Airline Hub Project Report

Chak Sin Michael Wong, Siqi Yan
Instructor: Hassan Safouhi

Introduction

The SHORT-HOP airline is planning to provide transportation service in North Central USA, between 9 different cities, include: Dayton, Fort Wayne, Green Bay, Grand Rapids, Kenosha, Marquette, Peoria, South Bent and Toledo.

In order to minimize the cost for the company, it is required to route flights through hub cities, and the number of hub cities should be as small as possible, since we want to reduce the flight distance between any two cities, so that passengers can also get a better experience.

In our project, we want to set up a flight network to minimize the number of hub cities, given the pairwise distances between the nine cities. We will be mainly using graph theory to show that when this is possible, and we will prove a theorem to solve the problem. Finally, we are going to develop three different algorithms to solve such problem with any number of cities, and demonstrate the algorithms using MATLAB on a randomly generated situation with 100 cities.

Formulation of the Problem

Let G be a graph of order 9. The adjacent matrix of G is given by:

$$A = \begin{bmatrix} 0 & 102 & 381 & 232 & 272 & 494 & 294 & 170 & 134 \\ 102 & 0 & 279 & 133 & 175 & 395 & 235 & 72 & 91 \\ 381 & 279 & 0 & 159 & 134 & 143 & 273 & 215 & 298 \\ 232 & 133 & 159 & 0 & 115 & 262 & 255 & 94 & 139 \\ 272 & 175 & 134 & 115 & 0 & 275 & 156 & 103 & 229 \\ 494 & 395 & 143 & 262 & 275 & 0 & 416 & 341 & 387 \\ 294 & 235 & 273 & 255 & 156 & 416 & 0 & 186 & 320 \\ 170 & 72 & 215 & 94 & 103 & 341 & 186 & 0 & 139 \\ 134 & 91 & 298 & 139 & 229 & 287 & 320 & 139 & 0 \end{bmatrix}$$

A hub is a vertex of degree ≥ 2.

The goal is to find a subgraph $H \subseteq G$, such that:

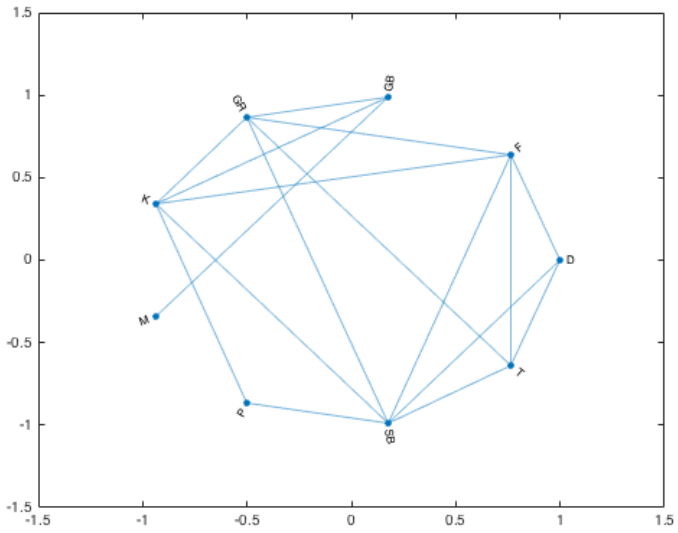
1) H is connected;

2) For all edges $e \in E(H)$, $w(e) \leq 200$;

3) The number of hubs in H is minimized.

Solution

First we need to verify that H exist. Form a graph G' by removing all edges of weight > 200 in G, so G' already satisfy condition 2), plot G' using Matlab:



Observed that G' is connected, satisfy condition 1), therefore, we confirm that H exist. We can now limit our searching for H on G'.

Theorem. The minimum number of hubs in H is 3.

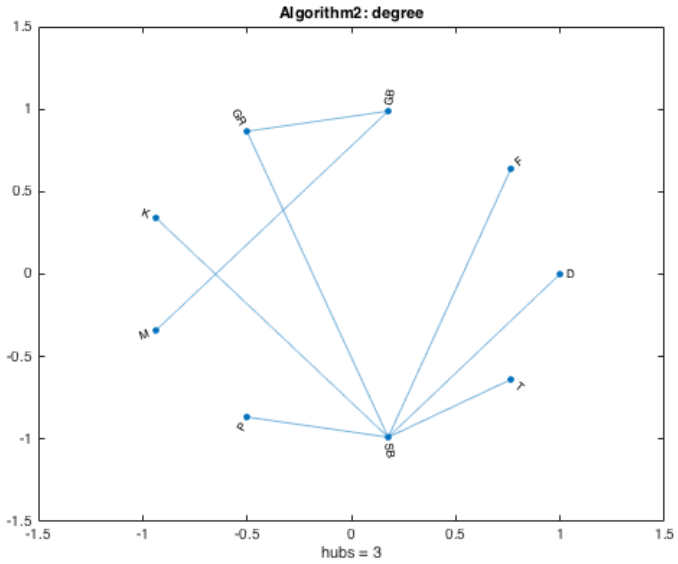
Proof.

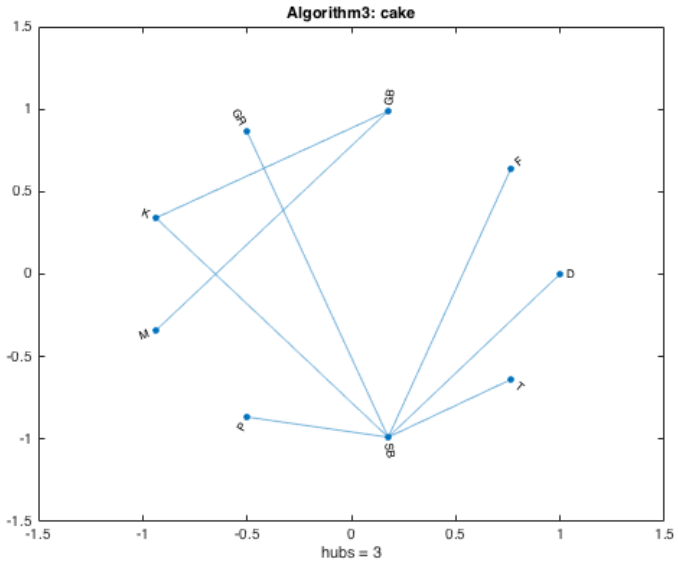
We want to start choosing hub with the highest degree, since in the best case, we only need one hub to connect all other vertices, in which case the degree of the hub is 8.

By observation, the vertex SB has the maximum degree in G', which is 6. Connect SB to all other possible vertices, there are still two vertices M, GB unconnected, therefore 1 hub is not possible.

Also by observation, vertex M is incident with only GB. So in order to make H connected, we need to make GB a hub for M. We still need one more hub to connect with GB, so that H is connected. Therefore, the minimum number of hubs is 3. ■

By the theorem above, any connected subgraph of G' with 3 hubs satisfy the condition. Because of the proof, GB can connect to either GR or K, H is not unique, and we have two solutions:





We can generalize this idea with G of any order n, by the following algorithm:

Algorithm1

1. Given a connected graph G, form a subgraph $G' \subseteq G$ by removing all edges of weight > 200;
2. If G' is connected, proceed, otherwise, stop running;
3. Let H be a graph where $V(H) = V(G')$ and $E(H) = \emptyset$;
4. Let $B = V(G')$;
5. For each vertex $v \in B$, calculate how many adjacent vertices in G' are not in the same component in H with v;
6. Choose the one vertex with the highest number, and connect it to all adjacent vertices in H while do not creating loop;
7. Remove that vertex from B;
8. Repeat 5 to 7 until H is connected.

Algorithm2: degree

1. Given a connected graph G, form a subgraph $G' \subseteq G$ by removing all edges of weight > 200;
2. If G' is connected, proceed, otherwise, stop running;
3. Let H be a graph where $V(H) = V(G')$ and $E(H) = \emptyset$;
4. Generate an ordered list L, with the degree of vertices in $V(G')$ in decreasing order;
5. For each number $d_i \in L$, find the corresponding vertex $v_i \in V(G')$, and add all incident edges of v_i to H, while do not creating loops;
6. Repeat 5 until H is connected.

Algorithm3: cake

Same as algorithm2 except in step 4, form an ordered list L by the following rule:

1. Let M be the adjacent matrix of G', let P be a matrix of the same size as P such that:
$$p_{ij} = \begin{cases} 1, if\ m_{ij} \neq 0 \\ 0, if\ m_{ij} = 0 \end{cases}$$
2. Let $u \in \mathbb{R}^n$, where $u_i = \frac{1}{degree(v_i)}$ for $v_i \in G'$;
3. Let $L = \{l_1, l_2, \dots, l_n\}$, where:
$$l_j = \sum_{i=1}^{n} p_{ij} \cdot u_i$$

Interpretation of Result

The matrix A represent the distances between cities, and the distance can be read by finding the row of one city, and then find the column of another city, the distance between these two cities is the number in the crossing point. Zero means that there is no connection between two cities.

The model treated the airline flights as graphs, where each vertex represents a city, and the distance (or the distance of a flight) between two cities is a weighted edge. A hub city is a city that route flights through it, so it represents a vertex with more than one edges. We have two conditions: 1. All cities should be 200 miles of a hub city; 2. The number of hub cities should be as small as possible. We added one more condition which is the graph H is connected, because we need to make sure that all cities are reachable by the airline flights. That way, we set up the model.

From the analysis of the model, we conclude that the minimum number of hub cities is 3, and the solution is not unique, we listed two solutions. One can use different strategies to get more solutions, however, the optimal number of hub cities should always equal to 3.

The SHORT-HOP can plan the flights with respect to the solution graphs, plan a flight between two cities only if there is edge between the corresponding vertices. Then the company will find an optimal way to set up the flights network that satisfy their assumptions.

We also developed three algorithms and implemented them in MATLAB to calculate the airline map for SHORT-HOP automatically, and they all get correct solutions with 3 hub cities (algorithm1 gives the same result as algorithm2, so we didn't show the result from algorithm1).

The idea behind the algorithms are: we construct an empty network, and we chose hub cities follow some certain order and add connections from the hub city to all its neighbor to construct the optimal solution. The reason we don't want to create loop is that we don't have to connect a same city to the network more than once. The difference between these algorithms is how to decide the order.

In algorithm 2, we choose the order simply by the degrees of vertices, that is, the city which has the most connections to neighbor cities that is less than 200 miles is added first, and the second most city gets added second, and so on.

In the algorithm 3, the order is works as follows: imaging the graph represents a social network, where each vertex is a person, and the edges connect to his/her friends. Each person has one cake of same size, and they are dividing and sharing their cakes equally to their friends. Everyone will give their cake to his/her friends, while also receive cakes from their friends. In the end, the order

is generated based on how many cakes each person have. The person with the most cakes represents a city with the highest order, and so on. In this scenario, the person who gets the highest order is either the person with the most friends, or the person who has some lonely friends, and therefore can receive some whole cakes from these friends. These kind of people are more important among the network, so they are more likely to be the hubs of the network.

The algorithm1 is slightly different, since the order does not stay the same. Each time when we choose hub cities, we compute how many "vacancies" each hub city has, that is, how many cities it can connects to, and generate the order based on the number of "vacancies". The city with the most "vacancies" be the hub city. Since the network changes as we add connection, the number of "vacancies" also changes. So we need to compute the order every time when we choose the hub city.

Critique of the Model

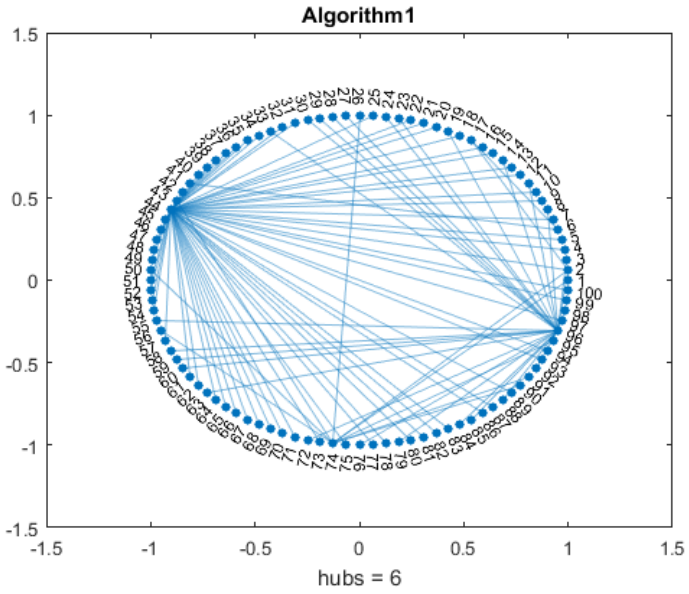
The algorithms can solve the airline hub problem, however, it does not provide a mathematically correct solution in general when solving randomly generated graphs. Specifically, they can give solutions with different number of hub cities. The reason is that the connections of graphs in general can be very complicated, and we cannot find a way to calculate the optical number of hub cities, therefore, we cannot prove that the algorithms always find the optical solutions. So in such cases, what we can do is that we can run the three algorithms on the same graph, and see which one gives the smallest number of hub cities.

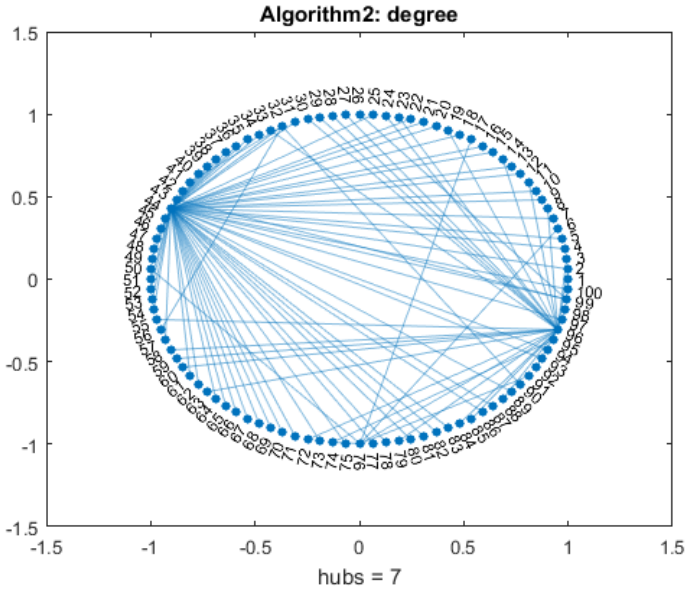
Even thought we cannot guarantee to find the optimal solution, these algorithms perform very well in our tests (one of our test is shown in appendix). We have tried running them on several randomly generated airline graphs with 100 cities, under the same condition as the original problem, and the maximum distance between cities is 400 miles. And the algorithms give the smallest numbers of hub cities ranging from 4 to 8, which we think is acceptable.

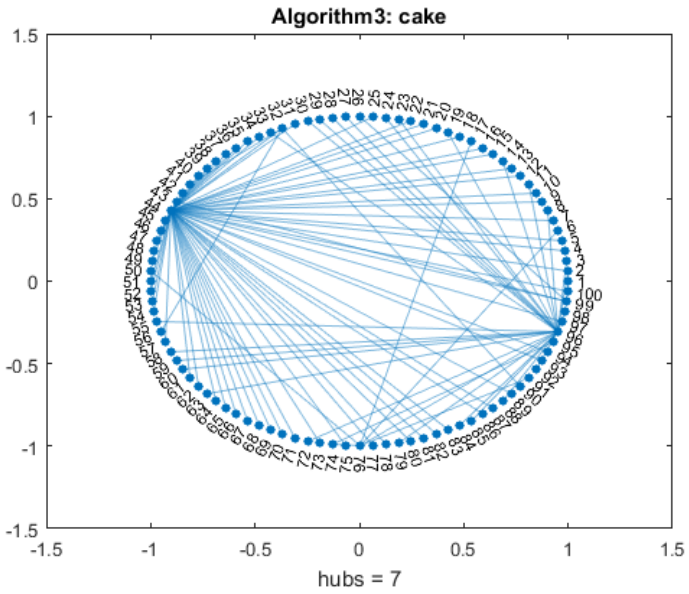
The other limitation of our model is that it does not minimize the total distance of the map. We have concluded that the solution for airline problem is not unique, and therefore there must exist one or more solutions that have the smallest total distance among all solutions, if we can find that solution, the cost for the airline company can be further minimized. But we haven't find a way to search for all solutions, therefore, we cannot find the solution with the smallest total distance.

Appendix

Test of algorithms on random graph with 100 vertices:







MATLAB Codes: (the following codes are just the main functions, and they require a lot of dependencies to run, which has been sent to the professor's email).

1. Algorithm1:

```
% This function computes a subgraph H of a given graph G, such that:
%   1. H is connected;
%   2. All edges have weight less than or equal to 200;
%   3. The number of hubs is minimized (hopefully).
% This function works as follows: for each vertex v, it computes the number of
% "vacancies" it can connects to, and then take the vertex with max number of
% vacancies, and connect all its neighbours, while do not creating loop. Then
% recalculates the "vacancies" for the remaining vertices, do this until the
% graph is connected.
%
% Parameters:
%   mG: the adjacent matrix of graph G
%
function [H num_of_hubs] = minhubs(mG)
    % Initialization
    mG = mG .* (mG <= 200);              % remove edges greater than 200
    G = graph(mG);                       % construct a graph G
    n = height(G.Nodes);                 % n = number of vertices in G
    H = graph(zeros(size(mG)));          % construct a graph H with no edges
```

```
   set = createset(G);                % used to check if two vertices are in the same
component

   % Check if G is connected after removing the long edges:
   [vlist1 vlist2] = find(adjacency(G));   % form a list of edges
   for i = 1 : length(vlist1),
      for j = 1 : length(vlist2),
         set = union_vert(set, vlist1(i), vlist2(j));
      end
   end
   assert(isconnected(set));          % assert if G is connected, if not the program will
terminate
   set = createset(G);                % re-initialize the set

   vertlist = [1 : n];                % from a list of all vertices
   while length(vertlist) > 0,

      % calculate how many "vacancies" each vertex can connects to
      count = zeros(1, length(vertlist));
      for i = 1 : length(vertlist),
         v = vertlist(i);
         neighbours = find((mG(v, :) ~= 0) .* [1 : n]);
         for j = 1 : length(neighbours),
            neighbour = neighbours(j);
            if findrep(set, v) ~= findrep(set, neighbour),
               count(i) = count(i) + 1;
            end
         end
      end

      [discard hub] = max(count);      % choose the one with the max "vacancies"
      hub = vertlist(hub);

      % add all its neighbours while do not creating loop
      neighbours = find((mG(hub, :) ~= 0) .* [1 : n]);
      for i = 1 : length(neighbours),
         neighbour = neighbours(i);
```

```
            [H set] = add_no_loop(H, set, hub, neighbour, mG(hub, neighbour));
        end

        vertlist(vertlist == hub) = [];     % delete the hub from vertlist

        if isconnected(set),
            break
        end
    end
    num_of_hubs = sum(degree(H) >= 2);
    disp(sprintf('exit: number of hubs = %d', num_of_hubs));
end
```

2. Algorithm2, 3:

```
% This function compute a subgraph H of a given graph G, such that:
%   1. H is connected;
%   2. All edges have weight less than or equal to 200;
%   3. The number of hubs is minimized (hopefully).
% And this is done by greedy algorithm, which generate a list of either degrees
% of vertices or "cakes" of vertices (determined by the second parameter). The
% function will then add edges to the graph H in the order of the list while do
% not greating loops. So after a certain number of iterations (less than the
% length of the list), the graph H will be connected and then we are done.
%
% Parameters:
%   mG: the adjacent matrix of graph G
%   option: only two options are available: 'degree' and 'cake', the last one
%       is faster in some cases.
%
function [H num_of_hubs] = minhubs_greedy(mG, option)
    % Initialization
    mG = mG .* (mG <= 200);              % remove edges greater than 200
    G = graph(mG);                       % construct a graph G
    n = height(G.Nodes);                 % n = number of vertices in G
    H = graph(zeros(size(mG)));          % construct a graph H with no edges
    set = createset(G);                  % used to check if two vertices are in the same
component
```

```
   % Check if G is connected after removing the long edges:
   [vlist1 vlist2] = find(adjacency(G));   % form a list of edges
   for i = 1 : length(vlist1),
      for j = 1 : length(vlist2),
         set = union_vert(set, vlist1(i), vlist2(j));
      end
   end
   assert(isconnected(set));              % assert if G is connected, if not the program will
terminate
   set = createset(G);                    % re-initialize the set

   hublist = get_hublist(G, mG, option);   % generate a list used by the greedy algorithm,
based on option
   for i = 1 : length(hublist),
      hub = hublist(i);                        % a hub is chosen based on the order of the hublist
      disp(sprintf('iter = %d, hub = %d', i, hub));
      neighbours = find((mG(hub, :) ~= 0) .* [1 : n]);   % get a list of the hub's neighbours
      for j = 1 : length(neighbours),
         neighbour = neighbours(j);             % try to add one of the neighbour
         [H set] = add_no_loop(H, set, hub, neighbour, mG(hub, neighbour));
      end
      if isconnected(set),          % we don't need to go over all the hublist, if the graph is
connected, we are done
         break;
      end
   end
   num_of_hubs = sum(degree(H) >= 2);
   disp(sprintf('exit: number of hubs = %d', num_of_hubs));
end
```