Aayush Gupta
Jeetendra Ahuja

1) **Algorithms**

We have referred A* algorithm from http://ieeexplore.ieee.org/document/7863246/keywords as mentioned in Part-1 of project. Brief description of algorithm is all follows:

A* algorithm is pathfinding algorithm and is based on dijkstra's algorithm. It is used in graph traversing applications to find the optimal path. The main difference dijkstra's algorithm is that unlike greedy approach it uses knowledge to get to destination. Heuristic is chosen based on knowledge and domain. Based on that an underestimate is calculated generally called as h(n). Total cost f(n), current cost plus this h(n) is evaluated. We have used Java and Swing to implement the A* algorithm.

Pseudo code:

- Initialize closed list to NULL
- Initialize open list to NULL
- Add start node to open list
- Initialize path to NULL
- While open list is not empty
    - Set current node with node s.t. value of f(n) is minimum
    - Remove current node from open list
    - If current node is destination, then break
    - For every neighbor of current node
        - Find f(n) = g(n) + h(n)
        - If neighbor is in open list and current cost is less than f(n) then continue
        - Else if neighbor is in closed list
            - If current cost is less than f(n) then continue then continue
            - Remove current node from closed list
            - Add node to open list
        - Else add current node to open list and update its h(n)
        - Update neighbor's g(n) with f(n)
        - Update path list with current node
    - Add current node to closed list
- End

2) **Heuristic**

Following two heuristics are implemented

Straight line: From the current node to destination node a straight-line distance is used as g(n). This is heuristic is underestimate and is calculated using point distance formula, a Euclidean distance.

Fewest links: Number bode to reach to destination can minimum be one. This underestimate is used in this second heuristic. Hence, in this type number of links are measured.

3) **Program Design**

We have 3 classes whose brief description is as follows:

- **Node**: This defines structure of one city. Each city is represented as node. It has co-ordinates(x & y both), connections with other cities(which is a list of Nodes), distance from start city(depends on heuristic and it is not a straight line distance), distance from end city(depends on heuristic and it is straight line distance from end city), color of city, total distance which is a sum of distance from start city and end city. All variables above have methods to access them accordingly.
- **AStarGUICode**: This class is responsible for GUI code. It has user input panel in which user has to browse for locations and connections file. After giving these inputs, user has to press on "Plot" button then all cities loaded from locations file are plotted on map of size 800*800 as discussed in project and will be connected to other cities as loaded from connection file. Each city is shown as circle and lines are used to show its connection to other cities.

  Now user has to give valid start and end city and choose heuristic. We have 2 heuristics as straight-line distance and fewest links given in drop down list and user choose one of them. Straight line distance heuristic is default if user doesn't select. Once, user give these inputs then he has to press on "Find Path" button to see path from start city to end city. An error will be shown if user gives invalid city. A green colored path will be drawn as evaluated on map plotted by "Plot" button.

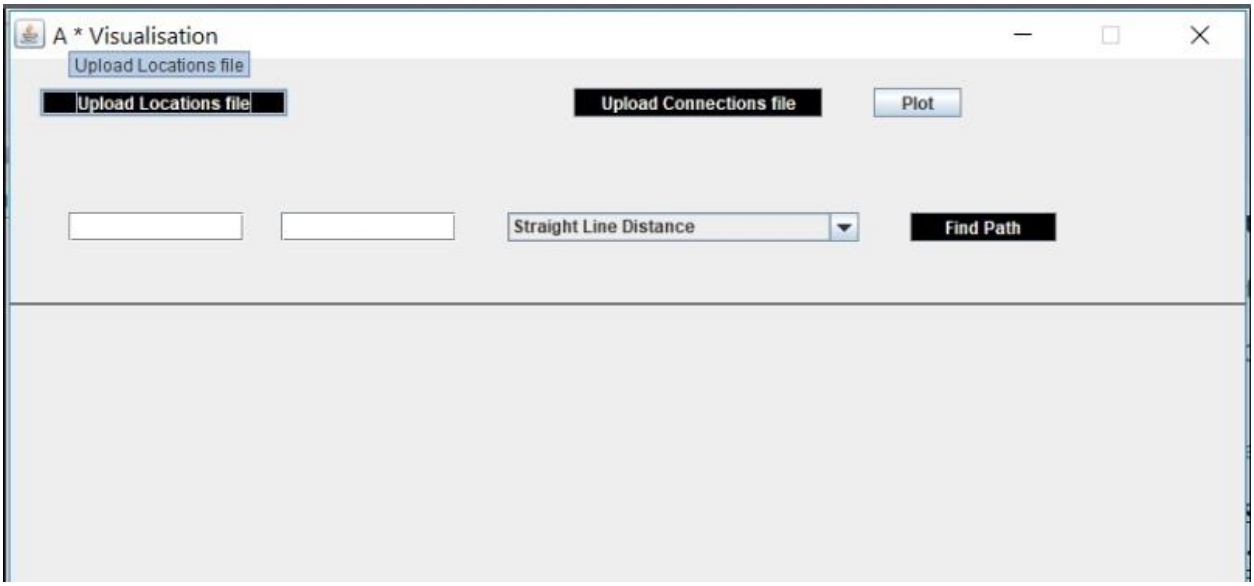
  User can play around on map plotted as below:

  1) Drag and Drop city.
  2) User can't drag and drop beyond map size which is 800*800.
  3) Right click on city to skip it. All connections will be removed.
  4) Skipped city can't be dragged.
  5) Skipped city will be shown as red colored.
  6) Right click on skipped city to revive it again. All connections and actual color will be restored.
  7) When path is getting drawn from start to end city, user cannot drag and drop any city.
  8) City name is shown on top of each city's circle.

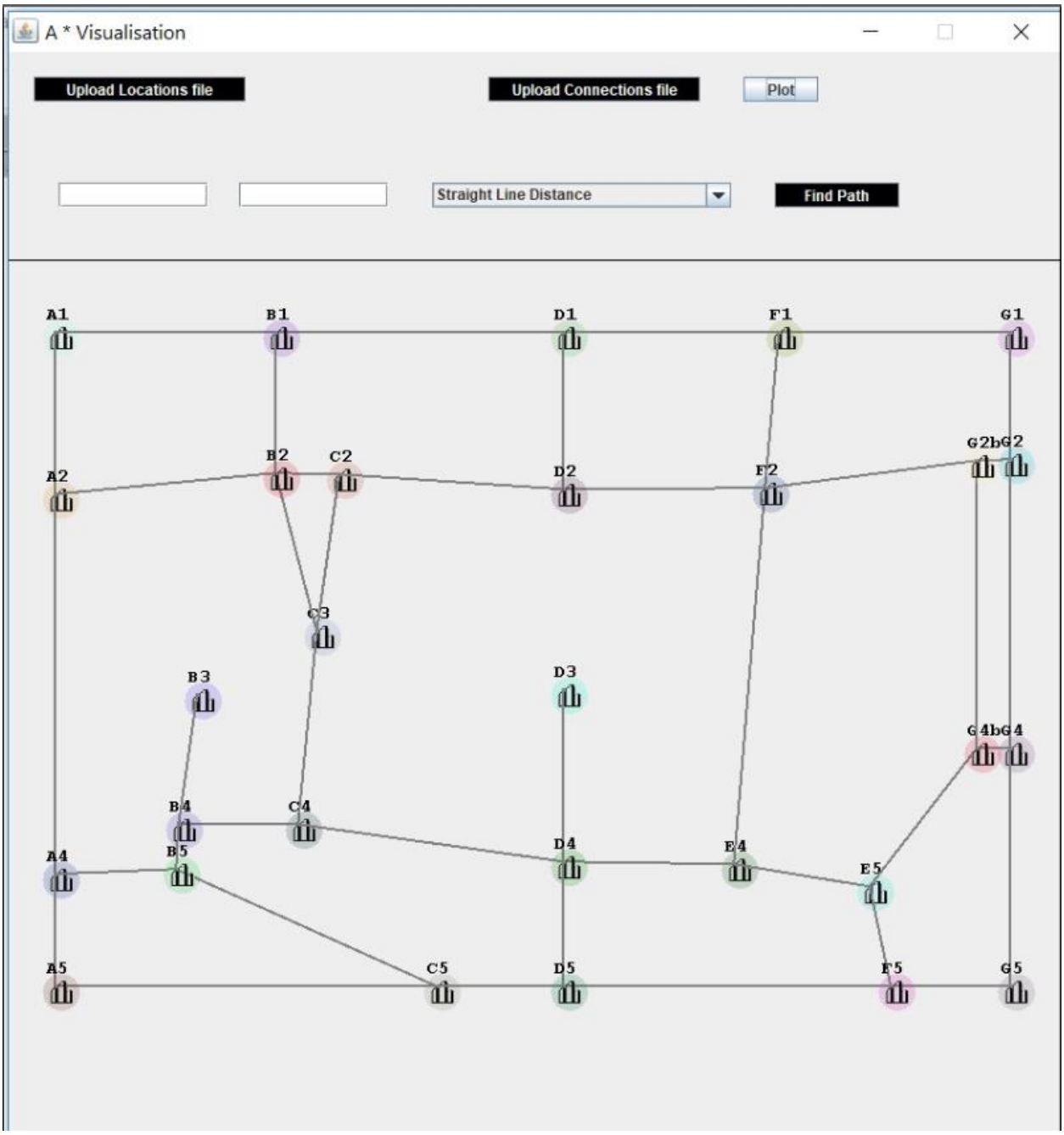
- **AStarCoreLogic:** This file contains mainly two methods: FileRead() and traversingAStar() First function is used to read metadata required to plot the map. File handling and reading city location and connection details are main task of this function. traversingAStar function will use node map created from File Read. Also uses skipped city list to find optimal path.

4) **Steps to execute the application**

- Upload map data: Use “Upload locations file” push button to upload locations data. Use “Upload connections file” to select connection details file. These data are used to create the map data. After uploading files need to click “Plot” button update the details to data structures. After clicking this button map should get displayed on panel.

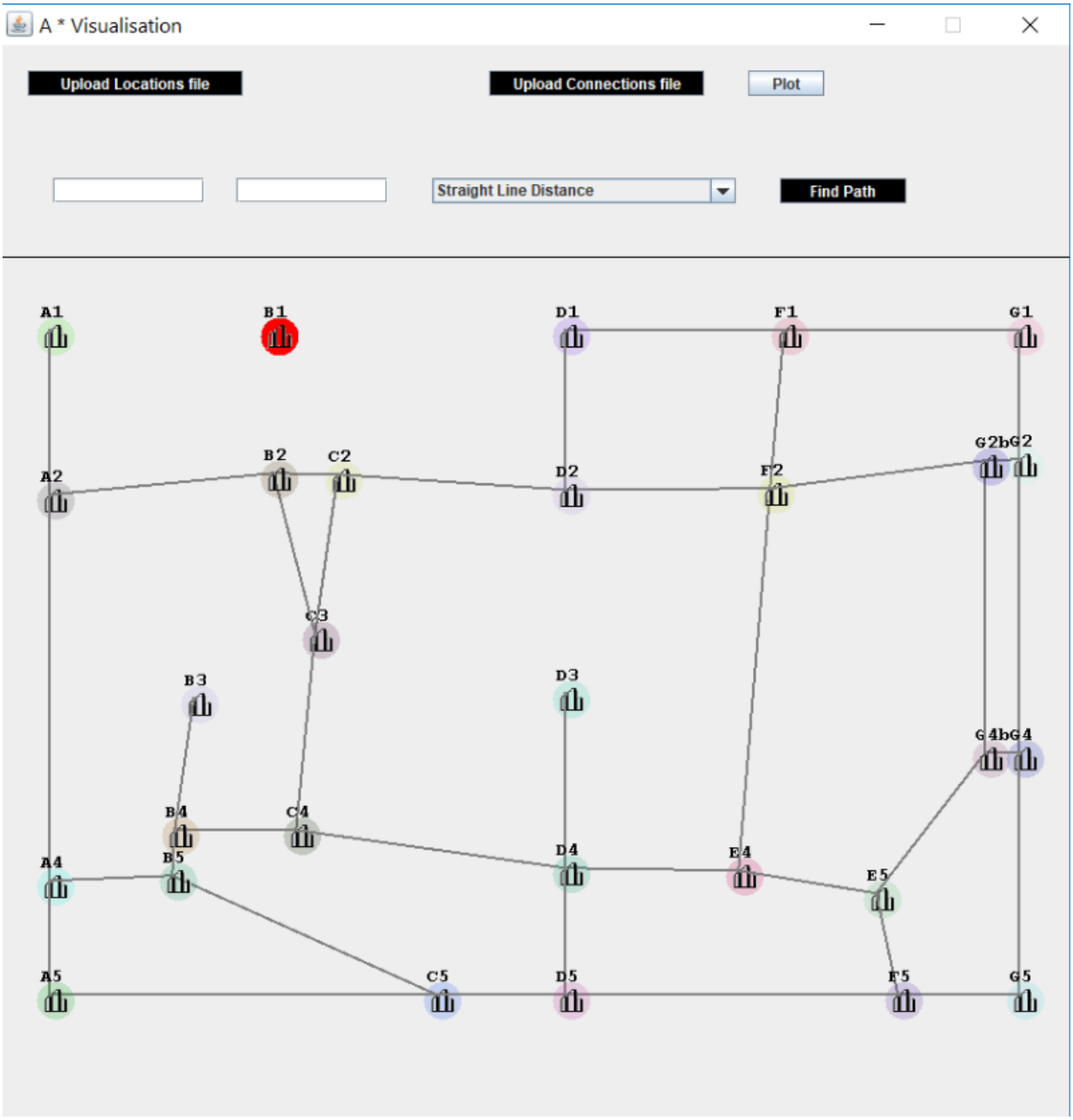


After clicking plot button map will get displayed as below:



- Enter city details: Start and end city names needs to be entered in text box. Moving cursor over it would give additional information about the test box.
- Select heuristic: Choose required heuristic from drop down for the specific run.
- Finding path: To get the optimal path “Find Path” button needs to click. This will show step by step execution of the code. Yellow lines are the city path which are traversed while finding the path. Whereas, green represents the final path found by A star algorithm.

- Node movements: City location can be changed by dragging them to new location. Constraint has been added such that no node can be dragged out of the range specified in specification.
- Skip node: A node can be skipped from map by using mouse right click button. To restore again right click on the node. Following image shows node B1 as skipped node.



Due to some technical difficulties in JFrame, you need to close GUI every-time if you find path and need to find path again between same/different start or end points. Didn't get a time to fix it, sorry ☹