CMPT125, Fall 2020

Homework Assignment 5

Due date: Wednesday, December 9, 2020, 23:59

For this assignment you will create a project in C++ that implements a solver for the Traveling Salesperson Problem (TSP).
You will need to design your own classes, decide how to partition them into files, decide what goes into .hpp and what goes into .cpp.

You will also need to write your own main() in a separate file.
The main() will test your TSP solver on examples that you will create yourself.

You may use any data structures we learned in the course.

Submit all your files in assignment5.zip to CourSys.

Good luck!

The Traveling Salesperson Problem

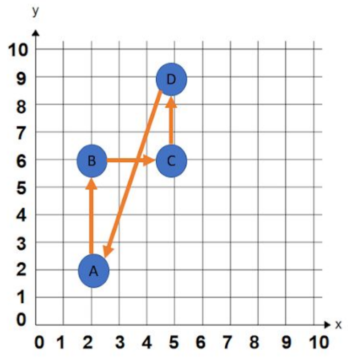
The input to the problem is a collection of *n* points in the plane.,

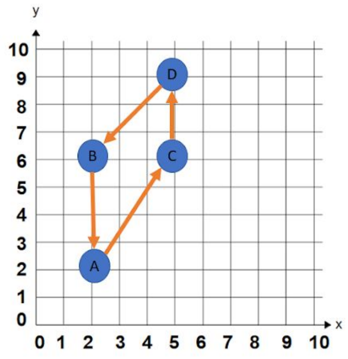
The goal of the traveling salesperson problem is to find the shortest path that visits every point exactly once and returns to the starting point. That is, we are looking for a cycle in the graph that visits each vertex exactly once, such that the total length is as small as possible.

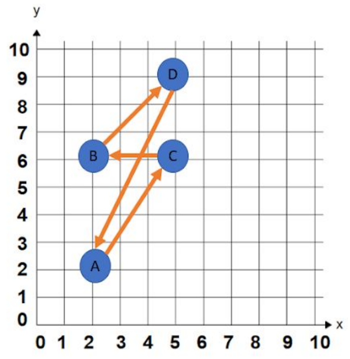
For example, suppose the input is the 4 points

A = (2,2), B = (2,6), C = (5,6), D = (5,9)

Then we may consider each of the following cycles through the 4 points.







- The length of the cycle ABCD (on the left) is

  dist(A,B)+dist(B,C)+dist(C,D)+dist(D,A) = $4+3+3+\sqrt{3^2+7^2} \approx 17.616$

- The length of the cycle ACDB (in the middle) is

  dist(A,C)+dist(C,D)+dist(D,B)+dist(B,A) = $\sqrt{3^2+4^2}+3+\sqrt{3^2+3^2}+4 \approx 16.243$

- The length of the cycle ACBD (on the right) is

  dist(A,C)+dist(C,B)+dist(B,D)+dist(D,A) = $\sqrt{3^2+4^2}+3+\sqrt{3^2+3^2}+\sqrt{3^2+7^2} \approx 19.858$

That is, among these three cycles, the better one is ACDB, whose length is 16.243...

In fact, this is the optimal solution for this input.

You will need to design your own classes in C++ to implement a solver for this problem.

Requirements:

The requirements for this project are the following [The exact implementation details are described below]

Storing the points:
You will need to write a class that stores a collection of points. You may use any data structure you want to do it.

Print the list of points:
Your solution should have a function that prints the list of all points. The format is up to you. For example, you can use the format
A = (2,2)
B = (2,6)
C = (5,6)
D = (5,9)

Drawing the points:
Your solution should have a function that draws the points on the screen.
No need to use anything fancy for drawing. A textual representation of the board suffices.
You may assume here that all coordinates are between 0 and 20 and all names are a single letter.
For example, you can print something like this:

```
- - - - - - - - - - -
- - - - - - - - D - -
- - - - - - - - - - -
- - - - - - - - - - -
- - B - - - - C - -
- - - - - - - - - - -
- - - - - - - - - - -
- - - - - - - - - - -
- - A - - - - - - -
- - - - - - - - - - -
- - - - - - - - - - -
```

TSPSolver:
This is the main part of this project. You will need to implement a heuristic algorithm that finds a solution to the TSP problem. For the algorithm see the part **TSP solver** below.

main():
You will need to write the main() function that will run several tests to check your solution.
Write a test for every part of your code. Write as many tests as you think are necessary.
At the very least, your tests should (1) provide inputs to the problem (2) print the list using the printList function (3) draw the points (4) run the solver and output the obtained solution: you need to print both the order of the vertices in the cycle and the total distance of the cycle.

TSP solver

The Traveling Salesperson Problem is NP-complete. Without saying exactly what this means, this implies that we don't know a polynomial time algorithm that funds an optimal solution for all inputs.

Instead of trying to find an optimal solution, your TSPSolver will need to implement the following heuristic algorithm, which we will call "nearest next point heuristic".

The algorithm starts with a path consisting of one point, and builds a path P by adding to it one point at a time.
In each iteration the algorithm is looking for a point that has not been added to P yet, which is the closest to the last point in P (breaking ties arbitrarily). This point is added to P as the last point.

Example:
Consider the example above with 4 points, and suppose we start from B.

1. In the first iteration, the closest point to B is C.
   Hence, we add C after B to the path.
2. Now, the path is B->C, and we are looking for the next point.
   C is the last point on the path, and D is the closest to C among those not added yet.
   Therefore, we add D to the path after C.
3. Now, the path is B->C->D, and we are looking for the next point.
   D is the last point on the path, and A is the closest point to D among those not added yet.
   We add A to the path after D.
   The new path becomes B->C->D->A
4. We added all points to the path, and this gives us the solution B->C->D->A.
5. The length of this cycle (including the edge A->B) is $4 + 3 + 3 + \sqrt{3^2 + 7^2} \approx 17.616$.

Other heuristics [NOT FOR SUBMISSION]:
You may try implementing several other heuristics if you want.
For example, consider the following two greedy ideas:

- *Nearest point heuristic:* The input is a list L[0...n-1] of n points.
  Start with the path consisting of L[0]. Then, in each iteration take the next point L[i], and add it to the current path after the point to which it is closest. Break ties arbitrarily.

- *Smallest increase heuristic:* The input is a list L[0...n-1] of n points.
  Start with the cycle consisting of a single point L[0].
  Then, in each iteration take the next point L[i], and add it to the current path so that the increase in the length is the least possible. That is, add L[i] to the cycle between P[j] and P[j+1] such that dist(P[j],L[i])+dist(L[i],P[j+1]) is minimized.

<u>Implementing classes:</u>

There are no specific requirements about which classes you need to implement.
Below are some suggestions you may use. They are also provided in the hw5.zip.
These are examples only. You may choose a completely different implementation if you want.

```
// the class represents a point in 2D and its name
class Point {
Private:
	string name;
	int x;
	int y;
	// computes the distance between this and the  other point
	float getDistance(const Point &other)
}


// the class stores an ordered list of points
// used to store the input to the problem
// may be also used to store a partial solution to the TSP problem
class ListOfPoints {
	// adds a new point to the beginning of the list
	void addToFront(Point &newPt)
	// adds a new point to the end of the list
	void addToBack(Point &newPt)
	// prints the list of points
	void printList()
	// draws the points
	void draw()
}


// the class stores a cycle that traverses all points in some order
// it is used to store the solution to the problem
// may be a subclass of ListOfPoints
class TSPCycle : public ListOfPoints {
	// returns the total length of the cycle
	float getLength()
	// may have additional methods such as getLast() addNext() and more
}


// the class implement the TSP solver
class TSPSolver {
public:
	// an example of a constructor
	TSPSolver(ListOfPoints &list);
	// solves the problem, and stores the solution
	void solve();
	// returns the solution obtained by the solve() method.
	TSPCycle getSolution();
}
```