

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 be 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 states¹. Generally, the set of all leaf nodes available for expansion at any given point is called **frontier** or **fringe**. The process of expanding nodes 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 expanded.

¹Every time we expand the current node, new states 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.

²We 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 necessarily 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

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

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.

^aWe 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 order¹.

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

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

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 $l > d$.

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 iteratively 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 branching 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

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

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

³Iterative 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.

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

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

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

⁴Comparison 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 state¹.

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.

¹Until 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 example².

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, $h_{SLD}(Arad) = 366$. 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 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_{SLD} , 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.

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

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 algorithm³. 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

The first condition required for optimality is that the heuristic function $h(n)$ is a **feasible** heuristic. A feasible heuristic is one that *never overestimates* 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.

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

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.
- **G₂**: 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 **G₂** 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 $f(G_2)$, denying to A* algorithm to expand the node **G₂** 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.

$$\begin{aligned} f(G) &= g(G) + h(G) = g(G) + 0 = g(G) \\ f(G) &= g(G). \end{aligned}$$

2. Evaluation of the sub-optimal goal.

The evaluation function of **G₂** 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 **G₂**.

Knowing that **G₂** is the sub-optimal path, the total cost to reach the goal node via **G₂** 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 \mathbf{n} .

Given \mathbf{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 \mathbf{n} must be less or equal to the real cost of \mathbf{n} :

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

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

$$\begin{aligned} f(n) &\leq g(n) + h^*(n) = f(G) \\ f(n) &\leq f(G). \end{aligned}$$

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.