

CMPT125, Fall 2020

Homework Assignment 5

Due date: Monday, December 6, 2021

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.

You will need to decide how to partition the classes into files.

You will need to 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 abstract data type we learned in the course.

The file hw5-cmpt125-fall21.zip contains several .cpp and .hpp files as well as Makefile. You may modify any of them.

Make sure that Makefile creates the executable called ***tsp solver***.

Submit all your files in assignment5.zip to Canvas.

Good luck!

The Traveling Salesperson Problem

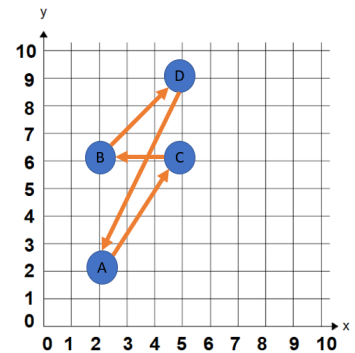
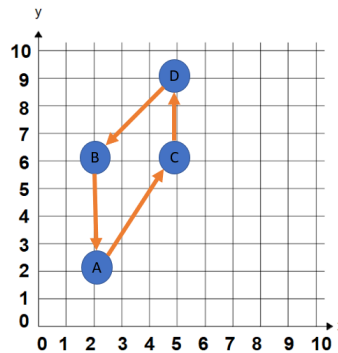
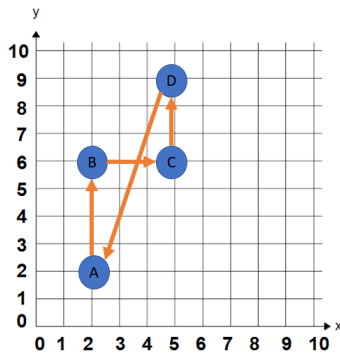
The input to the problem is a collection of n points in the plane. The points have int values.

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

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

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

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

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

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

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 (array, linked list or vector in C++)

Print the list of points:

Your solution should have a method 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 method 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 names of points are single letters. 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 method (3) draw the points (4) run the solver and output the obtained solution: you need to print to screen both the order of the vertices in the cycle and the total length 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 an efficient algorithm that finds an optimal solution for every input.

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

Nearest existing point heuristic:

- The input is a list $L[0 \dots n-1]$ of n points.
- The algorithm starts with the path consisting of $L[0]$.
- 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.
- Close the cycle by connecting the last point in the path to the first point.

Example:

Consider the example above with 4 points, and suppose that the list is $[C, B, D, A]$.

1. We start with the path consisting of only one point, C.
2. In the first iteration, we choose where to add B. Since C is the only point in the path so far, we add B after C. Now the path is $C \rightarrow B$.
3. Next we want to add D to the cycle.
 $\text{dist}(B, D) = 4.24264$
 $\text{dist}(C, D) = 3$
Therefore, we add D after C, and get the path $C \rightarrow D \rightarrow B$.
4. Next we want to add A to the cycle.
 $\text{dist}(B, A) = 4$
 $\text{dist}(C, A) = 5$
 $\text{dist}(D, A) = 7.61577\dots$
Therefore, we add A after B, and get the path $C \rightarrow D \rightarrow B \rightarrow A$.
5. Finally, we close the cycle by connecting A to C.
6. The length of this cycle (including the edge $A \rightarrow C$) is $3 + 4.24264 + 4 + 5 = 16.24264$.

Implementing classes:

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-cmpt125-fall21.zip

These are examples only. You may choose a completely different implementation if you want.

```
class Point {
private:
    string name;
    int x;
    int y;
public:
    Point(int x, int y, string nm);
    // 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 {
public:
    // adds a newPt after a point with the given name
    void addAfter(Point &newPt, string name)
    // 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()
}

// stores a cycle that traverses all points in some order
class TSPCycle : public ListOfPoints {
public:
    // returns the total length of the cycle
    float getLength()
}

// implementation of 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();

private:
    TSPCycle sol;
}
```

Other heuristics [NOT FOR SUBMISSION]:

You may try implementing several other heuristics if you want.

For example, consider the following two greedy ideas:

- Smallest increase heuristic: The input is a list $L[0 \dots n-1]$ of n points. The algorithm starts with the cycle consisting of a single point $L[0]$. In the i 'th iteration the algorithm takes the point $L[i+1]$, and adds it to the current cycle in the location that minimizes the increase in the length. That is, add $L[i+1]$ to the cycle between $C[j]$ and $C[j+1]$ such that $\text{dist}(C[j], L[i]) + \text{dist}(L[i+1], C[j+1])$ is minimized.
- Nearest last point heuristic: The algorithm starts with a path consisting of one point. In each iteration the algorithm builds a path P by adding to it one point at a time. Specifically, the algorithm finds 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. In the end the last point is connected to the first point to close the cycle.