

DESIGN DECISIONS

Before describing the design decisions, we would like to provide a brief overview of our software by summarizing the classes used in our software and their responsibilities:

Class	Responsibility
MainFrame	This class is the starting point for our application and contains the <code>main()</code> . It extends <code>JFrame</code> to display a GUI, and comprises of a panel to display the graphical data and a menu bar that contains functions like New, Load, and Save.
City	This class is the data structure for a city (node in the shortest path graph). It contains methods to draw the city and also the path from this city to another city.
Route	This class is the data structure for the route between two cities (edge in the shortest path graph).
WorkSpace	This class is the repository that contains the list of cities and route information.
WorkSpacePanel	This class extended <code>JPanel</code> and displays the graphical data. It also allows the user to mark the cities with a mouse click and move the cities with a mouse drag operation.
TSP	This class implements a greedy algorithm for the Travelling Salesman Problem.

In our software, the idea is to allow the user to add/move cities on the screen and whenever the cities are added/moved, the shortest path between the cities should be re-evaluated and updated on the screen in a real-time manner. We have implemented the Observer pattern. The following section describes the rationale behind our decision:

Design Decision #1: Make the `WorkSpace` class an `Observable` and the `TSP` class an `Observer`

To re-evaluate the shortest path, the trigger is the addition/movement of a city. The observer pattern helps us to design an optimal solution for this case. We can make the `WorkSpace` class an `Observable` as it contains information about the cities and the `TSP` class an `Observer` of `WorkSpace`, so that changes in `WorkSpace` is notified to `TSP`. The `TSP` can then do its job of re-evaluating the shortest path. To facilitate notification, we have implemented two methods `addNewCity(City)` and `moveExistingCity(City, int, int)` in the `WorkSpace` class which will allow the other classes to add a new city and move an existing city respectively. Both these methods call the `setChanged()` method to indicate a change in the object data and then the `notifyObservers()` methods to notify the observers. This way, whenever a city is added/moved the shortest path is re-evaluated.

Design Decision #2: Make the `TSP` class an `Observable` and the `WorkSpacePanel` class an `Observer`

To update the route on screen, the trigger is the re-evaluation of the shortest path. The observer pattern helps us to design an optimal solution for this case. We can make the `TSP` class an `Observable` as it re-evaluates the route and the `WorkSpacePanel` class an `Observer` of `TSP`, so that whenever the route is re-evaluated, the change is notified to `WorkSpacePanel`. The `WorkSpacePanel` can then do its job of updating the route on screen.

Since, the `TSP` is an `Observer` of `WorkSpace`, the re-evaluation of the shortest path is done within the `update(Observable, Object)` method in `TSP`. In this method, once the shortest path is evaluated, we call the `setChanged()` method to indicate a change in the route and then the `notifyObservers()` methods to notify the observers. This way, whenever the route is re-evaluated the route is updated on the screen.