Austin Lackey
Gabriel Contreras

Project 1: A*

Algorithms:

Both algorithms share a few data structures in common. They share a mapping of all the names of the cities, and their data. They also share a mapping of all the city names to the names of the cities that came before them, that is, the mapping of the name of the third city in the path is the name of the second city in that path.

Straight Line Distance:
This uses distance as a heuristic for the A* algorithm. This is calculated using a static method in the main class. Here, the distance is defined as the Euclidian distance from each city to the destination city. The comparison is done inside a priority queue of all the cities based on their distance from the destination. The city objects store their distance from the destination, while the priority queue looks up the distances using the outward facing methods.

Fewest Links:
In practice, this algorithm uses a queue to simulate a breadth first search. As the heuristic is simply the opposite of depth, this technique yields the best results. Because of this, a simple linked queue can be used instead of a priority queue. This, coupled with the lack of distance calculations, significantly increases performance. Elements in the queue are placed and removed in constant time, while the priority queue needs logarithmic time to achieve this. This too, aids in performance gains.

Function Description and Program Design:

The program starts by asking the user for some information regarding the files and execution like if the user would like straight line distance, the locations of the files, etc. Then, the files are read and a hash table (or hash map as Java calls them) of all the cities is created. Finally, the algorithm specified by the user is run.

To achieve this, there are two classes. The first class is the City class. It holds all the data for each node on the graph. It includes such city information as

- X location
- Y location
- Name
- A list of strings corresponding to the connections to other cities.
- The straight distance to the destination (not calculated unless necessary)

The second class, Project, houses the majority of the program. It:

- Sets up the data structures
- Accepts input from the user
- Handles file input
- Runs the chosen algorithm
- Times both previous activities
- Contains the important helper methods (discussed below)
- Outputs the results to the user

The algorithms are aided by several helper methods. The first, getPath, returns a string representing the path the algorithm has thus far traveled to get to the current location from the starting location. The second, getDistanceTraveled, returns the distance traveled so far given a particular current location and starting location. The third method, getConnections, retrieves the number of links through which the algorithm has traveled to reach the current location in the fewest links methodology.

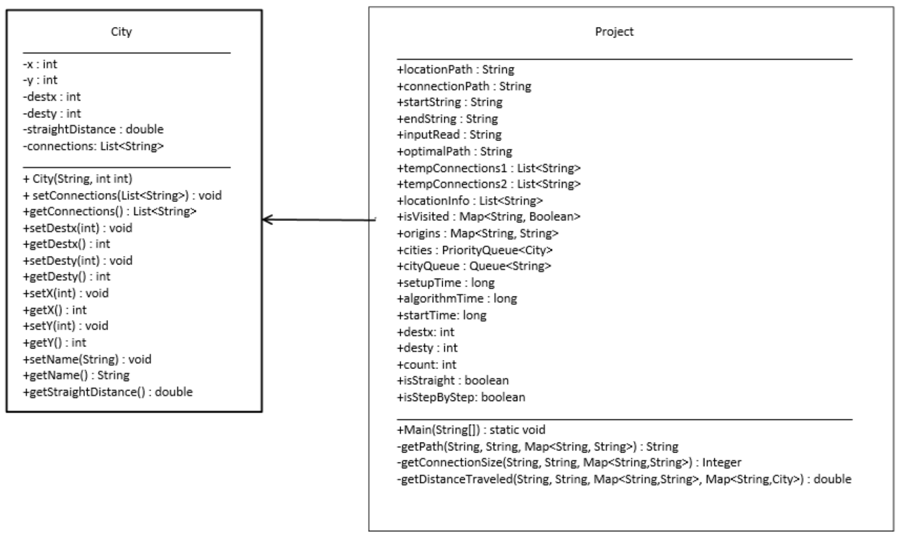
Perhaps the most crucial part of the program is the origins hash map. For the name of any city location, this map returns the name of the previous location. This is important because there can be a great deal of backtracking in the A* algorithm. As such, it is useful to store a path going backward. This aids in measuring distance traveled as well as creating the outputted string path that the user sees.

In addition, a set (in Java, an implementation of a hash table) is used to determine if a city about to be added to the program is one that the user has specified not to include. Not only does the program have to prevent the city from being added to the list of cities, it must also be removed from the connection lists of any other cities that would previously have connected to it. A set is a good data structure for this, as it can usually inquire about containment in constant time.

Here is a UML Diagram of the project:

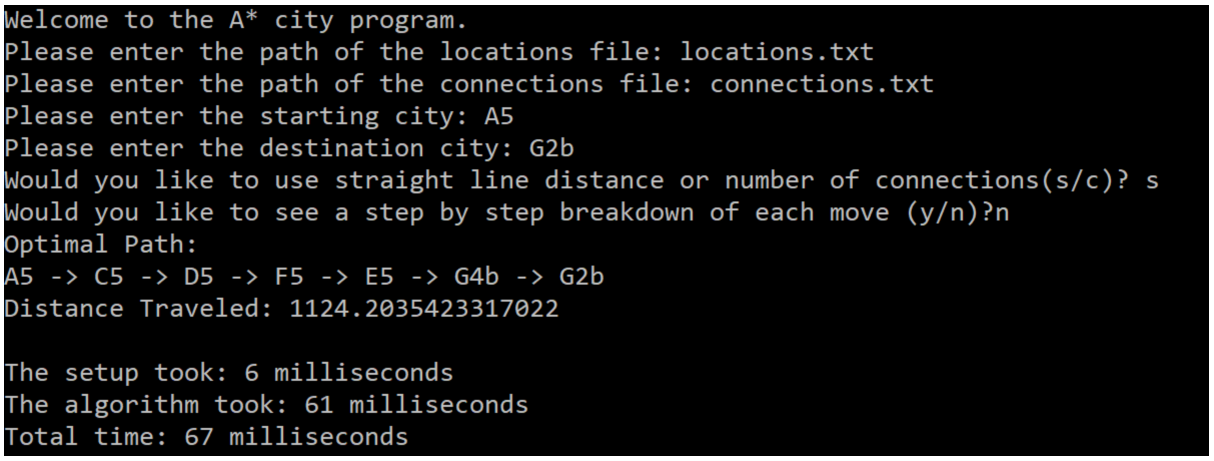
Project 1 UML Diagram

Monday, October 23, 2017 8:42 PM

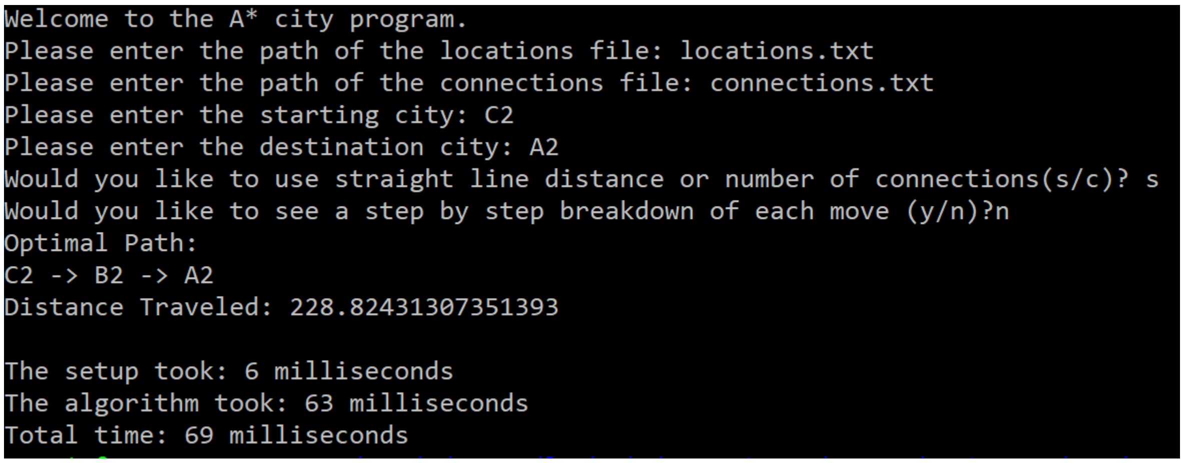

City
-x : int
-y : int
-destx : int
-desty : int
-straightDistance : double
-connections: List<String>
+ City(String, int int)
+ setConnections(List<String>) : void
+getConnections() : List<String>
+setDestx(int) : void
+getDestx() : int
+setDesty(int) : void
+getDesty() : int
+setX(int) : void
+getX() : int
+setY(int) : void
+getY() : int
+setName(String) : void
+getName() : String
+getStraightDistance() : double
Project
+locationPath : String
+connectionPath : String
+startString : String
+endString : String
+inputRead : String
+optimalPath : String
+tempConnections1 : List<String>
+tempConnections2 : List<String>
+locationInfo : List<String>
+isVisited : Map<String, Boolean>
+origins : Map<String, String>
+cities : PriorityQueue<City>
+cityQueue : Queue<String>
+setupTime : long
+algorithmTime : long
+startTime: long
+destx: int
+desty : int
+count: int
+isStraight : boolean
+isStepByStep: boolean
+Main(String[]) : static void
-getPath(String, String, Map<String, String>) : String
-getConnectionSize(String, String, Map<String,String>) : Integer
-getDistanceTraveled(String, String, Map<String,String>, Map<String,City>) : double

Results:

The algorithm successfully runs through the cities, and finds the most efficient path. It outputs this path either only at the end, or in steps along the way. Here are a couple of examples of representative output:

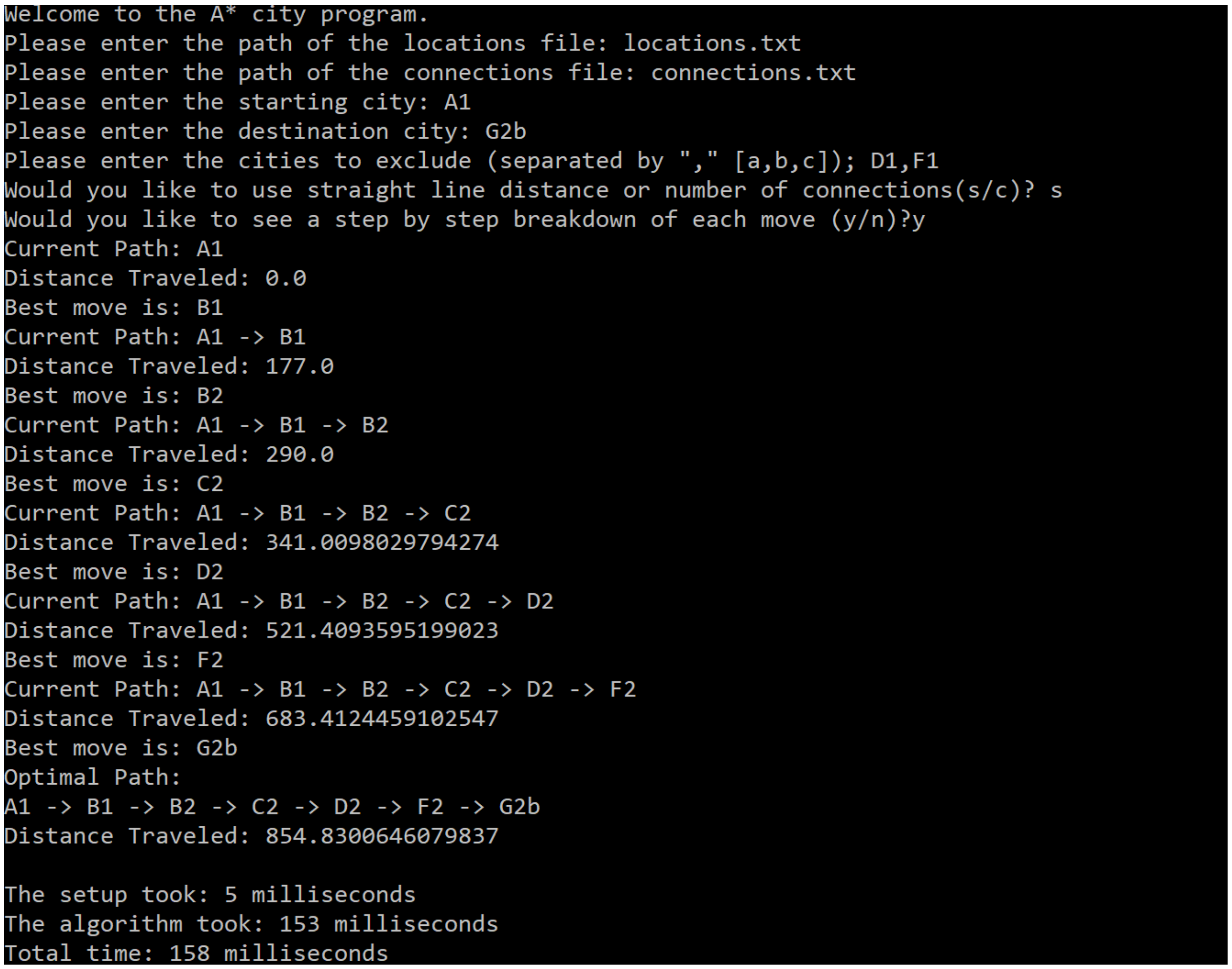


```
Welcome to the A* city program.
Please enter the path of the locations file: locations.txt
Please enter the path of the connections file: connections.txt
Please enter the starting city: A5
Please enter the destination city: G2b
Would you like to use straight line distance or number of connections(s/c)? s
Would you like to see a step by step breakdown of each move (y/n)?n
Optimal Path:
A5 -> C5 -> D5 -> F5 -> E5 -> G4b -> G2b
Distance Traveled: 1124.2035423317022

The setup took: 6 milliseconds
The algorithm took: 61 milliseconds
Total time: 67 milliseconds
```



```
Welcome to the A* city program.
Please enter the path of the locations file: locations.txt
Please enter the path of the connections file: connections.txt
Please enter the starting city: C2
Please enter the destination city: A2
Would you like to use straight line distance or number of connections(s/c)? s
Would you like to see a step by step breakdown of each move (y/n)?n
Optimal Path:
C2 -> B2 -> A2
Distance Traveled: 228.82431307351393

The setup took: 6 milliseconds
The algorithm took: 63 milliseconds
Total time: 69 milliseconds
```

When cities are omitted, the pathway becomes considerably longer, because often the fastest path no longer exists. Here is such an example:



```
Welcome to the A* city program.
Please enter the path of the locations file: locations.txt
Please enter the path of the connections file: connections.txt
Please enter the starting city: A1
Please enter the destination city: G2b
Please enter the cities to exclude (separated by "," [a,b,c]); D1,F1
Would you like to use straight line distance or number of connections(s/c)? s
Would you like to see a step by step breakdown of each move (y/n)?y
Current Path: A1
Distance Traveled: 0.0
Best move is: B1
Current Path: A1 -> B1
Distance Traveled: 177.0
Best move is: B2
Current Path: A1 -> B1 -> B2
Distance Traveled: 290.0
Best move is: C2
Current Path: A1 -> B1 -> B2 -> C2
Distance Traveled: 341.0098029794274
Best move is: D2
Current Path: A1 -> B1 -> B2 -> C2 -> D2
Distance Traveled: 521.4093595199023
Best move is: F2
Current Path: A1 -> B1 -> B2 -> C2 -> D2 -> F2
Distance Traveled: 683.4124459102547
Best move is: G2b
Optimal Path:
A1 -> B1 -> B2 -> C2 -> D2 -> F2 -> G2b
Distance Traveled: 854.8300646079837

The setup took: 5 milliseconds
The algorithm took: 153 milliseconds
Total time: 158 milliseconds
```

Performance:

The algorithm as written consistently runs in under 70 milliseconds on a portable computer with 4GB of RAM and an i5-6300 processor, while the setup can add a few milliseconds on top of that. This is shown as the last few lines of output on each run of the program. This is increased, or course, if the user specifies the detailed step-by-step breakdown of the algorithm, output to the console is one of the slowest operations the program performs. The fewest links version runs significantly faster, because it does not have to calculate distances, and it does not have to organize the cities in the priority queue, which takes longer to operate on than a standard queue.

The program was written with performance in mind. The data structures chosen all operate quickly on the common methods called, usually under logarithmic (priority queue) constant (hash maps) time. That said, the program was written primarily to be simple without too many lines of code or too much shared state. This leads to better code, but sometimes lacking performance. For instance, there is a fair amount of passing by value of data structures in the methods. A more performant model would be to use static or instance variables in the project class and share references to these.

Task Management:

Austin Lackey: initial project setup, straight line algorithm, report
Gabriel Contreras: setup modifications, file io, fewest links algorithm
Both: tons of small adjustments (difficult to report without version history)