Version 2022.1

Lecture 2: Search Problems

Introduction to Artificial Intelligence

Chandra Gummaluru *University of Toronto*

Introduction

In the previous lecture, we framed the problem we are trying to solve as a game. In this lecture, we continue by considering a special type of game called a search problem.

Search Problems

A search problem is a game with the following assumptions:

- There is only one-player, i.e., $N = 1$, and
- all transitions to terminal states have the same utility, i.e., $R(s_1, a^{(1)}, s_1') = R(s_2, a^{(2)}, s_2')$ for any $s_1', s_2' \in \mathcal{T}$
- all transitions to non-terminal states yield zero utility, i.e., $R(s, a, s') = 0$ for any $s' \notin \mathcal{T}$

Example (The Slider Puzzle) : The N-tile Slider Puzzle consists of an N by N board with tiles numbered $1, \ldots, N-1$, that can slide horizontally/vertically. The goal is to move the tiles so that they are in order.

s_0

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7	8	

$s \in \mathcal{G}$

Figure 1: A 3-tile Slider Puzzle (left) and its solution (right).

The General Search Algorithm

The most obvious way to search is as follows:

1. Start with some initial state, $s \leftarrow s_0$, and the initial path $p_0 := \langle s_0 \rangle$.
2. During each iteration i, check if the current path, $p_i = \langle s_0, \ldots, s_i \rangle$ is goal terminating, i.e., check if $s_i \in \mathcal{T}$:
   (a) if so, return p_i as the solution,
   (b) otherwise, perform an action, $a \in \mathcal{A}(s_i)$, resulting in a new state, $s_{i+1} = a(s_i)$, let $p_{i+1} = \langle s_0, \ldots, s_{i+1} \rangle$ and proceed to iteration $i + 1$.

Example (The Slider Puzzle) : In the N-tile Slider Puzzle, we define the set of states, $\mathcal{S}$ to be all possible board configurations, and $\mathcal{G}$, to be the board configuration in which the tiles are in the correct order.

Given some initial board configuration, we choose the corresponding state in $\mathcal{S}$ as s_0. The set of valid actions $\mathcal{A}(s)$ from a given state s involving moving a tile either up, down, left, or right (if possible). We keep applying valid actions until a goal state is found. One possible path is shown below:

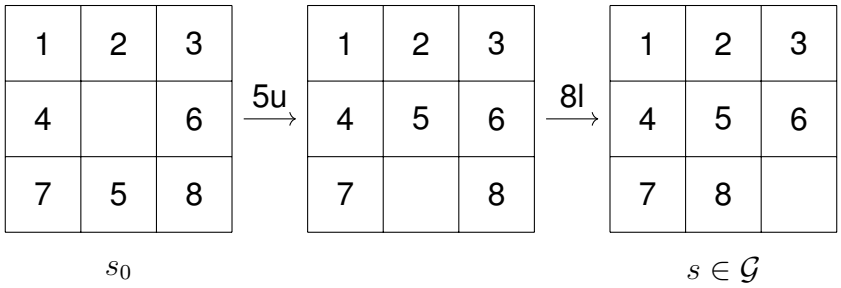

Figure 2: A possible path, $p = \langle s_0, s_1, s_2 \rangle$, where $s_2 \in \mathcal{G}$ for the 3-tile Slider Puzzle.

We now formalize the aforementioned approach.

Algorithm 1 The General Search Algorithm

```
1: 𝒪 ← {(⟨s_0⟩, 0)}                                   ▷ initialize 𝒪 with s_0
2: procedure search(𝒪)
3:     if 𝒪 = ∅ then                                   ▷ the search failed to find a goal
4:         return NULL
5:     n ← remove(𝒪)
6:     if final(p) ∈ 𝒯 then                            ▷ check if the final state in p is a terminal state
7:         return n
8:     𝒪 ← 𝒪 ∪ successors(p)                           ▷ add p's successors to 𝒪
9:     search(𝒪)
```

In Alg. **??**, the data-structure, $\mathcal{O}$, is called the **open set**; it contains the states that have been discovered but unexplored. We do not enforce any specific implementation of the *remove* function on Ln. 5, only that it removes *some* path from $\mathcal{O}$. When actually implementing *remove*, one may use various properties of the paths on $\mathcal{O}$ (e.g., their costs) to determine which one to remove next. As such, it is often useful to explicitly keep track of such properties using a separate data-structure. We will refer to this data-structure as a **node**. For now, we can view a node as containing; a path, and its cumulative cost (e.g., $n = (\langle s_0, \ldots, s_i \rangle, 7)$).

Example (The Slider Puzzle) : The (partial) tree of possible paths for the 3-tile Slider Puzzle starting from some arbitrary initial state is shown below:

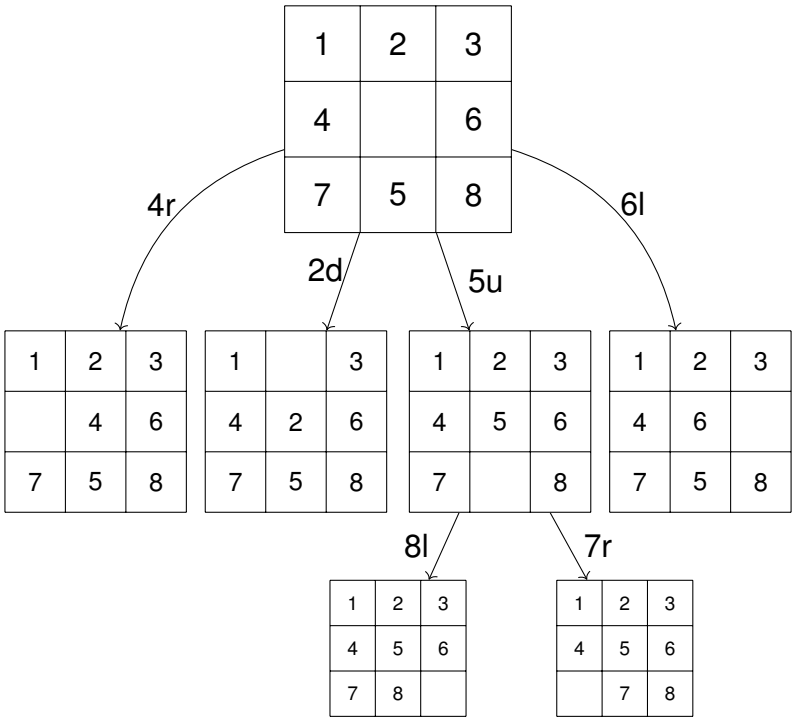

Figure 3: An (incomplete) tree of possible paths for the 3-tile Slider puzzle.

The order in which the paths are explored, i.e., the implementation of *remove* on Ln. 5 of Alg. **??** is irrelevant in the sense that it will not change whether a solution is found or not. However, it may change the properties of the solution found, and that of the algorithm itself.

Properties of the General Search Algorithm

There are a few properties of interest:

- *the time complexity*: the order of the number of states that must be generated before the algorithm terminates.

- *the space complexity*: the order of the maximum number of states that exist on the open at any given iteration.
- *completeness*: whether the algorithm always finds a solution assuming at least one exists.
- *optimality*: whether the algorithm finds the optimal solution if multiple exists.

The ideal search algorithm is complete, optimal, and minimizes the time and space complexity, but we will see that this is difficult to achieve. To reduce the time/space complexity, we may want to prune certain paths. However, we may be concerned that doing so might cause our algorithm to no longer be optimal or even complete. Fortunately, we can make a few modifications to Alg. **??** to reduce the time/space complexity if a few assumptions hold.

Intra-Path Cycle Checking

This technique relies on the following theorem.

Theorem 0.1 (*Test*)

If all actions have non-negative costs, i.e., $C(s, a, s') \geq 0, \forall s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$, then there exists an acyclic path that is the optimal solution.

Proof : Suppose there exists a solution path, $p = \langle a^{(1)}, \ldots, a^{(n)} \rangle$ that yields a cyclic goal-terminating state sequence, $\langle s_0, \ldots, s_n \rangle$, where $s_n \in \mathcal{T}$. The cost of p is

$$C(p) = \sum_{k=1}^{n} C(s_{k-1}, a^{(k)}, s_k).$$

Since the state sequence is cyclic, by definition, there exists an $j < i$ where $0 \leq i < j \leq n$, such that $s_j = s_i$. Therefore, $a^{(j)}(s_{j-1}) = s_i$, and so $p' = \langle a^{(1)}, \ldots, a^{(j)}, a^{(i+1)}, \ldots, a^{(n)} \rangle$ is also a solution path; the resulting state sequence is $\langle s_0, \ldots, s_{j-1}, s_i, \ldots, s_n \rangle$, and its cost

$$C(p') = \sum_{k=1}^{j} C(s_{k-1}, a^{(k)}, s_k) + \sum_{k=i+1}^{n} C(s_{k-1}, a^{(k)}, s_k).$$

It follows that

$$C(p') - C(p) = \sum_{k=j+1}^{i} \underbrace{C(s_{k-1}, a^{(k)}, s_k)}_{\geq 0} \geq 0.$$

Hence p' is at least as good as p. ■

Thm. **??** allows us to modify Alg. **??** so that whenever we discover a cyclic path, we need not expand it (since it cannot be optimal).

Algorithm 2 The General Search Algorithm (w/ Intra-Path Checking)

```
1:  O ← {(⟨s_0⟩, 0)}                                   ▷ initialize O with s_0
2:  procedure search(O)
3:      if O = ∅ then                                  ▷ the search failed to find a goal
4:          return NULL
5:      n ← remove(O)
6:      if final(p) ∈ T then                           ▷ check if the final state in p is a terminal state
7:          return p
8:      if acyclic(n.p) then
9:          O ← O ∪ successors(n)                      ▷ add p's successors to O
10:         search(O)
11:     return
```

Inter-Path Cycle Checking

Suppose *remove* is implemented such that the first time Alg. **??** discovers a path to a given state, s, it has found the optimal path to s. By construction, we can then modify Alg. **??** so that whenever another path to s is discovered, we need not expand it (since it cannot be optimal).

Algorithm 3 The General Search Algorithm (w/ Inter-Path Checking)

```
1:  O ← {(⟨s_0⟩, 0)}                                   ▷ initialize O with s_0
2:  C ← ∅
3:  procedure search(O)
4:      if O = ∅ then                                  ▷ the search failed to find a goal
5:          return NULL
6:      n ← remove(O)
7:      if final(p) ∈ T then                           ▷ check if the final state in p is a terminal state
8:          return p
9:      if n ∉ C then
10:         C ← C ∪ {p}                                ▷ add p to C
11:         O ← O ∪ successors(n)                      ▷ add n's successors to O
12:         search(O)
13:     return
```

In Alg. **??**, the data-structure, $\mathcal{C}$, is called the **closed set**; it contains the states that have been discovered and explored.