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

Lecture 4: September 2

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

4.1 Recap of Week 3

4.1.1 BFS, UCS, and A^*

Last week, we began with discussing BFS and UCS, and we came to the conclusion that we will be able to do better than these uninformed search algorithms if we are able to get some information about the goal. This led us to the discussion of A^* search.

4.1.2 Consistency and admissibility

We have shown that if $f(s_i) \leq f(s_{i+1})$, where:

- $f(s_i) = g(s_i) + h(s_i)$,
- $g(s_i)$ represents the minimum path cost from the starting node s_0 to node s_i, and
- $h(s_i)$ represents the estimated cost from node s_i to the goal node.

then A^* will be optimal.

In our proof, we have also discussed the conditions for consistent and admissible heuristics. Assuming that the optimal path from the starting node s_0 to some goal node s_g is $(s_0, s_1, \ldots, s_i, s_{i+1}, \ldots, s_g)$,

- A heuristic is **consistent** if $h(s_i) \leq c(s_i, s_{i+1}) + h(s_{i+1})$, where $c(s_i, s_{i+1})$ represents the cost to get from s_i to s_{i+1}.
- Additionally, if $h(s_g) = 0$, then $h(s_i) \leq OPT(s_i)$, where $OPT(s_i)$ represents the optimal path cost from s_i to s_g. This would imply that the heuristic is **admissible**.
- Consistency implies admissibility, but admissibility does not necessarily imply consistency.

We then concluded with the following theorem.

Theorem 4.1. *If h is consistent, then A^* with graph search is optimal.*

4.2 Optimal Heuristics

Currently, we do not have a sure-fire method of coming up with consistent heuristics. However, it is possible to design admissible heuristics. Admissible heuristics are sufficient for optimality if we are using A^* with tree search.

Theorem 4.2. *If h is admissible, then A^* with tree search is optimal.*

4.2.1 Designing admissible heuristics

Idea: find an optimal path from s_i to s_g that respects the restrictions of the problem (e.g. obstacles), and relax some of the restrictions so that $h(s_i)$ is **computable** and **informative**.

- Computable: the heuristic can be computed in finite time.
- Informative: the heuristic should provide us with information which would assist in speeding up our search.
  - A heuristic which gives $h(s_i) = 0$ for all $i = 0, 1, 2, ...$ is considered to be uninformative, as it does not provide any information that helps to speed up our search.

4.2.1.1 Heuristic 1: Euclidean/Straight-line distance

Problem: find the shortest path from point s_0 to point s_g in NUS.

By ignoring the presence of obstacles/unwalkable terrain in our path (relaxing the restrictions), we can set $h(s_i) = d(s_i, s_g)$, where $d(s_i, s_g)$ represents the Euclidean distance from s_i to s_g.

This heuristic is admissible since $h(s_i) \leq OPT(s_i)$.

- If the path from s_i to s_g is a straight line, then $h(s_i) = OPT(s_i)$.
- Else, if the path is not a straight line, then $h(s_i) < OPT(s_i)$.

This heuristic is also consistent.

1. By the triangle inequality, $d(s_i, s_g) \leq d(s_i, s_{i+1}) + d(s_{i+1}, s_g)$.
2. Also, $d(s_i, s_{i+1}) \leq c(s_i, s_{i+1})$, where $c(s_i, s_{i+1})$ represents the cost of getting from s_i to s_{i+1}. This is the case since $d(s_i, s_{i+1}) = c(s_i, s_{i+1})$ when the path from s_i to s_{i+1} is straight, and $d(s_i, s_{i+1}) < c(s_i, s_{i+1})$ when the path is curved.
3. From the definition of $h(s_i) = d(s_i, s_g)$, it follows that $h(s_i) \leq c(s_i, s_{i+1}) + h(s_{i+1})$. This implies that the heuristic is consistent.

4.2.1.2 Heuristic 2: Manhattan distance

Problem: find the shortest path from point s_0 to point s_g in a matrix with obstacles.

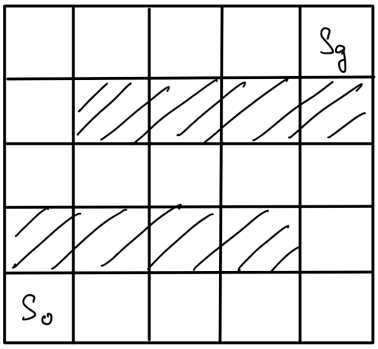

Figure 4.1: A matrix with start state s_0 and goal state s_g. Shaded boxes indicate obstacles.

With the additional restriction of only being able to move vertically and horizontally, the Manhattan distance would be a better heuristic than the Euclidean distance. This is because more restrictions of the original problem are being respected, thus the heuristic would give a more accurate representation of the cost from s_i to s_g.

This heuristic is admissible because:

- If the path from s_i to s_g does not have an obstacle in between, then $h(s_i) = OPT(s_i)$.
- If the path from s_i to s_g has an obstacle in between, then $h(s_i) < OPT(s_i)$.

4.3 Greedy Breadth First Search

How is it different from the other search algorithms?

- UCS uses $\hat{g}(u)$ to make decisions on which node to visit next, where $\hat{g}(u)$ represents the minimum path cost of reaching u (discovered so far).
- A^* search uses $\hat{g}(u) + h(u)$ to make decisions on which node to visit next.
- Greedy BFS uses only $h(u)$ to make decisions on which node to visit next.

Exercise: Greedy BFS is not optimal.

Proof:
We will prove by contradiction. Let us assume that we have a graph as shown below:

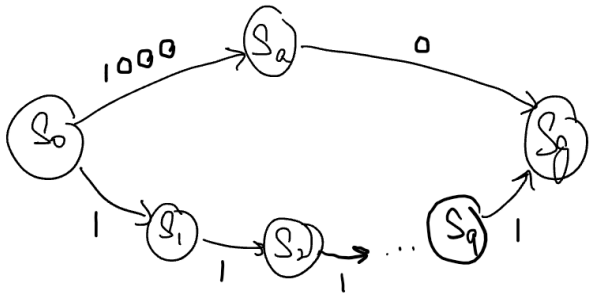

Figure 4.2: Graph with start state s_0 and goal state s_g

Since $c(s_a, s_g) = 0$ while $c(s_9, s_g) = 1$, consistent or admissible heuristic functions would likely produce a value of $h(s_i)$ such that $h(s_9) > h(s_a)$, even though $c(s_0, s_a) + c(s_a, s_g) = 1000 > c(s_0, s_1) + c(s_1, s_2) + \ldots + c(s_9, s_g) = 10$. However, since greedy BFS does not take $\hat{g}(s_i)$ into consideration, it will pick the path $s_0 \rightarrow s_a \rightarrow s_g$. Hence, greedy BFS is not optimal. □

4.4 Adversarial Search

In this section, we will be visiting problems involving multiple agents which will compete with each other, in the form of games.

4.4.1 Game 1: Bags and balls

Suppose that we have a game involving two bags, containing two balls each. We will be denoting the bags and balls as such (my notation differs slightly from what was delivered in lecture):

- B_1: bag 1.
- B_2: bag 2.
- $b_{1,1}$: ball 1 in bag 1.
- $b_{1,2}$: ball 2 in bag 1.
- $b_{2,1}$: ball 1 in bag 2.
- $b_{2,2}$: ball 2 in bag 2.

We will be playing as player *MAX*, and our goal will be to maximize our score. Our opponent will be playing as player *MIN*, and their goal will be to minimize our score (and thus maximizing their score).[1]

We can construct our game tree as such:

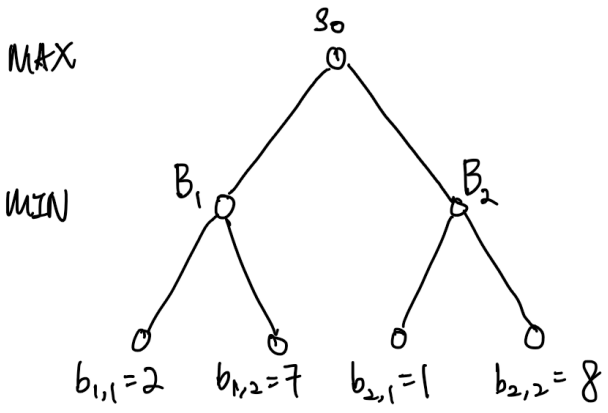

Figure 4.3: Game tree for bags and balls

In this graph, s_0 is our start state, whereas $b_{i,j}$ are the terminal states. We will also denote the branching factor of the graph by b, and depth by d. The number of possible states will be b^d.

We will examine what happens when both agents *MAX* and *MIN* plays optimally,

[1] This is commonly known as a *zero-sum game*.

- If B_1 is selected, *MIN* will pick $b_{1,1} = 2$ to minimize *MAX*'s score. If B_2 is selected instead, *MIN* will pick $b_{2,1} = 1$.
- Anticipating what *MIN* would play in both cases, *MAX* will select B_1 in order to maximize our score.

4.4.2 Game 2: Chess

Chess can also be modelled as a game. Here, our utility can be defined as follows:

- 1, if white wins.
- -1, if black wins.
- 0, if the game ends in a draw.

The objective of chess is the same as in Game 1, where we are aiming to maximize our utility, and our opponent is aiming to minimize our utility (and thus maximizing theirs).

However, the number of possible states is approximately 10^{120}, which cannot possibly be computed by modern computers.

4.4.3 Minimax function

In games, we can use the minimax function to help us get the optimal solution (the process is similar to that described in Game 1). The minimax function is recursively defined as such:

$$Minimax(s) = \begin{cases} Utility(s) & \text{if } Terminal(s) = 1 \\ \max\limits_{a \in \text{actions}} Minimax(Result(s, a)) & \text{if player} = MAX \\ \min\limits_{a \in \text{actions}} Minimax(Result(s, a)) & \text{if player} = MIN \end{cases}$$

- $Utility(s)$ represents the utility gained at state s.
- $Terminal(s)$ is a test that returns 1 if the node is a terminal node, and 0 otherwise.
- $Result(s, a)$ defines the resulting state when we perform an action a from state s.

4.4.4 Alpha-beta pruning

Since some game trees may be very large (such as the game tree for chess), is there any way to reduce the number of branches that we need to search?

Let us consider an example:

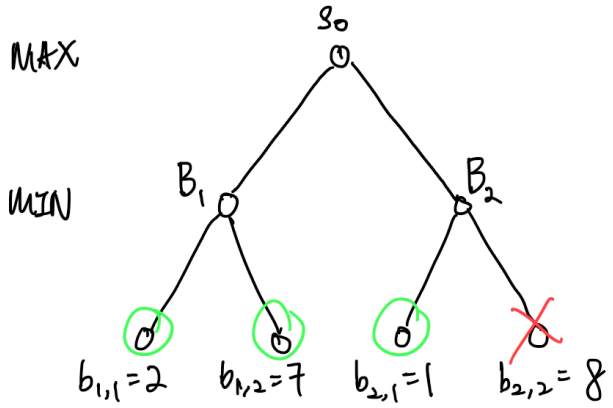

Figure 4.4: Game tree for bags and balls, with alpha-beta pruning

- We will explore this tree from left to right.
- We will first visit B_1.
  - Starting from the left, we will explore $b_{1,1} = 2$. Hence, MIN will be able to secure a score of ≤ 2.
  - MIN will then explore $b_{1,2} = 7$. However, since $b_{1,1} < b_{1,2}$, MIN will pick $b_{1,1} = 2$.
- After exploring the left subtree, MAX is able to guarantee a score of ≥ 2.
- We will then explore B_2 next.
  - Starting from the left again, we will first explore $b_{2,1} = 1$. hence, MIN will be able to secure a score of ≤ 1.
  - Now, if $b_{2,2} > b_{2,1}$, we know that MIN will select $b_{2,1} = 1$. On the other hand, if $b_{2,2} < b_{2,1} < b_{1,1} = 2$, we know that MAX will not pick from the right subtree. Hence, there is no need to check the value of $b_{2,2}$.

Remark: The order of evaluation of the nodes matter for alpha-beta pruning.

If the nodes were evaluated in the order $b_{1,1} \to b_{1,2} \to b_{2,2} \to b_{2,1}$ instead, then we will need to check the value of the last node. This is because $b_{2,2} = 8 > b_{1,1}$, and thus, if the last node has a value of > 2, MAX will pick from the right subtree instead.

4.5 Exercise

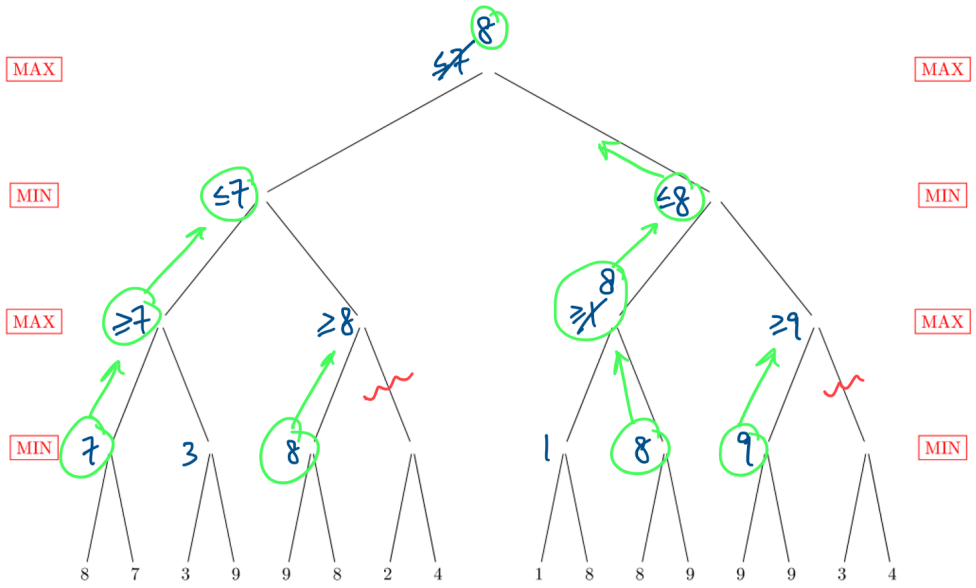

Figure 4.5: Exercise for Lecture 4

Using alpha-beta pruning, we do not need to explore the subtree containing the nodes (2, 4) and the subtree containing the nodes (3, 4).