Faculty of Engineering
Department of Management Sciences
MSCI 332 – Deterministic Optimization Models and Methods

Term Project: Flight Scheduling Part 2

Professor Fatma Gzara

Team 3

Sumit Bansal	Rudra Bhatt	Patricia Hall	Stefanno Da Silva
20466623	20516603	20526041	20508389

Friday December 2nd 2016

Table of Contents

Abstract

This report describes the optimization of an airline routing and scheduling model. Based on the data donated by an anonymous European airline company, we are to deliver a flight and connections plan to increase their fleet efficiency under some measurable metric. The goal of this project is to develop a heuristic model that can solve any airline crew scheduling problem and result in a feasible solution.

The dataset that is currently being used contains an adjacency matrix representing all possible connections between flights, as well as base and sink flights. Base flights are any flights that can be the initial departing flight, and sink flights are any flights that can be the final arriving flight. Bases and sinks can be understood to belong to the same locations, that is, a plane that departs from a base and arrives at a sink ultimately reach its original location, thus allowing its crew member to return home. The connections were used to create a binary matrix, in which the first row and last column represent the base and sink nodes respectively. In this matrix, the columns are represented by j indexes and the rows by i indexes. The dataset also contains the arrival and departure information for each flight. With this information we created another matrix that calculates the respective layover time of each connection by subtracting the arrival time of flight i from the departure time of flight j. This matrix is used as a cost matrix, where each minute of flight layover time is considered a unit of cost. The values where there are no connections, that are 0's, are given a cost of a large number M (999999) in order to represent that flight connection as impossible. The goal is to develop a model that creates flight paths to minimize cost.

By using the data outlined above, a heuristic model has been developed. The point of creating a heuristic model is so that the steps outlined can be used to solve any complex crew scheduling model. The heuristic model is developed through an iterative process to improve the solution each time, until an optimal solution can be reached.

Mathematical Model

We are given data about the arrival and departure of 173 flights in which each flight is either able to leave from the base or return to the sink. An adjacency matrix providing all possible connections between flights is given to show which flights can happen after another. The final mathematical model that is created is a heuristic, and uses a variant of the Hungarian method to solve the problem. Before coming to the solution by using the Hungarian method, multiple heuristic models were considered. Interesting results are found by implementing different types of heuristics through iterations. The Greedy algorithm, Tabu search, and the application Hungarian method are all studied; it is found that the Hungarian method provides the best possible heuristic solution.

The goal of the project is to produce a heuristic solution that respects the constraints that only connecting flights can be on a path, and to minimize the cost of flights. Each model explores the problem uniquely.

Heuristic Model Exploration

We implemented three heuristic models to solve the flight and crew scheduling problem. These are the Greedy Algorithm, Priority List and Tabu Search, and the Hungarian method.

The Greedy Algorithm

The first heuristic that was developed builds off of the Greedy algorithm. The greedy method is a procedure that builds by finding local optimal in a dataset and uses this to expand to find the global optimal if possible. The Greedy Algorithm was set to follow the steps outlined in Appendix III to solve the problem, however, the heuristic did not create a feasible solution for our data set. Each time the model was run, not all 173 flights were scheduled and there was nothing in place to ensure that all flights were scheduled.

The Greedy Tabu Search Algorithm

The original greedy algorithm that was implemented was not able to schedule every flight in the given adjacency matrix. In order to prevent the algorithm from not scheduling a flight, a priority list was created to track which flights had not yet been visited in the previous iterations. All missing flights are either added to the bottom of the priority list, or if they already exist on it, are moved one index up the list. Thus the list is updated at each iteration to take into account the flight missed by the previous iterations, while keeping all flight missed in all iteration so far. When looking for a possible connection from flight i to flight j the algorithm now looks for prioritized flights first, then for shortest layover time. The result was still unable to schedule all flights; thus more tweaking was made in order to create a feasible solution.

To improve the solution, we modified the search algorithm. The priority list is organized by flight number, that is, flights that happen first are prioritized first within the priority list itself. The big change between the algorithms is that flights in the priority list must be assigned to a route unless absolutely impossible. While still checking if connections to a flight in the priority list is possible, it also verifies if the current flight happens after a flight in the priority list. If yes, and the flight in the priority list was not visited, we return to the previously connected flight and check over once more. To make sure we visit as many prioritized nodes as possible in each route we set a "checkpoint" system, in which we have several sub-routes to each checkpoint, and that sub-route can also have a checkpoint. For example, at the start of the iteration we set the first flight of the route as the current "checkpoint" and try to reach a flight in the priority list from that checkpoint. If it is possible to reach that flight, it will be found and a new checkpoint will be set, from where we will attempt to reach the next item if the item is reachable. A recursive function called *NextVertexSearch* was utilized to navigate up and down the connection nodes. Please view Appendix III for a better understanding of the algorithm, and Appendix II for the source code utilized. The steps of the final Tabu search algorithm are outlined in Appendix III.

Hungarian Method Variant, The Pairing and Diving Heuristic

After being unsuccessful in designing a heuristic that would schedule one route at a time, we took a different approach and designed a 3-part heuristic that first determined the optimal pairs of flights (minimal ground time), appended the pairs to each other and then divided up the long chains of flights into smaller, more realistic flight paths for flight crews. We utilized *munkres 1.0.8,* a free Python package extension that implements the Hungarian method given a quadratic matrix. The steps of the Hungarian method are outlined in Appendix II with and example. Since non connecting flights are assigned the value of 999999, the Hungarian algorithm is forced to avoid them. The first row, containing the base to flights connections has all of its values set to 999999, so we will ignore that row for now, focusing only on finding all possible connections instead. While lowering the cost, some flights will be unable to connect, these being flights connected to the sink, thus they will randomly be assigned an impossible flight. This is fine, and will be taken care on the next step. The large costs assigned to impossible flight connections prevents the algorithm from scheduling these flight connections. For a brief review of the Hungarian algorithm, please view Appendix II.

The next step in this heuristic was matching and appending the optimal pairs with each other, creating routes. The output given by the Hungarian algorithm implemented consisted of flight tuples in the following format: (1,9), (2,12), (3,10), etc. For each pair, in ascending order of the first item on each pair the following algorithm was created:

1. Verify there exists a route that contains the first flight in the tuple. If yes, append the second flight of the tuple to the existing item inside its route. Repeat this step for the next item. Else, proceed to step 2.
2. If the flight the algorithm is appending next does not exist in any of the existing routes, create a new route. Add both items in the tuple to the route, with the item of the first index of the tuple forming the source of the route and the item to the left directly connected to the first item. Go back to step 1 if more tuples are available. If not end algorithm and move into the next step.

The final step in this heuristic was to divide the long routes into shorter sub-routes that represent crew schedules. It is inhumane to schedule crews for lengthy shifts hence there is a need to change this crew while ensuring that the initial crew is returned to their place of origin. For each of the routes created in the previous steps follow the following procedure:

1. Set a minimum amount of consecutive flights, so that crews can rotate. This minimum accounts for the fact that the algorithm attempts to divide flights as much a possible, but won't always be able to. Start at the first flight to sink that allows for 3 consecutive prior flights. For example, in a route containing 10 flights, go to flight 3.
2. Verify that the current flight is connected to a sink, that is, that it can be a final flight. Verify then that the next flight to it is connected to a source, that is, the flight can be the first flight of a route.
3. If both the two conditions above are met set the current flight as a sink and cut the route at the current flight, from the source flight to the new sink flight. Add the second part of the route, now a new route by itself, to the collection of other routes and start over to the next available route. If the conditions are not met, move to the next flight and go back to step 2.

Model Complexity

This model takes into account the complexity of the problem through the constraints that are created. The base constraint and the sink constraints have been created to ensure that only the flights that can leave from the base and enter the sink do so. This allows crews to returned to the original location they departed from. It is also designed to minimize the amount of layover time between each connecting flight because it is inefficient and costly for planes sit idle on the ground. Finally, this model will work with any airline scheduling problem. The model can take into account multiple variables and large, feasible datasets, allowing any airline's flight and crew scheduling data (in the form of an adjacency matrix) to be modelled finding feasible flight paths and crew schedules.

Comparison of Part 1 and Part 2

The model in part 1 was implemented using GAMS on NEOS server. For the first part, we built our model around an objective function to minimize cost (layover time), while following the constraints that all flight must be connected to either another flight on a sink/source flight, as well as assuring that the number of flights leaving from a source equals the number of flights reaching a sink. The heuristic models created in part 2 were coded in C# and Python. While heuristics, they still aim to follow the same objectives and constraints set in part 1. The final Tabu Search algorithms ensured all flights to be connected with the use of a priority list and by using checkpoints to ensure that as many priority flight are added into one route. By starting at the source and finishing at a sink we ensure that all routes are complete. And finally, we try to get as close as possible to the local minima with the use of the greedy algorithm for connection assignment. On the second model, the Hungarian variant, we first create connection to minimize cost while penalizing impossible connections in a way they cannot happen unless unavoidable (If either a flight cannot be connected to any others, that flight being connected to the sink, or if the solution itself is infeasible). After all connection are formed we form routes, each with a clear origin and end, thus obeying the constraint set by the original mode.

Results

The Greedy Tabu Search algorithm iterated 1001 times passing several flights through a priority list, however, after 1001 iterations, it was still unsuccessful in finding a feasible solution without excluding at least one flight from it. The resultant output of this model can be seen in Appendix V. Flight 40 is the most frequent flight missing from the solution set. Instead, using the Hungarian method, another heuristic was developed resulting in a feasible solution consisting of 11 routes; which were then divided into 27 sub routes. What this means is that all 173 flights were scheduled using 11 airplanes and 27 flight crews. The solution also ensures that the flight crew returns to their place of origin. The flights each plane completes with the number of crew changes can be seen graphically in Appendix I. The code output for this model can also be seen in Appendix VI.

Conclusion

The heuristic models that are studied throughout the report represent iterations that the group went through to find a feasible solution. The models studied aim to solve a major issue involving airline scheduling. First, we came up with a simple heuristic model building off the Greedy algorithm and created airline paths by combining flights occurring closest to one another. Realizing that this model is not strong or feasible, we added a unique priority list to it and implemented Tabu search. This model was strong, but did not give us a feasible solution either. After considering the previous two techniques, we built a model from the concepts of the Hungarian method as well as a combination rule to find a feasible solution. This solution was feasible and also enhanced the airline flight and crew scheduling problem by providing routes that each plane can take and breaking these routes down into sub-routes that can each be possibly be accomplished by one flight crew.
This solution scheduled all flights using 11 airplanes and 27 flight crews while aiming to minimize the total layover time for all flights and ensuring the flight crew returns to their place of origin.

References

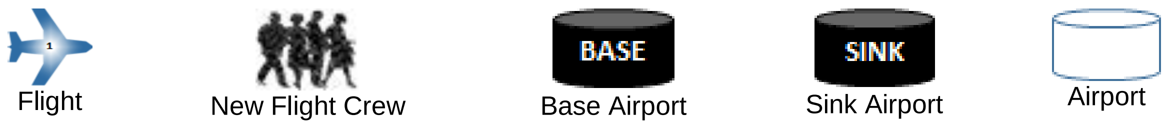
- The assignment problem and the Hungarian method (no date) Available at: http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf (Accessed: 2 December 2016).

- Publications, U.T. (no date) Crew assignment problem. Available at: http://www.universalteacherpublications.com/univ/ebooks/or/Ch6/crewass.htm (Accessed: 2 December 2016).

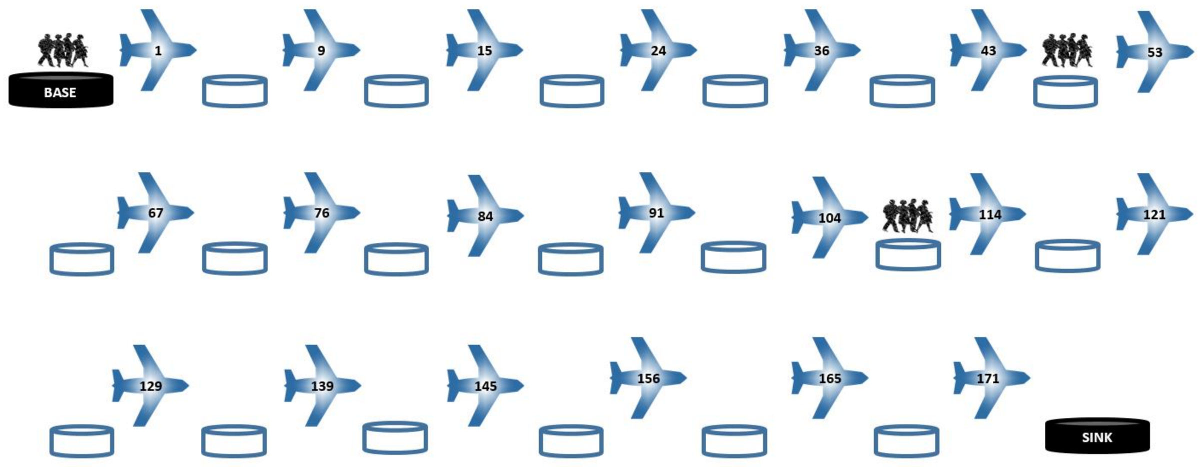
- Zhu, X. (no date) Advanced search hill climbing, simulated annealing, genetic algorithm. Available at: http://pages.cs.wisc.edu/~jerryzhu/cs540/handouts/hillclimbing.pdf (Accessed: 2 December 2016).

Appendix I – Solution Sets

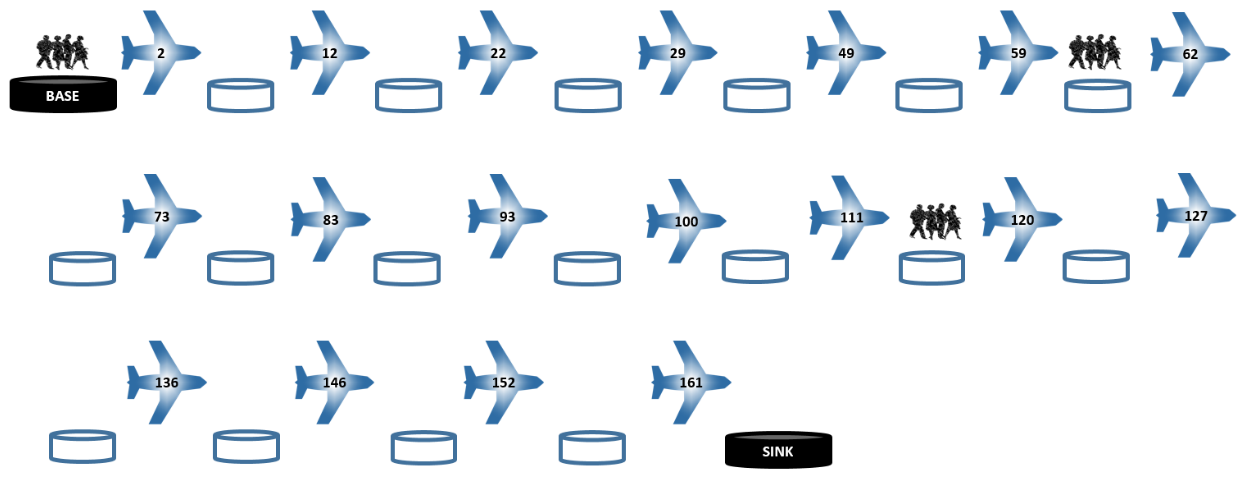
The solution set below consists of the tour each airplane takes and the number of crews needed for the airplane to run starting from the “Base” and ending at the “Sink”.



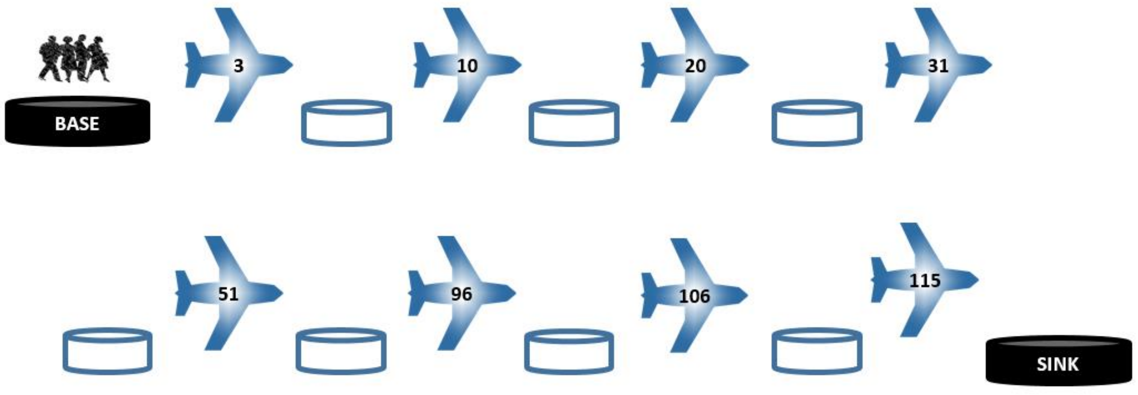
Airplane 1:
20 flights
3 flight crews



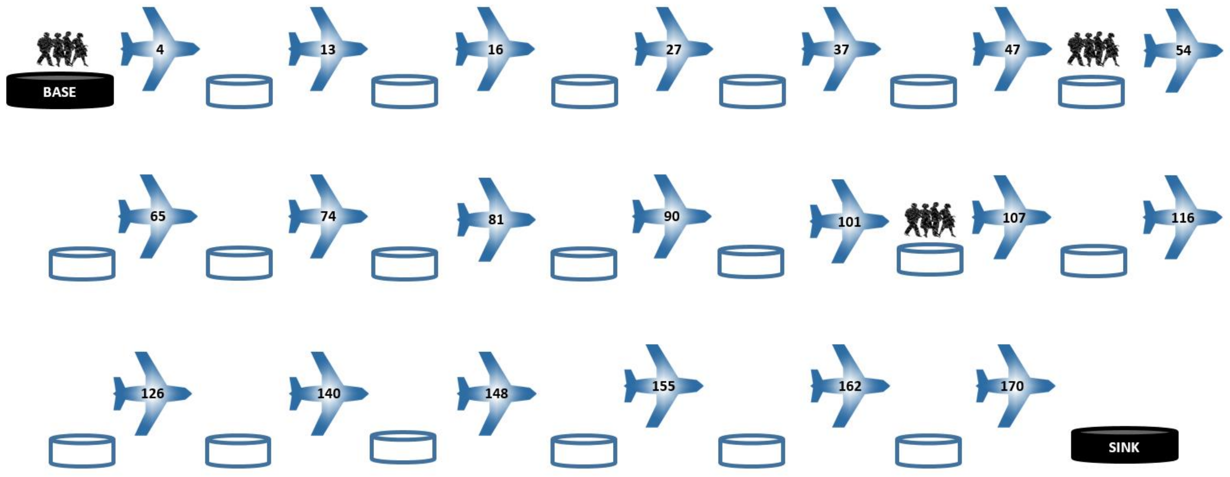
Airplane 2:
18 flights
3 flight crews



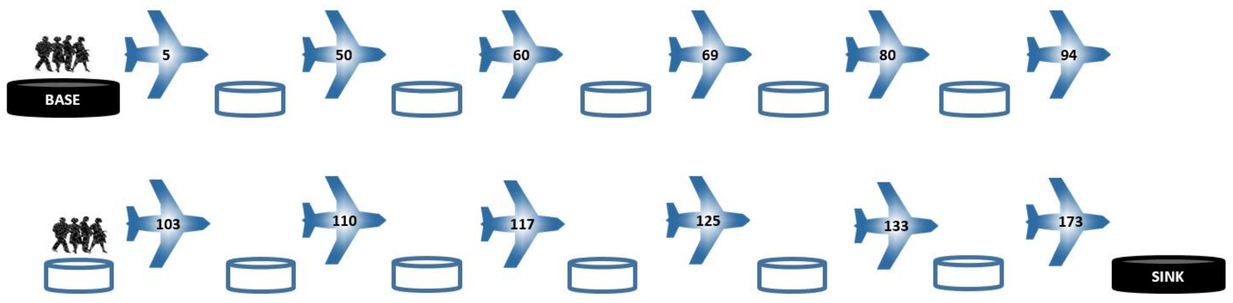
Airplane 3:
8 flights
1 flight crew



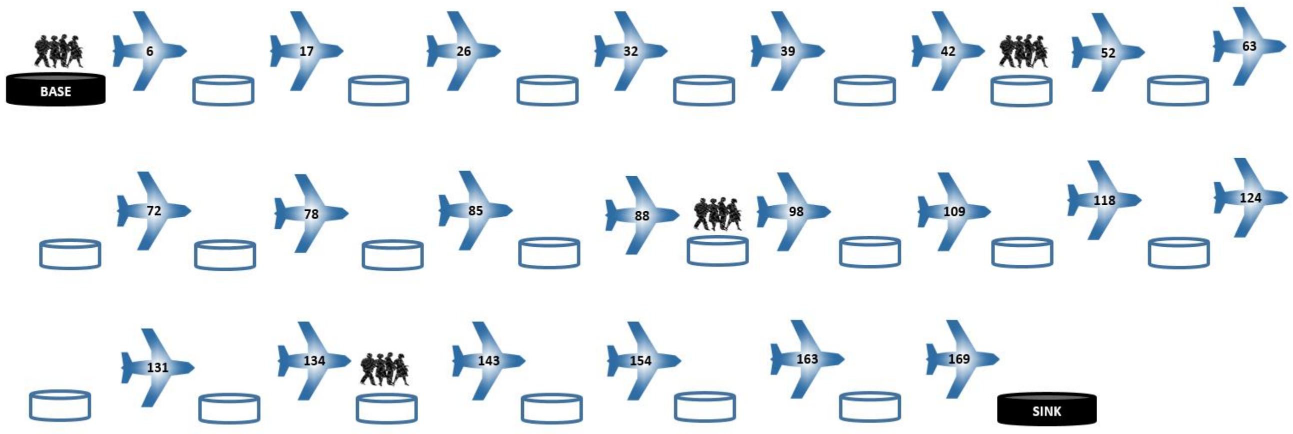
Airplane 4:
20 flights
3 flight crews



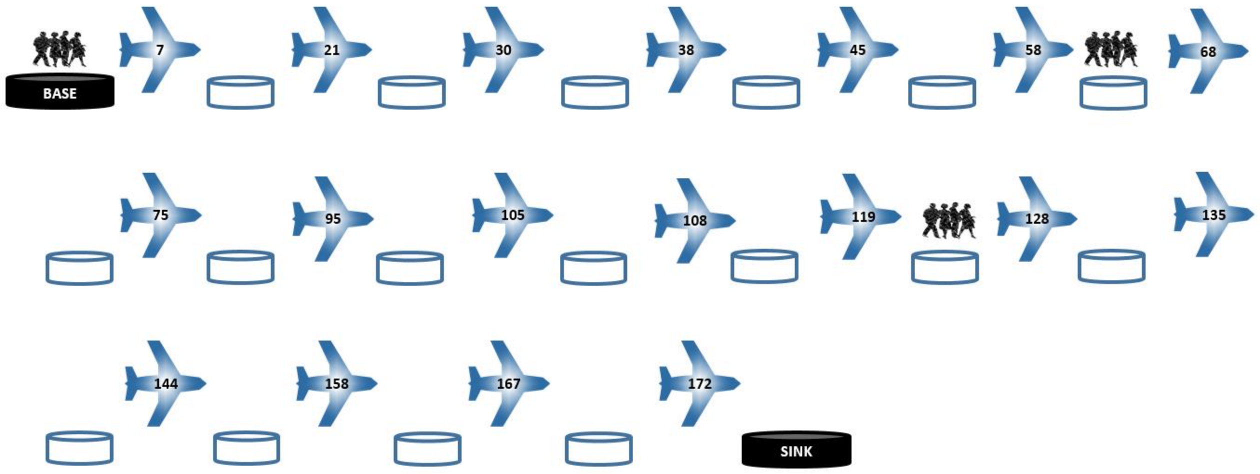
Airplane 5:
12 flights
2 flight crews



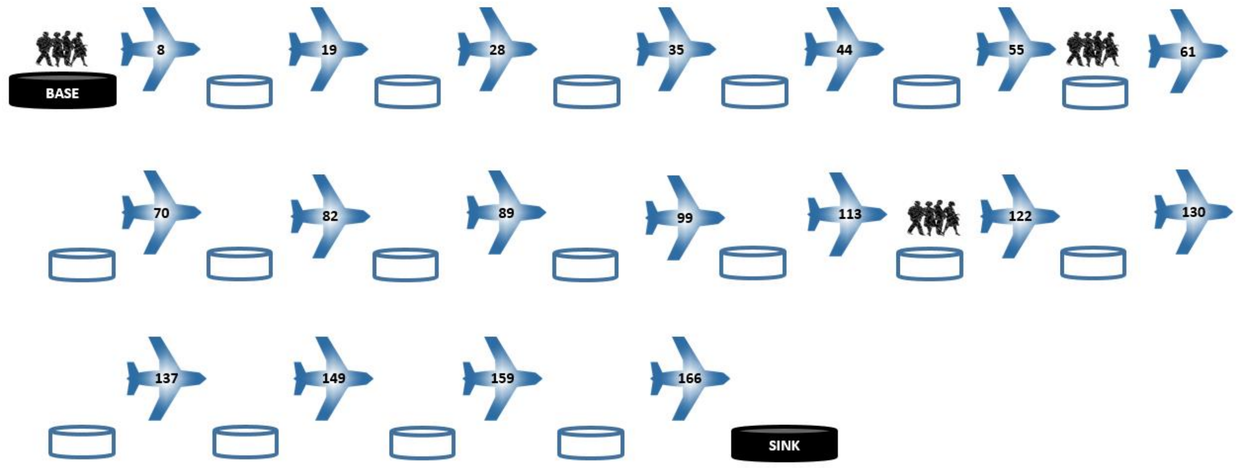
Airplane 6:
22 flights
4 flight crews



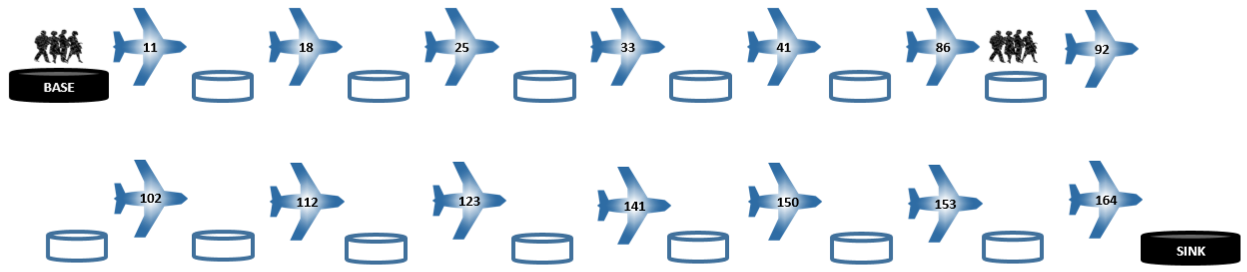
Airplane 7:
28 flights
3 flight crews



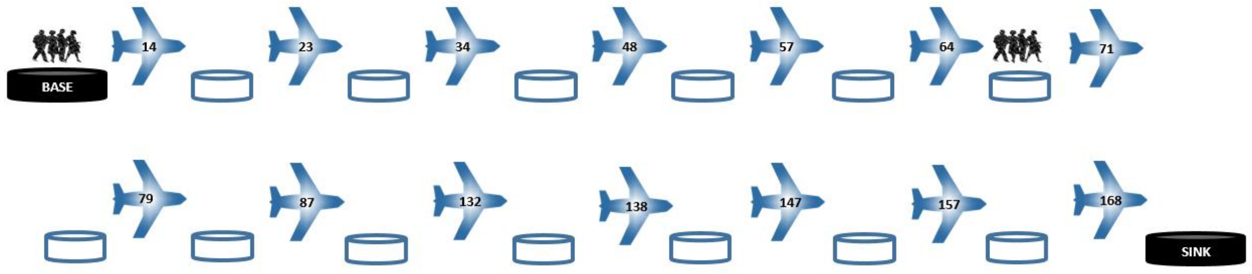
Airplane 8:
18 flights
3 flight crews



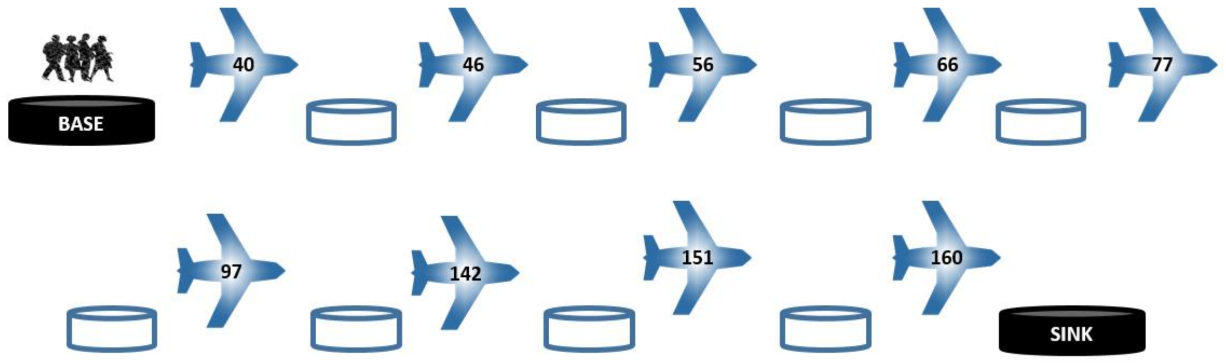
Airplane 9:
14 flights
2 flight crews



Airplane 10:
14 flights
2 flight crews



Airplane 11:
9 flights
1 flight crews



Appendix II: Hungarian Method Variant

Hungarian Method Steps

1. Substitute adjacency matrix 0's with 99999's.
2. Subtract the shortest layover time in each row from all the layover times of its row.
3. Subtract the shortest layover time in each column from all the layover times of its columns
4. Draw lines covering all 0's while ensuring minimum number of lines are used.
5. Find the shortest layover time that is not covered by a line. Subtract this time from all layover times that are not lined out and add it to times that are lined out twice.
6. Repeat steps 4 and 5 till the minimum number of lines equals the number of rows in matrix.
7. The first 0 that has less than two lines drawn on it in each row is the connection chosen between flights that creates a feasible solution forming flight pairs.

Hungarian Variant Example

- Assume a 10x10 matrix of flight connections
- Connected to Sources: 1,3,4,6 (Sources are represented by 0s)
- Connected to Sinks: 2,5,7,8 (Sinks are represented by 9s)

Original Matrix*:
[0,1,0,1,1,0,1,0,0,0],
[0,0,45,0,60,0,0,0,0,0],
[0,0,0,0,35,45,0,0,0,150],
[0,0,0,0,20,0,0,70,75,0],
[0,0,0,0,0,10,0,25,0,0],
[0,0,0,0,0,0,0,0,0,200],
[0,0,0,0,0,0,0,30,45,0],
[0,0,0,0,0,0,0,0,0,40],
[0,0,0,0,0,0,0,0,0,35],
[0,0,0,0,0,0,0,0,0,0]
* From all rows except the first and last, a zero in
the last column represents a source, a sink otherwise.

Part 1: Setup matrix for the Hungarian Variant Model
Hungarian Matrix:
[99999,99999,99999,99999,99999,99999,99999,99999,99999,99999],
[99999,99999,45,99999,60,99999,99999,99999,99999,99999],
[99999,99999,99999,99999,35,45,99999,99999,99999,150],
[99999,99999,99999,99999,20,99999,99999,70,75,99999],
[99999,99999,99999,99999,99999,10,99999,25,99999,99999],
[99999,99999,99999,99999,99999,99999,99999,99999,99999,200],
[99999,99999,99999,99999,99999,99999,99999,30,45,99999],
[99999,99999,99999,99999,99999,99999,99999,99999,99999,40],
[99999,99999,99999,99999,99999,99999,99999,99999,99999,35],
[99999,99999,99999,99999,99999,99999,99999,99999,99999,99999]

Part 2: Solve matrix applying Hungarian.
Connected Nodes:
(1, 2) -> 45
(2, 5) -> 45
(3, 4) -> 20
(4, 7) -> 25
(5, 3) -> 99999**
(6, 8) -> 45
(7, 6) -> 99999**
(8, 9) -> 35
(9, 1) -> 99999**
**Nodes connected to sinks may connect to 9, or if 9 is taken they will be connected to a lower, impossible connection. This represents that the node will connect to a sink, and is equivalent to a 9.

Part 3: Connect nodes to generate routes, ignore length of routes.
Routes:
[0, 1, 2, 5, 9]
[0, 3, 4, 7, 9]
[0, 6, 8, 9]

Part 4: For long routes divide into shorter sub-routes if possible.

Appendix III: Prior Heuristic Model Steps

The Greedy Algorithm Steps

In terms of the crew scheduling problem being studied, the Greedy Algorithm is set to follow the steps outlined below. Each possible connection posses 2 attributes: "Length of Layover" (In minutes) and "Visited" (True of False).
Steps to the First Iteration Heuristic:

1. Start at flight i = 1, and find the first flight it connects to (Connection with the lowest "Length of Layover" and "Visited" is set to False) -> set that as flight j and set "Visited" to True for the connection between flight i and flight j.

2. Set that flight j as the next scheduled flight i (flight i = flight j). Record previous flight.

3. Find the first flight that connects to flight i (flight with the lowest ground time) -> set that as flight j and the set "Visited" to True for the connection between flight i and flight j.

4. Set that flight j as the new flight i (flight i = flight j). Record previous flight.

5. Repeat steps 3, 4 until no more connecting flights to flight i.

6. Check if current flight i can only connect to sink. If yes then the route start and ends at base and this route is possible. Add all flight in the route to the list of successful flights. Go back to step 1 and repeat.

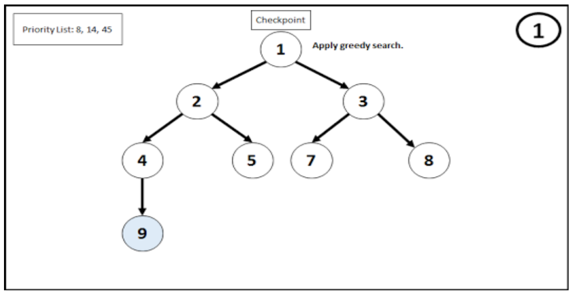
7. If flight i is not connected to a sink retreat to the previously recorded flight. Set the flight before as flight i. Refer back to step 6.

8. Stop when list of successful flights is complete. contains all flights in the heuristic is complete.

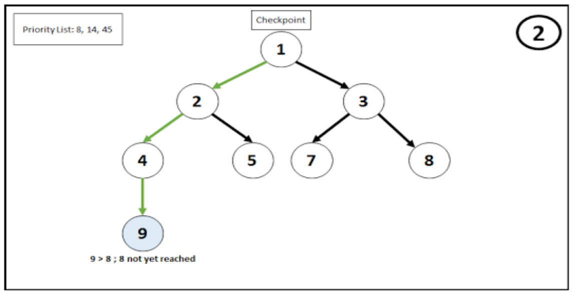
The steps for the heuristic above are simple and build a model that creates paths with the shortest amount of layover time between flights.

The Greedy Tabu Search Steps

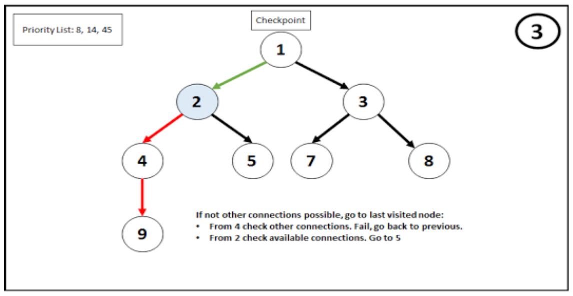
(Example for 1 checkpoint and 8 as current priority)



Apply greedy search.

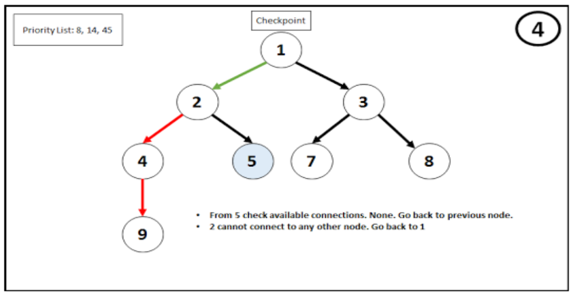


9 > 8 ; 8 not yet reached

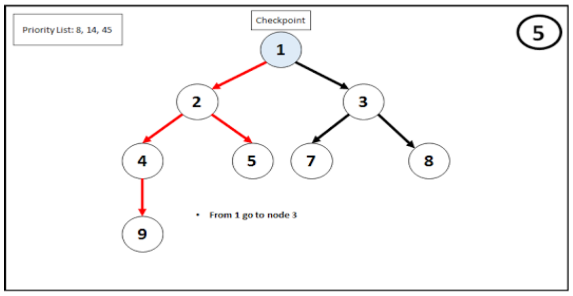


If not other connections possible, go to last visited node:

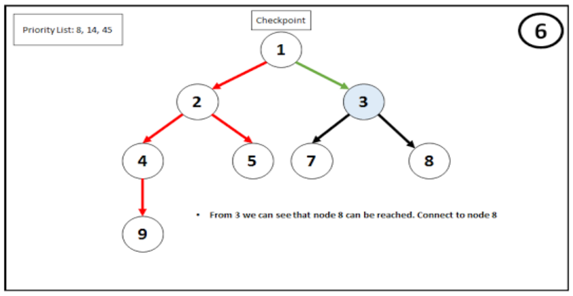
- From 4 check other connections. Fail, go back to previous.
- From 2 check available connections. Go to 5



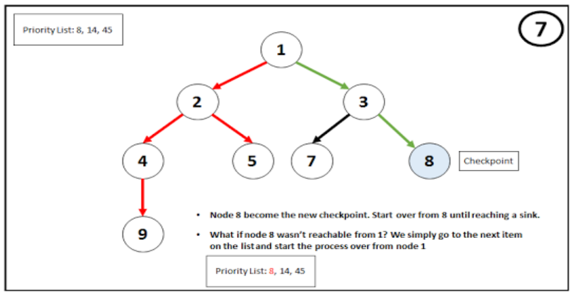
- From 5 check available connections. None. Go back to previous node.
- 2 cannot connect to any other node. Go back to 1



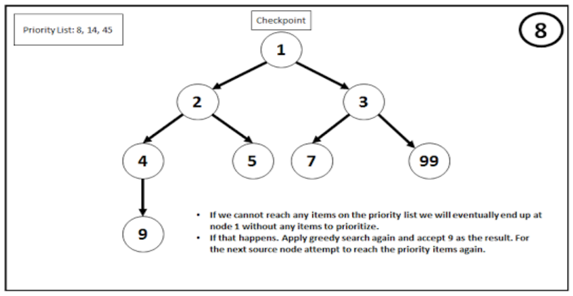
- From 1 go to node 3



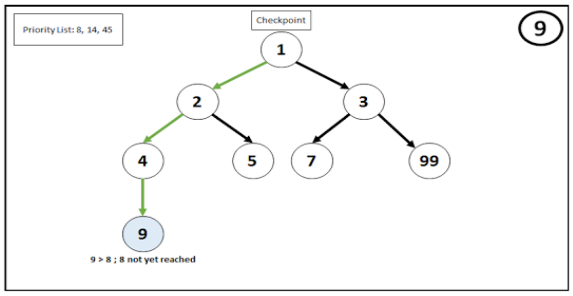
- From 3 we can see that node 8 can be reached. Connect to node 8



- Node 8 become the new checkpoint. Start over from 8 until reaching a sink.
- What if node 8 wasn't reachable from 1? We simply go to the next item on the list and start the process over from node 1



- If we cannot reach any items on the priority list we will eventually end up at node 1 without any items to prioritize.
- If that happens. Apply greedy search again and accept 9 as the result. For the next source node attempt to reach the priority items again.



9 > 8 ; 8 not yet reached

1. Starting a flight i = 0. Record it as previous flight. Set checkpoint.

2. Select a flight j to connect based on both priority and layover time. If a connection to that flight is possible (Visited = False) and the flight is currently on top of the priority list choose that flight. Otherwise choose the first that allows the plane to depart from the current location first.

3. Set connection between flight i and flight j as visited = True. Set that flight j as the next scheduled flight i (flight i = flight j). Record previous flight.

4.

    a. Verify if the current flight happens after the flight on top of the priority. If yes, it means we "missed" to add a priority flight to the route. If the previously visited flight was a checkpoint remove the first item from the priority list. Set flight i as the previously visited flight in the connection and repeat steps 2-3. If the previously visited flight was a checkpoint remove the first item from the priority list.

    b. If the current flight if the current item on top of the priority list, remove it from the list and set a new checkpoint at this flight.

5. Check if current flight i can only connect to sink. If yes, then the route start and ends at base and this route is possible. Add all flights in the route to the list of successful flights. Return to flight. If flight cannot connect to any others nor a sink you reached a dead end. Return to previously connected flight and repeat steps 2-3.

6. After all possible routes were found, check for any missing flights. Add these flights to the priority list. Reiterate. Stop if no missing flight or iteration = 1001

Appendix IV: Source Codes

Greedy Tabu Search (C#)

```
class Vertex
{
    public int flight_no;
    public int value;
    public bool visited = false;
    public bool checkpoint = false;
}
class MetaHeuristic
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamReader arr_read = new StreamReader("flight_arrival_time.txt");
            Dictionary<int, int> arrivals = new Dictionary<int, int>();
            string arr = arr_read.ReadLine().Trim();
            int count = 0;
            while (arr != null)
            {
                arrivals.Add(count, int.Parse(arr));
                arr = arr_read.ReadLine();
                ++count;
            }
            arr_read.Close();


            StreamReader dep_read = new StreamReader("flight_departure_time.txt");
            Dictionary<int, int> departures = new Dictionary<int, int>();
            string dep = dep_read.ReadLine().Trim();
            count = 0;
            while (dep != null)
            {
                departures.Add(count, int.Parse(dep));
                dep = dep_read.ReadLine();
                ++count;
            }
            dep_read.Close();


            StreamReader matrix = new StreamReader("adjecency_matrix.txt");
            string[] cells = null;
            string row = matrix.ReadLine();
            int i = 0;
            List<Vertex> connections_j = new List<Vertex>();
            Dictionary<int, List<Vertex>> connections_i = new Dictionary<int,
List<Vertex>>();
            while (row != null)
            {
                connections_j = new List<Vertex>();
                cells = row.Split(' ');
                for (int j = 0; j < cells.Length - 1; ++j)
                {
                    if (i == 0 || j == 0)
```

```
                        {
                            connections_j.Add(new Vertex { flight_no = j, value =
int.Parse(cells[j]) });
                        }
                        else
                        {
                            if (j == 174)
                            {
                                connections_j.Add(new Vertex { flight_no = j, value =
int.Parse(cells[j]) * (arrivals[i]) });
                            }
                            else
                            {
                                connections_j.Add(new Vertex { flight_no = j, value =
int.Parse(cells[j]) * (departures[j] - arrivals[i]) });
                            }
                        }
                    }
                    connections_j = connections_j.OrderBy(v => v.value).ThenBy(f =>
f.flight_no).ToList();
                    connections_i.Add(i, connections_j);
                    ++i;
                    row = matrix.ReadLine();
                }


                List<int> flights = new List<int>();
                foreach (KeyValuePair<int, List<Vertex>> flight in connections_i)
                {
                    if (flight.Key != 0 && flight.Key != 174)
                    {
                        flights.Add(flight.Key);
                    }
                }


                StreamWriter sw = new StreamWriter("results.txt");
                int Vertexs_count = 0;
                int iteration = 0;


                List<int> priority_list = new List<int>();
                while (iteration < 1001)
                {
                    ++iteration;
                    Vertexs_count = 0;
                    sw.WriteLine("Iteration " + iteration);
                    sw.Write("Priority List: ");
                    foreach (int item in priority_list)
                    {
                        sw.Write(item + ", ");
                    }
                    sw.WriteLine();
                    List<int> visited_Vertexs = new List<int>();
                    for (int k = 0; k < connections_i[0].Count; ++k)
                    {
                        if (connections_i[0][k].visited == false && connections_i[0][k].value >
0)
                        {
                            connections_i[0][k].visited = true;
                            Stack<int> path = null;
                            List<int> tmp_priority_list = new List<int>(priority_list);
                            for (int q = 0; q < tmp_priority_list.Count; ++q)
                            {
                                if (visited_Vertexs.Contains(tmp_priority_list[q]))
                                {
                                    tmp_priority_list.RemoveAt(q);
                                    --q;
                                }
                            }
                            for (int p = 0; p <= priority_list.Count(); ++p)
                            {
                                if (p == priority_list.Count())
                                {
                                    path = NextVertexSearch(connections_i, 
 connections_i[0][k].flight_no, new List<int>());
                                }
                                else
                                {
                                    path = NextVertexSearch(connections_i, 
 connections_i[0][k].flight_no, tmp_priority_list);
                                    if (path == null && tmp_priority_list.Count > 0)
                                    {
                                        tmp_priority_list.RemoveAt(0);
                                    }
                                    else
                                    {
                                        break;
                                    }
                                }
                            }

                            List<int> visited_columns = new List<int>();
                            if (path != null)
                            {
                                while (path.Count > 0)
                                {
                                    int Vertex = path.Pop();
                                    ++Vertexs_count;
                                    sw.Write(Vertex + "->");
                                    visited_Vertexs.Add(Vertex);
                                    visited_columns.Add(Vertex);
                                }
                            }
                            sw.WriteLine();
                            for (int l = 0; l < connections_i.Count; ++l)
                            {
                                for (int m = 1; m < connections_i[l].Count; ++m)
                                {
                                    if 
 (visited_columns.Contains(connections_i[l][m].flight_no))
                                    {
                                        connections_i[l][m].visited = true;
                                    }
                                }
                            }
                        }
                    }
                    sw.WriteLine("Count:" + Vertexs_count);
                    List<int> missing_flights = flights.Except(visited_Vertexs).ToList();
```

```
                sw.Write("Missing flight: ");
                foreach(int item in missing_flights)
                {
                    sw.Write(item+", ");
                }
                sw.WriteLine();
                if (missing_flights.Count > 0)
                {
                    for (int l = 0; l < connections_i.Count; ++l)
                    {
                        for (int m = 0; m < connections_i[l].Count; ++m)
                        {
                            connections_i[l][m].visited = false;
                        }
                    }
                    foreach (int item in missing_flights)
                    {
                        if (!priority_list.Contains(item))
                        {
                            priority_list.Add(item);
                        }
                    }
                    priority_list.Sort();
                }
                else
                {
                    break;
                }
            }
            sw.Close();
            Console.WriteLine("DONE");
        }


        public static List<Vertex> Shuffle(List<Vertex> list)
        {
            var rnd = new Random();
            int n = list.Count;
            while (n > 1)
            {
                n--;
                int k = rnd.Next(n + 1);
                Vertex value = list[k];
                list[k] = list[n];
                list[n] = value;
            }
            return list;
        }


        public static Stack<int> NextVertexSearch(Dictionary<int, List<Vertex>> matrix, int 
flight, List<int> priority_list)
        {
            if (priority_list.Count > 0 && flight > priority_list[0] && 
matrix[flight].First(f => f.flight_no == priority_list[0]).visited == false)
            {
                return null;
            }
```

```
                if (priority_list.Count > 0 && matrix[flight].First(f => f.flight_no ==
priority_list[0]).value > 0 && matrix[flight].First(f => f.flight_no ==
priority_list[0]).visited == false)
                {
                    matrix[flight].First(f => f.flight_no == priority_list[0]).visited = true;
                    int checkpoint;
                    checkpoint = priority_list[0];
                    priority_list.RemoveAt(0);
                    Stack<int> path;
                    while (true)
                    {
                        path = NextVertexSearch(matrix, checkpoint, priority_list);
                        if (path == null || path.Count <= 1)
                        {
                            if (priority_list.Count == 0)
                            {
                                break;
                            }
                            priority_list.RemoveAt(0);
                            continue;
                        }
                        else
                        {
                            path.Push(flight);
                            return path;
                        }
                    }
                    return NextVertexSearch(matrix, checkpoint, priority_list);


                }
                matrix[flight] = matrix[flight].OrderBy(v => v.value).ThenBy(f =>
f.flight_no).ToList();
                for (int j = 0; j < matrix[flight].Count; ++j)
                {
                    if (flight == 2)
                    {
                        //Check
                    }
                    if (matrix[flight][j].visited == false && matrix[flight][j].value > 0)
                    {
                        matrix[flight][j].visited = true;
                        if (matrix[flight][j].flight_no == matrix[flight].Count - 1)
                        {
                            // Sink found
                            Stack<int> path = new Stack<int>();
                            path.Push(flight);
                            return path;
                        }
                        else
                        {
                            Stack<int> path = NextVertexSearch(matrix,
matrix[flight][j].flight_no, priority_list);
                            if (path == null)
                            {
                                matrix[flight][j].visited = false;
                                continue;
                            }
                            else if (priority_list.Count > 0)
                            {
```

```
                            if (!path.Contains(priority_list[0]))
                            {
                                path = null;
                            }
                        }
                        else
                        {
                            path.Push(flight);
                            return path;
                        }
                    }
                }
            }
            return null;
        }
    }
}
```

Hungarian Method Variant (Python Script)

```
from munkres import Munkres, print_matrix


arrivals = open('flight_arrival_time.txt', 'r')
flight_arrivals = {}
j = 0
for line in arrivals:
   arr = line.split()
   flight_arrivals[j] = int(float(arr[0]))
   j += 1
arrivals.close()
j = 0
departures = open('flight_departure_time.txt', 'r')
flight_departures = {}
i = 0
for line in departures:
   dept = line.split()
   flight_departures[i] = int(float(dept[0]))
   i += 1
departures.close()
i = 0


read_matrix = open('connections.txt', 'r')
connection_graph = []
base = []
for line in read_matrix:
   line = line.replace("\n", "")
   connections = list(map(int, line.split(',')))
   vertices = []
   if i == 0:
           base = connections
   for j in range(0, len(connections)):
       if float(int(connections[j])) == 0 or i == 0:
           vertices.append(999999)
       else:
           vertices.append(abs(flight_departures[j] - flight_arrivals[i]))
   connection_graph.append(vertices)
   i += 1


m = Munkres()
paths = []
indexes = m.compute(connection_graph)
print_matrix(connection_graph, msg='Lowest cost through this matrix:')
total = 0
for row, column in indexes:
   value = connection_graph[row][column]
   total += value
   print('(%d, %d) -> %d' % (row, column, value))
   new_path = True
   for path in paths:
       if row in path:
           path.append(column)
           new_path = False
           continue
   if new_path:
       path = [row, column]
       paths.append(path)


connected_flights = 0
for path in paths:
   if path[-2] != 174:
       path[-1] = 174
   else:
       del path[-1]
   if path[0] != 0:
       print(path)
       connected_flights += len(path) - 1


print(connected_flights)
print('total cost: %d' % total)


minimun_len = 3
for route, path in enumerate(paths):
       n = 0
       for index, item in enumerate(path):
           if item == 174:
               break
           if n >= minimun_len and base[item] == 0 and base[path[index + 1]] == 1:
               sub_path = path[n+1:]
               if len(sub_path) >= minimun_len:
                   copy_path = path[:n+1]
                   copy_path.append(174)
                   paths[route] = copy_path
                   paths.append(sub_path)
               break
           else:
               n += 1


for path in paths:
   source = 0
   path.insert(0, 0)
   print(path)
print('Done')
```

Appendix V: The Greedy Tabu Search Model Output

Iteration 1
Priority List:

```
1->9->15->24->34->48->57->64->71->79->87->94->103->110->117->125->133->140->148->155->162->170->
2->12->20->31->36->43->52->63->72->78->85->88->98->109->118->124->131->134->143->154->163->169->
3->10->22->29->37->47->53->67->74->81->90->101->107->116->126->146->152->161->
4->13->16->27->39->42->54->65->76->84->91->104->112->123->128->135->144->158->165->171->
5->50->60->69->80->132->136->173->
6->17->26->32->41->55->61->70->82->89->99->113->120->127->137->149->157->168->
7->21->25->33->44->86->92->102->114->121->129->139->145->156->167->172->
8->19->28->35->45->58->66->77->83->93->100->111->122->130->138->147->159->166->
11->18->30->38->46->56->68->75->95->105->108->119->141->150->153->164->
14->23->49->59->62->73->97->142->151->160->
51->96->106->115->
Missing flight: 40
```

Iteration 2
Priority List: 40

```
1->40->44->55->61->70->80->94->103->110->117->125->133->140->148->155->162->170->
2->12->20->31->34->48->57->64->71->79->87->101->107->116->126->146->152->161->
3->10->22->29->36->43->52->63->72->78->85->88->98->109->118->124->131->134->143->154->163->169->
4->13->15->24->37->47->53->67->74->81->90->132->136->173->
5->50->54->65->76->84->91->104->112->123->128->135->144->158->165->171->
6->17->26->32->39->42->60->69->82->89->99->113->120->127->137->149->157->168->
7->21->25->33->41->86->92->102->114->121->129->139->145->156->167->172->
8->19->28->35->45->58->66->77->83->93->100->111->122->130->138->147->159->166->
11->18->30->38->46->56->68->75->95->105->108->119->141->150->153->164->
14->23->49->59->62->73->97->142->151->160->
16->27->51->96->106->115->

Missing flight: 9
```

Iteration 3
Priority List: 9, 40

1->9->15->40->44->55->61->70->80->94->103->110->117->125->133->140->148->155->162->170->
2->12->20->31->34->48->57->64->71->79->87->101->107->116->126->146->152->161->
3->10->22->29->36->43->52->63->72->78->85->88->98->109->118->124->131->134->143->154->163->169->
4->13->16->27->37->47->53->67->74->81->90->132->136->173->
5->50->54->65->76->84->91->104->112->123->128->135->144->158->165->171->
6->17->26->32->39->42->60->69->82->89->99->113->120->127->137->149->157->168->
7->21->25->33->41->86->92->102->114->121->129->139->145->156->167->172->
8->19->28->35->45->58->66->77->83->93->100->111->122->130->138->147->159->166->
11->18->30->38->46->56->68->75->95->105->108->119->141->150->153->164->
14->23->49->59->62->73->97->142->151->160->
51->96->106->115->

Missing flight: 24

.
.
.
.

Iteration 1001
Priority List: 3, 9, 10, 12, 13, 17, 18, 19, 21, 22, 23, 24, 26, 27, 29, 30, 31, 32, 33, 35, 38, 39, 40, 42, 49, 60, 62, 68, 69, 72, 91, 95, 100, 104, 105, 106, 108, 111, 112, 119, 120, 131, 151

1->9->22->29->39->42->60->69->91->104->112->123->131->134->151->160->
2->12->26->32->49->59->62->73->95->105->108->119->126->140->148->155->162->170->
3->10->30->38->68->75->100->111->120->127->136->146->152->161->
4->13->15->24->72->78->106->115->128->135->143->154->163->169->
5->50->52->63->74->81->90->101->107->116->129->139->144->158->165->171->
6->17->28->35->44->55->61->70->80->94->103->110->117->125->133->173->
7->21->25->33->41->48->57->64->71->79->87->132->137->149->157->168->
8->19->34->86->92->102->114->121->138->147->159->166->
11->18->36->43->53->67->76->84->97->142->145->156->167->172->
14->23->37->47->54->65->82->89->98->109->118->124->141->150->153->164->
16->27->45->58->66->77->83->93->99->113->122->130->
20->31->46->56->85->88->
51->96->

Missing flight: 40

Appendix VI: The Hungarian Method Variant Model Output

Routes Formed:

```
[1, 9, 15, 24, 36, 43, 53, 67, 76, 84, 91, 104, 114, 121, 129, 139, 145, 156, 165, 171, 174]
[2, 12, 22, 29, 49, 59, 62, 73, 83, 93, 100, 111, 120, 127, 136, 146, 152, 161, 174]
[3, 10, 20, 31, 51, 96, 106, 115, 174]
[4, 13, 16, 27, 37, 47, 54, 65, 74, 81, 90, 101, 107, 116, 126, 140, 148, 155, 162, 170, 174]
[5, 50, 60, 69, 80, 94, 103, 110, 117, 125, 133, 173, 174]
[6, 17, 26, 32, 39, 42, 52, 63, 72, 78, 85, 88, 98, 109, 118, 124, 131, 134, 143, 154, 163, 169,
174]
[7, 21, 30, 38, 45, 58, 68, 75, 95, 105, 108, 119, 128, 135, 144, 158, 167, 172, 174]
[8, 19, 28, 35, 44, 55, 61, 70, 82, 89, 99, 113, 122, 130, 137, 149, 159, 166, 174]
[11, 18, 25, 33, 41, 86, 92, 102, 112, 123, 141, 150, 153, 164, 174]
[14, 23, 34, 48, 57, 64, 71, 79, 87, 132, 138, 147, 157, 168, 174]
[40, 46, 56, 66, 77, 97, 142, 151, 160, 174]
total cost: 12035998
```

Sub-Routes Formed by Dividing Routes:

```
[0, 0, 174]
[0, 1, 9, 15, 24, 174]
[0, 2, 12, 22, 29, 174]
[0, 3, 10, 20, 31, 174]
[0, 4, 13, 16, 27, 174]
[0, 5, 50, 60, 69, 174]
[0, 6, 17, 26, 32, 174]
[0, 7, 21, 30, 38, 174]
[0, 8, 19, 28, 35, 174]
[0, 11, 18, 25, 33, 174]
[0, 14, 23, 34, 48, 174]
[0, 40, 46, 56, 66, 77, 174]
[0, 36, 43, 53, 67, 174]
[0, 49, 59, 62, 73, 174]
[0, 51, 96, 106, 115, 174]
[0, 37, 47, 54, 65, 174]
[0, 80, 94, 103, 110, 174]
[0, 39, 42, 52, 63, 174]
[0, 45, 58, 68, 75, 174]
[0, 44, 55, 61, 70, 174]
[0, 41, 86, 92, 102, 174]
[0, 57, 64, 71, 79, 174]
```

```
[0, 97, 142, 151, 160, 174]
[0, 76, 84, 91, 104, 174]
[0, 83, 93, 100, 111, 174]
[0, 74, 81, 90, 101, 174]
[0, 117, 125, 133, 173, 174]
[0, 72, 78, 85, 88, 174]
[0, 95, 105, 108, 119, 174]
[0, 82, 89, 99, 113, 174]
[0, 112, 123, 141, 150, 174]
[0, 87, 132, 138, 147, 174]
[0, 114, 121, 129, 139, 174]
[0, 120, 127, 136, 146, 174]
[0, 107, 116, 126, 140, 174]
[0, 98, 109, 118, 124, 174]
[0, 128, 135, 144, 158, 174]
[0, 122, 130, 137, 149, 174]
[0, 153, 164, 174]
[0, 157, 168, 174]
[0, 145, 156, 165, 171, 174]
[0, 152, 161, 174]
[0, 148, 155, 162, 170, 174]
[0, 131, 134, 143, 154, 174]
[0, 167, 172, 174]
[0, 159, 166, 174]
[0, 163, 169, 174]
Done
```