## 3. Solving problems by search

## 3.1 Problem solving agents

Intelligent agents are supposed to maximize their performance measure. Achieving this is sometimes simplified if the agent can adopt a goal and aim at satisfying it. **Goals** help organize behaviour by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider. **Goal formulation**, based on the current situation and the agent’s performance measure, is the first step in problem-solving

**Problem formulation** is the process of deciding what actions and states to consider, given a goal.

*In general, an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.*

The process of looking for a sequence of actions that reaches the goal is called **search**. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution phase**.

An optimisation problem is called **tractable** if it can be solved in polynomial time.

## 3.2 Some examples

## 3.3 Searching for solutions

### 3.3.1 Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

* STATE: the state in the state space to which the node corresponds;
* PARENT: the node in the search tree that generated this node;
* ACTION: the action that was applied to the parent to generate the node;
* PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers

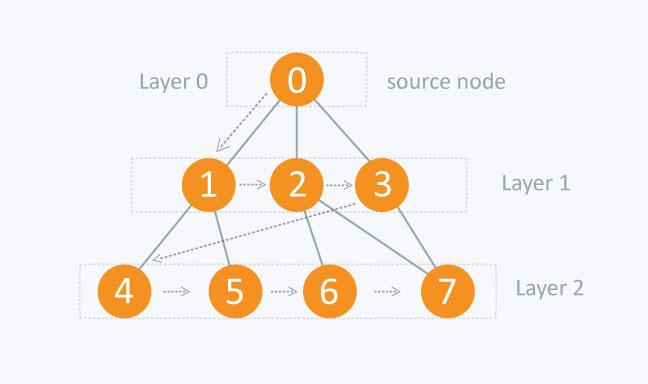
### 3.3.2 Measuring problem-solving performance

**Completeness**: Is the algorithm guaranteed to find a solution when there is one  
**Optimality**: Does the strategy find the optimal solution (aka. The shortest path)  
**Time complexity**: How long does it take to find a solution? (can also be measured in nodes generated)  
**Space complexity**: How much memory is needed to perform the search?

## 3.4 Uninformed search strategies

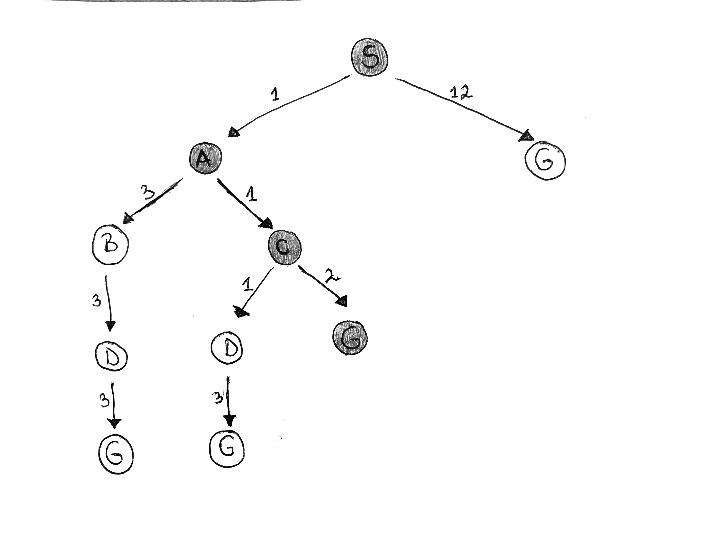
### 3.4.1 Breadth-first search

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, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. BFS is optimal when all step costs are equal. Implemented with FIFO queue



### 3.4.2 Uniform cost search

Uniform-cost search expands the node n with the lowest path cost *g(n).* This is done by storing the frontier as a priority queue ordered by *g.*



### 3.4.3 Depth-first search

Depth-first search always expands the deepest node in the current frontier of the search tree. Implemented with LIFO queue.

### 3.4.4 Depth-limited search

Supply depth-first search with a predetermined depth limit l. That is, nodes at depth l are treated as if they have no successors. Depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose ld. Its time complexity is O(b^l) and its space complexity is O(bl). Depth-first search can be viewed as a special case of depth-limited search with l =∞.

### 3.4.5 Iterative deepening search

IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically we do DFS in a BFS fashion.

## 3.5 Informed search algorithms

Use the information beyond the definition of the problem itself to improve efficiency

The general approach we consider is called **best-first search**. Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, f(n). The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first.

Most best-first algorithms include a component of a heuristic function, denoted **h(n)**

*h(n) = estimated cost of the cheapest path from the state at node n to a goal state*

* *What is a heuristic?*
  + *“Heuristic” means “serving to aid discovery”*
  + *A rule of thumb to find answers*
  + *It helps estimate the quality or potential of partial solutions*
  + *It helps when no algorithmic solution is available*

Admissible heuristics:  a heuristic function is said to be admissible **if it never overestimates the cost of reaching the goal**

Consistency (or sometimes monotonicity): A heuristic function h(n) is consistent if, for every node n and every successor n’ of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n’ plus the estimated cost of reaching the goal from n’.

**Every consistent heuristic is also admissible**

### 3.5.1 Greedy best-first

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n)

### 3.5.2 A\*

It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n) . Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.

## 4.1 Local search

Local search algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbours of that node. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages: **(1)** they use very little memory—usually a constant amount; and **(2)** they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.

**Local search often keeps a single "current" state and tries to improve it**

**Disadvantage**: Depending on the initial state, search can get stuck in local maxima or minima.

### 4.1.1 Hill-climbing search

The hill-climbing search algorithm (steepest-ascent version) is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a “peak” where no neighbour has a higher value.

Reaching a local minima/maxima will stop the search.

Disadvantages: hitting a local maximum, ridge or plateaux. Local max is a peak that is higher than each of its neighboring states but lower than the global maximum.

**Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than the steepest ascent, but in some state landscapes, it finds better solutions.

**First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors. *Good for big problems.*

**Random-restart hill climbing** adopts the well-known adage, “If at first you don’t succeed, try, try again.” It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state.

### Simulated annealing

* Escape local maxima by allowing some bad moves
* Gradually decrease size and frequency of allowed bad moves as search proceeds

To explain simulated annealing, (gradient descent i.e., minimizing cost), imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

The innermost loop of the simulated-annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the “badness” of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the schedule lowers T slowly enough, the algorithm will find a global optimum with a probability approaching 1.

### Local beam search

Optimises best-first-search by limiting memory usage. Keep track of *k* states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats

## 3.7 Summary

Text

Description automatically generatedText

Description automatically generated