_UNIT_CLAUSE(clauses, model)
    if ( P is non-null ) then return DPLL(clauses, symbols - P, model + {P=value})
    P <- First(symbols)
    rest <- Rest(symbols)
    return or(DPLL(clauses, rest, model + {P=true}),
              DPLL(clauses, rest, model + {P=false}))
```

Listing 12.1: The code for DPLL.

Entailment

SATSolvers can be used to check entailment. In particular, we have to remember the refutation theorem. It states that a sentence α entails another sentence β if $\alpha \wedge \neg\beta$ is unsatisfiable

$$\alpha \models \beta \iff \alpha \wedge \neg\beta \text{ unsatisfiable}$$

Thus, we can use SATProblem to prove that $\alpha \wedge \neg\beta$ is unsatisfiable (i.e. that we can't find a model) to prove entailment.

Efficiency

DPLL is much more efficient that truth tables, in fact even a naïve implementation can solve a problem that requires around 100 symbols (which was too long for truth tables). Furthermore it's possible to add some improvements to speed up execution and improve efficiency. Some improvements are

- Variables and values ordering.
- Using divide and conquer.
- Caching unsolvable subcases.
- Applying indexing and incremental recomputation.

Real word implementations can find models for problems with around 10 million propositional symbols. In practice SATSolvers can be used for

- Circuit verification.
- Software verification and synthesis.
- Protocol verification and synthesis.

12.3 Theorem proving inference procedures

Theorem-proving inference procedures try to build formulas starting from formulas already built. Among all theorem-proving inference procedures we will describe

- **Resolution**.
- **Forward chaining**.
- **Backward chaining**.

These algorithms can be applied to formulas in Propositional Logic and First Order Logic.

12.3.1 Resolution

As for PDLL we can divide the resolution precedure in two phases

1. If we want to prove $\alpha \models \beta$ we have to consider the formula $\alpha \wedge \neg\beta$.
2. Initially we need to transform the formula in CNF.
3. Finally we have to apply the actual algorithm that iteratively applies a resolution rule. The algorithm is initiated with the clauses obtained in the second step.

Algorithm

The core of the resolution algorithm is the resolution rule. Let us consider two clauses C_1 and C_2 (remember that clauses are set of literals in disjunctions, i.e. connected with or). Suppose that C_1 contains a literal l and C_2 contains a literal l^C that is complementary to l (i.e. l^C has the opposite sign of l and both refer to the same propositional symbol). The resolution rule says that C_1 and C_2 can be resolved to a new clause C, that is called **resolvent**, and contains all the literals of C_1 and all the literals of C_2, except the complementary literals.

$$C = \big(C_1 \setminus \{l\}\big) \cup \big(C_2 \setminus \{l^C\}\big)$$

Applying this rules can lead to building the empty clause $\perp$. The empty clause means false in fact we obtain the empty clause if C_1 and C_2 only contain l and l^C. In this case the conjunction of two unit clauses that are complementary is always false because if $l = 0$ then $l^C = 1$ and $l \wedge l^C = 0$ (the same is true if we consider $l = 1$). Given this fact, when we reach the empty clause we can say that the input formula is unsatisfiable, thus if the formula was $\alpha, \neg\beta$ then we can say that $\alpha \models \beta$.

To better understand how the resolution rule works let us consider the following example (Alice's coat example)

$$C_1 : R$$
$$C_2 : \neg RCA$$
$$C_3 : (\neg R \vee \neg W \vee RCA)$$
$$C_4 : W$$

The following steps can be applied

1. We can start by applying the resolution rule on clauses 2 and 3 to obtain

$$C_5 : (\neg R \vee \neg W)$$

2. We can now apply the rule on the new clause C_5 and C_4 to obtain

$$C_6 : (\neg R)$$

3. Finally if we apply the rule on C_6 and C_1 we obtain the empty clause.

Notice that the order in which the rule is applied doesn't impact the result but it has an effect on efficiency. There exists heuristics that allow us to choose on which clauses to apply the rule for improved efficiency.

Properties

Resolution is

- **Sound**.
- **Complete** in fact the new clauses are always smaller than the previous one, thus the algorithm either continues reducing the size of clauses until the empty one is reached or it can't apply the rule. In both cases the algorithm ends.

Resolution heuristics

Since there is no restriction on the order in which the rule has to be applied, we can define some heuristics that allow us to choose an order of execution.

- **Unit resolution** states that at least one of the parent clauses has to be a unit clause. This heuristic is complete for Horn clauses but not complete in general.
- **Input resolution** states that at least one of the two parent clauses has to be a member of the initial (i.e., input) set of clauses. This heuristic is complete for Horn clauses but not complete in general.
- **Linear resolution** is a generalisation of input resolution in which at least one of the parents is either in the initial set of clauses or is an ancestor of the other parent.
- **Set of support resolution** states that given a set of support S (which is a subset of the initial clauses such that the clauses not in S are satisfiable), every resolution involves a clause in S (the resolvent is added to S). If we are trying to prove $KB \models g$, we can initially consider $S = \{g\}$ because all the sentences in the KB are true.

These strategies allows to reduce the number of combinations to check, thus reducing execution time.

Horn clause

An Horn clause is a special formula of propositional logic made of

- A conjunction of positive symbols that implies a symbol.
- A propositional symbol.

If a Knowledge Base is made only of Horn clauses, then it's said to be in Horn form. For instance the following KB is in Horn form.

$$C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$$

If we transform an Horn clause in CNF, we notice that it has at most one positive symbol. If the clause has exactly one positive symbol then it's said to be a **definite clause**.

Horn clauses can be used to represents useful relations for agents, in fact we can use them to describe

- **Rules** e.g. $C \wedge D \Rightarrow B$.
- **Facts** e.g. C.
- **Goals** e.g. $A \wedge B \Rightarrow \bot$.
- **Empty clauses** (i.e. a contraddicion) e.g. $1 \Rightarrow \bot$.

12.3.2 Forward chaining

Forward chaining allows to prove that the KB (in general a statement) entails a goal g. The algorithm applies Modus Ponens (i.e. a rule of inference) to generate new facts starting from the KB. Modus Ponens infers (i.e. generates) Y, given $X_1 \wedge \cdots \wedge X_n \Rightarrow Y$ and $X_1, \ldots, X_N$. Modus Ponens is applied until

- it can't be applied anymore. In this case we have proven that the KB doesn't entails the goal g.
- the goal g is generated, thus proving that $KB \models g$.

To apply forward chaining, the Knowledge Base has to be made only of definite clauses (i.e. with only one positive literal).

And-or graph

Forward chaining is applied on an and-or graph in which

- Each node represents a propositional symbol.
- An arc connects n_1 to n_2 if n_1 is in the premise of n_2. Forward chaining uses Horn clauses, thus only propositional symbols (e.g. A) or conjunctions that imply a propositional symbol (e.g. $A \wedge B \Rightarrow C$). For this reason we have to represent the fact that two arcs are in conjunction (e.g. to represent $A \wedge B \Rightarrow C$) while other are in disjunction. In the former case we add an arc of circumference between the edges that participate in the relationship.

To better understand how and-or graphs work let us consider the knowledge base above

$$\begin{aligned} P &\Rightarrow Q \\ L \wedge M &\Rightarrow P \\ B \wedge L &\Rightarrow M \\ A \wedge P &\Rightarrow L \\ A \wedge B &\Rightarrow L \\ A & \\ B & \end{aligned}$$

This KB can be represented with the graph in Figure 12.1.

Algorithm

The algorithm uses

- An **agenda** to keep track of the symbols that are true but that haven't been processed yet. In our example the agenda is initialised with A and B.
- A **counter** for every symbol that counts how many premises of a letter are still unknown. The counter initially counts all the premises. For instance M's counter is 2.
- A list of **inferred** symbols. A symbol is inferred when its counter is 0. This list is used to avoid redundant work, in fact if a symbol in the inferred list won't be putted it in the agenda. Initially the inferred list is empty.

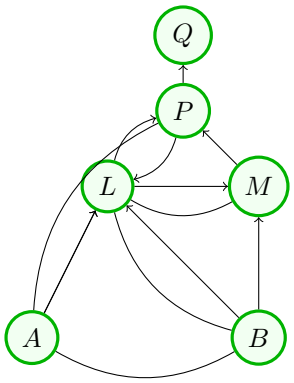

Figure 12.1: And-or graph for forward chaining.

Now that we know what the algorithm needs we can explain its execution. Forward chaining keeps executing a loop until the agenda is empty. The loop

1. Pops a symbol S from the agenda.
2. If the symbol is the goal then stops with success.
3. If the symbol is inferred, breaks the current iteration (like a continue instruction).
4. Adds S to the inferred list.
5. Decrements the counter of the symbols that have S in their premise. If the counter of one of the symbols goes to 0, the symbol is added to the agenda.

If the loop ends, false is returned because the goal hasn't been found.

Properties

If we consider KBs composed of definite clauses, forward chaining is

- **Sound** because Modus Ponens is sound.
- **Complete** because the algorithm reaches a fixed point from where no new atomic sentences can be derived.

Forward chaining is a **data-driven**, automatic, "unconscious" process that can derive everything that is entailed by the KB, but lots of work is irrelevant to a specific goal.

12.3.3 Backward chaining

Backward chaining works like forward chaining but starting from a goal q (i.e. a positive literal) and

- If q is already in the KB, we have proved the goal.
- Otherwise we should prove all the premises of the causes that derive q.

We should also avoid

- Loops.
- Repeated work to improve performance.

Properties

Backward chaining is **sound** and **complete** for KBs composed of definite clauses for the same reasons seen in forward chaining.

Backward chaining is a **goal-driven**, appropriate for problem-solving process and its complexity can be much less than linear in the size of the KB.

Chapter 13

Planning

In previous chapters we have defined the different logics an agent can use. Now we have to analyse the behaviour of such agents. In particular we are going to analyse how logical agents use tree-search to solve a problem. The combination of tree-search and logic is called **classical planning**. Classical planning is a specialisation of tree-search, in fact we can reuse the same concepts and algorithms seen in tree-search but actions, goal and states are defined using logical sentences.

Planning language Since every core aspect of tree-search is described using logical statements, we need a language to specify goals, actions and states. Some of the most important languages are

- **STRIPS** (STandford Research Institute Problem Solver). STRIPS is based on a simplified version of First Order Logic (without functions). This language is quite simple yet rather expressive.
- **PDDL** (Planning Domain Definition Language). PDDL is an extension of STRIPS thus it's more general than STRIPS. PDDL actually refers to a family of languages that share a common core but in which every language has different extra functionalities.

13.1 Syntax

The syntax we are going to analyse is valid for both STRIPS and PDDL.

Example To better understand how STRIPS and PDDL work we are going to model a cleaner robot that can move between two rooms. Each room can be dirty and the robot can suck the dirt.

13.1.1 Constants and predicates

The most basic operation is the definition of constants and predicates.

Constant

A constant can be simply defined using a word or a letter. There is no standard condition for representing a constant but it is usually a good idea to use names that are related to the object.

Example In our example we can define

- A constant *Robot* that defines the cleaner robot.
- Two constants $R1$ and $R2$ that model the rooms.

Predicates

A predicate (also literal) defines a property of an object or a relationship between multiple objects. Predicates have a name and a list of variables between parentheses. The list is used to indicate the arity of the predicate. The set of predicates is also called **conditions set**.

Example In our example we can define

- A predicate $\text{In}(x, y)$ that is true if object x is in room y. This predicate defines a **relation** between objects. Notice that there is no concept of type, in fact in principle we can assign any constant to y, thus even *Robot* which is not a room. In practice the designer defines the meaning of predicate so that it can be used correctly.
- A predicate $\text{Clean}(x)$ that is true if room x is not dirty. This predicate defines a **property** of a room.

13.1.2 State

After defining constants and predicates we should define an initial state (i.e. the state from which the agent starts). Before analysing the initial state we should describe how a state is defined.

A state S is a set of positive, ground and atomic predicates that are true in S (i.e. a set of predicates that are true when the agent is in state S). The predicates in S are connected with an **and** operation (i.e. the predicates are in conjunction). Before going on with the definition of a state we should emphasise the characteristics of the predicates in a state, in fact a predicate in a state should be

- **Positive**, this means that we can't have a negated predicate. For this reason we can't have a state like $In(Robot, R1) \land \neg Clean(R2)$.
- **Ground**, this means that only constants can be passed to predicates. For this reason the state $Clean(x)$ isn't valid.
- **Atomic**.
- In **conjunction**. For this reason we can't define state $Clean(R1) \lor Clean(R2)$.

Assumptions When defining states we have to make three fundamental assumptions

- **Unique Name Assumption**. Different constants always denote (refer to) different objects in the world. The same assumption isn't true in first order logic, in fact it's possible to give the same interpretation to two different variables.
- **Domain Closure Assumption**. The world only includes objects that are denoted by a constant. In other words in the world there is no other thing that what is asserted by the states.

- **Closed World Assumption**. All literals (i.e. everything) that are not explicitly written in a state are false. For instance if a state is $Clean(R1) \wedge In(Robot, R1)$ then we assume $\neg Clean(R2)$ and $\neg In(Robot, R2)$ are true.

Initial state The initial state for a logic agent is a normal state that has to be specifically defined. The other states are obtained with search (i.e. with reasoning).

Example In our example the initial state can be

$$In(Robot, R1) \wedge Clean(R1)$$

13.1.3 Goal

A goal is the set of predicates that defines when an agent has to stop (i.e. when it has reached its goal). A goal is defined as a conjunction (and) of literals (predicates) that are also called **sub-goals**. Goals can be defined differently in STRIPS and PDDL, in fact in the first case we can only use positive literals while in the second case we can also use negated sub-goals. This means that the goal $Clean(R2) \wedge \neg Clean(R1)$ is a valid goal only for PDDL.

Satisfaction A goal G is satisfied in a state S when

- All positive predicates in G are present in S. If we define S as a subset of the conditions than G is satisfied by S when G is a subset of S.
- All negative literals in G are not contained in S.

In other words a goal finds in a state everything it is looking for and doesn't find in the state what it wants to avoid.

Differently from states, goals can contain variables. In such cases the variable represents a predicate with an existential quantifier, thus G is satisfied if there exists a literal in S that matches with the one in G. For instance the goal $Clean(x)$ is true if there exists an assignment (of a constant to variable x) that makes the predicate true.

Set of states It's important to underline that a goal doesn't define a single state but a set of states that satisfy the goal. For instance goal $Clean(R1)$ is satisfied both by states $Clean(R1) \wedge Clean(R2) \wedge In(Robot, R1)$ and $Clean(R1) \wedge In(Robot, R2)$

Example In our example the goal state can be

$$Clean(R1) \wedge Clean(R2)$$

13.1.4 Action

An action is defined by

- A **name**.
- A set of **preconditions**. The preconditions are the literals that have to be true in state S in order to apply the action to state S. Preconditions have to be defined using the same rules of goals.

- A **delete set**. The delete set is the set of predicates that have to be removed from S (if the action is applied to S).
- An **add set**. The add set is the set of predicates that have to be added to S after executing the action.

The add and delete set can also be merged in an effect list in which all the items in the add list are present in positive form while the items in the delete list are present in negated form.

Applicability Action A is applicable in state S if all positive literals in A's precondition are present in S (i.e. the precondition is a subset of the state) and all negative non literals are not present in the S.

Destination state The state S' generated applying action A to state S is obtained

1. Copying S.
2. Removing the elements in the delete list from S'.
3. Adding the elements in the add list to S'.

Action schema Actions aren't parametric, thus to represents actions "clean room R1" and "clean room R2" we have to define two different actions $CleanR1$ and $CleanR2$. This problem can be solved using action schema that allow to define parametric actions. The parameters are variables and are listed between parenthesis after the name of the action. In our example the two actions can be condensed in the scheme $Clean(x)$.

When defining an action scheme we have to ensure that

- Every variable used in the scheme appears in the signature of the scheme.
- The variables used in the add and delete lists have to appear in the preconditions. A variable is **unbound** if it doesn't appear in the preconditions.

Example In our example we can define two actions

- $$\begin{aligned} &Clean(x) \\ &\qquad \text{preconditions} = In(Robot, x) \\ &\qquad \text{delete list} = \\ &\qquad \text{add list} = Clean(x) \end{aligned}$$

- $$\begin{aligned} &Move(x, y) \\ &\qquad \text{preconditions} = In(Robot, x) \\ &\qquad \text{delete list} = In(Robot, x) \\ &\qquad \text{add list} = In(Robot, y) \end{aligned}$$

13.1.5 Extensions

PDDL can also be extended to add functionalities. In particular we can

- Add predicates for **equality** ($=$) and **inequality** ($\neq$).
- Add **algebraic expressions** ($+$, $-$).
- Add **states constraints** that allow to reduce the literals in the precondition. For instance say we want to model two adjacent cells. Normally we have to state that cell 1 is adjacent to cell 2 and cell 2 is adjacent to cell 1. The repetition can be avoided adding the constraint $Adjacent(x, y) \Rightarrow Adjacent(y, x)$ (i.e. we have encoded in the constraint the fact that the adjacent relation is symmetric).

13.2 Search problem solving

13.2.1 Forward planning

In forward planning the agents

1. Starts from the initial states.
2. Search the state space until a goal is reached. Search can be done using any of the algorithm seen in search problems (Breadth First, Depth First, Monte Carlo, ...).

Actions When a node has to expand a state S, it

1. Checks what actions can be applied to S.
2. Generate the next states applying (removing literals in the delete list and adding the one in the add list) each action to S.

Disadvantages This solving method can be rather inefficient in real problems because each state can have a large number of applicable actions but only a few number of good actions (i.e. actions that lead to a goal state). In some cases forward planning can be simplified relaxing the problem. In particular we can ignore the delete list (in practice S' is obtained only adding the add list to S). This method generates inconsistent states but makes it easier to solve the problem.

13.2.2 Backward planning

Backward planning has the opposite approach with respect to forward planning, in fact

1. We start from the goal.
2. We reason backward searching (using any search algorithm) the goal space. The search is executed generating new goals until the initial state is satisfied by a goal.

Relevant states

To define the operation that allows us to obtain new states, we have to define what a relevant state is.

A state A is relevant for goal G if at least one of the positive effects of A satisfies a sub-goal of G.

Regression

Regression allows to expand a goal G. In particular regression of G through a relevant action A is the less constraining goal $R[G, A]$, such that given a state S that satisfies R the following conditions hold

- The preconditions of A are satisfied in S.
- The application of A in S generates a state S' that satisfies G.

Practically to expand a goal G we have to

1. Find all the actions Rel that have a literal in the add list in common with G (e.g. $G = Clean(R1) \wedge In(Robot, R1)$, $A = ..., \text{add} = Clean(R1)$).
2. Generate the goal G' as a copy of goal G.
3. Remove the elements of the action's add list from G'.
4. Add the elements of the action's precondition to the new goal G'.
5. Repeat steps 2-5 with the next action in Rel.

After creating a goal state G' through regression we also have to check that the new state is consistent with the model. For instance if we obtain a state $In(Robot, R1) \wedge In(Robot, R2)$, we know that this state has to be discarded (i.e. not expanded) because the robot can't be in both rooms at the same time.

Problems The main problem of this approach is that we can obtain inconsistent goals that are hard to detect (in general it's hard to detect inconsistencies). A way to mitigate this problem is to use state constraints.

13.2.3 GraphPlan

GraphPlan searches the knowledge base space building a graph organised in levels of literals or actions. Levels of literals are interleaved by levels of actions. The nodes at level i are connected to the nodes in the next level $i + 1$. An example of graph used in GraphPlan is shown in Figure 13.1.

The initial literals represent the initial state of the agent. The successive action layer is the set of actions that can be applied in the initial state. In general at literal level i we have all the literals at level $i - 1$ plus the literals generated by the actions at action level i (e.g. at literal level 1 we find the literals of level 0 and the literals generated at action level 0). This means that a literal, once generated, is never deleted, thus the set of literals can only grow.

At every level all actions are applied, thus the actions applied at level i are also applicable at level $i + 1$. This means that the set of actions enlarges, too.

Algorithm Given a graph, the algorithm tries to find a sub-graph made only of actions compatible with each other.

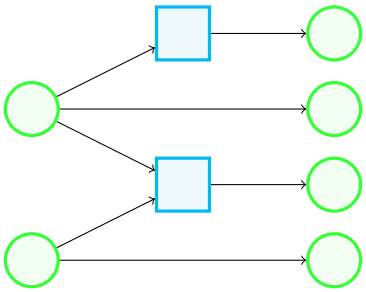

Figure 13.1: An example of graph used in GraphPlan (literals are circles, actions are squares).

13.2.4 Hierarchical plan

In hierarchical planning a plan is specified going from a general actions to more specific action until we reach atomic executable actions. In this case going down a path doesn't mean to explore a set of actions but to specify an action, thus the plan is the set of leaves of the tree. In the generated tree some actions are in conjunction and some in disjunction. When some actions are in disjunction, the parent action can be expanded in one of the children actions. On the other hand when some actions are in conjunction, the parent action is made of its children action.

An example of hierarchical plan for travelling from one point to another is shown in Figure 13.2.

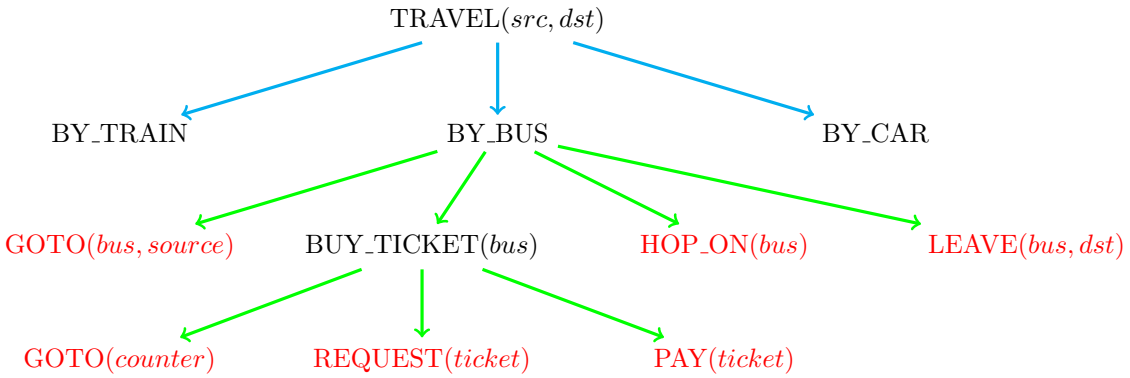

Figure 13.2: An example of hierarchical plan (and in green, or in light blue, final actions in red).

13.3 SATPlan

SATPlan is an efficient planning algorithm based on propositional logic. In particular SATPlan uses PL to represent a goal and the knowledge base of the agent and tries to find a model for the initial state, the goal and the knowledge base. The actual plan (i.e. the sequence of actions) can be extracted from the model. One of the most efficient techniques for finding a model in propositional logic is SAT.

The planning process can be summed as follows

1. Start from a problem defined in STRIPS or PDDL.

2. Represent the problem in Propositional Logic in Conjunctive Normal Form.
3. Solve the problem using the SAT Solver.
4. Obtain the plan from the model obtained at the previous stage.

13.3.1 SAT

The SAT solver checks plans of incremental length L until a model is found. In other words the solver checks plans with an increasing number of actions (starting from $L = 0$, in fact we want to check if the initial state trivially satisfies the goal). In particular at each iteration (i.e. the iteration $L = l$) the solver

1. Builds the propositional logic representation of the knowledge base and of the goal.
2. Tries to build a model for the knowledge base and the goal.
3. If a model is found then the sequence of actions is extracted, otherwise the same process is repeated with $L = l + 1$.

This loop is executed until a plan is found (i.e. a model is found) or a timeout is requested. Usually the timeout is implemented with a level limit L_{max}, thus the algorithm stops when $L = L_{max}$.

Example

To explore the SATPlan algorithm we can consider a box initially closed that contains a ring. The goal of the agent is to obtain the key. The solution of this problem requires two actions (thus we have to reach $L = 2$)

1. Open the box.
2. Pick up the key.

Normally the number of moves needed to solve the problem is unknown.

PDDL The problem can be represented in PDDL as follows

```
INIT = {
    Closed(B),
    In(R, B),
    Free
}

GOAL = {
    Holds(R)
}

ACTIONS = {
    OPEN(x) := {
        PRECONDITION={Closed(x), Free}
        ADD={Open(x)}
        REMOVE={Closed(x)}
    },

    PICKUP(x, y) := {
        PRECONDITION={In(x, y), Open(y), Free}
```

```
20          ADD={Holds(x)}
21          REMOVE={Free, In(x, y)}
22      }
23  }
```

13.3.2 Translating facts

The SAT solver has to translate the knowledge base in propositional logic, thus we have to create a propositional symbol for each fact. Because facts change over time, we have to add a propositional symbol for every instant t. This means that we have a propositional symbol for each fact and for each time instant.

For instance P^t means that the propositional symbol P at time t.

Example In our example the facts of the knowledge base are represented as follows

- `Open(B)` $\rightarrow OpenB^t$
- `Closed(B)` $\rightarrow \neg OpenB^t$
- `In(R, B)` $\rightarrow InRB^t$
- `Holds(R)` $\rightarrow HoldsR^t$
- `Free` $\rightarrow \neg HoldsR^t$

The propositional symbols are just a subset of the symbols generated by the actual algorithm, in fact in principle the algorithm tries to apply all constants to each fact to generate all possible combination of constants fact. In our example, if we consider `Open(x)`, the algorithm would have generated $Open(B)$ and $Open(R)$. In our example we will consider only the relevant symbols (i.e. symbols that are actually used).

Furthermore we have also taken some shortcuts to reduce the number of symbols. For instance we have translated `Closed(B)` with $\neg OpenB^t$ because we know that open is the opposite of closed. The algorithm doesn't have such information, thus it would have translated `Closed(B)` with $ClosedB^t$.

13.3.3 Translating actions

A SAT solver doesn't need only facts but also actions, thus we have to find a way to translate an action in term of propositional logic's symbols.

Action An action itself is represented simply using the same notation of facts. In our example we will translate the actions as follows

- `open(B)` $\rightarrow openB^t$
- `pickup(R, B)` $\rightarrow pickupRB^t$

Notice that the symbol $openB^t$ used to represent an action is different from the symbol $OpenB^t$ used to represent the fact that the box is closed.

Preconditions Apart from representing an action, we also have to represent the preconditions of the action, that is the facts that have to hold for the action to be applicable. A precondition is represented as follows

$$actionSymbol^t \Rightarrow precondFact^t \wedge precondFact2^t$$

and can be read as *if we want to apply action* $actionSymbol^t$ *at time* t *then the facts* $precondFact^t$ *and* $precondFact2^t$ *have to be true at time* t. A precondition is not in CNF form, thus we have to rewrite it as a conjunction of disjunctions. For instance the precondition $actionSymbol^t \Rightarrow precondFact^t \wedge precondFact2^t$ can be transformed in two predicates

$$\neg actionSymbol^t \vee \neg precondFact^t \qquad \neg actionSymbol^t \vee \neg precondFact2^t$$

In our example we need the following preconditions

- $openB^t \Rightarrow \neg OpenB^t \wedge \neg HoldR^t$
- $pickupRB^t \Rightarrow InRB^t \wedge OpenB^t \wedge \neg HoldsR^t$

Notice that, because the set of actions is always growing, for a generic level L we have to represent $L-1$ set of actions (i.e. all the actions for every level $L \neq 0$).

Effects The final block needed to represent an action is the effect. A predicate P is true at time $t+1$ if an action has added P at time t or P was true at time t and no action has made it not true. Formally we can write an effect as

$$P^{t+1} \iff actionCausingP^t \vee (P^t \wedge \neg actionCausingNotP^t)$$

An effect is also called **fluent axiom**.

As for preconditions, the effects have to be translated into CNF. A way to to that is to consider the implication in one direction ($A \Rightarrow B$) and than the implication in the other direction ($B \Rightarrow A$).

In our example the effects of the two actions are

- $OpenB^{t+1} \iff openB^t \vee OpenB^t$
- $InRB^{t+1} \iff InRB^t \wedge \neg pickupRB^t$
- $HoldsR^{t+1} \iff pickupRB^t \vee HoldsR^t$

Contemporary actions Sometimes two actions can't happen at the same timestamp t, thus we have to add an **exclusion axiom** (at most one for each couple of actions). In general an exclusion axiom is represented as follows

$$\neg actionA^t \vee \neg actionB^t$$

In our example we can't open the box and pickup the key at the same time. This can be represented with the following axiom

$$\neg openB^t \vee \neg pickupR^t$$

13.3.4 SATPlan execution

At each level L the SAT solver generates the symbols to represent

- The **initial state**. The initial state is the same for every level.
- The **goal**. A new goal is generated for every level. In particular the goal is updated considering $t = L$ (in our example the goal is $HoldsR^1$ at level 1 and $HoldsR^2$ at level 2).
- The **actions**.
- The **actions' preconditions**.
- The **actions' effects** (i.e. the **fluent axioms**).
- The **contemporary actions** (i.e. the **exclusion axioms**).

The symbols related to actions are obtained as the union of the symbols at time $L - 1$ and the symbols generated by the fluent axioms.

Plan

The plan can be obtained extracting the actions that are true at each time t. This means that the action to do at $t = 0$ is the one that is true at $t = 0$, the action at $t = 1$ is the one true at $t = 1$ and so on.

13.4 Planning in the real world

Until now we have made some assumptions regarding the world in which logical agents work in. In particular we have assumed that

- The world is **fully observable**.
- The world is **static**.
- The world is **discrete**.
- Only a **single agent** acts in the world.
- The world is **deterministic**.
- The world is **known** (i.e. the designer knows the rules and the laws that describe the environment).

In the real world some of these assumptions don't hold, thus we have to define some planning extensions.

13.4.1 Non deterministic planning

In non deterministic planning an agent hasn't the complete knowledge of the result of an action. This means that the agent has to consider the set possible states of an action. The set of possible states is called **believe state**.

13.4.2 Sensorless planning

In sensorless planning an agent can't discover what it doesn't know about the world. In general, planning for sensorless agents is hard but in some cases, non-sensor agents can solve non deterministic problem easily.

13.4.3 Conditional planning

In conditional planning an action can also sense the world. Conditions and loops can be inserted in the plan. This means that a plan can be something like

```
if condition then
    while loop_condition do
        action
else
    another_action
```

The big difference with normal planning is that the decisions are taken at planning time and not by the designer.

13.4.4 Multi-agent planning

Multi-agent planning allows to manage the interaction of multiple agents.

13.4.5 Monitoring agents

The life-cycle of an agent can be divided in two phases

1. Planning. During the planning phase the agent isn't actually executing the plan, in fact it is just thinking.
2. Executing the plan.

In some cases things might go wrong and the agent may find itself in a state different from the one stated by the plan. This means that an agent should be able to monitor the environment and the state of execution to check if everything is going as the plan says.

When an agent understands that it is in a state it shouldn't be in, it can do two things

- Compute a way to go back to the correct plan. This means that if the plan is $A \to B \to C \to D$ and the agent finds itself in B_1 instead of B, then it tries to come back to B.
- Compute a new plan to reach the goal. This means that if the plan is $A \to B \to C \to D$ and the agent finds itself in B_1 instead of B, then it computes a new path $B_1 \to C_1 \to D$.

Online continuous planning

Some agents monitor their progress iterating the planning-executing phases. The life-cycle of such agents is

1. Build a plan.
2. Execute the first step of the plan.
3. Execute step 1 to 3 until the goal is reached.

Definitions