FlightFinder: Graphs and Shortest Paths

Karsten Steinhaeuser        Aaron Wenger

Notre Dame, IN - December 15, 2003

Abstract

FlightFinder is a GUI-based application which allows the user to find the shortest flight route (in terms of time) between two airports in a flight network. The program represents the flight network as a graph, with the airports as vertices and the available flights as weighted edges. The shortest-path is identified using Dijkstra’s Algorithm and is presented to the user as a series of airport connections. The design team’s final program, which includes a custom graph class, provides an elegant solution to the problem of identifying optimal flight routes.

Keywords

graph, adjacency list, adjacency matrix, shortest path, Dijkstra’s Algorithm

1 Keyword Definitions

• **Graph** - Entity consisting of a non-empty set \( V \) of vertices and a (possibly empty) set \( E \) of edges

• **Adjacency List** - Graph representation using a list of source vertices, each of which is a list of pairs containing destination vertices and the corresponding edge weights

• **Adjacency Matrix** - Graph representation using a matrix to indicate the weight of the edges connecting two vertices
• **Shortest Path** - The path between two vertices in a weighted graph for which the sum of the weights of edges travelled is a minimum

• **Dijkstra’s Algorithm** - $O(n^2)$ shortest-path search algorithm for weighted, cyclic graphs discovered by Edsger Wybe Dijkstra (1930-2002)

## 2 Introduction

The design team sought to develop an application for finding the shortest airline connection (in terms of travel time) between the hometowns of any two students or Teaching Assistants in the Data Structures class. The team was initially confronted with two important design decisions: how to effectively represent a flight map and how to efficiently search the map to identify the optimal airline connection.

The team decided to represent the flight map as a graph, with the airports as vertices and the available flights as weighted edges. The team developed a custom Graph class for this purpose. As an additional note, the group actually developed two separate graph classes, one which represents the graph as an adjacency matrix and one which represents the graph as an adjacency list. These separate classes provide an identical interface, which is inherited from the Graph base class; they differ only in internal implementation.

While choosing a data structure with which to represent the flight map was fairly simple, the choice of a shortest path algorithm was more difficult. Ultimately, the group decided on Dijkstra’s Algorithm, an $O(n^2)$ algorithm for shortest path searching. The algorithm resembles a breadth-first search in that it visits vertices on a frontier moving outward from the source vertex. However, it differs in the way in which it determines which vertex should be visited next. This algorithm provided an appropriate means for solving the shortest path problem.

To complete the application, the team developed a graphical user interface (GUI) using Win32 API. The final application loads the flight information from a provided data file, displays a map with the available departure and destination airports, and allows the user to select the departure and desti-
nation airports of interest from two drop-down menus. After completing the search, the program marks the map to indicate the selected departure and destination airports with green and yellow squares, respectively. It also provides the user with the travel time for the shortest connection and a listing of all airports visited on the trip (includes departure/destination).

3 Implementation

The group implemented the solution described above as a Windows application written using the C++ programming language in the Visual Studio .NET Development Environment. Each major component was implemented separately as a class; once all the pieces were completed, they were interfaced together in the final solution. The group divided the solution into three primary tasks: implementing the graph data structure, implementing the shortest path algorithm, and implementing the graphical user interface. What follows is a detailed discussion of each task.

3.1 Graph Representations

3.1.1 Adjacency List

As mentioned above, the group provided two implementations of the custom graph class. One of these implementations represents the graph as an adjacency list. In general, an adjacency list contains of a list of $n$ source vertices, each of which contains a list of pairs. In turn, each of these pairs contains first the destination vertex, and second the weight of the edge connecting that vertex to the source. Please refer to the diagram below for a simple example of an adjacency list representation of a graph.
The AL\textunderscore Graph class (Adjacent List) inherits from a generic Graph class, which implements those functions that are independent of the graph representation and provides a standard graph interface. The adjacency list is implemented as a vector of lists of pairs. The vector container was chosen to hold the source vertices because it provides constant time access to elements using an index. Lists were chosen for the additional information because they provide constant time insertion and deletion, as well as linear search time. Pairs were used for the obvious reason that they provide a simple, templated container for storing two elements.

3.1.2 Adjacency Matrix

The group’s second implementation of the graph class represents the graph as an adjacency matrix. In general, a graph with $n$ vertices is represented by an $n \times n$ adjacency matrix. Each entry $i, j$ in the matrix contains the weight of the edge connecting vertices $i$ and $j$ if it exists; an entry of zero indicates that there is no edge between the two vertices. Please refer to the diagram below for a simple example of an adjacency matrix representation of a graph.

\[
\begin{array}{cccc}
A & B & C & D \\
A & 0 & 5 & 2 & 0 \\
B & 5 & 0 & 3 & 0 \\
C & 2 & 3 & 0 & 8 \\
D & 0 & 0 & 8 & 0 \\
\end{array}
\]

Like AL\textunderscore Graph, the AM\textunderscore Graph class (Adjacent Matrix) inherits its interface and representation-independent functions from the generic Graph class. Regarding internal data structures, the group chose to represent the adjacency matrix as a vector of vectors to take advantage of the constant time access, which is important for performing certain operations.
3.1.3 Class Hierarchy

While the two version of the graph have the same interface, there are important differences in implementation that manifest themselves as differences in the performance of algorithms which operate on the graphs. For example, determining whether there is an edge between two specified vertices is a constant time operation for an adjacency matrix implementation of a graph (look at the relevant entry $i, j$ in the matrix), but a linear operation on the number of edges for an adjacency list implementation (may have to scan the entire list to find the desired vertex).

As another example, iterating through all the edges in an adjacency matrix implementation of a graph requires $n^2$ time (where $n$ is the number of vertices in the graph; must check every entry in the matrix), but performing the same operation on a graph implemented as an adjacency list requires $n + m$ time (where $m$ is the number of edges in the graph). Consequently, for certain cases it is better to use the adjacency matrix representation, but for others the adjacency list implementation is preferable.

3.2 Iterators

The group implemented iterators to simplify use of the graph classes. Due to the complexity of the graph data structure, it was necessary to implement two types of iterators: one to move through the vertices (vertex_iterator), and one to move through the edges (edge_iterator). The iterators were implemented as friend classes nested within the graph classes, and provide a convenient interface for moving through a graph. This was particularly helpful in the implementation of the shortest path algorithm.

3.3 Shortest Path Algorithm

As was mentioned previously, the group chose to implement Dijsktra’s Algorithm to solve the shortest path problem. This algorithm operates by moving through a graph, building a set of vertices for which the shortest path from the source is known. In each iteration, the algorithm determines which of the vertices not in the set can be reached by the shortest path from the source consisting only of vertices in the set. This vertex is added to the set, and the algorithm continues. The algorithm terminates when the desti-
nation vertex is added to the set. The groups implementation of Dijkstra’s Algorithm relied on the edge and vertex iterators, as well as the color and value properties of the vertices. It operates in $O(n^2)$ time and completes sufficiently fast for this application (there is not a noticeable delay).

### 3.4 Graphical User Interface

The final component of the FlightFinder implementation was the graphical user interface. Due to the simplicity of the interface, the group chose to implement it using the Win32 API. The interface consists of a graphical representation of the available flight destinations (a map of the United States with dots for each airport in the known network), two drop-down menus for selecting the flight route, a button for submitting a flight request to the program, and a number of static labels for displaying text to the user. This simple interface was sufficiently flexible for the relevant application and provides a convenient means for the user to interact with the program. Please refer to the Appendix for a screenshot.

### 4 Future Work

There are a number of features which could be added to the FlightFinder program to make it more convenient, accurate, and useful. The first two improvements pertain the accuracy of the output. First, the current flight data contains only estimated flight times. The program would be much improved by selecting an airline and querying their database for real flight information. Secondly, time for layovers should be taken into account. It would probably be beyond the scope of this program to consider actual departure and arrival times; however, a constant amount of time could be added for each layover. This would likely avoid any four-or-more airport connections that currently result from some searches.

In addition, one could implement a utility which allows the user to conveniently edit the flight data. Currently, the input file must be created manually or using a self-composed program, both of which are inconvenient methods. The Graph class already provides the ability to load and store the graph data to a file. Therefore, the new utility would simply need to provide a user interface to add, edit, and remove airports and connections. It should
also allow for editing of the current flight times. This would allow a user to more easily add to the flight map while ensuring the integrity of the data.

5 Conclusion

The group sought to create a program to determine the shortest flight route between any two airports in a flight network. The solution was implemented in C++ by creating a graph data structure through which to represent the flight network (with the airports as vertices and the available flights as edges). The group also implemented Dijkstra’s shortest path algorithm, which operates on a graph object, for determining the best flight route. The program interacts with users through a graphical user interface. This solution accomplished the groups goals, providing an efficient and flexible means by which to determine optimal flight routes.

References


Biographies

Karsten Steinhaeuser is a junior computer science major currently living in Carroll Hall. Born and raised in South-Western Germany, he still enjoys traveling the world and returning home to visit his family. In his free time he participates in a variety of sports including ice hockey (his favorite), but also inline hockey and basketball, among others.

Aaron Wenger is a junior computer engineer who lives in Fisher Hall. Originally from Mechanicsburg, Pennsylvania, Aaron enjoys playing soccer, watching sports, and spending time with his friends and family.
Appendix