CSE 101 Winter 2016
Programming Assignment (PA) Four  (Heuristics!)

Due: Friday March 4, 11:59 PM

Link to starter code: https://github.com/UCSD-CSE101-W16/PA4

**Question 1: Heuristic for the Euclidean Traveling Salesperson Problem [50 Points]**

Input: Vector of distinct cartesian points \((x, y)\) where \(x, y\) are non-negative integers
(Note: indices of the points matter! They will represent the visit order)

Output: Vector of indices representing an undirected Hamiltonian cycle that approximates the TSP solution to within twice its optimal cost

The Traveling Salesperson Problem (TSP) is an NP-hard problem that asks to find the lowest possible cost with which we can visit each vertex in a weighted graph exactly once, with the exception of the start vertex which is visited twice -- once for the start, and once for the end (this is a Hamiltonian cycle, and we will refer to it as a tour). Naively, we can find a solution for the TSP on a graph with vertices \(V\) by generating all tours in \(O(|V|!)\) time and choosing the tour with the minimum cost as our solution. Finding the optimal or lowest-weight TSP tour is NP-hard; there is no known polynomial-time algorithm for finding such a tour.

For this question, you will design a polynomial-time algorithm that finds a tour which approximates an answer to the TSP that is at most twice the total cost of the optimal TSP tour. By approximation, we mean that (1) you will find a valid tour for the graph, but the tour you find is not necessarily the minimum weight tour; however, (2) the tour you find will have cost within a factor of two of optimal. Henceforth, we will refer to the optimal TSP solution as \(OPT\) and to our approximation as \(2-OPT\), with the required relation that:

\[
\text{cost}(OPT) \leq \text{cost}(2-OPT) \leq 2 \times \text{cost}(OPT)
\]

Notice that a tour necessarily has to visit each vertex once, and to do so, it must select exactly \(|V|\) edges to complete the Hamiltonian cycle in order to visit all vertices. Think back to PA 2 where we found the total cost of the resulting minimum spanning tree (MST) from Prim’s algorithm as the MST will be fundamental in our heuristic algorithm. We can guarantee the above relation if we construct \(2-OPT\) such that it satisfies:

\[
\text{cost}(MST) < \text{cost}(OPT) \leq \text{cost}(2-OPT) \leq 2 \times \text{cost}(MST) \leq 2 \times \text{cost}(OPT)
\]

Review the proof of correctness for 2-OPT in slides 20-24 of Lecture 9.

Taking all of the above into account, we will now move on to the programming question with more concrete terms, and at this point you should be thinking about how we can create a tour \(2-OPT\) such that its cost is less than or equal to twice the cost of the MST. Prim’s MST algorithm is essential to completing this problem. Come in to office hours if you are still having trouble with Prim’s MST from PA 2. The input for this question will come in as a list of distinct Cartesian points where each point is denoted by two non-negative integers \(x\) and \(y\). You can think of the input as points on the two dimensional Cartesian plane. For a given vector of points you will output the order in which to connect them to form \(2-OPT\) such that the above
requirements for 2-OPT are met. All vertices are directly reachable from each other, and you can visualize this as being able to draw a line to connect any two points (i.e., connect the dots from grade school!). The distance between any two points \((x_1, y_1)\) and \((x_2, y_2)\) is calculated using the Euclidean distance formula (which will yield strictly positive distances since we have distinct points):

\[
distance = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}
\]

This distance is our definition of cost, so the TSP tour \(OPT\) has minimal total distance travelled to visit all points exactly once and arrive back at where it starts. The distance method \(\text{dist}\) has already been written for you and it is recommended that you use the provided implementation. Notice that we have not yet defined a mechanism for achieving \(\text{cost}(2\text{-OPT}) \leq 2 * \text{cost}(\text{MST})\). This part is up to you to figure out using the methods described in the Lecture 9 slides. Any 2-OPT tour is valid as long as it meets the required bounds. [Hint: you should be able to see from the Lecture 9 slides that a DFS tour of the MST is useful.]

You will output the 2-OPT visit order by returning a vector of indices corresponding to the inputted points and in the same visit order that you found for 2-OPT (i.e. starting at the first element, it goes from the first element to the second element, then second to third, then third to fourth, etc.). The size of your outputted vector must be exactly the same size as the number of inputted points, since we have exactly the same number of edges as number of inputted points, and the tester will add the last edge connecting your last point to your first point for you. Be very careful as you must index the points using the same indices as in the inputted points vector. Only the original points vector inputted to the method stores all of the \((x, y)\) information, and your implementation should reference the inputted points vector when it needs to look up \((x, y)\) values.

The provided tester will handle all input and output for you as you’ve been accustomed to, and it also verifies correctness for you when calculating the distance by requiring that all inputted points are visited exactly once. If you implement the \(\text{MST}\) method, then the tester can also compare the cost of the MST to your result for 2-OPT (not required since only TwoOPT will be called when grading, but it is highly recommended so that you can break out your code into smaller parts, and have more peace of mind while testing -- you will still have to implement some MST method in TwoOPT if you choose to not implement \(\text{MST}\)). For the \(\text{MST}\) method, you will root the MST at the point indexed by 0, and output an array indicating the previous index for each index. (e.g. for 10 points indexed 0...9, then the MST[0] = -1 since index 0 is the root and has no previous. MST[3] stores the previous index for the point indexed by 3, etc. for all indices. Notice that we store previous and not next, which we would store instead with an adjacency list, because we have a tree and each vertex may have many next fields, but can only have 1 previous field. The only MST[] element with a -1 value indicating no previous should be MST[0] since there is only 1 root.)

Also provided is a point generator python script that will create tuples \((x, y)\) in an \(X \times X\) square. To use the script, you can modify the parameters for \(N\) and \(X\) where \(N\) is the number of points, and \(X\) is the size of the region such that for all points \((x, y)\), \(0 \leq x \leq X\) and \(0 \leq y \leq X\). You will need to edit \(N\) and \(X\) as you are accustomed to from previous PAs.

Your implementation will be tested with up to \(N = 5000\), and should complete within 60 seconds for this upper-bound case (when testing, your executable will be allowed to run for at most 120 seconds). We will test with up to \(X = 1000\) as well, although this latter bound is not as significant. Again, if you implement \(\text{MST}\) correctly you will be able to verify correctness for any test case by using the provided tester executable.
To use the point generator script:
   python point_gen.py  this will create an output file named POINTS_N_X in the current directory

To Make:
   make TestTSP

To Run:
   ./TestTSP input_file

To Run with MST verification:
   ./TestTSP input_file D
   The tester specifically looks for a second argument with starting character 'D' for Debug

Sample Input (generated input using POINTS_20_50):
   43 48
   35 8
   22 42
   50 44
   33 26
   28 35
   34 42
   2 23
   7 28
   11 26
   22 50
   40 29
   39 37
   1 42
   5 1
   11 46
   10 22
   7 29
   48 50
   32 28

Sample Output:
   ./TestTSP.out POINTS_20_50
   The cost of the 2-OPT TSP tour found is: 278.639

Sample Output with MST verification:
   ./TestTSP.out POINTS_20_50 D
   The cost of the 2-OPT TSP tour found is: 278.639
   The cost of the MST is: 165.95
   The ratio found between 2-OPT and MST is: 1.67905
   A correct implementation always has a 2-OPT/MST ratio between 1.0 and 2.0 for all valid test cases

Note: Your algorithm must be deterministic (same input file always produces same output results)
**Question 2: A* Heuristics [50 Points]**

**Input:**
- TwoDArray grid of coordinates of size $n \times m$ specified in the input file
- Coordinate $s$ representing the (row, column) coordinates of the source vertex
- Coordinate $d$ representing the (row, column) coordinates of the destination vertex
- Float $heuristic\_weight$ representing the degree to which the heuristic $h(n)$ affects A*

**Output:**
Number of vertices explored in grid and the length of the shortest path found by A*

In PA 2, you were asked to implement Dijkstra’s algorithm in order to find the shortest path from one point to another. Dijkstra’s is the algorithm we learned for shortest-path finding in general (non-negative edge-weighted) graphs. Nonetheless, when applied to video games, route planning, and other real-life applications, external information such as the actual distance between two points may allow computer scientists to develop heuristics based on such information to obtain much faster, yet still accurate, approximations of shortest paths in graph representations of a given environment.

Following the development of Dijkstra’s algorithm in 1959, researchers at Stanford Research International developed the **A* algorithm** in 1968 to utilize lower-bound estimates of each vertex’s remaining shortest-path distance to the path-finding destination, in order to prune away many of the unpromising choices Dijkstra’s considers when pathfinding. Specifically, while Dijkstra’s algorithm labels vertices based solely on the cost of the path from the start node to the current vertex considered ($= g(n)$), A* picks vertices to expand based on smallest sum of (1) the cost of the path from the start vertex to the current vertex considered ($= g(n)$), and (2) a lower-bound estimate of the cost of the path from the current vertex to the destination ($= h(n)$).

\[
\text{Dijkstra’s: } f(n) = g(n) \\
\text{A*: } f(n) = g(n) + h(n)
\]

- $f(n)$: function used to determine the relative cost of the total path from the source to $n$
- $g(n)$: the cost of the path from the source to $n$ determined thus far
- $h(n)$: a heuristic **lower-bound** estimate of the remaining cost of the path from $n$ to destination

As can be seen, because A* factors in a score derived from a heuristic function that impacts the path being considered by the algorithm, A* will naturally prefer picking paths that the heuristic deems favorable for approaching the goal, before it considers unfavorable paths. Such a behavior often significantly reduces the number of vertices visited in A*’s search for the shortest path.

**Important note #1:** If $h(n)$ is not a lower bound on the remaining shortest-path cost from vertex $n$ to the destination, then the first start-destination path found by A* is not guaranteed to be a shortest path. We will be exploring the behavior of A* with different weightings of $h(n)$ that may result in this case.

**Interesting note #1:** If $h(n)$ is 0 for all $n \in V$, the particular instance of the A* algorithm would be equivalent to Dijkstra’s algorithm.

In this assignment, we are developing an efficient pathfinding algorithm to help a character in a video game move from one position to another. While Dijkstra’s will consider all paths leading from the source position - considering every vertex at least once before terminating, you will be using A* with a **Manhattan distance**
based heuristic to narrow down the search space. Manhattan distance, then, is simply the distance between two points in a grid based on a strictly horizontal and/or vertical path (i.e. no diagonals). To analyze the behavior of A* heuristics, you are given a float heuristic_weight by which you will scale (that is, multiply by) h(n) to modulate the significance of the value returned by the heuristic function. Note that your algorithm cannot utilize any path that enters a grid location that is designated as an obstacle. During the execution of your algorithm, be sure to keep a counter to record the number of vertices visited (i.e. popped from the priority queue) during A*'s execution.

After you complete your A* algorithm, you will now analyze the effects of obstacles on the relative time complexity of A*. We have included a Python script with input variables n, m, h, and o defining the n x m size of the grid, the weight of the heuristic function h, and the proportion of the grid that will be covered with obstacles o ∈ [0.0, 1.0]. On a 5000 x 5000 grid, you will fill out a table with values o ∈ {0.0, 0.1, 0.2, 0.3, 0.4} as the columns and h ∈ {0.0, 1.0, 2.0} as the rows, with each cell containing the number of vertices explored and the length of the "best" path found by A*. To take advantage of the size of the grid, keep the source coordinates at (50, 50) and the destination coordinate at (4950, 4950). Note that if your A* algorithm was unable to reach the destination vertex (by returning -1 for the length of the path, as noted in the example output below) after generating two random grids with the same test parameters, you should note this in your table. You will then write a qualitative analysis in the form of a write-up PDF analyzing the phenomena you observe.

Your analysis must, at minimum, touch on the following points:
- What are the tradeoffs in increasing or decreasing the weight of your heuristic function?
- Explain why changing the magnitude by which the heuristic function affected A*'s decision making process affected the number of vertices explored - and more importantly, the lengths of the best-paths found.
- What are the effects of obstacles upon the efficacy of the A* algorithm? Describe a case where your heuristic for the A* algorithm could negatively affect your search speed with respect to Dijkstra's in this problem’s scenario.

To use the grid generator script:
```
python grid_gen.py // this will create an output file named GRID_N_M_O_H
```

To Make:
```
make TestAStar
```

To Run:
```
./TestAStar input_file
```

To Run with Grid Printing:
```
./TestAStar -P input_file
(for sanity checks on your output)
```
Example Input #1:

```
1 0 1 0 // size of n x m grid
0 0 // start coordinates
9 9 // end coordinates
2.0 // heuristic weight (i.e., multiplier of h(n) in calculating f(n))
9 0 // obstacle coordinates
8 1
7 2
6 3
5 4
3 6
2 7
1 8
0 9
```

Example output #1 [found destination, ‘-P’ option to print]: (output matrix slightly reformatted for readability)

```
$ ./TestAStar.out -P <testfile>
Number of vertices explored: 28 with path length 18
36 35 inf inf inf inf inf inf obs
35 34 inf inf inf inf inf obs inf
34 33 inf inf inf inf inf obs inf
33 32 33 32 31 30 obs inf inf
32 31 30 29 28 27 26 inf inf inf
31 30 31 28 obs 26 25 inf inf inf
30 29 28 obs 28 25 24 inf inf inf
29 28 obs inf 27 24 23 inf inf inf
28 obs inf inf 26 23 22 23 22 inf
obs inf inf inf 25 22 21 20 19 18
```

Example Input #2:

```
10 10 // size of n x m grid
1 1 // start coordinates
8 9 // end coordinates
2.0 // heuristic weight (i.e., multiplier of h(n) in calculating f(n))
0 4 // obstacle coordinates
1 4
2 4
3 4
4 4
5 4
6 4
7 4
8 4
9 4
```

Example output #2 [failed to find destination]:

```
$ ./TestAStar.out <testfile>
Number of vertices explored: -1 with path length -1
```