Project Report for Intro to AI HW2

Daniel Pattathil, Kush Baldha, Frank Jiang
dp865, kb727, fj99

March 13, 2019

Contents

1 A* Search Expanded Nodes

Given the straight-line distances to Bucharest, we can trace A* search from Lugoj to Bucharest as follows:

NOTE: Bold is the **chosen city**, strikethrough is ~~visited~~, format is (City, f, g, h)

Initial State:
(Lugoj, 244, 0, 244)

First Expansion, on Lugoj:
(Timisoara, 440, 111, 329), **(Mehadia, 311, 70, 241)**

Second Expansion, on Mehadia: **(Drobeta, 387, 145, 242)**, (Lugoj, ~~384~~, ~~140~~, 244), (Timisoara, 440, 111, 329)

Third Expansion, on Drobeta: **(Craiova, 425, 265, 160)**, (Mehadia, ~~461~~, ~~220~~, 241), (Timisoara, 440, 111, 329)

Fourth Expansion, on Craiova: (Rimnicu Vilcea, 604, 411, 193), (Piteri, 503, 403, 100), (Drobeta, ~~627~~, ~~385~~, 242), **(Timisoara, 440, 111, 329)**

Fifth Expansion, on Timisoara: (Arad, 595, 229, 366) (Rimnicu Vilcea, 604, 411, 193), **(Piteri, 503, 403, 100)**

Sixth Expansion, on Piteri: (Arad, 595, 229, 366) (Rimnicu Vilcea, 604, 411, 193)*, **(Bucharest, 504, 0, 504)**, (Craiova, ~~701~~, ~~541~~, 160)

In the final expansion, we complete the A* search with an estimated cost or, f(n) value, of 504.

*Riminicu value not updated, because it is already in the queue

2 Successor State Space

2.1

2.1.1 Breadth First

With breadth-first search, we are expanding/traversing through the space with the most shallow states/nodes. In other words, if the goal state is 11, the order in which the states/nodes will be visited is:
$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11$

2.1.2 Depth-Limited Search

With depth-limited search, we are given the limit of 3. This means that we are given the restriction of not expanding/traversing to states/nodes beyond the level of the that we are given. Level 0 represents the first level, therefore level 3

represents the fourth level and the furthest level that we can traverse due to the limit. Given the goal state of 11, the order in which the states will be visited is: 1→ 2 → 4 → 8 → 9 → 5 → 10 → 11

2.1.3 Iterative-Deepening Search

With iterative-deepening search, we are visiting the states/nodes in iterative, or level by level. Therefore, the order in which the states/nodes will be visited will be:

First iteration: 1
Second iteration: 1→ 2 → 3
Third iteration: 1→ 2 → 4 → 5 → 3 → 6 → 7
Fourth iteration: 1→ 2 → 4 → 8 → 9 → 5 → 10 → 11

2.2 Bidirectional Search

Bidirectional search should work just as well because if we were to search in reverse, for any level, we can see that the successor is just half of that. In other words, the only successor of any n in the reverse, or upwards, direction is simply floor(n/2). This is especially helpful because doing bidirectional search can eliminate extraneous exploration and keeps the focus in the search. The order in which the states will be visited is as follows:

(Forward) 1→ 2 → 3
(Backwards) 11→ 5,
(Forwards) 2→ 3 → 4 → 5
(Backwards) 5→ 2

At this point the search concludes because we know that 5 is present in both the forward and backwards direction. Furthermore, the branching factor for bidirectional search is 2 in the forward direction and 1 in the backwards direction.

3 Search Truth

3.1 Breadth-First Special Case

This statement is true. We know that Uniform-cost Search expands on the node with the lowest path cost. Therefore, when all step/path costs are equivalent in a Breadth-first Search, then it is simply an unique case of Uniform-cost Search.

3.2 Depth-First Special Case

This statement is true. We know that Best-first Search is a greedy search algorithm that expands on the node closest to the goal state. Therefore, in the case where Depth-first Search has the opposite depth, in other words f(n) = -depth(n), Depth-first Search becomes a special case of Best-first Search.

3.3 Uniform-Cost Special Case

This statement is true. Uniform-cost Search (UCS) expands on the node/state with the lowest path cost. UCS also uses f(n) = g(n) where g(n) represents the

path cost from initial state to current state/node. We know that in A* search, we use the evaluation function of f(n) = g(n) + h(n). Therefore, UCS becomes a special case of A* search when h(n) = 0. Meaning if h(n) or, heuristic function, during UCS is equivalent to 0, it will produce the same results as A* search.

4 Consistency Proof

For a heuristic to be admissible h(n) must be less than h(n)* (the true cost) for all n.
For a heuristic to be consistent, h(n) must be less than c(n,a,n') + h(n') for all n,a,n'.

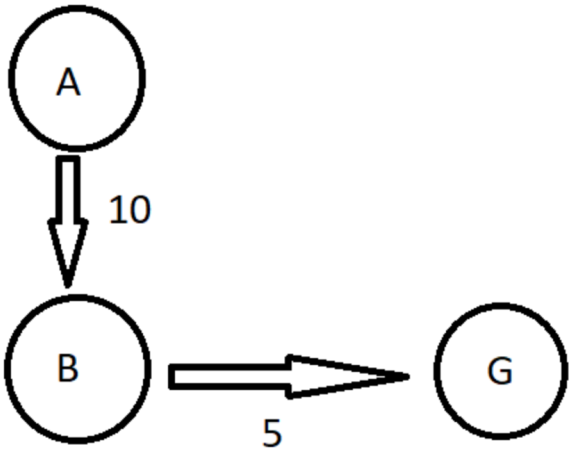

Figure 1: Consistency

For our heuristic to be admissible in this problem, we must have
h(A) <= 15
h(B) <= 5
h(G) <= 0
For our heuristic to be non consistent, we must have
h(A) >= (c(A,B)) + h(B)
h(A) >= (10) + h(B)
Thus if we h(A) = 14, h(B) = 1, h(G) = 0 then,
14 >= 10 + 1 = 11
Therefore this example is an of a admissible heuristic that is not consistent.

5 Constraint Satisfaction Problem

We should choose the variable that is most constrained because this helps with pruning the tree. If the variable is likely to fail, then we can prune out that branch and avoid wasteful searching. However, we choose the **value** that is least constraining because it is the one that is most likely to succeed. A variable doesn't fail unless it tries all possible values so there is no efficiency in trying

the "fail first" method we used before. Using the value that is least constraining also allows for maximum flexibility.

6 MAX and MIN Players

6.1 Best Choice

The best move for the MAX player is to go to C. If we take a look at the branch on the left, B chooses the minimum value which would be 3. Therefore the branch on the left of A has a value of 3.

Now, let's take a look at the branch on the right of A. We can start with the choice that I takes. I chooses the minimum value of 0. Therefore F chooses the maximum value of 5 because it has to choose between 0 and 5. G chooses the maximum value of 8. Now C has to choose the minimum value between 5,8,4 which is 4. A now chooses the maximum value between 3 and 4 and therefore chooses the right branch that has a value of 4.

6.2 Left to Right Pruning

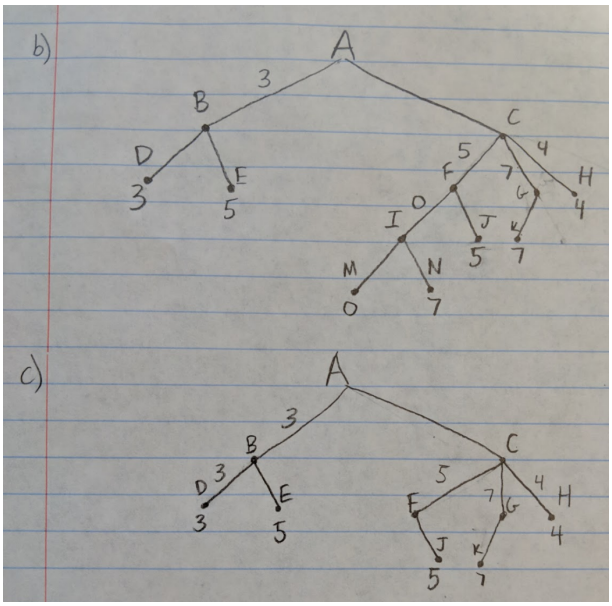

Figure 2: Pruning

6.3 Right to Left Pruning

The pruning is different on the right to left ordering because we after looking at minimum of 4 at H, we know we do not need to expand branches F and G because they have maximum values of at least 5 and 7.

7 Admissible/Consistent (*Reference Fig. 1)

7.1 h(n) = min{h1(n), h2(n)}

7.1.1 Admissible

Yes, this heuristic is admissible, because the taken h(n) value will always be the minimum value of two admissible values, and as a result, for all values of n, h(n) will be less than or equal to h*(n).

7.1.2 Consistent

Yes, this heuristic is consistent, because we already know that h1(n) and h2(n) are consistent. If we choose h1 for A and h2 for A, it is definitely consistent. The only case to consider, is if we choose h1 for A and h2 for B (differing heuristics for each node). In this case, we know that h2 for A is greater than h2 for B, which means that the h2 for B must still be within the range for consistency.

Ex) h(A) <= 15, h(B) <= 5, c(A,B) = 10
h1(A) = 12, h1(B) = 5, h2(A) = 13, h2(B) = 4, h(A) <= 10 + h(B)
h1(A) <= 10 + h2(B)
12 <= 10 + 4

7.2 h(n) = w*h1(n) + (1-w)*h2(n)

7.2.1 Admissible

Yes, this heuristic is admissible, because either the h value will be within the range of [min(h1,h2), max(h1,h2)], both of which we know are already admissible, so there is no way it can become larger than h*.

7.2.2 Consistent

Consistent - Assuming that the w chosen is used throughout the h-value determination, we can say that this heuristic is consistent. In the case that w is 0 or 1, we are simply taking the min or max of the heuristics which we have already determined are both consistent. The only other significant case would be to look at is if h1 and h2 each are weighted equally with w = 0.5. This would mean averaging the the h-values and as a result, there is not a case where the cost of getting between two nodes would become higher than either of the initial cost distances, because it averages out the distances between them.

Ex) h(A) <= 15, h(B) <= 5, c(A,B) = 10
h1(A) = 15, h1(B) = 5, h2(A) = 10, h2(B) = 0, h(A) <= 10 + h(B)
h(A) = 12.5, h(B) = 2.5

7.3 h(n) = max{h1(n), h2(n)}

7.3.1 Admissible

Yes, this heuristic is admissible, because the taken h(n) value will always be the maximum value of two admissible values, and as a result, for all values of n, h(n) will be less than or equal to h*(n).

7.3.2 Consistent

Consistent - Yes, this heuristic is consistent, because we already know that h1(n) and h2(n) are consistent. If we choose h1 for A and h2 for A, it is definitely consistent. The only case to consider, is if we choose h1 for A and h2 for B (differing heuristics for each node). In this case, we know that h2 for A is less than h2 for B, which means that the h2 for B must still be within the range for consistency.

Ex) h(A) <= 15, h(B) <= 5, c(A,B) = 10
h1(A) = 13, h1(B) = 4, h2(A) = 12, h2(B) = 5, h(A) <= 10 + h(B)
h1(A) <= 10 + h2(B)
> 13 <= 10 + 5

8 Simulated Annealing and Hill Climbing

8.1 Hill Climbing Better than Simulated Annealing

Hill climbing will work better than simulated annealing in cases where all the local maxima (or minima depending on the optimization problem) are very similar or in the case where there is simply one local maxima which is the global maximum.

8.2 Random State and Simulated Annealing

The hill climbing part of simulated annealing could be ignored in a function where the entire range is a plateau. This would mean that no matter which state, the values found would be the same and hill climbing would not assist in finding an optimality.

8.3 Simulated Annealing Optimal Case

Using the answers from parts a and b, we can see that simulated annealing is useful, because we assume that value functions are not either horizontal or contain only one maxima (or minima). This suggests that there will be a value function that has many local maxima with very different values. The shape of this function would end up continually going up and down at different input states and would seemingly be hard to find the actual global maxima without searching all possible options. Any shape where hill climbing even with random placement would still likely not find the maxima would be a good example of where to use simulated annealing.

8.4 Measure of Goodness

If our main concern is that simulated annealing runs out faster than we believe it takes to find an optimal value, we can increase the duration of our searching. If we are able to measure the goodness, we can say that as we increase goodness, we reset the value that will slow down our schedule.

8.5 Increased State Count

The better option that we would want to pursue is using the additional memory to keep track of more states for the Simulated Annealing instance. This amount of states that we would use could be a variable called S in order to keep track of the factor. Then, we would evenly spread out instances of Simulated Annealing throughout the domain of our function. If we have the range of the domain as N, we would divide that by the amount of instances we have I which would be $2000000/N$. This would give plenty of opportunities to different parts of the value function, and then, many different directions would be explored thanks to the probability of accessing with random probability.

9 2 Player Game

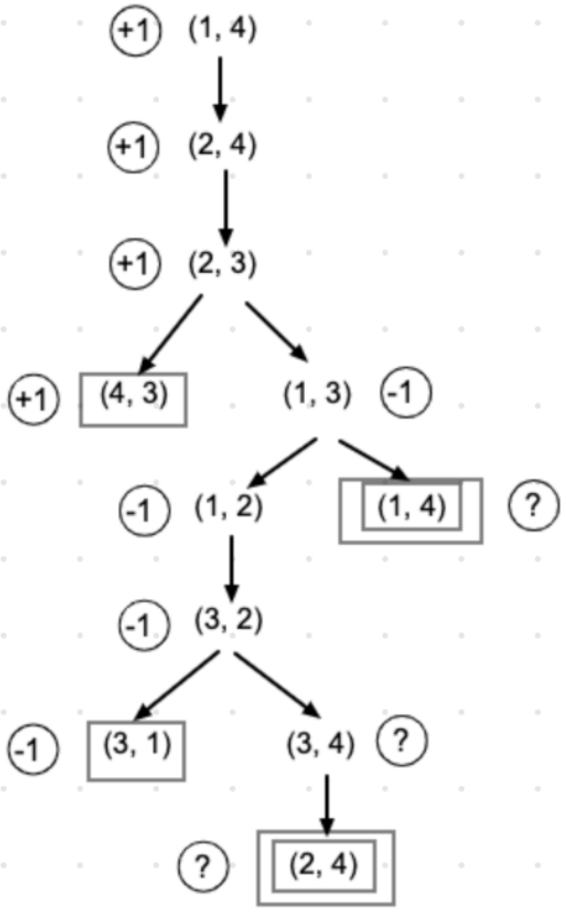

Figure 3: Game Tree with Labels

We applied the (?) notation to any state that would loop backwards, because it is not clear how to assign the values for the loop states. We ended up only

having to apply an additional (?) to the state (3,4) when it is A's turn, because it would not have been assigned anything else according to the backed-up minmax value. Everything else would get backed up based on the winning state and player choice.