Automated Search

Daniel Gigliotti

1 Problem solving agents

Search algorithms are one of the most important areas of Artificial Intelligence.

Search algorithms are universal problem-solving techniques. They allow the AI system to efficiently find a solution to a problem among a large set of possibilities. These algorithms can be used for a variety of tasks, such as planning, decision-making, and problem-solving. They can also be used to optimize machine learning models by finding the best parameters or architecture.

Given that:

- **State space** is the set of possible solutions which a system may have;
- **Start state** is the state in which the agent is initially;
- **Goal state** is the state the agent needs to reach.

a problem could be viewed as moving in the state space from the initial state towards a goal state. Searching is a step by step procedure to solve this kind of problem.

A representation of the search problem is called **search tree**. The root of the search tree is the initial state, the nodes are the other states and the solution (goal state) is among them. We don't know where the solution is in the tree, so we need to search. The choice of the node being **expanded** represents the **search strategy**.

- a **state** is physical configuration;
- a **node** is a data structure part of the search tree and includes **parent, children, depth, path cost** $g(n)$.

states do not have parent, children and so on.

2 Search strategies

A strategy is defined by picking the order of node expansion.

2.1 Properties of search algorithms

In what terms one search algorithms is better than another one? To compare the efficiency of these algorithms we can use the following properties:

- **completeness**: a search algorithm is said to be complete if for any random input it guarantees to return a solution if at least one solution exists;
- **optimality**: a search algorithm is said to be optimal if it is guaranteed that the solution found is the best one (lowest path cost) among other;
- **Time complexity**: time complexity measures the time an algorithm takes to complete a task;
- **Space complexity**: space complexity measures the storage space required for an algorithm at any point during the search;

Finding the optimal solution entails proving optimality. This can be done by finding all solutions or by proving that no solution can have better cost than the one found already. In either case, at least one solution has to be found.

If there is no solution, neither an optimal nor a complete algorithm would find one of course.

$$optimality \ \rightarrow completeness \tag{1}$$

$$!\ complete \ \rightarrow !\ optimal \tag{2}$$

2.2 Types of search algorithms

Based on the search problem itself, we can classify the search strategies into:

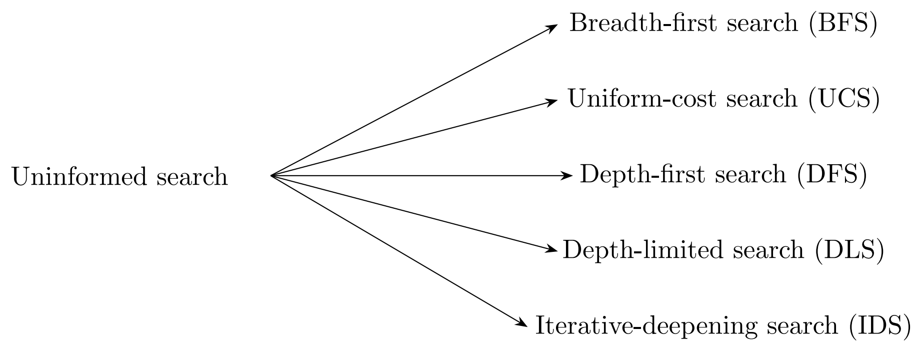
- **uninformed**: does not contain any domain knowledge such as closeness or the location of the goal; an uninformed search algorithm will examine each node of the three until it achieves the goal node;
- **informed**: problem information is available which can guide the search; they are based on the use of **heuristics**;

Heuristic functions estimate the cost of the cheapest path from the current state (node) to the goal. An heuristic does not guarantees to find the best solution but guarantees to find a good solution in reasonable time.

Informed search algorithms are generally more efficient than uninformed search algorithms because they can focus the search process on more promising areas of the search space. However, the quality of the solution found by an informed search algorithm depends on the accuracy of the heuristic function used.

3 Uninformed search

Algorithms have no additional information on the goal node other than the ones provided in the problem definition.



Uninformed search
Breadth-first search (BFS)
Uniform-cost search (UCS)
Depth-first search (DFS)
Depth-limited search (DLS)
Iterative-deepening search (IDS)

3.1 Breadth-first search (BFS)

The tree is explored looking first at the root node, then at all its neighbors, then at the next level neighbors and so on. This process continues until all the nodes in the tree have been explored.

The iterative implementation uses a **queue** (**FIFO** structure). It is complete and optimal: that means that it will find the shallower solution, it it exists, with the minimum amount of actions. It finds the optimal solution in $O(b^d)$, scanning the entire tree, and occupies $O(b^d)$ of memory.

$$frontier = queue$$

3.2 Uniform-cost search (UCS)

Uniform-cost search **expands the node with the lowest path cost** $g(n)$.

Uniform-cost search uses the lowest cumulative cost to find a path from the source to the destination. Nodes are expanded, starting from the root, according to the minimum cumulative cost $g(n)$. The search is implemented using a **priority queue** with $p(n) = g(n)$.

$$\textit{frontier} = \textit{priority queue}$$

It is complete (if each step cost $\geq \epsilon$ i.e. there are no negative costs) and optimal. It can be inefficient in situations where the cost of reaching a goal state is large, but there are also many lower-cost paths that lead to states that are not the goal state. In such cases, the algorithm may explore a large number of unnecessary states before finding the goal state, leading to a high time complexity. Additionally, if the cost function is not consistent, the algorithm may get stuck in an infinite loop.

When encountered a second time with a different cost, a node is also kept. We keep all the nodes we encounter if they are not in the visited list.

3.3 Depth-first search (DFS)

Depth-first search is a method for traversing a graph or tree data structure. The root node is visited first, the next one is chosen among its children by going deeper into the tree/graph. This process continues until a leaf node (a node with no children) is reached, then the process backtracks to the most recent node that has unvisited children. DFS is implemented using a **stack** data structure to keep track of the nodes to be visited next (**LIFO** structure). DFS is used for tasks such as topological sorting, solving mazes and puzzles, and searching for connected components in a graph. It is very efficient in all the applications in which we need to explore the space going as deep as we can. 121

$$frontier = stack$$

It is NOT complete since it fails in infinite-depth spaces or spaces with loops. We can easily keep track of visited nodes and avoid re-expanding them, so the algorithm can be extended to support spaces with loops, but the problems with infinite-depth spaces persist. It is NOT optimal since it could miss some shallower solution going deeper and deeper. It occupies $O(b * m)$ memory (only the current path) and finds a solution in the finite-depth space case in $O(b^m)$.

3.4 Depth-limited search (DLS)

This search is like the Depth-first search but there is a limit in the tree's depth. In this way, the search process cannot continue infinitely. If there is not a solution in the subtree considered, the algorithm will not be able to find any solution, and is therefore NOT complete. It is not optimal for the same reason. It finds a solution in $O(b^d)$ if present and occupies $O(b * l)$ memory (l being the depth limit).

3.5 Iterative-deepening search (IDS)

This search algorithms is nothing more than an iteration of DLS, increasing the limit at each step. It starts from $l = 1$ and increases it until the shallowest solution is found. It IS complete because it finds a solution acting in a similar way of BFS, but requiring significantly less memory. The complexity is $O(b^d)$ for time and $O(bd)$ for memory occupation.

3.6 Uniformed search comparison

Search algorithm	Completeness	Optimality	Time complexity	Space complexity
Breadth-first	yes (if b is finite)	yes (if cost = 1 for each step)	$O(b^d)$	$O(b^d)$
Uniform-cost	yes (if step cost $\geq \epsilon$)	yes	$O(b^{\lceil 1+C^*/\epsilon \rceil})$	$O(b^{\lceil C^*/\epsilon \rceil})$
Depth-first	no (if domain is depth-infinite)	no (does not find the lowest path solution)	$O(b^m)$	$O(bm)$
Depth-limited	no	no	$O(b^l)$	$O(bl)$
Iterative-deepening	yes	yes	$O(b^d)$	$O(bd)$

with:

- b: average branching factor;
- d: depth;
- l: limit on the depth;
- m: maximum depth;
- C^*: cost of the optimal solution;
- ϵ: least cost of a step;

4 Search direction

Search can be:

- **forward** (or data driven);
- **backward** (or goal driven);
- **bidirectional** (or mixed);

An alternative approach called bidirectional search simultaneously searches forward and backwards if the problem allows it, hoping that the two searches will meet midway. The motivation is that

$$b^{d/2} + b^{d/2} << b^d \tag{3}$$

We have a solution when the two frontiers collide. Completeness and optimality depends on the search strategies adopted.

5 Informed search

In general, in **tree-search** and **graph-search**, a **strategy** is a set of rules that defines the order in which nodes are selected for expansion and how the search progresses from one node to another.
Uninformed search algorithms use a static set of rules (e.g. expand the current node, then repeat for each of its child, for the Breadth-first search).

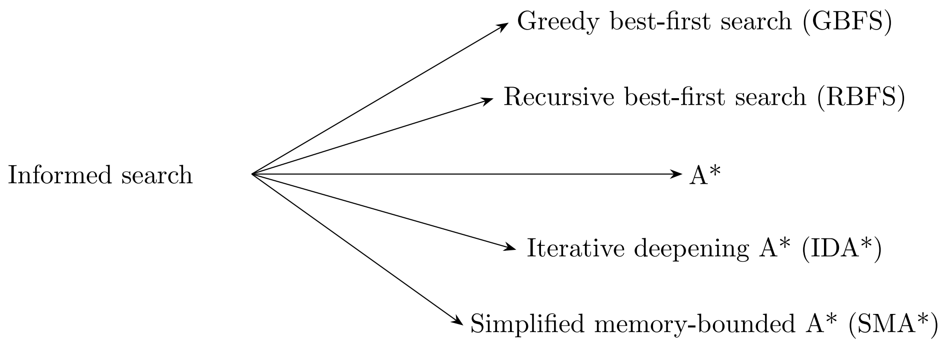
Some algorithms instead make use of problem-specific knowledge and are thus called **informed algorithms**. In this case we talk about **best-first search** instead of three-search/graph-search, and the set of rules that defines the order of expansion is a function of the the node itself, called **evaluation function**, $f(n)$. Evaluation function is a cost estimate: we expand the lowest cost node first.

The choice of $f(n)$ determines the search strategy. Often, for best-first search algorithms, f is defined in terms of a **heuristic function**, $h(n)$, which estimates the cost of the cheapest path from the current node to the goal. Heuristic functions are the most common form in which **additional knowledge** of the problem is passed to the search algorithm.

Best-first search algorithms constitute a large family of algorithms, with different evaluation functions. Some implementations of the Best-first search are **Greedy Best-first search** and **A***.

5.1 Building an heuristic

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem (relaxed problem: some of the constraints are removed). Problem relaxation is a good way to find a good heuristic.


Informed search
Greedy best-first search (GBFS)
Recursive best-first search (RBFS)
A*
Iterative deepening A* (IDA*)
Simplified memory-bounded A* (SMA*)

$$\begin{tikzcd}
\begin{array}{l}\text{GBFS}\\ \text{not complete, not optimal}\\ \text{time complexity } O(b^m)\\ \text{spatial complexity } O(b^m)\end{array} \arrow[d] & & \\
\begin{array}{l}\text{A*}\\ \text{complete, optimal}\\ \text{time complexity } O(b^m)\\ \text{spatial complexity } O(b^m)\end{array} \arrow[d] & & \\
\begin{array}{l}\text{IDA*}\\ \text{complete, optimal}\\ \text{time complexity } O(b^m)\\ \text{spatial complexity } O(bm)\end{array} \arrow[dr] & \text{VS} & \begin{array}{l}\text{RBFS (can backtrack)}\\ \text{complete, not optimal}\\ \text{time complexity ?}\\ \text{spatial complexity } O(b^m)\end{array} \arrow[dl] \\
 & \text{SMA*} &
\end{tikzcd}$$

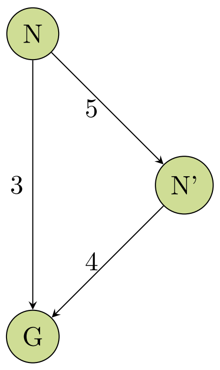
5.2 Heuristic function properties

- **Dominance**: a heuristic function $h_1(n)$ dominates another heuristic function $h_2(n)$ iff:
$$\forall n : h_1(n) \geq h_2(n) \tag{4}$$
The dominant heuristic is better for search.

- **Admissibility**: a heuristic function is admissible if it never overestimates the cost to reach the goal;

- **Consistency**: a heuristic function is consistent if the estimated cost to reach the goal from n is not greater than the cost to reach n', child of n, plus the estimated cost from n' to the goal:

$$h(n) \leq c(n, n') + h(n') \tag{5}$$



If a heuristic is not consistent, we can use the **pathmax**:

$$f(n') = max(g(n') + h(n')), f(n)) \tag{6}$$

Using pathmax, f is always **non decreasing**.

5.3 Combination of heuristics

What if we have several heuristics not dominating each other?

Let $h_1, ..., h_m$ be a collection of such heuristics, define:

$$h(n) = max(h_1(n), ..., h_m(n)) \tag{7}$$

h is admissible and dominates $h_1, ..., h_m$.

5.4 Effective Branching Factor

The effective branching factor (EBF) is a measure that quantifies the average branching factor of a search tree by taking into account the depth or level of the tree. It is used to estimate the size or complexity of a search problem based on the number of nodes at a certain depth.

To measure the effectiveness of a heuristic of a heuristic we estimate the effective branching factor.

$$N + 1 = 1 + EBF + EBF^2 + ... + EBF^d \tag{8}$$

N = total number of nodes expanded by the algorithm
d = depth of the solution
b^* = branching factor of a uniform tree of depth d with $N + 1$ nodes

The closer the EBF is to one, the better the heuristic is.

Example: We have two heuristics, $h_1(n)$ and $h_2(n)$. A search algorithm expands $N_1 = 17$ nodes with the first heuristic, $N_2 = 14$ nodes with the second. The expansion tree has $d = 3$ branching factor.

$$17 + 1 = 1 + EBF_1 + EBF_1^2 + EBF_1^3 \rightarrow \approx 2.165 \tag{9}$$

$$14 + 1 = 1 + EBF_2 + EBF_2^2 + EBF_2^3 \rightarrow = 2 \tag{10}$$

Thus, the second heuristics is better.

5.5 Greedy best-first search (GBFS)

Greedy BFS uses the following evaluation function $f(n) = h(n)$, which is just the heuristic function $h(n)$, that estimates the closeness of n to the goal. Hence, GBFS tries to expand the node that is thought to be closest to the goal, without taking into account previously gathered knowledge (i.e. $g(n)$). The algorithm does not account for the possibility that a node with a lower heuristic value may have a higher path cost, potentially leading to suboptimal solutions.

It has time complexity $O(b^m)$ where b is the branching factor (the average number of children per node) and m is the maximum depth of the search space, and space complexity $O(b^m)$ (it keeps all the nodes in memory). Can get stuck in loops and it is therefore not complete. Not accounting for $g(n)$ it may not always find the shortest path.

If the heuristic function is not accurate or consistent, it may lead to suboptimal solutions or even get stuck in loops. A good heuristic on the other hand can give dramatic improvements.

5.6 Dijkstra's algorithm

Dijkstra's algorithm allows us to calculate the shortest path between a set of nodes. It works on weighted graphs and can find the shortest paths from the start node to all the other nodes on the graph. The shortest path is the sequence of nodes, in the order they are visited, which results in the minimum cost to travel between the start and the end node.

A^* is basically Dijkstra's algorithm for two nodes and with an heuristic function.

5.7 A^* search

The most common informed search algorithm is A^*.

A^* uses a combination of a heuristic function and a cost function $f(n) = g(n) + h(n)$ to evaluate the potential path and determine the best one to take. The heuristic function $h(n)$ estimates the cost of the cheapest path from the current node to the goal node, while the cost function $g(n)$ measures the actual cost of the path from the starting node to the current node. A^* repeatedly selects the node with the lowest estimated total cost $f(x) = g(x) + h(x)$ until it reaches the goal node.

A^* is guaranteed to find the shortest path if heuristic function is admissible (never overestimate the real distance) and consistent (the distance from current node to the goal node is always less than from neighbor to goal node through current node) but GBFS may not.

A^* is complete, unless there are infinitely many nodes with $f < f(G)$ (estimated cost of the goal). It is also optimal. The problem is that the spacial complexity is exponential $O(b^m)$ as in GBFS.

Pros: it is complete (in some cases) and optimal.
Cons: the memory necessary to store the frontier may be exponential.

5.8 F-countours

An f-contour represents the set of nodes that have the same estimated total cost ($f(x) = g(x) + h(x)$) in the pathfinding process. In other words, it is the boundary between the set of nodes that have been expanded (visited) and the set of nodes that have not yet been expanded.

F-contour is important in A^* algorithm as it can help to prune search space by considering only nodes that are on the current f-contour. It can also help to identify the suboptimal solution and discard it, as well as to identify the optimal solution faster.

$$\text{Contour } i \text{ has all nodes with } f = f_i, \text{ where } f_i < f_{i+1}$$

5.9 Iterative-deepening A^* (IDA^*)

IDA^* gives us the benefits of A^* (completeness and optimality) without the requirement to keep all reached states in memory, at a cost of visiting some states multiple times; it is, for this reason, very commonly used for problems that do not fit in memory.

Each iteration exhaustively searches an **f-contour**, finds a node just beyond that contour and uses that node's cost as the next contour value. We can limit this way memory consumption and thus have linear spacial complexity $O(bm)$. This is because it uses a limited amount of memory at each iteration. It does not need to store the whole search tree as A^* does.

5.10 Recursive best-first search (RBFS)

RBFS is a valid alternative to IDA^*

Recursive best-first search stores the f-value of the unexpanded nodes and goes depth-first until the f of the current node does not become worse of the one of the previous alternatives. Whenever the cost of the current node exceeds that of some other node in the previously expanded portion of the tree, the algorithm backs up to their deepest common ancestor, and continues the search down the new path.

5.11 RBFS vs IDA^*

Both algorithms have their own advantages and disadvantages depending on the problem you are trying to solve. IDA^* is more space efficient than RBFS, but RBFS is more flexible in terms of the heuristic function it can use.

In general, IDA^* is considered to be more efficient than RBFS when the heuristic function is consistent and admissible, and when the branching factor of the search space is moderate or low.

In contrast, RBFS is considered to be more efficient than IDA^* when the heuristic function is highly inconsistent or inadmissible, and when the branching factor of the search space is high.

5.12 SMA^*

IDA^* uses too little memory: only the current f-value; $RBFS$ uses too little memory: only the cost of the nodes in the depth first search are recorded;

Simplified memory bounded A^* can use all the available memory for the search. It behaves like A^* till there is memory available. When a new node needs to be generated and the memory is full, the node in memory with the highest f-value is discarded while keeping its cost in the parent node. A discarded node will be re-generated only when all the other paths are worse than the forgotten node.

6 Local search

Local search algorithms operate by searching from a start state to neighbouring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic: they might never explore a portion of the search space where a solution actually resides.

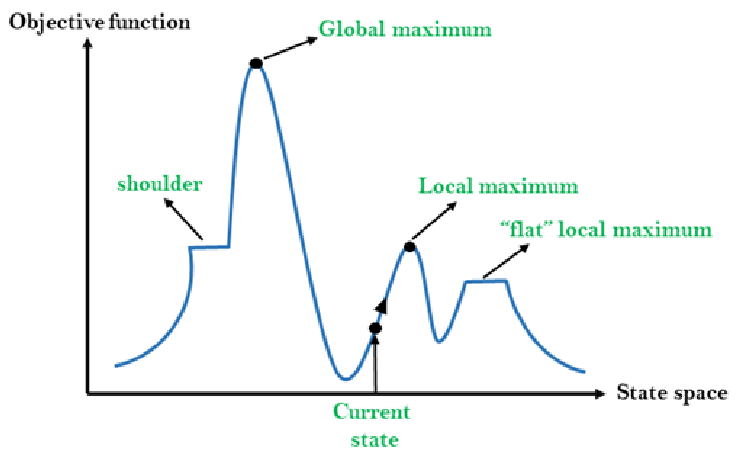
They have two key advantages:

- they use very little memory;
- they can often find a solution in large or infinite state spaces for which systematic algorithms are unsuitable.

6.1 Hill climbing

Hill climbing is a heuristic optimization algorithm that attempts to find the local maximum of a function. It starts at a randomly chosen point on the search space and repeatedly moves in the direction of increasing value until no more such steps can be taken. Hill climbing can be useful for solving optimization problems, but it can also get stuck in many scenarios:

- local maxima/minima: a peak that is higher/lower than any of its neighbors but lower/higher than the global maxima/minima;
- ridges result in a sequence of local maxima/minima that is very difficult for greedy algorithm to navigate;
- plateaus: a flat area of the state-space;



To avoid getting stuck in local maxima, variations of the algorithm such as simulated annealing and genetic algorithms can be used.

6.2 Simulated annealing (SA)

Simulated Annealing is a probabilistic algorithm for global optimization that is often used to find an approximate solution to difficult optimization problems. The algorithm models the process of heating a material and then slowly cooling it down to find a state close to the global minimum energy state.

The algorithm starts with an initial solution and iteratively explores the search space, gradually refining the solution. At each iteration, it considers neighboring solutions and decides whether to accept or reject them based on a probabilistic criterion. Simulated Annealing allows for occasional moves that worsen the objective function in order to escape local optima and explore other regions of the search space.

The key idea behind Simulated Annealing is to balance exploration and exploitation. Initially, the algorithm allows for more exploration, accepting moves that increase the objective function value even if they are worse. However, as the search progresses, the algorithm reduces the probability of accepting worse moves.

The acceptance probability of a worse move is determined by the current temperature parameter and the difference in objective function values between the current and candidate solutions. The temperature parameter controls the level of randomness in accepting worse solutions. As the temperature decreases over iterations, the probability of accepting worse moves diminishes, leading the algorithm towards exploitation and convergence to a local optimum.

The basic process of the algorithm is as follows:

1. Start with an initial solution (e.g., a randomly generated state);
2. Generate a new solution by making small random changes to the current solution;
3. Calculate the energy difference between the new solution and the current solution;
4. Decide whether to accept the new solution as the current solution, based on a probability that depends on the energy difference.
5. Repeat steps 2-4 until a stopping criterion is met.

The time complexity of the SA algorithm is generally considered to be $O(kT^n)$, where k is the number of iterations at a given temperature, T is temperature parameter, and n is the number of variables in the problem. The space complexity is $O(n)$.

Simulated Annealing can be a powerful optimization technique, especially in large and complex search spaces where finding the global optimum is challenging. However, it does not guarantee finding the global optimum, as it can still get stuck in local optima. Various techniques can be employed to enhance its performance, such as adjusting the temperature parameter, neighborhood exploration strategies, or employing multiple runs with different starting points (random restarts).

6.3 Random restart vs Simulated annealing

Random restart and simulated annealing are both techniques used to improve the performance of hill climbing algorithms, which are optimization algorithms that try to find the maximum or minimum value of a function by making incremental changes to the current solution and accepting the change if it improves the solution.

Random restart is a technique where the algorithm starts from a randomly generated solution and repeatedly applies the hill climbing algorithm until it finds a satisfactory solution. The idea behind random restart is that by starting from different initial solutions, the algorithm has a higher chance of escaping local optima and finding the global optimum.

Simulated Annealing starts from an initial solution and then generates a new solution that is randomly different from the current one. The algorithm will accept the new solution if it is better than the current one, otherwise, it will accept it with a probability that decreases as the number of iterations increases. The idea behind this is that the algorithm will not get stuck in a local optimum but it will be able to explore a wider range of solutions by allowing some bad moves.

In summary:

- random restart is a technique that makes the algorithm starts from different initial states in order to overcome local minima/maxima;
- simulated annealing is an algorithm that randomly generates new states from the current one, accept them as new states if they are better than the current or if they are worse than the current but this is the choice to make by generating a random value. Going on the algorithm reduces the likelihood of accepting worse states.

6.4 Local beam search (LBS)

Each iteration, LBS generates a set of new states from the current beam, and then selects the best k states from the combination of the current beam and the new states to form the next beam. The process repeats until a goal state is found or a certain stopping criterion is met.

The problem is that the states converge in the same region of the search space.

LBS can easily get stuck in local optima. Because LBS only generates new states from the states in the current beam, it has a limited view of the search space and may miss global optima that are outside of its current beam. This can lead to the algorithm converging on a suboptimal solution, rather than the true optimal solution.

The steps are:

- From a **single** start state, maintain K states and not just a single one (like in other search algorithms);
- At each iteration, the best K successor states for each state are identified;
- If the goal is found, then halt, else iterate the procedure.

6.5 Genetic algorithms

Genetic algorithms work by simulating the logic of Darwinian selection.
Genetic algorithms starts by generating an initial population (random strings).

1. This **initial population** consists of all the probable solutions to the given problem.

2. The **fitness function** helps in establishing the fitness of all the individuals in the population. It assigns a fitness score to each of them, which further determines the probability of being chosen for reproduction. The higher the fitness score, the higher the changes of being selected.

3. In the **selection phase**, individuals are selected and then arranged in pairs of two to enhance reproduction. The genetic algorithm uses the **fitness proportionate selection technique** to ensure that useful solutions are used for recombination.

4. The **reproduction phase** involves the creation of the child population. The two main operators in this phase include crossover and mutation.

    - **Crossover**: this operator swaps the genetic information of the two parents to produce the offspring; the child population is of equal size as the parent population.
    - **Mutation**: this operator adds new genetic information to the new child population; this is achieved by flipping some bits in the chromosomes. Mutation **solves the problem of local minimum** and enhances diversification.

5. **Genetic replacement** takes place next: the old population is replaced with the new child population. The new population consists of higher fitness score than the old population.

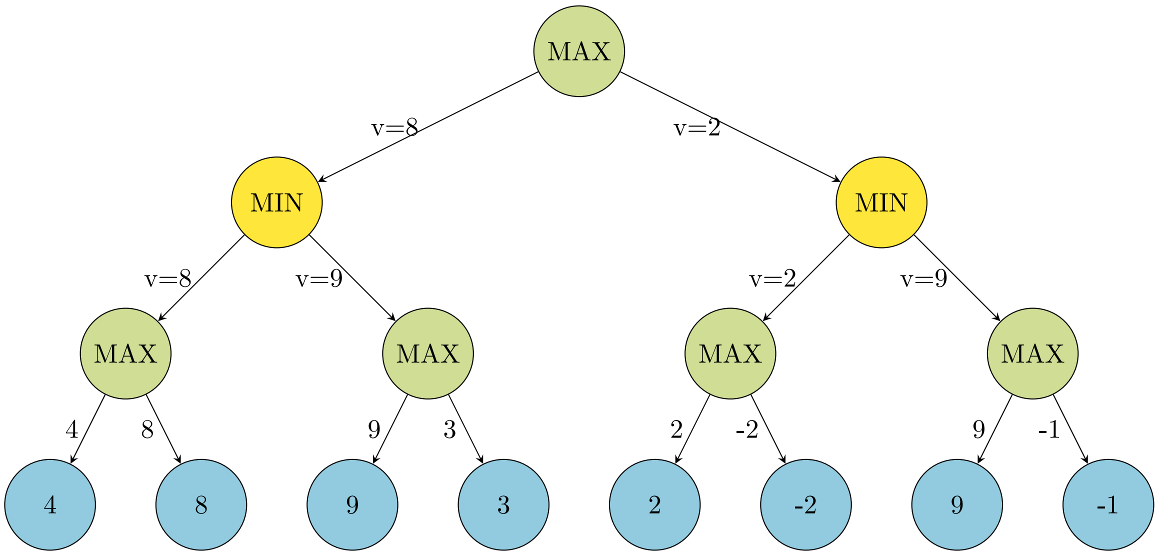
6. If a **stopping criterion** is met, we could terminate (threshold in fitness).

7 Minimax

Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.

Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.

Mnimax works with a game tree. We have two agents, one called **MAX** and one called **MIN**. MAX tries to maximize the reward of the game (utility), while MIN tries to minimize that reward.



Explanation on how the Minimax algorithm works

8 Monte Carlo Tree Search (MCTS)

Explanation on how the Monte Carlo Tree Search algorithm works

9 Expectiminimax

Explanation on how the Expectiminimax algorithm works