Assignment 2: Search Problems in AI

Submitted By: Brandon Goldstein (bbg17), Taqiya Ehsan (te137)
Date: 10 March 2023

Problem 1

The following table lists down the nodes and their children along with the corresponding f, g, and h values in the order the nodes were expanded during A* search (tree version, no closed list).

Expanded Node `(city, f(city), g(city), h(city))`	Children Node
(Lugoj, 244, 0, 244)	(Timisoara, 440, 111, 329), (Mehadia, 311, 70, 241)
(Mehadia, 311, 70, 241)	(Drobeta, 387, 145, 242), (Lugoj, 384, 140, 244)
(Lugoj, 384, 140, 244)	(Timisoara, 580, 251, 329), (Mehadia, 451, 210, 241)
(Drobeta, 387, 145, 242)	(Craiova, 425, 265, 160), (Mehadia, 461, 220, 241)
(Craiova, 425, 265, 160)	(Drobeta, 627, 385, 242), (Rimnicu Vilcea, 604, 411, 193), (Pitesti, 503, 401, 100)
(Timisoara, 440, 111, 329)	(Arad, 595, 229, 266), (Lugoj, 466, 222, 244)
(Mehadia, 451, 210, 241)	(Drobeta, 527, 285, 242), (Lugoj, 524, 280, 244)
(Mehadia, 461, 220, 241)	(Drobeta, 537, 295, 242), (Lugoj, 534, 290, 244)
(Lugoj, 466, 222, 244)	(Timisoara, 580, 251, 329), (Mehadia, 533, 292, 241)
(Pitesti, 503, 403, 100)	(Bucharest, 504, 504, 0), (Craiova, 701, 541, 160), (Rimnicu Vilcea, 693, 500, 193)
(Bucharest, 504, 504, 0)	–

Table 1: Sequence of Expanded Nodes for A* Search from Lugoj to Bucharest

During the A* search, the node with the lowest f value is given priority for expansion. Because we don't use a closed list in this version of A*, the same node is expanded multiple times, like Lugoj and Mehadia are expanded 3 times each. The expansion of Pitesti led us straight to the goal location, Bucharest which also had the lowest f value in the fringe. Hence, our A* search was concluded.

The final optimal path is: `Lugoj --> Mehadia --> Drobeta --> Craiova --> Pitesti --> Bucharest`.

Problem 2

a

Breadth First Search: `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11`

Depth-Limited Search (limit 3): `1 -> 2 -> 4 -> 8 -> 9 -> 5 -> 10 -> 11`

Iterative Deepening Search:

Iteration	Order of States Visited
0	`1`
1	`1 -> 2 -> 3`
2	`1 -> 2 -> 4 -> 5 -> 3 -> 6 -> 7`
3	`1 -> 2 -> 4 -> 8 -> 9 -> 5 -> 10 -> 11`

Table 2: Trace of Iterative Deepening Search

b

Bidirectional search would work very well on this problem as we perform this search bottom up and, for every node k, the parent is $\lfloor \frac{k}{2} \rfloor$, which is simple to compute.

Direction	Visited State	Fringe
forward	1	2, 3
backward	11	5
forward	2	3, 4, 5
backward	5	2

Table 3: Trace of Bidirectional Search

The algorithm halts when the search in backward direction reaches node 5 as it is also present in the fringe of forward search.

The branching factor is 2 for forward direction as each node `k` has two children, `2k` and `2k+1`. On the other hand, every node `k` has only one parent, $\frac{k}{2}$, so the branching factor is 1 for backward direction.

Problem 3

a True

b True

c True

d False

e False

f True

g True

h False

i True

Problem 4

One advantage of iterative deepening over BFS is the use of substantially less memory. As iterative deepening only stores the current path in memory, the space complexity of iterative deepening is $O(bd)$. The space complexity of BFS is $O(b^d)$ because it stores the entire tree. In these expressions, b is the maximum branching factor and d is the depth of the shallowest goal state. One disadvantage of iterative deepening as compared with BFS is runtime. Although both iterative deepening and BFS have time complexities of $O(b^d)$, iterative deepening revists the same states many times. This results in the runtime for iterative deepening being worse than BFS in practice for certain cases, such as the worst case.

Problem 5

Let $h(s)$ be any consistent heuristic, and let $c(s, s')$ be the corresponding step cost of moving from state `s` to next state `s'`. By the definition of consistency, $h(s) \leq c(s, s') + h(s')$. To show that `h` is an admissible heuristic, we must show that $h(s) \leq p(s)$ where `p` is the true cost of the shortest path from state `s` to the goal state. Suppose there is no path from `s` to the goal state. In that case, $p(s)$ is infinite, and satisfies $h(s) \leq p(s)$ for any finite $h(s)$. Now suppose that there is some shortest path from `s` to the goal state s_{goal} with intermediate states $s_1, s_2, s_3, ..., s_n$. Since `h` is consistent, $h(s_n) \leq c(s_n, s_{goal}) + h(s_{goal}) = c(s_n, s_{goal})$, as $h(s_{goal}) = 0$. Similarly, $h(s_{n-1}) \leq c(s_{n-1}, s_n) + h(s_n) \leq c(s_{n-1}, s_n) + c(s_n, s_{goal})$. By induction on `s`, $h(s_k) \leq c(s_k, s_{k+1}) + ... + c(s_{n-1}, s_n) + c(s_n, s_{goal})$ for any `k` between 1 and `n-1`. But the shortest path is $(s, s_1, s_2, ..., s_n, s_{goal})$, and so $p(s_k) = c(s_k, s_{k+1}) + ... + c(s_{n-1}, s_n) + c(s_n, s_{goal})$ therefore satisfying the inequality $h(s_k) \leq p(s_k)$. Hence, if a heuristic is consistent, it must also be admissible.

Example: Let us take a simple 3-node graph where the values on the edges are the path costs and `h(S)`, `h(A)` are the pre-decided heuristics.

$$\texttt{S} \overset{1}{\longrightarrow} \texttt{A} \overset{3}{\longrightarrow} \texttt{G}$$

`h(S)= 4`
`h(A)= 1`

The heuristic in this problem is admissible as it never overestimates the true cost of the shortest path from a node to the goal node. However, it is not consistent since the step cost between states S and A is overestimated. The estimated step cost is $h(S) - h(A) = 4 - 1 = 3$, while the true step cost is 1. In other words, it is evident that $4 > 1 + 1 = h(S) > cost(S, A) + h(A)$, which violates the triangle inequality.

This proof and example are based on *Proof: Every Consistent Heuristic is also Admissible by R. Niwa* from `https://rniwa.com/2009-09-06/proof-every-consistent-heuristic-is-also-admissible/` and *CMU School of Computer Science* from `https://www.cs.cmu.edu/~15381-s19/recitations/rec2/rec2_sol.pdf`.

Problem 6

Per the lecture slides, in choosing the variable that is most constrained, we are choosing the variable that can fail quickest. If the variable fails, we can prune the corresponding branch of the search tree and consequently avoid pointless searches through other variables. Moreover, if we select the value that is least constraining, we rule out the fewest choices for the neighboring variables in the constraint graph. This allows us maximum flexibility for future variable assignments and ideally gets us to a solution to the CSP fastest. This answer is based on the lecture slides.

Problem 7

a

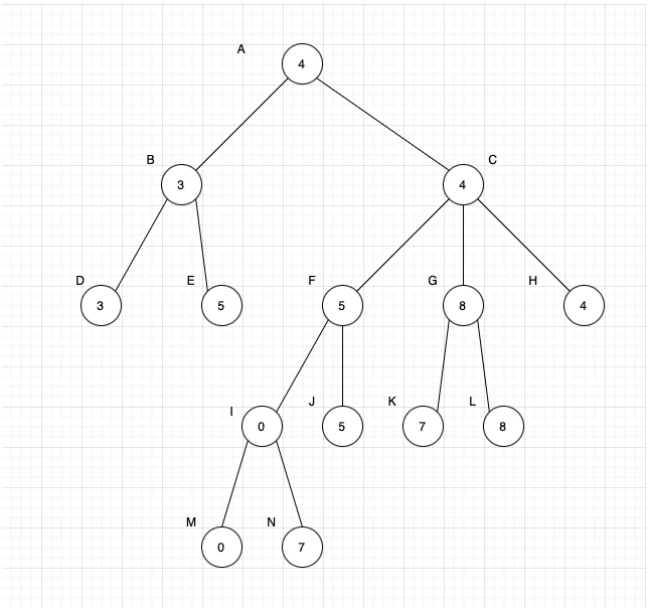

Figure 1: Minimax Procedure

The best move for the max player using the minimax procedure is the action that leads to C, as it results in a payoff of 4. Choosing the action that leads to B only results in a payoff of 3.

b

Please see Figure 2.

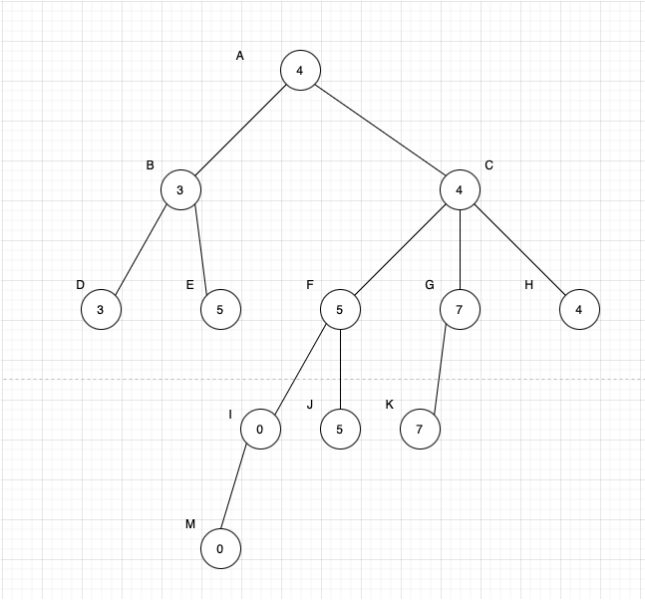

Figure 2: Left-to-Right Alpha-Beta Pruning

c

Please see Figure 3. When pruning left-to-right, we prune N because B has already been set to 3. As I (first child of

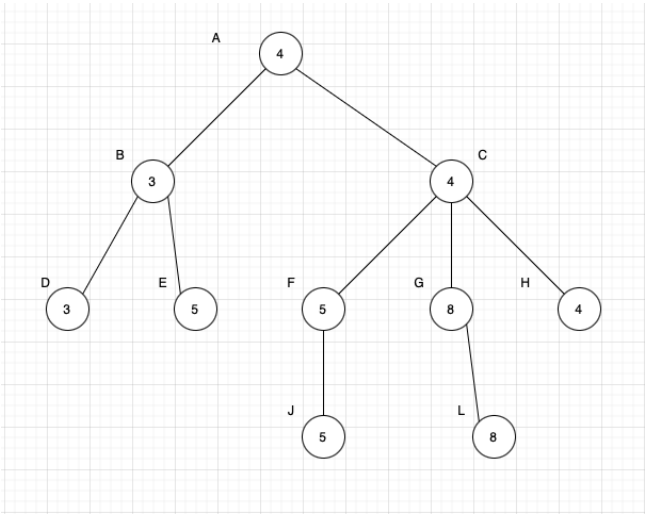

Figure 3: Right-to-Left Alpha-Beta Pruning

F in left-to-right pruning) is a move by the min player and M is set to 0, I must be less than or equal to 0 ($\beta = 0$). However, the max player can already get 3 by selecting B ($\alpha = 3$). Consequently, the max player will never choose I ($3 > 0$), and we need not further explore the children of I (namely, N). When pruning right-to-left, we not only prune N, we prune I (N's parent). As F is a move by the max player and J (first child of F in right-to-left pruning) has already been set to 5, F must be at least 5 ($\alpha = 5$). However, as the value of H has already been set to 4 ($\beta = 4$) and F must be at least 5 ($\alpha = 5$), the min player will never choose F at C ($5 > 4$). As the min player will never choose F at C, we need not further explore the children of F (namely, I). We also prune L when pruning left-to-right. As K (first child of G in left-to-right pruning) is set to 7 and G is a move by the max player, we know G is at least 7 ($\alpha = 7$). However, we also know that F is set to 5 ($\beta = 5$). As F is set to 5 ($\beta = 5$) and G is at least 7 ($\alpha = 7$), the min player will never choose G at C ($7 > 5$). As the min player will never choose G at C, we need not further explore the children of G (namely, L). Lastly, we prune K when pruning right-to-left. As the value of L (first child of G in right-to-left pruning) is set to 8 and G is a move by the max player, we know G is at least 8 ($\alpha = 8$). However, we also know that H is set to 4 ($\beta = 4$). As H is set to 4 ($\beta = 4$) and G is at least 8 ($\alpha = 8$) the min player will never choose G at C ($8 > 4$). As the min player will never choose G at C, we need not further explore the children of G (namely, K).

Problem 8

a

We will prove that, given admissible heuristics h_1 and h_2, $h(n) = min\{h_1(n), h_2(n)\}$ is admissible. As h_1 and h_2 are admissible, $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$ for all n where $h^*(n)$ is the true cost of the shortest path from n to the goal. As $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$ for all n, $h(n) = min\{h_1(n), h_2(n)\} \leq h^*(n)$ for all n. As $h(n) = min\{h_1(n), h_2(n)\} \leq h^*(n)$ for all n, $h(n) = min\{h_1(n), h_2(n)\}$ is admissible.

We will prove that, given consistent heuristics h_1 and h_2, $h(n) = min\{h_1(n), h_2(n)\}$ is consistent. As h_1 and h_2 are consistent, $h_1(n) \leq c(n, a, n') + h_1(n')$ and $h_2(n) \leq c(n, a, n') + h_2(n')$ for all (n, a, n') tuples. As $h_1(n) \leq c(n, a, n') + h_1(n')$ and $h_2(n) \leq c(n, a, n') + h_2(n')$ for all (n, a, n') tuples, $h(n) = min\{h_1(n), h_2(n)\} \leq c(n, a, n') + h_1(n')$ and $h(n) = min\{h_1(n), h_2(n)\} \leq c(n, a, n') + h_2(n')$ for all (n, a, n') tuples. As $h(n) = min\{h_1(n), h_2(n)\} \leq c(n, a, n') + h_1(n')$ and $h(n) = min\{h_1(n), h_2(n)\} \leq c(n, a, n') + h_2(n')$ for all (n, a, n') tuples, $h(n) = min\{h_1(n), h_2(n)\} \leq min\{c(n, a, n') + h_1(n'), c(n, a, n') + h_2(n')\}$ for all (n, a, n') tuples. As $h(n) = min\{h_1(n), h_2(n)\} \leq min\{c(n, a, n')+h_1(n'), c(n, a, n')+h_2(n')\}$ for all (n, a, n') tuples, $h(n) = min\{h_1(n), h_2(n)\} \leq c(n, a, n')+min\{h_1(n'), h_2(n')\}$ for all (n, a, n') tuples. As $h(n) = min\{h_1(n), h_2(n)\} \leq c(n, a, n')+min\{h_1(n'), h_2(n')\}$ for all (n, a, n') tuples, $h(n) = min\{h_1(n), h_2(n)\} \leq c(n, a, n') + h(n')$ for all (n, a, n') tuples. As h_1 and h_2 are consistent, $h_1(n) = 0$ and $h_2(n) = 0$ if n is the goal. As $h_1(n) = 0$ and $h_2(n) = 0$ if n is the goal, $h(n) = min\{h_1(n), h_2(n)\} = 0$ if n is the goal. As $h(n) = min\{h_1(n), h_2(n)\} \leq c(n, a, n') + h(n')$ for all (n, a, n') tuples and $h(n) = min\{h_1(n), h_2(n)\} = 0$ if n is the goal, $h(n) = min\{h_1(n), h_2(n)\}$ is consistent.

b

We will prove that, given admissible heuristics h_1 and h_2, $h(n) = wh_1(n) + (1 - w)h_2(n)$ where $0 \leq w \leq 1$ is admissible. As h_1 and h_2 are admissible, $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$ for all n where $h^*(n)$ is the true cost of the shortest path from n to the goal. As $0 \leq w \leq 1$, $h(n) = wh_1(n)+(1-w)h_2(n) \leq max\{h_1(n), h_2(n)\}$ for all n. As $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$ for all n, $max\{h_1(n), h_2(n)\} \leq h^*(n)$ for all n. As $h(n) = wh_1(n)+(1-w)h_2(n) \leq max\{h_1(n), h_2(n)\}$ for all n and $max\{h_1(n), h_2(n)\} \leq h^*(n)$ for all n, $h(n) = wh_1(n)+(1-w)h_2(n) \leq h^*(n)$ for all n. As $h(n) = wh_1(n)+(1-w)h_2(n) \leq h^*(n)$ for all n, $h(n) = wh_1(n)+(1-w)h_2(n)$ where $0 \leq w \leq 1$ is admissible.

We will prove that, given consistent heuristics h_1 and h_2, $h(n) = wh_1(n) + (1 - w)h_2(n)$ where $0 \leq w \leq 1$ is consistent. As h_1 and h_2 are consistent, $h_1(n) \leq c(n, a, n') + h_1(n')$ and $h_2(n) \leq c(n, a, n') + h_2(n')$ for all (n, a, n') tuples. As $h_1(n) \leq c(n, a, n') + h_1(n')$ and $h_2(n) \leq c(n, a, n') + h_2(n')$ for all (n, a, n') tuples, $wh_1(n) \leq wc(n, a, n') + wh_1(n')$ and $(1 - w)h_2(n) \leq (1 - w)c(n, a, n') + (1 - w)h_2(n')$ for all (n, a, n') tuples. As $wh_1(n) \leq wc(n, a, n') + wh_1(n')$ and $(1 - w)h_2(n) \leq (1 - w)c(n, a, n') + (1 - w)h_2(n')$ for all (n, a, n') tuples, $wh_1(n) + (1 - w)h_2(n) \leq wc(n, a, n') + wh_1(n') + (1 - w)c(n, a, n') + (1 - w)h_2(n')$ for all (n, a, n') tuples. As $wh_1(n) + (1 - w)h_2(n) \leq wc(n, a, n') + wh_1(n') + (1 - w)c(n, a, n') + (1 - w)h_2(n')$ for all (n, a, n') tuples, $h(n) \leq c(n, a, n') + h(n')$ for all (n, a, n') tuples. As h_1 and h_2 are consistent, $h_1(n) = 0$ and $h_2(n) = 0$ if n is the goal. As $h_1(n) = 0$ and $h_2(n) = 0$ if n is the goal, $h(n) = wh_1(n) + (1 - w)h_2(n) = 0$ if n is the goal. As $h(n) \leq c(n, a, n') + h(n')$ for all (n, a, n') tuples and $h(n) = wh_1(n) + (1 - w)h_2(n) = 0$ if n is the goal, $h(n) = wh_1(n) + (1 - w)h_2(n)$ where $0 \leq w \leq 1$ is consistent.

c

We will prove that, given admissible heuristics h_1 and h_2, $h(n) = max\{h_1(n), h_2(n)\}$ is admissible. As h_1 and h_2 are admissible, $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$ for all n where $h^*(n)$ is the true cost of the shortest path from n to the goal. As $h_1(n) \leq h^*(n)$ and $h_2(n) \leq h^*(n)$ for all n, $h(n) = max\{h_1(n), h_2(n)\} \leq h^*(n)$ for all n. As $h(n) = max\{h_1(n), h_2(n)\} \leq h^*(n)$ for all n, $h(n) = max\{h_1(n), h_2(n)\}$ is admissible.

We will prove that, given consistent heuristics h_1 and h_2, $h(n) = max\{h_1(n), h_2(n)\}$ is consistent. As h_1 and h_2 are consistent, $h_1(n) \leq c(n, a, n') + h_1(n')$ and $h_2(n) \leq c(n, a, n') + h_2(n')$ for all (n, a, n') tuples. As $h_1(n) \leq c(n, a, n') + h_1(n')$ and $h_2(n) \leq c(n, a, n') + h_2(n')$ for all (n, a, n') tuples, $h_1(n) \leq max\{c(n, a, n') + h_1(n'), c(n, a, n') + h_2(n')\}$ and $h_2(n) \leq max\{c(n, a, n') + h_1(n'), c(n, a, n') + h_2(n')\}$ for all (n, a, n') tuples. As $h_1(n) \leq max\{c(n, a, n') + h_1(n'), c(n, a, n') + h_2(n')\}$ and $h_2(n) \leq max\{c(n, a, n') + h_1(n'), c(n, a, n') + h_2(n')\}$ for all (n, a, n') tuples, $h(n) = max\{h_1(n), h_2(n)\} \leq max\{c(n, a, n') + h_1(n'), c(n, a, n') + h_2(n')\}$ for all (n, a, n') tuples. As $h(n) = max\{h_1(n), h_2(n)\} \leq max\{c(n, a, n')+h_1(n'), c(n, a, n')+h_2(n')\}$ for all (n, a, n') tuples, $h(n) = max\{h_1(n), h_2(n)\} \leq c(n, a, n') + max\{h_1(n'), h_2(n')\}$ for all (n, a, n') tuples. As $h(n) = max\{h_1(n), h_2(n)\} \leq c(n, a, n') + max\{h_1(n'), h_2(n')\}$ for all (n, a, n') tuples, $h(n) = max\{h_1(n), h_2(n)\} \leq c(n, a, n') + h(n')$ for all (n, a, n') tuples. As h_1 and h_2 are consistent, $h_1(n) = 0$ and $h_2(n) = 0$ if n is the goal. As $h_1(n) = 0$ and $h_2(n) = 0$

if n is the goal, $h(n) = max\{h_1(n), h_2(n)\} = 0$ if n is the goal. As $h(n) = max\{h_1(n), h_2(n)\} \leq c(n, a, n') + h(n')$ for all (n, a, n') tuples and $h(n) = max\{h_1(n), h_2(n)\} = 0$ if n is the goal, $h(n) = max\{h_1(n), h_2(n)\}$ is consistent.

Problem 9

a

For problems with no local maxima, the random part of simulated annealing is not necessary, so hill climbing works better than simulated annealing for problems in which the value function does not exhibit local maxima.

b

For problems in which the area around the global maximum looks no different than other areas, the hill-climbing part of simulated annealing is not necessary, as the topology of the area would not steer the agent to the global maximum under the hill-climbing procedure. So, randomly guessing the state would work just as well as simulated annealing for problems in which the value function does not exhibit a structure wherein the area around the global maximum is different from other areas to the degree that the agent is steered to the global maximum under the hill-climbing procedure.

c

In the design of simulated annealing, we implicitly assume that the shape of the value function is such that local maxima exist and the area around the global maximum is different from other areas to the degree that the agent is steered to the global maximum under the hill-climbing procedure. So, simulated annealing is a useful technique for problems that feature a value function that boasts local maxima and is shaped such that the area around the global maximum is different from other areas to the degree that the agent is steered to the global maximum under the hill-climbing procedure.

d

If we know the value (measure of goodness) of each state we visit, we could keep track of the visited state with the highest value. We would return this state, which has a value at least as high as the current state, when the end of the annealing schedule is reached and if the annealing schedule is slow enough. This is "smarter" because this method results in the return (upon the termination of the algorithm) of the visited state that is at least as good as every local maxima visited, whereas returning the current state at the termination of the algorithm only guarantees a local maxima (which is not necessarily better than other visited local maxima).

e

A notable weakness of simulated annealing is the chance that it will get stuck in a local maxima, especially when the temperature is very low. We propose the following modification to simulated annealing, which both addresses this weakness and makes productive use of the additional memory. We will use the additional memory to store the most recently visited states which were not deemed to be local maxima. Upon reaching what we believe to be a local maxima (as indicated by consecutive states with the same value), we randomly select one state from the additional memory and proceed from that state. This allows the algorithm to escape from local maxima, and the random element of the restart leads the algorithm to avoid getting trapped in the same local maxima. The research of H.Y. Tarawneh, Masri Ayob, and Zulkifli Ahmad in "A Hybrid Simulated Annealing with Solutions Memory for Curriculum-based Course Timetabling Problem" suggests that our proposed modification to simulating annealing makes productive use of the additional memory, enhances the base simulating annealing algorithm, and performs better than just running simulated annealing a million times consecutively with random restarts.

f

In gradient ascent, the agent uses the update rule $x \leftarrow x + \alpha \nabla f$, where α is the step-size parameter and ∇f is the gradient of the objective function at state x. Unfortunately, gradient ascent is susceptible to becoming stuck at local maxima, just like hill-climbing. We can adopt a variant of the strategy employed by simulated annealing to avoid local maxima traps. Specifically, we can, with some probability p use the update rule $x \leftarrow x - \sigma\alpha\nabla f$ and with probability $1 - p$ use the standard update rule $x \leftarrow x + \alpha\nabla f$, where σ is an expansion factor. p would be a function of $\alpha\nabla f$ and a temperature variable T, which itself would follow a schedule such that $T \to 0$ as $time \to \infty$. As T falls, so would p. This variant of the strategy employed by simulated annealing would allow gradient ascent

to avoid local maxima traps. Empirically, the research of Zhicheng Cai in "SA-GD: Improved Gradient Descent Learning Strategy with Simulated Annealing" supports this conclusion.