Assignment 2

Search Problems in A.I.

Abid Azad (netID: aa2177, RUID: 202005452)

CS440, March 19th, 2024

Contents

1 Problem 1

Problem 1: Trace the operation of A* search (use the tree version, i.e. without using a closed list) applied to the problem of getting to Bucharest from Lugoj using the straight-line distance heuristic. That is, show the sequence of nodes that the algorithm will consider and the f, g, and h score for each node. You don't need to draw the graph, just write down a sequence of (city, f(city), g(city), h(city)) in the order in which the nodes are expanded.

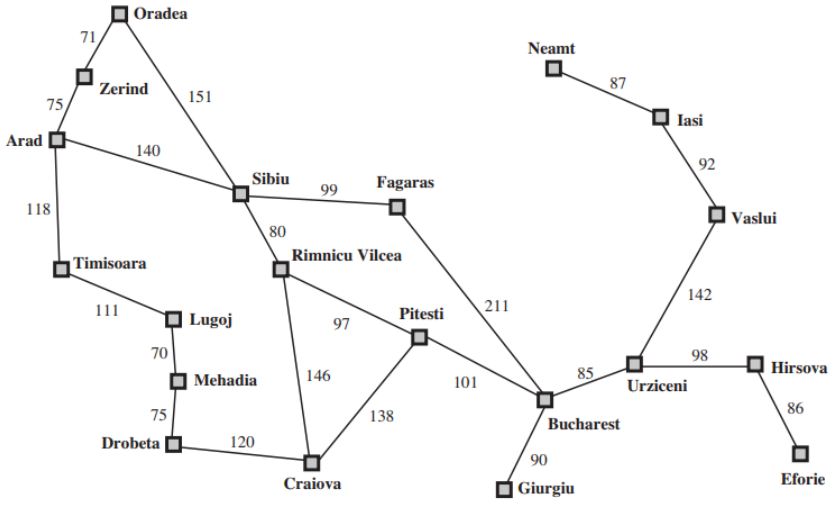

Figure 1: A simplified road map of part of Romania indicating distances between different cities.

Starting from the city Lugoj and aiming for Bucharest, we trace the A* search using the straight-line distance heuristic. We'll list all the possible states considered by the algorithm and highlight the nodes chosen for expansion.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 2: Straight-line distances to Bucharest

Step 1:

(Lugoj, 244, 0, 244)

Open Nodes: Lugoj

Expand Lugoj.

This step initializes the search with Lugoj as the starting node. The f-score (244) is the sum of g-score (0) and h-score (244), where g-score represents the cost from the start node to the current node (0 as it's the start node), and h-score represents the heuristic estimate of the cost from the current node to the goal (244).

Step 2:

(Mehadia, 311, 70, 241)(Timisoara, 440, 111, 329)

Open Nodes: Mehadia, Timisoara

Expand Mehadia.

Here, we expand Mehadia from Lugoj. Mehadia has a lower f-score compared to Timisoara, hence it's chosen for expansion. Mehadia's g-score (70) represents the cost from Lugoj to Mehadia, and the h-score (241) is the heuristic estimate of the remaining distance to Bucharest.

Step 3:

(Drobeta, 387, 145, 242)

Open Nodes: Drobeta, Timisoara

Expand Drobeta.

Drobeta is expanded from Mehadia. It has a higher f-score compared to Timisoara, so it's chosen for expansion. Drobeta's g-score (145) represents the cost from Lugoj to Drobeta, and the h-score (242) is the heuristic estimate of the remaining distance to Bucharest.

Step 4:

(Craiova, 425, 265, 160)

Open Nodes: Craiova, Timisoara

Expand Craiova.

Craiova is expanded from Drobeta. It has a lower f-score compared to Timisoara, hence it's chosen for expansion. Craiova's g-score (265) represents the cost from Lugoj to Craiova, and the h-score (160) is the heuristic estimate of the remaining distance to Bucharest.

Step 5:

(Rimnicu Vilcea, 604, 411, 193)(Pitesti, 503, 403, 100)

Open Nodes: Timisoara, Pitesti, Rimnicu Vilcea

Expand Timisoara (f value of Timisoara is lower than Pitesti's).

From Craiova, Timisoara is expanded as it has the lowest f-score among the open nodes. Timisoara's g-score (366) represents the cost from Lugoj to Timisoara, and the h-score (238) is the heuristic estimate of the remaining distance to Bucharest.

Step 6:

(Arad, 595, 229, 366)

Open Nodes: Pitesti, Arad, Rimnicu Vilcea

Expand Pitesti (f value of Pitesti is lower than Arad's).

Pitesti is expanded from Timisoara as it has the lowest f-score among the open nodes. Pitesti's g-score (473) represents the cost from Lugoj to Pitesti, and the h-score (30) is the heuristic estimate of the remaining distance to Bucharest.

Step 7:

(Bucharest, 504, 504, 0)

Open Nodes: Bucharest, Arad, Rimnicu Vilcea

Expand Bucharest.

Bucharest is expanded from Pitesti as it has the lowest f-score among the open nodes. Bucharest's g-score (504) represents the total cost from Lugoj to Bucharest, and the h-score (0) is the heuristic estimate of the remaining distance to Bucharest.

Step 8:

(Bucharest, 504, 504, 0)

Open Nodes: Arad, Rimnicu Vilcea

We stop here as the goal (Bucharest) is reached.

The search algorithm terminates as the goal state (Bucharest) is expanded. No further nodes need to be expanded since the goal has been reached.

2 Problem 2

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

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

Breadth First Search: In breadth-first search, the algorithm explores all the neighbor nodes at the present depth before moving on to the nodes at the next depth level. Starting from the initial state 1, the search expands outward level by level until it reaches the goal state 11. The order of states visited in breadth-first search is as follows:

$$1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11$$

Depth Limited Search with Limit 3: Depth-limited search limits the depth of exploration to a predefined level. Here, with a depth limit of 3, the search starts from the initial state 1 and explores its successors until the depth limit is reached or the goal state 11 is found. The order of states visited in depth-limited search with a limit of 3 is as follows:

$$1 \to 2 \to 4 \to 8 \to 9 \to 5 \to 10 \to 11$$

Iterative Deepening Search: Iterative deepening search combines the benefits of depth-first and breadth-first search strategies by gradually increasing the depth limit with each iteration until the goal is found. It performs depth-limited searches with increasing depth limits until the solution is found. The order of states visited in iterative deepening search for each iteration is as follows:

First Iteration (Limit 1): $1 \rightarrow 2 \rightarrow 3$

Second Iteration (Limit 2): $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 7$

Third Iteration (Limit 3): $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 9 \rightarrow 5 \rightarrow 10 \rightarrow 11$

(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?

Bidirectional search is effective for this problem as it explores the search space from both the start and goal nodes concurrently, meeting in the middle. This approach can significantly reduce the search space compared to traditional search methods.

To illustrate bidirectional search for this problem:

- **Forward Direction:** Starting from node 1, we explore forward successors:
  - Begin at node 1. Forward successors: 2, 3.
  - Move to node 2. Forward successors: 4, 5.
  - Move to node 3. Forward successors: 6, 7.

- **Backward Direction:** Starting from node 11 (the goal), we explore backward predecessors:
  - Begin at node 11. Backward predecessors: 5.
  - Move to node 5. Backward predecessors: 2.

The bidirectional search meets when the nodes explored from both directions intersect. In this case, the intersection occurs at node 5. Therefore, the search path from the start node (1) to the goal node (11) is found efficiently.

Branching Factor:

- In the forward direction, the branching factor is 2 because each node has two successors.
- In the backward direction, the branching factor is 1 since each node has only one predecessor.

Bidirectional search leverages this low branching factor to efficiently explore the search space in both directions, converging towards the goal state.

3 Problem 3

Problem 3: Which of the following statements are correct and which ones are wrong?

1. Breadth-first search is a special case of uniform-cost search.
   - **True:** Uniform-cost search expands to the lowest cost path, similar to breadth-first search which explores nodes level by level. When all path costs are equal, uniform-cost search behaves identically to breadth-first search.

2. Depth-first search is a special case of best-first tree search.

   - **True:** In best-first tree search, nodes with the lowest cost or heuristic value are expanded first, akin to depth-first search. If the heuristic function is such that it equals the negative of the depth, best-first search behaves the same as depth-first search.

3. Uniform-cost search is a special case of A* search.

   - **True:** Uniform-cost search and A* search both maintain an open list of states and a cost function. In A* search, the cost function includes both the actual cost (g-value) and a heuristic estimate. If the heuristic estimate is zero, A* search behaves the same as uniform-cost search.

4. Depth-first graph search is guaranteed to return an optimal solution.

   - **False:** Depth-first search does not guarantee an optimal solution; it returns the first path it encounters, not necessarily the best one.

5. Breadth-first graph search is guaranteed to return an optimal solution.

   - **False:** Breadth-first search does not ensure an optimal solution unless all edge costs are equal. It explores all nodes at the same level before moving to the next level.

6. Uniform-cost graph search is guaranteed to return an optimal solution.

   - **True:** When uniform-cost search selects a node for expansion, it ensures that the optimal path to that node is found.

7. A* graph search is guaranteed to return an optimal solution if the heuristic is consistent.

    - **True:** A* search guarantees an optimal solution when the heuristic is consistent, meaning it never overestimates the cost to reach the goal.

8. A* graph search is guaranteed to expand no more nodes than depth-first graph search if the heuristic is consistent.

    - **False:** Depth-first search can potentially reach sub-optimal solutions faster than A* search, even with a consistent heuristic.

9. A* graph search is guaranteed to expand no more nodes than uniform-cost graph search if the heuristic is consistent.

    - **True:** A* search expands no more nodes than uniform-cost search when the heuristic is consistent. In the worst case scenario where the heuristic always returns 0, both A* and uniform-cost search will expand the same number of nodes.

4 Problem 4

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.

- Advantage of Iterative Deepening over BFS: Iterative deepening has a lower space complexity compared to breadth-first search (BFS). Iterative deepening has a space complexity of $O(bd)$, where b is the branching factor and d is the depth of the solution, which is less than or equal to the space complexity of BFS, which is also $O(b^d)$. This means that iterative deepening requires less memory to store the nodes in the search tree.

- Disadvantage of Iterative Deepening compared with BFS: Iterative deepening may lead to redundant computations. In each iteration, it explores nodes to a certain depth, potentially revisiting nodes already explored in previous iterations. This redundancy increases the runtime compared to BFS, which avoids redundant computations by exploring each node only once.

5 Problem 5

Problem 5: Prove that if a heuristic is consistent, 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).

Proof: Assume that $x(n)$ represents the cost of the cheapest path from node n to the final node. We aim to prove by induction that $h(n) \leq x(n)$, where $h(n)$ is the heuristic value for node n.

Induction: Suppose n is y steps away from the goal. Then, there exists a successor of n, denoted as n', created by some action a, such that n' lies on the optimal path from n to the goal and n' is $y-1$ steps away from the goal. We can express this relationship as: $h(n) \leq c(n, a, n') + h(n')$. By the induction hypothesis, we know $h(n') \leq x(n')$, thus:

$$h(n) \leq c(n, a, n') + x(n') = x(n)$$

Since n' is on the optimal path from n to the goal, we conclude that the heuristic is both consistent and admissible.

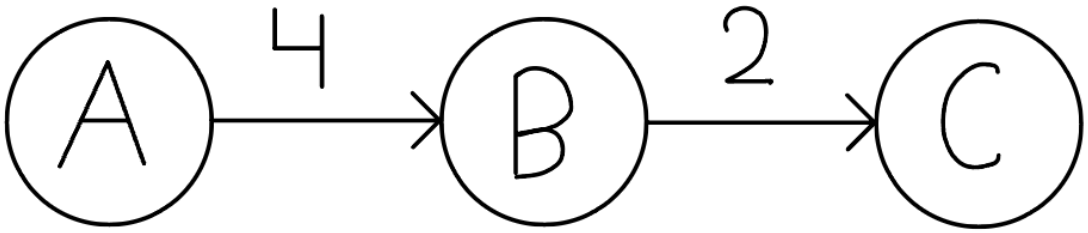

Figure 3: Diagram Example of Consistency Contradiction

In the provided diagram, C is the goal and A is the start, thus, $x(C) = 0$, $x(B) = 2$, and $x(A) = 6$. For the heuristic to be admissible, we assign heuristic values such that $h(C) = 0$, $h(B) \leq 2$, and $h(A) \leq 6$. Let's assign arbitrary values: $h(C) = 0$, $h(B) = 1$, and $h(A) = 6$. If this heuristic were consistent, it should satisfy $h(A) \leq h(B) + c(A, a, B)$. However, substituting the values, we get $6 \leq 7$, which is not true. Hence, this heuristic is admissible but not consistent.

6 Problem 6

Problem 6: In a Constraint Satisfaction Problem (CSP) search, explain why it is a good heuristic to choose the variable that is most constrained but the value that is least constraining.

In a Constraint Satisfaction Problem (CSP) search, it is advantageous to select the most constrained variable while choosing the least constraining value.

This strategy is beneficial because selecting the most constrained variable increases the likelihood of detecting a failure early in the search process, thereby making the search more efficient. By identifying potential conflicts sooner, we can backtrack and explore alternative paths, leading to faster convergence.

Choosing the least constraining value ensures that the selected variable has the maximum flexibility for future assignments. This reduces the likelihood of conflicts and facilitates smoother progress through the search space.

Explanation: In a CSP, where the problem is defined by a set of variables $X_1, X_2, X_3, \ldots, X_n$ and constraints $C_1, C_2, C_3, \ldots, C_m$, selecting the most constrained variable is crucial as it helps in quickly identifying potential conflicts. This early detection allows the search algorithm to backtrack efficiently, thereby improving the overall search performance. Additionally, ensuring arc consistency helps in efficient constraint propagation, further pruning the search space and reducing the computational burden. Backtracking, on the other hand, occurs when no legal assignment is possible. It proceeds by selecting one value at a time and backtracking variables when no legal values are available, thereby allowing exploration of alternative paths to find a solution.

Example: Consider a scheduling problem where tasks need to be allocated to team

members in a way that satisfies various constraints. Each team member has different constraints regarding the tasks they can perform. For instance, Bob cannot work on Task 1 and Task 2 simultaneously, and Carol must work on Task 3 with at least one other team member present. By selecting the most constrained team member for each task assignment and choosing the least constraining task, the CSP solver can efficiently allocate tasks while satisfying all constraints and ensuring smooth progress through the search space.

7 Problem 7

Problem 7 (10 points): 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.

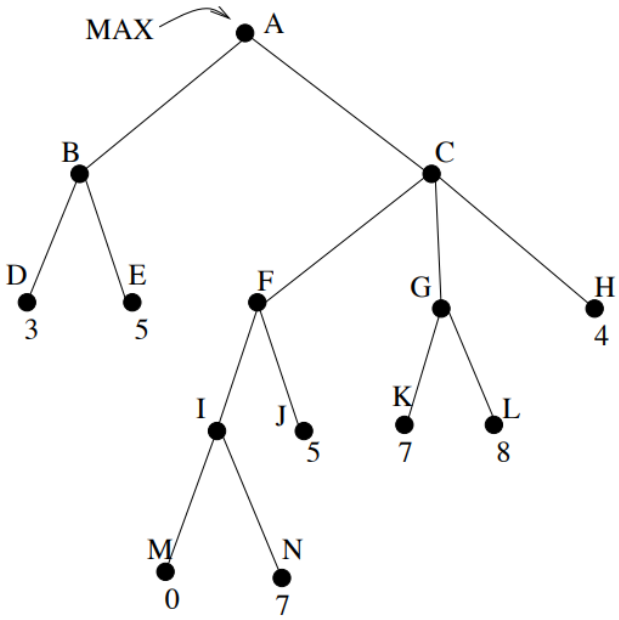

Figure 4: Provided Game Tree

(a) What is the best move for the MAX player using the minimax procedure?

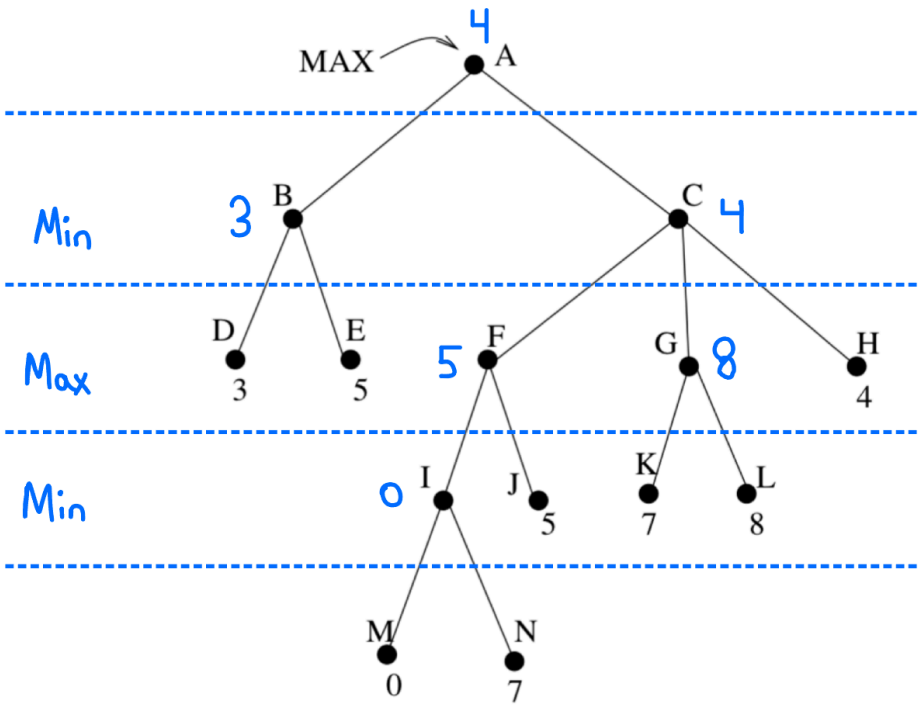

Figure 5: Game Tree (MinMax Procedure)

The best move for the MAX player is 4.

(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 do not draw branches that are cut off (no need to show the alpha or beta values).

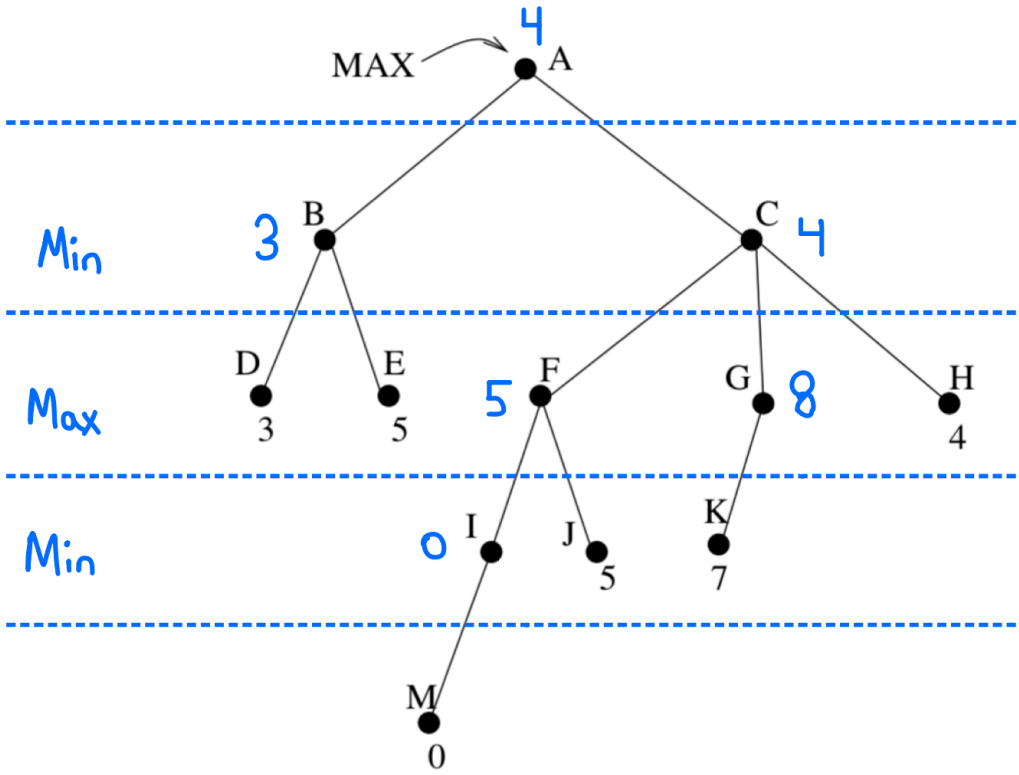

Figure 6: Game Tree (L-to-R Alpha Beta Pruning)

(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.

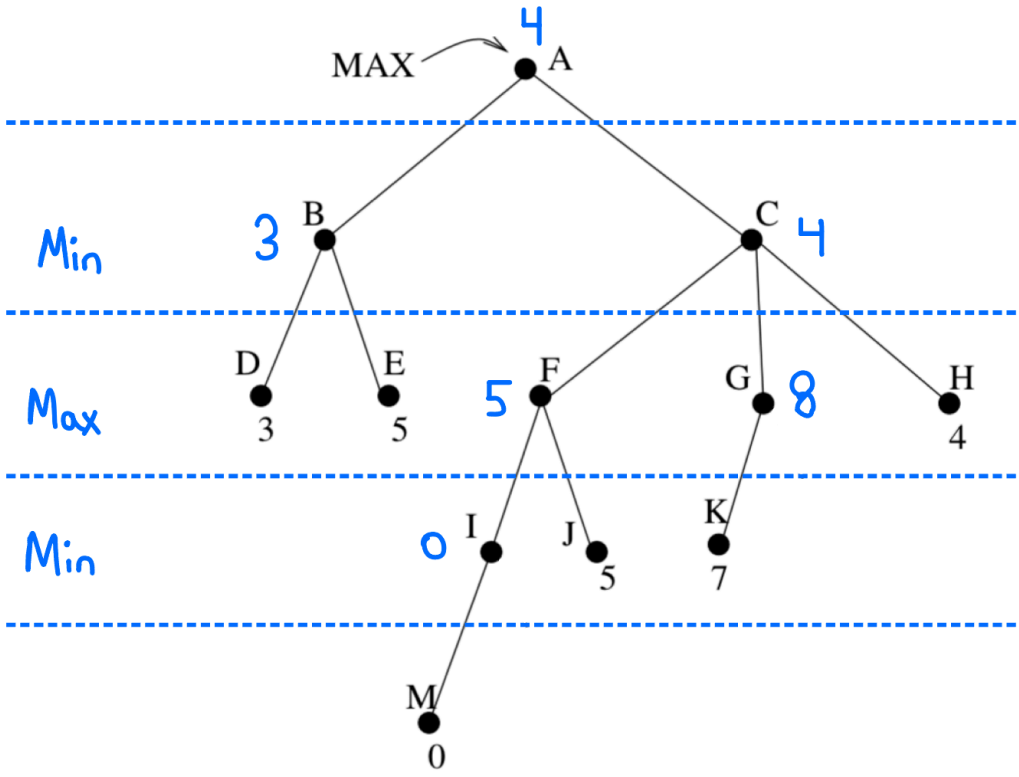

Figure 7: Game Tree (R-to-L Alpha Beta Pruning)

The reason why different pruning occurs when using right-to-left ordering of actions in alpha-beta pruning compared to left-to-right ordering lies in the way alpha-beta pruning exploits the properties of the minimax algorithm and the specific structure of the game tree.

In left-to-right ordering, the algorithm explores the leftmost branches first, which means it initially propagates alpha and beta values from the root node towards the leaves along the leftmost path. As it progresses, it updates alpha and beta values based on the encountered values, and prunes subtrees whenever possible. However, if a cutoff occurs, it may not be able to prune other subtrees along the rightmost path because they have not been explored

yet. This can lead to suboptimal pruning.

Conversely, in right-to-left ordering, the algorithm explores the rightmost branches first. This means that as it propagates alpha and beta values from the root node towards the leaves along the rightmost path, it can potentially prune subtrees earlier than in the left-to-right ordering. This is because, if a cutoff occurs early along the rightmost path, it can update beta values, which may lead to pruning of subtrees along the leftmost path that have already been explored. Therefore, right-to-left ordering may result in more efficient pruning and potentially a shallower search depth compared to left-to-right ordering.

8 Problem 8

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 answer.

(1) $h(n) = \min\{h_1(n), h_2(n)\}$

The heuristic $h(n) = \min(h_1(n), h_2(n))$ is admissible because given that $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$, we can deduce $\min(h_1(n), h_2(n)) \leq h^*(n)$. Additionally, it is consistent because it always underestimates the true cost to reach the goal. This is because if both $h_1(n)$ and $h_2(n)$ are underestimates, then their minimum will also be an underestimate.

(2) $h(n) = wh_1(n) + (1 - w)h_2(n)$, where $0 \leq w \leq 1$

The heuristic $h(n) = wh_1(n) + (1-w)h_2(n)$ is admissible because given that $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$, we can deduce $wh_1(n) + (1-w)h_2(n) \leq h^*(n)$. It is also consistent for the same reason as the previous heuristic.

(3) $h(n) = \max\{h_1(n), h_2(n)\}$

The heuristic $h(n) = \max\{h_1(n), h_2(n)\}$ is admissible because given that $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$, we can deduce $\max\{h_1(n), h_2(n)\} \leq h^*(n)$. It is also consistent as it always overestimates the true cost to reach the goal. This is because if both $h_1(n)$ and $h_2(n)$ are overestimates, then their maximum will also be an overestimate.

Which one of these three new heuristics (a, b or c) would you prefer to use for A*? Justify your answer.

In choosing a heuristic for A*, we prefer $h(n) = \min\{h_1(n), h_2(n)\}$ (heuristic a). This is because it provides a conservative estimate of the true cost to reach the goal, ensuring that A* explores the fewest number of unnecessary nodes while still guaranteeing optimality. Additionally, it is both admissible and consistent, making it a reliable choice for guiding the search process efficiently.

9 Problem 9

Problem 9: Simulated annealing is an extension of hill climbing, which uses randomness to avoid getting stuck in local maxima and plateaux.

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 well-suited for problems such as job shop planning, programmatic programming, circuit structuring, and vehicle routing. It is particularly useful for solving optimization problems where the goal is to find the best state according to the objective function. Hill climbing requires fewer conditions compared to other search methods. However, it does not guarantee a solution, and there is a high possibility of variation between runs due to its random nature.

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?

Randomly guessing the state may work as well as simulated annealing when faced with the need to backtrack to earlier nodes, make large jumps, or explore multiple directions simultaneously. Hill climbing, being a local method, can get stuck at local maxima and be inefficient in large problem spaces, failing to reach the global maximum.

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 valuable for finding global optima in the presence of numerous local optima. It is often used in discrete solution spaces, such as the traveling salesman problem and VLSI routing. Simulated annealing randomly selects moves and accepts worse moves with a probability proportional to the change in the objective function. This makes it effective for exploring large solution spaces and escaping local optima.

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?

One improvement could be to continuously update the maximum value encountered during the simulated annealing process. If the algorithm ends with a lower value, the recorded higher value is likely to be the global maximum.

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 efficient use of the additional memory.

With increased memory, we can store all states visited during simulated annealing. We continuously update and compare them to find the maximum value encountered. If the algorithm terminates with a lower value, we can compare it with the recorded maximum or check the saved states to return the optimal state.

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 to avoid the trap of local maximum.

To adapt simulated annealing to gradient ascent search, we can randomly select gradients. If the gradient is positive, move to the new state. However, if the gradient is negative, move to the new state with a probability proportional to the gradient. Over time, decrease the likelihood of accepting locally bad moves, making it less likely to get stuck in local optima.