Search for Artificial Intelligence and Machine Learning

Tim Won, Kevin Zhu, Saagar Sanghavi

This note was first developed by Team SKT for CS189, Project T Final Fall 2020.

1 Introduction

1.1 Conceptual Background

This note will focus on how search algorithms can be applied in the context of artificial intelligence and robotic control. You have seen graph search algorithms in a class like CS61B, and you have also been exposed to some common challenges in robotics and control from EECS16B. In this note, we shall connect these two concepts, applying the ideas of graph searches to solve some fundamental robotics problems and give you a taste for how the state-space search technique can be used to solve a wide variety of problems in artificial intelligence and machine learning.

In EECS16B, we studied controllable systems from a linear-algebraic perspective. Our state variables (angle, position, etc) would be represented by a vector, and the system would be represented by matrix transformation. We used this linear system perspective as a means of controlling our robot car.

After we have a robot and can control its movement, the next challenge is path planning and manipulation. We want the robot to be able to perform our desired tasks! It almost sounds trivial in nature, but to a newly created robot, any simple task is a challenge. In this context, we must define every movement of the robot mathematically.

In order to do so, we'll need to reconsider how we think about our "state"—rather than viewing a state as a vector that can take on continuous values, we will instead define a set of discrete values that our variables can take on and consider the state as a node in a graph.

You'll recall from EECS16B that given a continuous-time system, we can sample or discretize in the time dimension and model some scenarios with a discrete-time system. Graph state-space models take another step of also discretizing in the spatial dimensions to have a finite number of possible states.

1.2 Contents

In this note, we first discuss how to formulate a **search problem**. In a search problem, we need to have a well defined space of states that we shall then search. To fully describe how to search these states requires a few key pieces of information that we will examine. We then describe the framework in which we analyze search algorithms. There are two categories of algorithms we will detail - **uninformed search** and **informed search** algorithms. We will look at a few fundamental techniques for both. Finally, we conclude by tying back to our keystone example of the **robotic planning**.

2 Search Problems

2.1 Search Intuition

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**.

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 a formulation is called a search problem.

2.2 Search Components

A **search problem** asks the question of how the agent can search through the possible states and make the best possible step towards its goal. In order to fully express the problem, we require four key 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 the successor state.

Start State - The initial state where the agent begins in.

Goal Test - A function that takes in a state and determines whether or not the goal has been achieved.

2.3 Search Main Idea

How do we utilize these four components? We start at the start state and then use the successor function to explore the state space. We must use our goal test at each state to see whether or not we are finished. This process is repeated iteratively at different successor states until we reach the goal.

The states are considered in a predetermined order called the **strategy**. This is what differentiates search algorithms from one another. Different strategies have different behaviors on certain search spaces, and thus choosing the correct search algorithm for the task is a critical task for the engineer. Finally, we return the path from start state to the goal, called the plan.

2.4 State Space Sizing

How large is a state space? We calculate the size of the state space using fundamental counting principles. Essentially, we multiply together, for each parameter of our state space, how many possible configurations they could have, resulting in the total size of the state space. Notice, this means that the size of the state space grows exponentially. The size of state space is often too large to save in memory, and thus successor functions are used to save only part of the space currently used in search and compute the rest when necessary.

As an example, consider a robot that has 4 degrees of freedom. It has joints at base, shoulder, elbow, and wrist. At the base and shoulder, it can be rotated by 15 degrees, and it can rotate all 360 degrees, resulting in 24 positions each. At elbow and wrist, it can rotate 60 degrees at a time, but it can only rotate 180 degrees total, which means there are only 3 positions that they can be in in a state.

How would we calculate the size of state space for this robot? We can follow the rules of counting. If we ignore the self collisions, we can simply multiply all of the degrees of freedom together, which results in 24 * 24 * 3 * 3 = 5184!

As you can probably expect, the size of the state space can get out of hand very quickly in more complex settings.

2.5 Search Algorithms

Now that we have formulated the search space, our robot needs to find a path through it. However, due to the sheer size, 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. Luckily, even relatively simple algorithms like breadth first search (BFS) allow us to find some solution, eventually. Before we explore several algorithms, let us first establish a general framework in which we will use to examine each algorithm.

First and foremost, the **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 GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    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
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Figure 1: Search Algorithm. Source: CS188

The general idea is that we explore nodes that we haven't seen before, where "exploring" a node means to perform the goal test then add all its children to the fringe. The queueing protocol that the fringe employs is central to an algorithm's strategy, and is often what differentiates the search algorithms from one another.

2.6 Algorithm Analysis

How do we evaluate the performance of our algorithms? We would like to detail and distinguish various algorithms from each other based on whether is actually feasible, if it can find the best solution, and how efficient it runs. 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.

In order to evaluate these properties, we also define some variables that will be employed in our analyses of each algorithm:

Depth - m

Branching factor - b (how many successors a typical node has)

Cost of optimal path - C^*

Minimum cost in optimal path - ε

3 Uninformed Search

The first type of search algorithm we will examine is the **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). Hopefully, BFS, DFS, and Dijkstra's are familiar to you from a companion course like CS61B. We review some of the key insights of the algorithms here.

3.1 Depth-First Search (DFS)

Depth First Search is a search algorithm that hones in on searching deep rather than wide. Since you have already seen it in CS61B, we will not explain it in detail.

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 $O(b * m)$.

3.2 Breadth-First Search (BFS)

Breadth First Search is a search algorithm that hones in on searching wide rather than deep. Since you have already seen it in CS61B, we will not explain it in detail.

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 node, 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. The maximum number of nodes of the same depth is $O(b * s)$ which can be reached in depth s.

3.3 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) is essentially Dijkstra's algorithm with a slightly modified end behavior - terminating when the end state is added into the fringe. Since you have already seen Dijkstra's in CS61B, we will not explain it in detail.

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 starting node to the current node. Priority queue ensures that the lowest cost node is prioritized by reordering itself after additions and removals.

Completeness - UCS is complete and always finds a solution since the shortest path to solution must exist in state space.

Optimality - UCS is optimal, provided that all edge weights are positive. It can be proved that if there is a shortest path to the goal node (there has to be if start and goal node are connected) then UCS finds the optimal solution. You'll see the rigorous proof if you take a class like CS 188.

Time and Space Complexity

Runtime - $O(b^{\frac{C^*}{\varepsilon}})$

Space Complexity - $O(b^{\frac{C^*}{\varepsilon}})$

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 results in a more complex runtime, the derivation of which is out of scope.

The key way this differs from Dijkstra's algorithm that you may have seen is that in Dijkstra's algorithm, we start off with ALL our nodes in the priority queue at a distance of infinity. For UCS, however, we only add nodes to our priority queue when they are visited. This avoids taking up unnecessary memory to store nodes that are never visited.

4 Informed Search

Uniform Cost Search is complete and optimal, but it can be fairly slow. Given some additional information, say, 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**.

These extra pieces of information are called **heuristics**. Heuristics are techniques that allow us to make estimations very quickly to speed up the process of our potentially slow, search algorithms, and thus are a central part of informed search algorithms. If we can compute the heuristic about the potential state, we can prioritize them to have much better runtime.

However, we cannot just use any old heuristic. Heuristics need to be a lower bound on the remaining distance for the informed search solution to be optimal. More specifically, a heuristic must be admissible for tree search and consistent for graph search, although this conceptual depth is out of scope for this class. This is a central topic that you may explore in CS188. Now, let us take a look at two fundamental informed search algorithms.

4.1 Greedy Search

Greedy search is a naive algorithm that employs the usage of only the heuristics, or what is known as "forward cost." Indeed, we do not even use the edge costs in this algorithm at all, and instead rely solely on our prior estimates of the edge costs through the notion of the heuristic.

Strategy - Select the lowest heuristic node from the fringe.

Fringe - Priority Queue. 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.

4.2 A* Search

A* is a search algorithm that essentially adds a heuristic to Dijkstra's algorithm. Since you have already seen it in CS61B, we will not explain it in detail.

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. You will prove this in later classes like CS188. Intuitively, an underestimate is optimistic in nature, and may try nodes that might disappoint in the end, but they will certainly find all the good opportunities. An overestimating heuristic is pessimistic in nature and may pass over some promising nodes and thus may miss the optimal solution for another solution.

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

Figure 2: Search Algorithms Summary

5 Robotic Planning

Now given our knowledge of how to formulate a search problem and how to solve it using various search algorithms, let us put it all together—applying our search algorithms to robotic path planning.

In the state-space representation of our robot, each node represents a configuration of our robot. We need to discretize our continuous space to create a search graph.

One method of discretizing our search graph is at systematic (fixed) intervals. For example, if a joint angle could be any value from 0 to 90, we could discretize in intervals of 15, meaning that for our simplified configuration space the only values that our joint can take on are 0, 15, 30, 45, 60, 75, or 90 degrees. This is a simple approach, however, it may often lead to graphs that miss good paths to the solutions. Thus, what's commonly used in practice are probabilistic, sampling-based methods to generate the search graph.

One sampling method is called a **Probabilistic Road Map (PRM)**, where we choose points in the state space uniformly at random, and remove the points that are in collision with obstacles. We'll specify a fixed value k, and we connect each of sampled points to its k nearest neighbors and again remove any edges that would collide with an obstacle. This can generate a graph that is a useful model of our state space, then we can run a graph search on this and generate a sufficiently short path to our goal state. The picture below shows a configuration space with obstacles (blacked out) and the search graph generated by PRM.

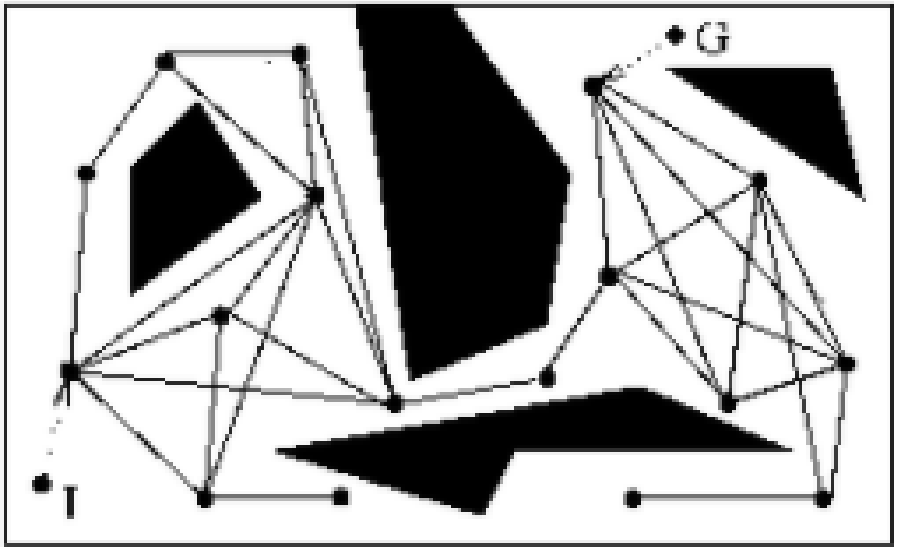

Figure 3: PRM Search Graph

Another sampling method is **Rapidly Exploring Random Tree** (RRT), which has a bias towards unexplored regions when creating new edges and exploring them. It is probabilistically complete, but the paths that are generated are often very suboptimal, as it will tend to zigzag alot (kind of like how modern American suburban neighborhoods with dead ends and cul-de-sacs tend to be hard to navigate). The figure below shows an example of a graph generated by a rapidly exploring random tree.

Figure 4: RRT Search Graph

A variant of this algorithm, called **RRT***, also rewires nodes in the graph at every step, choosing nearest neighbors to rewire. This allows for much more optimal solutions to be found, and is used in state-of-the-art robotic path planning algorithms even today.

References

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

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

https://ucb-ee106.github.io/106a-fa20site/assets/lec/L17-Motion-Planning.pdf