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 four non-informed search strategies:

- BFS.
- 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

[1]We 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

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