Week 13

State Spaces and Search Note

Saagar Sanghavi, Jin Seok Won, Kevin Zhu

EECS 16ML

Introduction

In 1956, Edsger Dijkstra created what is known as the Dijkstra's algorithm without pen or paper in 20 minutes. In the many decades that followed, Dijkstar's algorithm and its variants became and remained one of the most fundamental and widely used algorithms in all of computing. This algorithm and maybe even its backstory might not be anything new to you. This note will focus on how search algorithms like Dijkstra's can be applied in the context of artificial intelligence and robotic control, as many of you might be familiar with from EECS 16AB courses.

Robotics and Control

After making the body of the robot, one big problem in controlling the robot is how it can go from one place to another. It almost sounds trivial, but to a newly created robot any simple task is a challenge. In this context, we must define every movement of the robot mathematically. Most of the time, when we walk outside, walking to class, we do not think about how we are doing it, since we have done it many times, and the brain makes most decisions subconsciously. But every movement of the robot must be precisely ordered by its programming, at least with our current knowledge.

Agents and State Space

In the study of artificial intelligence, the goal is to create an entity that makes a rational decision given the knowledge about its **environment**. Such an entity is called an **agent**. Agents and the environment that it is in is called the **world**.

In the case of our robot, the agent is the robot itself. The environment can be anything surrounding it. If we want the robot to walk to the destination, the environment would be its starting and goal locations, any obstacles on the ground, or any humans getting in the way.

In order for the agent to make a rational decision, we, the ones programming the agent, need to be able to describe the world in terms of mathematics. Such formulation is called a **search problem**. It asks the question of how can the agent search through the possible states and make the best possible step towards its goal. A search problem is expressed with four components:

- **State Space** - The set of all possible states in the given world
- **Successor function** - A function that takes in a state and an action and returns the cost of given action and the state after taking that action, which is called **successor state**
- **Start State** - The initial state where the agent begins in
- **Goal test** - A function that takes in a state and returns whether the goal is achieved

A search problem consists of starting at start state and using the successor function to explore the state space. This process is repeated iteratively at different successor states until reaching the goal.

The path from start state until reaching the goal is called the **plan**.

The states are considered in a predetermined order called **strategy**.

The size of the state space can usually be computed using fundamental counting principles. Since the size of state space is often too large to save in memory, successor functions are used to save only part of the space currently used in search and compute the rest when necessary.

Search Algorithms

Now we know that our robot needs to find a path. But as we have seen, the search space can be vast beyond imagination. Without any strategy, searching through every combination of path can easily take thousands of years or more. Luckly, even relatively simple algorithms like BFS allow us to find some solution, eventually.

Fringe is a central part in most search algorithms. A search protocol usually involves maintaining an outer fringe which contains potential search targets, selecting the next node to remove using the strategy and adding all of the removed node's children to the fringe. This fringe contains partial plans which would be completed when the goal node is removed from the fringe. The following pseudocode encompasses most of the search algorithms discussed here:

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Any strategy can be evaluated by some of its basic properties:

- **Completeness** - Whether the strategy is guaranteed to find the solution if one exists within reasonable time and resources

- **Optimality** - Whether the strategy is guaranteed to find the lowest cost plan
- Time and Space Complexity - Determined by factors like branching factor, depth of minimum depth solution, maximum depth

Variables

- Depth m
- Branching factor b
- Cost of optimal path C*
- Minimum cost in optimal path epsilon

Uninformed Search

When given no information about where the solution might be, we can use uninformed search strategies to find the solution. Uninformed search algorithms that we will cover are Depth-First Search, Breadth-First Search, and Uniform Cost Search which is a slight variation from Dijkstra's algorithm.

Depth-First Search (DFS)

- Strategy - Select the deepest node in the fringe. There is usually a consistent tie-breaking scheme for multiple nodes with the same depth.
- Fringe - Since removing the current deepest node and adding its children to the fringe makes the children the newest deepest node, newest nodes get highest priority. This behavior can be most easily implemented with last-in, first-out (LIFO) stack.
- Completeness - DFS is not complete because when the state space graph contains a cycle it can effectively get stuck at the cycle.

- Optimality - DFS finds the “leftmost” solution, not an optimal one. “Left” is the node preferred by the tie-breaking scheme.
- Time Complexity - In the worst case, DFS explores the entire search tree which will take O(b^m) time.
- Space Complexity - In the worst case, at each depth, the explored node will have b children for m levels which will result in space complexity of O(bm).

Breadth-First Search (BFS)

- Strategy - Select the shallowest node in the fringe.
- Fringe - Shallowest nodes will be the oldest nodes in the fringe, which means highest priority should be given to nodes first added. This behavior can be most easily implemented with a first-in, first-out (FIFO) queue.
- Completeness - BFS will eventually get to any finite depth nodes, so it is complete
- Optimality - BFS finds the shallowest solution, which is not necessarily optimal if the node weights are not uniform.
- Time complexity - BFS will explore every node until it encounters a solution, so it has time complexity of O(b^s) which is the max total number of nodes up to depth s which is the depth of the shallowest node.
- Space complexity - BFS can contain all nodes of the same depth in the fringe. Max number of nodes of the same depth is O(b^s) which can be reached in depth s.

Uniform Cost Search

- Strategy - Select the lowest cost node in the fringe. Cost of the node is the sum of edge weights of the partial plan.

- Fringe - Usual choice for fringe of UCS is priority queue, ordered based on cost of the partial plan, or the sum of cost from the start node to v. Priority queue ensures that the lowest cost node is prioritized by reordering itself after additions and removals.
- Completeness - UCS always finds a solution since the shortest path to solution must exist in state space.
- Optimality - UCS is optimal. It can be proved that if there is shortest path to the goal node (there has to be if start and goal node are connected) then UCS finds the optimal solution
- Time and Space Complexity
    - Runtime: O(b^(C*/epsilon))
    - Space Complexity: O(b^(C*/epsilon))
    - Some of you might be more familiar with Dijkstra's algorithm's runtime analysis which involves the number of nodes v and number of edges e. However, UCS terminates upon reaching a goal state which creates more complex runtime.

Informed Search

Uniform Cost Search is complete and optimal but it can be fairly slow. Given information about where the goal node might be, the time and space complexity can be reduced significantly. Search problems where we are given prior information about where the goal node might be are called **informed search**.

Heuristics refer to the techniques that allow us to make estimations very quickly while known algorithms can be very slow. Heuristics are a central part of the informed search algorithms. If we can compute the heuristic about the potential state, we can prioritize them to have much

better runtime. Heuristics need to be a lower bound on the remaining distance for the informed search solution to be optimal.Sta

Greedy Search

- Strategy - Select the lowest heuristic node from the fringe.
- Fringe - Very similar to UCS, but instead of the cost of the partial plan, greedy search uses the heuristic value, which can be interpreted as estimated cost to the goal node.
- Completeness and Optimality - In general greedy search is not complete or optimal.

A* Search

- Strategy - Select the lowest estimated total cost node in the fringe, where estimated total cost is the sum of cost of partial plan and heuristic.
- Fringe - Very similar to UCS, but computed heuristic is added to the cost of partial plan when being added to the fringe.
- Completeness and Optimality - Given a heuristic that underestimates the total cost, A* search is both complete and optimal

Summary of Search Algorithms and their Properties

	Strategy	Complete	Optimal	Runtime	Space
DFS	Max depth	No	No	O(b^m)	O(bm)
BFS	Min depth	Yes	No	O(b^s)	O(b^s)
UCS	Lowest cost	Yes	Yes	O(b^(C*/epsilon))	O(b^(C*/epsilon))
Greedy	Lowest heuristic	No	No	*	*
A*	Lowest total cost	Yes*	Yes*	*	*

*depends on the heuristic

Sources:

https://inst.eecs.berkeley.edu/~cs188/fa20/assets/notes/note01.pdf

https://www.geeksforgeeks.org/search-algorithms-in-ai/