CS 440 : Intro to Artificial Intelligence

Search Problems for AI

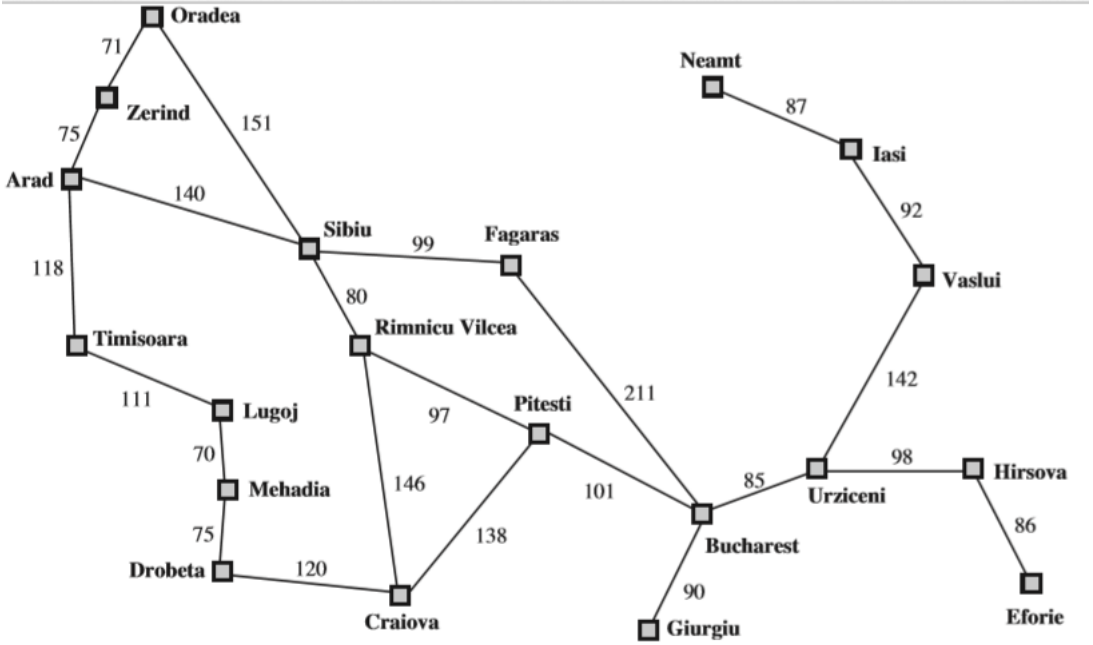
Devvrat Patel and Shubham Mittal

October 28, 2019

Abstract

For this project we will be solving some AI related problem with detailed explanation.

Problem 1 (20 points): Trace the operation of A* search (the tree version) 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.



1 Graph.PNG

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

1 table.PNG

Here, the starting state is Lugoj and the goal state is Bucharest. We will be listing all the possible states that the algorithm will consider and then list the ones that the algorithm will pick.

1. (Lugoj,244,0,244)
   Open Nodes - Lugoj.
   Expand Lugoj.

2. (Timisoara,440,111,329)
   (Mehadia,311,70,241)
   Open Nodes – Mehadia, Timisoara.
   Expand Mehadia.

3. (Drobeta,387,145,242)
   We will not be considering Lugoj here as it has already been visited.
   Open Nodes – Drobeta, Timisoara.
   Expand Drobeta.

4. (Craiova,425,265,160)
   Open Nodes – Craiova, Timisoara.
   Expand Craiova.

5. (Pitesti,503,403,100)
   (Riminicu Vilcea,604,411,193)
   Open Nodes – Timisoara, Pitesti, Riminicu Vilcea.
   We will expand Timisoara here instead of Pitesti as Timisoara's f value is less than Pitesti's and we have not expanded Timisoara yet.

6. (Arad,595,229,366)
   Open Nodes – Pitesti, Arad, Riminicu Vilcea.
   We will expand Pitesti now as the f value of Pitesti is less than Arad's.

7. (Bucharest,504,504,0)
   Open Nodes – Bucharest, Arad, Riminicu Vilcea.
   Expand Bucharest.

8. (Bucharest,504,504,0)
   Open Nodes – Arad, Riminicu Vilcea.
   We will stop here as we have reached our goal.

Problem 2 (10 points): 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.

(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?

a Breadth First Search – 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11
Depth First Search – 1 -> 2 -> 4 -> 8 -> 9 -> 5 -> 10 -> 11
Iterative Deepening Search –
First Iteration (Limit 1) 1 -> 2 -> 3
Second Iteration (Limit 2) 1 -> 2 -> 4 -> 5 -> 3 -> 6 ->7
Third Iteration (Limit 3) 1 -> 2 -> 4 -> 8 -> 9 -> 5 -> 10 -> 11

b Bidirectional Search would work well for this case as when we are starting from the bottom node n then the next node will be n/2. If we work this example as follows:

- Forward – Start at node 1. Forward list will be 2 and 3.
- Backward – Start at node 11. Backward list will be 5.
- Forward – Start at node 2. Forward list will be 3, 4, and 5.
- Backward – Start at node 5. Backward list will be 2.

The branching factor will be 1 in the reverse direction as when we are going backwards here, we are only going to the parent which is only one. The branching factor in the forward direction will be 2 as when we are going down, we will be exploring the two children of each parent.

Problem 3 (5 points): Which of the following statement are correct and which ones are wrong?

a Breadth-first search is special case of uniform-cost search.
• True. As, the only difference between BFS and UCS is that UCS expands to the lowest path instead of the shallowest node. Also, whenever the cost to go to every child is a constant, then UCS will operate like BFS.

b Depth-first search is a special case of best-first tree search.
• True. As, in the best first tree search, we operate the same way as DFS. The only difference is that best first expands the states with the lowest cost or f value. If the f value is equal to negative of the depth, then best first tree will expand a child, then expand its child as the children will always have a lower f values. This is same as DFS.

c Uniform-cost search is a special case of A* search.
• True. Both Uniform Cost Search and A* have an open list of states, and f value. In Uniform Cost Search, the f value is just the g value whereas in A* the f value is g value plus the heuristic. If the heuristic was always zero, then A* will be same as Uniform-cost search.

d Depth-first graph search is guaranteed to return an optimal solution.
• False. Depth first search does not guarantee optimal solution. It returns the first path that contains the goal, not the best path.

e Breadth-first graph search is guaranteed to return an optimal solution.
• False. Breadth first search does not guarantee optimal solution. It is only optimal if all the costs of the edges are equal.

f Uniform-first graph search is guaranteed to return an optimal solution.
• True. Whenever uniform-cost search selects a node n for expansion, the optimal path for that node is found.

g A* graph search is guaranteed to return an optimal solution if the heuristic is consistent.
• True. As the heuristic is consistent, we can say that A* will return an optimal solution.

h A* graph search is guaranteed to expand no more nodes than depth-first graph search if the heuristic is consistent.
• False. There can be a scenario where Depth first search could go directly to a sub-optimal solution.

i A* graph search is guaranteed to expand no more nodes than uniform-first graph search if the heuristic is consistent.
• True. As the heuristic is consistent A* will not expand more nodes than uniform-first graph search. In the worst scenario where, heuristic function returns 0 for all states, A* and Uniform-first graph search will both expand the same number of nodes, but not more.

Problem 4 (5 points): 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 – Iterative deepening has the space complexity of depth first search which is lesser compared to breadth first search. Iterative Deepening has space complexity of O(bd) whereas breadth first space complexity O(b^d).

Disadvantage – Iterative deepening will repeat computations for every iteration so it will take extra run time compared to Breadth First Search which does not repeat any computations.

Problem 5 (10 points): 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).

Assume that x(n) is the cost of cheapest path from n to the final node. We will prove by induction that h(n) <= x(n).

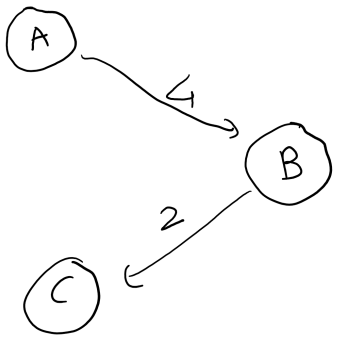
Induction – If n is some y steps away from the goal, then there will be a successor of n (n' will be the successor), created by some action a, such that n' will be on the optimal path from n to the goal and n' will no be only y-1 steps from the goal. Now, we can say that

h(n) <= c(n,a,n') + h(n')

Due to induction we know that, h(n') <= x(n') and we can that

h(n) <= c(n,a,n') + x(n') = x(n)

As n' is the optimal path from n to the goal, we can say that the heuristic is consistent and admissible.



5.png

Here, C is the goal and A is the start. Therefore, x(CO = 0, x(B) = 2, x(A) = 6. For this to be admissible, the heuristic h(C) = 0, h(B) <=2 , and h(A) <= 6. Let us start by assigning random values to the heuristic for this to be admissible. h(C) = 0, h(B) = 1 and h(A) = 6. If this was consistent, then h(A) <= h(B) + c(A,a,B). If we plug the values in we get 6<=7 which is not true. Therefore this is admissible but not consistent.

Problem 6 (10 points): 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.

It is a good heuristic to choose the variables that are the most constrained but the value that is least constrained in a constraint satisfaction problem. This reason behind this is that, by choosing the most constrained variable, there is a chance that this variable will cause a failure, and the variable is most efficient to fail as early as possible

If the least constrained value is chosen in CSP, there will be a chance of occurring more number of future assignments which will avoid conflict.

This problem of constraint satisfaction is well-defined by set of constraints and set of variables $X_1, X_2, X_3, ..., X_n$. $C_1, C_2, C_3, ..., C_m$. Each of the variable X_i contain a non-empty domain D_i of possible value. Each of the constraint X_i involves some variables subsets and identifies the combinations of values which are allowable. A problem state is defined as the assignment of values to some or all variables $(X_i = V_i, X_i = V_i....)$.

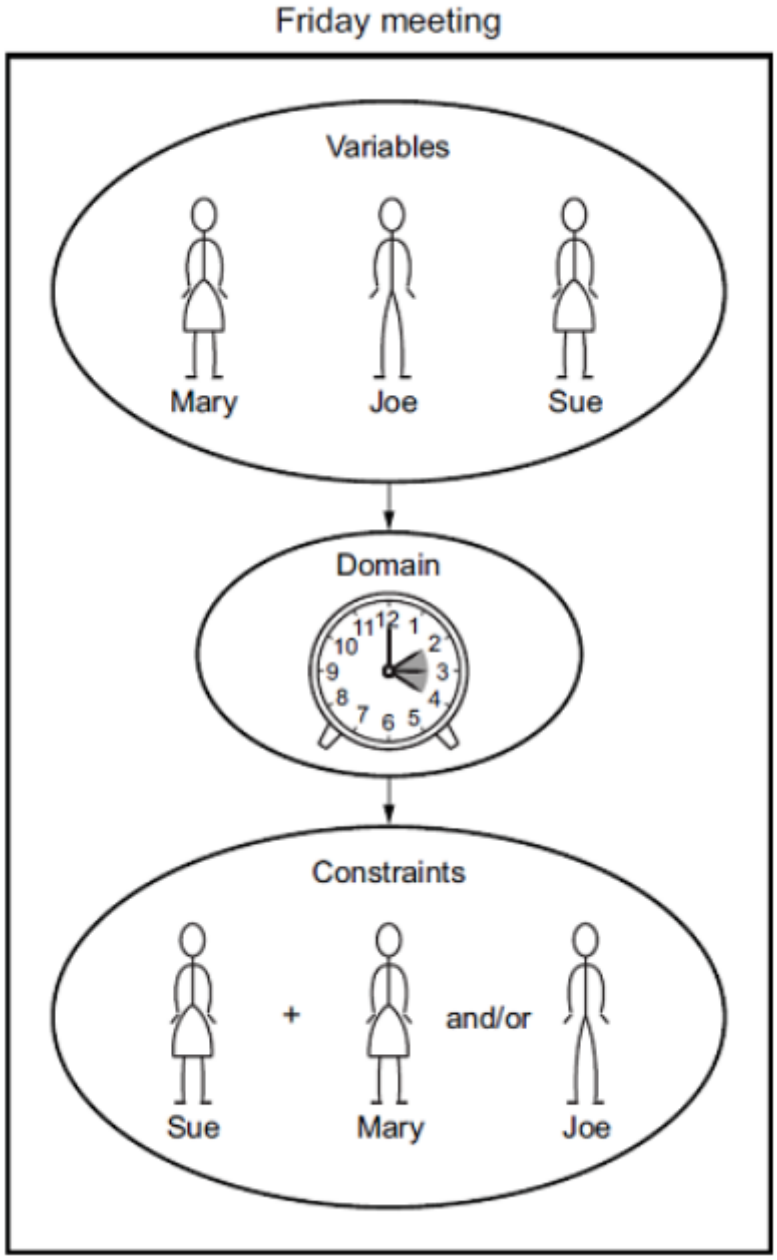
A constraint C_i deals with the subsets of variables and specifies the values combinations which are allowed for the subsets.

When there is no legal assignment Backtracking occurs. It chooses one value at a time and backtracks variables when there are no legal values to assign.

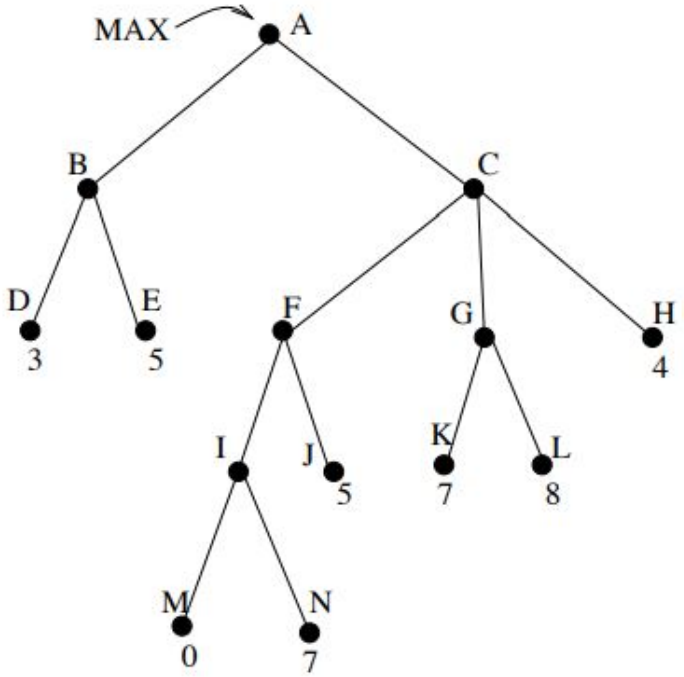
Arc consistency provides the quick method of constraint propagation that is also stronger than the forward checking. Arc points towards the directed arc in the constraint graph.

The method of back-jumping backtracks to the most newest variables in the conflict set. In this situation back-jumping would jump over. Every branch pruned by back-jumping is also pruned by forward-checking.

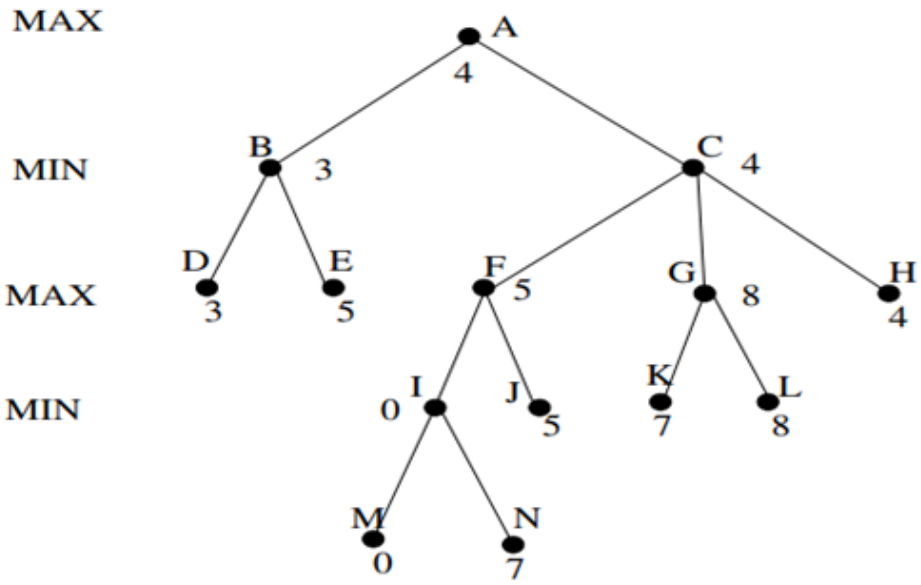
For example. Suppose we are trying to schedule meeting on Friday for Sue, Mary and Joe. Sue should be at the meeting with a minimum of one other person. This problem of Scheduling has three variables Sue Mary, and Joe. Each variable may have their availability hours. E.g. Mary has the domain range of 2, 3 and 4 P.M. The problem also contains 2 constraints. The first One is that Sue must attend the meeting. The second one is that minimum of two people must attend the meeting. Problem solver of constraint satisfaction is given with the three domains, two constraints, and three variables, it solves the problem without requiring that the user explain how. Figure below illustrates this example.


Friday meeting
Variables
Mary
Joe
Sue
Domain
Constraints
Sue
+
Mary
and/or
Joe

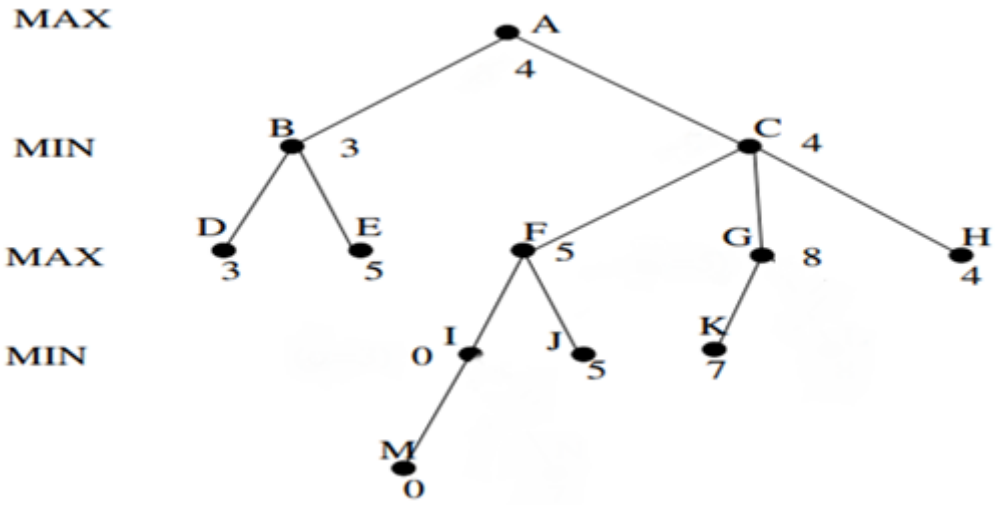
Problem 7 (10 points): Consider the following game tree, where the first move is made by the MAX player and the second move is made by the MIN player.



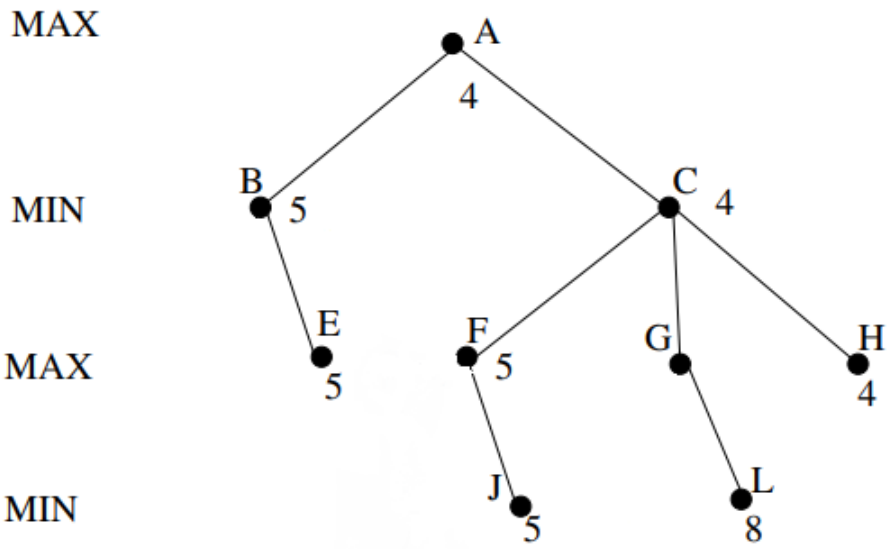
(a) What is the best move for the MAX player using the minimax procedure?



(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 cutoff (no need to show the alpha or beta values).



(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.



Problem 8 (10 points): Which of the following are admissible, given admissible heuristics h1, h2? Which of the following are consistent, given consistent heuristics h1, h2? Justify your answer.

a) $h(n) = minh_1(n), h_2(n)$
$h(n) = min(h_1(n), h_2(n))$ is admissible, since given that $h_1(n) \leq h^*(n) and h_2(n) \leq h*(n)$ we deduce $min(h_1(n), h_2(n) \leq h^*(n))$

b) $h(n) = wh_1(n) + (1-w)h_2(n)$, where 0 ≤ w ≤ 1
$h(n) = wh_1(n) + (1-w)h_2(n)$ is admissible, since given that $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$ we deduce $wh_1(n) + (1-w)h_2(n) \leq h^*(n)$

c) $h(n) = maxh_1(n), h_2(n)$ $h(n) = maxh_1(n), h_2(n)$ is admissible, since given that $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)(n)$h2(n) ≤ h*(n) we deduce max $h_1(n), h_2(n) \leq h^*(n)$
Which one of these three new heuristics (a, b or c) would you prefer to use for A^*? Justify your answer

Being admissible means that the heuristic does not overestimate the effort to reach the goal, i.e., $h(n) \leq h^*(n)h(n) \leq h^*(n)$ for all n in the state space (in the 8-puzzle, this means just for any permutation of the tiles and the goal you are currently considering) where h∗(n)h∗(n) is the optimal cost to reach the target. A∗ provides optimal solutions if h(n)h(n) is admissible is because it sorts all nodes in OPEN in ascending order of f(n)=g(n)+h(n)f(n)=g(n)+h(n) also, because it does not stop when generating the goal but when expanding it:

1. Since nodes are expanded in ascending order of f(n) you know that no other node is more promising than the current one. Remember: h(n) is admissible so that having the lowest f(n)f(n) means that it has an opportunity to reach the goal through a cheaper path that the other nodes in OPEN have not. And this is true unless you can prove the opposite, i.e., by expanding the current node.

2. Since A∗ stops only when it proceeds to expand the goal node (as opposed to stop when generating it) you are sure (from the first point above) that no other node leads through a cheaper path to it.

Problem 9 (10 points): 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 method is valuable in Job shop planning, programmed programming, circuit structuring, and routing of vehicle and management of portfolio. It is additionally useful to take care of unadulterated advancement issues where the goal is to locate the best state as per the goal work. It requires significantly less conditions than other search strategies. There isn't an assurance that the issue will be unraveled all or, now and again, even an upper time farthest point to acquire the arrangement. Other potential hindrance to this technique is that there is a high plausibility of change between keeps running since it's arbitrary.

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?

Backtrack to some earlier node and try going in a different direction. Make a big jump to try to get in a new section. Moving in several directions at once. Hill climbing is a local method, global information might be encoded in heuristic functions. Hill climbing gets stuck at all peaks, known as local maxima. It can be very inefficient in a large, rough problem space. Hill climbing cannot reach the optimal/best state (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?

It is valuable in finding worldwide optima within the sight of huge quantities of neighbourhood optima. Reproduced strengthening is normally utilized in discrete, however exceptionally huge, arrangement spaces, for example, the arrangement of potential requests of urban communities in the Voyaging Sales rep issue and in VLSI directing. Mimicked toughening picks an irregular move from the area (the review that slope climbing picks the best move from every one of those accessible – at any rate when utilizing steepest plummet (or rising)). On the off chance that the move is superior to its present position, at that point mimicked toughening will consistently take it. On the off chance that the move is more regrettable (for example lesser quality) at that point, it will be acknowledged dependent on some likelihood.

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 thing we can do is we can keep running value for max when we run the simulated annealing. Whenever there is a higher value, we keep updating our max values, while

recording the state. If simulated annealing ends in a lower position, then we will still have the record of higher state which is most likely to be the global maximum here.

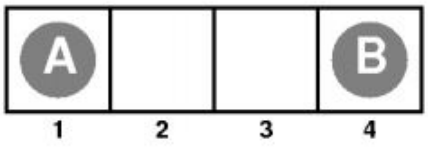
(e) Simulated annealing requires a very small amount of memory, just enough to store two states: the current state and the proposed next state. Suppose we had enough memory to hold two million states. Propose a modification to simulated annealing that makes productive use of the additional memory. In 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.]

We can do a similar thing as the question above. Since, we have a lot memory, we can keep track of all the states we visit and save them. We only update them if we find a higher value. If the simulated annealing is over and we end up with a value which is lower than we what we had then we can compare with the max value and return that or we can check the list of states that we have saved.

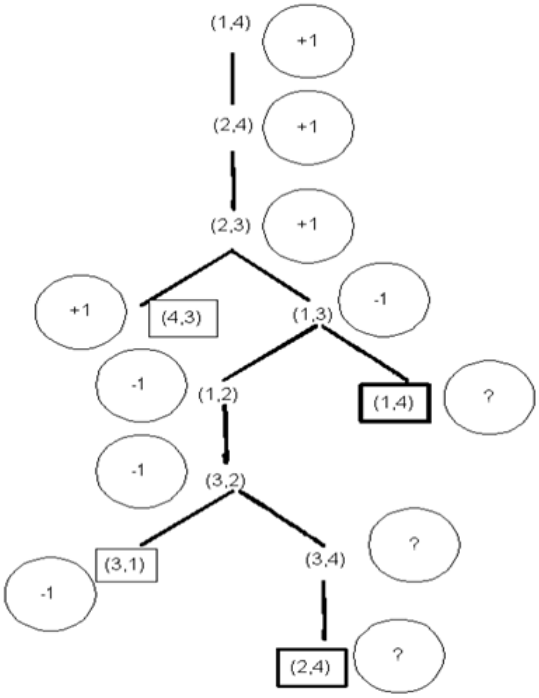
(f) Gradient ascent search is prone to local optima just like hill climbing. Describe how you might adapt randomness in simulated annealing to gradient ascent search avoid trap of local maximum.

Simulated Annealing is inspired twist on random walk. The basic ideas of simulated annealing is instead of choosing the best move, choose a randomly move. Say the change in objective function is d. if d is positive, then move to that state. Otherwise, move to this state with probability proportional to d. So the worse moves (very large negative d) are executed less often, but we have always a better a chance of escaping from local optima. Over time, make it less likely to accept locally bad moves. For adapting the simulated annealing to gradient ascent search, the gradient is chosen randomly. If the gradient is positive move to new state. The gradient can be consider as a heuristic. The gradient tell in which direction the move goes to optimum value. It guide to take decision of direction of move. the only problem is it tell about only local optimum direction. So adapting simulated annealing to gradient ascent make the gradient a good heuristic for searching optimum value.

Problem 10 (10 points) : Consider the two-player game described in Figure.



1. Draw the complete game tree, using the following conventions:
2. Write each state as (s_A, s_B) where s_A and s_B denote the token locations.
3. Put each terminal state in a square box and write its game value in a circle.
4. Put loop states (states that already appear on the path to the root) in double square boxes. Since it is not clear how to assign values to loop states, annotate each with a "?" in a circle.



5. Now mark each node with its backed-up minimax value (also in circle). Explain how you handled the "?" values and why
   The ? values are handled by assuming that an agent with a choice between winning the game and entering a ? state will choose to win. That is min(-1, ?) = -1 and max (+1,?) = +1. If all successors are ? then the backed up value is ?