Reading 1

☆ Article: Search algorithms in AI

- In AI, a search problem involves navigating from an initial state to a goal state through a series of valid actions.
- A well defined search problem cosists of:
  1. Initial State: Where the agent begins
  2. Actions: All possible moves or decisions from each state
  3. Goal State: The desired outcomes or target
  4. Path Cost: A numerical value representing the cost or effort required to reach a goal.

- Search Tree: branching structure generated from initial state without revisiting past states
- Search Graph: A more efficient structure that accounts for repeated states and shared paths, reducing redundancy.

- Key Terminology:
  - State Space: the complete set of all possible states the agent can reach.
  - Nodes: Represent states in the search tree, including metadata like parent node, path cost, and depth.
  - Frontier: Also known as the open list; it's a queue of nodes waiting to be explored.
  - Explored Set: The set of already visited nodes, used to avoid redundancy and loops.

  - Cost functions play a key role in evaluating and guiding searches:
    - $g(n)$: cost from the start node to node n.
    - $h(n)$: Heuristic estimate of the cost from n to the goal.
    - $f(n) = g(n) + h(n)$: Estimated total cost

  - Optimality: Whether the algorithm finds the best solution
  - Completeness: Whether it gaurantees finding a solution if one exists.
  - Time and Space Complexity: Resources needed for computation and memory.

- Types of Search Algorithms

1. Uniformed Search (Blind Search): these algorithms have no additional info about goal location other than the problem definition.

2. Informed Search (Heuristic Search): these use heuristic functions to estimate the cost to reach the goal.

- Uniformed Search Algorithms

1. Breadth-First Search (BFS): explores nodes level by level, expanding to all neighbors before moving to next depth level.

Pros: Gaurantees optimal solution, simple to implement

Cons: High Space Complexity, slows on deep trees

2. Depth-First Search (DFS): explores as deep as possible before backtracking.

Pros: Low memory usage, faster in deep trees.

Cons: Can get stuck in loops, doesn't gaurantee shortest path

3. Uniform Cost Search: Expands the least-cost node first using a priority queue

Pros: Always finds the lowest cost path

Cons: Slow in large spaces, requires cost function and priority queue.

4. Depth-Limited and Iterative Deepening Search: A DFS with a fixed depth limit to prevent infinite recursion.

Pros: Memory efficient, complete like BFS

Cons: Repeats work in earlier iterations, may be slow.

• Informed Search Algorithms (Heuristic Search)

1. **Greedy Best-First Search**: uses a heuristic function $h(n)$ to expand the node that appears closest to the goal, ignoring the path cost $g(n)$. It selects the node with the lowest $h(n)$ at each step.

   Pros: Fast, memory-efficient, good for approximate solutions.

   Cons: Not optimal or complete; can get stuck in local minima.

2. **A* Search (Tree and Graph Versions)**: widely used, combined evaluation function $f(n) = g(n) + h(n)$ where $g(n)$ = cost to reach node n, $h(n)$ = estimated cost from n to the goal. A* expands nodes with the lowest $f(n)$ value.

   Pros: Optimal, efficient with good heuristics.

   Cons: Can consume high memory, performance depends on heuristic quality.

3. **Hill Climbing and Variants**: continuously moves in the direction of increasing value (better heuristic score) until it reaches a peak. Prone to getting stuck at local maxima.

   Pros: Fast, memory efficient

   Cons: Not optimal, can get stuck easily

4. **Beam Search + Bidirectional Search**: Beam search is a heuristic-based optimization of BFS that expands only a fixed number of best nodes at each level.

   Pros: Reduces memory and time; suitable for approximate solutions

   Cons: Not complete or optimal; highly sensitive to beam width

- **Bidirectional Search**: runs 2 simultaneous searches → one forward from initial state and one backwards from the goal.

☆ Article: Introduction to the A* Algorithm

- Input: Graph
  - ↳ The algorithm ONLY sees the graph (not its surroundings or anything else)
- Output: Another graph
  - ↳ A* will tell you how to move from one location to another but it won't tell you how.
  - ↳ So any doorways in a maze need to be nodes or part of an edge.