The University of Newcastle, Australia
School of Information and Physical Sciences
COMP2230/COMP6230 Algorithms
Assignment 2 Marks 100 Weight 20%
Individual Submission via Canvas

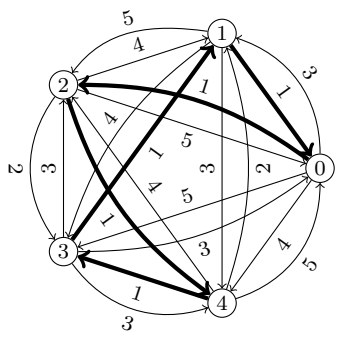
1 Assignment Overview

This assignment is to be done by each student *individually* using the Java programming language. You are given 5 Travelling Salesman Problem (TSP) instances. You have to write two separate programs to solve the TSP instances. Each program in each of its run will read a given TSP instance from a given file, solve the TSP instance, and give required output on the screen. You have to run the programs on the given TSP instances and compare the performances of the programs.

- `DynaTSP`: a program that implements a dynamic programming method.
- `ClimbTSP`: a program that implements a hill climbing metaheuristic method.

2 Problem Definition

Given a fully connected graph with n nodes $N \equiv \{0, \ldots, n-1\}$ and each link (i, j) from node i to j with length l_{ij}, the *travelling salesman problem* is to find a *cyclic tour* that starts at node 0, visits each other node exactly once, finishes at node 0, and has the *minimum total length*. An example travelling salesman problem and a solution is below in the figure and the table.



l_{ij}	0	1	2	3	4
0	0	3	1	5	4
1	1	0	5	4	3
2	5	4	0	2	1
3	3	1	3	0	3
4	5	2	4	1	0

Shortest Tour: $(0, 2, 4, 3, 1, 0)$
Min Tour Length: 5

3 Dynamic Programming

The recurrence relation, an example solution, and some notes on program implementation for the dynamic programming algorithm for TSP are described.

3.1 Recurrence Relation

- A cyclic tour $(0, \ldots, j, 0)$ has two parts: a path $(0, \ldots, j)$ from node 0 to node j and a link $(j, 0)$ to return back from node j to node 0 to obtain a cycle. Note j could be any node, except node 0. For example, the cyclic tour $(0, 2, 4, 3, 1, 0)$ shown above has a path $(0, 2, 4, 3, 1)$ from node 0 to node 1 and then a link $(1, 0)$ to return to node 0 from node 1.

- Assume S_j is the minimum toal length over all possible paths $(0, \ldots, j)$ i.e. paths from node 0 to node j visting all other $n-2$ nodes exactly once. This means we have a permutation of nodes $N \setminus \{0, j\}$ between nodes 0 and j. So the solution to a TSP problem is as below since we have to consider each other node as the last node and take the minimum.

$$S = \min_{j \in N \setminus \{0\}} S_j + l_{j0}$$

- Assume $I \subseteq N \setminus \{0, j\}$ with $j \neq 0$. So, $0 \leq |I| \leq n-2$. Further, assume a permutation of all nodes in I is in between nodes 0 and j on a path $(0, \ldots, j)$. Let $f(I, j)$ denote the minimum total length over all paths $(0, \ldots, j)$ that have a permutation of all nodes in I in between nodes 0 and j. Consider $|I|$ as the stage and j as the state for dynamic programming. Notice that $S_j = f(I, j)$ only when $I = N \setminus \{0, j\}$ i.e. I contains all the other $n-2$ nodes. Moreover, $f(I, j)$ can have an I that has fewer than $n-2$ nodes. Below we define $f(I, j)$.

- Boundary cases: $I = \Phi$ i.e. there is no node between nodes 0 and j where $j \neq 0$.

$$f(\Phi, j) = l_{0j} \text{ for each } j \in N \setminus \{0\}$$

- Regular cases: $I \neq \Phi$ and a path $(0, \ldots, i, j)$ has $i \in I$ as the last node before j where $j \neq 0$. Note that i could be any node in I and we have to take minimum over all candidate nodes for i. Moreover, when a node i is considered as the last node before j, the other nodes in $I \setminus \{i\}$ will be considered for permutations and will thus be between nodes 0 and i.

$$f(I, j) = \min_{i \in I} f(I \setminus \{i\}, i) + l_{ij}$$

3.2 Example Solution

We can implement $f(I, j)$ function recursively and the recursion starts from $S_j = f(I, j)$ with $I = N \setminus \{0, j\}$ and goes downward. However, the actual computation will take place in a bottom-up fashion, starting from $|I| = 0$ to $n-2$. Alternatively, you can write an iterative bottom-up algorithm to implement $f(I, j)$ function. Below we show the computations that take place bottom up. With the $f(I, j)$ values, we also show the argmin i values within brackets.

Example Graph

l_{ij}	0	1	2	3	4
0	0	3	1	5	4
1	1	0	5	4	3
2	5	4	0	2	1
3	3	1	3	0	3
4	5	2	4	1	0

$$f(\Phi, j) = l_{0j}$$

$$f(\Phi, 1) = l_{01} = 3(\cdot)$$
$$f(\Phi, 2) = l_{02} = 1(\cdot)$$
$$f(\Phi, 3) = l_{03} = 5(\cdot)$$
$$f(\Phi, 4) = l_{04} = 4(\cdot)$$

$$f(\{i\}, j) = f(\Phi, i) + l_{ij}$$

$$f(\{1\}, 2) = f(\Phi, 1) + l_{12} = 3 + 5 = 8(1)$$
$$f(\{1\}, 3) = f(\Phi, 1) + l_{13} = 3 + 4 = 7(1)$$
$$f(\{1\}, 4) = f(\Phi, 1) + l_{14} = 3 + 3 = 6(1)$$
$$f(\{2\}, 1) = f(\Phi, 2) + l_{21} = 1 + 4 = 5(2)$$
$$f(\{2\}, 3) = f(\Phi, 2) + l_{23} = 1 + 2 = 3(2)$$
$$f(\{2\}, 4) = f(\Phi, 2) + l_{24} = 1 + 1 = 2(2)$$
$$f(\{3\}, 1) = f(\Phi, 3) + l_{31} = 5 + 1 = 6(3)$$
$$f(\{3\}, 2) = f(\Phi, 3) + l_{32} = 5 + 3 = 8(3)$$
$$f(\{3\}, 4) = f(\Phi, 3) + l_{34} = 5 + 3 = 8(3)$$
$$f(\{4\}, 1) = f(\Phi, 4) + l_{41} = 4 + 2 = 6(4)$$
$$f(\{4\}, 2) = f(\Phi, 4) + l_{42} = 4 + 4 = 8(4)$$
$$f(\{4\}, 3) = f(\Phi, 4) + l_{43} = 4 + 1 = 5(4)$$

$$f(\{h, i\}, j) = \min[f(\{h\}, i) + l_{ij}, f(\{i\}, h) + l_{hj}]$$

$$f(\{1, 2\}, 3) = \min[f(\{1\}, 2) + l_{23}, f(\{2\}, 1) + l_{13}] = \min[8 + 2, 5 + 4] = 9(1)$$
$$f(\{1, 2\}, 4) = \min[f(\{1\}, 2) + l_{24}, f(\{2\}, 1) + l_{14}] = \min[8 + 1, 5 + 3] = 8(1)$$
$$f(\{1, 3\}, 2) = \min[f(\{1\}, 3) + l_{32}, f(\{3\}, 1) + l_{12}] = \min[7 + 3, 6 + 5] = 10(3)$$
$$f(\{1, 3\}, 4) = \min[f(\{1\}, 3) + l_{34}, f(\{3\}, 1) + l_{14}] = \min[7 + 3, 6 + 3] = 9(1)$$
$$f(\{1, 4\}, 2) = \min[f(\{1\}, 4) + l_{42}, f(\{4\}, 1) + l_{12}] = \min[6 + 4, 6 + 5] = 10(4)$$
$$f(\{1, 4\}, 3) = \min[f(\{1\}, 4) + l_{43}, f(\{4\}, 1) + l_{13}] = \min[6 + 1, 6 + 4] = 7(4)$$
$$f(\{2, 3\}, 1) = \min[f(\{2\}, 3) + l_{31}, f(\{3\}, 2) + l_{21}] = \min[3 + 1, 8 + 4] = 4(3)$$
$$f(\{2, 3\}, 4) = \min[f(\{2\}, 3) + l_{34}, f(\{3\}, 2) + l_{24}] = \min[3 + 3, 8 + 1] = 6(3)$$
$$f(\{2, 4\}, 1) = \min[f(\{2\}, 4) + l_{41}, f(\{4\}, 2) + l_{21}] = \min[2 + 2, 8 + 4] = 4(4)$$
$$f(\{2, 4\}, 3) = \min[f(\{2\}, 4) + l_{43}, f(\{4\}, 2) + l_{23}] = \min[2 + 1, 8 + 2] = 3(4)$$
$$f(\{3, 4\}, 1) = \min[f(\{3\}, 4) + l_{41}, f(\{4\}, 3) + l_{31}] = \min[8 + 2, 5 + 1] = 6(3)$$
$$f(\{3, 4\}, 2) = \min[f(\{3\}, 4) + l_{42}, f(\{4\}, 3) + l_{32}] = \min[8 + 4, 5 + 3] = 8(3)$$

$$f(\{g,h,i\},j) = \min[f(\{g,h\},i) + l_{ij}, f(\{g,i\},h) + l_{hj}, f(\{h,i\},g) + l_{gj}]$$

$f(\{1,2,3\},4) = \min[f(\{1,2\},3)+l_{34}, f(\{1,3\},2)+l_{24}, f(\{2,3\},1)+l_{14}] = \min[9+3, 10+1, 4+3] = 7(1)$
$f(\{1,2,4\},3) = \min[f(\{1,2\},4)+l_{43}, f(\{1,4\},2)+l_{23}, f(\{2,3\},1)+l_{13}] = \min[8+1, 10+2, 4+4] = 8(1)$
$f(\{1,3,4\},2) = \min[f(\{1,3\},4)+l_{42}, f(\{1,4\},3)+l_{32}, f(\{3,4\},1)+l_{12}] = \min[9+4, 7+3, 6+5] = 10(3)$
$f(\{2,3,4\},1) = \min[f(\{2,3\},4)+l_{41}, f(\{2,4\},3)+l_{31}, f(\{3,4\},2)+l_{21}] = \min[6+2, 3+1, 8+4] = 4(3)$

$S_1 = f(\{2,3,4\},1) = 4(3)$ $\quad$ $S_3 = f(\{1,2,4\},3) = 8(1)$
$S_2 = f(\{1,3,4\},2) = 10(3)$ $\quad$ $S_4 = f(\{1,2,3\},4) = 7(1)$

With all the shortest paths found, we now consider the cyclic tours.

$$S = \min[S_1 + l_{10}, S_2 + l_{20}, S_3 + l_{30}, S_4 + l_{40}] = \min[4+1, 10+5, 8+3, 7+5] = 5(1)$$

Taking argmin for the cycle 5(1) and then the argmins of $f(\{2,3,4\},1) = 4(3)$, $f(\{2,4\},3) = 3(4)$, $f(\{2\},4) = 2(2)$, and $f(\Phi,2) = 1(\cdot)$, the solution is $[0,2,4,3,1,0]$ with length 5.

3.3 Program Implementation

There is no restriction on the implementation approach. You can implement the function $f(I,j)$ recursively or iteratively. For dynamic programming, memoisation is important. So for a given I and j, $f(I,j)$ will be computed once and stored so that any subsequent call to $f(I,j)$ for the same I and j can directly return the stored value. Notice that I is a set of nodes and j is a single node. Set I has no ordering of the nodes but for convenience, you can impose a canonical ordering (e.g. ascending order). You can consider (I,j) as a tuple. For all (I,j) tuples, you can store the f values in a hashmap. So you can use appropriate tuple, set, hashset or hashmap classes for these.

4 Hill-Climbing Algorithm

We provide the pseudocode of a hill-climbing method and then discuss how that can be adapted to the TSP. We also discuss possible improvements that you can consider in your assignment.

4.1 Hill-Climbing Metaheuristic

We describe a sample pseudocode for a simple hill-climbing metaheuristic with random restarts.

```
function HillClimbingMetaheuristic
    Take an initial solution s, which is generated randomly or by using a heuristic algorithm.
    s* ← s ▷ the global best solution found so far
    for k ← 1 to N do ▷ N is the given max number of iterations
        Generate a neighbouring solution s' from s
        p ← 0 ▷ plateau size, when no better solution is found
        if s' is better than s then
            s ← s', p ← 0 ▷ a locally better solution found
            if s is better than s* then
                s* ← s ▷ a globally better solution found
        else
            p ← p + 1 ▷ the number of consecutive non-improving move grows
            if p = P then ▷ P is a given limit on consecutive non-improving moves
                Restart search: take a randomly generated solution and consider it as s
    return s* as the best found solution
```

4.2 Adapting Hill-Climbing to TSP

For the pseudocode for hill-climbing, below we define each term in the context of TSP.

- **Representing a TSP solution:** A cyclic tour is $(0, i, \ldots, j, 0)$ where $i, \ldots, j$ is a permutation of nodes $1, 2, \ldots, n-1$. Each such cyclic tour is a solution for TSP. Use an array $s[0..n]$ to store a TSP solution where $s[0]$ and $s[n]$ are 0, and $s[1..(n-1)$ is the permutation of $1, 2, \ldots, n-1$.
- **Computing Solution Length:** For a solution s, compute the sum of the lengths of the links $(s[k-1], s[k])$ for $k \leftarrow 1$ to n. Note $s[0]$ and $s[n]$ are both 0 for the cyclic tour.
- **Comparing Solutions:** Solution s' is better than Solution s if s' has a smaller length than s.
- **Generating a random TSP solution:** take numbers $1, 2, \ldots, n-1$ in an array $s[1..(n-1)]$. For $k \leftarrow 1$ to $n-1$, swap $s[k]$ with $s[k']$ where k' is a random number in $[1..(n-1)]$. Let $s[0] \leftarrow 0$, $s[n] \leftarrow 0$. So $s[0..n]$ is a randomly generated TSP solution.
- **Generating a heuristic solution for TSP:** $s[0] \leftarrow 0$, $s[n] \leftarrow 0$. Then, for $k \leftarrow 1$ to $n-1$, find node i that is nearest to $s[k-1]$ but not in $s[0..k-1]$ and then set $s[k] \leftarrow i$.
- **Generating a neighbour s' from a solution s:** Given $s = (0, 1, 2, 3, \mathbf{4}, \mathbf{5}, \mathbf{6}, 7, 0)$, we can have $s' = (0, 1, 2, 3, \mathbf{6}, \mathbf{5}, \mathbf{4}, 7, 0)$. Notice that segment $4, 5, 6$ in s has been reveresed to get $6, 5, 4$ in s'. See the figure below. This is essentially selecting two positions k_l and k_r and reversing the elements from indexes k_l to k_r in the solution array s to get the neighbour s'. You can randomly select k_l and k_r with $0 < k_l < k_r < n$ to generate a neighbour s' from a solution s.

- **Solution Length for Neighbours:** The inefficient way to compute the length of the neighbour solution s' would be to sum the lengths of all the links in s' from scratch. You can use an incremental and efficient alternative. Assume the segment between positions k_l and k_r in solution s has been reversed to obtain solution s'. So the length $l_{s'}$ of s' can be efficiently computed by subtracting the lengths of links $(s[k_l - 1], s[k_l])$ and $(s[k_r], s[k_r + 1])$ from the length l_s of s and then adding the lengths of links $(s[k_l - 1], s[k_r])$ and $(s[k_r], s[k_l - 1])$. For example, in the above figure, subtract lengths of links $(3, 4)$ and $(6, 7)$ from the length l_s of s and then add the lengths of links $(3, 6)$ and $(4, 7)$ to get the length $l_{s'}$ of s'.

4.3 Your Own Improvements

While a simple prototype algorithm has been discussed so far, you are free to bring any innovation in the hill-climbing approach to improve its performance. We will discuss a few potential improvements that you might consider, but feel free to bring your creativity and imagination here.

- Instead of an initial solution that is randomly generated or generated by the heuristic discussed above, you can design your own heuristic to get another initial solution.
- Instead of generating just one neighbour for a given k, you can generate several neighbours and consider the best of them as s' and then compare with s in the if statement.
- You can think of a different way to generate a neighbour rather than reversing a segment.

- While restarting the search, instead of taking a fully random solution, a number of times you can apply random segment reversing on s to get a new solution to start from.
- The restart takes place when there is no improvement in P consecutive iterations, you can consider accepting worse moves sometimes with some probability.
- Instead of running for a given number N of iterations, you might consider stopping when you do not see any improvement for a reasonably large number of consecutive iterations.

When you are implementing any of the above ideas or any of your own ideas, please clearly describe the ideas as comments in your program. Describe what you do and your intention behind.

5 Input and Output

You are given 5 problem TSP instances in text files. Download and use them with your programs.

`tsp22.txt`, `tsp23.txt`, `tsp24.txt`, `tsp25.txt`, `tsp26.txt`

Each file in the first line has a number n for the number of nodes. Coincidentally, the last two digits of the file names reflect the node counts of the corresponding TSP instances in the files.

A file with the first number n then has $2n$ numbers: 2 numbers in each of the n rows. The two numbers are x and y coordinates of a node. So the coordinates of all n points are given.

From the x and y coordinates of two nodes, you can compute the distance between the two nodes and consider that as the length of the link between the two nodes. You can create an adjacency matrix using the distances. After that, your TSP algorithms will run on the adjacency matrix.

Both of your program will show the followings as output on the screen.

- The best solution found for the TSP instance.
- The length of the best solution found
- The time taken by the program to finish

Note that the dynamic programming algorithm will return the same solution length every time you run the same program on the same problem instance. So running only once on a problem instance is fine. However, for the hill-climbing algorithm, because of randomness, every time you run, you will get a different solution, different solution length, and the time taken could be different as well. So to understand the perforamnce of a hill-climbing approach, we usually run the same algorithm on the same instance a number of times and then take the averages of the solution lengths and the execution times. Then, we use the average numbers to compare the performances.

6 Solution Comparison

To compare the two TSP algorithms, run your two programs on the given 5 TSP instances. Generate a table that compares the execution times and the solution lengths for the two programs on the 5 instances. You can use the following table format to show your results.

For dynamic programming, run the program only once on each problem instance. Dynamic programming, even with memoisation, can take long time on large instances. Please have patience. At the moment, no idea how long the largest instance could take, but allow up to half an hour. If you cannot find within that time, you can report that no solution found within half an hour. While writing the program, for test runs, you can create a small instance or use the instance shown in this document. Please run your program on the large instances, when you know your program is correct and you are running mainly to get the results. Otherwise, you will be wasting your time.

Table 1: Sample Data Table

Problem Size	Execution Time		Solution Length	
	DynaTSP	ClimbTSP	DynaTSP	ClimbTSP
22				
23				
24				
25				
26				

For the hill-climbing algorithm, run the program 5 times on each problem instance, and take the average of the solution lengths and the average of the execution times, and then fill in the table. The hill-climbing program should be very fast compared to the dynamic programming algorithm. Consider running the hill climbing algorithm for large values of N and P e.g. N could be several millions and P could be several hundreds. Along with the data table, please mention the N and P values that you finally decide to use when you run the program to collect data.

Based on the data collected and tabulated, write a very brief (50 words max) analysis of your results. That is, which method do you think performs better, and why?

7 Submission Information

1. You must use Java to do your programming. Your code should contain
   - Classes named correctly as specified in each part.
   - Comments written to explain your code segments.
   - Use command line arguments for input. For example,
     ```
     java DynaTSP tsp22.txt
     java ClimbTSP tsp22.txt 1000000 100 ▷ N=1000000, P=100
     ```
2. You must submit a readme file containing instructions on how to run your programs.
3. You must submit a document containing the table and your analysis of your test data.
4. You must include a filled in Assessment Item Coversheet; otherwise, cannot mark.
5. You should zip all files and submit the assignment via the Assignment link in Canvas.

8 Assessment Criteria

- **5 Marks** for reading the problem instance and creating the graph
- **25 Marks** for dynamic programming based TSP algorithm
  - **10 Marks** for correctly implementing $f(I, j)$ function
  - **5 Marks** for correctly computing S from S_js
  - **5 Marks** for correctly implementing memoisation
  - **5 Marks** for correctly outputting the solution
- **35 Marks** for hill-climbing based TSP algorithm
  - **5 Marks** for correctly generating an initial solution
  - **5 Marks** for correctly generating neighbouring solutions
  - **5 Marks** for getting the tasks in an iteration correctly
  - **5 Marks** for detecting situations when no improvement for long

  - **5 Marks** for generating solutions when no improvement for long
  - **5 Marks** for correctly maintaining the global best solution s^*
  - **5 Marks** for implementing at least one improvement idea

- **15 Marks** for comparing the `DynaTSP` and `ClimbTSP`
  - **10 Marks** for showing the comparison data
  - **5 Marks** for providing analysis/discussion

- **10 Marks** for using efficient data structures, explain your choices in comments
- **5 Marks** for using better code organization, explain your choices in comments
- **5 Marks** for useful commenting in the code.

END OF ASSIGNMENT