Alma Mater Studiorum · Università di Bologna

**Second Cycle Degree**
**Artificial Intelligence**

# Fundamentals of Artificial Intelligence and Knowledge Representation
## Module 1

**Student:**
Matteo Canghiari

**Academic Year 2025/2026**

# Chapter 1

# Searching for solutions

Many AI problems can be solved by exploring the **solution space**. A solution space is a set of all the sequences of actions that an agent can apply. The agent examines all the possible sequences of actions and chooses the best one. The goal is to reach the solution starting from a **initial state**. The process of trying different sequences is called **search**.

Usually, it's useful to think about the **search** process as a **search tree**, where:

- The initial state corresponds to the **root** of the tree.
- Each branch that makes up the tree defines the **action** that can performed by the current node.
- The nodes represent the subsequent reachable states. However, a certain node can be a **leaf node**. A leaf node is a new state to expand, a solution or a dead-end.

Previously, we said that a solution is a sequence of actions, so we need to define two main operations that allow us to build a sequence. We do this by **expanding** the current state, applying each possible action to the current node, **generating** a new set of states[1]. Generally, the set of all leaf nodes available for expansion at any given point is called **frontier** or **fringe**. The process of expanding nods continues until either a solution is found or there are no more states to expand.

Finally, concluding this first introduction, we say that all the search algorithms are named **search strategies**, and typically they all share the same structure, varying by the way they choose which state needs to be expand.

[1]Every time we expand the current node, new state are generated.

## 1.1 Infrastructure for search strategies

Every search strategy uses different kind of data structures to keep in mind how the search tree was built. Each node of the tree corresponds to a data structure, containing:

- State: the state in the state space.
- Parent: the node that generated this node.
- Action: the action taken by the parent to generate the node.
- Depth: defining how deep is the node, in which level it belongs to.
- Path-Cost: the cost of the path from the initial state to this node, usually denoted by $g(n)$.

Now that we have nodes, we need somewhere to put them. The fringe needs to be stored in such a way that the search algorithm can easily choose the next node to expand. The appropriate data structure is a **queue**. It can be a **FIFO**, **LIFO** or a **priority queue**[2].

## 1.2 Measuring effectiveness

Before discussing the large set of search strategies, we need to consider the basic criteria that define the effectiveness of the algorithms. Typically, the performance of an algorithm is evaluated by four ways, as follows:

- **Completeness**: does the algorithm guarantees to find a final solution?
- **Optimality**: does the strategy find the best solution?
- **Time complexity**: how long does the algorithm take to find a solution?
- **Space complexity**: how much memory is needed to carry out the search?

As we said, search algorithms are called search strategies, or briefly **strategies**. Strategies are divided into two main types:

- **Non-informed strategies**: they don't use any domain knowledge, apply rules arbitrarily, and do exhaustive search.

[2]We remember that:

- LIFO queues pop the newest element.
- FIFO queues pop the oldest element.
- Priority queues pop the element with the highest priority.

- **Informed strategies**: they use domain knowledge, apply rules following **heuristics**.

# Chapter 2

# Non-informed strategies

This chapter covers several search strategies that come under the heading of **non-informed strategies**. The term *non-informed* means that the strategies have no additional knowledge about the domain; all they can do is generate successors and distinguish a goal state from a non-goal state. We introduce five non-informed search strategies:

- BFS.
- Uniform-cost search.
- DFS.
- DFS with limited depth.
- Iterative deepening.

## 2.1 Breadth-first search

**Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, and so on; until it is found the goal-state or the goal-node.

At algorithmic level, this is achieved very simply using a FIFO queue for the fringe, so the oldest node will be the first expanded. Before selecting the next node to expand, the goal-test is applied to each node when it is generated.

This strategy ensures **completeness**, but the **shallowest** goal node is not necessary the **optimal** one. BFS can be optimal if all the actions have the same path-cost. In addition, breadth-first search seems to take a quite huge of time and memory.

Suppose a search tree where every node has $b$ successors. The root node generates $b$ nodes at the first level, each of which generates $b$ more successors, for a total of $b^2$ nodes. Now if we consider that the goal-node has $d$ depth, in the worst case the total number of nodes generated is

$$b + b^2 + b^3 + \cdots = O(b^d).$$

This complexity is the same for both time and memory. As the time complexity, the memory takes into account every node expanded inside the **explored set** to avoid **loopy path**; the space complexity grows exponentially with the number of $b$ successors and the depth $d$ of the goal node. The problem of memory seems to be the most serious.

In general, any exponential complexity seems to be scary, and in this case uninformed strategies cannot solve massive problems.

## 2.2 Uniform-cost search

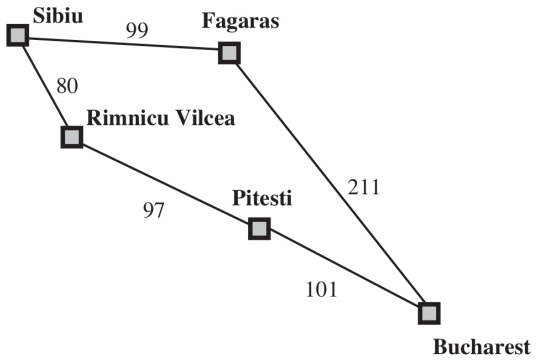
BFS algorithm can be a complete and optimal search strategy if and only if all the step costs are equal; generally this type of strategy is called **uniform-cost search**. Instead of expanding the shallowest node, so taking the *oldest* node inserted inside the fringe, uniform-cost search expands the node with the *lowest* path cost $g(n)$. Therefore, the fringe is not anymore a LIFO queue, but it becomes a *priority queue* ordered by $g$, so by the path cost made from the initial state to the current one.

In addition to the ordering of the queue, there is another significant difference from BFS. If in BFS algorithm the *goal test* is applied when each node is generated, in the uniform-cost search strategy the *test* is applied to a node when it is selected for expansion.

Let's consider an example for better understanding.

**Example**

i.e. Reach Bucharest from Sibiu.



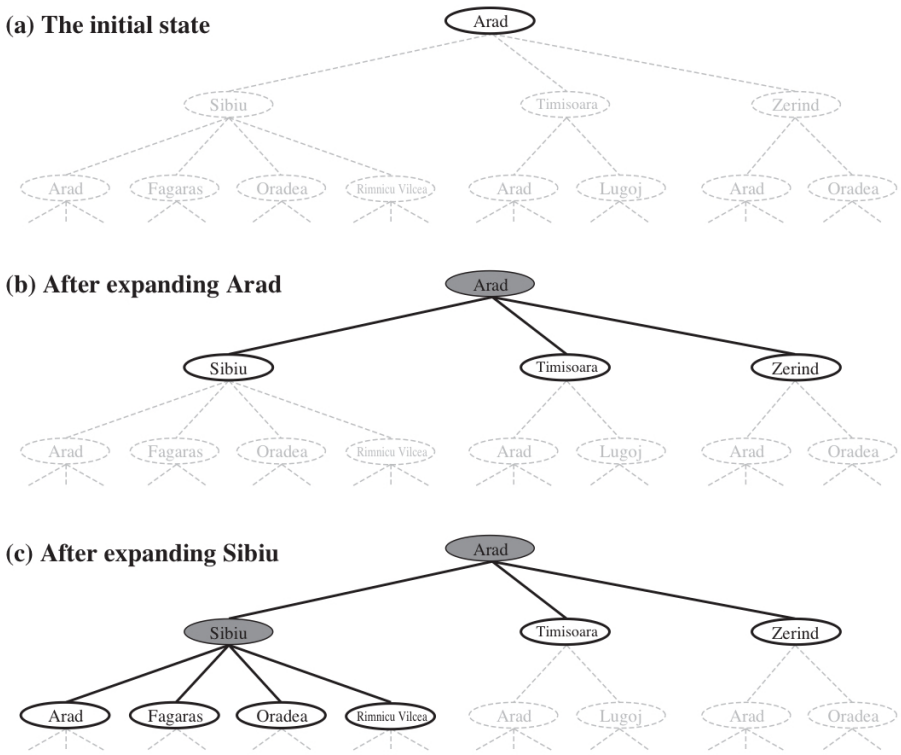

The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. Rimnicu Vilcea is the node selected for the expansion, the cost of which is lower than Faragas. Therefore, the fringe now contains Faragas and Pitesti, costing $80 + 97 = 177$. The least-cost node is now Fagaras, so it is expanded, adding Bucharest at a cost of $99 + 211 = 310$. Now a goal node has been generated, but uniform-cost search keeps going[a], selecting Pitesti and adding a second path to Bucharest with a cost of $80 + 97 + 101 = 278$. Finally, the algorithm chooses the second path generated, the cost of which is less than the first; then it applies the *test goal* and finds out that Bucharest is the *goal-node*, and the solution is returned.

[a] We remember that uniform-cost search applies the goal test only when a node is selected, not generated.

## 2.3 Depth-first search

**Depth-first search** always expands the *deepest* node in the current fringe. The search proceeds immediately to the deepest level of the the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so the search **goes back** to the next deepest node, which is located at $m-1$ depth of the current nodes deleted from the frontier.

While BFS uses a FIFO queue, DFS uses a LIFO queue, the most recently generated node is chosen for expansion. This search strategy ensures **optimality** but not **completeness**, for instance, in the figure shown below the algorithm will follow $Arad - Sibiu - Arad - Sibiu$ loop forever. However, the search tree can be modified at no-extra memory cost: DFS checks new generated states against those on the path from the root to the current node. This avoids infinite loops, but does not guarantees the proliferation of redudant paths.




Compared to BFS, depth-first search has one major advantage, the space complexity; for any search tree, it needs to store only a single path from the root to a leaf node. Given a branch factor $b$ and a maximum depth $m$, DFS requires storage of only $O(bm)$ nodes. Even if the space complexity seems to be linear, in the worst case the time complexity reaches the exponential order[1].

## 2.4 Depth-limited search

As we said, depth-first search is not complete: the algorithm could enter in infinite cycles. This problem can be avoided introducing some changes to the original algorithm, such as a predetermined depth limit $l$. All the nodes at depth $l$ are treated as leaves node, they have no successors. This search strategy is called **depth-limit search**. Unfortunately, even though it solves the problem of infinite cycles, it introduces another source of **no-completeness**; if we choose a limit $l < d$, the shallowest goal-node may not be achieved. Depth-limited search will also be **non-optimal** if we choose $l > d$.

[1]We remind you that the maximum depth $m$ can be bigger than the shallowest depth $d$, introduced in the breadth-first search strategy.

It's time complexity is exponential, $O(b^l)$, and its space complexity is linear, $O(bl)$[2].

## 2.5 Iterative deepening

**Iterative deepening search** is a strategy usually used with the depth-first search algorithm, which defines iteratevely the *best depth limit*. It does this by gradually increasing the limit $l$, as we have seen in depth-limited search, until the target node is found. This will occur when the depth limit $l$ reaches $d$, the depth of the shallowest node.

It combines the advantages of depth-first search and breadth-first search strategies. Like DFS, its memory complexity is linear, to be precise $O(bd)$[3]. Like BFS, it is **complete** when the brancing factor $b$ is finite and **optimal** when the path cost increases as a function of the node depth.

Iterative deepening may seem like a waste of computation because the initial nodes are generated multiple times, but the execution time never gets worsen. The reason is that in a search tree with the same branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times. Let's consider an example for better understanding.

**Example**

i.e. Considering the branching factor equal to 10 and the depth of the shallowest node equal to 5, briefly:

$b = 10$
$d = 5$

and given the equation which defines the total number of nodes generated by iterative deepening in the worst case

$$N(IDS) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$

we have

$$N(IDS) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$$

while we could get the same amount if we decide to use breadth-first search strategy

[2]Depth-first search can be viewed as a special case of depth-limit search with $l = \infty$.

[3]Iterative deepening search, such as the depth-first search, only maintains in memory one path at the time; it is discarded if it does not contains the goal node.

$$N(BFS) = 10 + 100 + 1.000 + 10.000 + 100.000 = 111.110$$

## 2.6 Summary of Non-informed strategies

| **Criterion** | BFS | UCS | DFS | DLS | IDS |
|---|---|---|---|---|---|
| **Time** | $O(b^d)$ | $O(b^d)$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ |
| **Space** | $O(b^d)$ | $O(b^d)$ | $O(bm)$ | $O(bl)$ | $O(bd)$ |
| **Optimal** | No | Yes | Yes | No | Yes |
| **Complete** | Yes | Yes | No | Yes if $l \geq d$ | Yes |

4

[4]Comparison of Breadth-first search, Uniform-cost search, Depth-first search, Depth-limited search and Iterative deepening search.

# Chapter 3

# Informed strategies

This paragraph shows how **informed search strategies** can find solutions more efficiently than **non-informed search strategies**.

The general approach about informed search strategies is defined by **best-first search strategy**. Best-first algorithm select a node for the expansion based on an **evaluation function**, briefly written as $f(n)$. Generally, this type of search strategy uses evaluation function as an estimation of the effort required to reach the final state; the node with the lowest evaluation is expanded first.

The choice of the evaluation function $f(n)$ determines the search strategy. Most best-first algorithms include as a component of $f(n)$ a **heuristic function**, denoted $h(n)$.

$h(n)$ = estimated cost of the path from the current node $n$ to the final state[1].

We will examine two main applications of the best-first search strategy, which are:

- Greedy best-first search.
- A* search.

## 3.1 Greedy best-first search

**Greedy best-first search** tries always to expand the node closest to the goal state, in a such a way it wanted to build the path that leads to the quickest solution.

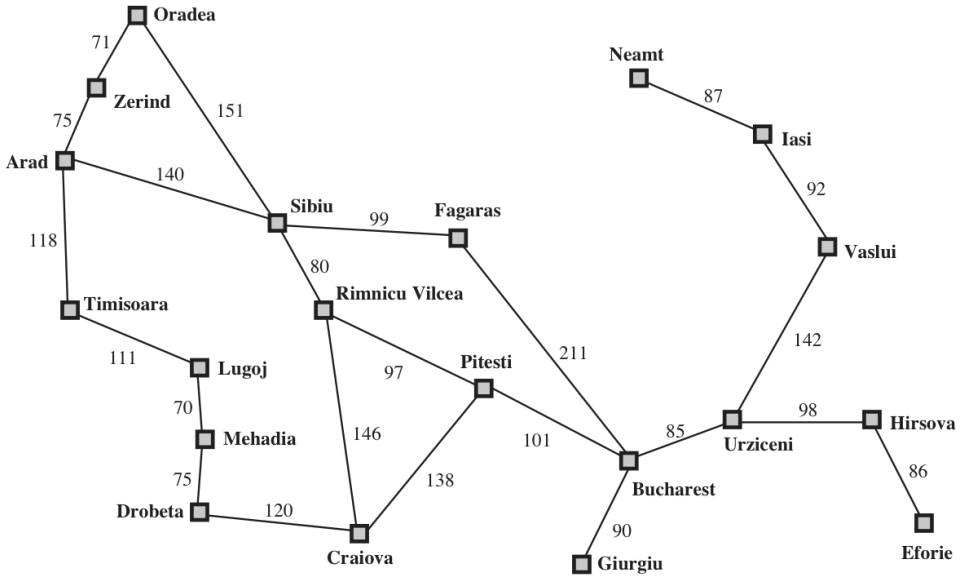
[1]Until now, we saw the $g(n)$ function as the path cost from the root node to the current state; selecting the node for the expansion with the lowest $g(n)$. Instead the heuristic function $h(n)$ defines the distance from the current node to the final state!

Thus, it evaluates nodes by using just the *heuristic function*, $h(n)$. In this case, the *evaluation function* is equal to the *heuristic function*, $f(n) = h(n)$.

Let's see how this works for the Romania graph example[2].

**Example**

i.e. Romania graph, starting from Arad and arriving in Bucharest.




We use the **straight-line distance** heuristic, which we will briefly call $h_{SLD}$. Given Bucharest as the goal state, we must know the straight-line distances to Bucharest. For instance, $h_{LSD}(Arad) = 366$. The table below shows each straight-line distance from any given node to Bucharest.

| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

Given the table, the first node to be expanded from Arad will be Sibiu, because

[2]Remember that is always possible to make a tree from a graph, and vice versa.

it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is closest. Fagaras generated Bucharest, which is the goal. For this problem, greedy best-first search, using the heuristic $h_{LSD}$, finds a solution without ever expanding a node that is not on the solution path.

$$\langle Arad, Sibiu, Fagaras, Bucarest \rangle = 450$$

However, the proposed solution is not optimal: the path via Sibiu, Rimnicu Vilcea, Pitesti and Bucharest is lower than the first one.

$$\langle Arad, Sibiu, Rimnicu, Pitesti, Bucharest \rangle = 418$$

This example shows why the algorithm is called **greedy**, at each step it tries to get as close to the goal as it can. By the way, this search strategy is **non-optimal** and may be **incomplete**, if the data structure presents loopy path.

In the worst case, if the depth of the shallowest node is $d$ and the branching factor is $b$, either the time and space complexity will be exponential, $O(b^d)$.

## 3.2 A* search

**A* search** is the most known case of best-first search algorithm[3]. It evaluates nodes for the expansion combining $g(n)$, the cost to reach the node $n$, and $h(n)$, the cost to get from the node $n$ to the goal state.

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

In this case, the evaluation function $f(n)$ give us the estimated cost of the cheapest solution through $n$. The algorithm is very similar to uniform-cost search, but instead of using only $g(n)$, it associates each node with the sum $g(n) + h(n)$. The next node selected for the expansion will be the state with the lowest value of $g(n) + h(n)$.

Provided the heuristic function $h(n)$ that satisfies certain conditions, A* search is both **complete** and **optimal**; if the heuristic function does not meet certain criteria, A* search does not guarantee to find the optimal solution. Which are these conditions?

**Feasibility**

[3]Basically, A* search try to combine the benefits of DFS, efficiency, with the ones of uniform-cost search, optimality and completeness.

The first condition required for optimality is that the heuristic function $h(n)$ is a **feasible** heuristic. A feasible heuristic is one that *never overstimates* the real cost to reach the goal state.

We suppose that with $h^*(n)$ we indicate the true distance between the current node and the goal state. The heuristic function $h(n)$ is feasible if we always have, for any node $n$ of the search tree, that:

$$h(n) \leq h^*(n).$$

Ensured this condition, the heuristic $h(n)$ is feasible.

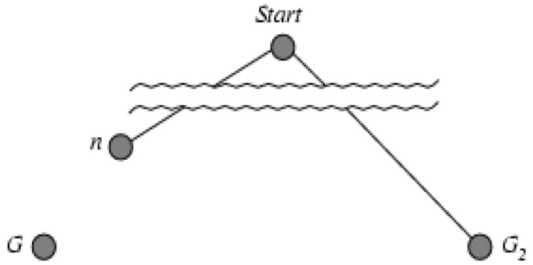
**Optimality theorem**

If $h(n) \leq h^*(n)$ for each node, then the A* search algorithm always finds the optimal path to the goal state.

Let's see an example for better understanding.

**Example**

i.e. Optimality proof.




Given this search tree, we have:

- **G**: the goal node or the goal state.
- **Start**: the state representing the root node.
- **n**: which is the unexpanded node already present in the priority queue, such that **n** is on the optimal path.
- $\mathbf{G_2}$: which defines the **sub-optimal** path. In this data structure, we are reaching the same goal node **G** from different trails.

We need to show that, if the A* algorithm has inside its queue a sub-optimal goal $\mathbf{G_2}$ and a node **n** which is on the optimal path to the goal node **G**, the evaluation function $f(n)$ will be less or equal to the evaluation function $f(G_2)$, denyng to A* algorithm to expand the node $\mathbf{G_2}$ first.

1. **Evaluation of the optimal goal.**

   The evaluation function of **G** is:

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

   $h(G)$ is equal to 0, because the cost to reach the goal node itself cannot be greater than 0.

   $$f(G) = g(G) + h(G) = g(G) + 0 = g(G)$$
   $$f(G) = g(G).$$

2. **Evaluation of the sub-optimal goal.**

   The evaluation function of $\mathbf{G_2}$ is:

   $$f(G_2) = g(G_2) + h(G_2)$$

   from the same reasons described in the first point, the evaluation function is equal to the real-cost function:

   $$f(G_2) = g(G_2).$$

3. **Comparison of G and $\mathbf{G_2}$.**

   Knowing that $\mathbf{G_2}$ is the sub-optimal path, the total cost to reach the goal node via $\mathbf{G_2}$ must be strictly greater than the total cost to reach the goal node via **G**, so:

   $$g(G_2) > g(G)$$

   the same condition can be expressed as follows:

   $$f(G_2) > f(G).$$

3. **Evaluation of node n.**

   Given **n** a node not expanded, but already present in the priority queue, the evaluation function $f(n)$ is:

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

   from the **optimality theorem**, the estimated cost of **n** must be less or equal to the real cost of **n**:

   $$g(n) + h(n) \leq g(n) + h^*(n)$$

   where the function $h^*(n)$ defines the real cost to reach the goal from **n**. Therefore, knowing that **n** is on the optimal path, the sum of the cost to reach **n** and the real-cost between **n** and **G**, is exactly the same effort to achieve **G**.

   $$f(n) = g(n) + h^*(n) = g(G)$$

   Replacing $g(G)$ with $f(G)$, from the first point, we have:

   $$f(G) = g(n) + h^*(n)$$

finally:

$$f(n) \leq g(n) + h^*(n) = f(G)$$
$$f(n) \leq f(G).$$

**4. Conclusion.**

Combining the results:

$$f(n) \leq f(G) \text{ and } f(G) < f(G_2)$$

the final conclusion is:

$$f(n) < f(G_2).$$

We can conclude saying that the heuristic function $h(n)$ is **feasible**, the A* algorithm will define always **optimal** solutions for search trees.

### Consistency

The second condition is slightly stronger than feasibility, and it is required only when A* algorithm is applied to search graph[4]. The A* algorithm for search graphs is optimal and complete if the heuristic function $h(n)$ is **consistent**. A heuristic function $h(n)$ is consistent if, for every node **n** and every successor $\mathbf{n}^*$ of **n** generated by an action **a**, the estimated cost of reaching the goal node **G** from **n** is no greater than the step cost of getting $\mathbf{n}^*$ and reach the goal **G** from it:

$$h(n) \leq c(n, a, n^*) + h(n^*).$$

This is a form of the **triangle inequality**, which says that each side of the triangle cannot be longer than the sum of the other two sides. The inequality makes perfect sense either for feasibility condition: if there were a trail from **n** to **G** via $\mathbf{n}^*$ cheaper than $h(n)$, that would violate the **feasibility** condition[5].

**Consistency theorem**

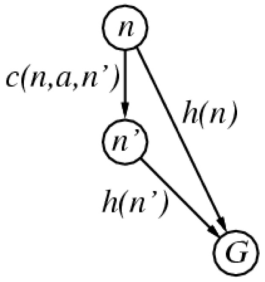
If $h(n)$ is consistent then A* search applied to graphs is optimal.

[4]Until now, we have assumed so far that the search space is a tree not a graph.

[5]We remember that for feasibility condition we mean: for every node $n$, the heuristic function $h(n)$ is always less or equal to the true distance between the node $n$ and the goal node $G$ ($h(n) \leq h^*(n)$).

**Example**

i.e. Consistency proof.




We want to prove that, if $h(n)$ is consistent, then the values of $f(n)$ along any path are non-descresing.

**1. Defining successor $\mathbf{n}^*$ of n.**

Suppose $\mathbf{n}^*$ is a successor of the node **n**, then the evaluation function is defined as:

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

if $\mathbf{n}^*$ is a successor of **n**, the real-cost function can be expressed as:

$$g(n^*) = g(n) + c(n, a, n^*)$$

therefore, the evaluation function $f(n^*)$ becomes:

$$f(n^*) = g(n) + c(n, a, n^*) + h(n^*).$$

**2. Comparison of $\mathbf{f(n^*)}$ and f(n)**

As we know, the evaluation function of **n** is:

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

and from the previous step the evaluation function of $\mathbf{n}^*$ is:

$$f(n^*) = g(n) + c(n, a, n^*) + h(n^*)$$

combining the two results we have:

$$g(n) + c(n, a, n^*) + h(n^*) \geq g(n) + h(n).$$

**3. Conclusion**

Knowing that:

$$f(n) = g(n) + h(n)$$
$$f(n^*) = g(n) + c(n, a, n^*) + h(n^*)$$

we can substitute the terms of the previous inequality, that becomes:

$$f(n) \leq f(n^*).$$

Since the evaluation function $f(n^*)$ never decreases, we can conclude saying

that the heuristic function $h(n)$ is **consistent**, the A* algorithm will define **optimal** solutions for search graphs.

# Chapter 4

# Local search

Until now, we have saw a precise category of algorithms, known as **constructive algorithms**: the final solution is achieved by a sequence of actions. Anyway, when we are solving problems focusing on the goal state and we don't matter about the steps necessary to reach that goal, we are considering **local search algorithms**.

Local search algorithms start from an initial solution and iteratively try to improve it through **local moves**. Every time a local move is performed, it generates a **neighborhood**.

In many problems, the path to the goal is irrelevant. Let's consider an example about it.

**Example**

i.e. 4-queens problem.

The purpose of this problem is to put the four queens on the chessboard so that they do not attack each other. In this example, what really matters is the final configuration of queens, not the order in which they are added.

**Local search algorithms** operate using a single **current node** and generally move only to neighbors of that node. But, what is a neighbor?

**Definition**

The function

$$N : S \to 2^S$$

assigns to each state $s$ of the search space $S$, $s \in S$, a set of neighbors $N(s) \subseteq S$. $\mathbf{N(s)}$ is called the neighborhood of $s$.

Usually, to understand local search, the search space is considered as a **landscape**. A landscape has both **location**, defined by the current state, and **elevation**, defined by the value assigned by the **heuristic function**. The aim of local search is to find the lowest value, global minimum, or the peak value, global maximum, of the landscape.

## 4.1 Hill-climbing search

**Hill-climbing search** is a very basic local search algorithm. It is simply a loop that continually moves in the direction of the increasing value, choosing always the highest solution inside the neighborhood. It terminates when it reaches a **peak**, that might be a local or global maximum, and there isn't a higher value inside the set of neighbors of the current solution.

Despite its simplicity, this method often gets stuck for the following reasons:

- **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. So, given $s^*$ a local peak and $f$ an evaluation function, such that for any state $s$ that belongs to the neighborhood $N(s^*)$, we have:

$$f(s^*) \geq f(s) \; \forall s \in N(s^*).$$

- **Ridges**: a ridge is a sequence of local maxima, that are very difficult for hill-climbing to reach. When we sample such a function, the sampling points will not fall exactly on the ridge line and will float nearby.

- **Plateaux**: a plateau is flat area of the landscape, where the neighboring states have the same value of the current solution.

Moreover, hill-climbing search cannot be used for complex problems; first of all, it has a very local view and, in the other hand, forgets about the solution already visited. It's like:

*Climbing Everest in thick fog with amnesia.*

## 4.2 Simulated annealing search

A hill-climbing algorithm that never does *downhill* is definitely incomplete, because it will get stuck in the first local optimum reached. Therefore, it seems reasonable to combine hill-climbing search with a random walk in some way, like changing the rules while searching the goal state.

An algorithm that follows this idea is the **simulated annealing**. Simulated annealing search allows moves resulting in solutions of worse quality than the current one. Instead of picking the best move, it picks a random move; if the move improves the situation, it is always accepted.

## 4.3 Genetic algorithms

A **genetic algorithm** is a search algorithm in which successor states are generated by picking two parent states, instead of choosing one state from the neighborhood of the current solution. Before moving on, we define some common terms about genetic algorithms.

Genetic algorithms are inspired by the process of **evolution**, composed as follows:

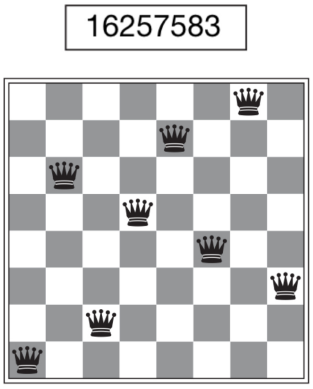
- **Inheritance**: offspring resemble their parents.
- **Adaptation**: organisms are suited to their habitats.
- **Natural selection**: new types of organisms emerge and those that fail to change are subject to extinction.

It's important to note that the specific terminology for genetic algorithms differs from that used for local search algorithms, it becomes:

- Each single solution defines a **genotype**.
- The set of solutions is called **population**.
- Chromosomes are made of units called **genes**.
- The domain of values of a gene is composed of **alleles**.

Having described the terminology of genetic algorithms, let's examine how they work. Genetic algorithms begin with a set of $k$ randomly generated states, representing the **population**. Each state, or **genotype**, is represented as a string over

a finite alphabet, most commonly, a string of 0s and 1s. For example, in the image below, which shows the 8-queens problem, each state can be specified as the positions of the 8 queens. Therefore, each genotype is an array of eight elements and the current position assumed by the queen will be inserted within each cell.

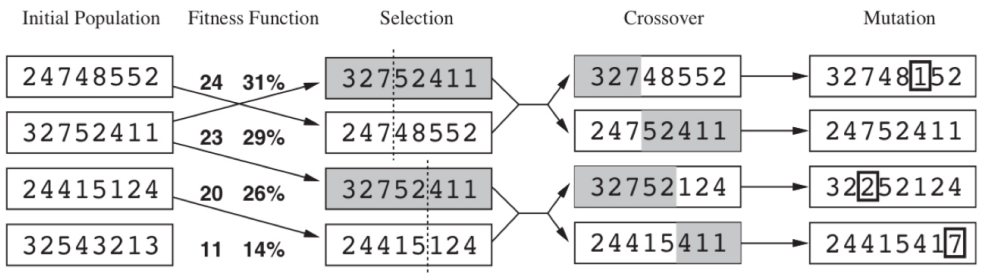



Here we are ordering the genotype by columns. It can also be expressed via rows.

Defined the representation of states, each of them is rated by a **fitness function**, such that the algorithm will take some genotypes for the production of the next generation of states. A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of non-attacking pairs of queens. In this case, the probability of being choosen for reproducing is directly proportional to the **fitness score**.

According to the fitness probabilities, pairs of genotypes are taken from the population and then combined for the reproduction. Hopefully, if the parent genotypes are great solutions, then the offspring generated might be a better solution than before, but it's not guaranteed. For each pair, a **crossover point** is chosen randomly from the position in the string. As in the example introduced, portions of the array representation are swapped.

Finally, the offspring generated is subject to a random **mutation**, one of the digit that composed the array representation is changed; this corresponds to choosing a queen randomly and moving it to a random cell. A visual summary on the main steps of genetic algorithms is shown below.




Replacing all the population each time a new generation is created, it determines a method that is computationally easier than any other algorithm seen so far; we substitute the old population with a new one, and that's it. However, this behavior can cause a major disadvantage: good solutions are not maintained in the new population. Generally, to avoid this, the best $n$ individuals from the new and old population are kept, hoping that the next generation can achieve a greater result.

Despite genetic algorithms being extremely simple and easy to associate with different types of purposes, they are influenced by two main constraints:

- It can be a little too simple according to the operations performed.
- We have to tune a lot of parameters; remember the representation of genotypes, results coming from mutation, permutation and selection or the crossover point.

## 4.4 Summary

| Preferable | **Local search** | **Genetic algorithm** |
|---|---|---|
| 1. | Neighborhood structures create a correlated search graph. | Solutions can be encoded as composition of states. |
| 2. | Computational cost of moves is low. | Computational cost of moves in local search is high. |
| 3. | Inventing moves is easy. | It is difficult to design effective neighborhood structures and a way to visit them. |

# Chapter 5

# Swarm intelligence

Concerning population based methods, now we explore another type of algorithms known as **Swarm Intelligence** algorithms.

**Definition**

**Swarm Intelligence**, briefly SI, is an artificial intelligence technique based around the study of collective behavior in decentralized and self-organized systems.

A simpler definition expresses Swarm Intelligence algorithms as a set of artificial intelligence methods that rely on **AI agents**, where each of them is totally autonomus and independent. So, any agent has its own goals and decision algorithm.

The entire swarm, in any kind of algorithm, has this nice feature: we can always examine an emerging behavior that comes from the interactions taken by any single agent[1]. Therefore, the key idea about Swarm Intelligence is to take some individuals, that are simple entities, and put them all together, thus defining a quite complex self-organization.

These agents can communicate either in a **direct** and **indirect** way. The last one way is performed by **stimmergy**, in other words, changing the environment they can communicate each other.

These methods are:

- **Adaptive**, because they adapt quickly to the changes in the environment.

[1]The **emerging behavior** does not come from the planning established by the algorithm!

- **Robust**, because if some agents stop working, the rest of the swarm still doing its job.

Each type of self-organization is based on three major ingredients, as follows:

- **Multiple interactions among agents**.
  As we said, the organization is composed by some simple agent, and each of them has its own goals and decision algotithm. Putting them together, we are defining a **multi-agent system**: from the taken interactions we can retrieve an emerging behavior.
- **Positive feedback**.
  Imitating, with positive feedback, successfull behaviors.
- **Negative feedback**.
  Suppose we have a *food source*, for us means a solution with a higher fitness score. If the food is already exhausted, so in the neighborhood of the current solution $N(s)$ nothing is left, we move to somewhere else.

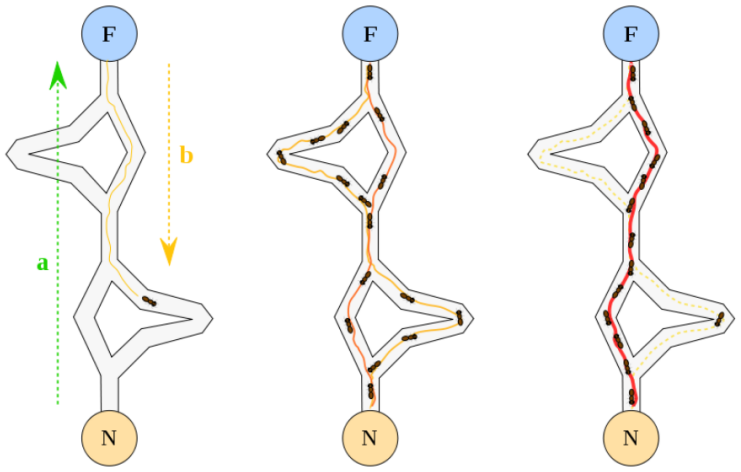
Swarm Intelligence algorithms are driven by natural phenomena, in particular, by groups of animals that interacting allow us to extrapolate an emerging behavior. We will consider three main algorithms, which are:

- Ant Colony Optimization.
- Particle Swarm Optimization.
- Artificial Bee Colony Algorithm.

## 5.1 Ant Colony Optimization

From observing the **ants**, such that real-world metaphor can be extrapolated, we discover that:

- Ants deposit **pheromone** trails while walking from the **nest** to the **food** and vice versa.
- Ants tend to choose the paths marked with **higher pheromone**, as described by the fitness function.
- A cooperative interaction leads to the **emergent behavior** to find the **shortest path**.




2

Found out that this algorithm is **not deterministic**, ant colony optimization uses a **probabilistic parametrized model**, briefly named as *pheromone model*, necessary to handle pheromone trails.

In a pratic way, ants traverse a graph, which is called **constructed graph**, composed by vertices and edges, as usual. Every time an ant follows a new part of the graph it is defining an **incremental solution**.

Now we want to examine how each ant decides. For better understanding we do this with a simple example.

**Example**

i.e. Given a constructed graph define the shortest path to the solution.

Starting from a nest node, each ant has two main informations to decide its path:

1. **Pheromone**, defined as: $\tau_{i,j}$. It represents the pheromone from node $i$ to node $j$, also coming from the other ants.
2. **Inverse of the length**, defined as: $\eta = \frac{1}{d_{i,j}}$. A heuristic value, a sort of parameter to define the distance[a].

Combining these informations, we can choose the next node in a **probabilistic** way. We said that this probabilistic choice depends on $\tau$ and $\eta$, and basically,

[2]The orange line defines a major concentration of pheromone. As we can see from the example, some ants are not following the same trail, so the algorithm is **not deterministic**.

we have a probability of moving from node $i$ to node $j$ equals to:

$$p_{i,j} = \begin{cases} \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_k [\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}} & \text{if j is consistent} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

$j$ consistency means that there must be an arc between node $i$ and node $j$; if it is not there then the value related will be 0.

But, how can we update the value associated with the **released pheromone**? First of all, suppose one ant that decides to follow this path

$$\langle i, j, n, q \rangle$$

which, in this particular case, allow us to reach the final solution. At each step done, this ant has left a certain amount of pheromone along the trail.

By the way, the nest consists of a large number of ants; consequently, several different paths are achieved. Every time a new node is reached, it is released a certain quantity of pheromone. The released pheromone is updated following this rule:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}$$

where:

- $\rho$ is an **evaporation coefficient**, used to decresease the previous pheromone value. For instance, suppose that the initial pheromone value between $i$ and $j$ is equal to 100, and the evaporation coefficient is equal to 10%; so, in the next round, the total value of pheromone will be 90, such that the 10% of the previous total value was released in the previous step.
- The summation ($\sum_{k=1}^{m} \Delta\tau_{i,j}^{k}$) defines the **overall pheromone** deposited on that portion of the trail, by the $m$ ants that traverse it. Therefore, we have to sum the pheromone left for each ant that decided to traverse that path. The amount of pheromone deposited is inversionally proportional to the length of the overall path done by ant $k$.

$$\Delta\tau_{i,j} = \begin{cases} \frac{1}{L_{i,j}} & \text{if ant k used arc (i, j)} \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

Ants that found the **shortest path** provide the major contribution to the pheromone released along the trail.

[a] The algorithm is inspired by the ants behavior is not an exact replica.

## 5.2 Artificial Honey Bee Colony Algorithm

**Artificial honey bee colony** is the second method that tell us something about Swarm Intelligence, even thought it is a little bit less used than **ant colony** algorithm. However, it represents a perfect example of Swarm Intelligence algorithm that uses *different types* of agents.

In this case, the food source is a point in a *n-dimensional space*, which means there is only one solution. In addition, the last statement reveals the algorithm as a population-based method, where we could have a population of solutions but only one can be the best one. There are three main types of bees, which are:

- **Employed bees**: each of them is associated with a specific nectar source.
- **Onlooker bees**: the bees in charge of intensifying the search around a nectar source. They choose the most **promising** food source already discovered by **employed bees**.
- **Scout bees**: scouts are instead the *diversification* bees, those that start a new search looking for the best solution.

The food or nectar source represents one of the solutions. Each solution has a fitness score equal to its size; larger solutions are considered more promising by agents. Like in local search, bee colony algorithm does not guarantee to converge to the best solution, so this algorithm is **not complete**.

The algoritm itself is not too complicated, it can be described in four fondamentals steps, as follows:

```
InitializationPhase()
repeat
EmployedBeePhase()
OnlookerBeePhase()
ScoutBeePhase()
Sol=BestSolutionSoFar
Until (Cycle=MaxCycleNum or MaxCPUtime)
```

1. **InitializationPhase()**.
Initially, a set of solutions are randomly selected from the food sources by the employed bees. Each solution $X_m$ is composed by $n$ variables $X_{mi}$, and each of them is subject to a lower and upper bound.

2. **EmployedBeePhase()**.
Each employed bee is associated with a food source and evaluates it through a fitness function. Basically, it performs a local search on that food source:

starting from the neighborhood $N(s)$, where $s$ is the current solution, it try to maximize or minimize over the landscape.

3. **OnlookerBee()**.
As we said, the most promising nectar sources attract more onlooker bees and they intensify the exploration phase already started from the employed bees. The probability that an onlooker bee join an exploration depends only by the fitness score related to that solution.

4. **ScoutPhase()**.
Once that the food source is exhausted, all the points that belong to the neighborhood $N(s)$ are evaluated, the employed bees become the scout ones and they choose the next point on the landscape randomly.

## 5.3 Particle Swarm Optimization

**Particle Swarm Optimization** is the last Swarm Intelligence algorithm discussed. *PSO*, acronym of Particle Swarm Optimization, is a technique inspired by the behavior of flocks of birds and, in particular, considers the interaction mechanisms between individuals that make up the swarm as the main knowledge.

From the interactions we can retrieve an unique emerging behavior: each entity of the swarm follows its neighborhood, staying inside the flock and trying to avoid collisions with his neighbors.

With these rules it is possible to describe and model the collective move of a flock with no common objective, but for simplicity we suppose that the flock is looking for a food source. Defined the food source, each individual of the swarm has two possibilities:

- **Individualistic choice**: move away from the group to reach the food.
- **Social choice**: stay inside the group.

If more than one entity moves toward the food, the other flock members do the same; this is the so called **positive feedback**, individuals imitate successfull behaviors changing direction toward promising areas.

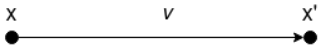
An important feature of this method is the concept of **promixity**, in which individuals are influenced by the actions of the other individuals closest to them. Therefore, given that individuals are part of multiple sub-groups, which are the neighborhood of those individuals, the spread of information through the flock is guaranteed. PSO is the unique algorithm seen so far that has a **common knowledge**.

Let's examine PSO algorithm in details.

### Particle Swarm Optimization algorithm

Summing up, PSO optimizes a problem by setting a population of solutions, and moves these solutions, also called **particles**, in the search space through simple mathematical formulas, using **vectors**.

Given an initial position $x$, a point in a N-dimensional space, it moves the flock to a new solution $x^*$, as shown in the image below.




Our main goal is to define the speed of movement; the movement of particles is done if and only if a better solution is discovered, rather than the current one. This behavior can be expressed as:

$$f : R^n \to R.$$

It takes a solution, vector-format like, and produces a fitness value. The algorithm terminates when it finds a solution $a$, such that:

$$f(a) \leq f(b)$$

for all $b$ in the search space[a].

So, given:

- $\mathbf{S}$, number of particles in the population.
- $\mathbf{x_i}$, current position of particle $i$.
- $\mathbf{v_i}$, current speed of particle $i$.
- $\mathbf{p_i}$, best solution found so far by the particle $i$.
- $\mathbf{g}$, best solution found so far by the entire swarm[b].

The initialization phase goes as follows:

1. Initialize the particle's position with a uniformly distributed random vector.
2. Initialize the particle's best known position to its initial position: $p_i \leftarrow x_i$.
3. If $f(p_i) < f(g)$ update the swarm's best known position: $g \leftarrow p_i$.
4. Initialize the particle's velocity.

Until a termination criterion is met, PSO repeats the subsequent actions for each particle of $S$:

1. Takes random numbers.
2. From those random numbers updates the particle's velocity.
3. From those random numbers updates the particle's position.
4. If $f(x_i) \leq f(p_i)$, updates the particle's best position: $p_i \leftarrow x_i$.
5. If $f(p_i) \leq f(g)$, update the swarm's best known position: $g \leftarrow p_i$.

At the end it returns the best found solution $\mathbf{g}$.

[a] It can be seen as a kind of minimization problem.

[b] $\mathbf{g}$ represents the shared knowledge between entities of the swarm. Remember, PSO is the unique algorithm seen so far that has a common knowledge.

# Chapter 6

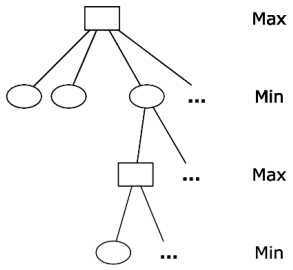
# Games

In the Swarm Intelligence part we have saw different types of algorithms, in which single entities are in a such way cooperating to reach the same goal state. However, multiagent environments has to account all the moves done by any single agent, either they are cooperating or competing.

In this chapter we cover **competitive environments**, also called **games**, in which the agents' goals are in conflict. Usually, in AI applications are considered games that have the following properties:

- **Two-player games**.
  Games designed only for two players, in which they take turns and have complementary objective functions.
- **Perfect Knowledge**.
  Perfect knowledge means games in which players have the same informations. For instance, the algorithms below are not applicable for card games, where dominates *information asymmetry*.

In this case, games are described as a part of *search strategies*, so any match development can be expressed as a search tree, in which the root is the starting position and the leaves are the final positions. Each node, that composed the tree, is a **state** of the game and edjes represents all the possible **actions** coming out from a state. In addition, below each leaf node is reported an **utility value**, that are simply integer values associated to **terminal** states.




The image above shows all the features defined previously. Before moving on, we focus on two more aspects of the illustrated **game tree**:

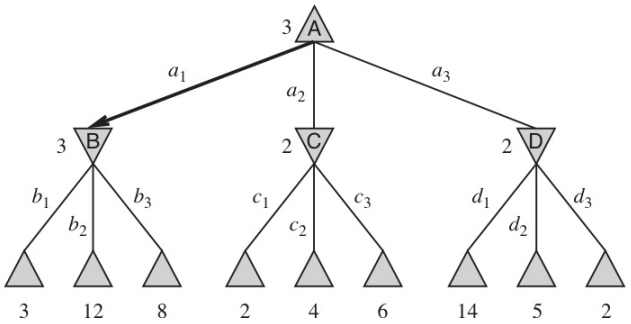
- The two players are called *MAX player* and *MIN player*, and MAX player always moves first.
- The **square nodes** are related to MAX player moves, instead the **circle nodes** define all the possible moves for the MIN player.

In a traditional search problem, the optimal solution is achieved by a sequence of steps, but in games this is not possible. The main difference between games and search strategies is the impossibility to decide the whole trail that bring us from the root node to the goal node; in games there are two players taking turns and selecting each time their own moves.

Let's see an example for a better understanding.

**Example**

i.e. General game tree.




The structure of the game tree is quite easy to understand. As we said, the first move is given to the MAX player. MAX player will choose the node with the highest utility value possible. After that, MIN player moves and chooses the lowest value.

Conceptually, we are doing a **straight-forward propagation**, once a choice is selected we propagate it until a final state is reached. But, it sounds weird:

*How does any player involved know about the next move taken by its opponent?*

Often, in game trees is performed the so called **backward propagation** method. Generated the data structure, we start from the leaf nodes, or the terminal states, and then we carry up the initial decision to the root node. Let's apply this observation to the previously game tree.

The utility values applied to the final states are:

$\langle 3, 12, 8 \rangle$ for the $1^{st}$ branch
$\langle 2, 4, 6 \rangle$ for the $2^{nd}$ branch
$\langle 3, 2, 2 \rangle$ for the $3^{rd}$ branch

MIN player chooses for each branch the lowest value, so it carries up to the above nodes: $\langle 3, 2, 2 \rangle$. Now it's MAX turn, the utility values are:

$\langle 3, 2, 2 \rangle$

It takes the highest value, which is 3.

This example shows how the **Min-Max algorithm** works.

## 6.1 Min-Max Algorithm

**Min-Max algorithm** computes the decision from the current state. It uses a simple recursive computation of the **minmax values** of each successor state, implementing directly the following definition.

**Min-Max Algorithm**

Given $s$ the current state such that $s \in S$, where $S$ is the set containing all the possible game states. The minmax choice is:

$$minmax(s) = \begin{cases} Utility(s) & \text{If game is over} \\ max_{a \in Actions(s)} & \text{If MAX moves} \\ min_{a \in Actions(s)} & \text{If MIN moves} \end{cases}$$

The algorithm computes a complete *DFS* exploration of the game tree, so it doesn't guarantee **completeness**, even though is an **optimal** solution, as we already know. Given $m$ the maximum depth and $b$ the branching factor, the time complexity is $O(b^m)$ and the space complexity is $O(bm)$.

For a real game, this complexity is totally impractical, but this method serves as

the main basis for more pratictal algorithms.

## 6.2 Min-Max Algorithm Revised

By the way, there's a revised version of the Min-Max algorithm, based on this assumption:

*Look forward few levels and assess the configuration of a non-terminal node.*

In practice we apply the Min-Max algorithm up to a certain depth. If we develop the whole tree, Min-Max method becomes very inefficient, it grows exponentially according to the depth of the tree.

The solution undertaken consists in an **evaluation funtion** for estimating the quality of a certain node, that can be expressed as:

$$E(n) = \begin{cases} -1 & \text{If MIN wins} \\ +1 & \text{If MAX wins} \\ 0 & \text{If they have the same probability} \end{cases}$$

Anyway, a huge problem persists: *how do we decide if to expand a node or not?* Well, such that Min-Max revised follows the same concept of *Depth-limited search*, we should expand the game tree until a certain depth $l$ is reached, defined in the starting phase of the computation.

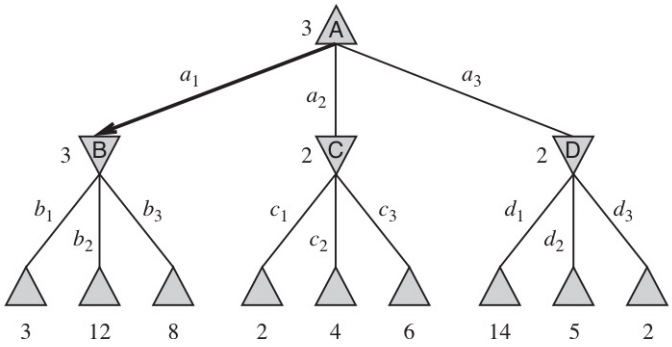
However, more tactically moves necessitate evaluatation to a **higher depth**, continuing the search until the queiscence condition is achieved - the point where the evaluatation function $E(n)$ values stabilize. Additionally, the resulting game tree generates a large number of **useless moves** which, in a real scenario, would never be considered.

## 6.3 Alpha-Beta Pruning Algorithm

The exponential growth in the number of game states appears to be the biggest problem with the Min-Max algorithm. We have discovered that computers simply play all the possible matches, evaluate leaf nodes and then propagate back the initial choice. Additionally, Min-Max revised algorithm shows us that the resulting game tree generates soo many useless moves, even though they will never occur in a real scenario.

Therefore, the following idea to avoid this bad behavior may be to reduce the size of the tree. The best-known technique is **Alpha-Beta Pruning**.

Consider the two players game used before.




The outcome received by the figure is that we can identify the minmax choice without evaluate the other two final states of the node $C$. Another way to express what happens can be described through a semplification of Min-Max formula.

Let the two unevaluated states of node $C$ have values x and y. Then the utility value of the root node is given by:

$$minmax(root) = max(min(3, 12, 8), min(2, x, y), min(15, 5, 2))$$
$$minmax(root) = max(3, 2, 2)$$
$$minmax(root) = 3.$$

In other words, it doesn't matter which are the real values of the unvaluated expressions, MAX will choose always the utility value 3, such that is the highest number between all the possible choices. Starting from some definitions we can generalize the algorithm behavior.

**Definition**

The values in **MAX** nodes are **Alpha-values**.

**Definition**

The values in **MIN** nodes are **Beta-values**.

Given these assumptions, we summarize Alpha-Beta Pruning method, splitting the cases in base of which player is currently moving.

**Definition**

The game tree is pruned as follows:

- If a **Alpha-value** is greater than a **Beta-value** of a descending node: stop the generation of children of the descending node.
- If a **Beta-value** is smaller than a **Alpha-value** of a descending node: stop the generation of children of the descending node.

The effectiveness of Alpha-Beta Pruning is highly dependent on the order in which the states are examined. For instance, we could not prune any successors of node $D$ at all because the worst successors were generated first. If the third successor of $D$ had been generated first, we would have been able to prune the other two. If this can be done[1], then it turns out that Alpha-Beta requires $O(b^{m/2})$ to pick the best move, instead of $O(b^m)$ for Min-Max. Anyway, in the average case with random distribution of the node values, the number of nodes examined becomes about $O(b^{3/4})$.

[1] When the best nodes are evaluated first.

# Chapter 7

# Automated Planning

**Automated planning** is an important problem solving activity, which consists in summerizing a sequence of actions performed by an agent that leads from an initial state to a given target state.

For humans this type of tasks are very easy to perform, but for computers is one of the most difficult. Additionally, we need a set of symbols that allow us to describe every single action taken, that cannot be **natural language**. Generally, natural language should be avoided as it is the primary cause of ambiguity.

The foundations about automated planning are given by:

1. Initial state.
2. State to achieve, also called goal state.
3. Set of actions that the agent can perform. These actions should not be macro-actions or actions that our agent cannot carry out.

Despite the goal consists to reach a target state, we focus on the steps necessary that could bring us from the initial state to the final one. Before moving on, we have to define some basic concepts about automated planning.

**Definition**

An **automated planner** is an intelligent agent that operates in a certain domain, described by:

1. A representation of the initial state.
2. A representation of the goal state.
3. A formal description of the executable actions.

**Definition**

**Domain theory** defines a formal description about all the executable actions.

**Definition**

Each action is identified by a name[a] and formalized through **pre-conditions** and **post-conditions**, which are expressed as follows:

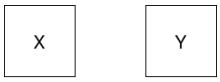
- Pre-conditions are the conditions which **must hold** to ensure that the action can be executed.
- Post-conditions represent the **effects** of the action in the environment.

[a]So many times we can perform the same actions with different variables, therefore can be useful call them with a **class name**.

Let's consider a trivial example just to fix these concepts.

**Example**

i.e. Block world.




The executable actions are:

1. **Stack(x,y)**
   Pre-conditions: holding(x), clear(y)
   Post-conditions: handempty, clear(x), on(x,y)
2. **Unstack(x,y)**
   Pre-conditions: handempty, clear(x), on(x,y)
   Post-conditions: holding(x), clear(y)
3. **Pick-up(x)**[a]
   Pre-conditions: handempty, ontable(x), clear(x)
   Post-conditions: holding(x)
4. **Put-down(x)**
   Pre-conditions: holding(x)
   Post-conditions: ontable(x), clear(x), handempty

From a logic point of view, all the inner functions are **predicates**, by these properties we can describe all the possible situations. As the example shows, predicates are generally used for pre-conditions and post-conditions of the actions.

[a]Different from **Stack**, **Pick-up** grabs a block on the table that it is not on any other

block.

When an action is performed all the pre-conditions become false, all the post-conditions become true and all the untouched conditions should be unchanged. In logic this is called **frame problem**; everything that is untouched by an action should be unchanged, moved to the next state.

Our goal is to determine a resolution process, a sequence of actions that leads us to the final state. This is achieved by an intelligent agent that implements an algorithm. As usual, the embedding algorithm has some features, as:

- **Completeness**.
  A planner or an agent is **complete** if it always finds a plan when it exists.
- **Correctness**.
  A planner or an agent is **correct** when the solution found leads from the initial state to the goal.

In addition, it's possible to divide planners according to the type of strategies used for the planning, which are:

- **Offline planners**.
  Offline planners, also called **generative planners**, work on representations of the world, they draw up plans and then computed them.
- **Online planners**.
  Online planners, also named **reactive planners**, work in the world, doing changes in the current environment.

The main difference between these strategies is that plans generated by generative agents are purely **abstract**; they do not change the current environment. This characteristic means that the plan design process can be directly seen as a **search process**; we may find out a list of actions or a sequence of steps that allow us to reach the final goal[1].

The **execution** phase involves the implementation of the plan. This phase is typically characterized by two main attributes:

- **Irreversibility**.
  Once a state transition is made, it is generally not backtrackable, making the execution non-reversible.
- **Non-deterministic**.
  Actions are performed under real-world uncertainty, meaning the outcome may

[1] They are, in a such way, both **constructive algorithms**.

not be entirely predictable and unexpected behaviors might occur.

We will look at the following planning strategies:

- Planning as search.
- Linear planning.
- Deductive planning.
- Nonlinear planning - Partial Order Planning (POP).
- Hierarchical planning.
- Conditional planning.
- Graph-based planning.

## 7.1 Planning as Search

Planning is indeed a **search activity**. We already know which are uninformed and informed search strategies, how to represent a search tree and when we have to stop the search activity; the translation might be very easy.

Simply game states should be transformed in tree nodes, from each node a certain number of outgoing arcs represent all the executable actions and, finally, the search activity will be stop when we reach a node that contains our goal state.

Why don't we use logic for solving planning problems?

## 7.2 Linear planning

## 7.3 Deductive planning

**Deductive planning** employs logic to represent states, goals and actions, and consequenctly generates a plan that resembles the proof of a theorem. To solve planning problems using logic formulas, we just need to add a state in which properties or predicates are true. It is no longer called **predicate**, but instead is called **fluent**.

The conceptual difference is that a **fluent's truth** is verified only within the context of the current state, whereas a traditional **predicate** typically represents a property that is independent of the specific state.

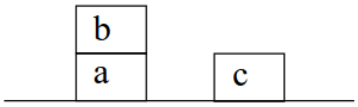
Deductive planning employs two main logic formulations, named:

- **Green formulation**
- **Kowalsky formulation**

We will start by examing the Green logic formulation.

**Example**

i.e. Block world.




Instead of writing

$$on(a, b), ontable(c)$$

we can express the same predicates adding a new term

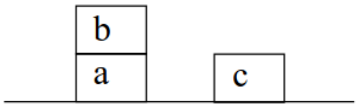
$$on(a, b, S), ontable(c, S)$$

where $S$ symbolises a **game state**.

This example introduces the **situation** entity. A situation is a state in which something is true. Therefore, the logic fluent $on(b, a, S)$ describes a state $S$ where the block $b$ is above the block $a$. In addition, actions can be seen as **clauses**.

**Example**

i.e. Block world.




The action $putontable(b)$ is defined as:

$$on(b, a, S), clear(b, S) \rightarrow$$
$$ontable(b, do(putontable(b), S)), clear(a, do(putontable(b), S))$$

where:

$1^{st}$ $on(b, a, S), clear(a, S)$ are the pre-conditions.
$2^{nd}$ $do(putontable(b), S)$[a] is the new state achieved by the current action.
$3^{th}$ $ontable(b, do(putontable(b), S)), clear(a, do(putontable(b), S))$ are the post-conditions.

[a] **putontable(...)** is the union of the previous **stack(...)** and **putdown(...)** predicates.

Specifically, the last instance is a **First Order Logic** clause, which we can formalize as a **general clause**:

*disjunction of negative and positive literals.*

We remember the basic rule of any given literals:

$$A \to B \text{ can be converted in } \neg A \vee B$$

In Green formulation, similar to the Kowalsky formulation, we have a simple mechanism to proof the plan, which is based on the **resolution process**. To better understand this concept, let's consider an example.

**Example**

i.e. Resolution process of the fluent

$$on(a, b, S_1).$$

So, we are asking if it exists a state $\mathbf{S_1}$ such that the block a is over the block b. Starting from an initial state $S$, we use the resolution process to formalize if the final goal is reachable.

$1^{st}$ step: define the domain theory, formal descriptions about all the executable actions.

A.1 $on(a, d, s0)$
A.2 $on(b, e, s0)$
A.3 $on(c, f, s0)$
A.4 $clear(a, s0)$
A.5 $clear(b, s0)$
A.6 $clear(c, s0)$
A.7 $clear(g, s0)$
A.8 $diff(a, b)$
A.9 $diff(a, c)$
A.10 $diff(a, d)$
A.11 ...

Note that $diff(\dots)$ are considered predicates, meaning their truth values are independent from the current state.

The required action $move(x, y, z)$ is expressed in general clause as:

$$on(x, y, S), clear(y, S), clear(z, S), diff(x, z) \to$$
$$clear(y, do(move(x, y, z), S)), on(x, z, do(move(x, y, z), S))$$

and then we can extract the following **effect axioms**:

A.12 $\neg on(x, y, S) \vee \neg clear(y, S) \vee \neg clear(z, S) \vee \neg diff(x, y)$
$\vee on(x, z, do(move(x, y, z), S))$
A.13 $\neg on(x, y, S) \vee \neg clear(y, S) \vee \neg clear(z, S) \vee \neg diff(x, y)$
$\vee clear(y, do(move(x, y, z), S))$

$2^{nd}$ step: negate the goal state, as required by the resolution process.

$$\neg on(a, b, S_1)$$

For solving it we have to find out its **positive complementary**, that is given by the *A.12* effect axiom.

$$on(x, z, do(move(x, y, z), S))$$

$3^{rd}$ step: match the negative literal[a] $\neg on(a, b, S_1)$ by unifying it with positive literal from the domain theory. Replacing the negative literal $\neg on(a, b, S_1)$ in the effect axiom $A.12$, from

$$\neg on(x, y, S) \vee \neg clear(y, S) \vee \neg clear(z, S) \vee \neg diff(x, y) \\ \vee on(x, z, do(move(x, y, z), S))$$

it becomes

$$\neg on(a, Y, S) \vee \neg clear(a, S) \vee \neg clear(b, S) \vee \neg diff(a, b) \\ \vee on(a, b, do(move(a, Y, b), S))$$

then, every negative literal is unified with the positive ones defined in the domain theory. Each successful unification results in the elimination of the literal.

- $\neg on(a, Y, S) \rightarrow A.1$
- $\neg clear(a, S) \rightarrow A.4$
- $\neg clear(b, S) \rightarrow A.5$
- $\neg diff(a, b) \rightarrow A.8$
- $\neg on(a, b, S_1) \rightarrow on(a, b, do(move(x, y, z), S))$

Once all the negated conditions are solved by the domain theory, we reach the **empty clause**. The empty clause formalizes the proof of the plan, meaning there is a state $S_1$ such that the block a is over the block b.

Anyway, if we cannot unify all the negative literals using our domain theory or general clauses, the proof of the plan is not verified: there is no state $S_1$ such that the block a is over the block b.

[a]Literals are the elements that build a clause. We remember that clauses are the disjunction of two or more literals.

How does the Green formulation treat the **frame problem**[2]? By Green formulation, to describe any action we have to specify all fluents and predicates that would not change after the execution phase. We need a frame axiom for each condition that is not changed by each action.

[2]The frame problem is to keep unchanged everything that has not been touched by the actions.

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

For simple problems, this is manageable. But for complex domains, the number of frame axioms to specify every non-change for every action becomes computationally intractable.

Let's proceed to the Kowalsky logic formulation. **Kowalsky formulation** is just another version of the First Order Logic for solving planning problems. It resolves the huge problem about the large amount of frame axioms required by Green formulation. It uses three main fluents, which are:

$1^{st}$ $\mathbf{holds(rel, S/A)}$.
This fluent describes all the properties *rel* that are true in a given state $S$ or made true by the execution of an action $A$.

$2^{nd}$ $\mathbf{poss(S)}$.
The predicate $poss(s)$ indicates if a state $S$ is possible, or in other words if it is *reachable*.

$3^{th}$ $\mathbf{pact(A, S)}$.
It defines if it is possible to execute an action $A$ in a state $S$. In logical terms, the pre-conditions of the action $A$ must be true in the state $S$.

If a state $S$ is possible and the pre-conditions of $A$ are satisfied, then it is possible also the following state produced by $S$ and action $A$. The general logic formulae of this assumption is:

$$poss(S) \wedge pact(A, S) \rightarrow poss(do(A, S))$$

where $do(A, S)$ produce the following state $S'$.

The power of this formulation is given by its **compactness**, we have to determine only one frame axiom per action. Taking into account the previous example, we would have:

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

The logic assertion formalizes that every relation $V$ different from $clear(Z)$ and $on(X, Y)$ are true, even after the execution of the action $move(X, Y, Z)$.

**Example**

i.e. Resolution process of the fluent

$$:\text{-}\ poss(S), holds(on(a,b),S), holds(on(b,g),S).$$

We are asking if it is possible a state $S$ such that the block a is over the block b and the block b is over the block g. We consider the action $move(x,y,z)$, which is a generalization of the simpler predicates: **stack(...)** and **unstack(...)**. The Kowalsky formulation requires a different approach than Green formulation. Therefore, it is often useful to take into account all the pre-conditions and post-conditions of the action being considered.

A particular data structure is generally used, consisting of three different lists, which are: *add list*, *delete list* and *frame list*. The add list contains all the post-conditions, the delete list contains all the pre-conditions and the frame list contains all the relations that remain true after the execution of the action. For the action $move(x,y,z)$, the structure would be:

$move(x,y,z)$
Pre-conditions:
$on(x,y)$
$clear(x)$
$clear(y)$
Post-conditions, also called effects:
$clear(y)$
$on(x,z)$

$1^{st}$ step: as before, define the initial state in First Order Logic.

- $poss(S_0)$
- $holds(on(c,f),S_0)$
- $holds(on(a,d),S_0)$
- $holds(on(b,e),S_0)$
- $holds(clear(g),S_0)$
- $holds(clear(f),S_0)$
- $holds(clear(d),S_0)$
- $holds(clear(e),S_0)$

$2^{nd}$ step: define the effect of the action, in other words the fluents that hold after the execution phase.

$$holds(clear(y), do(move(x,y,z),S))$$
$$holds(on(x,y), do(move(x,y,z),S))$$

$3^{rd}$ step: define the clause about the pre-conditions, determing which are the relations removed by the execution phase.

$$pact(move(x, y, z), S) \text{ :-}$$
$$holds(on(x, y), S) \wedge holds(clear(X), S) \wedge holds(clear(y), S)$$

$4^{th}$ step: define the frame conditions, describing the fluents that remain unchanged after the action's execution.

$$holds(V, do(move(x, y, z), S)) \text{ :-}$$
$$holds(V, S) \wedge \textit{diff}(V, on(x, y)) \wedge \textit{diff}(V, clear(x)) \wedge \textit{diff}(V, clear(y))$$

$5^{th}$ step: define the state **reachability** axioms.

$$poss(S_0)$$
$$poss(do(move(x, y, z), S)) \text{ :- } poss(S) \wedge pact(move(x, y, z), S)$$

## 7.4 STRIPS

So far we have seen general purpose planning approaches. Since planning is very complicated, why don't we use specific algorithms for solving them? One possibility is given by **STRIPS linear planner**.

STRIPS stands for *Stanford Research Institute Problem Solver*, it is a linear planner that uses **backward search** to develop a plan. Its main features are:

- **State representation**.
  Any state is defined by a set of fluents that are true in that state.
- **Goal representation**.
  The goal is a set of fluents that hold in the goal state.
- **Action representation**.
  Each action is defined by three different lists, which are:
  * **Pre-conditions**: containing fluents that should be true for applying the move.
  * **Delete**: list of fluents that become false after the move.
  * **Add**: composed of fluents that become true after the move.

Everything which is not in the **Add** and **Delete list** is unchanged after the execution of the move. Sometimes the Add and Delete list are glued in an **Effect list** with positive and negative axioms.

As we already said, STRIPS is based on backward search and it relies on two data structures:

- **Goal stack**.
  A FIFO queue, so First In First Out data structure, initialized with the goals to be achieved and proceeding backward.

- **Current state**.
  A logic description about the initial state. Unlike the goal stack, it proceeds forward until the goal state is reached.

**Example**

i.e. Resolution of the clause

$$on(c, b) \wedge on(a, c).$$

- $1^{st}$ step: describe the **current state** and initialize the **goal stack**.

| **State** | **Goal stack** |
|---|---|
| $clear(b), clear(c)$ | $on(c, b) \wedge on(a, c)$ |
| $on(c, a), ontable(a)$ | |
| $ontable(b), handempty$ | |

- $2^{nd}$ step: divide the goal in subgoals and select one of them from the goal stack.

| **State** | **Goal stack** |
|---|---|
| $clear(b), clear(c)$ | $\mathbf{on(c, b)}$ |
| $on(c, a), ontable(a)$ | $on(a, c)$ |
| $ontable(b), handempty$ | $on(c, b) \wedge on(a, c)$ |

- $3^{th}$ step: check if the selected subgoal is already satisfied by the current state. If it is, delete the subgoal from the goal stack, otherwise perform some actions such that the subgoal can be satisfy.

| **State** | **Goal stack** |
|---|---|
| $clear(b), clear(c)$ | $holding(c) \wedge clear(b)$ |
| $on(c, a), ontable(a)$ | $\mathbf{stack(c, b)}$ |
| $ontable(b), handempty$ | $on(a, c)$ ... |

  None of the $\mathbf{stack(c, b)}$ preconditions are satisfied by the current state, they become the new subgoals to achieve.

| **State** | **Goal stack** |
|---|---|
| $clear(b), clear(c)$ | $clear(b)$ |
| $on(c, a), ontable(a)$ | $holding(c)$ |
| $ontable(b), handempty$ | $holding(c) \wedge clear(b)$ ... |

  The current state tell us that the **block c** is above the **block a**, we have to execute the **unstack(...)** action.

| **State** | **Goal stack** |
|---|---|
| $clear(b), clear(c)$ | $clear(c) \wedge on(c, Y) \wedge handempty$ |
| $on(c, a), ontable(a)$ | $\mathbf{unstack(c, Y)}$ |
| $ontable(b), handempty$ | $clear(b)$ ... |

As we can see, all the preconditions about the **unstack**$(\mathbf{c}, \mathbf{a})$ action are alredy satisfied. Remove them from the goal stack and execute the action.

| **State** | **Goal stack** |
|---|---|
| $clear(b), clear(c)$ | **unstack**$(\mathbf{c}, \mathbf{a})$ |
| $on(c, a), ontable(a)$ | $clear(b)$ |
| $ontable(b), handempty$ | $holding(c)$ ... |

Update the current state after the execution phase, report all the effect of the specific action inside the current state.

| **State** | **Goal stack** |
|---|---|
| $holding(c)$ | $clear(b)$ |
| $clear(a), clear(b)$ | $holding(c)$ |
| $ontable(a), ontable(b)$ | $holding(c) \wedge clear(b)$ ... |

The last step allow us to pop out some subgoals from the stack. Every time we have to check a goal conjuction any subgoals that composed it must be verified. Now we execute the action **stack**$(\mathbf{c}, \mathbf{b})$, deleting the pre-conditions and adding the post-conditions inside the current state.

| **State** | **Goal stack** |
|---|---|
| $holding(c)$ | **stack**$(\mathbf{c}, \mathbf{b})$ |
| $clear(a), clear(b)$ | $on(a, c)$ |
| $ontable(a), ontable(b)$ | $on(c, b) \wedge on(a, c)$ |

| **State** | **Goal stack** |
|---|---|
| $clear(a), clear(c)$ | $on(a, c)$ |
| $ontable(a), ontable(b)$ | $on(c, b) \wedge on(a, c)$ |
| $on(c, b), handempty$ | |

Looking at the current state the predicate **on**$(\mathbf{a}, \mathbf{c})$ is not satisfied, we have to execute the action **stack**$(\mathbf{a}, \mathbf{c})$.

| **State** | **Goal stack** |
|---|---|
| $clear(a), clear(c)$ | **stack**$(\mathbf{a}, \mathbf{c})$ |
| $ontable(a), ontable(b)$ | $on(c, b) \wedge on(a, c)$ |
| $on(c, b), handempty$ | |

| **State** | **Goal stack** |
|---|---|
| $clear(a), clear(c)$ | $holding(a) \wedge clear(c)$ |
| $ontable(a), ontable(b)$ | **stack**$(\mathbf{a}, \mathbf{c})$ |
| $on(c, b), handempty$ | $on(c, b) \wedge on(a, c)$ |

The **holding**$(\mathbf{a})$ predicate is not verified, **pickup(...)** allow us to grab any block from the table.

| State | Goal stack |
|---|---|
| $clear(a), clear(c)$ | **pickup**(**a**) |
| $ontable(a), ontable(b)$ | $holding(a) \wedge clear(c)$ |
| $on(c,b), handempty$ | **stack**(**a**, **c**) |

| State | Goal stack |
|---|---|
| $clear(a), clear(c)$ | $ontable(a) \wedge clear(a) \wedge handempty$ |
| $ontable(a), ontable(b)$ | **pickup**(**a**) |
| $on(c,b), handempty$ | $holding(a) \wedge clear(c)$ |

All the preconditions of the action **pickup**(**a**) are already verified by the current state, move to the execution phase. Finally, the resulting data structures allow us to perform the last action **stack**(**a**, **c**).

| State | Goal stack |
|---|---|
| $clear(c)$ | $holding(a) \wedge clear(c)$ |
| $ontable(b)$ | **stack**(**a**, **c**) |
| $on(c,b), holding(a)$ | $on(b,c) \wedge on(a,c)$ |

| State | Goal stack |
|---|---|
| $clear(a)$ | $on(b,c) \wedge on(a,c)$ |
| $ontable(b)$ | |
| $on(c,b), on(a,c)$ | |

- $4^{th}$ step: once the goal stack is empty, stop the running process. It means that we have reached the goal state.

As we saw in the example, STRIPS solves one goal at the time and then moves on to the next one. In any case, this approach is influenced by one main problem: if the chosen order of the goal stack is wrong, a necessary action to reach the next goal may destroy the results of the previous goal.

**Example**

i.e. Resolution of the clause

$$on(c,b) \wedge on(a,c).$$

In the previous example we chose the following order:

| State | Goal stack |
|---|---|
| $clear(b), clear(c)$ | $on(c,b)$ |
| $on(c,a), ontable(a)$ | $on(a,c)$ |
| $ontable(b), handempty$ | $on(c,b) \wedge on(a,c)$ |

the goal $on(c,b)$ comes before the goal $on(a,c)$. If the planner chooses the

order where $on(a, c)$ comes before $on(b, c)$, the plan might proceeds as follows:
- A sequence of actions $\langle unstack(c, a), putdown(c), pickup(a), stack(a, b)\rangle$ is executed to satisfy the first goal, $on(a, c)$.
- The planner then attempts to satisfy the second goal, $on(c, b)$. Since the block $c$ is now under $a$, the required actions to move $c$ would inevitably destroy the first goal $on(a, c)$, which was just achieved.

Consequently, the planning process must perform backtracking, reinserting the failed subgoal into the goal stack and forcing the exploration of an entirely new planning strategy. This highlights the inefficiency of sequential planning when faced with goal dependencies.

## 7.5 Non-linear planner

To overcome the limitations of linear planners that explore the **state space**, the field shifts to **non-linear planners**, specifically **Partial Order Planning**, which instead operate within the **plan space**.

A non-linear planner treats a plan as an object that is gradually refined. It has two main charateristics, which are:

- **Search space**.
  Each node of the tree search represents a **partial plan**, and the operators are **plan refinement moves**.
- **Least commitment planning**.
  The planner avoids imposing decisions and orderings that are not strictly necessary. This strategy allow us to reduce the amount of backtracking actions required by linear planners.

A plan, defined by this type of planners, is represented as:

- A set of **actions**.
- A set of **casual links**.
- A set of **partial orderings** between the actions. For example, taking two actions $A$ and $B$, the order $A < B$, means that the action $A$ comes before the action $B$.

Each plan starts with two fake actions, described as follows:

$1^{st}$ **Start action**.
It is an action **without preconditions** and its effects match the initial state. Therefore, it relies only to the **initial state**.

$2^{nd}$ **Stop action**.
Unlike the start one, the stop action is an action **without postconditions** and its preconditions match the goal state. Therefore, it relies only to the **goal state**.
$3^{rd}$ **Ordering**.
From these two actions we can derive a general rule: *start action* $<$ *stop action*. In other words, in any plan designed by a non-linear planner the start action comes before the stop action.

The chore mechanisms for managing goal interaction in POP involves **casual links** and the resolution of **threats**.

- **Casual links**.
  A casual link, denoted $\langle S_i, C, S_j \rangle$, is a triple indicating that the action $S_i$ defines the condition $C$ required as precondition for the action $S_j$. In addition, a casual link stores the causal relations between actions: it traces why a given operator has been introduced in the plan.
- **Threats**.
  An action, $S_k$, is a **threat** to a casual link $\langle S_i, C, S_j \rangle$ if $S_k$ has an effect that negates the condition $C$, and no existing ordering constraint prevents $S_k$ from being executed between $S_i$ and $S_j$.

Threats must be resolved to ensure the established condition $C$ remains true when the actions $S_j$ is executed. The two primary strategies are:

- **Demotion**: imposing the ordering $S_k < S_i$, ensures the destructive action $S_k$ occurs **before** the casual link.
- **Promotion**: imposing the ordering $S_j < S_k$, ensures the destructive action $S_k$ occurs **after** the casual link.

**Example**

i.e. Purchasing schedule.

Given an initial state

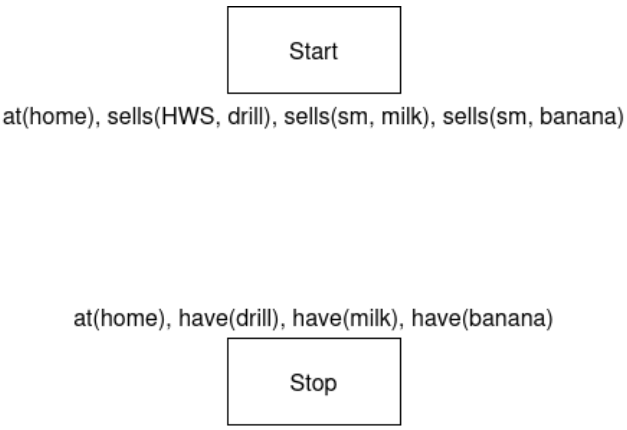
**Initial state**
$sells(sm, banana), sells(sm, milk)$
$sells(HWS, drill)$
$at(home)$

define a plan that bring us to the following goal state

**Goal state**
$have(milk), have(banana)$
$have(drill)$
$at(home)$

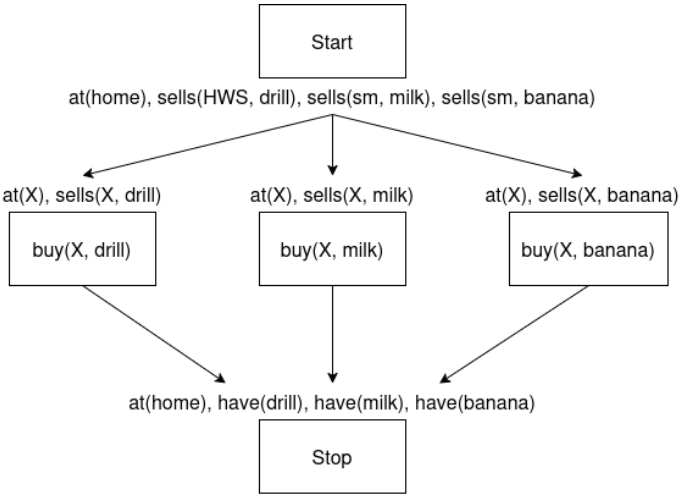
The executable actions are:

| **Go(X,Y)** | **Buy(S,Y)** |
|---|---|
| Precond: $at(X)$ | Precond: $at(S), sells(S,Y)$ |
| Effect: $at(Y), \neg at(X)^a$ | Effect: $have(Y)$ |




$1^{st}$ step:
- Select a precondition from the goal state to be fulfilled: $have(drill)$.
- Select an action that has the precondition $have(drill)$ as postcondition: $buy(S,Y)$.
- Represent the plan refinement:
  - Link the variable $Y$ with the term $drill$.
  - Impose ordering constraint: $start < buy(S, drill) < stop$.
  - Insert the casual link $\langle buy(S, drill), have(drill), stop \rangle$.

$2^{nd}$ step:
- Same process in the first step for $have(milk)$.
- Same process in the first step for $have(banana)$.

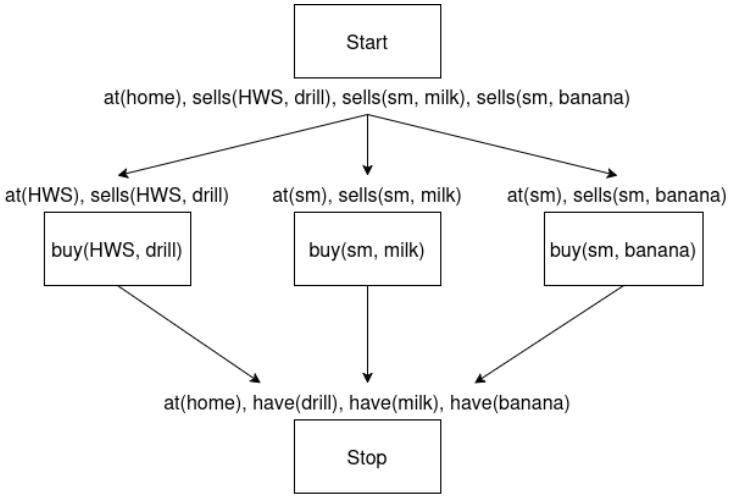



$3^{rd}$ step:

- Select a precondition from the $buy(S, drill)$ state to be fulfilled: $sells(S, drill)$.
- Select an action that has the precondition $sells(S, drill)$ as postcondition: $start$.
- Represent the plan refinement:
  - Link the variable $s$ with the term $HWS$.
  - Impose ordering constraint: $start < buy(HWS, drill) < stop$.
  - Insert the casual link $\langle start, sells(HWS, drill), buy(HWS, drill)\rangle$.

$4^{th}$ step:

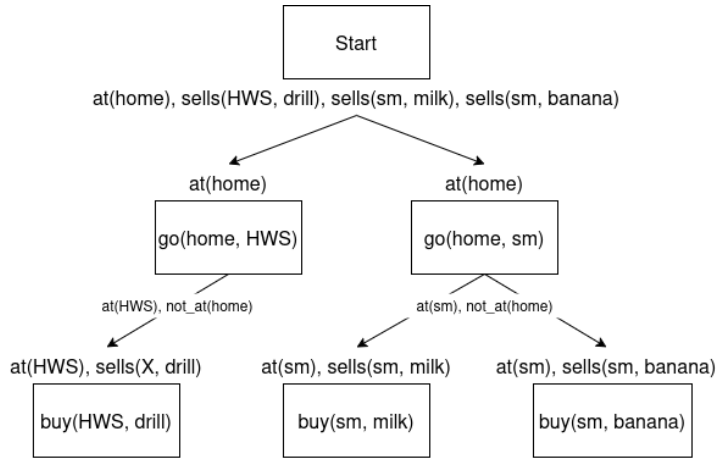
- Same process in the third step for $sells(S, milk)$.
- Same process in the third step for $sells(S, banana)$.



$5^{th}$ step:
- Select a precondition not yet satisfied: $at(HWS)$.
- Select an action that has the precondition $at(HWS)$ as postcondition: $go(X, HWS)$.
- Represent the plan refinement:
  - Impose ordering constraint: $start < go(X, HWS) < buy(HWS, drill)$.
  - Insert the casual link $\langle start, at(X), go(X, HWS)\rangle$.

$6^{th}$ step:
- Select a precondition not yet satisfied: $at(X)$.
- Select an action that has the precondition $at(X)$ as postcondition: $start$.
- Represent the plan refinement:
  - Link the variable $X$ to $home$.
  - Redefine the ordering constraint: $start < go(home, HWS) < buy(HWS, drill)$.
  - Redefine the casual link $\langle start, at(home), go(home, HWS)\rangle$.

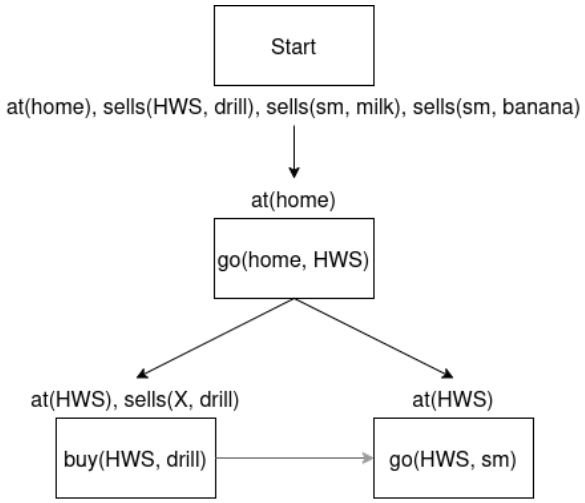
$7^{th}$ step:
- Same process in the fifth and sixth step for $at(sm)$.




After few steps, one huge problem comes out: the planner cannot perform at the same time the procedure $go(home, HWS)$ and $go(home, sm)$ from the start action. Demotion and promotion in this case are not enough to solve the problem, any constraint ordering does not work.

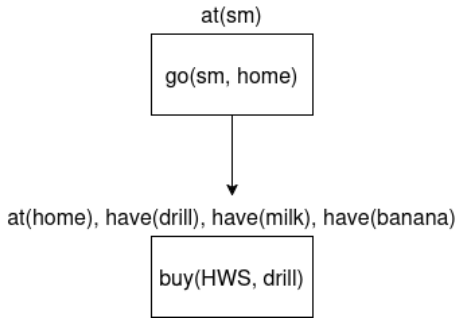
Instead of going back at *home* between stores, we take as advantage the **current position**. The planner decides to satisfy the precondition $at(sm)$ not from the **start action**, but **after** the procedure $go(home, HWS)$. At the end,

we have a constraint ordering as follows: $buy(HWS, drill) < go(HWS, sm)$.




$8^{st}$ step:

- Solve the preconditions $at(home)$ of the stop action.
- The only way to do this is to put the action $go(X, home)$ before the stop action.




[a]This effect list is represented as the conjuction of the add and delete list.

The eighth step is named the **white knight**, a refinement strategy which inserts a new operator in the plan between two conflict actions[3].

Let's examine an other example.

[3]We remember that two actions are in conflict when the effect of any procedure $S_i$ negate the preconditions of any procedure $S_j$.

**Example**

i.e. Sussman anomaly.

Given an initial state

**Initial state**
$ontable(a), ontable(b)$
$clear(b), clear(c)$
$on(c, a)$

define a plan that reaches the goal state

**Goal state**
$on(b, c) \wedge on(c, a)$

The executable actions are:

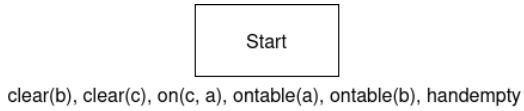
**Putdown(X)**
Precond: $holding(X)$
Effect: $handempty, clear(X), ontable(X), \neg$Precond

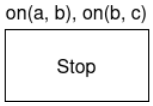
**Stack(X,Y)**
Precond: $holding(X), clear(Y)$
Effect: $handempty, on(X, Y), clear(X), \neg$Precond[a]

**Pickup(X)**
Precond: $handempty, clear(X), ontable(X)$
Effect: $holding(X), \neg$Precond

**Unstack(X,Y)**
Precond: $handempty, on(X, Y), clear(Y)$
Effect: $holding(X), clear(Y), \neg$Precond

Here we are trying to solve the **Sussman anomaly** using POP. As usual, we have to define the initial empty plan, containing two fake actions: the start action and the stop action.



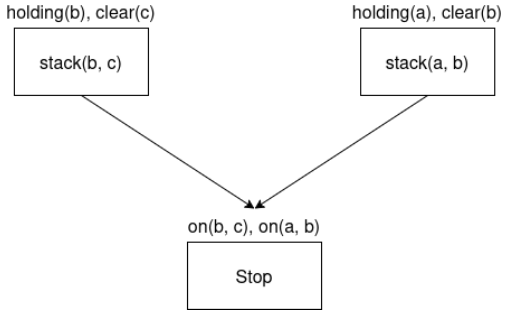




$1^{st}$ step:

- Select a precondition from the goal state to be fulfilled: $on(b, c)$.
- Select an action that has the precondition $on(b, c)$ as postcondition: $stack(b, c)$.
- Represent the plan refinement:
  - Impose ordering constraint: $start < stack(b, c) < stop$.
  - Insert the casual link $\langle stack(b, c), on(b, c), stop \rangle$.

$2^{nd}$ step:

- Same process in the first step for $on(c, a)$.




$3^{rd}$ step:

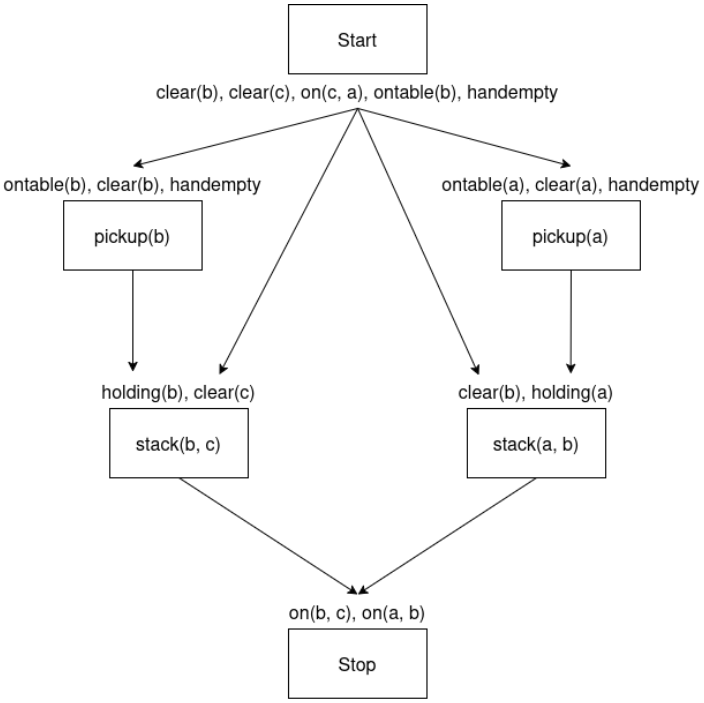
- Select a precondition from the $stack(c, b)$ state to be fulfilled: $holding(b)$.
- Select an action that has the precondition $holding(b)$ as postcondition: $pickup(b)$ or $unstack(b, Y)$.
- Since the initial state defines the block $b$ is on the table, we should perform the $pickup(X)$ action.
- Represent the plan refinement:
  - Impose ordering constraint: $start < pickup(b) < stack(b, c)$.
  - Insert the casual link $\langle pickup(b), holding(b), stack(b, c) \rangle$.

All the $pickup(b)$ preconditions are already satisfied by the start action.

Therefore, we have to insert the casual link from the start action to the pickup(...) action. In addition, as we can see, the precondition $clear(c)$ of the action $stack(b, c)$ is already satisfied by the knowledge base given by the start action.

$4^{th}$ step:

- Exactly the same procedure described in the previous step occurs for $stack(a, c)$ predicate.




By this initial formalization, some **threats** are already coming out, which are:

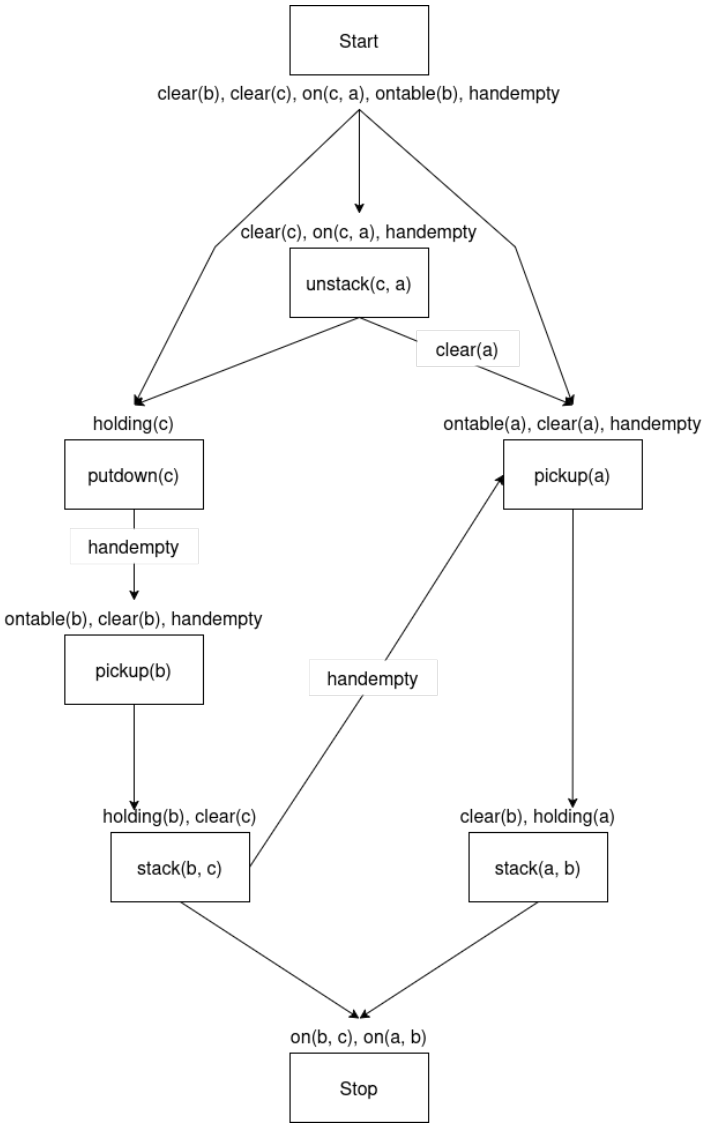
- Performing the action $pickup(b)$ negates the *handempty* precondition necessary for the predicate $pickup(a)$. This problem can be easily solved by imposing a **promotion** ordering constraint. The action $pickup(a)$ must be executed after the predicate $stack(b, c)$, in other words $stack(b, c) < pickup(a)$.
- If the first subgoal $on(b, c)$ is reached, then it's impossible to execute the action $stack(a, b)$ given this initial state. Any ordering constraint cannot solve this situation.

Following this ordering constraint, the planner must perform backtracking, forcing the exploration of a new planning strategy and, consequently, losing computational efficiency. The unique solution in this case is to think outside the main scheme, reasoning about how all the different actions can be linked

together.

The main intuition is: before performing $stack(b, c)$ action we have to **unstack**$^b$ the block $c$ over the block $a$. Holding the block $c$, the next move will put it down on the table. Finally, the *handempty* condition will be true, allowing us to execute all the **stack(...)** predicates. The final ordering scheme is shown below.

$$start < unstack(c, a) < putdown(c) < pickup(b) < ...$$
$$... < stack(b, c) < pickup(a) < stack(a, b) < stop$$



[a] To save space, we did not list every precondition that was denied by the action.

[b] Unstack action is a white knight strategy, we are inserting a new action inside the initial plan.

. . .