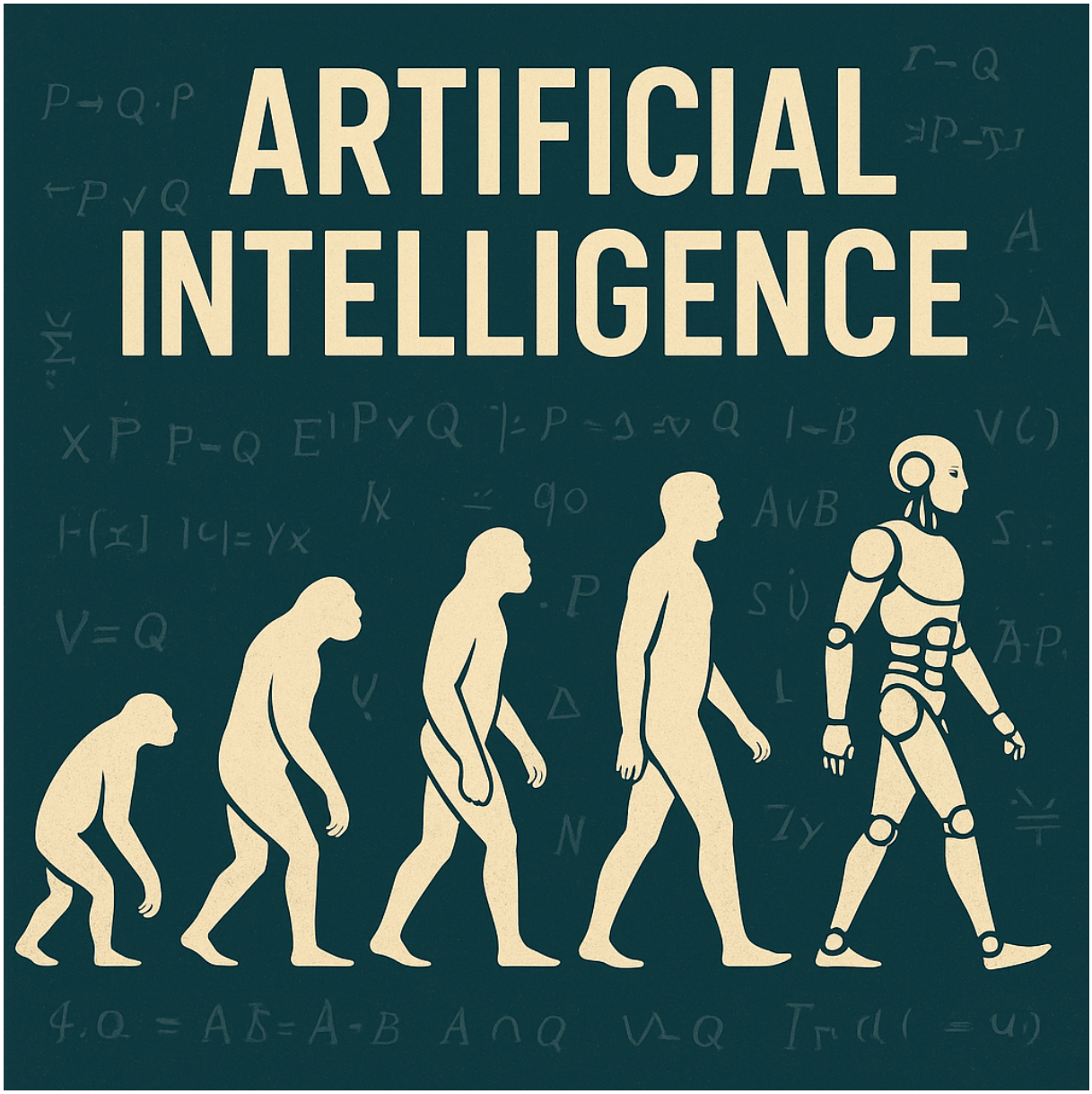
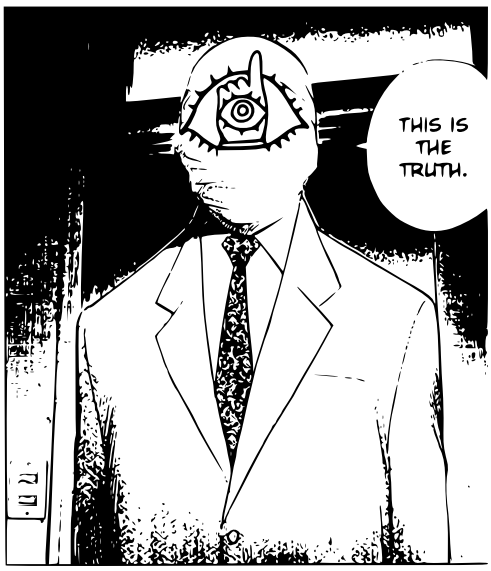
Marco Casu



SAPIENZA
UNIVERSITÀ DI ROMA

Faculty of Information Engineering, Computer Science and Statistics
Department of Computer, Control and Management Engineering
Master's degree in Artificial Intelligence and Robotics

This document summarizes and presents the topics for the Artificial intelligence course for the Master's degree in Artificial Intelligence and Robotics at Sapienza University of Rome. The document is free for any use. If the reader notices any typos, they are kindly requested to report them to the author.



CONTENTS

CHAPTER

1

INTRODUZIONE

1.1 Basic Definitions

In the context of the artificial intelligence, an **agent** is an entity that can

- Perceive the environment through *sensors* (percepts)
- Act upon the environment through *actuators* (actions).

We say that an agent is **rational** if he selects the action that maximize a given *performance measure*, informally, he attempts to do "the right thing". The best case is hypothetical and often unattainable, because the agent usually can't perform all the actions needed, and can't perceive all the information about the environment.

An agent has a performance measure M and a set A of all possible actions, given percept a sequence A and knowledge K (data), he has to select the next action $a \in A$, is a map

$$M \times P \times K \longrightarrow A. \tag{1.1}$$

An action a is optimal if it maximize the expected value of M, given the sequence P and the knowledge K. An agent is rational if he always chooses the optimal action. More specifically, an agent consists in two components:

- an architecture which provides an interface to the environment
- a program executed on that architecture.

There are some limitation that we aren't considering, such as the fact that determining the optimal choice could take too much time or memory on the architecture.

1.1.1 Types of Agents

There are different kinds of agents, a **Table Driven Agent** is the simplest form of agent architecture. It's essentially a look-up table that maps every possible sequence of percepts (what the agent has sensed so far) to a corresponding action the agent should take. His behavior can be resumed in the algorithm 1.

A **Reflex Agent** consists in three components:

- sensors to get information from the environment

Algorithm 1 Table Driven Agent

Require: *percepts*
  persistent: *percepts*, a sequence, initially empty
    table, a table of actions, indexed by percept sequences, initially fully specified
  append *percept* to the end of *percepts*
  action←`LookUp`(*percepts,table*)
  return *action*

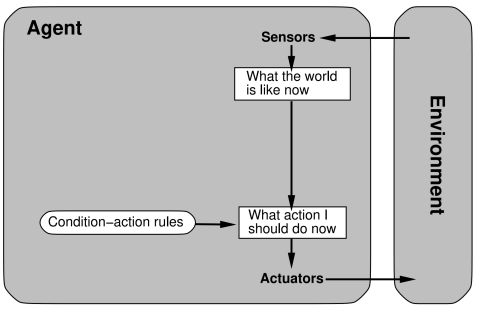


Figure 1.1: Reflex agent diagram

- a decision making process, in form of a *condition-action rules*, typically looks like `IF (condition) THEN (action)`.
- actuators, the outputs that allow the agent to affect or change the environment.

A **Model-Based Reflex Agent** is an enhanced version of the previous one, the key enhancement here is the inclusion of an *Internal State* and a *Model of the World* to make up for the agent's limited view of the environment. The internal state cannot simply be the last thing the agent saw; it needs to be updated to reflect reality. This is done using a Model of the World, which contains two key pieces of knowledge:

- How the world evolves independently of the agent, his accounts for changes in the environment that occur regardless of the agent's actions (e.g., a clock ticking, an external event).
- How the agent's own actions affect the world, this is the effect of the agent's previous action (e.g., if the agent drove forward, its position changed).

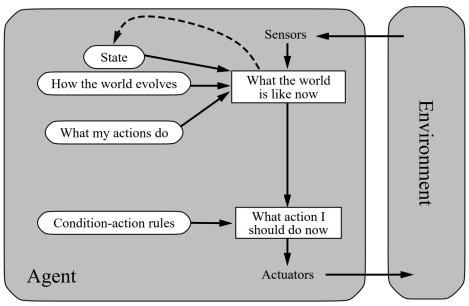


Figure 1.2: Model Based Reflex agent diagram

If a model based reflex agent consider the future prospective, is a **Goal Based Agent**, as shown in figure 1.3.

A **Utility Based Agent** is equipped with a *utility function* that maps a state to a number which represents how desirable the state is. Agent's utility function is an internalization of the performance function.

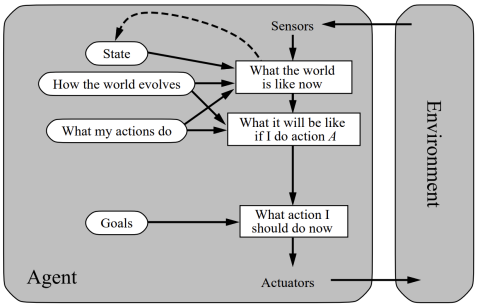


Figure 1.3: Goal Based agent diagram

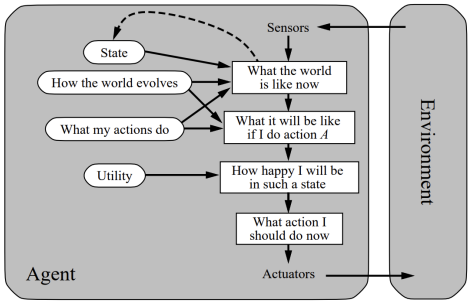


Figure 1.4: Utility Based agent diagram

A **Learning Agent** is an architecture designed to improve its efficiency over time by separating four functions:

- the performance element selects actions
- the critic provides feedback on those actions against a standard
- the learning element uses this feedback to update the agent's internal knowledge
- the problem generator suggests exploratory actions to gain new knowledge. This structure enables the agent to continuously adapt and improve its decision-making.

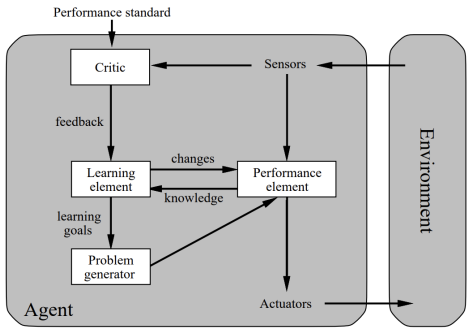


Figure 1.5: Learning agent diagram

An agent can be classified in one of the following groups:

- a **domain specific agent** is a solver specific to a particular problem (such as playing chess), is usually more efficient.
- a **general agent** is a solver for general problems, such as learning the rule of any board game, is usually more intelligent but less efficient.

1.1.2 The Environment

An environment can be classified in terms of different attributes:

- An environment can be **fully observable** if all the relevant information are accessible to the sensors, otherwise is **partially observable**.
- If there are no uncertainty, the environment is **deterministic**. An environment is **stochastic** if uncertainty is quantified by using probabilities, otherwise is **non deterministic** if uncertainty is managed as actions with multiple outcomes.
- An environment is **episodic** if the correctness of an action can be evaluated instantly, otherwise if are evaluated in the future developments, is **sequential**.
- An environment can be **static** or **dynamic**, if itdoes not change, but the agent's performance score changes, the environment is called **semi-dynamic**.
- An environment can be perceived as **discrete** or **continuous**.
- In a single environment there may be multiple agent, that can be **competitive** or **cooperative**.

Many sub-areas of AI can be classified by:

- Domain-specific vs. general.
- The environment.
- Particular agent architectures sometimes also play a role, especially in Robotics.

It follows a classification of some areas in terms of the attributes we discussed:

- **Classical Search**, the environment is
  - fully observable
  - deterministic
  - static
  - sequential
  - discrete
  - single-agent

  and the approach is *domain specific*.
- **Planning**, the environment is
  - fully observable
  - deterministic
  - static
  - sequential
  - discrete
  - single-agent

  and the approach is *general*.
- **Adversarial Search**, the environment is
  - fully observable
  - deterministic
  - static
  - sequential
  - discrete
  - multi-agent

and the approach is *domain specific*.

- **General Game Playing**, the environment is
  - fully observable
  - deterministic
  - static
  - sequential
  - discrete
  - multi-agent

  and the approach is *general*.
- **Constraint Satisfaction & Reasoning**, the environment is
  - fully observable
  - deterministic
  - static
  - episodic
  - discrete
  - single-agent

  and the approach is *general*.
- **Probabilistic Reasoning**, the environment is
  - partially observable
  - stochastic
  - static
  - episodic
  - discrete
  - single-agent

  and the approach is *general*.

CHAPTER

2

SEARCH PROBLEMS

2.1 Classical Search

Let's consider two basic example of classical search problems, the first one is the following:

Starting from Zurigo, we would like to find a route to Zagabria. We have an initial state (Zurigo), and we have to apply actions (drive) to reach the goal state (Zagabria). Another example is the following, we want to solve the tiles-puzzle game, shown in figure 2.1, to reach the left state, starting from the right one, the actions to perform is the move of the tiles. A performance measure could be to minimize the summed-up action costs.

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

—

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

Figure 2.1: The tile game

In the classical search context, we restrict the agent's environment to a very simple setting, with a finite number of states and actions, a single agent, a fully observable stati environment that doesn't evolve, given that assumption, the classical search problems are the simplest one, despite that, are very important problems in practice.

Every problem specifies a state space.

Definition 1 *A **State Space** is a 6-tuple $\Theta = (S, A, c, T, I, S^G)$ where:*

- *S is a finite set of the states.*
- *A is a finite set of actions.*
- *$c : A \to \mathbb{R}^+$ is the cost function.*
- *$T \subseteq S \times A \times S$ is the transition relation, that describes how an action on a given state make the agent evolve to the next state. We assume that the problem is deterministic, so for all $s \in S$, $a \in A$, if $(s, a, s') \in T$ and $(s, a, s'') \in T$ then $s' = s''$.*
- *$I \in S$ is the initial state*
- *$S^G \subseteq S$ is the set of the goal states, where we want to end.*

A transition (s, a, s') can be denoted $s \xrightarrow{a} s'$, we say that $s \to s'$ if $\exists a$ such that $(s, a, s') \in T$. We say that Θ has *unit costs* if $\forall a \in A, \quad c(a) = 1$. A state space can be illustrated as a directed labeled graph.

Definition 2 *Let $\Theta = (S, A, c, T, I, S^G)$ to be a state space, we say that*

- *s' is a **successor** of s if $s \to s'$*
- *s' is a **predecessor** of s if $s' \to sT$*
- *we say that s' is **reachable from** s if*

$$\exists(a_1 \dots, a_n) \subseteq A \tag{2.1}$$
$$\exists(s_2 \dots, s_{n-1}) \subseteq S \tag{2.2}$$
$$(s, a_1, s_2) \in T \tag{2.3}$$
$$(s_2, a_2, s_3) \in T \tag{2.4}$$
$$\vdots \tag{2.5}$$
$$(s_{n-1}, a_n, s') \in T \tag{2.6}$$

  we can write the sequence as follows

$$s \xrightarrow{a_1} s_2, \dots, s_{n-1} \xrightarrow{a_n} s'. \tag{2.7}$$

- *We say that s is **reachable** (without reference state) if is reachable from I.*
- *s is **solvable** if there exists $s' \in S^G$ such that s' is reachable from s, otherwise s is **dead end**.*

Definition 3 *Let $\Theta = (S, A, c, T, I, S^G)$ to be a state space, and let $s \in S$. A **solution** for s is a path from s to some goal state $s' \in S^G$. The solution is **optimal** if it's cost is minimal, let H to be the set of all possible solution (sequence of states) for s*

$$H = \{\text{paths from } s \text{ to } s' \in S^G\} = \{(s_{i0}, s_{i1}, s_{i1}, \dots s_{in}) : s_{in} \in S^G, \ s_{i0} = s\} \tag{2.8}$$

where n^i is the length of the i-th solution. The optimal solution is

$$\arg \min_{(s, s_{i1}, \dots s_{in}) \in H} \sum_{j=0}^{n^i} c(s_{ij}). \tag{2.9}$$

A solution for I is called **solution for** Θ, if such that solution exists, Θ is **solvable**.

2.1.1 Vacuum Cleaner Example

Let's consider a vacuum cleaner, that is the agent of our problem, the goal is to clean a room, the vacuum cleaner can be in two possible points (left and right), this points can be clean or dirty. The agent can perform the following actions

- move right
- move left
- suck the dust on the floor

there are 8 possible states

- left point clean, right point clean, vacuum cleaner is on right point
- left point dirty, right point clean, vacuum cleaner is on right point
- left point clean, right point dirty, vacuum cleaner is on right point
- left point dirty, right point dirty, vacuum cleaner is on right point
- left point clean, right point clean, vacuum cleaner is on left point
- left point dirty, right point clean, vacuum cleaner is on left point
- left point clean, right point dirty, vacuum cleaner is on left point
- left point dirty, right point dirty, vacuum cleaner is on left point

the initial state is the one with the left point dirty, right point dirty, and the vacuum cleaner on the left point, we denote the actions R (move right), L (move left), S suck. The state space of the problem is show in figure 2.2.

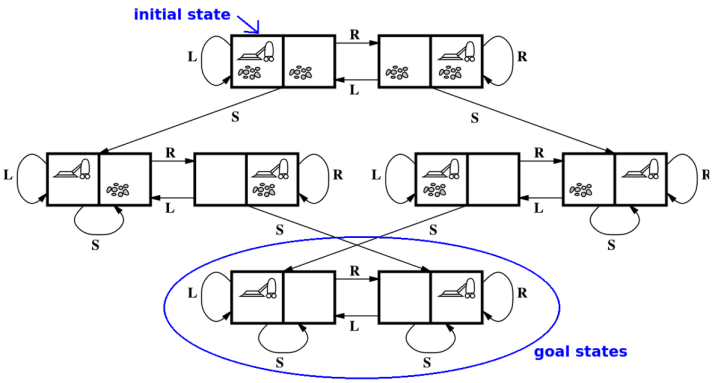


Figure 2.2: The vacuum cleaner state space

Some example of set of actions that can lead from the initial state to a goal state are

$$S \longrightarrow R \longrightarrow S$$
$$S \longrightarrow R \longrightarrow R \longrightarrow S$$
$$R \longrightarrow S \longrightarrow L \longrightarrow S$$

Typically, the state space is exponentially large in the size of its specification, search problems are typically computationally hard and/or NP-complete. We say that we can give an *explicit description* of a search problem if we can define his state space as a graph.

2.2 Problem Descriptions

Definition 4 *We have a **black box description** of the problem if we can't describe the state space explicitly but we can*

- *Know which is the initial state*
- *Check if a given state is a goal state*
- *Check the cost of a given action a*
- *Given a state s, check all the actions that are applicable to state s*
- *Given a state s and an applicable action a, we can get the successor state.*

We can think about it in a programming-way, given a problem described by $\Theta = (S, A, c, T, I, S^G)$, we can't check directly Θ, but we have an *API* of the problem that provide the following functions

- `InitialState()` : return the initial state of the problem
- `GoalTest(`s`)` : return true if and only if $s \in S^G$
- `Cost`$(a$: return $c(a))$
- `Actions`(s) : return the set $\{a \ : \ \exists s \xrightarrow{a} s' \in T$ for some $s' \in S\}$
- `ChildState`(s, a) : return s' if $s \xrightarrow{a} s' \in T$.

We **specify** a search problem if we can program/access to such an *API*. There are a declarative description too.

Definition 5

We have a **declarative description** of the problem if is described by the following sets:

- P is a set of boolean variables (*propositions*)
- $I \subseteq P$ is the subset of P indicating which propositions are true in the initial states.
- $G \subset P$ is the subset of P describing the goal states in the following way
  - a state s is a set of propositions
  - $s \subseteq G \iff s$ is a goal state
- A is a set of actions, each action a is described by
  - a set $pre_a \subseteq P$ of *precondition*, a can be performed if and only if the conditions in pre_a are true.
  - a set of propositions add_a
  - a set of propositions del_a
  - the outcome of each action is the state $(\{s\} \cup add_a) \backslash del_a$
  - $c : A \to \mathbb{R}$ is the cost function.

Declarative descriptions are strictly more powerful than black box ones. In this section we assume the black box description. In principle, the search strategies we will discuss can be used with any problem description that allows to implement the black box *API*.

2.2.1 Missionaries and Cannibals Example

The problem is the following

- there are a river and a boat that can carry the people from the left bank to the right
- there are 6 people, 3 missionaries and 3 cannibals
- the boat can carry 0, 1 or 2 people at the same time, not 3
- the goal is to get everybody to the left bank
- if at any time, there are more cannibals than missionaries in one bank, the missionaries get killed and the game is lost.

We can model the problem as follows

- the state space S is

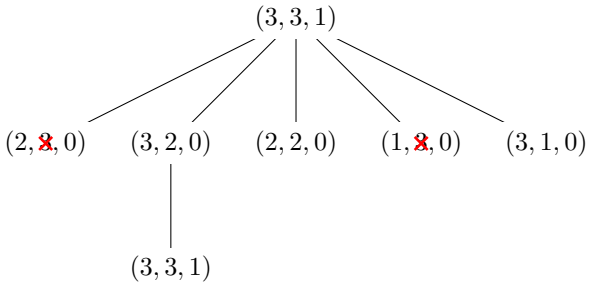
$$S = \{(M, C, B) \ : \ M + C = 6, \ 0 \leq M \leq 3, \ 0 \leq C \leq 3, \ B \in \{0, 1\}\} \tag{2.10}$$

  M represents the current number of missionaries on the right bank, S represents the current number of cannibals on the right bank, $B = 1$ if the boat is on the right bank, otherwise is on the left.

- the initial state is $(3, 3, 1)$
- the goal states are $S^G = (0, 0, 0), (0, 0, 1)$
- each actions have the same cost, is negligible
- the action that can be performed are the following
  - if $B = 1$, we can subtract (in total) 1 or 2 from M or C (or both), and set $B = 0$.
  - if $B = 0$, we can add (in total) 1 or 2 from M or C (or both), and set $B = 1$.

  an action is applicable if and only if the following condition are satisfied after
  - $M \geq C$ if $M > 0$, this decode the facts that the cannibals can't be more or equals than the missionaries on the right bank if there are missionaries on the left bank
  - if $M < 3$ (some missionaries are on the left bank) then $(3 - M) > (3 - C)$ the missionaries on the left bank must be greater then the cannibals on the left bank.

To search a solution we can start from the initial state and expand the tree of possible solutions accordingly to the applicable actions.



2.2.2 Tree and Graph Search

In the search context the following terminology is used

- Search node n: Contains a state reached by the search, plus information about how it was reached.
- Path cost $g(n)$: The cost of the path reaching n.
- Optimal cost g^*: The cost of an optimal solution path. For a state s, $g^*(s)$ is the cost of a cheapest path reaching s.

- Node expansion: Generating all successors of a node, by applying all actions applicable to the node's state s. Afterwards, the state s itself is also said to be expanded.
- Search strategy: Method for deciding which node is expanded next.
- Open list: Set of all nodes that currently are candidates for expansion. Also called frontier.
- Closed list: Set of all states that were already expanded. Used only in graph search, not in tree search (up next). Also called explored set.

When we explore the state space of a problem we can maintain a closed list of all the node that has been already searched, to check for each generated new node if is already in the list (if so, we discard it). If such list is used, the search is called **graph search**, else, if the same state may appear in many search nodes, is called **tree search**. The tree search doesn't use a list so require less memory.

When we analyze a search algorithm, we are interested in various properties

- **Completeness**: the algorithm is guaranteed to find a solution (it there are one).
- **Optimality**: the returned solution is guaranteed to be optimal.
- **Time Complexity**: How long does it take to find a solution? (Measured in generated states).
- **Space Complexity**: How much memory does the search require? (Measured in states).
- **Branching Factor**: The number b of how many successor a state may have.
- **Gal depth**: the number d of action required to reach the shallowest (nearest to the initial state) goal state.

2.3 Blind Search

We talk about *blind search* if the problem does not require any input beyond the problem API. Does not require any additional work from the programmer. For each node n in the search context, we define the following data structure:

- n.State is the state which te node contains
- n.Parent is node in the search tree that generated this node
- n.Action is the action that was applied to the parent to generate the node
- n.PathCost, also denoted $g(n)$, is the cost of the path from the initial state to the node (as indicated by the parent pointers).

On a node we can perform the following operations

- Solution(n) returns the path from the initials state to n
- ChildNode(n, a) returns the node n corresponding to the application of action a in state n.State.

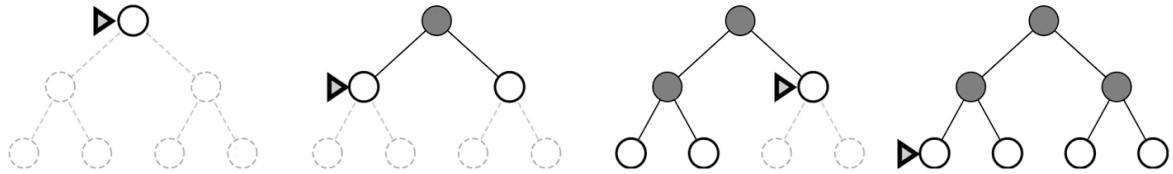
We also have the open list (called frontier) where we can perform the following actions:

- Empty?(frontier) returns true if and only if there are no more elements in the open list.
- Pop(frontier) returns the first element of the open list, and removes that element from the list.
- Insert(element,frontier) inserts an element into the open list.

The insert function can put the element in front, in the last positions, or in other positions, it depends from the implementation (different implementations yield different search strategies).

2.3.1 Breadth-First Search

The strategy is to expand nodes in the order they were produced, as a FIFO queue, we expand the shallowest unexpanded node.



This algorithm 2 is complete and optimal (in case of unit cost function). We are using the black box *API*.

Algorithm 2 Breadth-First Search

```
Require: problem
  node ←InitialState()
  if GoalTest(node.State) then
      return Solution(node)
  end if
  frontier ← a FIFO queue with node in it
  explored ← an empty set
  while true do
      if Empty?(frontier) then
          return Failure
      end if
      node ←pop(frontier)
      add node.State in explored
      for each action in Actions(node.State) do
          child ←ChildNode(node, action)
          if child.State is not in explored or frontier then
              if GoalTest(child.State) then
                  return Solution(child)
              end if
              frontier ←Insert(child,frontier)
          end if
      end for
  end while
```

Let b to be the maximum branching factor and d the depth of the shallowest goal state, an upper bound (in the worst case) for the number of nodes generated (time complexity) is

$$b + b^2 + b^3 \cdots + b^d \in O(b^d) \tag{2.11}$$

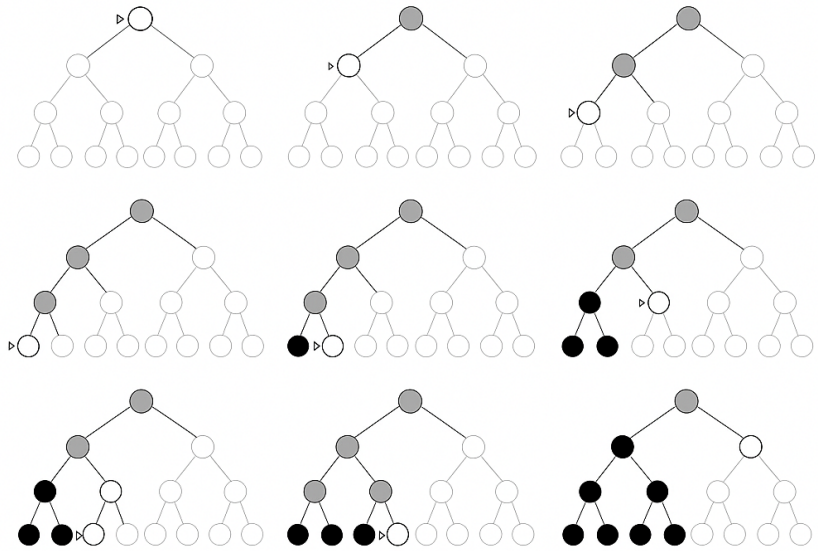
the same for the space complexity since all generated nodes are kept in memory. Let's see an example, assume that $b = 10$, the agent can generate 10^4 nodes per second, and each node has a size of 1 kilobyte, we have the following data:

Depth	Nodes	Time	Memory
2	110	0.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes

The critical resource for this method is the memory.

2.3.2 Depth-First Search

The strategy is to expand the most recent explored node, as a LIFO queue, we expand the deepest unexpanded node.



The algorithm is not complete since it may take infinite time, since there is no check for cycles along the branches. It's not optimal since he chooses a direction and looks for a path to a goal state. Is typically implemented as a recursive function, as in algorithm 4.

Algorithm 3 Depth-First Search

Require: *problem*, a node n
  if `GoalTest`(n.`State`) **then**
    return empty action sequence
  end if
  for each *action* in `Actions`(*node*.`State`) **do**
    $n' \leftarrow$ChildNode(*node*,*action*)
    result$\leftarrow$Depth-First Search(*problem*,n')
    if result$\neq$ failure **then**
      return *action* $\circ$ result
    end if
  end for
  return failure

With *action* $\circ$ result is denoted the concatenation of actions. About the space complexity, this methods stores only a single path of actions, from the root to a leaf node, since once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

If m is the maximal depth reached, the space occupied is in $O(bm)$. About the time complexity, in the worst case the nodes generated is in $O(b^m)$.

2.3.3 Uniform-Cost Search

This methods is equivalent to the well known Dijkstra's algorithm. We expand the node with the lowest path cost $g(n)$, the frontier is ordered by the path cost, with the lowest first. It differs from the BFS since a test is added to check if a better path is found to a node currently on the frontier.

Theorem 1 *The Uniform-Cost search algorithm is optimal, since the Dijkstra's algorithm is optimal, and Uniform-cost search is equivalent to Dijkstra's algorithm on the state space graph.and*

The algorithm is complete if we assume that, the costs are strictly positive and the state space is finite. The time and space complexity are

$$O(b^{1+\lfloor g^*/*\rfloor}) \tag{2.12}$$

Algorithm 4 Uniform-Cost Search

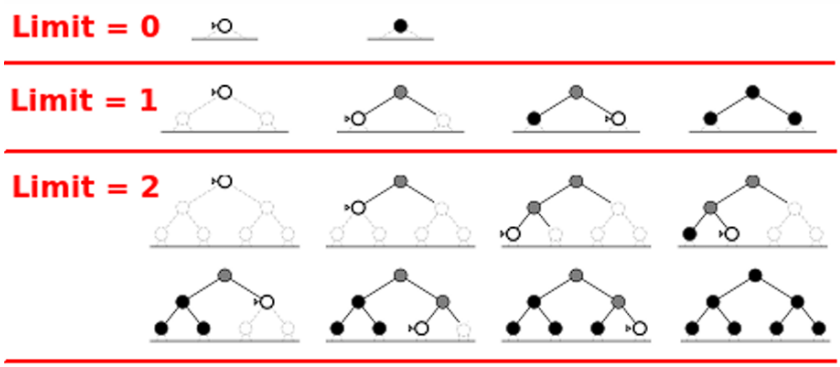
Require: *problem*
  node ←`InitialState()`
  $frontier$ ← priority queue ordered by ascending g
  explored ← empty set of states
  while true **do**
    if Empty?($frontier$) **then**
      return failure
    end if
    n ←Pop($frontier$)
    if `GoalTest`(n.State) **then**
      return Solution(n)
    end if*explored* ← *explored* ∪ n.State
    for each *action* in `Actions`(*node*.`State`) **do**
      n' ←ChildNode(n,a)
      if n'.State∉ *explored*∪States($frontier$) **then**
        $frontier$ ←insert(n',$frontier$)
      else if $\exists n'' \in frontier$: n''.State=n'.State∧ $g(n') < g(n'')$
        replace n'' with n' in $frontier$
      end if
    end for
  end while

where

- g^* is the cost of an optimal solution
- $\epsilon = \min c$ is the positive cost of the cheapest action, the minimum of the function c.

2.3.4 Iterative Deepening Search

This is an altered version of the Depth-First Search algorithm, where we define a predetermined depth limit, and apply the DFS in function of that limit, iteratively applying this by increasing the depth limit each time.



We split the algorithm in three different function.

Algorithm 5 Iterative Deepening Search

Require: *problem*
  for $depth = 0, 1 \ldots \infty$ **do**
    result ←Depth Limited Search(*problem*,*depth*)
    if *result* ≠cutoff **then**
      return *result*
    end if
  end for

Algorithm 6 Depth Limited Search

Require: $problem, limit$
  $node \leftarrow$ `InitialState()`
  return Recursive DLS($node$,$problem$,$limit$)

The algorithm is complete since we are keep searching until a solution is found, is also optimal if all the cost are unitary, the space complexity is $O(bd)$. The time complexity is in $O(b^d)$, this methods combines the advantages of breadth-first and depth-first search. It is the preferred blind search method in large state spaces with unknown solution depth.

Algorithm 7 Recursive DLS

Require: $n, problem, limit$
  if `GoalTest(`n`.State)` **then**
    return empty action sequence
  end if
  if $limit$==0 **then**
    return cutoff
  end if
  $cutoffOccurred \leftarrow$false
  for each $action$ in `Actions(`$node$`.State)` **do**
    $n' \leftarrow$ChildNode($node$,$action$)
    result$\leftarrow$Recursive DLS($problem$,n')
    if $result$==cutoff **then**
      $cutoffOccurred \leftarrow$true
    else
      if result$\neq$ failure **then**
        return $action \circ$ result
      end if
    end if
  end for
  if $cutoffOccurred$ **then**
    return cutoff
  end if
  return failure

The following table is a summary of the methods that we considered in this section, confronting the time and space complexity.

Criterion	BFS	Uniform Cost	DFS	Depth Limited	Iterative Deeping
Completness	Yes, if a is finite	Yes, if a is finite and action costs is positive	No	No	Yes, if a is finite
Optimality	Yes if action costs are 1	Yes	No	No	Yes if action costs are 1
Time Complexity	$O(b^d)$	$O(b^{1+\lfloor g^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space Complexity	$O(b^d)$	$O(b^{1+\lfloor g^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$

where

- b: finite branching factor
- d: goal depth
- m: maximum depth of the search tree

- l: depth limit
- g^* : optimal solution cost
- $\epsilon > 0$: minimal action cost.

2.4 Informed Search

Definition 6 *Let* Π *to be a problem with* S *the set of states, a* ***heuristic*** h *is a function*

$$h : S \to \mathbb{R}^+ \cup \{\infty, 0\}$$

such that, if s *is a goal state, then* $h(s) = 0$.

A heuristic function h^* is perfect if $h^*(s)$ is exactly the cost of a cheapest path from s to a goal state, and $h^*(s) = \infty$ if no such path exists. h^* is also called the *goal distance* of s.

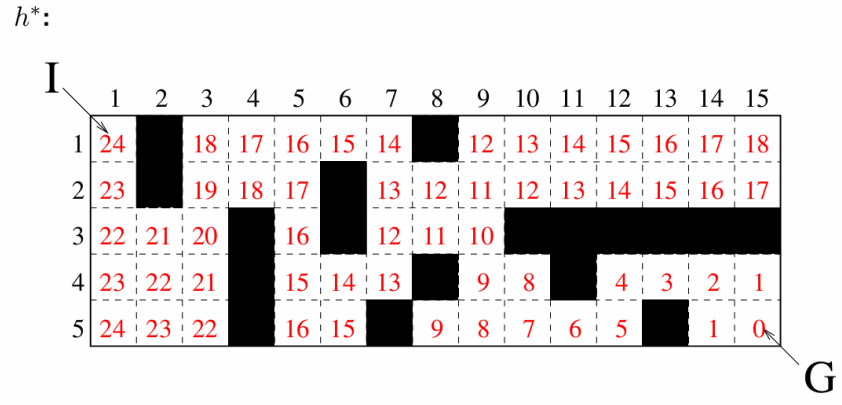
h depend only on the state s and not on the search node (the path taken to reach s don't affect $h(s)$). If n is a search node, we usually write $h(n)$ instead of $h(n.State)$ as an abuse of notation.

The purpose of h is to estimate how far we are from a solution, clearly time is needed to calculate the value of h, ideally we would like a function h that is very precise in the estimate, and which is quick to calculate, these two objectives are generally in conflict with each other.

Given a problem Π, a heuristic function h for Π can be obtained as goal distance within a simplified (relaxed) problem Π'. A classic example is using Euclidean distance as a heuristic function in a graph representing points on a map.

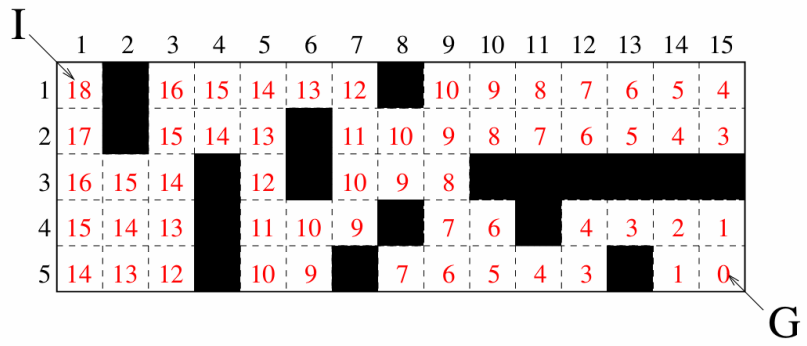
Another example is when we have to reach a goal in discrete grid with some "pitfalls" to avoid.

h^*:



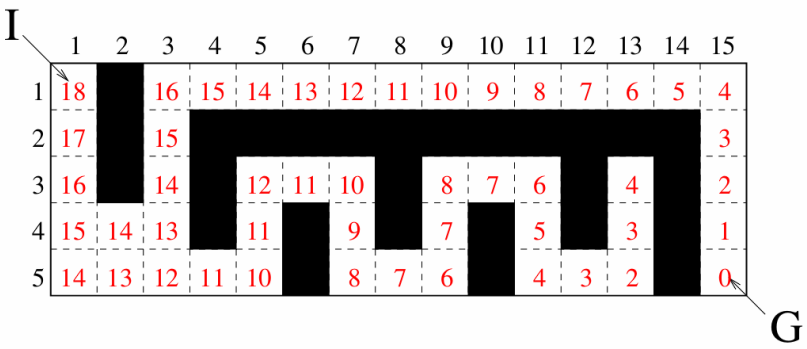
in each cell is presented the value of the perfect heuristic h^*. A reasonable and accurate heuristic may be the Manhattan distance.

Manhattan Distance, "accurate h":



For some specific grid configurations this heuristic might be inaccurate:

Manhattan Distance, "inaccurate h":



Definition 7 *Let h^* to be a perfect heuristic. A heuristic h is **admissible** if $\forall s \in S$ we have $h(s) \leq h^*(s)$.*

An admissible heuristic function never overestimates the cost to reach the goal.

Definition 8 *A heuristic h is **consistent** if, for all transition $s \xrightarrow{a} s'$ in Θ, we have $h(s) - h(s') \leq c(a)$.*

The consistency is important, when applying an action a, the heuristic value cannot decrease by more than the cost of a.

Proposition 1 *If h is consistent then is admissible.*

Proof: The proof is done by induction.

- Base case: let s to be the goal state, so $h(s) = h^*(s) = 0$, so $h(s) \leq h^*(s)$ trivially holds.
- Assume that the claim holds for all states s' where the cheapest path to the goal state have length n. Let s to be a state with goal length $n+1$, which have the first transition

  $$s \xrightarrow{a} s'$$

  so

  $$h(s) \leq h(s') + c(a) \tag{2.13}$$

  by the inductive hypothesis:

  $$h(s') \leq h^*(s') \tag{2.14}$$

  by construction:

  $$h^*(s) = h^*(s') + c(a) \tag{2.15}$$

by combining the equations:

$$\begin{cases} h(s) \leq h(s') + c(a) \\ h(s') \leq h^*(s') \\ h^*(s) = h^*(s') + c(a) \end{cases} \implies h(s) \leq h^*(s) \tag{2.16}$$

■

The Euclidean distance for the map-travel problem is consistent.

Systematic search vs. local search:

- **Systematic search strategies:** No limit on the number of search nodes kept in memory at any point in time.

  → Guarantee to consider all options at some point, thus complete.

- **Local search strategies:** Keep only one (or a few) search nodes at a time.

  → No systematic exploration of all options, thus incomplete.

Tree search vs. graph search:

- For the systematic search strategies, we consider graph search algorithms exclusively, i.e., we use duplicate pruning.
- There are tree search versions of these algorithms. These are easier to understand, but aren't used in practice. (Maintaining a complete open list, the search is memory-intensive anyway.)

The informed search uses problem-specific knowledge, by using an evaluation function f at each node, that function estimate the "desirability" of expanding that node, and we choose to expand the most desirable unexpanded node

$$\arg \max_{\text{unexpanded } n} f(n).$$

We usually implement this with a sorted queue ordered based on the desirability of expansion, this queue is called **frontier**.

2.4.1 Greedy Best-First Search

We use the heuristic function f as the evaluation function and we expand the node that is closest to the goal. This algorithm is complete thanks to the assumption that the state space is finite, but is not

Algorithm 8 Greedy Best-First Searcj

Require: A problem Π
  $n \leftarrow$ a node n such that $n.State = \Pi.InitialState$
  $frontier \leftarrow$ a priority queue ordered by ascending h
  $explored \leftarrow$empty set of states
  while True do
    If $frontier$ is empty **Then Return** failure
    $n \leftarrow Pop(frontier)$
    If $n.State$ is the goal state **Then Return** $Solution(s)$
    $explored \leftarrow explored \cup n.State$
    for each action a in $\Pi.Actions(n.State)$ **do**
      $n' \leftarrow ChildNode(\Pi, n, a)$
      If $n'.State \notin explored \cup States(frontier)$ **Then** $insert(n', h(n'), frontier)$
    end for
  end while

optimal, there exists an algorithm that is optimal and complete.

2.4.2 The A* Algorithm

In A* the evaluation function is

$$f(n) = h(n) + g(n) \tag{2.17}$$

where h is the heuristic and g is the cost needed so far to reach the node, we consider a frontier ordered by ascending $g + h$. TODO

Algorithm 9 A*

Require: A problem Π
  $n \leftarrow$ a node n such that $n.State = \Pi.InitialState$
  $frontier \leftarrow$ a priority queue ordered by ascending $g + h$
  $explored \leftarrow$empty set of states
  while True do
    if $frontier$ is empty **Then Return** failure
    $n \leftarrow Pop(frontier)$
    if $n.State$ is the goal state **Then Return** $Solution(s)$
    $explored \leftarrow explored \cup n.State$
    for each action a in $\Pi.Actions(n.State)$ **do**
      $n' \leftarrow ChildNode(\Pi, n, a)$
      if $n'.State \notin explored \cup States(frontier)$ **then**
        $insert(n', h(n') + g(n'), frontier)$
      else if $\exists n''$ s.t. $n''.State = n'.State$ and $g(n') < g(n'')$
        Replace n'' in $frontier$ with n'
      end if
    end for
  end while

2.5 Local Search

TODO

2.6 Adversarial Search

One of the oldest sub-areas of AI is *Game Planning*, we model games as search problems that takes in account the competition between two opponents (such as chess). We consider simple games that satisfies the following restrictions:

- the set of game states is discrete
- the number of possible moves at each step is finite
- the game state is fully observable and each move's outcome is deterministic
- there are only two players
- the players playing in alternating turns
- the utility function u must be maximized from one player and minimized from the other
- there are no infinite runs of the game, after a finite number of steps, the game must end.

We consider only zero-sum games, where the two players play with the same conditions, without favor. We denote Max and Min the two players.

Definition 9 *A **game state space** is a 6-tuple $\Theta = (S, A, T, I, S^T, u)$ where*

- S is the set of states, can be partitioned in $S = S^{Max} \cup S^{Min} \cup S^T$, denoting the state where Max or Min plays.
- A is the set of actions, can be partitioned in $A = A^{Max} \cup A^{Min}$

  - A^{Max} is the set of actions that Max can take, A^{Min} is the set of actions that Min can take.
  - for $a \in A^{Max}$, if $s \xrightarrow{a} s'$ then $s \in S^{Max}$ and $s' \in S^{Min} \cup S^T$
  - for $a \in A^{Min}$, if $s \xrightarrow{a} s'$ then $s \in S^{Min}$ and $s' \in S^{Max} \cup S^T$
- T is the set of deterministic transition relation
- I is the initial state
- S^T are the set of terminal states
- $u : S^T \to \mathbb{R}$ is the utility function

Definition 10 *A **strategy** for Max is a function $\sigma^{Max} : S^{Max} \to A^{Max}$ such that, a is applicable to s if $\sigma^{Max}(s) = a$. Analogous for Min with σ^{Min}.*

σ^{Max} (or σ^{Min}) is defined for all states in S^{Max} (or S^{Min}), since the agent don't know how the opponent will react he needs to prepare for all possibilities. A strategy is optimal if it yields the best possible utility assuming perfect opponent play. For an adversarial search problem there are three types of solutions:

- **ultra weak**: Prove whether the first player will win, lose or draw from the initial position, given perfect play on both sides.
- **weak**: Provide a strategy that is optimal from the beginning of the game for one player against any possible play by the opponent.
- **strong**: Provide a strategy that is optimal from any valid state, even if imperfect play has already occurred on one or both sides.

Computing a strategy is often unfeasible, the number of reachable states are big, in chess, there are 10^{40} possibile board states. We will consider a *Black Box* description of the games.

2.6.1 Minimax Search

This is the canonical algorithm to solving games, by computing an optimal strategy. Remember that the player Max attempts to maximize the utility function $u(s)$ of the terminal state that will be reached during play, when Min attempts to minimize it.

We describe the algorithm playing as Max, and we assume that the opponents will play by trying always to minimize the utility function. Starting from the actual state, the algorithm explores all the possible outcomes from all possible feasible sequences of moves, expanding a tree.

Algorithm 10 Minimax

Require: s
  $v \leftarrow$MaxValue(s)
  return an action $a \in$`Actions`(s) yielding value v

Algorithm 11 MaxValue

Require: s
  if TerminalTest(s) **then**
    return $u(s)$
  end if
  $v \leftarrow -\infty$
  for each $a \in$`Actions`(s) **do**
    $v \leftarrow \max(v$,MinValue(ChildState(s,a)))
  end for
  return v

Consider the tree in figure 2.3, starting from the root, the player consider all possible outcomes, the leaf are terminating states, assuming that the opponents will always tries to minimize u:

Algorithm 12 MinValue

Require: s
  if TerminalTest(s) **then**
    return $u(s)$
  end if
  $v \leftarrow +\infty$
  for each $a \in$`Actions`(s) **do**
    $v \leftarrow \min(v,\text{MaxValue}(\text{ChildState}(s,a)))$
  end for
  return v

- if Max choose the left branch, Min will choose the action that leads to the terminal state with $u(s) = 3$
- if Max choose the middle branch, Min will choose the action that leads to the terminal state with $u(s) = 2$
- if Max choose the right branch, Min will choose the action that leads to the terminal state with $u(s) = 2$

So the best mooves is the one that leads to the left branch.

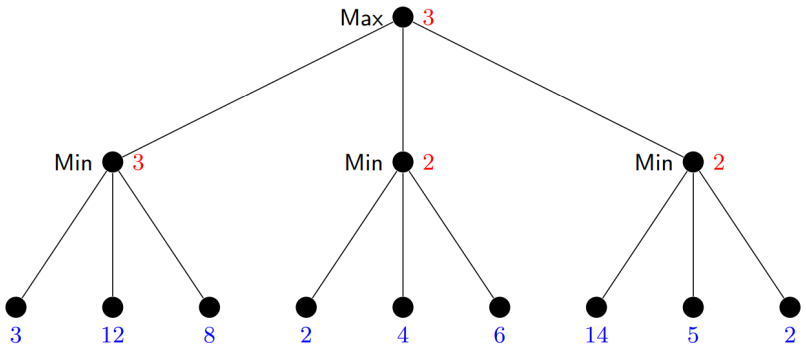


Figure 2.3: Game's tree

This is the simplest possible game search algorithm, but in practice is unfeasible since the search tree are way too large to expand. The minimax algorithm is:

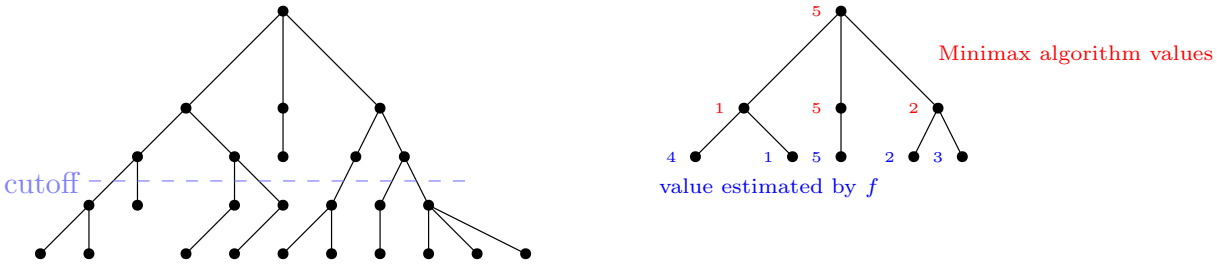
- Complete, if the tree is finite.
- Optimal against an optimal opponent.

Since it is a depth first search, the time complexity is $O(b^m)$, where b is the branching factor and m is the depth of the solution. The space complexity is $O(bm)$. In chess, $b \simeq 35$ (possible moves approximately) and a reasonable game terminates in 80 turns, so $m \simeq 80$, in such a case the state space is $O(35^{80})$ (it's impossible in practice to expand the tree).

2.6.2 Evaluation Functions

Since the Minimax game tree are too big, we impose a depth limit d (called horizon) on the search, and we apply an evaluation function to the non-terminal states in that horizon. An evaluation function $f : S \to \mathbb{R}$ should work to estimate the actual value reachable from a state such as in the unlimited-depth Minimax. If a state is terminal, we use the actual value u. We want f to be accurate and fast to compute.

While applying this algorithm, we can consider a depth limit d, evaluate f on all the states at that depth, and then considering these cut-off tree and apply the Minimax algorithm.



Usually, the evaluation function is a linear weighted function, such as

$$f(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{2.18}$$

where $f_1 \dots f_n$ are features extracted from the state s, designed by human experts of the considered game/domain, and $w_i \in \mathbb{R}$ are real weights, that can be learned automatically (with machine learning algorithms). For example, features in the tetris game could be the number of holes in the block grid, or the maximum height reached by a placed block, as shown in figure 2.4.

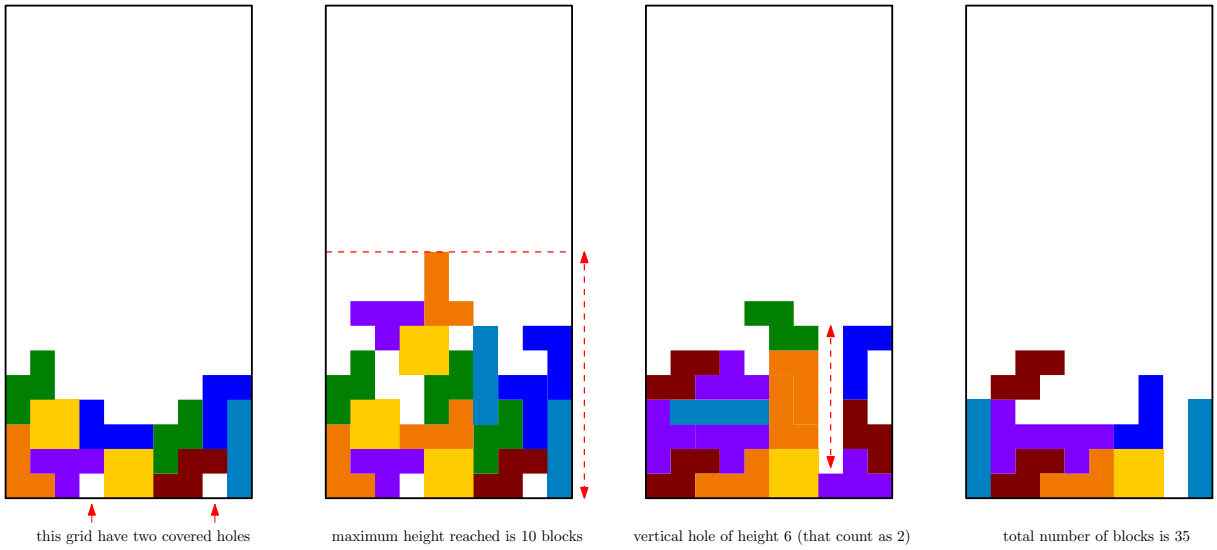


Figure 2.4: Grid's properties

This algorithm is critical since critical aspects of the game may be cut-off by the horizon. Since we want to search *as deeply as possible in a given time*, we could apply the *iterative deepening search*, until the time's up, and then return the solution of the deepest completed search.

A better solution is called *Quiescence search*, Instead of using a fixed search depth d, quiescence search dynamically adapts the depth to handle "unquiet" positions where the evaluation function's value is likely to fluctuate quickly—typically due to immediate, forcing moves like captures or checks.

For example, in Chess, if a piece exchange is underway, the search continues past the standard depth limit until a "quiet" state is reached, ensuring the final evaluated position is stable and doesn't miss an immediate, significant change in material or safety, thus leading to a much more accurate evaluation.

2.6.3 Alpha-Beta Pruning

There is a way to save time during the computation, we can cut-off some branches of the search for which we know at priori that will not lead to the solution. Consider the tree in figure 2.5.

THe left branch leads to a utility at least n, the right branch have one sub-branch with utility value $m < n$, so we can say for sure that the left tree will not provide values for the utility grater than m (since it's the Min turn), at this point we can discard the left branch and continue with the right one.

We consider two additional variables α, β defined on each node during the search, such that:

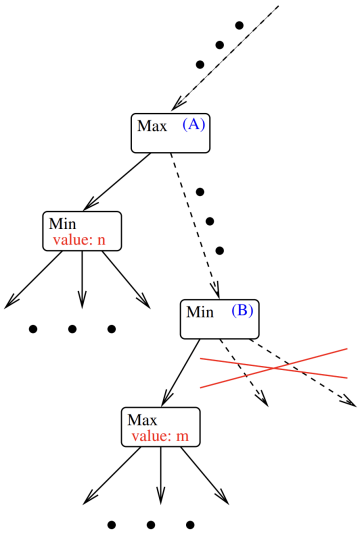


Figure 2.5: Let's say $n > m$

- for each node n, α is the highest utility that search has already found for Max on its path to n.
  - In a Min node, if one of the successors has a utility value less or equal than α, we can stop considering n and cutting it off (pruning out its remaining successors).
- We can consider a dual method for Min, for each node n, β is the lowest utility that search has already found for Min on its path to n.
  - in a Max node, if one of the successors has a utility value greater or equal than β, we can stop considering n and cutting it off (pruning out its remaining successors).

Algorithm 13 AlphaBetaPruning

Require: s
  $v \leftarrow$MaxValue(s,$-\infty$,$+\infty$)
  return an action $a \in$`Actions`(s) yielding value v

Algorithm 14 MaxValue

Require: s, α, β
  if TerminalTest(s) **then**
    return $u(s)$
  end if
  $v \leftarrow -\infty$
  for each $a \in$`Actions`(s) **do**
    $v \leftarrow \max(v$,MinValue(ChildState(s,a)),α,$\beta)$
    $\alpha \leftarrow \max(\alpha, v)$
    if $v \geq \beta$ **then return** v
  end for
  return v

Algorithm 15 MinValue

Require: s, α, β
  if TerminalTest(s) **then**
    return $u(s)$
  end if
  $v \leftarrow +\infty$
  for each $a \in$`Actions`(s) **do**
    $v \leftarrow \min(v,\text{MaxValue}(\text{ChildState}(s,a)),\alpha,\beta)$
    $\beta \leftarrow \min(\beta, v)$
    if $v \leq \alpha$ **then return** v
  end for
  return v

2.6.4 Monte-Carlo Tree Search

TODO

CHAPTER

3

CONSTRAINT SATISFACTION PROBLEMS

Definition 11 *A **CSP** (constraint satisfaction problem) is composed by a set of variables, each associated with it's domain, and a set of constraints over these variables, the goal is to find an assignment of variables so that every constraint is satisfied.*

In SuDoKu, the variables are the content of the cells, the domain of each cell is $\{1, 2, \dots 9\}$, and the constraints are that, each number should appear only once in each row, column or block. Another CSP problem is the *graph coloring* problem, where we should assign one of the k possible color to each node, such that two adjacent nodes must have different colors (this problem is NP-hard for $k = 3$).

3.1 Constraint Networks

Definition 12 *A **Binary Constraint Network** is a tuple $\gamma = (V, D, C)$ where:*

- $V = \{v_1, v_2 \dots v_n\}$ is a finite set of variables.
- $D = \{D_{v_1}, D_{v_2} \dots D_{v_n}\}$ is a corresponding set of finite domains.
- C is a set of binary relations that models the constraints. A relation is denoted C_{uv} with u, v variables in V.
$$C_{uv} \subseteq D_u \times D_v$$
If $C_{uv} \in C$ and $C_{xy} \in C$, then $\{u, v\} \neq \{x, y\}$ (no redundancy).

C_{uv} defines the permissible combined assignments to u and v. For example, if

$$u, v \in V \tag{3.1}$$
$$v' \in D_v \tag{3.2}$$
$$u' \in D_u \tag{3.3}$$
$$C_{uv} \subseteq D_v \times D_u \in C \tag{3.4}$$
$$(v', u') \notin C_{uv} \tag{3.5}$$

the assignment $v = v', u = u'$ violates the constraints. Let's see an example, we consider the map of australia show in figure 3.1. We want to colorate each state with one of 3 possible colors, without having two adjacent state with the same color.

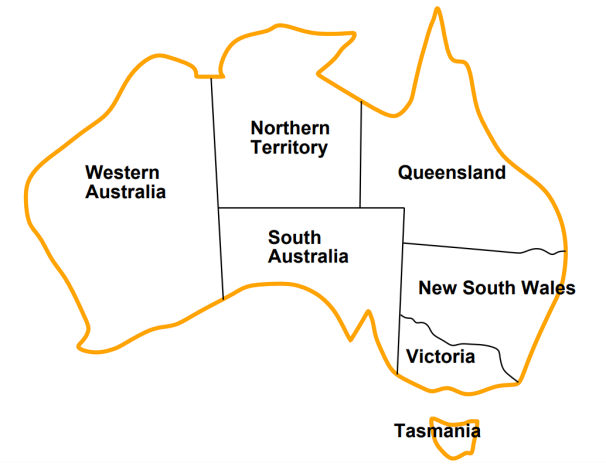


Figure 3.1: Coloring Australia

The variables are the states

$$V = \{WA, NT, SA, Q, NSW, V, T\}$$

The domain for each variable $v \in V$ is $D_v = \{red, green, blue\}$. If all the variables have the same domain

$$\forall v \neq u, D_v = D_u \tag{3.6}$$

we do an notation abuse saying that the domain of the problem is $D = \{red, green, blue\}$. For each adjacent couple of states u, v, there is a constraint

$$C_{uv} = \{(d, d') \in D \times D \ : \ d \neq d'\}.$$

Constraint Networks can be extended

- the domains D_v may be infinite or uncountable, like $D_v = \mathbb{R}$.
- the constraints may have an arity higher than 2, with relations over $k > 2$ variables, like in CNF satisfiability.

A constraint may be **unary**: $C_v \subseteq D_v$, in this case the domain of a single variable is restricted, equivalently to setting $D_v = C_v$.

There exists CSP solvers, generic algorithms that find assignment for constraints problem, by taking in input a constraint network as the generic language to describe the problem. The core exercise in this context is to model a problem with a constraint network.

Definition 13 *Let $\gamma = (V, D, C)$ to be a constraint network, a **partial assignment** is a function*

$$a : V' \to \bigcup_{v \in V} D_v$$

where

- $V' \subseteq V$
- $a(v) \in D_v$ *for all* $v \in V'$

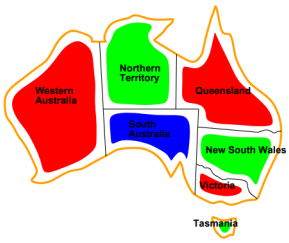
*if $V' = V$ is a **total assignment** (or just assignment). A partial assignment assigns some variables to values from their respective domains. A total assignment is defined on all variables.*

Definition 14 *Let $\gamma = (V, D, C)$ to be a constraint network and let a to be a partial assignment, we say that a is **inconsistent** if there exists variables $u, v \in V$ such that*

- a is defined on u and v

- $C_{uv} \in C$
- $(a(u), a(v)) \notin C_{uv}$.

If a partial assignment is not inconsistent, is **consistent**. An assignment a is a **solution** if it's total and consistent. A constraint network γ is **solvable** if there exists a solution a, otherwise is **unsolvable**.



- **Variables:** $V = \{WA, NT, SA, Q, NSW, V, T\}$.
- **Domains:** All $v \in V$: $D_v = D = \{red, green, blue\}$.
- **Constraints:** C_{uv} for adjacent states u and v, with $C_{uv} = \{(d, d') \in D \times D \mid d \neq d'\}$.
- **Solution:** $\{WA = red, NT = green, SA = blue, Q = red, NSW = green, V = red, T = green\}$.

Definition 15 *Let $\gamma = (V, D, C)$ to be a constraint network, and let a to be a partial assignment. a can be **extended** to a solution if there exists a solution a' that agrees with a on the variables where a is defined.*

- the domain of a is $V' \subset V$
- for each $v \in V'$, we have that $a(v) = a'(v)$.

Proposition 2 *if a can be extended to a solution, then it's consistent (the opposite doesn't necessary holds).*

Given a constraint network $\gamma = (V, D, C)$, if $n = |V|$ and $\forall D_v \in D$, $|D_v| = k$, the number of total assignments is k^n. There are at most n^2 constraints, each constraint have size at most k^2, the number of assignments is *exponentially bigger* than the size of γ.

Theorem 2 *The problem to decide if a constraint network $\gamma = (V, D, C)$, is solvable is* NP*-complete.*

3.2 Naive Backtracking

The following algorithm is simple and used to find a solution for a constraint network. The method is recursive, the first step given assignment a is the empty assignment. With *Backtracking*, we mean the

Algorithm 16 NaiveBacktracking

```
Require: γ = (V, D, C), a
  if a is inconsistent then
      return inconsistent
  end if
  if a is a total assignment then
      return a
  end if
  let V' to be the domain of a
  select some v ∈ V − V' (select a variable for which a is not defined)
  for each d ∈ D_v in some order do
      a' = a ∪ {v = d}
      a'' =NaiveBacktracking(γ,a')
      if a'' is not inconsistent then
          return a''
      end if
  end for
  return inconsistent
```

action of recursively instantiate variables one-by-one, backing up out of a search branch if the current partial assignment is already inconsistent. Note that in the algorithm 16 we are iterating the possible

values in D_v in a specific order that is not described yet, and is not described which variable are we picking at each recursion step.

The Naive Backtracking algorithm is

- simple to implement
- much more efficient than enumerating all possible assignment
- complete, if there is a solution, backtracking will find it.

This algorithm can't predict if an assignment a can't be extended to a solution unless a is already inconsistent.

3.2.1 About the Order

The ordering in the for loop of algorithm 16 can influences the search space size. If no solution exists below current node, the order doesn't matter, the algorithm will search the whole sub-tree anyway, otherwise, if a solution does exist below current node, by choosing the "correct" value, then no backtracking is needed.

A common strategy is to consider the variable "most constrained", at each recursion step, we pick the variable v that minimize the size of the set

$$\{d \in D_v \ : \ a \cup \{v = d\} \text{ is consistent } \} \tag{3.7}$$

by choosing a most constrained variable v first, we reduce the branching factor (number of sub-trees generated for v) and thus reduce the size of our search tree.

Another common heuristic is to choose the variable that is the "most constraining", we choose v that maximize the size of the set

$$\{u \in V \ : \ a(u) \text{ is undefined } \wedge C_{uv} \in C\} \tag{3.8}$$

by choosing a most constraining variable first, we detect inconsistencies earlier on and thus reduce the size of our search tree.

For the order of the value in the for loop, we can choose the least constraining value first, for a variable v, we choose the value $d \in D_v$ that minimize the size of the set

$$\{d' \ : \ d' \in D_u \wedge a(u) \text{ is undefined } \wedge C_{uv} \in C \wedge (d', d) \notin C_{uv}\} \tag{3.9}$$

by choosing a least constraining value first, we increase the chances to not rule out the solutions below the current node.

3.3 Inference

Given a constraint networks, we would like to obtain an equivalent network with more constraint, without losing any feasible solution, in such case, we are giving a tighter description of the problem, and we obtain a network with a smaller number of consistent partial assignments. Already known constraints may implies additional (unary or binary) constraints.

Definition 16 *Let $\gamma = (V, D, C)$ and $\gamma' = (V, D', C')$ to be two constraint networks, sharing the same set of variables. We say that γ are **equivalent** to γ', and we denote*

$$\gamma \equiv \gamma'$$

if every solution (assignment of variables) of γ is a solution of γ' and vice versa.

Definition 17 *Let $\gamma = (V, D, C)$ and $\gamma' = (V, D', C')$ to be two constraint networks, sharing the same set of variables. We say that γ' is **tighter** than γ and we denote*

$$\gamma' \sqsubseteq \gamma$$

if

- $\forall v \in V, \quad D'_v \subseteq D_v$
- $\forall u, v \; such \; that \; u \neq v \; either \; C_{uv} \notin C \; or \; C'_{uv} \subseteq C_{uv}$.

If at least one of these inclusions is strict, we say that γ' is **strictly tighter** than γ:

$$\gamma' \sqsubset \gamma.$$

Theorem 3 *Let γ and γ' to be two constraint networks, if*

- $\gamma' \equiv \gamma$
- $\gamma' \sqsubset \gamma$

*then γ' has the same solutions as γ but fewer consistent partial assignments than γ. In such case, γ' is a **better encoding** of the problem.*

We define *inference* the procedure to derive a tighter equivalent network. We want to use it to define an algorithm that finds a solution for a constraint network. We will incorporate inference in the backtracking algorithm, at every recursive call, a complex inference procedure leads to

- a smaller number of search nodes
- larger runtime needed at each node

We encode partial assignments as unary constraints

- if $a(v) = d$
- we set the constraint $D_v = \{d\}$

Algorithm 17 BacktrackingWithInference

Require: $\gamma = (V, D, C), a$
  if a is inconsistent **then**
    return inconsistent
  end if
  if a is a total assignment **then**
    return a
  end if
  $\gamma' =$ copy of γ
  $\gamma' =$ Inference(γ')
  If $\exists v \; : \; D'_v = \emptyset$ **then return** inconsistent
  let V' to be the domain of a
  select some $v \in V - V'$ (select a variable for which a is not defined)
  for each $d \in D'_v$ in some order **do**
    $a' = a \cup \{v = d\}$
    $D'_v = \{d\}$
    $a'' =$NaiveBacktracking(γ',a')
    if a'' is not inconsistent **then**
      return a''
    end if
  end for
  return inconsistent

With the function Inference we denote any procedure deriving a tighter equivalent network. One simple inference procedure called *forward checking* is described in algorithm 18.

The forward checking algorithm tightens the constraints without ruling out any solutions, it guarantees to deliver an equivalent network. The computation can be done incrementally, instead of performing the first loop, we may run only the second loop every time a new assignment $a(v) = d'$ is added.

- The forward checking procedure is cheap and useful
- In rare situations is not a good idea to use it, there are stronger inference methods, but in many cases, forward checking is the best choice.

Algorithm 18 Forward Checking

Require: γ, a
  for each v where $a(v) = d'$ is defined **do**
    for each u where $a(u)$ is undefined and $C_{uv} \in C$ **do**
      $D_u = \{d \mid d \in D_u, (d, d') \in C_{uv}\}$
    end for
  end for
  return γ

3.3.1 Arc Consistency for Stronger Inference

We now describe a properties between two variables that can be used to describe a stronger inference method than the forward checking.

Definition 18 *Let $\gamma = (V, D, C)$ to be a constraint network, a variable $u \in V$ is **arc consistent** to another variable $v \in V$ if*

- *either $C_{uv} \notin C$*
- *or for every $d \in D_u$ there exists $d' \in D_v$ such that $(d, d') \in C_{uv}$.*

When two variables in a constraint network are arc consistent, it means that the domain of each variable contains at least one value that is compatible with at least one value in the domain of the other variable, according to the constraint between them.

As an example, let's say the variable v describes the color of your shirt, that can be red, blue, yellow or green, and the variable u define the color of your pants, that can be black, white, gray or brown. The constraint is that the shirt and the pants cannot be the same color, the variable v is arc consistent to u because for every shirt color you can find a pant color and vice versa.

> **Note**: Arc consistency is a non commutative relation, if v is arc consistent to u, it's not implied directly that u is arc consistent to v.

Definition 19 *Let $\gamma = (V, D, C)$ to be a constraint network, if every variable $u \in V$ is arc consistent relative to every other variable $v \in V$, then we say that the network γ is **arc consistent**.*

We can define an inference procedure that aims to make the network γ arc consistent, by removing variable domain values. Let's say that AC(γ) is a procedure that makes γ arc consistent, in such case:

- is guaranteed that AC(γ) returns an equivalent network.
- AC(γ) is more powerful (more general) tha forward checking, in details:

$$\text{AC}(\gamma) \sqsubseteq \text{ForwardChecking}(\gamma) \tag{3.10}$$

We define the following subroutine:

Algorithm 19 Revise

Require: $\gamma, u \in V, v \in V$
  for each $d \in D_u$ **do**
    if $\neg\exists d' \in D_v$ such that $(d, d') \in C_{uv}$ **then**
      $D_u = D_u \backslash \{d\}$
    end if
  end for
  return γ

We apply pairwise Revisions in algorithm 20 and define the first version of the inference procedure that makes a constraint network γ arc consistent.

It's not hard to show that this enforce arc consistency. If there are n variables, m constraints and k is the cardinality of the biggest domain, then the time complexity is $O(mk^2 \cdot nk)$, this algorithm performs

Algorithm 20 AC_1

Require: γ changes = False
  while changes = False **do**
    for each $C_{uv} \in C$ **do**
      If D_u after applying Revise(γ, u, v reduces) **then** changes = True
      If D_v after applying Revise(γ, v, u reduces) **then** changes = True
    end for
  end while

Algorithm 21 AC_3

Require: γ
  $M = \emptyset$
  for each $C_{uv} \in C$ **do**
    $M = M \cup \{(u, v), (v, u)\}$
  end for
  while $M \neq \emptyset$ **do**
    remove any element (u, v) from M
    Revise(γ, u, v)
    if D_u has changed in the call Revise **then**
      for each $C_{wu} \in C$ where $w \neq v$ **do**
        $M = M \cup \{(w, u)\}$
      end for
    end if
  end while

redundant computation since u and v are revised even if their domains haven't changed since the last time.

There is another version of the arc consistency procedure, described in algorithm 21. At any moment during the while loop, if $(u, v) \notin M$ then u is arc consistent to v.

In the algorithm, we check the constraints where $w \neq v$ because v is the reason why D_u just changed, if v was arc consistent to u before, that continues to hold, the values just removed from D_u did not match any values from D_v anyway.

Theorem 4 *Let γ to be a constraint network with m constraint and maximal domain size k. The algorithm AC3 showed in 21 have a time complexity in $O(mk^3)$.*

Proof: Each Revise call takes $O(k^2)$, it is sufficient to prove that at most $O(mk)$ calls to revise are made. The number of calls to Revise is the number of iterations of the while-loop, which is at most the number of insertions into M.

- Consider any constraint C_{uv}
- two variables pairs corresponding to the constraint C_{uv} are inserted into M in the first for loop
- then beforehand the domain of either u or v was reduced, which happens at most $2k$ times
- thus we have $O(k)$ insertions per constraint, and $O(mk)$ insertions overall. ■

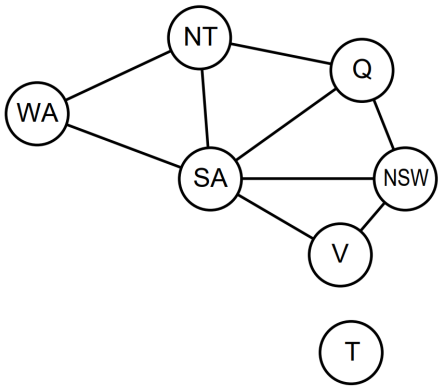
3.4 Decomposition

If γ is a network with n variables and maximal domain size k, we may search a feasible assignment by checking among all the possible ones, this will require to check $O(k^n)$ assignments in the worst case. We saw how the inference help to shrink the search space, another useful methods is called *decomposition.*

> **decomposition**: exploit the structure of a network to decompose it into smaller (and easier to solve) independent networks.

Definition 20 *Given a constraint network $\gamma = (V, D, C)$, the **constraint graph** of γ is the undirected graph, where the vertices are the variables V and there are an edge (u, v) if $C_{uv} \in C$.*

For example, the graph for the "coloring Australia" problem shown in figure 3.1 is the following:



Note how this graph is composed by two connected sub-graph, since the variable T (Tasmania) is disconnected from the other variables. The idea is to separate a network in function of his connected sub-graphs.

Theorem 5 *Let $\gamma = (V, D, C)$ to be a constraint network. Let G to be the constraint graph, we denote $\mathcal{V}(G)$ the vertices of G and $\mathcal{E}(G)$ the edges of G, we know that $\mathcal{V}(G) = V$. We consider a partition of $\mathcal{V}(G)$*

$$\mathcal{V}(G) = \bigcup_i V_i \tag{3.11}$$

such that each V_i is connected and

$$\begin{cases} v \in V_i \\ u \in V_j \\ V_i \neq V_j \end{cases} \implies \begin{matrix} (u,v) \notin \mathcal{E}(G) \\ (v,u) \notin \mathcal{E}(G) \end{matrix} \implies C_{uv} \notin C$$

so the components of the partition are disconnected from each other. Let a_i to be an feasible assignment (solution) for the variables V_i, then

$$a = \cup_i a_i \tag{3.12}$$

is a solution to γ.

The theorem states that if two parts of γ are not connected (in term of his constraint graph), then they are independent and we can find separately a solution for each connected component.

Theorem 6 *Let $\gamma = (V, D, C)$ to be a constraint network with n variables and maximal domain size k. Let G to be the constraint graph, if G is acyclic, we can find a solution for γ (or prove that γ is inconsistent) with a time complexity of $O(nk^2)$.*

The procedure is the following (assuming that a constraint network is connected, if not, we can apply the procedure to each connected component).

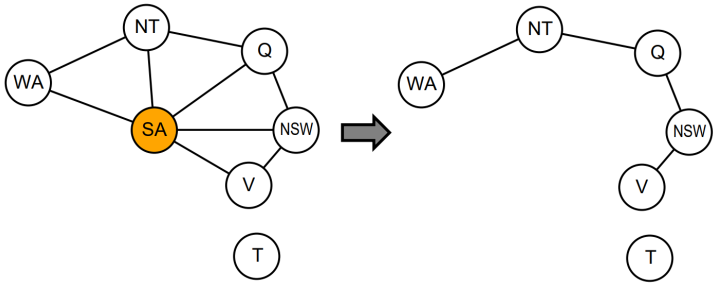
1. We consider the constraint graph of γ, denoted G. We choose an arbitrary node $v \in \mathcal{V}(G) = V$, and we consider the BFS spanning tree G' starting from v.
2. G' is a directed graph, we consider a topological ordering $v_1 \ldots, v_n$ of the variables.
3. we iterate through the variables v_i in the opposite order respect to the topological ordering: for each $i = n, n-1, n-2, \ldots, 3, 2$ (except for v_1)
   (a) we perform the procedure Revise(γ, parent(v_i), v_i)
   (b) if $D_{\text{parent}(v_i)} \neq \emptyset$, return inconsistent

   at this point, every variable is arc consistent relative to its children
4. We perform the algorithm 17 using the variable order $v_1, v_2 \ldots, v_n$.

3.4.1 Cutsets

Definition 21 *Let $\gamma = (V, D, C)$ to be a constraint network. Let G to be the constraint graph, a* ***cutset*** *$V_0 \subseteq V$ is a set such that, the graph G' where $\mathcal{V}(G') = V \backslash V_0$ is acyclic.*

A cutset is a subset of variables removing which renders the constraint graph acyclic. The graph of the "coloring Australia" problem is *almost acyclic* since, removing SA will make the graph a tree.



The following procedure use cutsets to find a solution for γ.

V_0 := a cutset; **return** CutsetConditioning$(\gamma, V_0, \emptyset)$

function CutsetConditioning(γ, V_0, a) **returns** a solution, or "inconsistent"
  γ' := a copy of γ; γ' := ForwardChecking(γ', a)
  if exists v with $D'_v = \emptyset$ **then return** "inconsistent"
  if exists $v \in V_0$ s.t. $a(v)$ is undefined **then** select such v
    else a' := AcyclicCG(γ'); **if** $a' \neq$ "inconsistent" **then return** $a \cup a'$
      else return "inconsistent"
  for each $d \in$ copy of D'_v in some order **do**
    a' := $a \cup \{v = d\}$; $D'_v := \{d\}$;
    a'' := CutsetConditioning(γ', V_0, a')
    if $a'' \neq$ "inconsistent" **then return** a''
  return "inconsistent"

TODO: Riscrivere questo algoritmo, l'identazione è ambigua

The forward checking operation is required to ensure that $a \cup a'$ is consistent in γ, the runtime is exponential in V_0. It is true that finding an optimal cutset for a graph is NP-hard, but practical approximated procedures exists.

CHAPTER

4

PROPOSITIONAL LOGIC

This chapter presents a brief summary of propositional logic. Let Σ to be a set of boolean variables, also called *Atoms*, the definition of a formula over the variables Σ is recursive.

Definition 22 *Let Σ to be a set of atoms, then*

1. *$\top$ (True) and $\bot$ (False) are Σ-formulas.*
2. *Each atom $P \in \Sigma$ is a Σ-formula.*
3. *If φ is a formula[1], then $\neg\varphi$ is a formula.*

If φ and ψ are formulas then:

1. $\varphi \wedge \psi$ is a formula (*conjunction*)
2. $\varphi \vee \psi$ is a formula (*disjunction*)
3. $\varphi \rightarrow \psi$ is a formula (*implication*)
4. $\varphi \leftrightarrow \psi$ is a formula (*equivalence*)

We recall that

$$\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi \tag{4.1}$$
$$\varphi \leftrightarrow \psi \equiv (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi) \tag{4.2}$$

Definition 23 *An **interpretation** I over Σ is an assignment $I : \Sigma \rightarrow \{0, 1\}$.*

We say that I **satisfy** a formula φ, and we write $I \models \varphi$ if the assignment makes the formula true. We can also say that I is a **model** of φ and the set of all the models of φ (all the assignments that satisfies φ) is denoted $M(\varphi)$.

Example:

$$\varphi = (P \vee Q) \iff (R \vee S) \tag{4.3}$$

the interpretation

$$I = \begin{cases} P = 1 \\ Q = 0 \\ R = 1 \\ S = 1 \end{cases} \tag{4.4}$$

[1] We abbreviate "Σ-formula" with "formula" leaving it implicit that the formula is on the set of variables defined in the context.

is a model since $I \models \varphi$:

$$(P \vee Q) \leftrightarrow (R \vee S) \implies (1 \vee 0) \leftrightarrow (0 \vee 0) \implies 1 \leftrightarrow 1 \implies \top \tag{4.5}$$

A set of formulas KB is called **Knowledge Base** and an interpretation I models KB if, for all $\varphi \in KB$ we have that $I \models \varphi$. We also say that I is a model of KB.

Definition 24 *Let φ to be a formula:*

- *φ is **satisfiable** if $\exists I$ such that $I \models \varphi$*
- *φ is **unsatisfiable** if is not satisfiable.*
- *φ is **falsifiable** if there exists I that doesn't satisfy φ.*
- *φ is a **tautology** if $\forall I$, $I \models \varphi$ (it's not falsifiable).*

Definition 25 *Two formulas φ and ψ are **equivalent***

$$\varphi \equiv \psi$$

if $M(\varphi) = M(\psi)$ (I models φ if and only if models also ψ).

Definition 26 *Let Σ to be a set of atoms, a knowledge base KB over Σ **entails** a formula φ*

$$KB \models \varphi$$

if, all the models I of KB, are models of φ.

$$M(\bigwedge_{\psi \in KB} \psi) \subseteq M(\varphi)$$

We say that the formula φ follows from KB, and would be redundant to include φ in KB. It's also important to derive new logic formula form a given knowledge base.

Theorem 7 *$KB \models \varphi$ if and only if $KB \cup \{\neg\varphi\}$ is unsatisfiable.*

Proof $\implies$: Let $KB \models \varphi$ for any $I \models KB$ we have $I \models \varphi$ so $I \not\models \neg\varphi$.
Proof $\impliedby$: If $KB \cup \{\neg\varphi\}$ is unsatisfiable then for any I such that $I \models KB$ we have $I \not\models \neg\varphi$ hence $I \models \varphi$.

Given a formula with n variables, the number of possible assignment is 2^n, so it's hard to test the satisfiability of a formula by testing all the interpretations.

Definition 27 *A formula is in **conjunctive normal form (CNF)** if is a conjunction of disjunction of literals.*[2].

$$\bigwedge_i \left(\bigvee_j L_{i,j} \right) \tag{4.6}$$

Definition 28 *A formula is in **disjunctive normal form (DNF)** if is a disjunction of conjunction of literals.*

$$\bigvee_i \left(\bigwedge_j L_{i,j} \right) \tag{4.7}$$

It is important to know that **every formula** can be expressed in CNF or DNF, or, for each formula φ, there exists an equivalent formula ψ that is a CNF or a DNF. It is possible to construct the equivalent CNF (or DNF) from φ by exploiting the equivalences such as

- $\varphi \to \psi \equiv \neg\varphi \vee \psi$
- $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$

[2] A literal is an atom P or the negation of an atom $\neg P$.

- and so on...

$((P \vee H) \wedge \neg H) \to P$	Eliminate " $\to$ "
$\neg((P \vee H) \wedge \neg H) \vee P$	Move "$\neg$" inwards
$(\neg(P \vee H) \vee H) \vee P$	Move "$\neg$" inwards
$((\neg P \wedge \neg H) \vee H) \vee P$	Distribute " $\vee$ " over " $\wedge$ "
$((\neg P \vee H) \wedge (\neg H \vee H)) \vee P$	Distribute " $\vee$ " over " $\wedge$ "
$(\neg P \vee H \vee P) \wedge (\neg H \vee H \vee P)$	

4.1 Resolution

We can construct a simple algorithm that, given a CNF formula ψ, tests if is satisfiable or not, by trying to derive from ψ new formulas: If it finds a derived formula ψ that is unsatisfiable then ψ is unsatisfiable.

We use a method called *Calculus* that, given a single *rule* (we will define a rule later), allows us to derive new formulas from a given one. If φ is derived from ψ, we write

$$\psi \vdash \varphi.$$

Definition 29 *A clause C is a disjunction of literals, like:*

$$\neg P \vee Q \vee R \vee \neg Q \tag{4.8}$$

To shrink the notation, we identify C with the set of the literals included:

$$P \vee \neg Q \text{ became } C = \{P, \neg Q\} \tag{4.9}$$

Since a CNF is a conjunction of disjunction we can identify each CNF with a set of clauses, usually denoted Δ. For example, the CNF

$$\Delta = \{\{P, \neg Q\}, \{R, Q\}\} \tag{4.10}$$

is

$$(P \vee \neg Q) \wedge (R \vee Q) \tag{4.11}$$

We denote the empty clause $\square$. An interpretation I satisfies a single clause C

$$I \models C$$

if there exists a literal $l \in C$ such that $I \models l$. I satisfies Δ

$$I \models \Delta$$

if, for all $C \in \Delta$ we have that $I \models C$.

Definition 30 *A **inference rule** is a rule that derive a new formula from a given one, the **resolution rule** is the following:*

$$\frac{C_1 \dot\cup \{l\}, C_2 \dot\cup \{\neg l\}}{C_1 \cup C_2} \tag{4.12}$$

You may wonder what this formula means and how it is applied, the symbol $\dot\cup$ identify a disjoint union, so $C_1 \dot\cup \{l\}$ is a union and is implied that $l \notin C_1$. l is a literal. The formula is written as a fraction:

- the term over the fraction line represents the clauses that we already have in our CNF
- the term under the fraction line represents a new formula that we derive

In practice, if we have two clauses $C_1' = C_1 \dot\cup \{l\}$ and $C_2' = C_2 \dot\cup \{\neg l\}$ in Δ, we can derive a disjunction $\phi = C_1 \cup C_2$.

> if you have two clauses where the same literal appears, but in one it is affirmative (l) and in the other it is negated ($\bar{l}$), you can "add them" by eliminating l and $\bar{l}$ and merging everything else.

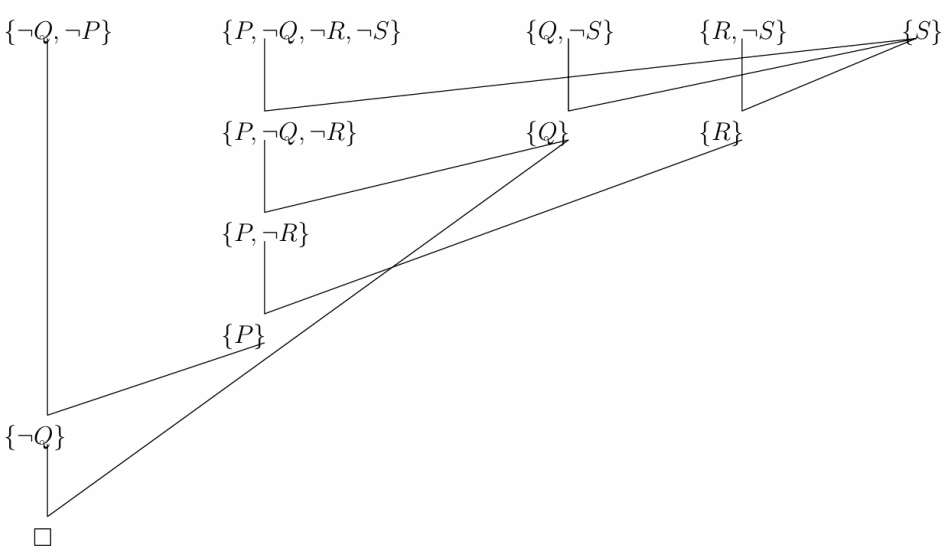
For example, if you have $\{P, \neg R\}$ and $\{R, Q\}$, you can resolve them and find a new formula $\{P, Q\}$.

Theorem 8 *The **soundness theorem** affirms that, if $\Delta \vdash D$ (the formula D can be derived by some clause in Δ with the resolution rule) then $\Delta \models D$ (the CNF Δ entails D).*

In the definition of *entailment*, is a knowledge base that "entails" a formula, and not a formula that "entails" another formula, but a knowledge base KB is equivalent to a CNF by putting in conjunction all the formulas in KB.

The resolution algorithm takes as an input a knowledge base KB and a formula ϕ and guess if $KB \models \phi$ by the following steps:

1. transform KB in a CNF
2. write the CNF as a set of clauses Δ, by adding to Δ also the clause $\{\neg\phi\}$
3. apply the resolution rule on Δ until it's derived the empty clause $\square$.



It is important to notice that $\Delta \models \varphi$ doesn't implies that $\Delta \vdash \varphi$, but, by the contradiction theorem, if we run the resolution on $KB \cup \{\neg\varphi\}$ and we derive the empty clause $\square$ ($KB \cup \{\neg\varphi\}$ is unsatisfiable) then $KB \models \varphi$.

Theorem 9 *A CNF Δ is unsatisfiable if and only if $\Delta \vdash \square$.*

4.2 The DPLL procedure

The SAT problem is to decide whenever a given propositional formula φ is satisfiable or not, as we well know, this is the most classic example of a problem NP-complete. The SAT problem is a "subset" of the constraint satisfaction problem:

- SAT can be viewed as a CSP problem in which all variable domains are Boolean, and the constraints have unbounded arity.

Every CSP problem can be encoded as a SAT problem:

- Given any constraint network γ, we can in low-order polynomial time construct a CNF formula $\varphi(\gamma)$ that is satisfiable if and only if γ is solvable.

This and many other problems can be "encoded" as a SAT formula through reduction.

A **SAT solver** is an algorithm that, given a CNF Δ, returns an interpretation I (if exists) such that $I \models \Delta$. We consider the DPLL procedure: A complete SAT solver. A SAT solver can be use in different contexts:

- We can use it to entail a new formula φ from Δ.

Algorithm 22 Davis-Putnam-Logemann-Loveland (DPLL)

```
1: function DPLL(Δ, I)
2:     /* Unit Propagation (UP) Rule: */
3:     Δ′ := a copy of Δ; I′ := I
4:     while Δ′ contains a unit clause C = {l} do
5:         extend I′ with the respective truth value for the proposition underlying l
6:         /* remove false literals and true clauses */
7:         simplify Δ′
8:     end while
9:     /* Termination Test: */
10:    if □ ∈ Δ′ then
11:        return "unsatisfiable"
12:    end if
13:    if Δ′ = {} then
14:        return I′
15:    end if
16:    /* Splitting Rule: */
17:    select some proposition P for which I′ is not defined
18:    I″ := I′ extended with one truth value for P; Δ″ := a copy of Δ′; simplify Δ″
19:    if I‴ := DPLL(Δ″, I″) ≠ "unsatisfiable" then
20:        return I‴
21:    end if
22:    I″ := I′ extended with the other truth value for P; Δ″ := Δ′; simplify Δ″
23:    return DPLL(Δ″, I″)
24: end function
```

- We need an assignment I that models Δ.

The Unit Propagation (UP) section of the algorithm correspond to the application of the following rule.

Definition 31 *The **unit resolution** is the inference rule:*

$$\frac{C \dot{\cup} \{\neg l\}, \{l\}}{C} \tag{4.13}$$

That is, if Δ contains parent clauses of the form $C \dot{\cup} \{\neg l\}$ and $\{l\}$ the rule allows to add the resolvent clause C.

Does the UP rule have the soundness property of Theorem 8? Yes, we have to show that if Δ' can be derived from Δ then $\Delta \models \Delta'$, this is true since any derivation made by unit resolution can also be done by the full resolution, which we already know has this property.

Is the UP rule complete? To be complete, we have to show that, if $\Delta \models \Delta'$, then Δ' can be derived from Δ by the UP rule, this is false, a counter example is given by the following CNF

$$\Delta = \{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$$

is unsatisfiable but the UP rule cannot derive the empty clause $\square$.

4.3 Conflict Analysis and Clause Learning

The DPLL is a systematic way of searching for a resolution proof. We define the *number of decisions* of a DPLL run as the total number of times a truth value was set by either unit propagation or the splitting rule, the following relationship between the DPLL algorithm and the resolution procedure holds:

> If DPLL returns "unsatisfiable" on Δ, then $\Delta \models \square$ with a resolution derivation whose length is at most the number of decisions.

So, DPLL is an effective practical method for conducting resolution proofs. The DPLL is a "tree resolution", this is a problem since there are some Δ whose shortest tree resolution proof is exponentially longer than their shortest (general) resolution proof. In tree resolution, every time a clause is needed, it must be derived from scratch because results aren't stored, this makes DPLL "make the same mistakes over and over again", leading to proofs that are exponentially longer than they need to be. Modern SAT solvers use Clause Learning (CDCL). When the solver hits a conflict, it analyzes the cause, creates a new clause to "remember" that mistake, and adds it to the database so it never explores that specific failing path again.

During the execution of the DPLL procedure, there are many branches of the recursive run, where for each branch we compute a partial interpretation, a literal during a branch can be

- *choice literals*: the value of an atom P is set by the splitting rule, assigning $P = 0$ or $P = 1$.
- *implied literal*: the value is set automatically by the UP rule.
- *conflict literal*: when the UP rule derive an empty clause $\square$, the branch became unsatisfiable.

Definition 32 *The **implication graph** for a branch β is a directed graph G^{impl} such that:*

- the vertices $V(G^{impl})$ are the choice literals and implied literals of the branch, plus a special vertex $\square_C$ for each clause C that becomes empty.
- the edges $E(G^{impl})$ are the following:
  - when a clause $C = \{l_1 \dots l_k, l'\} \in \Delta$ becomes unit with the implied literal l', there will be the edges

$$\neg l_1 \to l' \tag{4.14}$$
$$\vdots \tag{4.15}$$
$$\neg l_k \to l' \tag{4.16}$$

  - when a clause $C = \{l_1 \dots l_k\}$ becomes empty there will be the edges

$$\neg l_1 \to \square_C \tag{4.17}$$
$$\vdots \tag{4.18}$$
$$\neg l_k \to \square_C \tag{4.19}$$

Example

Consider

$$\Delta = \{\{P, Q, \neg R\}, \{\neg P, \neg Q\}, \{R\}, \{P, \neg Q\}\}$$

1. First step: Apply the UP rule on R with the clause $\{P, Q, \neg R\}$, and get the set:

$$\{\{P, Q, \neg R\}, \{\neg P, \neg Q\}, \{R\}, \{P, \neg Q\}\} \text{ becomes } \{\{P, Q\}, \{\neg P, \neg Q\}, \{P, \neg Q\}\} \tag{4.20}$$

   the literal R is an implied literal and becomes a vertex in the graph.

2. Second step: We apply the splitting rule on P, creating two branches, one with $P = 0$ and the other with $P = 1$, we continue with the branch with $P = 0$. The literal $\neg P$ becomes a choice literal so there are the vertex $\neg P$ in the graph.

$$\{\{P, Q\}, \{\neg P, \neg Q\}, \{P, \neg Q\}\} \text{ becomes } \{\{Q\}, \{\neg Q\}\} \tag{4.21}$$

3. We apply the UP rule on Q and Q becomes an implied literal, the edges

$$R \to Q \tag{4.22}$$
$$\neg P \to Q \tag{4.23}$$

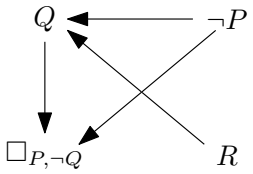
are added to the graph. After the UP rule it remains the empty clause

$$\{\square\}$$

So we add the conflict literal $\square_{P,\neg Q}$ and the edges

$$Q \to \square_{P,\neg Q} \tag{4.24}$$
$$\neg P \to \square_{P,\neg Q} \tag{4.25}$$

the final graph of this branch is:



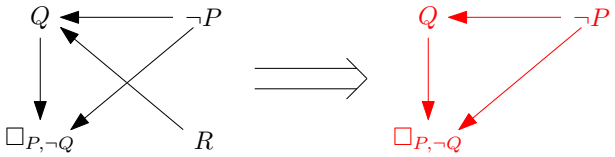
An implication graph cannot have cycle because it keeps track of chronological behavior along the current DPLL search branch β. Unit propagation cannot derive l' whose value was already set beforehand. The implication graph it is used to visualize and trace the logical path that led the algorithm to assign certain truth values to the variables.

There is another tool used to capture "what went wrong" in a failed node (when a branch return "unsatisfiable").

Definition 33 *Let Δ to be a set of clauses and G^{impl} the implication graph relative to a branch of the DPLL algorithm, a* ***conflict graph*** *G^{conf} is a sub-graph of G^{impl} such that:*

- *G^{conf} contains exactly one conflict vertex $\square_C$*
- *if l' is a vertex in ^{confl}, then, all the parents of l' that are vertices $\neg l_i$ with the arc $(\neg l_i, l') \in E(G^{impl})$ are also vertices in G^{conf}.*
- *all the vertices in G^{conf} have a path to $\square_C$,*

We can construct the conflict graph by starting from a conflict vertex and going back through the implication graph until reaching choice literals.



Let $choiceList(G^{conf})$ to be the set of all the choice literals in the conflict graph, given Δ, if we set as true[3] all the literals in $choiceList(G^{conf})$ we will have a false statement:

$$\Delta \models \left(\bigwedge_{l \in choiceList(G^{conf})} l \right) \to \bot \tag{4.26}$$

This can be rewritten as

$$\Delta \models \bigwedge_{l \in choiceList(G^{conf}) \neg l} \tag{4.27}$$

Proposition 3 (Clause Learning) *Let Δ to be a set of clauses and let G^{conf} be a conflict graph at some time point during a run of DPLL on Δ. Let $choiceList(G^{conf})$ to be the set of all the choice literals in the conflict graph, then*

$$\Delta \models \{\neg l \mid l \in choiceList(G^{conf})\} \tag{4.28}$$

The negation of the choice literals in a conflict graph is a valid clause.

So during the execution of the DPLL we can analyze a conflict graph and derive a new clause

$$C = \{\neg l \mid l \in choiceList(G^{conf})\} \tag{4.29}$$

we can add C into Δ and retract the last choice literal l'.

[3] We recall that a literal l can be an atom or a negated atom. If $l = P$ where P is an atom, setting l to true means $l = P = 1$, if $l = \neg P$, setting l to true means $l = 1 = \neg P \implies P = 0$.

Example

Let

$$\Delta = \{\{\neg A, \neg B\}, \{\neg A, B\}\}$$

we apply the DPLL and set in a branch $A = 1$ so

$$\{\{\neg B\}, \{B\}\}$$

we have a conflict and the new clause derived by negating the choice literals is

$$C = \{\neg A\} \tag{4.30}$$

Now we add this to the set of clauses and we have

$$\Delta = \{\{\neg A, \neg B\}, \{\neg A, B\}, \{\neg A\}\}$$

Now the algorithm instantly knows that A must be False and will no longer waste time proving $A = 1$.

After adding the new clause C, the process follows three fundamental logical phases to avoid repeating the same mistake:

1. The algorithm retracts the last choice made, denoted as l' (the truth value assigned to l' is removed and it becomes an unassigned literal again). This is necessary because the combination of previous choices $(l_1, \ldots, l_k)$ together with l' has just been shown to be inconsistent.
2. once C is added and the l' choice is cancelled, the C clause becomes a unit clause (only one unassigned literal remains). In this case, $\neg l'$ is no longer a "free choice" of the algorithm, but becomes a literal implied by Unit Propagation (UP).
3. The algorithm reruns Unit Propagation and continues the analysis. If a new conflict were to occur, the learned clause will be even more ”powerful” because it will only contain the even older choices $(l_1, \ldots, l_k)$, allowing you to move up the decision hierarchy.

We saw how the original DPLL algorithm generates proof exponentially longer than their shortest (general) resolution proof. This is not true with clause learning, that renders DPLL equivalent to full resolution.

CHAPTER

5

PREDICATE LOGIC (FOL)

A logic is a family of formal languages, which has the purpose of *representing* information and *manipulate* knowledge. Each logic is provided with a **syntax** and a **semantics**, the syntax defines a series of symbols and the structure that the formulas must take on to be considered valid, the semantics defines the meaning of these formulas.

In classical logic, each formula, based on the "world" to which it is applied, can be true or false. To define the syntax I must establish:

- What symbols can I use (the alphabet)
- Which finite sequences of symbols can compose a formula

Semantics, on the other hand, establishes the *truth* of formulas in the so-called possible "worlds", i.e the **interpretations**. In first-order logic, also called **FOL**, one establishes

- Syntax
- Semantics
  - Interpretation
  - Variable assignment
  - Model
  - Evaluating a formula
- Satisfaction
- Unsatisfiability
- Validity

5.1 Syntax

In a FOL formula there are:

- Variables: $x, x_1, x_2, x', x'' \dots$
- True ($\top$) or False ($\bot$) symbols
- Operators: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

- Quantifiers: $\forall, \exists$

We can have constant symbols that represents an object

$$BlockA, BlockB, a, b \tag{5.1}$$

predicate symbols that returns true or false taking an object as an input:

$$Block(.), Above(.,.) \tag{5.2}$$

The arity (number of argument) can be ≥ 1. For example:

$$Chase(Cat, Mouse) = \top \tag{5.3}$$

Means that the cat chase the mouse. Then we have the function symbols:

$$WeightOf(.), Succ(.), Sum(.,.) \tag{5.4}$$

these functions returns an object and not a truth value

$$Sum(3, 4) = 7 \tag{5.5}$$

where $3, 4$ and 7 are constant symbols. Constant symbols are just function symbols of arity zero.

Definition 34 (Signature) *A signature Σ in predicate logic is a finite set of constant symbols, predicate symbols, and function symbols.*

Another example of formula is

$$\forall x[Dog(x) \rightarrow \exists y Chase(x, y)] \tag{5.6}$$

Means: For every x such that x is a dog, there exists something (y) such that x chase y, in a few words: Every dog chases something.

Definition 35 (Term) *Let Σ to be a signature:*

- *every variable and every constant symbol is a term.*
- *if $t_1 \ldots, t_n$ are terms and f is a n-ary function symbol, then $f(t_1 \ldots, t_n)$ is a term.*

Definition 36 (Atoms) *Let Σ to be a signature:*

- *$\top$ and $\bot$ are atoms*
- *if $t_1 \ldots, t_n$ are terms and P is a n-ary predicate symbol, then $P(t_1 \ldots, t_n)$ is an atom.*

Definition 37 (Formula) *Let Σ to be a signature:*

- *each atom is a formula.*
- *if φ is a formula, $\neg\varphi$ is a formula*
- *if φ and ψ are formulas then*
  - *$\varphi \wedge \psi$ is a formula (conjunction)*
  - *$\varphi \vee \psi$ is a formula (disjunction)*
  - *$\varphi \rightarrow \psi$ is a formula (implication)*
  - *$\varphi \leftrightarrow \psi$ is a formula (equivalence)*
- *if φ is a formula and x a variable then*
  - *$\forall x\varphi$ is a formula*
  - *$\exists x\varphi$ is a formula*

Generally:

- terms represents object

- predicates represents relations on the universe

Definition 38 (Interpretation) *Let Σ be a signature. A Σ-interpretation is a pair (U, I) where U, the universe, is an arbitrary non-empty set*

$$U = \{o_1, o_2, \dots\}$$

and I is a function, notated as superscript, so that:

1. *I maps constant symbols to elements of U: $c^I \in U$*

$$Lassie^I = o_1$$

2. *I maps n-ary predicate symbols to n-ary relations over U:$P^I \subseteq U^n$*

$$Dog^I = \{o_1, o_3\}$$

3. *I maps n-ary function symbols to n-ary functions over U:$f^I \in [U^n \mapsto U]$*

$$FoodOf^I = \{(o_1 \mapsto o_4), (o_2 \mapsto o_5), \dots\}$$

We will often refer to I as the interpretation, omitting U. Note that U may be infinite.

Definition 39 (Ground Term Interpretation) *The interpretation of a ground term under I is*

$$(f(t_1, \dots, t_n))^I = f^I(t_1^I, \dots, t_n^I)$$

Definition 40 (Ground Atom Satisfaction) *Let Σ be a signature and I a Σ-interpretation. We say that I satisfies a ground atom $P(t_1, \dots, t_n)$, written $I \models P(t_1, \dots, t_n)$, iff $(t_1^I, \dots, t_n^I) \in P^I$. We also call I a model of $P(t_1, \dots, t_n)$.*

$$I \models Dog(Lassie) \text{ because } Lassie^I = o_1 \in Dog^I$$

Example "Integers":

$$U = \{1, 2, 3, \dots\}$$
$$1^I = 1, 2^I = 2, 3^I = 3, \dots$$
$$Even^I = \{2, 3, 4, 6, \dots\}$$
$$Equals^I = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \dots\}$$
$$Succ^I = \{(1 \mapsto 2), (2 \mapsto 3), \dots\}$$

Definition 41 (Variable Assignment) *Let Σ be a signature and (U, I) an interpretation. Let X be the set of all variables. A variable assignment α is a function $\alpha : X \mapsto U$.*

Definition 42 (Term Interpretation) *The interpretation of a term under I and α is:*

1. $x^{I,\alpha} = \alpha(x) \quad [x^{I,\alpha} = o_1]$
2. $c^{I,\alpha} = c^I \quad [Lassie^{I,\alpha} = Lassie^I]$
3. $(f(t_1, \dots, t_n))^{I,\alpha} = f^I(t_1^{I,\alpha}, \dots, t_n^{I,\alpha})$

Definition 43 (Atom Satisfaction) *Let Σ be a signature, I a Σ-interpretation, and α a variable assignment. We say that I and α satisfy an atom $P(t_1, \dots, t_n)$, written $I, \alpha \models P(t_1, \dots, t_n)$, if and only if $(t_1^{I,\alpha}, \dots, t_n^{I,\alpha}) \in P^I$. We also call I and α a model of $P(t_1, \dots, t_n)$.*

From now, we write $\alpha\frac{x}{o}$ to overwrite x with o in α.

Definition 44 (Formula Satisfaction) *Let Σ be a signature, I a Σ-interpretation, and α a variable assignment. We set:*

$$
\begin{aligned}
&I, \alpha \models \top \quad \textit{and} \quad I, \alpha \not\models \bot \\
&I, \alpha \models \neg\varphi \quad \textit{iff} \quad I, \alpha \not\models \varphi \\
&I, \alpha \models \varphi \wedge \psi \quad \textit{iff} \quad I, \alpha \models \varphi \textit{ and } I, \alpha \models \psi \\
&I, \alpha \models \varphi \vee \psi \quad \textit{iff} \quad I, \alpha \models \varphi \textit{ or } I, \alpha \models \psi \\
&I, \alpha \models \varphi \rightarrow \psi \quad \textit{iff} \quad \textit{if } I, \alpha \models \varphi, \textit{ then } I, \alpha \models \psi \\
&I, \alpha \models \varphi \leftrightarrow \psi \quad \textit{iff} \quad I, \alpha \models \varphi \textit{ if and only if } I, \alpha \models \psi \\
&I, \alpha \models \forall x\varphi \quad \textit{iff} \quad \textit{for all } o \in U \textit{ we have } I, \alpha\frac{x}{o} \models \varphi \\
&I, \alpha \models \exists x\varphi \quad \textit{iff} \quad \textit{there exists } o \in U \textit{ so that } I, \alpha\frac{x}{o} \models \varphi
\end{aligned}
$$

If $I, \alpha \models \varphi$, we say that I and α satisfy φ (are a model of φ).

Like the propositional logic, a formula can be

- satisfiable if there exist I, α that satisfy φ.
- unsatisfiable if φ is not satisfiable.
- falsifiable if there exist I, α that do not satisfy φ.
- valid if $I, \alpha \models \varphi$ holds for all I and α. We also call φ a tautology.

Definition 45 *Given two formulas φ, ψ, we say that φ **entails** ψ*

$$\varphi \models \psi$$

if every model of φ is also a model of ψ. If

$$\varphi \models \psi \wedge \psi \models \varphi$$

*then the two formulas are **equivalent**:*

$$\varphi \equiv \psi$$

Let's consider the formula:

$$\varphi = \forall x[R(x, y)] \tag{5.7}$$

the variable y is called *free*, so the formula is not closed and cannot be evaluated in a truth value. We define the **Knowledge Base** as a set KB of closed formulas.
TODO: Normal Forms and all the slide chapter 12

CHAPTER

6

PLANNING

Planning (or automatic planning) is a branch of artificial intelligence that deals with generating a strategy or sequence of actions to allow an intelligent agent (such as a robot or software) to achieve an objective starting from an initial state. It's a general way to define a problem solving procedure.

6.1 STRIPS

STRIPS (acronym for STanford Research Institute Problem Solver) is one of the most important automatic planning languages and systems in the history of artificial intelligence.

Developed in 1971 by Richard Fikes and Nils Nilsson, it was used to control the actions of Shakey, one of the first mobile robots with logical reasoning. In modern terms, STRIPS represents the standard for so-called classic planning.

STRIPS has only boolean variables, the preconditions and the goals are conjunctions of positive atoms, the effects are conjunctions of literals (positive or negated atoms).

Definition 46 (STRIPS Planning Task) *A STRIPS planning task, short planning task, is a 4-tuple* $\Pi = (P, A, I, G)$ *where:*

- P *is a finite set of* ***facts*** *(aka* ***propositions****).*
- A *is a finite set of* ***actions****; each* $a \in A$ *is a triple* $a = (pre_a, add_a, del_a)$ *of subsets of* P *referred to as the action's* ***precondition****,* ***add list****, and* ***delete list*** *respectively; we require that* $add_a \cap del_a = \emptyset$*.*
- $I \subseteq P$ *is the* ***initial state****.*
- $G \subseteq P$ *is the* ***goal****.*

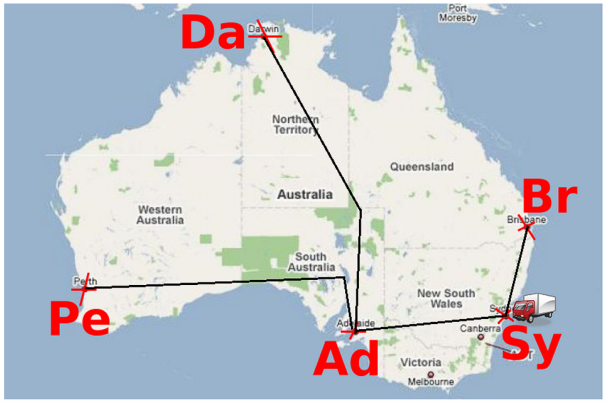
We will often give each action $a \in A$ a **name** (a string), and identify a with that name.

Example: Australian Logistics Planning Task

- **Facts** P: $\{at(x), visited(x) \mid x \in \{Sydney, Adelaide, Brisbane, Perth, Darwin\}\}$.
- **Initial state** I: $\{at(Sydney), visited(Sydney)\}$.
- **Goal** G: $\{at(Sydney)\} \cup \{visited(x) \mid x \in \{Sydney, Adelaide, Brisbane, Perth, Darwin\}\}$.
- **Actions** $a \in A$: $drive(x, y)$ where x, y have a road.
  - **Precondition** pre_a: $\{at(x)\}$.

- **Add list** add_a: $\{at(y), visited(y)\}$.
- **Delete list** del_a: $\{at(x)\}$.

- **Plan:** $\langle drive(Sydney, Brisbane), drive(Brisbane, Sydney), drive(Sydney, Adelaide), drive(Adelaide, Perth), drive$
  $drive(Darwin, Adelaide), drive(Adelaide, Sydney)\rangle$.



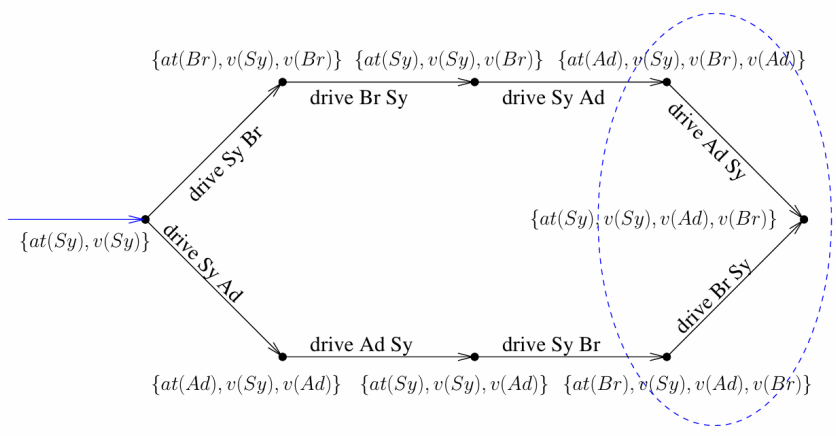
Definition 47 (STRIPS State Space) *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. The state space of Π is $\Theta_\Pi = (S, A, T, I, S^G)$ where:*

- *The states (also **world states**) $S = 2^P$ are the subsets of P.*
- *A is Π's action set.*
- *The transitions are $T = \{s \xrightarrow{a} s' \mid pre_a \subseteq s, s' = (\llbracket s \rrbracket, a)\}$.*
  *If $pre_a \subseteq s$, then a is **applicable** in s and*

$$(\llbracket s \rrbracket, a) := (s \cup add_a) \setminus del_a$$

  If $pre_a \not\subseteq s$, then $(\llbracket s \rrbracket, a)$ is undefined.
- *I is Π's initial state.*
- *The goal states $S^G = \{s \in S \mid G \subseteq s\}$ are those that satisfy Π's goal.*

A **plan** for $s \in S$ is a path from s to $s' \in S^G$, a plan for I is a **solution** and is called *plan for* Π, if such plan exists, Π is **solvable**.



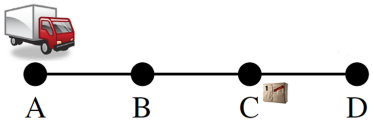
These planning examples draw heavily on the research problems presented in previous chapters, as in this case, it is possible to define a heuristic function that estimates the distance (or cost) between a current state s and the goal state G.

It is important to perform a complexity analysis of the problems that we face to identify special cases that can be solved in polynomial time, relax the input into the special case to obtain a heuristic function.

6.1.1 Only-Adds STRIPS Tasks

Let's consider the following example:

- **Facts** P: $\{truck(x) \mid x \in \{A, B, C, D\}\} \cup$
  $\{pack(x) \mid x \in \{A, B, C, D, T\}\}$.
- **Initial state** I: $\{truck(A), pack(C)\}$.
- **Goal** G: $\{truck(A), pack(D)\}$.
- **Actions** A: (Notated as "precondition $\Rightarrow$ adds, $\neg$ deletes")
  - $drive(x, y)$, where x, y have a road:
    "$truck(x) \Rightarrow truck(y), \neg truck(x)$".
  - $load(x)$: "$truck(x), pack(x) \Rightarrow pack(T), \neg pack(x)$".
  - $unload(x)$: "$truck(x), pack(T) \Rightarrow pack(x), \neg pack(T)$".



It is clear that this STRIPS planning task models a truck that must collect a package and deposit it in a designated place. One simple relaxation is the following:

Only-Adds Relaxation: Drop the preconditions and deletes on the actions.

We want to use this to generate a heuristic function. The general problem is

- Given a STRIPS task $\Pi = (P, A, I, G)$, we want to find an action sequence $\vec{a}$ leading from I to a state that contains G when pretending that preconditions and deletes are empty.

The simplest possible approach is the following: This algorithm is applied for each state s and calculates

Algorithm 23 Solution 1

```
1: a⃗ = ⟨ ⟩
2: while G ≠ ∅ do
3:     select a ∈ A
4:     G = G\add_a #Subtract from the remaining objectives anything that the chosen action adds
5:     a⃗ = a⃗ ∘ ⟨a⟩
6:     A = A\{a}
7: end while
8: return h = |a⃗|
```

the value of h. This heuristic is not admissible, admissibility is only guaranteed if we find a shortest possible $\vec{a}$; else, $\vec{a}$ might be longer than a plan for Π itself. Selecting an arbitrary action each time, $\vec{a}$ may be longer than needed. Another solution might be:

Algorithm 24 Solution 2

```
1: a⃗ = ⟨ ⟩
2: while G ≠ ∅ do
3:     select a ∈ A such that |add_a| is maximal.
4:     G = G\add_a
5:     a⃗ = a⃗ ∘ ⟨a⟩
6:     A = A\{a}
7: end while
8: return h = |a⃗|
```

h is not admissible yet, large add_a doesn't help if the intersection with G is small.

Algorithm 25 Solution 3

1: $\vec{a} = \langle\ \rangle$
2: **while** $G \neq \emptyset$ **do**
3: select $a \in A$ such that $|add_a \cap G|$ is maximal.
4: $G = G \backslash add_a$
5: $\vec{a} = \vec{a} \circ \langle a \rangle$
6: $A = A \backslash \{a\}$
7: **end while**
8: **return** $h = |\vec{a}|$

Also this is not admissible. To have an admissible (and optimal) heuristic in the relaxed world, we should find the shortest possible sequence of actions (optimal $\vec{a}$). But finding this sequence is equivalent to solving the Minimum Cover problem, since Minimum Cover is an NP-complete problem, finding the optimal heuristic even for a relaxed problem is computationally expensive.

We call **PlanEx** the problem of deciding, given a STRIPS planning task Π, whether or not there exists a plan for Π.

Lemma 1 *PlanEx is* PSPACE*-hard.*

Proof Sketch: Given a Turing machine with **space bounded by polynomial** $p(|w|)$, we can in polynomial time (in the size of the machine) generate an equivalent STRIPS planning task. Say the possible symbols in tape cells are $x_1, \ldots, x_m$ and the internal states are $s_1, \ldots, s_n$, accepting state s_{acc}.

- The contents of the tape cells: $in(1, x_1), \ldots, in(p(|w|), x_1), \ldots, in(1, x_m), \ldots, in(p(|w|), x_m)$.
- The position of the R/W head: $at(1), \ldots, at(p(|w|))$.
- The internal state of the machine: $state(s_1), \ldots, state(s_n)$.
- Transitions rules $\mapsto$ STRIPS actions; accepting state $\mapsto$ STRIPS goal $\{state(s_{acc})\}$; initial state obvious.
- This reduction to STRIPS runs in polynomial-time because we need only polynomially many facts.

Lemma 2 *PlanEx is in* PSPACE*.*

It is easy and sufficient to prove that PlanEx is in NPSPACE, since PSPACE = NPSPACE. From the two lemmas, the following theorems follows.

Theorem 10 *PlanEx is* PSPACE*-complete.*

6.1.2 Transition System

The state space of a planning task, like the one described in Definition 47, is generalized in the following definition.

Definition 48 (Transition Systems) *A transition system is a 6-tuple* $\Theta = (S, L, c, T, I, S^G)$ *where:*

- S *is a finite set of states*
- L *is a finite set of transition labels (a generalization of an action)*
- $c : L \mapsto \mathbb{R}^+ \cup \{0\}$ *is the cost function*
- $T \subseteq S \times L \times S$ *is the transition relation*
- $S^G \subseteq S$ *is the set of the goal states.*
- $I \in S$ *is the initial state.*

The size of Θ is $|S|$. If Θ has the transition $(s, l, s') \in T$ we write

$$s \xrightarrow{l} s'$$

or

$$s \to s'$$

if we are not interested in the label l. Θ has unit cost if c is the constant function $c(l) = 1, \forall l$. The classic definitions already addressed follow:

- s' is a successor of s if $s \to s' \in T$
- s' is a predecessor of s if $s' \to s \in T$
- a path from s to s' is a sequence of transitions:

$$s \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \dots \xrightarrow{l_n} s_n = s' \tag{6.1}$$

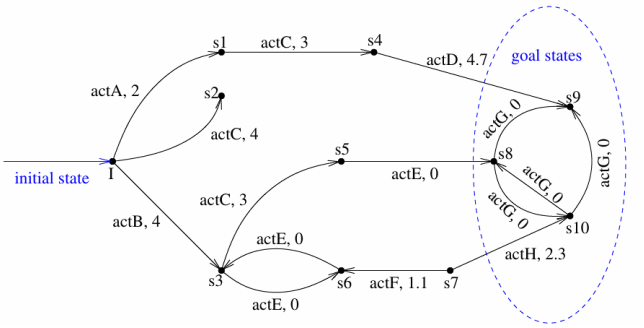
  and the cost of the path is

$$\sum_{i=1}^{n} c(l_i) \tag{6.2}$$

  we say that s is reachable (without specifying the origin) meaning that is reachable from I.

A solution for Θ is a path from I to a state $s \in S^G$. If such path exists, Θ is solvable.

Directed labeled graphs + mark-up for initial state and goal states:



We are interested in solving **huge** transition systems (unfeasible to apply Dijkstra), represented in a compact way as planning tasks.

6.2 FDR

STRIPS and FDR (Finite Domain Representation) are two different formalisms for describing planning tasks, both aimed at defining how an agent should move from an initial state to a goal. FDR is a more modern and often more efficient evolution for automatic solvers (like Fast Downward):

- Finite Domain Variables: Instead of having many Boolean facts (e.g. *truck_in_A*, *truck_in_B*), use a single multivariate variable (e.g. *TruckPosition*) that can take a value in a finite set $\{A, B, C, D\}$
- State: A function that assigns a value to each variable.
- Advantage: Reduces the number of "mutually exclusive" facts the planner must explicitly handle, making the problem structure clearer.

Definition 49 (FDR Planning Task) *A finite-domain representation planning task, short FDR planning task, is a 5-tuple* $\Pi = (V, A, c, I, G)$ *where:*

- *V is a finite set of state variables, each $v \in V$ with a finite domain D_v.*
  We refer to (partial) functions on V, mapping each $v \in V$ into a member of D_v, as (partial) variable assignments.

- *A is a finite set of actions; each $a \in A$ is a pair (pre_a, eff_a) of partial variable assignments referred to as the action's precondition and effects.*
- *$c : A \mapsto \mathbb{R}^+ \cup \{0\}$ is the cost function.*
- *I is a complete variable assignment called the initial state.*
- *G is a partial variable assignment called the goal.*

We say that Π has *unit costs* if, for all $a \in A$, $c(a) = 1$. In the FDR context we call **fact** an assignment

$$(v, d) \text{ or } v = d$$

we mean that the variable v assumes the value d. If p is a partial variable assignment, we write

$$V[p] = \{v \in V \mid p(v) \text{ is defined}\}$$

to refers to all variables where p is defined. For the map example in Australia:

- **Variables** V: $at : \{Sydney, Adelaide, Brisbane, Perth, Darwin\}$; $visited(x) : \{T, F\}$ per ogni città x.
- **Initial state** I: $at = Sydney, visited(Sydney) = T, visited(x) = F$ per $x \neq Sydney$.
- **Goal** G: $at = Sydney, visited(x) = T$ per ogni x.
- **Actions** $a \in A$: $drive(x, y)$ dove x, y hanno una strada.
  - Precondizione pre_a: $at = x$.
  - Effetto eff_a: $at = y, visited(y) = T$.
- **Cost function** c:

$$c(drive(x, y)) = \begin{cases} 1 & \{x, y\} = \{Sydney, Brisbane\} \\ 1.5 & \{x, y\} = \{Sydney, Adelaide\} \\ 3.5 & \{x, y\} = \{Adelaide, Perth\} \\ 4 & \{x, y\} = \{Adelaide, Darwin\} \end{cases}$$

Definition 50 (FDR State Space) *Let $\Pi = (V, A, c, I, G)$ to be an FDR planning task, the **task space** of Π is the labeled transition system $\Theta_\Pi = (S, L, c, T, I, S^G)$ where*

- *the states S are all the possible complete variable assignments.*
- *the labels $L = A$ are the actions of Π, hence, the cost c of Π defined on A is equal of the cost c of Θ_Π defined on L.*
- *The transitions are*

$$T = \{s \xrightarrow{a} s' \mid a \in A[s], s' = s[\![a]\!]\} \tag{6.3}$$

  where
  - *$A[s] = \{a \in A \mid pre_a \subseteq s\}$ are the actions that can be applied in s. $s[\![a]\!]$ represents the resulting state after applying the a action.*

$$s[\![a]\!] = \begin{cases} \text{undefined if } a \notin A[s] \\ eff_a(v) \text{ if } v \in V[eff_a] \\ s(v) \text{ if } v \notin V[eff_a] \end{cases}$$

Although the STRIPS language is still used for historical convenience, modern planning systems (such as Fast Downward) prefer to translate everything internally into FDR for reasons of practical efficiency. The FDR format is superior because it reduces unnecessary states, allows you to better map dependencies via causal graphs, and facilitates the creation of more powerful heuristics, offering a much more natural and compact way of modeling reality than the rigid Boolean variables of STRIPS.

6.2.1 Conversion between FDR and STRIPS

An easy way to convert an FDR planning task in a STRIPS planning task is the following

- for each variable v with domain $\{d_1 \dots d_k\}$, we make k STRIPS facts

$$\begin{aligned} v &= d_1 \\ &\vdots \\ v &= d_k \end{aligned}$$

Definition 51 *Let $\Theta = (S, L, c, T, I, S^G)$ and $\Theta' = (S', L', c', T', I', S'^G)$ to be two transition systems. We say that Θ is **isomorphic** to Θ'*

$$\Theta \sim \Theta'$$

if there exists two bijective functions $\varphi : S \mapsto S'$ and $\psi : L \mapsto L'$ such that

- $\varphi(I) = I'$
- $s \in S^G$ *if and only if* $\varphi(s) \in S'^G$
- $(s, l, t) \in T$ *if and only if* $(\varphi(s), \psi(l), \varphi(t)) \in T'$
- *for all* $l \in L$, $c(l) = c'(\psi(l))$

Isomorphisms typically result from compilations between different formalisms (see later this chapter); we will also sometimes use them as a technical device.