**AIML 1 - Search problems and uninformed search algos**

***The problem solving process***

* Goal formulation → limit actions
* Problem formulation → define model structure, acceptable inputs and feasible solutions
* Search → search for that solution (i.e. **learn it**)
* Execution

We will assume a fully observable, deterministic, known environment and an *open loop* in the agent (capability of ignoring percepts mid-execution)

Some definitions:

* **State space S** set of every state possible (S)
* **Initial state** the starting state
* **Goal states** could be multiple, aim of search, subset of S

F: S → B B is a Boolean (is goal or no)

* **Actions A** applicable actions to elements of S

F **:** S → 2ᴬ 2ᴬ is a power set (the set of all possible subsets of S)

F(s) returns the set of actions that can be executed in *s*

* **Transition model** description of what each action does

F : S X A ↛ SF(s,a) returns the state that results from doing an action *a* in a state *s*

* **Action-cost fct** evaluates model performance (assuming cost > 0)
* **Path** sequence of actions
* **Solution** a path from the initial state to one of the goal states
* **Performance measure (seen in Agents)**

***Abstraction of a problem***

A fundamental part of problem solving is the process of removing irrelevant data to pursue an optimal solution

The abstraction is *valid* if we can elaborate any abstract solution in a believable, quasi-concrete real world example, the *usefulness* of the abstraction comes from its simplicity

(further examples in book)

***Tree vs Graph search***

The basic **differences** are:

* In the case of a graph search, we use a list, called the **closed list** (also called **explored set**), to keep track of the nodes that have already been visited and expanded, so that **they are not visited** and expanded again (therefore considered more space consuming, we have to remember at least a walk we did)
* In the case of a **tree search**, we do **not** keep this closed list. Consequently, **the same node can be visited multiple** (or even infinitely many) times, which means that the produced tree (by the tree search) may contain the same node multiple times (therefore considered more time consuming, it is memoryless)

It’s easier to look for **cycles** than to remember where exactly we have been (a path instead of all of them)

***Blind search (uninformed search)***

Graphical user interface, text

Description automatically generatedThe uninformed search algorithm does not have any domain knowledge such as closeness, location of the goal state, behaving in a brute-force way, opposed to **heuristics in informed search**

***Best-first search (example of informed search)***

Best-first search refers to a **class** of search algorithms, in which the most promising (according to predefined rules) is chosen to be explored first, usually implying the usage of **distance as heuristics** via setting a **priority queue**

To search the graph space, the BFS\* method uses two lists for tracking the traversal:

* + An ‘Open’ list that keeps track of the current ‘immediate’ nodes available for traversal
  + A ‘Closed’ list that keeps track of the nodes already traversed.

A natural question would be “**how does this differ from Dijkstra’s algorithm?”**

BFS\* fails on weighted graphs, as it will choose **distance** over **cost of edge traversal**

Some algorithms definitions, with code implementation in the slides:

* **BFS – Breadth first search**

In breadth-first search, the root node is expanded first, then all its successors, and then all their successors, and so on until a goal state is found or the entire graph has been traversed

* **DFS**

Depth-first search algorithms will always expand the deepest node in the frontier first

The search proceeds to expand nodes deeper and deeper until the deepest level of the search tree, where the nodes have no successors and then “backtracks” to the next deepest node

* **Uniform cost search (variant of Dijkstra’s shortest path algorithm)**

Here, instead of inserting all vertices into a priority queue, we insert only source, then one by one insert when needed

In every step, we check if the item is already in priority queue (using visited array)

If yes, we perform decrease key, else we insert it

**UCS is goal-based**, while Dijkstra’s is not (it finds the shortest path to **all** nodes)

* **Depth-limited search**

DLS is a limitation of DFS, to prevent unnecessary nodes to be visited, with the issue being: what if the goal is further down?

In that case, the algorithm terminates, and there is a trade-off: the gain of time removes the fact that the goal will certainly be found as the tree is not traversed entirely

* **Iterative Deepening (depth-first) search**

This algo works **only** on d-graphs

Combines DLS and BFS and certainly finds a solution and a path to it

**The idea is to try out incremental values *0,1,2,...* as the depth limit *l* until DLS finds a target node.**

* **Bidirectional search**

**Bidirectional search, unlike DFS and BFS, runs:**

* + **Forward search (from root to goal)**
  + **Backward search (from goal to root)**

**Replacing each search graph with two smaller subgraphs, and terminates when they intersect**

**It is complete if BFS is used for both subgraphs**

**Table

Description automatically generated**

The branching factor is the **out-degree,** the number of children at each node