CS 337, Fall 2023
Search

Scribes*: Shreya Tiwari, Divyansh Singhal, Nikhil Verma, Shubham Roy,
Addanki Sanjeev Varma, Arpana Prajapati, Magham Dipen Anjan, Motupalli Yaswanth
Edited by: Tejpalsingh Siledar

October 26, 2023

1 Introduction

After having finished the ML part of the course, we are now starting with AI.
First what and why AI? It's nothing but a culmination of research aimed at creating an intelligent system that is capable of mimicking human cognitive functions. While ML focuses on developing algorithms that can learn from data, AI has a broader range of capabilities, including problem-solving, natural language modeling, perception, and decision-making. All in all it aims to create a system that emulates human intelligence and adapts to changes, without being explicitly programmed for each task. At its core, the aim is to build intelligent agent that can perceive their environment, reason about it, plan action, and ultimately achieve specific goals. Whereas ML, being a subset of AI focuses on pattern recognition and statistical modeling.

So, we need a thing that perceives its environment and takes actions to maximize the chances of achieving its goals, here we call it an agent. Now the primary goal of an agent is to act **rationally**, which means making decisions and taking actions that are expected to lead to the best outcomes given the available information and its objectives. In short, they are **problem-solving agents**. They explore the possibilities and find optimal or satisfactory solutions to a wide range of problems.

In the realm of problem-solving agents and AI, two fundamental models, namely **reflex-based** and **state-based** models, illustrate different approaches to handling complex tasks and decision-making.

(a) **Reflex based models**: These models simply perform a fixed sequence of computations on the input. This includes most of the models used in machine learning. The computations involved are feed-forward, the agent doesn't backtrack and consider alternative computations. They are particularly suitable for tasks that require quick and instinctive responses.

(b) **State-based models**: These types of models contain state information (representation of the environment), actions, goals, and the transitions between states that are triggered by certain actions. These models have the ability to plan ahead unlike reflex-based models which have expertise in pattern inference from the given data which in turn helps to engage in more complex and dynamic problem-solving.

Searching for a solution is a fundamental concept in AI and a key component of problem-solving agents, it encompasses exploring a space of possible states or actions to find the best solution to a given problem.

*All scribes contributed equally to this document.

Searching algorithms play a critical role here it aims to solve tasks like route planning puzzle-solving and optimizing decision-making processes.

In the section further, we will look a bit deeper into searching and, kinds of searching. We describe some general-purpose searching that can be used to solve problems, and then we will see different **uniformed search algorithms**- these algorithms have no information about the problem other than its definition. While some of these algorithms can solve any solvable problem, none can do so efficiently. We also brief on the contrast, i.e., **informed search algorithms** can perform well when provided guidance on where to look for solutions.

2 Search Problem

A Search problem is typically used to frame a specific task or puzzle in a way that can be systematically solved or explored by search algorithms. It operates under the assumption that the environment is fully known.

A Search problem can be defined using :

1. **State space (S)**: Set that encompasses all possible states or configurations that a particular problem or system can exist in.
2. **Start state (S_0)**: The starting state(s) from which the search process begins.
3. **Goal state (G)**: The state(s) where the search algorithm seeks to reach.
4. **Transition function** : A function that maps a state **s** and state **s'** on an action **a** .
5. **Path Cost (c(s, a, s'))**: A function that assigns a cost to each action, indicating the cost of transitioning from one state s to another state s' by applying a specific action a.

A **Solution** to a search problem is an action sequence that leads from the initial state to a goal state. Solution quality is often measured by the path cost function, and an **Optimal solution** has the lowest path cost among all possible solutions to the problem.

Note: Without Loss of Generality we assume the start state and goal state to contain only a single state

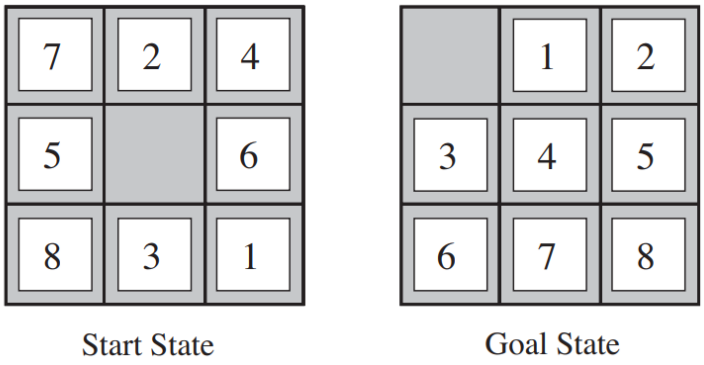


Figure 1: A typical instance of the puzzle

An example to illustrate a search problem is the **8-tile-puzzle**, an instance of which is shown in Figure 1, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space (A Transition). The objective is to reach a specified goal state, such as the one shown

on the right of the figure from a start state as one on the left.

The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure 1, the resulting state has the 5 and the blank switched. A reasonable path cost function would consider the cost of each transition in the path to be 1, so the path cost is the number of steps in the path.

This framework is commonly used in various problem-solving scenarios, including grid-based puzzles, game AI, and automated planning, where understanding how actions affect the state and optimizing the path cost are key components of the problem-solving process.

3 Understanding Search Problems

All search problems consist of 3 classes of states, the **explored states**, the **unexplored states**, and the **frontier** as shown in Figure 2. Frontier separates the explored and unexplored states. In every search strategy, we pick a state from the frontier based on some strategy and expand it to child states. Consequently, we move the chosen frontier state to the explored class, and all the child states are moved to the frontier.

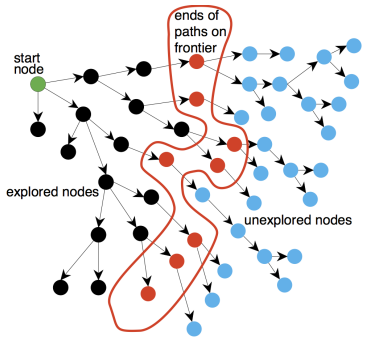


Figure 2: All search problems consist of 3 classes of states (mentioned in the image as nodes), the explored states, the unexplored states, and the frontier

These sets are defined formally as follows.

1. **Explored Set**: It contains a set of states that have been fully examined and processed by the search algorithm. These states have typically been checked against the goal test, and the search algorithm has determined whether they are part of a solution or not.

2. **Frontier**: Also known as the **Open set**, is a set of states that are on the "edge" of the current exploration. These states are candidates for further exploration, as they represent the states that have been generated by applying operators to the states in the explored set but have not yet been fully examined.

3. **Unexplored Set**: It contains the states that have not yet been examined by the search algorithm. These states are typically on the frontier initially and may also include the start state.

4 Uninformed Search Strategies

Uninformed search (also called **blind search**) means that the strategies have no additional information about states beyond that provided in the problem definition. Uninformed search strategies are limited to generating successors and discerning a goal state from a non-goal state without leveraging any specific

information about the state space or the nature of the goal state. These strategies do not consider or prioritize the goal state's characteristics or properties during their exploration of the state space. All search strategies are distinguished by the order in which nodes are expanded.

> **Note**: The key difference between the Informed and Uninformed Search strategy is that the Informed search uses heuristic information to guide the search process towards the goal, leading to more efficient exploration. Uninformed search, on the other hand, explores the search space without using any additional knowledge of the goal state, which can be less efficient in complex domains.

4.1 Breadth-First Search

Frontier: First-In-First-Out queue
Strategy: Expand the shallowest (open) node

Breadth-First Search (**BFS**) is a search strategy that explores a problem's state space in a broad and systematic manner. BFS expands the shallowest unexplored states first, ensuring that it explores all states at a one-step distance from the initial state before moving on to states that are at a two-step distance and so on. The frontier in BFS is crucial for maintaining this breadth-first exploration pattern, as it keeps track of the states that need to be examined in a **first-in, first-out (FIFO queue)** manner.

4.2 Depth-First Search

Frontier: Last-In-First-Out state
Strategy: Expand the deepest node first

Depth-First Search (**DFS**) is a search strategy that explores a problem's state space by diving as deeply as possible into a specific branch of the search tree before backtracking to explore other branches. In DFS, the frontier is managed as a stack, where states are pushed onto the stack as they are encountered and popped off when backtracking. The **stack-based (LIFO) frontier** in DFS allows for this depth-first exploration strategy by keeping track of the states to explore next and enabling backtracking when necessary.

4.3 Uniform-Cost Search

Frontier: Priority queue (Cumulative cost)
Strategy: Expand the lowest cost node

Uniform-Cost Search (**UCS**) is a search strategy that explores a problem's state space by considering the cost of each path and selecting the path with the **minimum cumulative cost** at each step. In UCS, the frontier is managed as a **priority queue** (least cost states from the start state are at first), where states are ordered based on their path cost, with the least costly states explored first.

UCS is an optimal search algorithm, meaning that it guarantees finding the lowest-cost solution or path when costs are associated with transitions between states. The key invariant of the UCS algorithm can be stated as follows:

Key invariant for UCS

At any given point during the search, the priority queue (or frontier) in UCS contains the states in a non-decreasing order of their path costs. This invariant means that the state with the lowest path cost is always at the front of the priority queue, and the algorithm selects and explores it first. As UCS progresses, it expands states in order of increasing path cost, guaranteeing that the first goal state encountered will have the lowest possible cost. By maintaining this, UCS ensures that it explores the state space systematically and optimally, providing the lowest-cost solution when cost considerations are involved.

5 Uniform Cost Search

5.1 Uniform Cost Search Algorithm

Algorithm 1 UCS Algorithm

Input: *start*: The starting state, $cost(s)$: cost of reaching state s from the *start*, $successor(s, a)$: next state reached taking action a from state s,
Output: a solution or failure
$explored \leftarrow$ empty set
$frontier \leftarrow$ priority queue with *start* as the only element
cost(A) $\leftarrow 0$
while do
  if *frontier is empty* **then**
    return *fail*
  $s \leftarrow frontier$.pop()
  $p \leftarrow$ cost(s)
  //remove s with lowest cost p from the *frontier*
  $explored \leftarrow explored \cup \{s\}$
  if *isGoal(s)* **then**
    return p *and the solution-path*
  foreach *action* $a \in Actions(s)$ **do**
    $s' \leftarrow$ successor(s, a)
    if s' *is not in explored or* $frontier$ **then**
      $frontier \leftarrow frontier \cup \{s'\}$
      cost(s') $\leftarrow p +$ cost(s, a, s')
    else if s' *is in* $frontier$ *with higher path cost* **then**
      //replace that with the current cost of s'
      cost(s') $\leftarrow p +$ cost(s, a, s')

Few points to note:

- UCS assumes the costs to be all non-negative.
- The algorithm fails to find an optimal path when there is no path from the start state to the goal state.

Let's take an example of four states - A, B, C, D with actions and associated costs as given in the figure below. Let A be the starting state and D be the goal state.

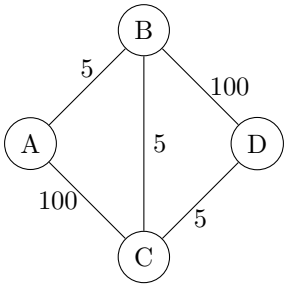
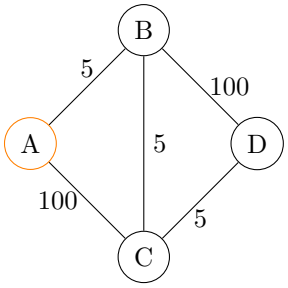
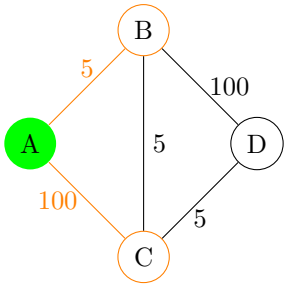


Figure 3: An example state graph

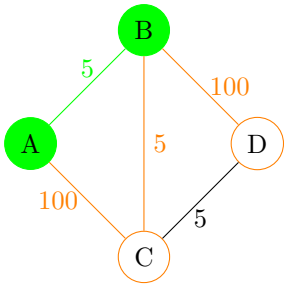
Then UCS will find the optimal path by exploring the states according to the steps given below -



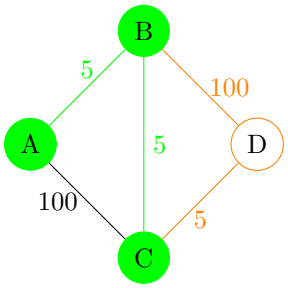
(a)



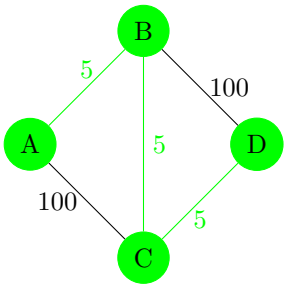
(b)



(c)



(d)



(e)

1. Initially the set of explored states is empty and state A is in the frontier.
   frontier = {A(0)}, explored = {}
2. Remove A from the frontier and put it in explored. Consider all the successor states of A, which are B and C. Put these in the frontier and update their costs.
   frontier = {B(5), C(100)}, explored = {A(0)}
3. Now remove B from the frontier and put it in explored. Consider its successor states (C and D), and find their path-costs through B. We are not considering A since it's already been explored. Put D in frontier with the cost being 105 and update the cost of C to 10.
   frontier = {C(10), D(105)}, explored = {A(0), B(5)}
4. Remove C from the frontier and put it in explored. Consider its successor states (D), find the cost to D through C which is 15 which is lesser than its previous cost therefore, update its cost. We are not

considering A and B since these are already explored.
frontier = {D(15)}, explored = {A(0), B(5), C(10)}

5. Finally pop D frontier and put it in explored. We reached the goal state D with cost 15 and path as $A \to B \to C \to D$
frontier = {}, explored = {A(0), B(5), C(10), D(15)}

> Note that frontier is a priority queue with lower costs states getting higher priority. Also, note that the algorithm does not stop as soon as the goal state comes inside the frontier. It stops only when it has popped the goal state from the frontier i.e., it has explored the goal state to get the optimal cost path.

5.2 Main limitations of UCS

UCS is a powerful algorithm for finding the shortest path between two states in a graph. However, it has a few limitations:

1. UCS is agnostic to the goal state, meaning that it expands states in the order of their cost, regardless of their distance to the goal state. This can lead to UCS exploring states that are far from the goal state, even if there are shorter paths to the goal state from other states.

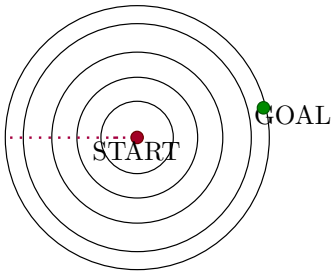


Figure 5: The points to the left are wastefully explored in UCS

2. UCS can also be computationally expensive, as it maintains a frontier of all states that have been generated but not yet explored. This frontier can grow quite large for large graphs, leading to high computational overhead.

6 Informed Search Strategies

In this search strategy, the selection of a frontier state depends on a heuristic estimate of the proximity to the goal state. These approaches rely on information that may not be explicitly provided in the problem statement but is assumed or derived from prior experience and estimations.

Best-first searches are search strategies in which a node is selected for expansion based on the evaluation function **f(n)**. The evaluation function is interpreted as a cost estimate, so the node with the lowest evaluation is expanded first. For Uniform Cost Search, **f(n) = g(n)**, where g(n) is the lowest cost from the start state to n. The choice of **f** determines the search strategy. Most best-first algorithms include as a component of **f** a heuristic function, denoted h(n). Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. Below are two of the best search strategies.

6.1 Greedy Best First Search Algorithm

For this search strategy, the evaluation function is the heuristic function **h(n)**, which is defined as a heuristic estimate of the distance from the n to the goal state. Here, **f(n) = h(n)**.

Greedy Best First search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. This shows why the algorithm is called "greedy"—at each step it tries to get as close to the goal as it can. However, this strategy is not guaranteed to give an optimal solution.

6.2 A* Search Algorithm

In the context of the A* search algorithm, the function **f(n) = g(n) + h(n)**.
f(n) is defined as the sum of two crucial components:

(a) **g(n)**, which represents the actual path cost from the starting state to state **n**, and

(b) **h(n)**, which estimates the remaining cost to reach the goal from state **n**.

This combination of **g(n)** and **h(n)** is what makes A* highly effective. It strikes a balance between considering the cost incurred so far and the estimated cost to the goal, ensuring that the algorithm explores paths that are both promising and efficient.

A* search algorithm is guaranteed to find the optimal solution if **h(n)** is a consistent heuristic.

Consistent Heuristic : A heuristic h is said to be consistent if it satisfies

$$h(S) \leq h(S') + cost(S, a, S')$$

where $cost(S, a, S')$ gives the cost associated in transitioning from a state S to S' with some action a.
A consistent heuristic is also **admissible**. An admissible heuristic is one that never overestimates the cost of reaching the goal. That is, for an admissible heuristic h, $h(n) \leq h^*(n)$ for all values of n, where h^* is the true cost function.

[0]You can explore the demonstration of search algorithms by visiting the following link.