1 Uninformed and Informed Search

1.1 Problem Formulation

Problem-solving agents are result-driven. They always focus on satisfying their goals, i.e., solving the problem. While problems are often given in a human-understandable way, we need to reformulate the problem for our agent. These agents employ algorithms to find solutions.

Steps to formulate a solvable problem:

1. **Formulate the goal**
2. **Formulate the problem** given the goal

1.1.1 Key Terminology

The State Space / States

A state describes a possible situation in our environment. The **state space** is a set of all possible situations (states).

Transition / Action

Transitions describe possible actions to take between one state and another. We only count direct transitions between two states (single actions).

Costs

Often transitions aren't alike and differ. We express this by adding a "cost" to each action. Often the goal in search algorithms is to **minimize the cost** to reach the goal.

A **single state problem** is defined by 4 items:

1. **State space and Initial state** Description of all possible states and the initial environment as state.
2. **Description of actions** Typically a function that maps a state to a set of possible actions in this state.
3. **Goal test** Typically a function to test if the current state fulfills the goal definition.
4. **Costs** A cost function that maps actions to its cost. An easy way is to have additive costs (sum of costs for all actions taken).

1.1.2 The State-Space Graph

The state space is the set of all states reachable from the initial state. It is implicitly defined by the initial state and the successor function, forming a **state-space graph**.

- **Path:** A sequence of states connected by a sequence of actions.
- **Solution:** A path that leads from the initial state to a goal state.
- **Optimal Solution:** A solution with the minimum path cost.

1.1.3 Core Search Definitions

Planning Problem

A planning problem is one in which we have an initial state and want to transform it into a desired goal, considering future actions and their outcomes.

Search

The process of finding the (optimal) solution for such a problem in the form of a sequence of actions.

1.2 Search Fundamentals

1.2.1 Tree Search vs. Graph Search

The state-space graph can be explored by building a **search tree**.

- **Tree Search:** Treats the state space as a tree. It does not keep track of visited states, so it might re-explore the same state via a different path. This can lead to exponential work for problems with loops or redundant paths.
- **Graph Search:** Remembers states that have been visited in an **explored set** (or "closed set"). It avoids expanding states that are already in the explored set, thus handling loops and redundant paths efficiently.

1.2.2 States vs. Nodes

State

Representation of a physical configuration. Describes a specific situation in our environment (e.g., "in Arad").

Node

A data structure to represent a part of a search tree. It includes a **state**, a **parent node**, the **action** taken, the **path cost** ($g(n)$), and the **depth** (e.g., "the path Arad $\rightarrow$ Sibiu").

1.2.3 Key Search Tree Terminology

Fringe

The set of all nodes at the end of all visited paths is called the fringe. (Also known as **frontier** or "open set"). These are the nodes available for expansion.

Depth

Number of levels in the search tree.

1.2.4 Evaluating Search Strategies

Search strategies are evaluated along the following dimensions:

- **Completeness:** Does it always find a solution if one exists?
- **Time Complexity:** Number of node expansions.
- **Space Complexity:** Maximum number of nodes in memory.
- **Optimality:** Does it always find the optimal (least-cost) solution?

Complexity is measured in terms of:

- b: maximum **branching factor** of the search tree.
- d: the **depth** of the optimal solution.
- m: the **maximum depth** of the state space (may be ∞).

1.3 Uninformed Search Strategies

Uninformed Search

Do not have any information about the problem except the problem definition. (Also called **Blind Search**).

Breadth-First Search (BFS)

A special case of Uniform-Cost Search where all step costs are equal. It starts at the tree root and explores the tree **level by level**. It uses a FIFO (First-In-First-Out) queue for the fringe.

- **Completeness:** Yes.
- **Time:** $O(b^d)$ (The summary table uses $O(b^{d+1})$).
- **Space:** $O(b^d)$. Memory consumption is its biggest drawback.
- **Optimality:** Yes (if all costs are equal).

Uniform-Cost Search (UCS)

Each node is associated with a cost, which accumulates over the path. UCS expands the node with the **lowest cumulative path cost** ($g(n)$). It is often implemented with a **priority queue**.

- **Completeness:** Yes (if step costs are positive, i.e., $> \epsilon > 0$).
- **Time:** $O(b^(1 + \lfloor C^/\epsilon \rfloor))$, where C^* is the cost of the optimal solution.
- **Space:** $O(b^(1 + \lfloor C^*/\epsilon \rfloor))$.
- **Optimality:** Yes.

Depth-First Search (DFS)

Starts at the tree root and explores as far as possible along one branch before backtracking. It uses a LIFO (Last-In-First-Out) stack for the fringe.

- **Completeness:** No. Fails in infinite-depth spaces or spaces with loops.
- **Time:** $O(b^m)$, where m is the max depth. Can be terrible if $m \gg d$.
- **Space:** $O(b \times m)$. This linear space complexity is its key advantage.
- **Optimality:** No.

Depth-limited Search (DLS)

A variation of DFS where the search is limited to a predetermined depth l. Nodes at depth l are not expanded.

- **Completeness:** No (if $l < d$).
- **Time:** $O(b^l)$.

- **Space:** $O(b \times l)$.
- **Optimality:** No.

Iterative Deepening Search (IDS)

Combines the benefits of BFS and DFS. It runs DLS repeatedly with increasing depth limits: $l = 0, 1, 2, \ldots, d$.

- **Completeness:** Yes.
- **Time:** $O(b^d)$ (Despite re-generating upper levels, the overhead is small).
- **Space:** $O(b \times d)$.
- **Optimality:** Yes (if costs are uniform).

Bidirectional Search

Performs two searches simultaneously: one forward from the initial state, one backward from the goal state. Stops when the two searches meet.

- **Completeness:** Yes.
- **Time:** $O(b^{d/2})$.
- **Space:** $O(b^{d/2})$.
- **Notes:** Only possible if actions can be reversed.

1.3.1 Summary of Uninformed Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes*	Yes	No	No	Yes*

Table 1: *Comparison of uninformed search strategies. (*Assumes uniform step costs or $l \geq d$ where applicable).*

1.4 Informed Search Strategies

Informed Search

Have additional knowledge about the problem (beyond the definition) and an idea of where to "look" for solutions.

This "hint" is provided by a heuristic function.

1.4.1 Heuristics

Heuristic $h(n)$

Informally denotes a "rule of thumb". In tree-search, a heuristic is a function $h(n)$ that **estimates the remaining cost** to reach the goal from node n.

1.4.2 Greedy Best-first Search

Greedy Best-first Search

Uses an evaluation function $f(n) = h(n)$ to estimate the cost from node n to the goal. It expands the node that appears to be closest to the goal, according to the heuristic. It does not care about the actual cost/distance.

- **Completeness:** No. Can get stuck in loops. (Complete in finite spaces with loop detection).
- **Time:** Worst case $O(b^m)$.
- **Space:** Worst case $O(b^m)$ (keeps all nodes in memory).
- **Optimality:** No. The solution depends entirely on the heuristic.

1.4.3 A* Search

A* Search

An informed tree search algorithm, building on best-first search. It is the "best-known" form. It avoids expanding paths that are already expensive.

A* evaluates nodes using the function: $f(n) = g(n) + h(n)$.

- $g(n)$ = **true cost** so far to reach node n.
- $h(n)$ = **estimated cost** to get from n to the goal.
- $f(n)$ = **estimated cost** of the cheapest solution path that goes through node n.
- **Completeness:** Yes (unless there are infinitely many nodes with $f(n) \leq f(G)$).
- **Time:** Can be exponential unless the error of the heuristic $h(n)$ is bounded.
- **Space:** Has to keep all nodes in memory. This is the primary drawback of A*.
- **Optimality:** Yes, **if the heuristic** $h(n)$ **is admissible**.

1.4.4 Heuristic Properties

Admissible Heuristics

A heuristic is **admissible** if it **never overestimates** the true cost to reach a goal. Formally:

$$h(n) \leq h^*(n) \quad \text{for all nodes } n,$$

where $h(n)$ is the heuristic estimate and $h^*(n)$ is the actual optimal cost from n to the goal. (For example, the straight-line distance heuristic h_{SLD} is admissible in route-finding problems.)

Consistent Heuristics

A heuristic is **consistent** if for every node n and every successor n' generated by action a, the "triangle inequality" holds: $h(n) \leq c(n, a, n') + h(n')$. This means the heuristic difference between adjacent nodes never overestimates the actual step cost.

- **Lemma 1:** Every **consistent** heuristic is also **admissible**.
- **Lemma 2:** If $h(n)$ is consistent, then the values of $f(n)$ along any path are **non-decreasing**.

Relaxed Problems

A problem with fewer restrictions on the actions is called a relaxed problem. The cost of an optimal solution to a relaxed problem is an **admissible heuristic** for the original problem.

Example (8-puzzle):

- $h_1(n) =$ Number of misplaced tiles. (Relaxed rule: tile can move anywhere).
- $h_2(n) =$ Total Manhattan distance. (Relaxed rule: tile can move to any adjacent square).
- Both h_1 and h_2 are admissible.

Dominance

If h_1 and h_2 are both admissible and $h_2(n) \geq h_1(n)$ for all n, then h_2 **dominates** h_1.

A* will expand fewer nodes with a dominant heuristic. (e.g., for the 8-puzzle, h_2 (Manhattan) dominates h_1 (misplaced tiles)).

Combining Heuristics If we have several admissible heuristics $h_1(n), \ldots, h_m(n)$, we can combine them. $h(n) = \max h_1(n), h_2(n), \ldots, h_m(n)$ is also admissible and dominates all of its components.

1.4.5 Optimality of A

A (using Tree Search) is optimal if its heuristic $h(n)$ is admissible.

Proof (Informal):

1. Let G be an optimal goal state, with path cost C^*.
2. Assume for contradiction that A is about to return a suboptimal goal G_2, with path cost $g(G_2) > C^*$.
3. At this moment, G_2 is in the fringe. Because A^* chose G_2, its f-value must be the lowest, so $f(G_2) \leq f(n)$ for all other fringe nodes n.
4. Let n be any unexpanded node on a true optimal path to G. This node n must be in the fringe.
5. **Analyze $f(G_2)$:** For a goal state, $h(G_2) = 0$. So, $f(G_2) = g(G_2) + h(G_2) = g(G_2)$. Since G_2 is suboptimal, $f(G_2) = g(G_2) > C^*$.
6. **Analyze $f(n)$:** $f(n) = g(n) + h(n)$. Because h is admissible, $h(n) \leq h^{(}n)$ (where $h^{(}n)$ is the true cost from n to G). The true cost of the optimal path is $C^{=}g(n) + h^{(}n)$. Therefore, $f(n) = g(n) + h(n) \leq g(n) + h^{(}n) = C^*$.
7. **Contradiction:** We have shown $f(n) \leq C^*$ and $f(G_2) > C^*$. This means $f(n) < f(G_2)$. A would be forced to expand n (on the optimal path) *before* it could ever expand G_2. Thus, A* can never select a suboptimal goal. It is optimal.

1.4.6 Memory-Bounded Heuristic Search

The main problem with A* is its space complexity. Alternatives include:

1. **Iterative-deepening A* (IDA):** Like IDS, but the "depth" cutoff is the f-cost $(g + h)$.
2. **Recursive best-first search (RBFS):** Mimics best-first search using linear space by recursively re-expanding nodes and updating f-values from ancestors.
3. **(Simple) Memory-bounded A ((S)MA*):** When memory is full, drops the worst (highest f-value) leaf node.

1.5 Tree Search vs. Graph Search (Revisited)

Failure to detect repeated states can turn a linear problem into an exponential one!

Graph Search

Uses an **explored set** (or "closed set") to store all states that have been expanded. When expanding a node, its successors are only added to the fringe **if they are not in the fringe or explored set**.

Optimality of A* Graph Search

- If $h(n)$ is only **admissible**, Graph Search A* is not guaranteed to be optimal. It might find a suboptimal path to a node first, add it to the explored set, and never find the optimal path.
- If $h(n)$ is **consistent**, Graph Search A* **is optimal**.
- **Why?** A consistent heuristic guarantees that f-values are non-decreasing along any path. This means the *first* time A* expands a node n, it is *guaranteed* to have found the shortest possible path to it. Therefore, we never need to re-expand any node in the explored set.