CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 3: August 26

Lecturers: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

3.1 Recap of Week 2

3.1.1 BFS

BFS is not optimal. How do we fix it?

1. We can replace the queue used by our frontier F in our BFS algorithm with a priority queue.
    - The nodes will be ordered according to their cost from the starting node.
    - When node u is popped from F, it will be the minimum cost node that is unexplored.
2. We should not mark the child nodes as "explored" to early in our BFS algorithm.
    - By marking them as "explored" early on, we will not update/relax their weights after going through them once more from another path.

3.2 Uninformed Search (continued)

3.2.1 Uniform Cost Search

After going through these modifications, we obtain a new algorithm, which we call **uniform cost search (UCS)**. Our modified algorithm will look like:

Algorithm 1 Uniform Cost Search

```
procedure UniformCostSearch($u$)
    $F \leftarrow$ PriorityQueue($u$)
    $E \leftarrow [u]$                                    ▷ a dict
    $\hat{g}[u] = 0$              ▷ keeps track of min. path cost of reaching $u$ discovered so far
    while $F$ is not empty do
        $u \leftarrow F$.pop()
        if GoalTest($u$) then
            return path($u$)
        end if
        $E$.add($u$)
        for all children $v$ of $u$ do
            if $v \notin E$ then
                if $v \in F$ then
                    $\hat{g}[v] \leftarrow \min(\hat{g}[v], \hat{g}[u] + c(u,v))$
                else
                    $F$.push($v$)
                    $\hat{g}[v] \leftarrow \hat{g}[v] + c(u,v)$
                end if
            end if
        end for
    end while
    return FAILURE
end procedure
```

Remark: UCS is different from Dijkstra in the sense that Dijkstra initializes $\hat{g}$ for all the nodes.

- However, we may have infinitely many nodes in our graph.
- This would cause us to be stuck in the initialization step, and our algorithm would not work.

3.2.2 Properties of UCS

Exercise: Is UCS complete?

Proof:
UCS is only complete when there does not exist an infinitely long chain of edges with weights 0.

Suppose we have a graph containing edges with weight 0, as seen in the following image:

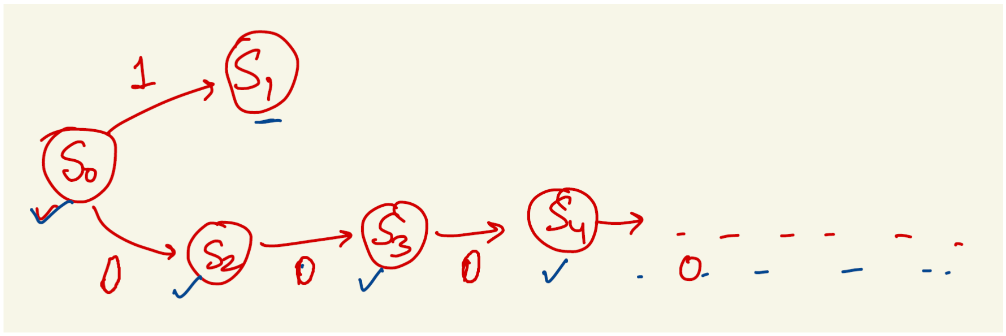

Figure 3.1: Graph with edges of weight 0

UCS would continue down this chain $(s_2, s_3, \ldots)$ containing edges of weight 0. If this chain is infinitely long, UCS would never explore s_1 in finite time.

However, suppose that all of the edges in this chain $c(s_0, s_2), c(s_2, s_3), \ldots$ have nonzero weights. That is, all the edges $c(s_0, s_2), c(s_2, s_3), \ldots$ have some weight $\epsilon > 0$. Then, at some point down the chain, the sum of the edge weights down this chain would exceed $c(s_0, s_1)$. This would then cause UCS to be able to explore s_1 and reach the goal state. Hence, UCS is complete in this case. $\square$

Remark: An algorithm can still be complete even though there are infinite states. This is because we do not necessarily need to traverse through all the states in order to get to our goal.

Exercise: Is UCS optimal?

UCS is optimal if we can ensure that when we are performing the goal test, we have found the optimal path.

Proof:
We will prove a more general claim: when we pop a node u from F, we would have found the optimal path to u. We shall first define our notations as such:

- $g(u)$: actual min. path cost from initial node to node u.
- $\hat{g}(u)$: min. path cost from initial node to node u, based on our current knowledge of the graph.
- $\hat{g}_{\text{pop}}(u)$: the value of $\hat{g}(u)$ when we pop u from F.

Thus, we can remodel our claim into a mathematical equation, i.e. $\hat{g}_{\text{pop}}(u) = g(u)$.

Let us now consider the optimal path from the initial node s_0 to node u, say $(s_0, s_1, \ldots, s_k, u)$. We will prove our claim by induction on this path.

- Base case: $\hat{g}_{\text{pop}}(s_0) = g(s_0) = 0$. This is clearly true.

- Inductive hypothesis: For all $\{s_0, s_1, \ldots, s_k\}$, we assume that $\hat{g}_{\text{pop}}(s_i) = g(s_i)$.
- Finally,
  1. We have $\hat{g}_{\text{pop}}(u) \geq g(u)$ for all $u \in$ state space. This is clearly true since we will have $\hat{g}_{\text{pop}}(u) = g(u)$ if we have already found the optimal path to u while u is still in the priority queue, and $\hat{g}_{\text{pop}}(u) > g(u)$ if the optimal path to u has yet to be found.
  2. Furthermore, when we pop s_k,

$$\begin{aligned}
\hat{g}_{\text{pop}}(u) &= \min\{\hat{g}(u), \hat{g}_{\text{pop}}(s_k) + c(s_k, u)\} \\
&\leq \hat{g}_{\text{pop}}(s_k) + c(s_k, u) \\
&= g(s_k) + c(s_k, u) && \text{(from IH)} \\
&= g(u)
\end{aligned}$$

  3. From (1) and (2), we get $\hat{g}_{\text{pop}}(u) = g(u)$. This proves our initial claim, and thus UCS is optimal.

□

Exercise: Time and space complexities of UCS?

Solution:
Time complexity: $O(b^{d+1})$, where

- b is the branching factor of the recursion tree, and
- d is the depth of the recursion tree (from root to goal state).

Space complexity: $O(b^{d+1})$ □

Remark: Is there any way to use less space than $(O(b^{d+1}))$?
Yes, by using UCS with tree search.

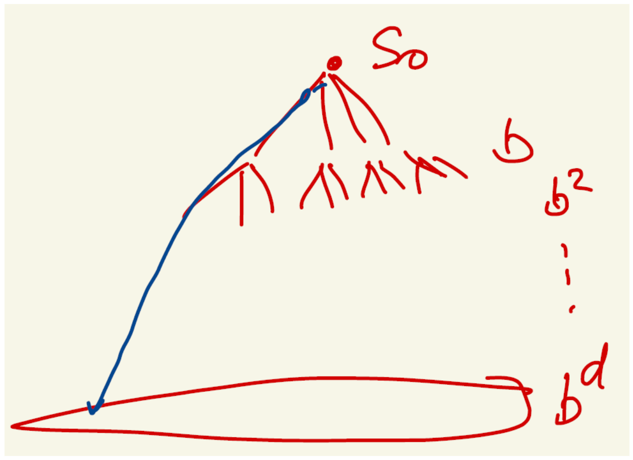

Figure 3.2: Visual representation of tree search

This way, we will only have to use $O(bd)$ space to keep track of the frontier. We can implement this by replacing the queue in Algorithm 1 with a stack. However, there will be tradeoffs if we decide to use UCS with tree search instead of UCS with graph search. For instance, we will lose optimality and completeness.

- With tree search, the graph may keep going down a single path, and never get to the goal.

Remark: There will always be tradeoffs.

- If we want to ensure completeness, the space complexity of our algorithm will be high.
- If we want to minimize our space consumption, we would have to sacrifice completeness.
  - However, on some occasions, it is perfectly fine to implement an algorithm that is not complete (e.g. when there are finite states).

3.2.3 BFS, DFS, and UCS

So far, we have gone through various algorithms which explore our state space without any additional information provided to them (i.e. uninformed search algorithms). Sometimes, these algorithms may perform redundant moves.

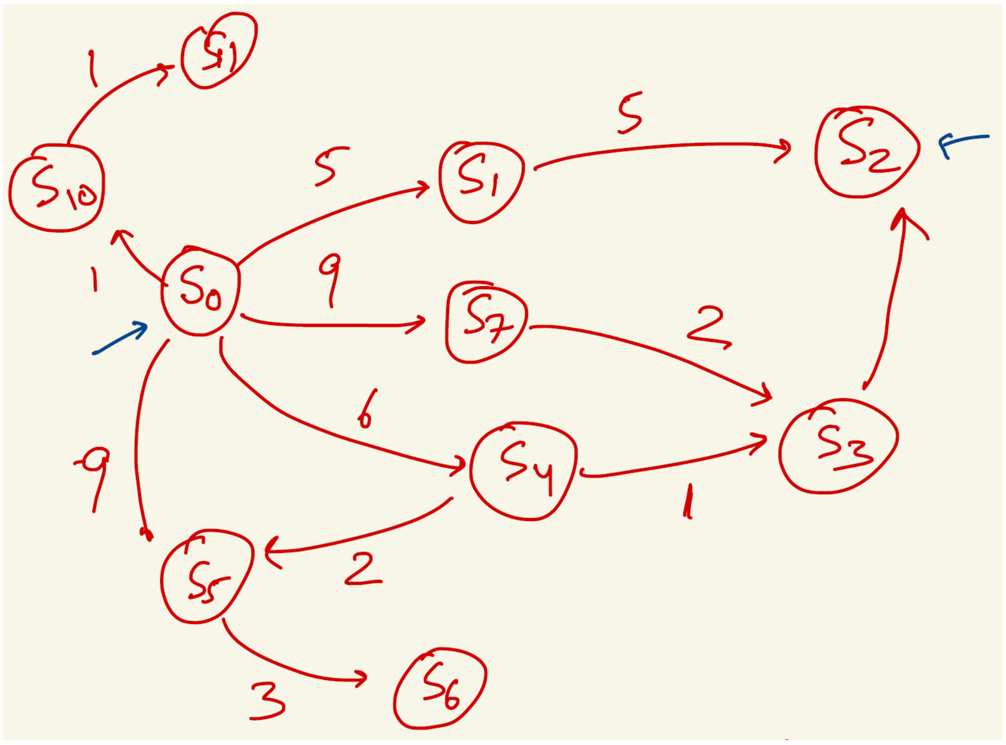

Figure 3.3: Graph with initial state s_0 and goal state s_2

For instance, when we run UCS on this graph, UCS will first traverse $s_0 \to s_{10} \to s_1$, even though the state s_1 is not along the path from s_0 to s_2. If we have some information (e.g. heuristics), our algorithms might be able to make better decisions.

3.3 Informed Search

3.3.1 Heuristics

We shall begin by defining what a heuristic is. Heuristics will be useful in helping us make more informed decisions in our search process.

Definition 3.1 (Heuristic).

A heuristic, denoted as $h(u)$, is the estimated cost from a state u to the goal state.

3.3.2 A^* search

A^* search is basically UCS, but at each iteration, we will pop the node with the smallest $\hat{f}(u) = \hat{g}(u) + h(u)$ instead of the node with the smallest $\hat{g}(u)$. The algorithm for A^* is as follows:

Algorithm 2 A^* Search

```
1: procedure A*Search($u$)
2:     $F \leftarrow$ PriorityQueue($u$)                ▷ Prioirity queue should be implemented with $\hat{f}$
3:     $E \leftarrow [u]$                                ▷ a dict
4:     $\hat{g}[u] = 0$
5:     while $F$ is not empty do
6:         $u \leftarrow F$.pop()
7:         if GoalTest($u$) then
8:             return path($u$)
9:         end if
10:        $E$.add($u$)
11:        for all children $v$ of $u$ do
12:            if $v \notin E$ then
13:                if $v \in F$ then
14:                    $\hat{g}[v] \leftarrow \min(\hat{g}[v], \hat{g}[u] + c(u, v))$
15:                    $\hat{f}[v] \leftarrow \hat{g}[v] + h(v)$
16:                else
17:                    $F$.push($v$)
18:                    $\hat{g}[v] \leftarrow \hat{g}[v] + c(u, v)$
19:                    $\hat{f}[v] \leftarrow \hat{g}[v] + h(v)$
20:                end if
21:            end if
22:        end for
23:    end while
24:    return FAILURE
25: end procedure
```

Exercise: Is A^* optimal?

If we can ensure that for an optimal path $s_0, s_1, \ldots, s_k, u$:

$$\hat{f}_{\text{pop}}(s_0) \leq \hat{f}_{\text{pop}}(s_1) \leq \ldots \leq \hat{f}_{\text{pop}}(u) \tag{3.1}$$

and

$$\hat{f}_{\text{pop}}(s_i) = f(s_i) \tag{3.2}$$

where $f(s_i) = g(s_i) + h(s_i)$, then we would have shown that A^* is optimal.

Proof:
If $f(s_0) \leq f(s_1) \leq \ldots \leq f(s_k) \leq f(u)$ and equation 3.2 both hold, then equation 3.1 would hold, and thus we will have shown that A^* is optimal.

Let us pick some arbitrary s_i, where $f(s_i) \leq f(s_{i+1})$. This would imply that:

$$\begin{aligned}
& g(s_i) + h(s_i) \leq g(s_{i+1}) + h(s_{i+1}) \\
& \quad \Longrightarrow h(s_i) \leq g(s_{i+1}) - g(s_i) + h(s_{i+1}) \\
& \quad \Longrightarrow h(s_i) \leq c(s_i, s_{i+1}) + h(s_{i+1}) && \text{(this implies a \textbf{consistent heuristic})} \\
& \qquad\qquad\quad \leq c(s_i, s_{i+1}) + c(s_{i+1}, s_{i+2}) + h(s_{i+2}) \\
& \qquad\qquad\quad \vdots \\
& \qquad\qquad\quad \leq c(s_i, s_{i+1}) + \ldots + c(s_k, u) + h(u) && \text{(where } h(u) = 0\text{)} \\
& \quad \Longrightarrow h(s_i) \leq OPT(s_i) && \text{(this implies a \textbf{admissible heuristic})}
\end{aligned}$$

where $OPT(s_i)$ refers to the optimal path cost from s_i to the goal state.

Hence, if we are able to obtain a heuristic function h that satisfies the properties of **consistency** and **admissibility**, then our algorithm will be optimal.

□

Exercise: The value of $f(s_i)$ along *any path* from s_0 to u is:

Solution:
Neither increasing nor decreasing. However, the *optimal path* from s_0 to u will have nondecreasing values of $f(s_i)$. □