Lab 1 Write Up
Jesse Vogel
Intro to Intelligent Systems
10/2/16

Introduction

The program I created uses two different versions of the A* algorithm depending on the event. The classic event uses a more traditional A* algorithm while the score-o event makes use of a modified iterative deepening approach within the context on the A* algorithm. It also uses a connected graph. Both events produce what I consider to be the shortest path to satisfy the event criteria. The problem environment is defined as follows:

State space: dynamic - (x,y) pixel location, static - map data (terrain, elevation)
Init: starting point (x, y)
Goal: last point in sequence w/ all other points visited in order (classic)
at starting point w/ as many controls visited in allowed time (scoreo)
Actions: move to neighboring pixel (increment/decrement x and/or y by 1)

findPath()

The classic and score-o events share the findPath() function. This function implements the core of the A* algorithm and returns the best time to the given goal point from the given start point, as well as the parents array (used to trace the exact path.) The function follows the pseudocode for A* we discussed in class. IsGoal() simply compares the x and y values of the state to the x and y values of the goal state. The priority queue is sorted by a cost value produced by the cost function, (more on that later.) The successor function produces all valid next-moves.

costFuncton()

The most notable portion of findPath() is the costFunction(). This function uses the current terrain, elevation, and heading to produce a time cost from one state to the next, as well as an estimate from the next state to the goal state. To implement this function, I started by using the heading and terrain data to generate the distance traveled from one point to the next. The next portion of code uses a series of if conditions to map the current and next terrain to their respective speeds. The following chart shows the speeds I choose for each terrain.

Terrain	Speed (m/s)
Open land	3.0
Rough meadow	2.0
Easy movement forest	2.5
Slow run forest	2.0
Walk forest	1.0

Impassible vegetation	0
Lake/Swamp/Marsh	0
Paved Road	4.0
Footpath	3.0
Out of bounds	0

I ranked the speed as follows: paved road > open land, footpath > easy movement forest > rough meadow, slow run forest > walk forest > impassible vegetation, lake/swamp/marsh, out of bounds.

This ranking was based on personal observation and experience moving through these terrains. These numbers are personal to me and can vary based on the person. The next step in this function is to translate the distance into time, (the final unit of cost.) I do this by using Jack Daniels running formula to generate time as a function of distance and grade. The formula states, "for every percent of incline you experience in an uphill, your running time will slow by 12 to 15 seconds per mile." For downhill, "you improve your time by approximately eight seconds per mile for every percent gradient of decline." I decided to use this method for my program because it was more reliable than any estimates I could make via my own observations. It provided a consistent way to modify time based on the grade of the path. Using this equation, I generated constants to multiply with the distance and grade. The result is either added or subtracted to the time depending on the direction – up or down – the person is traveling. That time is used to answer the question, how long will it take to get from my current pixel to my next pixel? To generate the total path cost estimate I make a call to my heuristic function.

heuristic()

My heuristic uses the diagonal distance method. It computes an estimate to the goal by computing the number of steps to take if you can't move diagonal, then subtracting the steps you would save by using the diagonal. I found this approach by reading Amit's Thought's on Pathfinding. There are two constants D and D2 which stand for the minimum cost to move up, down, left, right, and the minimum cost to move diagonal. To calculate D I reasoned the minimum cost my costFunction() could produce would be moving left or right while traveling on a paved road. This would equal 7.55/4.0 = 1.8875. For D2 you would get 12.76/4.0 = 3.19. On flat ground this heuristic is admissible, but when you take elevation into consideration costs could be overestimated due to the time you would save going downhill. I left elevation out of the constants because it seemed unreasonable to assume a large downhill grade on the minimum cost. Adding this could cause D and D2 to be close to 0 and therefore useless altogether.

Score-o Algorithm

I implemented my score-o algorithm by iterating over permutations of the given controls. I added several features to the algorithm to make sure the efficiency was adequate. The first thing I did was create a connected graph with the controls as vertices and the best cost to get from one control to the other as the edges. Pre-computing all the costs saves having to re-compute the same cost multiple times later in the algorithm.

scoreo()

scoreo() function takes this graph and performs a modified iterative deepening A* search on permutations of the list of controls. Instead of deepening, the algorithm starts with all controls and iteratively takes one away. This way, if I find a path that can reach all controls, I can stop searching paths that wouldn't reach as many controls. This saves a large amount of excess computation. For each attempt to reach a certain number of controls, I keep track of the best permutation seen so far. For each permutation I simply move from one control to the next, appending the pre-computed best path to the total time. If at any point the time surpasses the allowed time, I break out of that iteration and try the next permutation. This check saves a lot of time because it ignores any invalid paths. If the permutation was able to reach all the controls in time, I compare the time it took to the best time I have seen so far. If it's better I save it. After checking all permutations of a given length, I check to see if I found at least one path. If so I end the search because all searches to come after will visit less controls. At this point I return the best permutation with it's time and controls it visited.

createGraph()

This function creates an adjacency list representation of a complete graph. Each control is a vertex and the edges are the computed best baths from one control to the next. The graph also stores the parents array as well as the time on each edge. The parents array is used later to recreate the path.

generatePath()

This function takes the pixel access object and writes the path onto it. It uses the parents array to trace backwards starting from the goal control. A state whose parent is None represents the starting control.

createDirection()

This function is responsible for creating the written directions that describe the best path. The main idea here is to break the path into two classes – traveling on a path/road, or traveling through any other terrain. The algorithm starts by declaring that it has reached a control point. Next it decides what class the last point is in as well as the parent of that point. While the two points share the same class, it simply uses the parents array to move backwards one step to the next point. As soon as the current point is in a different class, (i.e we move

off the trail into the woods,) the algorithm ends the path and computes the straight line distance traveled. It also computes the clockwise angle that the path moved – where 0 is North, 90 is East, etc... This path information is then added to a buffer, which will be written to the file once all the text is computed. This pattern repeats until we reach the next control. The function that calls createDirections() iterates over the best path and calls the function for each path between two controls.

Extra Credit!

I love getting outdoors so this extra credit assignment sounded just up my alley. The weather on Sunday really gave me a run for my money though. It was pouring rain the whole time! My solution turned out to be really efficient. It started by taking the road out from the parking lot, across the main road, then onto a trail. The first point it brought me to was control point #9. Here is a photo of me at control #9.

From there it had me cut through the forest to the next control #12. This cut through turned out to be a big time saver over staying on the trails, especially when the forest looks like this!

I was able to move effortlessly through this. From control #12 I continued to follow trails all the way to control #14. This was a good decision because the elevation off of the trail was really steep. Here I am at control #14.

From control #14 I backtracked a bit then cut through some woods, back across the road, and through the field to the last control, #15. Overall the course took me 34 minutes, where as my algorithm said it would take 13 minutes. This is mostly due to the fact that I was lazy and didn't run on the roads and trails

like my algorithm assumed I would do. (I ran when it really started raining hard though!) I also took one wrong turn. If the weather wasn't so bad and I was to run I think my time would be much closer to the estimated time. Here is the full path my algorithm generated