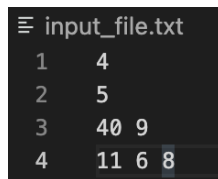


Assignment 2 – Final Report

I started my implementation of this assignment by writing out a table that describes the relationship (travel time) between the cities. I used the text input example, provided in the task sheet as my reference for this.

City	City				
		1	2	3	4
	1	0	5	40	11
	2	5	0	9	6
	3	40	9	0	8
	4	11	6	8	0

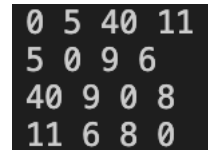
This table is critical to my understanding because it clearly describes and illustrates the format for the matrix that I needed to create in my program. In my program, the elements for the matrix are provided by the input.txt file. Each line of the file is read through, and the elements have been assigned accordingly. The screenshots below show the input.txt content and the associated matrix created:



```

≡ input_file.txt
1 4
2 5
3 40 9
4 11 6 8
  
```

Screenshot of the input_file.txt



```

0 5 40 11
5 0 9 6
40 9 0 8
11 6 8 0
  
```

Screenshot of the created matrix
outputted into terminal

As illustrated the created matrix is a recreation of the above table. Conditions have been included through the program for determine if logs need to be made to a logfile. A logfile is only create if log has been included in the terminal run command for the program.

After some trial and error, I was able to successfully implement the sequential version of the Traveling Salesperson (TSP) algorithm. I have used setters and getters to assign the matrix value as well as to allocate memory space. The pseudo code I developed for my implementation is:

```

int * allocate(matrix size){
    allocate memory space for the size of the matrix
}

void set(row, column, element value){
    set the matrix value at position row x column to the input value
}

int get(row, column){
    return the value at a position row x column of the matrix
}

int tsp(int current, int visited[], int currentDist) {
    check if we've reached the end of a path {
        if reached the end of a path {
            check if it's the cheapest path so far {
                if cheapest {
                    update cost
                }
                else {
                    move to the next path
                }
            }
        }
    }

    check if current path is worth persuing (current cost < the best
    cost) {
        if worth persuing {
            continue
        }
        else {
            stop path and move to next
        }

        loop through all possible next cities (recursively) {
            if city has already been visited {
                skip
            }
            if the city is the current city we are at {
                skip
            }
            else {
                move to this city
                add cost to the running total for the path
            }
        }
    }
}

```

The TSP algorithm remains the same for both the `tsp_sequential` and `tsp_parallel` programs. The function takes in the following parameters:

- the current city
- the visited cities array
- and the current cost.

These parameters are then passed through several for loops and if statements to determine what conditions are currently being met to determine what needs to happen next. The TSP is recursively called to calculate the final cost. The recursive call is made when the following conditions are met:

- The city has not already been visited

The next city is not the current city

The difference between the two programs is in the main function. In the `tsp_sequential` program, the main function follows this sequential order of events:

the input file is read.

A log file is created (if logging)

The distance matrix is created

The tsp function is called, passing 0 through as the current city and cost

The fastest path is printed

The best cost is printed

However, in `tsp_parallel` the main function uses the set number of processors to distribute different branches of the tree. The main function follows a parallel order of events:

Process 0 reads the input and creates the distance matrix

Process 0 evenly splits the tree and distributes the different branch of the distance matrix among the remaining processors

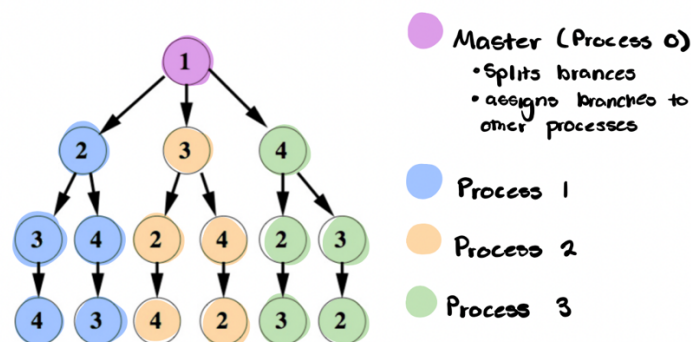
Each process computes the tsp functionality on their branch (This happens in parallel)

Process 0 brings all branches together and compare their results to determine the best route

The fastest path is printed

The best cost is printed

The breakdown of branches has been visualised in the below tree.



Using the same input file as shown above when

```
./tsp_sequential input_file.txt log
```

is called in the terminal, the following information is logged in the logfile:

```
3 1 : dropped path because 40 > 19
```

Screenshot of log.txt (logfile)

and outputted to the terminal:

```
Fastest path: 1,2,4,3
Distance: 19
```

Screenshot of the terminal

To find the runtime for this program a clock is started at the beginning of the program and ended at the program's completion. After compiling the program with:

```
clang tsp_sequential.c -o test.out
```

and running the program with:

```
time ./test.out
```

we are provided with details about the runtime in the terminal:

```
./test.out 0.00s user 0.00s system 63% cpu 0.005 total
```

I was unsuccessful in getting the parallel program to work and the current copy is segment faulting when running. Therefore, I cannot provide screenshots of terminal or log outputs.

However, if you wanted to compile the program, you will need to call:

```
mpicc -g -o tsp_parallel tsp_parallel.c -Wall
```

on the university cluster, in order to run the program

```
mpiexec -np **./tsp_parallel input_file log
```

** indicates the number of processes being used

The runtime information can be found by running:

```
time ./tsp_parallel
```

If the program were to be working, I would expect the runtime of the parallel program to be faster than the sequential program. This is because the work is evenly distributed between processes, which means that each process will complete their task and report back to the master (these do not need to be completed in order). Whereas in the sequential program, the computations need to occur in order, meaning that the program must wait for calculations and computations in order to continue.

I believe my method for distributing the work between processors will achieve the best speedup time because the work is evenly distributed between all processors.