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

[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

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

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

[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.

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

[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 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.

[2]Remember 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 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

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.

[3]Basically, 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.
- $\mathbf{G_2}$: 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:

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

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

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.

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 $\mathbf{f}(\mathbf{n}^*)$ 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**

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

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:

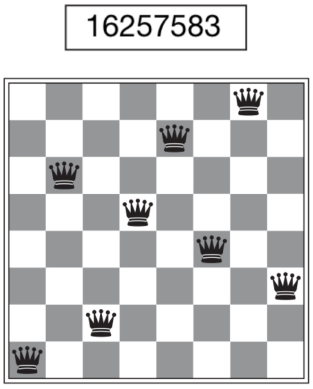
- **Inheritance**: offspring resemble their parents.
- **Adaptation**: organisms are suited to their habitats.
- **Natural selection**: new types of organisms emerge and those that fail to change are subject to extinction.

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.

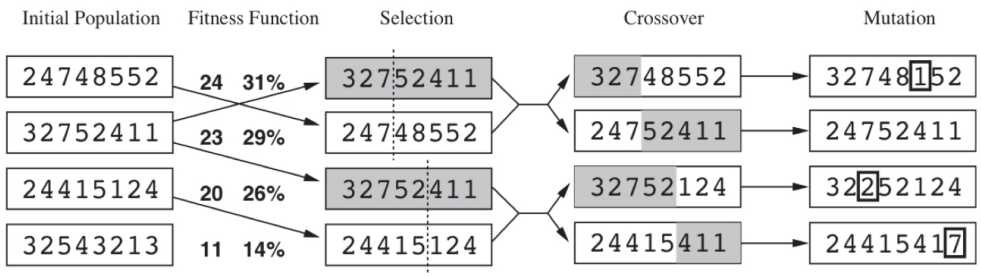


Here we are ordering the genotype by columns. It can also be expressed via rows.

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 soo simple, and put them all together, thus defining a quite complex self-organization.

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.

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

- **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.

We will consider three main algorithms, based on the last intuition defined, 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.

[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**.

Example

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

Starting from a root 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].

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