

Problem Set 1

Due in class October 19, 2006

Question 1. (from Russell & Norvig, p. 135, 4.8) [15 points]

The traveling salesperson problem (TSP) can be solved with A^* search. In particular, the state space is the set of partial paths that do not revisit any nodes in the graph, and the “next state” operator takes this partial path and adds a new, previously unvisited, node and associated edge to the path. The cost of this move is equal to the weight of the edge added.

The minimum spanning tree (MST) and the straight-line distance (SLD) are two possible heuristics for this problem. The MST heuristic is the weight of the MST for the (yet) unvisited nodes in the graph, plus the start and end nodes of the partial path. The SLD heuristic is the value of the shortest path in the graph between the start and end nodes of the partial path.

- a. Show that the MST heuristic can be derived from a relaxed version of the TSP.
- b. Show that the MST heuristic dominates the SLD heuristic.

Question 2. [15 points]

Assume we have a game, like Reversi (or Othello), in which moves are taken alternatively by the two players and in the end, the player with the most number of tokens left on the board wins. Let $E1$ be the evaluation function that returns -1 if MIN has more tokens on the board, 0 if the players have an equal number, and 1 otherwise. Let $E2$ be the evaluation function that returns the difference between the number of tokens MAX has on the board and the number MIN has.

part a. Assume the game tree is small enough to be complete traversed. If we use full (unpruned) search to select our moves, will our ability to play (*ie* win) be better with $E1$ or $E2$? How about if we use alpha-beta search instead of full search?

part b. How will the running time of alpha-beta search compare for $E1$ and $E2$?

part c. What if the game tree is too big and we implement a depth bound so as to only search to a fixed number of ply, at which point we apply the evaluation function to the unfinished board. Assume we use alpha-beta search. How will our ability to win using alpha-beta search compare for the two evaluation functions?

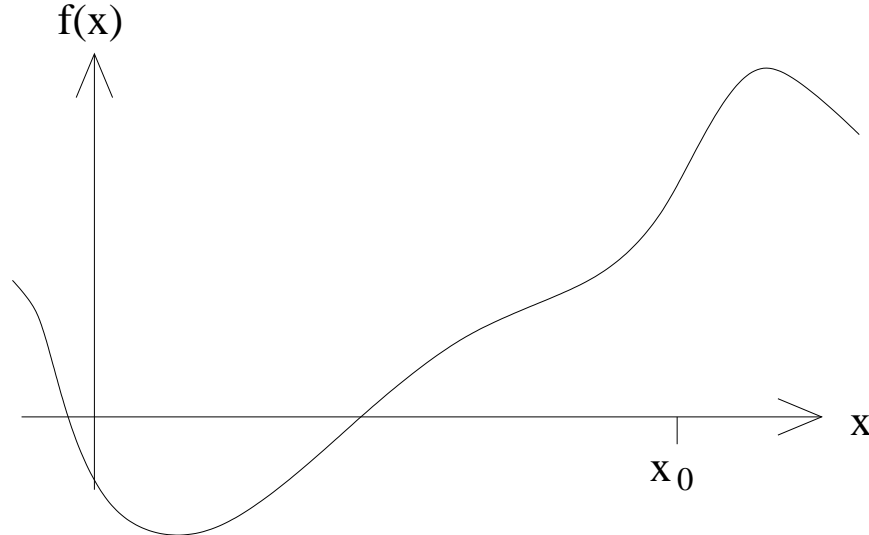
part d. What if the game has an element of chance? Imagine that between the players' turns dice are rolled and some of the tokens are changed as a result. Does your answer to part (c) change? If so, how? If not, why not?

Question 3. [15 points]

Assume we have a scalar function $f(x)$ for which we wish to find a zero-crossing (that is, a point x^* for which $f(x^*) = 0$). Further assume that we have the ability to evaluate $f(x)$ and its derivative $f'(x)$, but that we do not have an analytic form for f . We wish to find this crossing iteratively. We will start with an arbitrary value x_0 . On each iteration we will construct x_i from x_{i-1} , $f(x_{i-1})$, and $f'(x_{i-1})$.

part a. Assume the function is linear. What update would allow us to jump immediately to the answer in a single iteration?

part b. We can still apply this procedure even if f is non-linear. Hopefully it will be (locally) linear enough so that it will converge. Plot the results of the first five iterations on the function below.



part c. Will this procedure always converge? Why or why not?

part d. What if we have a function $g(x)$ which we want to maximize. How can we utilize the zero crossing finding procedure above? What are the updates in terms of $g(x)$ and its derivatives?

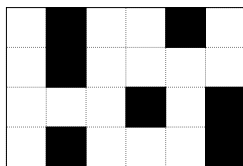
part e. Will this procedure always find a maximum of g ? Provide some explanation.

Question 4. [55 points]

Implement A* search and use it to maneuver a blind robot.

part a. Implement A* search. I highly recommend implementing this in the abstract, using C++ design principles. A* is a naturally encapsulated algorithm which has only a limited and well-defined relationship with the problem to be solved. This should be a stand-alone module that implements A* search for any problem specification.

part b. You have a “blind robot” and you would like to command it so that it reaches a goal location (facing any direction). In particular, your robot lives in a grid. Each cell in the grid is either filled (a wall) or open. Below is one such example



Your robot has a discrete location in this grid (one of the cells) and an orientation (one of the four cardinal directions). Each step it can do one of three things: move forward, turn left, or turn right. If it tries to move into a wall, it stays in the same location. Your goal is to give it a sequence of commands that will result in the robot reaching a specific goal location.

So far, so good. The problem is that your robot has no sensors, so it cannot tell where it is. Furthermore, it doesn't know where it starts. Instead, it knows a set of location-direction pairs where it *might* be. It is guaranteed to be at one of those location-direction pairs, but it has no idea which one.

Use the A* algorithm you wrote in part a to solve this problem. Your program can use the main function provided and the maze class provided as a start. It should read in the maze, goal location, and the list of starting poses and find a sequence of moves that will guarantee the robot will end at the goal location, regardless of which starting location it began at.

Seven example problems (mazes, goal locations, and starting positions) have been provided. Try your algorithm out on each of them. On which inputs can your code find the correct answer in less than 5 minutes?

part c. Add a closed list to your search algorithm. Report the same tests (as part b) with this addition.

part d. Find a consistent and admissible heuristic that results in faster search. Explain your heuristic and report the same tests (as part b) for this new heuristic. You will be judged on the quality of your heuristic. Make sure that there are no simple changes to your heuristic that would result in better performance.