CSE 574 Homework 2

Zackary Crosley

September 14, 2018

Problem 1. Uninformed Search - Iterative Leng

1. Show this algorithm is optimal for general path costs.

   Suppose, to the contrary iterative lengthening selects a suboptimal route. That is for an optimal path A, A has a unit length less than or equal to all other possible paths. Suboptimal path B has unit length larger than A. Selecting B would require it to be seen before or in the same iteration as A. However, since iterative lengthening increases path length by a unit value each time A would have been encountered in an iteration before B and thus B wouldn't have been selected. Therefore, this algorithm must be optimal.

2. Consider a uniform tree with branching factor b, solution depth d, and unit cost. How many iterations will iterative lengthening require?

   Since the paths are all unit cost, the number of iterations of the iterative lengthening algorithm would be equal to the unit cost of the most optimal path to the solution. Since each successive layer of depth in the tree is one greater path cost, this is equivalent to the solution depth d.

3. Now consider step costs drawn from the continuous range [ϵ, 1], where $0 < \epsilon < 1$. How many iterations are required in the worst case?

   Since the cost is continuous it is possible for the search to take considerably longer. If you imagine a tree that is non-finite and has a never ending path leading away from the solution with path increments of exactly ϵ, then the tree will take $\frac{n}{\epsilon}$ iterations where n is the optimal cost to reach the solution. This is because the iteration will step up the path cost by exactly ϵ each iteration. The path cost to reach the solution would be $d\epsilon \leq cost \leq d$ since the path cost is in range [ϵ, 1]. In the worst case, where the cost is d, this would result in a total number of iterations $\frac{d}{\epsilon}$

Problem 2. Apply the A* search on this graph to reach Bucharest from Lugoj using the straight-line distance heuristic function. Show the sequences of nodes which are considered by the algorithm. Show the f,g,and h for each node.

Search Process by Phase

Phase 1 From Lugoj

Mehadia h(n) = 241, g(n) = 70, f(n) = 311 Timisoara h(n) = 329, g(n) = 111, f(n) = 440

Phase 2 From Mehadia

Dobreta h(n) = 242, g(n) = 145, f(n) = 387

Phase 3 From Dobreta

Craiova h(n) = 160, g(n) = 265, f(n) = 425

Phase 4 From Craiova

Rimnicu Vilcea h(n) = 193, g(n) = 411, f(n) = 604 Pitesti h(n) = 100, g(n) = 403, f(n) = 503

Phase 5 From Timisoara

Arad h(n) = 366, g(n) = 229, f(n) = 595

Phase 6 From Pitesti

Bucharest h(n) = 0, g(n) = 504, f(n) = 504

Problem 3. Heuristic Properties

1. Is h(n) admissable?

   No. Admissability requires that for every point in the graph h(n) overestimates the distance to the goal. In this case, our goal is G. At point A the heuristic provides an estimate distance of 1, but the actual distance is 4. Therefore the heuristic cannot be admissable.

2. Is h(n) consistent?

   No. Consistency (monotonicity) requires that for any node n and an arbitrary successor n' the heuristic estimation for n cannot exceed the actual cost of reaching n' plus the heuristic estimation for n'. If we look at nodes S and A, the heuristic evaluation of S is 7. However, the actual cost to A is 4 and A has a heuristic cost of 1, for a total of 5. Since 5 ¡ 7, this is not a consistent heuristic.

3. Show A* using GRAPH-SEARCH on the provided state space returns a suboptimal solution.

   Phase 1 From S

   B h(n) = 5, g(n) = 2, f(n) = 7 A h(n) = 1, g(n) = 4, f(n) = 5

   Phase 2 From A

   G h(n) = 0, g(n) = 8, f(n) = 8

   This totally misses the possible route S -¿ B -¿ A -¿ G which has a path cost of 7. A* will not find alternative routes once the destination is reached. Therefore this search will return a suboptimal result.

Problem 4. See Figure 1 and 2.

	Var assigned or de-queued	List all values eliminated from neighboring variables	Back track ?
ex	X	Y ≠ 3, 4 Z ≠ 3 (example)	☑
1	L=1	H≠1 ; A ≠1, 2; EL≠1; HY≠1	☐
2	H=2	EL≠2; HY≠2	☐
3	A=3	EL≠2, 3, 4; HY≠3	☐
4	A=4	EL≠3, 4; HY≠4	☒
5	H=3	EL≠3; HY≠3	☐
6	A=3	EL≠2, 3, 4; HY≠3	☒
7	A=4	EL≠3, 4; HY≠4	☐
8	EL=2	M≠2, B≠2	☐
9	HY=2		☐
10	M=1	B=1	☐

	Var assigned or de-queued	List all values eliminat neighboring variables
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		

Figure 1: Problem 4 Constraint Satisfaction Search Table.

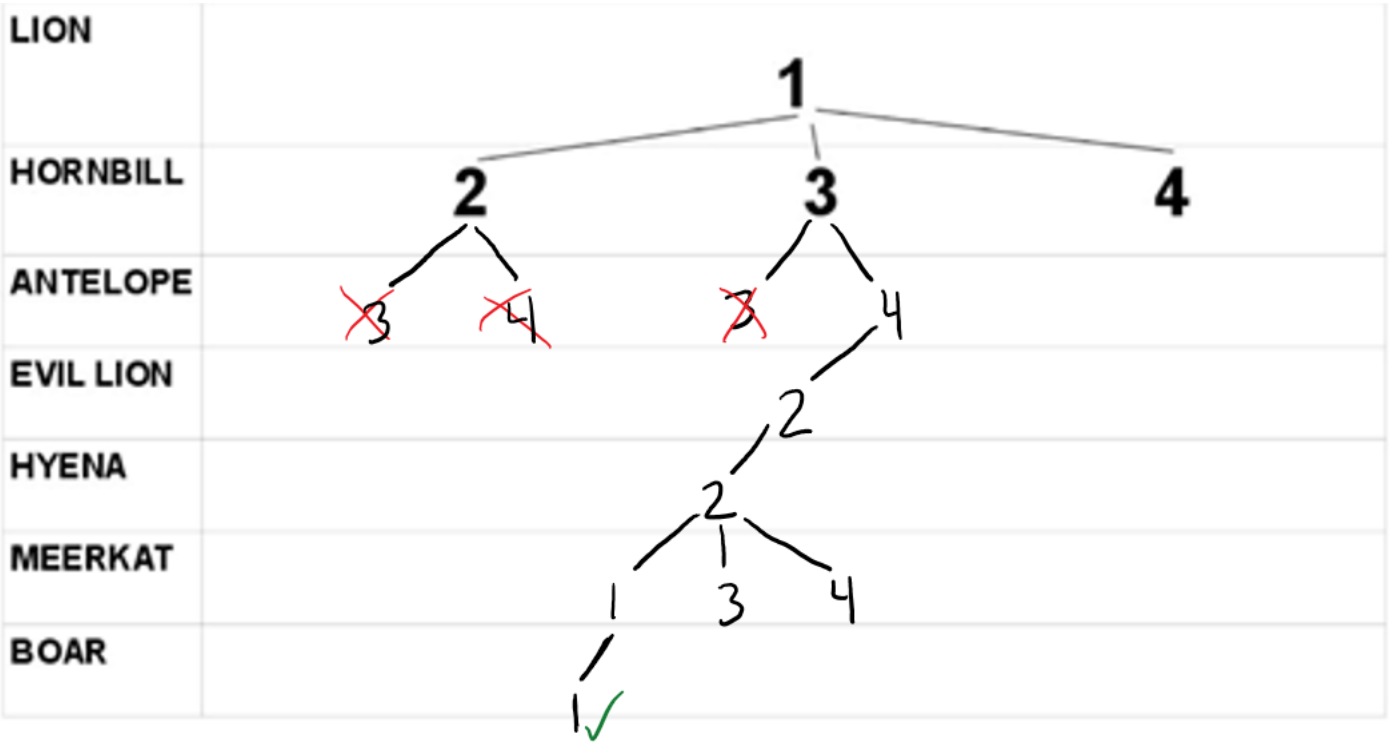

Figure 2: Problem 4 Constraint Satisfaction Search Tree.

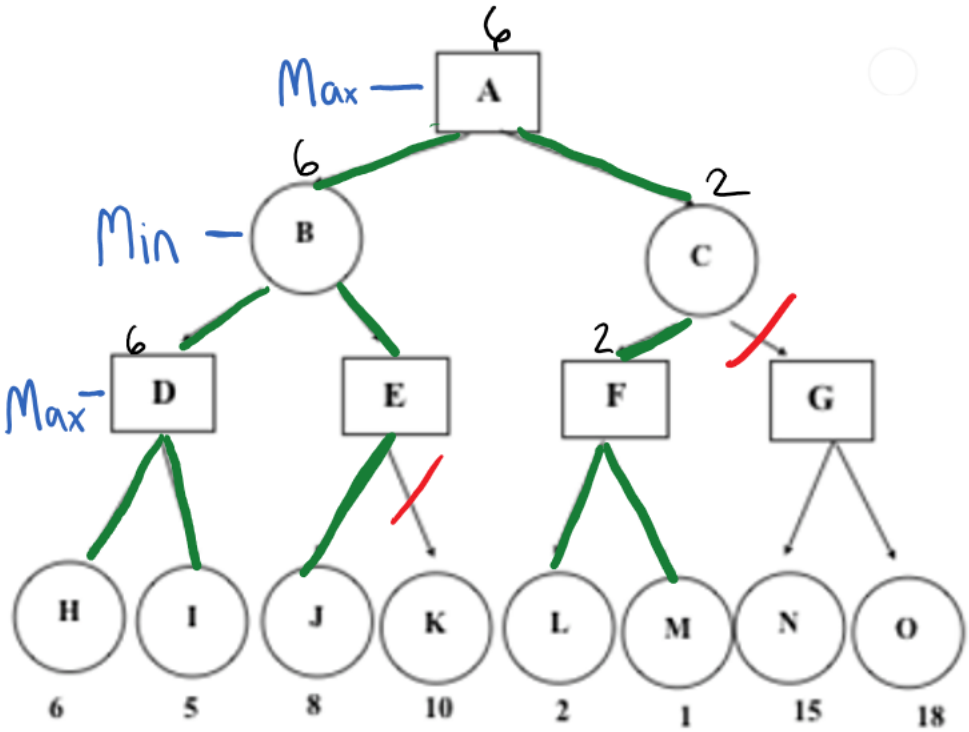

Figure 3: Problem 5 Alpha-Beta Pruning.

Problem 5. See Figure 3.

Problem 6. Adversarial Search

1. Draw the complete game tree.

   See Figure 4.

2. Mark each node with their minimax values.

   See Figure 5.

3. Explain why the standard minimax algorithm would fail on this example and provide a method to improve the algorithm for this use case.

   The standard minimax algorithm would never finish, as it would recurse down the and infinitely deep branch. A possible sequence of moves is A moving from 1 to 2 and back and B moving from 4 to 3 and back ad infinitum. The standard minimax algorithm would follow this infinite chain and thus never complete. There are a few ways this algorithm could be improved. One method would be to do a depth limited search that uses a heuristic evaluation, allowing the tree to only navigate down so many iterations for a decision. Since the number of states is equal to $n(n-1)$ where n is the number of boxes it is pretty easy to use a depth limit of n for an accurate evaluation. Another method, however, would be to use a dictionary to store the evaluations from one branch to be used by another. Since the state space is small with lots of repetition, we can go down one branch and get the minimax evaluations for that state and reuse them when that state is encountered again, without having to expand it. Both of these methodologies could be used for other problems with loops, given a few conditions. The first is a known methodology that would be useful in any problem with a good candidate for the depth to search down and a good heuristic for evaluating states. The latter would work in any problem where we can guarantee at least one state in an infinite sequence will be first encountered on the path to a leaf node. This would ensure that the state in the sequence would be able to return a value before entering a never-ending computation. Unfortunately this property would be hard to guarantee for a generic problem.

4. This game can be abstracted to any n squares laid out in one dimension. Prove that if n is even A wins and if n is odd B wins.

   For games with even size N A will always win as A is the first to make it to the center position, and thus has control when A and B meet. Take our base case with N=4: A moves to position 2 by necessity, B moves to position 3 by necessity, A jumps B to 4 and wins. If N is two larger, that is there is one more square until the center, A is still the first to get to the center since it is the first to move. B can choose to hold back and jump A when it gets closer, but because A can move first the furthest location B could theoretically wait is one position from the center. That is, A moves to center and then moves one more space, only to be jumped by B. This would put A and B equal distant from their respective goals, with B at the center square on A's side and A on the center square on B's side. A, however, moves first, and thus can reach its goal first so long as it never doubles back, giving it a guaranteed victory.

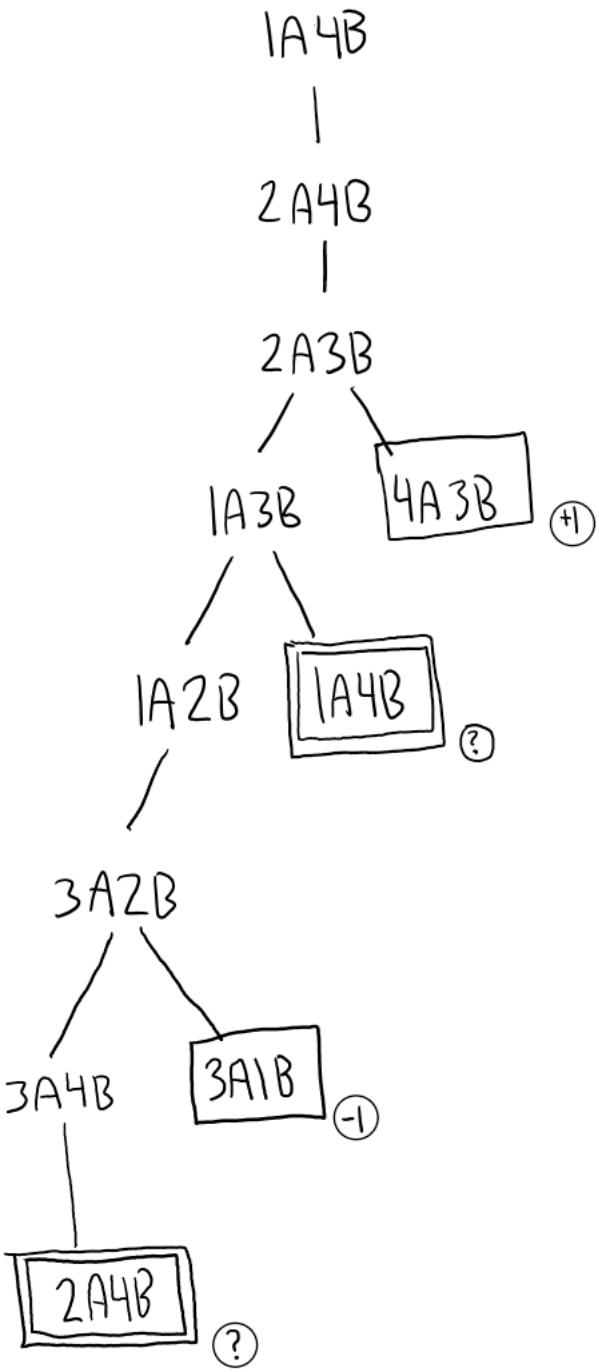

Figure 4: Problem 6a Adversarial Game Tree.

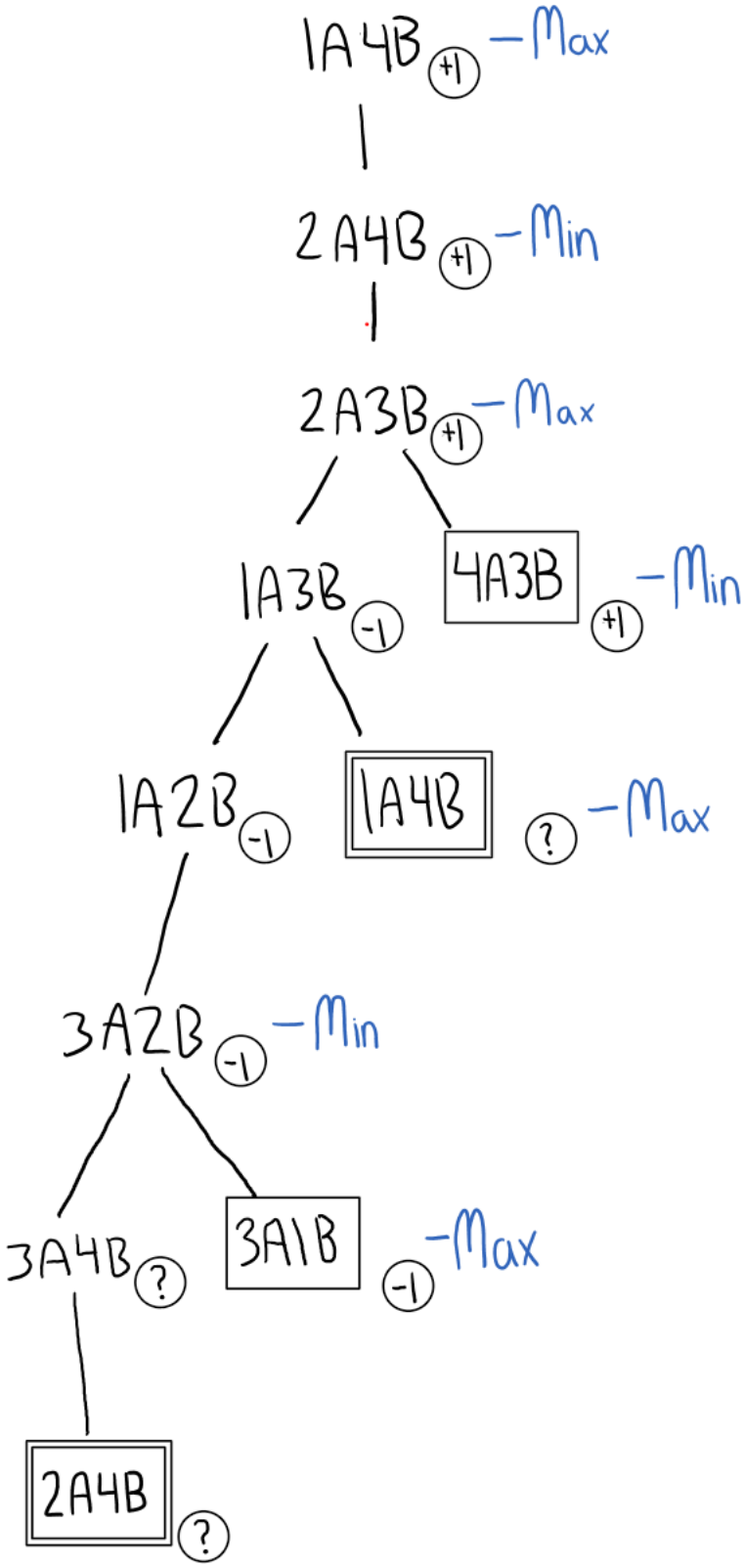

Figure 5: Problem 6b Adversarial Minimax.

For games with odd size N, the opposite is true and B will always in. Take the base case where N = 3. A moves to position 2 and is then jumped by B from position 1 to position 3, allowing B to win. With an odd size N, there is only one center square which A, as the first to move is guaranteed to reach first. Since A and B are equidistant from the squares adjacent to the center, B will be in the adjacent square and can now jump A making it one square closer to its goal than A. Despite A moving first, B will now reach its goal first. If A doesn't choose to move into the center position, it will have to move back one position and let B move into the center. A can now move back to the position adjacent to the center, but will be jumped by B which is even closer to its goal. The further A holds towards its own end the closer B will be to its goal relative to A when one finally does jump the other. Thus B can always win if it plays optimally.

Problem 7. Programming Assignment - Write Alpha-Beta Pruning Algorithm.

See alpha_beta.py file for implementation.