CAB201 Programming Principles - Semester 1, 2019
Assignment: Project – Flying Postman

Due Date: 26/05/2019 23:59
Weighting: 50%
Assessment type: Individual
Specification version: Version 1.0 (10/04/2019)

The flying Postman Service operates a regular mail service to remote communities in the outback. Each postman operates an ultra-light airplane enabling him to visit up to 50 properties in any one trip which may span several days. The Ultra-light can land at any of the properties. Every day before beginning his trip, the postman collects a mailbag, and a flight plan, which usually takes him through as many as 30 properties. A printed itinerary indicates the names of the properties in the order in which they must be visited. The pilot is using satellite navigation to fly directly from one station to the next without the need for an aerial map. As the ultralight must also be re-fuelled there is an indication on the itinerary as to which stops also involve re-fuelling. Any station can re-fuel the ultralight.

The mail-sort department produces a data file containing one line for each station having mail. On each line there are 3 data items. The station's name, and the relative `(x,y)` coordinates. The first station is always the Post Office, where the Ultra-light starts and finishes the journey.

Example of a mail file:

```
PostOffice  0   0
Yurrumba    25  70
Moombar     90  128
ZigZag      123 309
OyOyOy      140 80
…
```

As several different makes of ultra-light airplanes are available to the postman, he is also provided a specification data file containing the properties of the plane he will use for the trip. These properties are the range (in hours), the speed (in km/h), the time the plane takes to take off (in minutes), the time taken to land (in minutes), and the time needed to refuel (in minutes). The data file provides them in the following format:
`range speed take-off-time landing-time refuel-time`

Example of a plane specification file:

```
3 300 3 3 10
```

Your task

Write a command line program to accept a mail file, plane specification file, trip starting time (in 24-hour time format), and optionally an output file as input. The program is to produce an itinerary for the flight listing the order in which the stations should be visited so as to minimize the tour length. The tour starts, and ends, at the Post Office. Each station is visited exactly once. Round all the times to the nearest minute. The plane must re-fuel if it has insufficient fuel to complete the next leg of the tour. Your program should also save the itinerary to an output file if the flag `-o` is provided in the arguments.

Example program usage, assuming your program is named flying-postman.exe:

```
flying-postman.exe mail.txt boeing-spec.txt 23:00 –o itinerary.txt
```

Example program output:

```
Reading input from mail.txt
Optimising tour length: Level 1...
Elapsed time: 123 seconds.
Tour time: 7 Hours 24 Minutes.
Tour length: 972.5855
qaz      ->      yhn     23:00   23:28
yhn      ->      wsx     23:28   00:10
wsx      ->      edc     00:10   00:44
edc      ->      rfv     00:44   01:30
*** refuel 10 minutes ***
rfv      ->      tgb     01:40   02:14
tgb      ->      qaz     02:14   03:00
Saving itinerary to itinerary.txt
```

Several example tour and plane files and expected itinerary outputs will be provided on Blackboard which can be used to help validate and test your program. Please note that this is just an example output, rather than the expected itinerary output from the previous example files and input.

Finding the Best Tour

The problem of finding the shortest tour is a variation on the Travelling Salesman Problem (TSP). It is NP-hard, so do not even consider always completing an exhaustive search for the minimal tour length.

For this assignment, there are several solutions you may implement. Each level is worth different marks (higher levels are worth more), which will be specified in the CRA. Implementing the simplest level is worth at least a pass.

Level 1 - Simple Heuristic Approach

Implement a simple heuristic algorithm, which provides a reasonable solution, as follows:

The tour starts empty and we add one station at a time to the tour. The tour is always closed - i.e. it starts and ends at the Post Office station. Suppose that we already have N stations on the tour. The Post Office is always first, and all other stations follow in the order visited, with the final leg of the tour back from the last station to the Post Office. When we insert the `N+1` station it can be inserted at any one of `N` positions (from `2` to `N+1`). We need to find the position that leads to the shortest tour. This is done by computing the impact of inserting the new station between any pair of stations already on the tour and choosing the insertion that minimizes the new tour length.
Note that this is an approximate solution - it does not necessarily find a globally minimal tour length, but it is a reasonable start. This is a minimum requirement for this assignment.

Level 2 - Improved Heuristic Approach

This solution should improve upon your implementation of the level 1 heuristic approach.

After completing level 1 solution, the tour is complete. However, the actual tour obtained depends on the order in which stations were added to it. A different ordering in the input file may lead to a different solution, and different solutions will have different tour lengths. Level 1 finds one of these tours. Let's call the solution obtained in level 1 the ***current tour.***

The current tour can be re-arranged and possibly even become shorter. Suppose that a station on the current tour is moved to another position on the tour. It can be done by **removing it from the tour, and then inserting it in a different position**. This may result in a shorter tour, in the same length tour, or in a longer tour. If the tour is shorter then we make the new tour the current tour.

The optimization can proceed as follows: take the current tour and systematically go through each station, and try to move it to another position. Most of the moves will not lead to an improvement and so we leave the tour unchanged. However, if we find a shorter tour, we adopt that new tour and it becomes the current tour. **We must then restart the search** for an even shorter tour, in the same manner, always re-starting with the current tour. If we find that none of the stations on the tour can be moved such that the tour length becomes shorter then we are done.

Level 3 - Exhaustive Search

Implement an exhaustive search. This should only be run if the number of stations is such that an exhaustive search can be computed in a reasonable amount of time - the limit for this assignment is 12 or less stations. Otherwise, your program should run the heuristic approach (level 1 and level 2)
You should make use of an efficient algorithm to calculate all permutations, for example Heap's algorithm. If you have successfully completed the Week 7 AMS exercise, you should be able to adapt your solution from there.

Level 4 - An Improved Heuristic

To receive a higher grade, you must find an improved (more efficient) solution to the problem. The literature on this problem is quite large, and finding pseudocode for more efficient solutions should be quite straightforward. However, it is up to you to port this pseudocode to fit within the task. To receive marks for implementing a more advanced solution, **you must include a section in the statement of completeness report that states the source of the algorithm**, provides the pseudocode, and briefly compares the run times of your original approach to this new approach. More instructions on the format of this report and an example will follow closer to the due date.

Bonus - Account for Unreachable Stations

NOTE: this is **HARD!!** Do not attempt until all other levels are completed and tested.

The final level is to implement an algorithm which takes into account the possibility of an impossible flight leg. In all previous versions, your solution may assume that each station is within range of every other station based on the plane's properties. In this implementation, your algorithm will need to take into account the possibility that some stations may not be reachable from certain other stations. All stations will remain reachable from at least two stations (so that they can be arrived at, and departed from), but the tour is constrained to feasible legs.
This implementation should build upon level 2 or 4. It should not be an adaption of level 3 - generating all permutations and throwing away impossible legs is insufficient. The tour must be complete and visit all stations. For this version, you will also need to provide a description of your algorithm in the statement of completeness report.

While this may initially sound straight forward, it will be quite complex to do efficiently. Do **not** attempt this version until you have every other aspect of the assignment resolved and tested.
This level will be worth bonus marks. Please note that you can not receive more than full marks for the assignment, however these marks may make up for other areas where you did not receive full marks.

Code Quality and Submission Details

Coding style

An important aspect of this task is learning to take a problem description and transform it into object-oriented code. For this task, you are not provided with class specifications. Instead, you will need to take the problem definition and break it down into logical classes, making use of the object-oriented principles you have learnt throughout the semester.

You may want to use the following classes in your program:

Class STATION –
The station class maps to the real object station. The minimal attributes needed are the name and coordinates of the station.

Class PLANE -
Corresponds to the plane used to fly the tour. The attributes should reflect the properties in the provided specification details. It is up to you to determine the relevant behaviours.

Class TOUR –
The tour class corresponds to an optimized tour of stations. The attributes that must be kept includes the list of stations on the tour. The order in which stations are stored reflects the order of the tour. Information that can be calculated (through implemented methods) from the data is the tour length in Km and in Hours/Minutes, and the exact itinerary (given a starting time, plane characteristics, and stations).

However, these are just a starting point. You will likely need to implement other methods and classes to complete your program. For example, you may need to consider how to represent time in your program. Another common operation in the context of our program is the calculation of the Euclidean distance between two stations. This is defined as

$Ed = sqrt((x1 - x2)^2 + (y1 - y2)^2))$.

The implementation of this is simple, but you will need to consider where the appropriate place to implement it is.

What to hand in

You will be expected to hand in a solution file that can be opened and run in the visual studio environment of the QUT labs (as used in the pracs). The submission must also include all source code, and instructions for running the code. You will also need to provide a statement of completeness report detailing your implementation and any incomplete features or bugs. Failure to declare bugs will lead to additional loss of marks on account of poor testing. Detailed submission instructions and templates will follow closer to the due date.

Academic Integrity

Please read and follow the guidelines in QUT's Academic Integrity Kit, which is available from the Blackboard site on the Assessment page. Programs submitted for this assignment will be analysed by the MoSS (Measure of Software Similarity) plagiarism detection system (http://theory.stanford.edu/~aiken/moss/).

Please do not share your code with other students. You may of course help other students, or receive help, but code sharing is strictly forbidden.

Getting Started

Getting started on a large assignment like this can seem daunting at first, however it can be broken down into manageable chunks. You will be given starting points to particular aspects in the workshops. The following is a recommended way to begin to break the task down. Make sure you test your code, logic, and code quality thoroughly after each step before moving onto the next.

- Before you begin coding, you should design the structure of your solution. You should design the classes and their attributes and behaviours - UML diagrams would be helpful to represent this.
- Start by implementing a simple class, and testing it. For example, if you decide there should be a Station class, this is likely to be a simple place to start. Once you have created the class, test that it functions correctly. Refer back to the workshop exercises where we designed test cases for inspiration.
- Continue implementing the basic classes, and testing them. Create some basic functionality to generate the tour if it was simply in the order of stations provided, without worrying yet about calculating the optimal tour. Use this to test that your program outputs a tour which correctly calculates time, distance and refuelling. Initially test this by hard-coding the stations in Main.
- Write code to read the station file in from the user, then the plane file.
- Implement the Level 1 - Simple Heuristic Approach to calculate the tour. Compare your results and run time to the examples provided on Blackboard.
- Only move onto the next level if you complete the above and have tested it thoroughly.

Remember to keep backups of your work. Save a separate copy each time you complete a milestone in the assignment, so that you can go back to it if you make a new change which breaks your previously working version.

Final Comment

Though all care has been taken in the production of this specification and related documentation, there may be a need to notify by email any alterations/clarifications to this specification and related documentation.

CHECK YOUR QUT EMAIL & ANNOUNCEMENTS DAILY