

Question 1.

a. If we allow the salesperson to “teleport” (transport without cost) to any previously visited city, then the solution for the remaining cities is the minimum spanning tree of the (yet) unvisited cities, the start city, and the current end city.

This can easily be seen: the optimal solution to this teleport version of the problem is to link all of the (unseen plus start and end) cities together with the subgraph of minimum cost. This is exactly a minimum spanning tree problem. We do not care about which city is linked to which; in order to reach a new city, we can “teleport” to the closest city and then take the trip (and associated cost). This relaxation clearly only reduces the total cost. We have added a move, but the original solution — not teleporting — still exists.

b. The straight-line distance places a lower bound on the remaining costs by using the distance from the start city to the city at the end of the current path. If the cities are fully connected and the paths obey the triangle inequality, this is just the cost of the link that turns the path into a circuit. If the cities aren’t fully connected (or if the link costs do not obey the triangle inequality), then we can extend this to be the minimum cost path between the two nodes.

This cost clearly dominates the MST cost. The MST cost connects all of the remaining nodes in a minimal subgraph while the straight-line distance connects just two of those nodes with a minimal subgraph (*i.e.* a path).

Question 2.

a. Any game which is winnable for a player will be won using either evaluation function. For E2, it will select the win which has the largest “margin” of victory. For E1, it will select some arbitrary win. Either way, if a win exists, it will be selected. Alpha-beta search won’t change the result any. It always produces the same result as direct search.

b. The running time for alpha-beta search will be longer in the case of E2. It must find the maximal margin win (not just the first win it comes across). Therefore, it cannot prune as much.

c. We expect E2 to do better. E1 risks picking a situation that is only marginally better for the current player. It could be that the evaluation function miscalculated, and that situation is actually a loss for the current player. Such a reversal in sign is *less* likely to happen when E2 is very large. While E2 might not be a completely accurate prediction of who will win, the larger the absolute value of the metric, the more likely it is to have the correct sign.

d. If the game has chance, things are very different. Consider a chance node with three branches. The first branch results in a token-count difference of +11. The second and third branches each result in a token-count difference of -1. In this case, if we get to this node, the true probability of winning is $\frac{1}{3}$. In general, perhaps not too good. However, with a token-count metric, we will report a value of 3 for this node (the average of 11, -1, and -1). Matters only get worse if the first branch’s token-count difference is larger.

Therefore, the evaluation function must be carefully considered. In general, it must be equal to the probability of winning for the MAX player at a given node (or a positive linear transformation thereof) in order for the chance nodes to evaluate correctly. This is, of course, in general impossible (if we knew the true probability of winning, we wouldn’t need to search at all). However, it does mean that we must try hard to insure that our metric is not only correlated with the chance of winning, but that the correlation is linear in fashion: it can’t just be that bigger scores indicate better positions for MAX, it must also be true that if a score is twice as big, there must be (at least roughly) twice as great a chance for MAX to win.

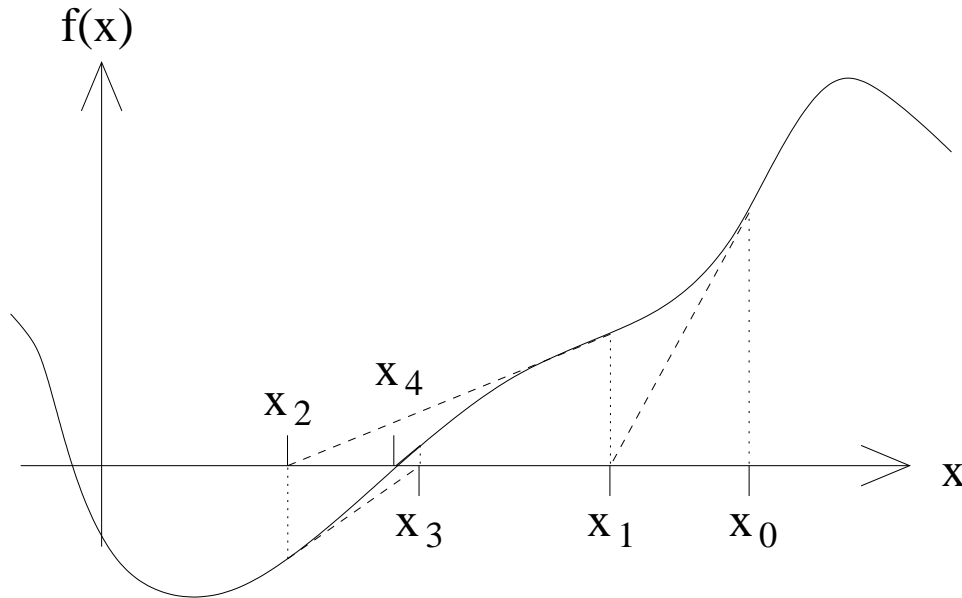
This is true regardless of whether the search can reach the leaves of the game tree.

Question 3.

a. If f is linear, then moving Δx from x_i will cause the function to change by $\Delta x f'(x_i)$. Therefore, to get to a zero crossing, $\Delta x f'(x_i) = -f(x_i)$. So, the update is

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} .$$

b.



c. This procedure might diverge. Consider the function $f(x) = \tanh(x)$ (the hyperbolic tangent). Then $f'(x) = \text{sech}^2(x)$. Therefore, $\Delta x = -\frac{1}{4}(e^{2x} - e^{-2x})$. It should be clear that if x is large, $\Delta x < -2x$ meaning that the next step will move to the other side of the origin (the only zero crossing of $f(x)$) and farther from the origin than the previous point. Therefore, this series diverges. This is because the derivative is too small relative to the distance to the zero crossing and the matter only becomes worse the farther from the zero the point gets.

d. A maximum of g is a zero crossing of f' . So, we can make the following update.

$$x_{i+1} = x_i - \frac{g'(x_i)}{g''(x_i)} .$$

e. This will not always find a maximum. For one, it may have the convergence problems of part c (now in terms of the ratio of the first and second derivative). However, it also might converge to a minimum. Both can be “solved” by never taking a step unless it increases the value of g . If g decreases, we can try decreasing the step size. If g'' is positive, then the step is going to be in the wrong direction and we can revert to gradient ascent.

Question 4.

My implementation is not the fastest. In particular, the closed list is a queue and not a hash table. However, it solves all of the given problems in under 20 seconds.

My state space is the space of sets of position-direction pairs. My heuristic is the maximal Manhattan distance from one of these positions to the goal. This could be improved by taking into account rotational direction. It could also be improved by finding the true distance (using, say, A* search on this simpler state space of just a single position-direction pair). However, the true distance is probably too much time spent thinking about the heuristic for most problems.

Note that the code is about 250 lines (including comments). Much of the A* algorithm is spent taking care of memory carefully. While templates could be used to make a "general version" of A*, that seems odd. Templates require the recompilation of the code for each different object type. Inheritance really works better here allowing the definition of an interface (using an abstract superclass). Additionally, the syntax is much easier to read.