Fundamentals of Artificial Intelligence and Knowledge Representation

Module 1

Matteo Donati

August 4, 2021

Contents

Chapter 1

Introduction

Intelligence has been defined in many ways, including: the capacity for logic, understanding, self-awareness, learning, emotional knowledge, reasoning, planning, creativity, critical thinking and problem solving. More generally, it can be described as the ability to perceive or infer information, and to retain it as knowledge to be applied towards adaptive behaviours within an environment or context.

1.1 Artificial Intelligence

There exist different definitions of the concept of artificial intelligence. In general, is the study of how to build computers which pass the Turing test.

1.1.1 The Turing Test (The Imitation Game)

Alan Turing developed a test in which three entities are involved: a human, a computer and an interrogator. These three entities sit in different rooms and the interrogator tries to understand whether he is talking with the computer or not. In order to solve the Turing test and mislead the interrogator, the computer should have the following abilities:

- Natural language processing.
- Knowledge representation.
- Automatic reasoning.
- Machine learning.

1.1.2 Systems Classification

Artificial intelligence systems can be categorized as follows:

- **Weak artificial intelligence**, which includes systems that acts as if they were intelligent.
- **Strong artificial intelligence**, which includes real intelligent systems.

- **General artificial intelligence**, which includes systems able to cope with any generalised task.
- **Narrow artificial intelligence**, which includes systems able to handle just one particular task.

1.1.3 Building Approaches

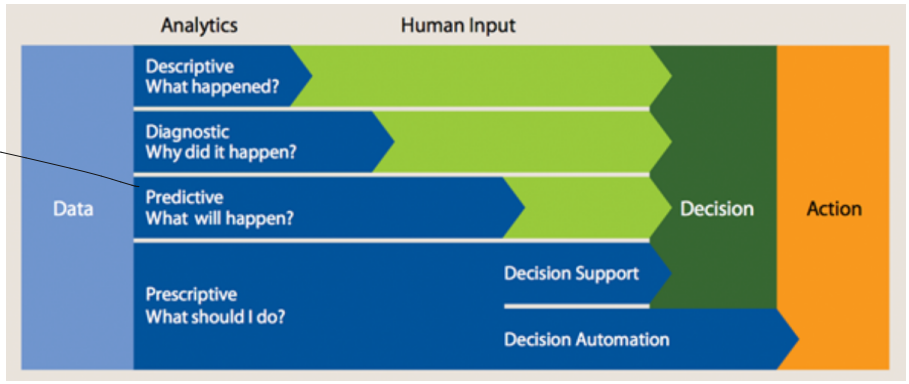
In order to build artificial intelligence systems, one can choose between of the following approaches:

- **Symbolic approach** (top-down approach): — computationally expensive
  - Symbolic representation of knowledge.
  - Logics, ontologies, rule based systems, declarative architecture.
  - Human understandable models.
- **Connessionist approach** (bottom-up approach):
  - Neural networks. The knowledge is not symbolic and it is encoded into the connections between neurons.
  - Concepts are learned by examples.
  - Not understandable by humans.

1.1.4 Applications

Below are some of the most important types of artificial intelligence applications:

- **Knowledge based systems**, which are systems that support humans in their decision making process. There exist various degree of decision support:



1. Simulators
2. Machine Learning.

- **Games**. Some examples are Deep Blue, Alpha Go, Jeopardy.
- **Computer Vision**.
- **Robotics**.
- Natural Language Processing.

1.1.5 Trustworthy Artificial Intelligence

Artificial intelligence should be beneficial with applications related to health, climate change, energy and other important aspects of today's world. In order to do so, the EU has delivered guidelines for trustworthy artificial intelligence. Trustworthy artificial intelligence should have the following properties:

- It should respect fundamental rights.
- It should be technically robust and reliable.

1.2 Machine Learning

Field of study that gives a computer the ability to improve its behaviour, without being explicitly programmed. The computer will learn from new data and examples. There exist three ways of learning:

- **Supervised learning**, when there exist a supervisor able to label the given training dataset.
- **Unsupervised learning**, when there exist no supervisor but through observation of the data is possible to infer informations. This category involves clustering problems.
- **Reinforcement learning**, which is mainly based on learning an optimal behaviour from past experiences.

1.2.1 Tasks

Below are some examples of machine learning tasks:

- **Classification**, which is the task of approximating a mapping function from input variables to discrete/categorical output variables. Classification can be binary or multi-class.
- **Regression**, which is the task of approximating a mapping function from input variables to a continuous output variable.
- **Clustering**, which is the problem of organizing objects into groups whose members are similar according to some specific criteria.

1.3 Reasoning and Logic

Regarding logic, there exist different types of reasoning:

- **Deductive reasoning**, in which some deductions are inferred. This type of reasoning does not allow one to learn a new knowledge (e.g. all men are mortal and Socrates is a man, then Socrates is mortal).

- **Inductive reasoning**, which extracts general rules from observations (e.g. several birds can fly then all birds can fly). This type of reasoning produces a new knowledge at the expense of correctness.
- **Hypothetical or abductive reasoning**, which dates back to the causes through observation of the effects at the expense of correctness (e.g. from the death of Socrates and knowing that all men are mortal, one hypothesizes that Socrates is a man).
- **Reasoning by analogy**, which uses the principle of similarity (e.g. k-nearest neighbours and support vector machine).
- **Constraint reasoning and optimization**, which uses constraints, probability and statistics.

1.4 Automated Planning

Automated planning is an important problem solving activity which consists in synthesizing a sequence of actions performed by an agent that leads from an initial state of the world to a given target state (goal). In particular, given an initial state, a set of possible actions and a final state, one has to find a plan, i.e. a partially or totally ordered set of actions needed to achieve the goal from the initial state.

Chapter 2

Searching

Many artificial intelligence problems can be solved by exploring the so called *solution space*. In particular, this domain contains all possible sequences of actions that might be applied by the specific agent. The search process can be thought as building a search tree, whose nodes are states and whose branches are actions, and traversing the tree. Once the solution is found, this is returned in the form of a sequence of actions. A possible implementation of a tree-search function is the following:

function TREE-SEARCH(*problem*, *strategy*)**returns** solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node constains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree

Nodes are expanded according to a successor function

The basic idea is a simulated exploration obtained by expanding states that have already been explored. The algorithm starts from the given initial state and expands nodes according to a given search strategy. In particular, there exist two possible types of strategy:

General purpose strategies

- **Uninformed strategies**, which do not use any domain knowledge.
- **Informed strategies**, which use domain knowledge. They apply rules following a heuristics.
  (not a perfect heuristics, just a sort of state evaluation)

Every strategy can be evaluated according to the following criteria:

- **Completeness**. A strategy is complete if, whenever a solution exists, it is able to find it.
- **Time complexity**.
- **Space complexity**.
- **Optimality**. A strategy is optimal if, whenever multiple solutions exist, it can find the best solution.

- **Optimality**. A strategy is optimal if, whenever multiple solutions exist, it can find the best solution.

2.1 Uninformed Strategies

Strategies which do not use any domain knowledge.

2.1.1 Breadth-First Search

This strategy always expands tree nodes at the lowest depth. In particular, the fringe is a queue (i.e. a FIFO data structure).

- Ensures completeness.
- In the worst case scenario, if the depth to the solution is d and the branching factor is b (the branching factor of a tree represents the average number of child-nodes for each node of the tree), then the maximum number of nodes expanded is b^d. Therefore, time and space complexity are $\mathcal{O}(b^d)$
- Ensures optimality only if $cost = 1$ per step.

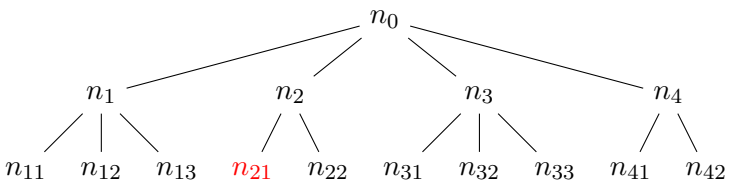
Moreover, if each node is labelled with a different cost, $g(n)$, then breadth-first search is called **uniform-cost search**. In particular, the queue is ordered based on the cost of each node. The optimal solution is obtained whenever the goal state is in front of the queue (i.e. it is the next node to be expanded).

2.1.2 Depth-First Search

This strategy always expands tree nodes at the highest depth. In particular, the fringe is a stack (i.e. a LIFO data structure).

- Ensures completeness in finite spaces, i.e. when there exist no loops in the state space.
- In the worst case scenario ($m \gg d$), if the depth to the solution is d, the branching factor is b and the maximum depth of the state space is m (which may be infinite), then the time complexity is $\mathcal{O}(b^m)$ and the space complexity is $\mathcal{O}(bm)$.
- Does not ensure optimality.

For example, considering the following tree, the expanded nodes are the following:



Breadth-first search:
1. n_0.
2. n_0, n_1, n_2, n_3, n_4.
3. $n_2, n_3, n_4, n_{11}, n_{12}, n_{13}$.
4. $n_3, n_4, n_{11}, n_{12}, n_{13}, n_{21}, n_{22}$.
5. $n_4, n_{11}, n_{12}, n_{13}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}$.
6. $n_{11}, n_{12}, n_{13}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}, n_{41}, n_{42}$.
7. $n_{12}, n_{13}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}, n_{41}, n_{42}$.
8. $n_{13}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}, n_{41}, n_{42}$.
9. $\mathbf{n_{21}}, n_{22}, n_{31}, n_{32}, n_{33}, n_{41}, n_{42}$. Success.

Depth-first search:
1. n_0.
2. n_0, n_1, n_2, n_3, n_4.
3. $n_{11}, n_{12}, n_{13}, n_2, n_3, n_4$.
4. $n_{12}, n_{13}, n_2, n_3, n_4$.
5. n_{13}, n_2, n_3, n_4.
6. n_2, n_3, n_4.
7. $\mathbf{n_{21}}, n_{22}, n_3, n_4$. Success.

2.1.3 Depth-Limited Search

This strategy is a variant of depth-first search. In particular, it includes a maximum depth parameter, l. Whenever the maximum depth is reached, the algorithm explores alternative paths. The maximum depth parameter avoids infinite cycles but if $d_{goal} > l$ then completeness is not ensured.

2.1.4 Iterative Deepening

This strategy iteratively applies the depth-limited search algorithm:

1. The maximum depth parameter is set to one.
2. The depth-limited search algorithm is applied.
3. If a solution has not been found, the maximum depth parameter is increased by one and point 2. is repeated.

In particular:

- Ensures completeness.
- In the worst case scenario, if the depth to the solution is d, the branching factor is b, then the time complexity is $\mathcal{O}(b^d)$ and the space complexity is $\mathcal{O}(bd)$.
- Ensures optimality only if $cost = 1$ per step.

2.2 Informed Strategies

Strategies which use domain knowledge.

2.2.1 Best-First Search

This strategy uses an evaluation function (heuristic), $h(n)$, that takes as input a specific node and compute a number which represents the desirability related to the node expansion (i.e. what might be the cost to reach the goal). In particular, best-first search expands the most promising node, according to a specific evaluation function, and applies a search algorithm.

- Does not ensure completeness.
- In the worst case scenario, if the depth to the solution is d, the branching factor is b, then the time complexity and space complexity are $\mathcal{O}(b^d)$.
- Does not ensure optimality.

However, with good heuristics the temporal and spatial complexity can be reduced substantially.

2.2.2 A* Algorithm

This strategy is based on the following evaluation function:

$$f(n) = g(n) + h'(n) \tag{2.1}$$

In particular, $g(n)$ is the cost of the path from the root node to n and $h'(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. In particular, this strategy selects the path that minimizes $f(n)$.

- Given the true distance between the current node and the solution, $h(n)$, if $h'(n) \leq h(n)$ the heuristic function, $h'(n)$, is said to be **feasible** and, therefore, the A* algorithm for trees is optimal.
- The A* algorithm can also be expanded to deal with graphs. In particular, in order to avoid further examination of already examined nodes, it is necessary to define two lists: one for open nodes and one for closed nodes. The algorithm follows the following steps:

  1. Let L_a be the open list of initial nodes of the problem.
  2. Let n be the node for which $g(n) + h'(n)$ is minimal. If the list is empty, then fail;
  3. If n is the goal then stop and return the found path.
  4. Otherwise, remove n from L_a and add it to the list of closed nodes, L_c, then add to L_a every child of n, labelling them with the cost from the starting node to n.
  5. If a child node is already in L_a, do not add it, but update its label (cost).
  6. If a child node is already in L_c, do not add it, but if its cost is better than the previous one, update its cost and the one of its children and descendant.
  7. Return to step 2.

  In order for the A* graph algorithm to be optimal, $h'(n)$ needs to be **consistent**. A given heuristic h is said to be consistent if, for every node n, any successor, n', of n generated by an action a, the following properties are valid:

  - $h(n) = 0$ if the corresponding state is the goal.
  - $h(n) \leq c(n, a, n') + h(n')$ if the corresponding state is not the goal.

2.3 Local Search

One is not guaranteed to obtain an optimal solution as in the case of constructive search.

The algorithms seen so far are constructive methods (tree search is a constructive approach) in which one generates a solution by adding components, in a particular order, to a starting state. On the other hand, local search algorithms start from an initial solution and iteratively try to improve it through local moves.
An example of this, is represented by the "N queens problem", in which one has to put n queens on a chessboard so that they do not threat each other. The first thing one has to do is to place all the n queens into the chessboard. By doing so, one already starts with a solution and tries to improve it.
Local search algorithms are based on the notion of *neighbourhood*:

- A **neighbourhood** is a function from state space to the power set of the same state space:

$$\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}} \tag{2.2}$$

  This function assigns to every $s \in \mathcal{S}$ a set of neighbours $\mathcal{N}(s) \subseteq \mathcal{S}$. $\mathcal{N}(s)$ is called neighbourhood of s.
  └ possible states from s

2.3.1 Iterative Improvement - Hill Climbing

Iterative improvement, or hill climbing, is a basic local search algorithm in which a move is only performed if the new solution is better than the current solution. The algorithm tries to find a minimum of some objective function and stops as soon as it finds a local minimum, without the certainty of it being the global minimum of the given function:

$s \leftarrow\!\shortmid$ GenerateInitialSolution()
while improvements are possible **do**
  $s \leftarrow\!\shortmid$ BestOf$(s, \mathcal{N}(s))$
end while

This algorithm is not effective in exploring the search space (i.e. it stops at the first local optimum) and does not remember the states already reached.

2.3.2 Simulated Annealing (meta-heuristic)

Simulated annealing is a local search algorithm which, in order to avoid local optimum, allows moves resulting in solutions of worse quality than the current one. In particular, the probability of doing such moves is decreased during the search process:

$s \leftarrow\!\shortmid$ GenerateInitialSolution()
$T \leftarrow\!\shortmid T_0$
while termination conditions not met **do**
  $s' \leftarrow\!\shortmid$ PickAtRandom($\mathcal{N}(s)$)
  if $f(s') < f(s)$ **then**
    $s \leftarrow\!\shortmid s'$
  else
    accept s' as a new solution with probability $p(T, s, s') = \frac{\exp\left(-|f(s') - f(s)|\right)}{T}$
  end if
Update(T) — Reducing the probability of choosing a worsening move
end while

2.3.3 Tabu Search (meta-heuristic)

Tabu search is a local search algorithm which, in order to avoid local optimum, explicitly exploits the search history to dynamically change the neighbourhood to explore. It uses a tabu list to keep track of recent visited solutions or moves and forbids them. This allows to escape from local optimum:

$s \leftarrow\!\shortmid$ GenerateInitialSolution()
$TabuList \leftarrow\!\shortmid \emptyset$ — e.g. max number of iterations
while termination conditions not met **do**
  $s \leftarrow\!\shortmid$ BestOf($\mathcal{N}(s) \backslash TabuList$)
  Update($TabuList$)
end while

2.3.4 Iterated Local Search (meta-heuristic)

Iterated local search is a local search algorithm which, in order to avoid local optimum, uses two different types of steps:

- **Subsidiary local search steps** for reaching local optima as efficiently as possible.
- **Perturbation steps** for effectively escaping from local optima.

At each iteration, an acceptance criterion is used to control the diversification and intensification behaviours. | ≡ exploring and exploiting

$s_0 \leftarrow\!\shortmid$ GenerateInitialSolution()
$s_* \leftarrow\!\shortmid$ LocalSearch(s_0) — applying hill-climbing, s^* is a local minimme
while termination conditions not met **do**
  $s^{'} \leftarrow\!\shortmid$ Perturbation($s_*, history$)
  $s_{*'} \leftarrow\!\shortmid$ LocalSearch($s^{'}$)
  $s_* \leftarrow\!\shortmid$ ApplyAcceptanceCriterion($s, s_{*'}, history$)
end while

2.3.5 Genetic Algorithms (population-based meta-heuristic)

Genetic algorithms are inspired by evolution. In particular, the key concepts are the following:

- The fittest individuals have high chance of having a numerous offspring.
- The children are similar, but not equal, to the parents.
- The traits characterizing the fittest individuals spread across the population, generation by generation.

In particular:

- A population is the set of individuals (solutions).
- Individuals are also called *genotypes*.
- Chromosomes are made of units called *genes*.

- The domain of values of a gene is composed of *alleles* (e.g. binary variable $\leftrightarrow$ gene with two alleles)

There exist multiple genetic operators:

- **Crossover**, which is a cross-combination of two chromosomes:
- **Mutation**, in which each gene has probability p_M of being modified:
- **Proportional selection**, in which the probability for an individual to be chosen is proportional to its fitness.
- **Generational replecement**, in which the new generation replaces entirely the old one.

```
InitializePopulation()
EvaluatePopulation()
while termination condition not met do
  while new population not completed do
    select two parents for mating
    apply crossover
    apply mutation to each new individual
  end while
  Population ↤ NewPopulation
  EvaluatePopulation()
end while
```

In particular:

- Crossover combines good parts from good solutions, but it might achieve the opposite effect.
- Mutation introduces diversity.
- Selection drives the population toward high fitness.

Chapter 3

Swarm Intelligence

- Swarm intelligence (SI) is an artificial intelligence technique based on the study of collective behaviour in decentralized self-organized systems.
- Swarm intelligence systems are typically made up of a population of simple agents which interact locally with one another and with the environment.
- Although there is normally no centralized control structure dictating how individual agents should behave, local interactions between such agents often lead to the emergence of global behaviour.
- Examples of systems like these can be found in nature, including ant colonies, flocks (birds), animal herding, moulds (bacteria), fish schooling and others.

Some features of swarm intelligence systems are the following:

- These systems are usually composed of simple individuals with limited capacities.
- The individuals are not aware of the environment in which they live in.
- There exist local communication patters between individuals. An example of this is stigmetry, a form of indirect communication, in which an individual modifies the environment and the others react to this modification.
- There exist distributed computation, i.e. there exist no centralized coordination of individuals activities.
- These systems are robust and adaptive.

3.1 Algorithms

Based on the swarm intelligence concept, many algorithm have been invented.

3.1.1 Ant Colony Optimization

Ant colony optimization (ACO) is based on the behaviour of ants. In particular:

- Ants deposit pheromone trails while walking from the nest to the food and vice verse (example of stigmetry).
- Ants tend to choose the paths marked with higher pheromone concentration.
- Cooperative interaction leads to the emergent behaviour to find the shortest path to destination.

In order to develop an algorithm able to describe ants behaviour, a probabilistic and parametrized model, the **pheromone model**, is used to model pheromone trails. In particular:

- The algorithm proceeds by taking stochastic steps on a fully connected graph called **construction graph**, $G(C, L)$, in which C is the set of vertices and L is the set of arcs.
- Arcs and vertices of the graph have the following associated informations:
  - A pheromone value, τ, which abstracts the natural pheromone trails.
  - A heuristic value, η, which represents the prior background knowledge on the problem.

An high-level proposal of this algorithm is the following:

InitializePheromoneValues() — as any meta-heuristics
while termination conditions not met **do**
  for all $a \in ants$ **do**
    $s_a \leftarrow\!\!\shortmid$ ConstructSolution(τ, η) — every ant builds a solution
  end for
  ApplyOnlineDelayedPheromoneUpdate() — Pheromone updates can also be applied at each step taken by each ant:
$\Delta\tau_{ij}^{k} = \begin{cases} 1/L_k & \text{if ant } k \text{ follows the } i\text{–}j \text{ path} \\ 0 & \text{otherwise} \end{cases}$
L_k: length of the path followed by the k-th ant so far (not of the entire path)
end while

In particular:

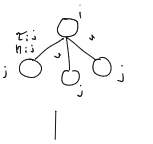
- Memory is used to remind past paths.
- Starting from node i, the next node to visit, j, is chosen probabilistically. In particular, given a pheromone value τ_{ij} and a heuristic value η_{ij}:

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{k \text{ feasible}} \tau_{ik}^{\alpha}\eta_{ik}^{\beta}} & \text{if } j \text{ is consistent} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

normalizing factor

where, α and β are weights.

$\eta_{ij} = \frac{1}{d_{ij}}$ — distance between node i and node j



ants can go from i to any node j

- The pheromone values are updated as soon as each ant has found the path to the solution:

$$\tau_{ij} = (i-\rho)\tau_{ij} + \sum_{k=1}^{|ants|} \Delta\tau_{ij}^{k}, \quad \text{with} \quad \Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{L_k} & \text{if ant } k \text{ uses arc } (i,j) \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

  where ρ is the evaporation coefficient and L_k is the length of the path followed by ant k.

3.1.2 Artificial Bee Colony

Artificial bee colony is based on the behaviour of bees and it is used for optimizing numerical problems. In particular, there exist three different types of bees:

- Employed bees, which are associated with a specific nectar source.
- Onlookers bees, which by observing the employed bees chose a nectar source.
- Scout bees, which discover new food sources.

Initially, all food source positions are discovered by scout bees. Thereafter, the nectar of food sources are exploited by employed bees and onlooker bees, and this continual exploitation will ultimately cause them to become exhausted. Then, the employed bee which was exploiting the exhausted food source becomes a scout bee in search of further food sources once again. In artificial bee colony, the position of a food source represents a possible solution to the problem and the nectar amount of a food source corresponds to the quality (fitness) of the associated solution. The number of employed bees is equal to the number of food sources (solutions) since each employed bee is associated with one and only one food source.
The general algorithm is the following:

```
InitializationPhase()
repeat
  EmployedBeesPhase()
  OnlookerBeesPhase()
  ScoutBeesPhase()
  Solution = BestSolutionSoFar()
until (Cycle = MaxCycleNum||MaxCPUTime)
```

In particular:

- In the initialization phase all the vectors of the population of food sources, x_{m_i} where ~~$m \in \{1, \ldots, n\}$~~, are initialized by scout bees. Each variable is subject to a lower and upper bound, l and u, and it is initialized to:

$$x_{mi} = l_i + rand(0,1)(u_i - l_i) \tag{3.3}$$

- In the employed bees phase, employed bees search for new food sources (v_m) having more nectar within the neighbourhood of the food source (x_m) in their memory. They find a neighbour food source and then evaluate its profitability (fitness). In particular, the fitness value function of the solution, fit_m, can be calculated as follows:

$$fit_m(x_m) = \begin{cases} \frac{1}{1+f_m(x_m)} & \text{if } f_m(x_m) \geq 0 \\ 1 + |f_m(x_m)| & \text{if } f_m(x_m) < 0 \end{cases} \tag{3.4}$$

  where $f_m(x_m)$ is the objective function value of a given solution.

- In the onlooker bees phase, an onlooker bee chooses a food source depending on the probability value associated with that food source:

$$p_m = \frac{fit_m(x_m)}{\sum_{i=1}^{SN} fit_m(x_i)} \tag{3.5}$$

  where SN is the population size.

  After a food source x_m for an onlooker bee is probabilistically chosen, a neighbourhood source v_m is determined and its value is computed. A greedy selection is applied between v_m and x_m.

- In the scout bees phase, employed bees whose solutions cannot be improved through a predetermined number of trials, specified by the user of the algorithm, become scouts and their solutions are abandoned. Then, the converted scouts start to search for new solutions, randomly.

3.1.3 Particle Swarm Optimization

Particle swarm optimization (PSO) is based on the observation of bird flocks or fish schools. Particle swarm optimization is used to solve optimization problems using the following analogy:

- Individuals move and sample the solution N-dimensional space. The search strategy can be seen as a balance between exploration and exploitation.
- There exist social interaction in which ~~and~~ individual agent can take advantage from other searches moving toward promising regions.

The algorithm is based on the following notions:

- A given problem is optimized by setting up a population (swarm) of candidate solutions (particles). The population is composed of S particles. Each particle has a position, $x_i \in \mathbb{R}^n$, in the search space and a ~~speed~~ velocity, $v_i \in \mathbb{R}^n$.
- The particles are moved in the search space through simple mathematical formulas.
- The movement of particles is guided by the best position so far in the search space and it is updated whenever a better solution is discovered.

- $f : \mathbb{R}^n \to \mathbb{R}$ is the specific cost function to minimize. The goal is to find the global minimum of the given cost function.

The steps of the algorithm are the following:

for each particle $i = 1, \ldots, S$ **do**
Initialize the particle's position with a uniformly distributed random vector, $x_i \sim U(b_l, b_u)$.
Initialize the particle's best known position to its initial position: $p_i \leftarrow\!\shortmid x_i$
If $f(p_i) < f(g)$ update thw swarm's best known position: $g \leftarrow\!\shortmid p_i$
Initialize the particle's velocity: $v_i \sim U(-|b_u - b_l|, |b_u - b_l|)$

while termination criteria are not met do
for each particle $i = 1, \ldots, S$ **do**
Pick random numbers: $r_p, r_g \sim U(0, 1)$ non-determinism
Update velocities: $v_i \leftarrow\!\shortmid \omega v_i + \varphi_p r_p(p_i - x_i) + \varphi_g r_g(g - x_i)$
Update positions: $x_i \leftarrow\!\shortmid x_i + v_i$
If $f(x_i) < f(p_i)$:
Update the particle's best known position: $p_i \leftarrow\!\shortmid x_i$
If $f(p_i) < f(g)$ update thw swarm's best known position: $g \leftarrow\!\shortmid p_i$
return g

In particular:

- p_i is the best solution found so far by particle i.
- g is the best solution found so far by the entire swarm.
- b_l and b_u are the lower and upper boundaries of the search space.
- $\omega, \varphi_p, \varphi_g$ are chosen parameters.

Parameters Tuning: one can find the best set of parameters by applying search strategies in the parameters space.

Chapter 4

Games

In order to deal with games it is necessary to introduce algorithms with are able to handle situations in which two players face each other. In this chapter, only games with the following properties are considered:

- Two-players games (MIN and MAX) in which players take turns. These two players have complementary objective functions.
- Games with perfect knowledge in which players have the same information.

Under such circumstances, the development of a match can be interpreted as a tree in which the root is the starting position and the leaves are the final positions. In this setting, the problem of selecting the best possible move is reducible to a search problem.

4.1 Minimax Algorithm

The minimax algorithm is designed to determine the optimal strategy for MAX and to suggest the first best move to be performed. The MIN player is assumed to play at his best. In particular:

- In order to avoid the exponential explosion of the state space (for a tree with branching factor b and depth d, the number of nodes is b^d), the search process is applied up to a certain depth, assessing the configuration of a non-terminal node.
- The quality of a certain node is estimated using a specific evaluation function $e : \mathcal{N} \to \mathbb{R}_{[-1,1]}$:

$$e(n) = \begin{cases} -1 & \text{if } MIN \text{ wins} \\ 1 & \text{if } MAX \text{ wins} \\ 0 & \text{if } MIN \text{ and } MAX \text{ have the same probability of winning} \end{cases} \tag{4.1}$$

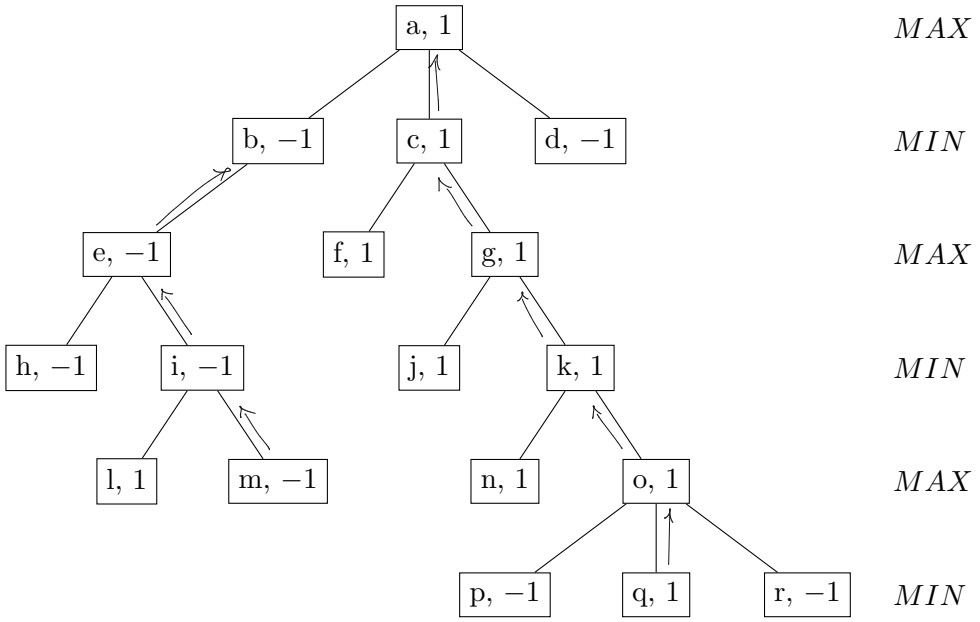
Given a node n an evaluation function e and a list of open nodes (i.e. not yet expanded nodes) L, the algorithm proceeds as follows:

1. Put n in L.
2. Let x be the first node in L:
   (a) If $x = n$ and there is a value assigned to it, then return the value and end the algorithm.
   (b) Otherwise, if x has an assigned value V_x, p is the father of x and V_p is the provisional value assigned to p:
   - If p is a MIN node, then $V_p = \min(V_p, V_x)$.
   - If p is a MAX node, then $V_p = \max(V_p, V_x)$.
   - Remove x from L and return to step 2.

   (c) Otherwise, if there is no value assigned to it and x is a leaf node or the tree expansion has been stopped, then assign to x the value given by $e(x)$. Put x in L and return to step 2.
   (d) Otherwise, if there is no value assigned to it and x is not a leaf, then:
   - If x is a MIN, then $V_x = \infty$.
   - If x is a MAX, then $V_x = -\infty$.

   Add the children of x to L and return to step 2.

A simple example of this algorithm is the following:

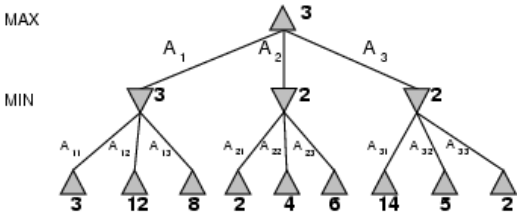


4.2 Alpha-Beta Algorithm

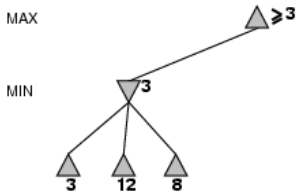
The alpha-beta algorithm is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. The algorithm maintains two values, α and β, which respectively represent the minimum score that the MAX player is assured of and the maximum score that the MIN player is assured of. In particular:

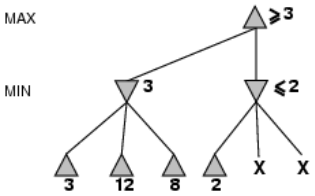
- Initially, $\alpha = -\infty$ and $\beta = \infty$, i.e. both players start with their worst possible score.
- If at a particular node n the maximum score that the MIN player is assured of (β) becomes less than the minimum score that the MAX player is assured of (α), i.e. $\beta < \alpha$, the maximizing player does not need to consider further descendants of this node, as they will never be reached in the actual play.
- In the best case, i.e. when the best nodes are evaluated first, the number of nodes becomes $\sim b^{\frac{d}{2}}$. In the average case, with random distribution of the node values, the number of nodes becomes $\sim b^{\frac{3}{4}d}$.

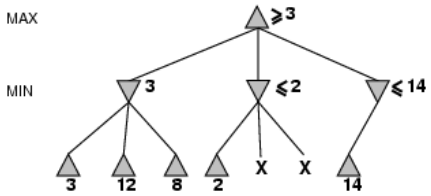
For example, from the following result of the minimax algorithm:

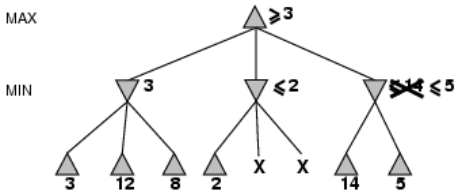


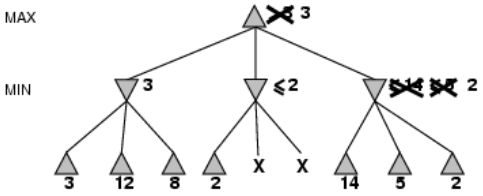
it is possible to prune the tree as follows:











Chapter 5

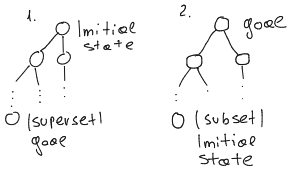
Automated Planning

* generative planning allows backtracking (i.e. reversible during searching) However the execution of actions in not reversible and non-deterministic.

Automated planning is an important problem solving activity which consists in synthesizing a sequence of actions performed by an agent that leads from an initial state of the world to a given target state. In particular:

- A **plan** is a partially or totally ordered set of actions needed to reach the target state.
- The solving process for deciding the steps is called **planning**. There exist two main types of planning: **generative planning**, which is an off-line type of planning that produces the whole plan before execution. In particular, it works on a snapshot of the world in which the initial state is fully known, each action has a deterministic effect and the plan execution is the only cause of changing in the world. Opposed to generative planning, there exist **reactive planning**, which is an on-line type of planning.
- A planner relies on a **domain theory**, i.e. an initial state, a goal state and a formal description of actions. The possible actions are identified by a name and are modelled through preconditions, which are the conditions which must hold to ensure that the action can be executed, and postconditions, which are the conditions which represent the effects of the specific action on the world.

  Sometimes, the domain theory consists of operators containing variables which define classes of actions. Thus, different instantiations of the variables could correspond to different actions.
- A planner is said to be:
  - **Complete**, whenever is able to find a plan if a plan indeed exists.
  - **Correct**, whenever the solution found leads from the initial state to the goal.
- Planning can be seen as a search activity. There exist many different views of planning as search, some of this are the following:
  - **Deductive planning as theorem proving**, in which states are sets of propositions that are true and operators are deductive rules.



- **Planning as search in the state space**, in which states are sets of propositions that are true and operators are actions.
- **Planning as search in the plan space**, in which states are partial plans and operators are plan refinement moves.

5.1 Linear Planning

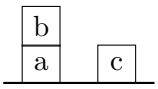
A linear planner is a planner which applies search in the state space and provides a totally ordered set of actions. In particular, the search algorithm can proceed in two ways:

1. forward, whenever the search process starts from the initial state and terminates as soon as a superset of the goal is found;
2. backward, whenever the search process starts from the target state and terminates as soon as a subset of the initial state if found. (goal regression)

5.1.1 Deductive Planning

Deductive planning uses first-order logic for representing states and actions and generates a plan by using theorem proving. In particular, there exist two formulations of deductive planning: the Green formulation and the Kowalsky formulation. In the Green formulation:

- States are described by means of predicates and they express which properties (*fluents*) are *true* in a given state s. For example, considering the block world problem:



  - $on(b, a, s)$, indicates that, in state s, b is on top of a.
  - $onTable(c, s)$, indicates that, in state s, c is on the table.

- Actions are described by means of logical implications in which the left-hand side of the implication includes the preconditions of the specific action, while the right-hand side of the implication includes the effects of the specific actions. For example, considering the action $putOnTable$ one can formulate:

$$on(X, Y, S) \wedge clear(X, S) \rightarrow \\ (onTable(X, do(putOnTable(X), S))) \wedge (clear(Y, do(putOnTable(X), S)))$$

  In particular, $on(X, Y, S) \wedge clear(X, S)$ are the preconditions, the functor $do(\cdot, S)$ returns a state and the predicates $onTable(X, \cdot)$ and $clear(Y, \cdot)$ are the postconditions.

- Plan construction is done by means of unification and resolution. For example, if the initial state is described as $\{on(b, a, s), onTable(c, s)\}$ is it possible to find a state S in which $onTable(b)$ is *true*? The resulting plan is $putOnTable(b)$, since every precondition of the action $putOnTable$ is *true* in the initial state.

* The Kowalsky formulation can be used to model a given move. For example, considering move(X, Y, Z):

Effects:
holds(clear(Y), do(move(X, Y, Z), S))
holds(on(X, Z), do(move(X, Y, Z), S))

Preconditions:
pact(move(X, Y, Z), S) :- holds(clear(X), S), holds(clear(Z), S), holds(on(X, Y), S), diff(X, Z)

Frame conditions:
holds(V, do(move(X, Y, Z), S)) :- holds(V, S), diff(V, clear(Z)), diff(V, on(X, Y))

Reachability:
poss(S0)
poss(do(A, S)) :- poss(A), pact(A, S)

Goal:
:- ...

- One has to explicitly specify all fluents that change and all fluents that do not change after a state transition, i.e. after taking an action. This problem is referred to as **frame problem**. For example, in the considered block world, the block c should have been considered by using **frame axioms**. For example, the following axioms tells that, after moving some block X from the top of Y to the top of Z, the block U is still on top of the block V and it is still clear:

$$on(\bigsqcup, V, S) \wedge diff(U, X) \rightarrow on(U, V, do(move(X, Y, Z), S))$$
$$clear(U, S) \wedge diff(U, Z) \rightarrow clear(U, do(move(X, Y, Z), S))$$

In the Kowalsky formulation: ✳

- Three main predicates are used:
  - $holds(rel, s/a)$, which describes all the relations rel that are $true$ in a given state s or made $true$ by the execution of the action a.
  - $poss(s)$, which indicates if a state s is reachable.
  - $pact(a, s)$, which indicate that it is possible to execute an action a in a state s, namely the preconditions of a are $true$ in s.

$$poss(S) \wedge pact(A, S) \rightarrow poss(do(A, S)) \qquad \text{(Transition axiom)}$$

- Only one frame axiom per action is needed instead of one frame axiom per fluent (e.g. in the Green formulation). For example, the frame axioms stated before can be summarized as follows:

$$holds(V, S) \wedge diff(V, clear(Z)) \wedge diff(V, on(X, Y)) \rightarrow holds(V, do(move(X, Y, Z), S))$$

  This states that every term different from $clear(Z)$ and from $on(X, Y)$ are $true$ after the execution of the action $move(X, Y, Z)$.

5.1.2 Stanford Research Institute Problem Solver (STRIPS)

STRIPS is an ad-hoc search algorithm used to solve planning problems, namely to construct plans. In particular:

- States are describes in terms of fluents, i.e. properties which are $true$ is the specific state.
- Actions are described by using three lists:
  - A PRECONDITIONS list, in which all fluents that should be $true$ for applying the specific action are stored.
  - A DELETE list, in which all fluents that become $false$ after the execution of the specific action are stored.
  - An ADD list, in which all fluents that become $true$ after the execution of the specific action are store.

Sometimes the ADD and DELETE lists are gathered in a EFFECTS list.

For example, considering the action $move(X, Y, Z)$:

PRECONDITIONS: $on(X, Y), clear(X), clear(Z)$
DELETE: $clear(Z), on(X, Y)$
ADD: $clear(Y), on(X, Z)$

- The frame problem is solved by means of the **strips assumption**: everything which is not in the ADD and DELETE list is unchanged.

The STRIPS algorithm is based on linear planning and uses backward search to construct a plan. Moreover, the algorithm uses two data structures: a goal stack and a data structure to store the description of the current state S:

Initialize the stack with the goals to reach
while the stack is not empty **do**
  if top(stack) $= a$ and $a\theta \subseteq S$ (a can be a conjunction of goals or a single goal)
    then $pop(a)$ and execute substitution θ on the stack
  else if top(stack) $= a$
    then select an action R with $a \in$ ADD(R), $pop(a)$, $push(R)$, $push$(PRECONDITIONS(R))
  else if top(stack) $= a_1 \wedge a_2 \wedge \cdots \wedge a_n$
    then $push(a_1), push(a_2), \ldots, push(a_n)$
  else if top(stack) $= R$
    then $pop(R)$ and apply R on S

The main drawbacks of this algorithm are the following:

- Since there exist a totally ordered set of actions to be made, i.e. in case of top(stack) $= a_1 \wedge a_2 \wedge \cdots \wedge a_n$ one has to decide in which order execute $push(a_1)$, $push(a_2)$, $\ldots$, $push(a_n)$, it could happen that in order to reach a goal, a_i, the effects obtained for reaching a previous goal, a_j, are cancelled, since the two goals could be not valid.
- The search space is large. To reduce the problem one could use some heuristics to explore the state space.

5.2 Non-Linear Planning

Non-linear planners are search algorithms which generate a plan by searching in the plans space. In particular, in the search tree each node is a partial plan and operators are plan refinement operations. Non-linear planners are represented by:

- A set of **actions** (instances of operators).
- A non-exhaustive set of **orderings** between actions.
- A set of **causal links**. In particular, a causal link is a triple, $\langle a_i, a_j, C \rangle$ which consists of two operators a_i, a_j and a sub-goal C, where C should be a precondition of a_j and an effect of a_i.

Moreover, the initial plan is an empty plan composed with only two additional actions:

- The **start** action, which has no preconditions and its effects match the initial state.
- The **stop** action, which has no effects and its preconditions match the goal state.

In particular:

- At each step either the set of operators or the set of orderings or the set of causal links is increased until all goals are met.
- A solution is a set of partially specified and partially ordered operators.
- To obtain a real plan, the partial order should be linearised in one of the possible total orders.

An intuitive non-linear planner is the following:

while the plan is not complete **do**
  select an action a_j which has a precondition not satisfied
  select an action a_i, which can be new or already in the plan, that has C among its effects
  add the order constraint $a_i < a_j$
  if a_i is a new action
    add the constraint $start < a_i < stop$
  add the causal link $\langle a_i, a_j, C \rangle$
  solve any threat on causal links

In particular:

- An action a_k is a **threat** for a given causal link, $\langle a_i, a_j, C \rangle$, if it has an effect that negates C and no ordering constraint exists which can prevent a_k to be executed between a_i and a_j. In some cases, in order to avoid this problem, a new ordering constraint has to be enforced (e.g. $a_k < a_i$ or $a_j < a_k$). However, not always this can be done. In these cases, it is necessary to backtrack and re-plan.

5.2.1 Partial Order Planning Algorithm (POP)

POP is a non-linear planner which takes as input the initial goal's operators and returns a plan:

function $POP(initialGoaloperators)$ **returns** $plan$:
  $plan \coloneqq INITIAL\_PLAN(start, stop, initialGoal)$
  loop:
    if $SOLUTION(plan)$
      then return $plan$
    else
      $S_n, C \coloneqq SELECT\_SUBGOAL(plan)$
      $CHOOSE\_OPERATOR(plan, operators, S_n, C)$
      $RESOLVE\_THREATS(plan)$
  end

In particular:

function $SELECT\_SUBGOAL(plan)$:
  select S_n from $STEPS(plan)$ with unsolved precondition C
  return S_n, C

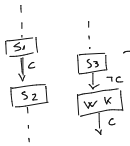
function $CHOOSE\_OPERATOR(plan, operators, S_n, C)$:
  select S with effect C from $operators$ or from $STEPS(plan)$
  if S does not exist
    then fail
  else
    add the causal link $\langle S, S_n, C \rangle$
    add the ordering constraint $S < S_n$
    if S is a new action added to the plan
      then add S to $STEPS(plan)$, add the constraint $start < S < stop$

function $SOLVE\_THREAT(plan)$:
  for each action S that threats a causal link between S_i and S_j
    choose either demotion (constraint $S < S_i$) or promotion (constraint $S_j < S$)
    if $plan$ is not consistent
      then fail

5.2.2 Modal Truth Criterion (MTC)

MTC is a construction process that guarantees planner completeness. In particular, MTC provides five plan refinement methods that ensure the completeness of the planner:

- **Establishment**, open goal achievement by means of:
  - A new action to be inserted in the plan.
  - An ordering constraint with an action already in the plan.
  - A variable assignment, namely a unification.
- **Promotion**, which represents an ordering constraint that imposes the threatening action before the first action of the causal link.
- **Demotion**, which represents an ordering constraint that imposes the threatening action after the second action of the causal link.



- **White knight**, which inserts a new operator, or one already in the plan, in order to re-establish the condition needed to execute the second action of the specific causal link. This is used when neither promotion nor demotion can be used, i.e. when the threatening action can only be put between the actions of the specific causal link.
- **Separation**, which inserts non co-designation constraints between the variables of the threatening action's effect and the threatened action's precondition so to avoid unification. This is useful when variables have not yet been instantiated. For example, given the following causal link:

$$pickup(X) \xrightarrow{holding(X)} stack(X, b)$$

any threat imposed by $stack(Y, c)$ can be solved by imposing $X \neq Y$.

5.3 Hierarchical Planning

Hierarchical planners are search algorithms which manage the creation of complex plans at different levels of abstraction, by planning the simplest details only after finding a solution for the most difficult ones. Given a specific goal:

- A hierarchical planner performs a **meta-level search** to generate a **meta-level plan** which leads from a state that is very close to the initial one to a state which is very close to the goal.
- The plan is the completed with a **lower-level search** which takes into account the omitted details.

In particular, there exist two main types of hierarchical planners:

- Planners based on values of criticality assigned to preconditions. These planners are linear planners.
- Planners based on macro operators and atomic operators. These planners are non-linear planners.

5.3.1 Abstraction-Based Stanford Research Institute Problem Solver (ABSTRIPS)

ABSTRIPS is a hierarchical planner which assigns to each precondition a specific criticality value. The algorithm proceeds at different levels of abstraction as follows:

1. A criticality threshold value is fixed.
2. All the preconditions whose criticality value is less then the threshold value are considered *true*.
3. STRIPS, or other planners, finds a plan that meets all the preconditions whose value is greater or equal to the threshold value.
4. The threshold value is than lowered.
5. The points 2, 3, 4 are repeated until all the preconditions of the original rules have been considered.

i.e. all these preconditions are not considered (they can also be false this is why the plan needs to be refined)

5.3.2 Atomic Operators and Macro Operators

Most hierarchical planners are based on atomic and macro operators. In particular:

- **Atomic operators** represent elementary actions which can be directly executed by an agent.
- **Macro operators** represents a set of elementary actions which can be decomposed into atomic actions.

In this case, the planning algorithm can be either linear or non-linear. A hierarchical non-linear algorithm is similar to POP where, at each step, one can choose between reaching an open goal with an operator (including macro operators) or expanding a macro operator of the plan. when a macro action is decomposed, then orderings and causal links need to be updated.

5.4 Graph Planning

Graph planning is one of the most efficient type of complete and correct generative planning. Graph planning introduces the concept of time, namely when an action is executed. In particular:

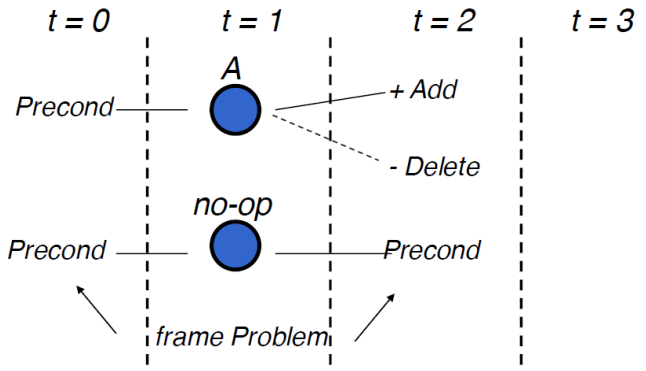
- Graph planners build a **planning graph** which contains all possible plans. At each step of the search this data structure is extended.
- Graph planners return either the shortest possible plan (optimal plan), starting from the initial state and ending in the target state, or an inconsistency.
- Graph planners inherits early commitment (perform action as soon as possible) from linear planners, and the ability to create partially ordered sets of actions from non-linear planner.
- In graph planning, each parameter has a type

- Actions are represented with a PRECONDITIONS list, and ADD list and a DELETE list.
- Objects have a type.
- There exist a *no-op* action which does not change the state.
- States are represented as sets of predicates which are *true* in a given state.

A planning graph is composed of proposition levels, in which nodes represent propositions, and action levels, in which nodes represent actions. In particular: the planming graph is a directed leveled graph:

- Level $t = 0$ of the graph corresponds to the initial state and it is a proposition level.
- Acrs between nodes are divided into:
  - Precondition arcs (proposition → action).
  - Add arcs (action → proposition).
  - Delete arcs (action → proposition).

An example of planning graph is the following:



- In each time step, an action A can be inserted if in the previous time step all preconditions of A are *true*.
- There exist a *no-op* action for each proposition. In this way, every precondition of the previous step is brought forward.
- In the same action level could exist two or more actions which can not be executed simultaneously.

During the construction of the planning graph inconsistencies are identified. In particular:

- At the same level, two actions can be inconsistent. These two are said to be **mutually exclusive**. Moreover:
  - The two actions have **inconsistent effects** when one of these negates the effect of the other. i.e the some proposition is reached from one add arc and one delete arc.
  - The two actions **interfere** with each other when one of these deletes a precondition of the other.
  - The two actions have **competing needs** when both of these have mutually exclusive preconditions.

- At the same level, two propositions can be inconsistent if one is the negation of the other or if all the paths used to reach the two preconditions are mutually exclusive. These two conditions are said to be **mutually exclusive**.

The algorithm for building the planning graph is the following:

1. Every *true* proposition in the initial state is included in the first proposition level.
2. Creation of an action level:
   (a) For every operator and every way to unify its preconditions to non-mutually exclusive propositions in the previous level, enter an action node.
   (b) For every proposition in the previous level, enter a *no-op* operator.
   (c) Check if the action nodes do not interfere with each other, otherwise mark them as mutually exclusive.
3. Creation of a proposition level:
   (a) For each action node in the previous level, add the propositions in his ADD list through solid arcs.
   (b) For every *no-op* in the previous level, add the corresponding proposition.
   (c) For each action node in the previous level, link propositions in his DELETE list using dashed arcs.
   (d) Identify inconsistent propositions.

Once the planning graph is built, one has to extract a **valid plan**, i.e. a connected and consistent sub-graph of the planning graph. This is done by backtracking from the last level of the planning graph. In particular, a valid plan has the following properties:

- Actions of the same level can be performed in any order, i.e. they do not interfere with each other.
- Propositions of the same level are not mutually exclusive.
- The last level contains all the literals of the goal and these are not marked as mutually exclusive.

5.5 Conditional Planning

Generative planners could have some problems during execution. For example, an action could be executed while its preconditions are not satisfied or its effects could be not the one expected. In order to avoid this problem, some planners run under the **open world assumption** instead of the closed world assumption. According to this hypothesis, all the unknown information is neither *true* nor $false$. In particular:

this problem can be solved using conditional planners or using reactive planners.

- Unknown information can be retrieved via **sensing actions**, which are actions with multiple outcomes that are added to the plan.
- A **conditional planner** is a search algorithm which generates various alternative plans for each source of uncertainty of the plan.
- A **conditional plan** is composed of:
  - Causal actions, i.e. classic actions.
  - Sensing actions, used to retrieve unknown information.
  - Several alternative partial plans of which only one will be then executed. If one has n sensing actions and each sensing action has 2 possible outcomes, then one will have 2^n partial plans.

Conditional planning, however, implies a combinatorial explosion of the search space and requires a lot of memory to be implemented. Moreover, not always all the alternative contexts are known in advance.

5.6 Reactive Planning

Reactive planners are on-line algorithms capable of interacting with the world, in order to deal with the dynamicity and the non-determinism of the environment. In particular:

- They observe the world during the planning stage.
- They acquire unknown information.
- They monitor the implementation of actions and check the effects.
- They interleave planning and execution.

Moreover, pure reactive systems do not plan, they only react as triggers to world variations. Modern planners are hybrid and integrate generative planning and reactive planning in order to exploit the advantages of both approaches.

Chapter 6

Constraint Satisfaction

Many artificial intelligence problems can be seen as constraint satisfaction problems (CSPs). The objective of these problems is to find a state in which a set of constraints are met. In particular, a constraint satisfaction problem is defined on a finite set of variables:

- $(X_1, X_2, \ldots, X_n)$, which are decisions that one has to take.
- $(D_1, D_2, \ldots, D_n)$, which are the domains of possible values.
- A set of constraint. A constraint $c(X_{i1}, X_{i2}, \ldots, X_{ik})$ between k possible variables is a subset of the Cartesian product $D_{i1} \times D_{i2} \times \cdots \times D_{ik}$ that specifies which values of the variables are compatible with each other.

A solution to a constraint satisfaction problem provides an assignment of all the variables that satisfies all the constraints. In particular, these problems can be solved through search:

- A possible search tree for a CSP is obtained after establishing an order for variables, i.e. each level of the tree corresponds to a variable and each node corresponds to a possible value assignment.
- A leaf of the tree would then represent an assignment of values to all the variables. If this assignment satisfies all the constraints, then the corresponding leaf represents a solution to the problem, otherwise a failure.
- In an n-variable problem in which all the domains have the same cardinality d, the number of leaves of the search tree is equal to d^n. Therefore, a smart tree exploration strategy is essential.

Given a CSP there exist two possible approaches to its solution:

- **Propagation algorithms**, which are based on the propagation of constraints to eliminate a priori, while searching, portions of the search tree which would lead to a failure.
- **Consistency techniques**, which are based on the propagation of constraints in order to derive a simpler problem than the original.

- Generate and test: assign a value to each variable and, only at the end, checks the assignments.
- Standard backtracking: like generate and test but checks if an assignment is feasible as soon as the assignment has been made.

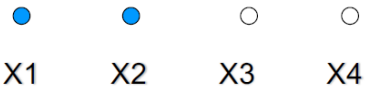
Both of these are a-posteriori techniques (i.e. check feasibility after assignment)

6.1 Propagation Algorithms

Propagation algorithms are, in a sense, opposed to the classic a-posteriori methods (e.g. generate and test and simple backtracking) in which constraints are used after the instantiation of variables, and not before. The idea behind a-priori propagation algorithms is an active use of the constraint during search.

6.1.1 Forward Checking

Forward checking is a propagation algorithm in which, after each assignment of the variable X_i, the algorithm propagates all constraints involving X_i and all variables that are not yet instantiated. In particular, this method is very effective especially when free variables' domains are associated with a reduced set of possible values and, therefore, are easily assignable. For example:



- X_1 and X_2 have already been instantiated.
- The standard backtracking algorithm would check $c(X_1, X_2)$.
- The forward checking checks $c(X_2, X_3)$ $\forall$ values $\in D_3$ and $c(X_2, X_4)$ $\forall$ values $\in D_4$. The values of D_3 and D_4 which are incompatible with the current assignment of X_2 are deleted.

*

6.1.2 Look Ahead

– does forward checking + checks other not yet instantiated variables

Look ahead algorithms are propagation algorithms which check the existence, in the domains associated with the non-assigned variables, of values that are compatible with the constraints containing only non-instantiated variables. In particular, supposed that one has already instantiated variables X_1 to X_{h-1}, there exist two different types of look ahead algorithms:

- The **partial look ahead algorithm** (PLA), which allows constraint propagation containing the not yet instantiated variable X_h with the not yet instantiated future variables, $X_{h+1}, \ldots, X_n$. Namely, for each value in the domain of X_h, it checks if in the domains of not yet instantiated variables $X_{h+1}, \ldots, X_n$ there are values compatible with it.
- The **full look ahead algorithm** (FLA), which allows constraint propagation containing the not yet instantiated variable X_h with the not yet instantiated variables, $X_{k+1}, \ldots, X_{h-1}, X_{h+1}, \ldots, X_n$. Namely, for each value in the domain of X_h, it checks if in the domains of not yet instantiated variables $X_{k+1}, \ldots, X_{h-1}, X_{h+1}, \ldots, X_n$ there are values compatible with it.

For example:

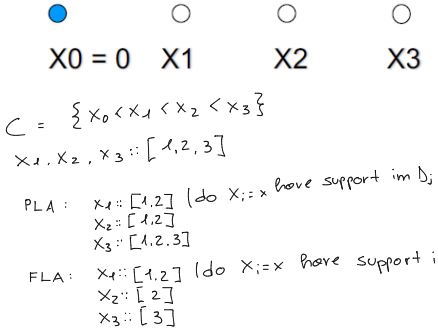
* First-fail heuristic: forward checking selecting always the variable whose domain is the smallest one.
(Minimum remaing values heuristic) i.e. most difficult variables are assigned first.

- PLA verifications:

  1. For each value in D_1 checks if there exists at least one value in D_2 and at least one value in D_3 which are compatible.
  2. For each value in D_2 checks if there exists at least one value in D_3 which is compatible.

- FLA verifications:

  1. Every verification done by PLA.
  2. For each value in D_2 checks if there exists at least one value in D_1 and at least one value in D_3 which are compatible.
  3. For each value in D_3 checks if there exists at least one value in D_1 and at least one value in D_2 which are compatible.

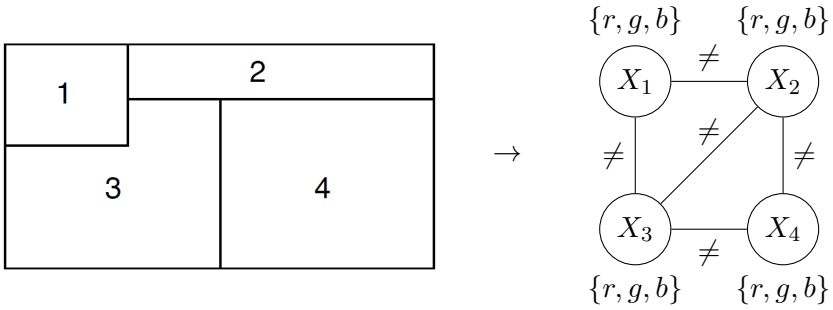


6.2 Consistency Techniques

In contrast to propagation algorithms consistency techniques reduce the original problem by eliminating domain values that cannot appear in a final solution. In particular:

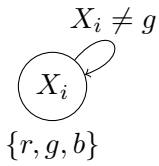
- All consistency techniques are based on a representation of the problem as a network (i.e. a graph) of constraints in which the nodes represent variables and the arcs are the constraints.
- The arcs can be oriented or non-oriented (e.g. the constraint $>$ is represented by a directed arc, while the constraint $\neq$ from a non-directed arc).

For example, supposing one needs to colour the following rectangle such that two contiguous regions are coloured with different colours, then the constraint-graph would be the following:



In particular, there exist several algorithms that implement different degrees of consistency:

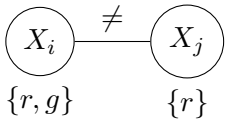
- **Node consistency** (consistency of level 1). A node of a graph of constraints is consistent if, for each value $x \in D_i$ the unary constraint (self-arc) on X_i is satisfied. Moreover, a graph is said to be node-consistent if all its nodes are consistent. For example:



  - The graph is not node-consistent because because of the value $g \in D_i$ which violates the unary constraint on X_i.
  - To make the graph node-consistent it is necessary to eliminate g from D_i.

- **Arc consistency** (consistency of level 2). This type of consistency is obtained from a node-consistent graph. An arc $a(X_i, X_j)$ is consistent if, for each value $x \in D_i$, it exists at least one value $y \in D_j$ such that the constraint between ~~i~~ and ~~j~~ is satisfied. In particular, y is called **support** for ~~i~~. Moreover, a graph is said to be arc-consistent if all its arcs are consistent. For example:

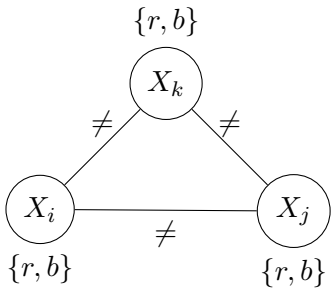
full look-ahead does only one iteration while arc-consistency iteratively apply full look-ahead if at least there has been one elimination in the previous iteration



  - The graph is not arc-consistent because, considering the value $r \in D_i$, there is no value belonging to D_j which satisfies the given constraint.
  - To make the graph arc-consistent it is necessary to delete the value r from D_i.

  The removal of a value from the domain of a variable makes further checks needed. Arc consistency is obtained through an iterative process that converges to a stable and arc-consistent network.

- **Path consistency** (consistency of level 3). This type of consistency is obtained from an arc-consistent graph. A path between the nodes X_i, X_j and X_k is consistent if, for each value $x \in D_i$ and $y \in D_j$ (which are node-consistent and arc-consistent), there exists a value $z \in D_k$ that satisfies the constraints $P(i, k)$ and $P(k, j)$. Moreover, a graph is said to be path-consistent if all its paths are consistent. For example:



  - The graph is not path-consistent since at least two of the three nodes, X_i, X_j, X_k, would assume the same value.
  - To make the graph path-consistent it is necessary to add a third possible value to each domain, namely $D_i, D_j, D_k = \{r, g, b\}$.

- **k-consistency** (consistency of level k). In principle, for a CSP of k variables, for each $(k-1)$-tuple of values which are consistent with the imposed constraint, it should exist a k-th value that satisfies the constraint among all k variables.

  In general, if a graph containing n variables is k-consistent, with $k < n$, then searching is still needed.

  If a graph containing n variables is n-consistent, then one can find a solution without searching.