CS 440: Introduction to Artificial Intelligence

Homework #2 Search Problems in AI

March 10, 2023

Professor Abdelsam Boularias

Jay Patwardhan 208001851
Alan Wu 208000574
Neel Shejwalkar 207004853

Problem 1

Tracing Operation A^* from Lugoj to Bucharest

By applying the A^* tree search algorithm on the graph using the straight line distance heuristic, we end up with the following sequence of nodes expanded in order, written in tuples in the form described in the assignment:

$$(Lugoj, 244, 0, 244) \tag{1}$$
$$(Mehadia, 311, 70, 241) \tag{2}$$
$$(Lugoj, 384, 140, 244) \tag{3}$$
$$(Drobeta, 387, 145, 242) \tag{4}$$
$$(Craiova, 425, 265, 160) \tag{5}$$
$$(Timisoara, 440, 111, 329) \tag{6}$$
$$(Mehadia, 451, 210, 241) \tag{7}$$
$$(Mehadia, 461, 220, 241) \tag{8}$$
$$(Lugoj, 466, 222, 244) \tag{9}$$
$$(Pitesti, 503, 403, 100) \tag{10}$$
$$(Bucharest, 504, 504, 0) \tag{11}$$

Problem 2

Consider a state space where the start state is number 1 and each state k has two successors: numbers $2k$ and $2k+1$.

Part A

Suppose the goal state is 11. List the order in which states will be visited for breadthfirst search, depth-limited search with limit 3, and iterative deepening search.

Solution. *The list of states visited for each search algorithm is as follows:*
For Breadth-First Search: $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)$
For Depth-Limited Search, limit 3: $(1, 2, 4, 8, 9, 5, 10, 11)$
For Iterative Deepening search:

$$\begin{aligned}
&Limit = 0 && (1)\\
&Limit = 1 && (1, 2, 3)\\
&Limit = 2 && (1, 2, 4, 5, 3, 6, 7)\\
&Limit = 3 && (1, 2, 4, 8, 9, 5, 10, 11)
\end{aligned}$$

Part B

How well would bidirectional search work on this problem? List the order in which states will be visited. What is the branching factor in each direction of the bidirectional search?

Solution. *The branching factor from the direction of the root node is 2, as every node has 2 children, and every parent node would have already been searched. The branching factor from the direction of the goal node is 3, as we have to account for its 2 children as well as the parent node.*
Asymptotically, bidirectional search is better than the other singular directional algorithms above (on the order of $O(2^{d/2} + 3^{d/2})$ *versus* $O(2^d)$*) but when d is a small value like in this problem, it is possible for it to not perform as well.*
The explored states are: $(1, 11, 2, 3, 5, 22, 23, 4, 5)$.

Problem 3

Correct vs. Not Correct

Part A

True: Breadth-first search is a special case of uniform-cost search, if we set the cost function c(s,s') = 1 for all states s and s'.

Part B

True: Depth-first search is a special case of best-first tree search with $f(n) := 1/depth(n)$

Part C

True: Uniform-cost search is a special case of A^* search with h(s) = 0 for all states s.

Part D

False: Depth-first graph search is NOT guaranteed to return an optimal solution, as it could go down directly to a sub-optimal solution simply because it's the first and longest path.

Part E

False: Breadth-first graph search is NOT guaranteed to return an optimal solution unless all actions have the same cost, as it could find a very expensive but short path to a solution state (think root $\rightarrow$ solution but the cost is 500, but there is another path root $\rightarrow$ intermediary $\rightarrow$ solution whose costs for both actions are 1).

Part F

True: Uniform-cost graph search is guaranteed to return an optimal solution, as it expands nodes based on smallest cost, so the solution it finds is optimal.

Part G

True: A^* graph search is guaranteed to return an optimal solution if the heuristic is consistent, since by theorems in the slides consistent heuristics are optimal (slide 17 on the A* lecture slides).

Part H

False: A^* graph search is NOT guaranteed to expand no more nodes than depth-first graph

Problem 3 continued on next page...

search if the heuristic is consistent, since DFS can go immediately to a sub-optimal solution in its first run down, while A* will take a long time expanding many nodes and determine the most optimal solution.

Part I

True: A^* graph search is guaranteed to expand no more nodes than uniform-cost graph search if the heuristic is consistent, since the heuristic makes the search more informed. We use that UCS is a special case of A* with $h(n) = 0$ for all n. Since h as the estimated shortest cost is necessarily nonnegative, it follows that $f(n) = g(n) + h(n) >= g(n) + 0$ for all n, and thus if we have two heuristics: 0 and h(n), we have a more informed heuristic h(n), which by definition will expand no more nodes than the less informed heuristic 0.

Problem 4

Iterative deepening is sometimes used as an alternative to breadth first search. Give one advantage of iterative deepening over BFS, and give one disadvantage of iterative deepening as compared with BFS. Be concise and specific.

An advantage that IDS has over BFS is its efficient use of space. BFS uses $O(b^d)$ space as it must store at worst every single node in the tree of a depth shallower than the goal node. In comparison, IDS is identical to Depth-First search during each one of its iterations. This means that IDS has a space complexity of $O(bd)$ or even $O(d)$ depending on the implementation, far better than the exponential space required for BFS.

IDS has the disadvantage of being much less efficient with time than BFS. While they are both similar asymptotically, with a complexity of $O(b^d)$, IDS needs to search a tree of depth i for every single iteration $0 \leq i \leq d$, whereas BFS needs to only search the tree of depth d once before finding the goal node. This means the runtime of IDS has a looser upper bound of $O(b^d + b^{d-1} + \ldots + b + 1)$, which is far more expensive practically.

Problem 5

Prove that if a heuristic is consistent, then it must be admissible. Construct an example of an admissible heuristic that is not consistent. (Hint: You can draw a small graph of 3 nodes and write arbitrary cost and heuristic values so that the heuristic is admissible but not consistent.)

We will prove this claim using induction over all nodes.
Base case: Let n_0 be the goal node, and n_1 be a node one step away. By the definition of a consistent heuristic,

$$\begin{aligned} h(n_1) &\leq c(n_1, a, n_0) + h(n_0) \\ h(n_1) &\leq c(n_1, a, n_0) + 0 \\ h(n_1) &\leq h^*(n_1) \end{aligned}$$

as the heuristic is defined to be 0 at the goal node, and the cost of the path from n_1 to n_0 is exactly the true cost of the path to the goal, $h^*(n_1)$. So, h demonstrates admissible behavior for the base case.
Inductive hypothesis: Let the condition $h(n_k) \leq h^*(n_k)$ hold for some node n_k that is k steps away from the goal node.
Inductive step: We'll now look at a node n_{k+1} that is $k+1$ steps away from the goal node. By the definition of consistency,

$$h(n_{k+1}) \leq c(n_{k+1}, a, n_k) + h(n_k)$$

By the inductive hypothesis,

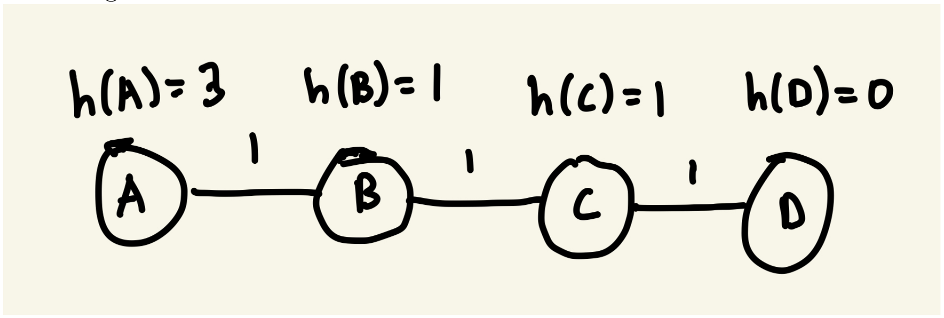
$$h(n_{k+1}) \leq c(n_{k+1}, a, n_k) + h^*(n_k)$$

Now notice that $c(n_{k+1}, a, n_k) + h^*(n_k)$ is exactly the true distance from the node n_{k+1}. So, then

$$h(n_{k+1}) \leq h^*(n_{k+1})$$

And so the heuristic is admissible for all nodes. □

The graph below provides an example of a heuristic that is admissible and not consistent. Note that node D is the goal node.



h is admissible: $3 \leq 3$, $1 \leq 2$, $1 \leq 1$, $0 \leq 0$
h is not consistent: $3 \nleq 1 + 1$

Problem 6

In a Constraint Satisfaction Problem search, explain why it is a good heuristic to choose the variable that is most constrained but the value that is least constraining.

Choosing a fail first/most constrained variable is a good heuristic because it allows us to quickly prune that branch of the search tree, letting us avoid pointless searches through other variables. In other words, if a variable has a lot of constraints, then it will likely have very few outcomes, if there even exist any at all which satisfy the constraints it has as well as the previous constraints which led to this point of the tree. Since it the most selective, it would be good to satisfy its constraints earlier rather than later down the tree. If we selected a less restrictive variable first, it could lead to many more branches, each of which will eventually settle upon this particular constraint, and most branches will terminate there because they don't satisfy such restrictive constraints. However, this takes up a lot of extra time (to compute down the tree) and space (to store each branch of the tree), and so it is best to consider the most restrictive variables first, because they will avoid these useless terminating searches where they are not necessary.

Next, it would be best to consider the least constraining value for the variable. The only way to fail is to test all the values, so there is no reason to attempt fail first strategies in which we purposefully test values that will result in failure first. In this moment, it is best to select a value which will allow the largest range of potential variables stemming from it, because there is a greater chance of finding a match in those possibilities. This is unlike choosing the most constraining variable because the values that we are ordering already satisfy the given constraints: by selecting the most constraining value for the other neighbors, it only limits the trees that one can make and reduces the chances of finding a solution because there are less branches to search. It is important to note that we are not saying that more branches is better, because this is untrue, as shown in choosing the most constrained variables first. Instead, we are saying that in the given constraints, it is best to select values that allow us to search more items, since in that moment there is no knowledge of which branch is better. Thus, since the probability of finding a solution for a branch is almost equal to every other branch, it is best that we allow ourselves the freedom to consider more branches first.

Thus, by choosing the most constrained variable and the least constraining value, we avoid pointless searches and search more branches earlier in time, allowing us to find solutions faster.

Problem 7

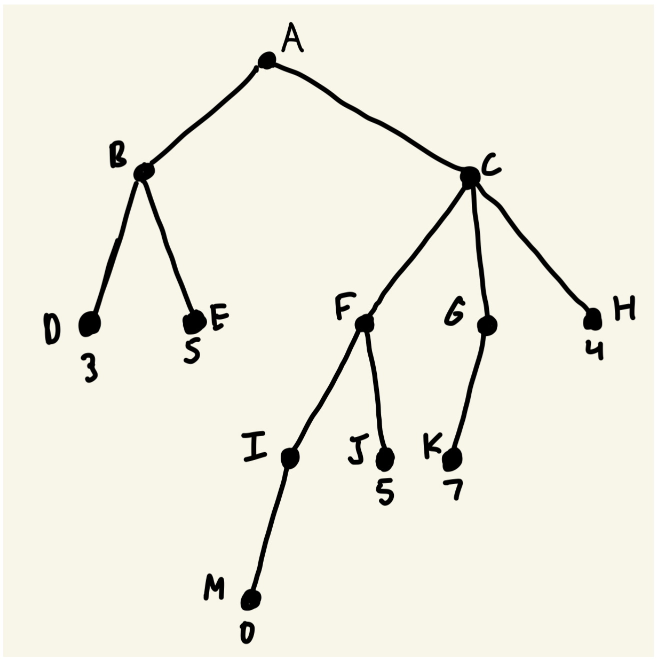
Consider the following game tree, where the first move is made by the MAX player and the second move is made by the MIN player.

Part A
What is the best move for the MAX player using the minimax procedure?

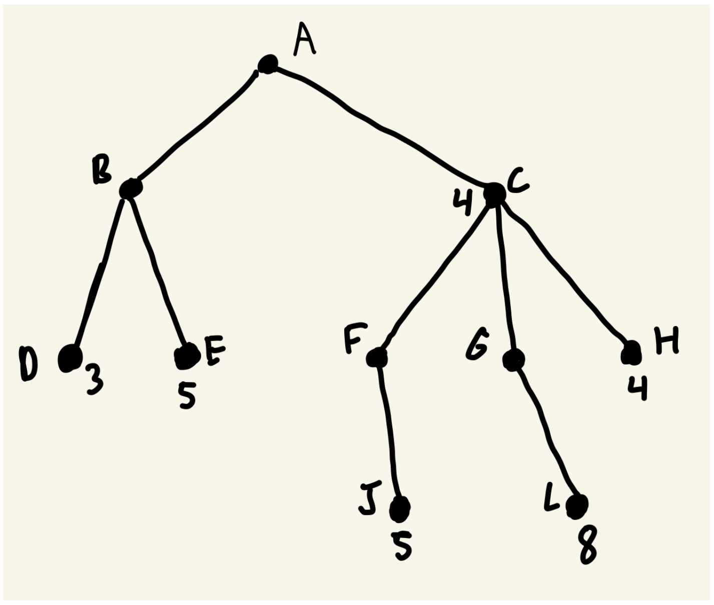
The MAX player here has two moves, to go to state B (we'll call this action a_1) or go to state C (a_2). If MAX chooses a_1, then MAX will end with a utility of 3. If MAX choose a_2, it ends with a utility of 4 as the game progresses towards state H. We can check this result directly using the definition of the MINIMAX function:

$$\begin{aligned} MINIMAX(root) &= max(min(3,5), min(max(min(0,7),5), max(7,8),4)) \\ &= max(3, min(5,8,4)) \\ &= max(3,4) = 4 \end{aligned}$$

Part B
Perform a left-to-right (left branch first, then right branch) alpha-beta pruning on the tree. That is, draw only the parts of the tree that are visited and don't draw branches that are cut off (no need to show the alpha or beta values).



Part C
Do the same thing as in the previous question, but with a right-to-left ordering of the actions. Discuss why different pruning occurs.



The differences in pruning arise because we are visiting the nodes in a different order. When going from left to right, the left node is checked with other states to see if pruning is a possibility, and if so, the right subtree is pruned. Conversely, going from right to left means that the right node is checked first and left subtrees are pruned. This difference in order is what changes the result. For example, when pruning left to right, we visited node K before node L. Seeing as the utility of node K was 7, we pruned off the right subtree of G (as the MIN player would rather go to node F regardless of what node K was). Similarly, when going from right to left, the left subtree of G was pruned (as the MIN player would have opted for node H).

Problem 8

Which of the following are admissible, given admissible heuristics h_1, h_2? Which of the following are consistent, given consistent heuristics h_1, h_2? Justify your ans

Part A

$h(n) = min(h_1(n), h_2(n))$

Assume h_1 and h_2 are admissible.

$$\begin{aligned} h(n) &= min(h_1(n), h_2(n)) \\ &\leq min(h^*(n), h^*(n)) \\ &= h^*(n) \end{aligned}$$

So, h is also admissible. Now assume h_1 and h_2 are consistent.

$$\begin{aligned} h(n) &= min(h_1(n), h_2(n)) \\ &\leq min(c(n,a,n') + h_1(n'), c(n,a,n') + h_2(n')) \\ &= c(n,a,n') + min(h_1(n'), h_2(n')) \\ &= c(n,a,n') + h(n') \end{aligned}$$

The last line follows from the definition of h. So, h is also consistent.

Part B

$h(n) = \omega h_1(n) + (1-\omega)h_2(n)$, where $0 \leq \omega \leq 1$.

Assume h_1 and h_2 are admissible.

$$\begin{aligned} h(n) &= \omega h_1(n) + (1-\omega)h_2(n) \\ &\leq \omega h^*(n) + (1-\omega)h^*(n) \\ &= \omega h^*(n) - \omega h^*(n) + h^*(n) \\ &= h^*(n) \end{aligned}$$

So, h is also admissible. Now assume h_1 and h_2 are consistent.

$$\begin{aligned} h(n) &= \omega h_1(n) + (1-\omega)h_2(n) \\ &\leq \omega(c(n,a,n') + h_1(n')) + (1-\omega)(c(n,a,n') + h_2(n')) \\ &= \omega c(n,a,n') + \omega h_1(n') + c(n,a,n') + h_2(n') - \omega c(n,a,n') - \omega h_2(n') \\ &= c(n,a,n') + \omega h_1(n') + h_2(n') - \omega h_2(n') \\ &= c(n,a,n') + \omega h_1(n') + (1-\omega)h_2(n') \\ &= c(n,a,n') + h(n') \end{aligned}$$

The last line follows from the definition of h. So, h is also consistent.

Part C

$h(n) = max(h_1(n), h_2(n))$

Problem 8 continued on next page. . .

Assume h_1 and h_2 are admissible.

$$\begin{aligned} h(n) &= max(h_1(n), h_2(n)) \\ &\leq max(h^*(n), h^*(n)) \\ &= h^*(n) \end{aligned}$$

So, h is also admissible. Now assume h_1 and h_2 are consistent.

$$\begin{aligned} h(n) &= max(h_1(n), h_2(n)) \\ &\leq max(c(n,a,n') + h_1(n'), c(n,a,n') + h_2(n')) \\ &= c(n,a,n') + max(h_1(n'), h_2(n')) \\ &= c(n,a,n') + h(n') \end{aligned}$$

The last line follows from the definition of h. So, h is also consistent.

Part D
Which of these heuristics, a, b, or c, would you choose?
Given that all three of these heuristics satisfy our needs for optimality (being either admissible or consistent), we want to pick the heuristic that is the "largest," or dominates the others. The perfect heuristic would be one where $h(n) = h^*(n)$ for all n, as the algorithm would travel along the optimal path. Getting as close to that would be ideal. So, we can immediately rule out the heuristic from part A, as part C is clearly bigger for all n. Our claim is that the heuristic from part C (call it h_c) dominates the one from part B (h_b). To show this, we prove that the function $h_c - h_b$ is always positive for all n.

$$h_c(n) - h_b(n) = max(h_1(n), h_2(n)) - \omega h_1(n) - (1-\omega)h_2(n)$$

$$= \begin{cases} h_2(n) - \omega h_1(n) + \omega h_2(n) - h_2(n) & h_2(n) > h_1(n) \\ h_1(n) - \omega h_1(n) + \omega h_2(n) - h_2(n) & h_1(n) > h_2(n) \end{cases}$$

$$= \begin{cases} \omega(h_1(n) + h_2(n)) & h_2(n) > h_1(n) \\ (1-\omega)(h_1(n) - h_2(n)) & h_1(n) > h_2(n) \end{cases}$$

which is clearly always positive. So the best heuristic function is h_c.

Problem 9

Simulated annealing is an extension of hill climbing, which uses randomness to avoid getting stuck in local maxima and plateaux.

Part A

For what types of problems will hill climbing work better than simulated annealing? In other words, when is the random part of simulated annealing not necessary?

Hill climbing is superior to simulated annealing when the value function has only one maximum throughout the search space. In that case, there is no benefit to the randomness of simulated annealing, as it specifically aims to overcome getting stuck at local extrema by making random jumps, but we don't need this if the local extrema we get stuck at is exactly the global extrema we aim to find.

Part B

For what types of problems will randomly guessing the state work just as well as simulated annealing? In other words, when is the hill-climbing part of simulated annealing not necessary?

When there are no extrema and the value function is flat, then hill climbing has no advantage as there are no hills. Any state in the search space is equally optimal, so randomly guessing would work just as well. Another case in which the hill climbing part isn't necessary is when the spread of values is random or doesn't have any good structure likes hills or continuous curves. An example could be a space in which every value except one is constant, and that other value is some positive integer, and thus the global maximum. In that case, randomly guessing the state will eventually lead us towards the global maximum, because there are no hills or structures to climb.

Part C

Reasoning from your answers to parts (a) and (b) above, for what types of problems is simulated annealing a useful technique? In other terms, what assumptions about the shape of the value function are implicit in the design of simulated annealing?

Simulated annealing is most useful when a range of local maxima and plateaus exist in the value function, with definite structures like hills. In this case, simulated annealing can leverage its randomness to explore the search space around nonglobal maxima. Furthermore, you'd like this structure to have multiple maxima, so that there is a use for annealing. For example, polynomials would be a good shape to utilize simulated annealing on.

Part D

As defined in your textbook, simulated annealing returns the current state when the end of the annealing schedule is reached and if the annealing schedule is slow enough. Given that we know the value (measure of goodness) of each state we visit, is there anything smarter we could do?

Problem 9 continued on next page. . .

We can store the current global maximum and its state in our memory, and update it if a bigger maximum is found. Since we know the value of each state, we can perform these comparisons. This allows us to determine the true global maximum in case annealing landed us in the wrong spot through an unlucky probability outcome, and the current state at the end of the annealing schedule isn't the global maximum but we had discovered this maximum earlier in our process.

Part E

Simulated annealing requires a very small amount of memory, just enough to store two states: the current state and the proposed next state. Suppose we had enough memory to hold two million states. Propose a modification to simulated annealing that makes productive use of the additional memory. In particular, suggest something that will likely perform better than just running simulated annealing a million times consecutively with random restarts. [Note: There are multiple correct answers here.]

For each random jump that we make, store the state (and its objective function value). Make sure that for each random jump we perform, this jump does not land on a space whose value is stored already. This allows us to avoid repeating the same jump over and over again, and allows us to determine new maxima faster. Like how a closed list in a search algorithm prevents wasting time searching the same thing, storing each visited state ensures that more of the state space is covered in less time, allowing us to converge on a solution faster and more efficiently. Furthermore, we can select a global maximum from all the states that we have stored in memory to ensure that what we end with is indeed the global maximum value.

Part F

Gradient ascent search is prone to local optima just like hill climbing. Describe how you might adapt randomness in simulated annealing to gradient ascent search avoid trap of local maximum.

We can use the ideas of simulated annealing to improve the robustness of gradient ascent. We start by first involving an appropriate schedule mapping from time to temperature. Rather than deterministically moving up the direction of the gradient, we can randomly sample a direction and find the slope in that direction. If the slope is positive, we move up in that direction, otherwise we only move in that direction with the same probability as defined in the normal simulated annealing algorithm. In this way, the algorithm is less likely to get stuck in local minima, as instead of taking every direction into consideration at the same time (and running the risk of getting caught in a local maximum that exists in any one of those dimensions), the algorithms only concerns itself with one direction at a time, and so can possibly avoid extrema that exist in the other directions.