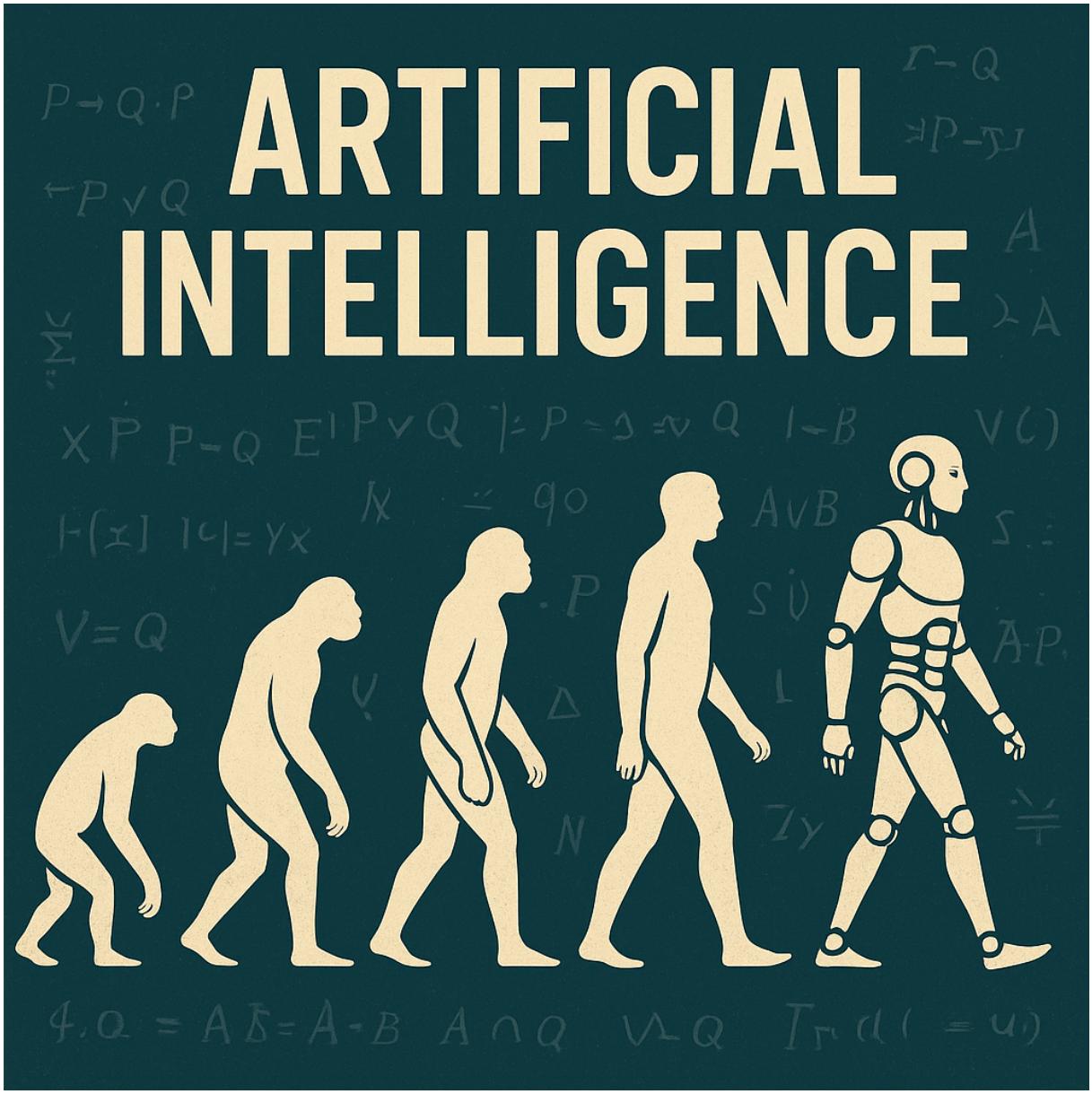
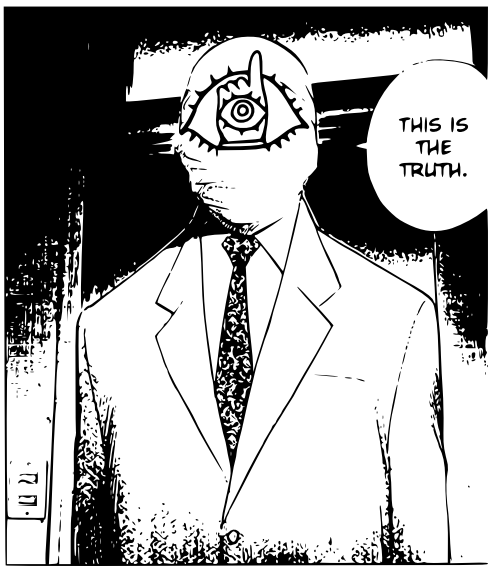
Marco Casu



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*

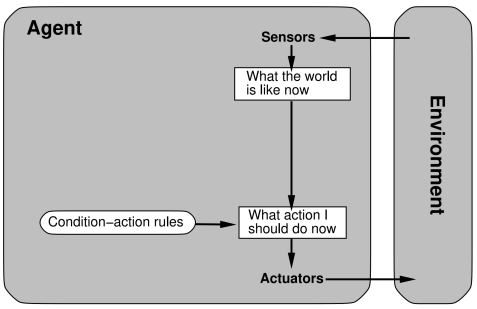


Figure 1.1: Reflex agent diagram

- 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).

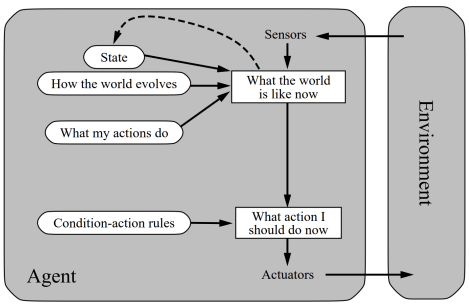


Figure 1.2: Model Based Reflex agent diagram

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.

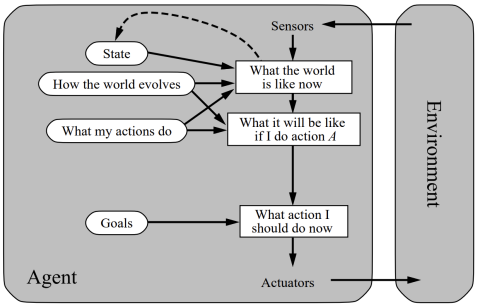


Figure 1.3: Goal Based agent diagram

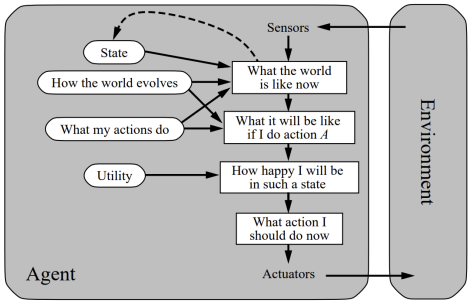


Figure 1.4: Utility Based agent diagram

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.

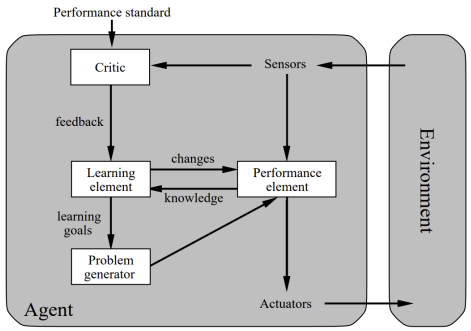


Figure 1.5: Learning agent diagram

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.

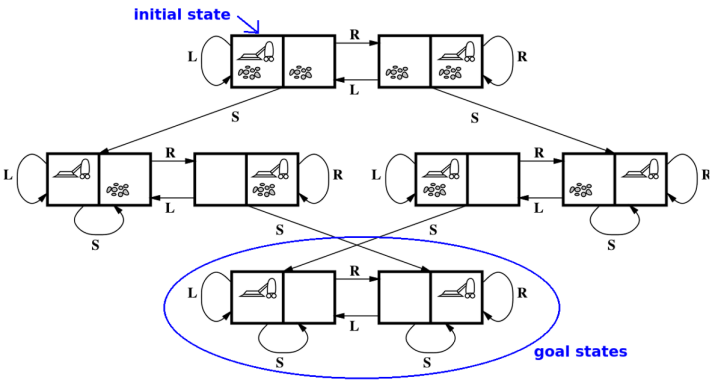


Figure 2.2: The vacuum cleaner state space

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

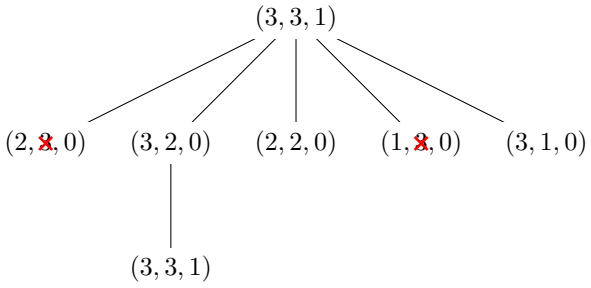
$$S = \{(M, C, B) \ : \ M + C = 6, \ 0 \leq M \leq 3, \ 0 \leq C \leq 3, \ B \in \{0, 1\}\} \tag{2.10}$$

  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.

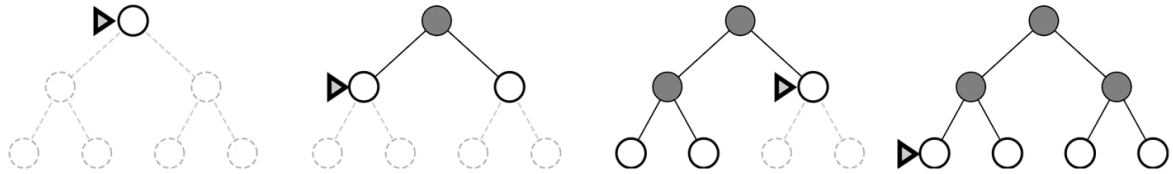
We also have the open list (called frontier) where we can perform the following actions:

- 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

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

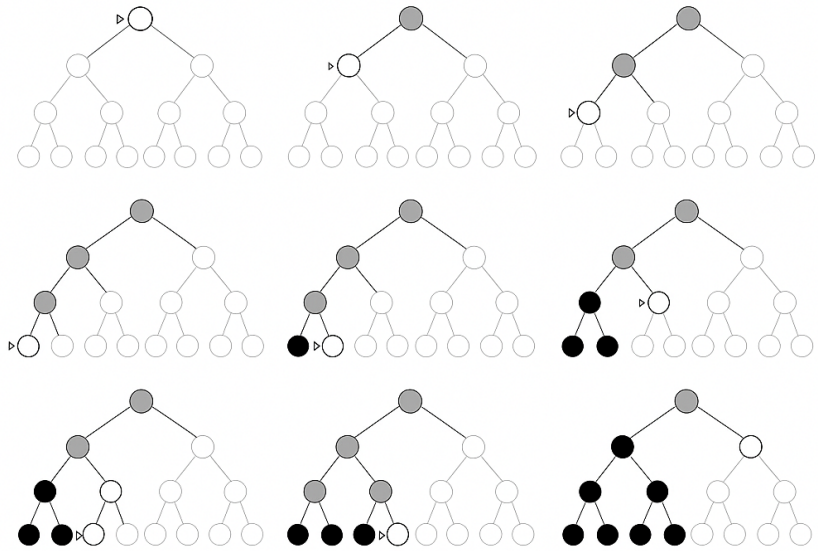
the same for the space complexity since all generated nodes are kept in memory. Let's see an example, assume that $b = 10$, the agent can generate 10^4 nodes per second, and each node has a size of 1 kilobyte, we have the following data:

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

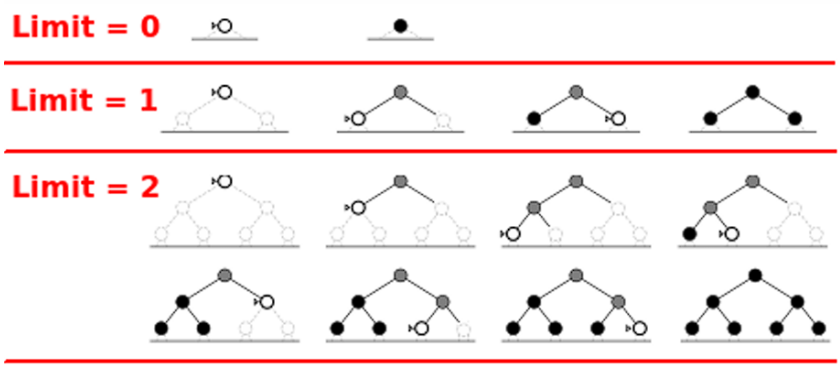
Require: *problem*
  node ←`InitialState()`
  $frontier$ ← priority queue ordered by ascending g
  explored ← empty set of states
  while true **do**
    if Empty?($frontier$) **then**
      return failure
    end if
    n ←Pop($frontier$)
    if `GoalTest`(n.State) **then**
      return Solution(n)
    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 ← `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←Recursive DLS(*problem*,n')
    if *result*==cutoff **then**
      $cutoffOccurred \leftarrow$true
    else
      if result≠ failure **then**
        return *action* ∘ 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

TODO

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 6 *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 7 *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 8 Minimax

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

Algorithm 9 MaxValue

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

Algorithm 10 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

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:

- 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.

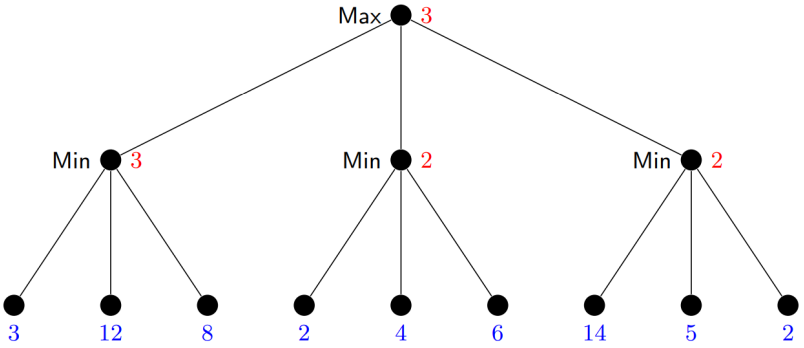


Figure 2.3: Game's tree

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:

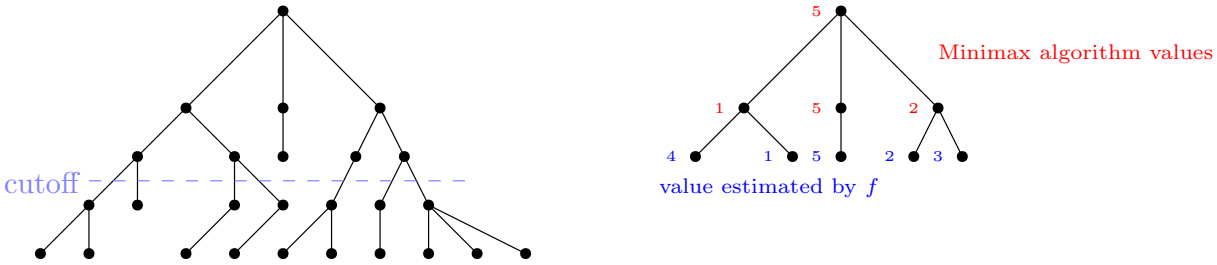
- Complete, if the tree is finite.
- Optimal against an optimal opponent.

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

$$f(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{2.13}$$

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.

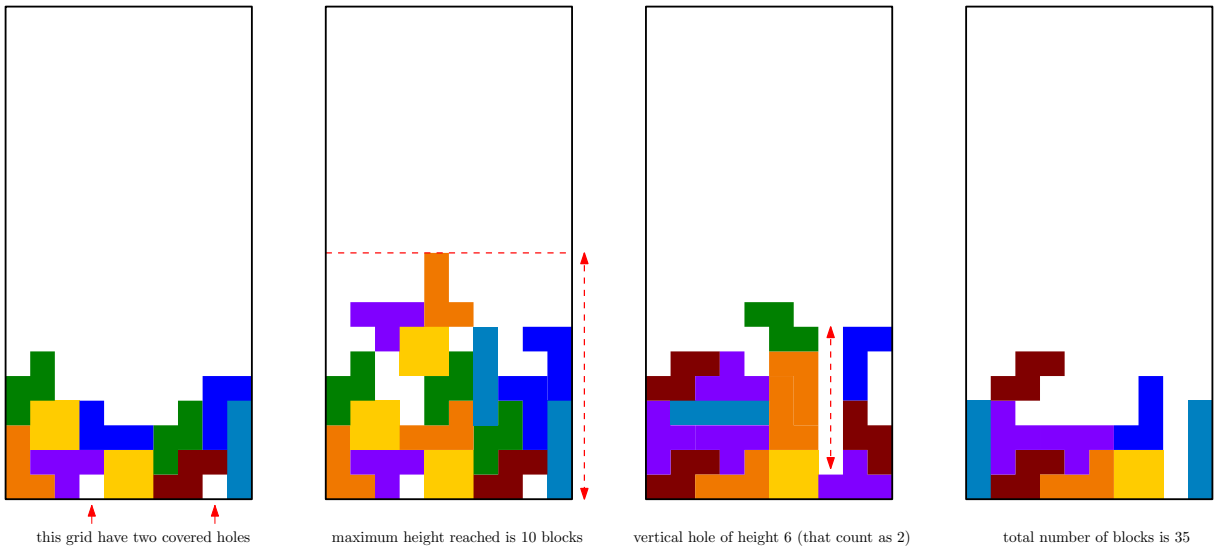


Figure 2.4: Grid's properties

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.

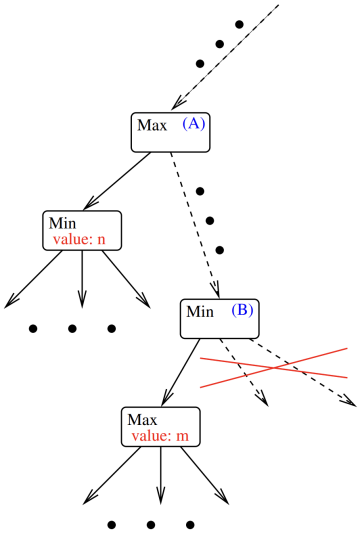


Figure 2.5: Let's say $n > m$

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:

- 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 11 AlphaBetaPruning

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

Algorithm 12 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 13 MinValue

Require: s, α, β
  if TerminalTest(s) **then**
    return $u(s)$
  end if
  $v \leftarrow +\infty$
  for each $a \in$`Actions`(s) **do**
    $v \leftarrow \min(v$,MaxValue(ChildState(s,a)),α,$\beta)$
    $\beta \leftarrow \min(\beta, v)$
    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 8 *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 9 *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.

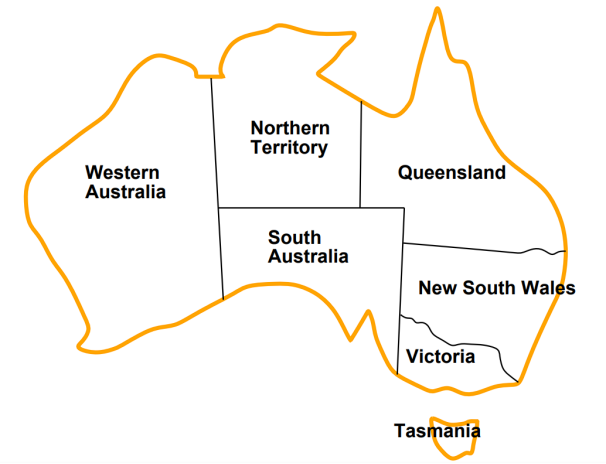


Figure 3.1: Coloring Australia

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 10 *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* $v \in V'$

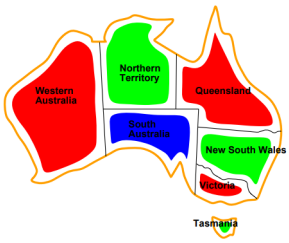
*if $V' = V$ is a **total assignment** (or just assignment). A partial assignment assigns some variables to values from their respective domains. A total assignment is defined on all variables.*

Definition 11 *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 12 *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 1 *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 14 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 14 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 14 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 13 *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 14 *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 15 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 16.

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 16 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 15 *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.

Definition 16 *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**.*