EVOLVING EFFICIENT AIRLINE SCHEDULES

by

Damon Marcus Lewis

S.B. Computer Science Engineering
Massachusetts Institute of Technology, 1999

Submitted to the Department of Electrical Engineering and Computer Science in
Partial Fulfillment of the Requirements for the Degree Of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

September 2000

©2000 Damon Marcus Lewis. All rights reserved.

The author hereby grants to MIT permission to reproduce
and to distribute publicly paper and electronic
copies of this thesis document in whole or in part.

Signature of Author:

Department of Electrical Engineering and Computer Science
July 24, 2000

Certified by:

Patrick Henry Winston
Professor of Computer Science
Thesis Supervisor

Accepted by:

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
E VOLVING E FFICIENT A IRLINE S CHEduLES

by

Damon Marcus Lewis

S.B. Computer Science Engineering
Massachusetts Institute of Technology, 1999

Submitted to the Department of Electrical Engineering and Computer Science in
Partial Fulfilment of the Requirements for the Degree Of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

September 2000

Abstract

This project is to solve the fleet scheduling problem for a large airline. This requires matching each flight in a schedule to a plane in the fleet that will fly the scheduled flight on a particular day. This thesis shows how to use the network flow model of a flight schedule in order to find the smallest fleet required to fly a schedule without dropping any flights. This is done by pruning the whole network to a smaller set of more relevant arcs.

It is also a goal of the project to perform these tasks without requiring a large powerful computer or workstation.
## Table of Contents

Acknowledgments ............................................................................. 4

The Fleet Scheduling Problem ............................................................... 6

Finding the Minimal Fleet Schedule ....................................................... 9

Overview of the Preflow-Push Algorithm ............................................... 18

The Network Model ............................................................................ 27

The Schedule Model ............................................................................ 32

Results .................................................................................................. 39

Contributions ....................................................................................... 58

Bibliography ......................................................................................... 60
Acknowledgments

This paper would not be possible without the help of others. Advisors, mentors, and friends all helped me immensely in putting this work together.

First, I would like to thank my thesis advisor, Patrick Henry Winston. Prof. Winston came to my aid when as a senior at MIT, and recent admit to the Masters of Engineering program, I was searching for a thesis project in the field of Artificial Intelligence. I was expecting him to listen to my interests and direct me to another professor who may be working on a similar project. Instead he offered to oversee my project himself, and also offered me a Teaching Assistantship for his class. This support allowed me to remain at MIT for the term.

I would also like to thank Philip Brou of Ascent Technology. Mr. Brou worked closely with me and shared information about real world airline activities. He provided valuable information necessary in order to make this project applicable to the real world.

Besides the more academic support offered by the aforementioned, I was also given plenty of attaboys, pats-on-back, and go-do-your-homeworks from
family and friends. So, I would like to give shoutouts\(^1\) to Mom, Dad, Grandma, Lauren, Phil the Mad Hindu, Ashwini, Ted, Andrew, Damian, Sara, and The Man.

I would also like to thank the family that granted a bed to sleep on, food to eat, and ample distraction\(^2\) while I finished up my research. To my godsister Donna, Allen, Marie, Edward, and Matthew, thank you.

\(^1\) A shoutout is a formal acknowledgement most often used in minority communities to their parents, siblings, peeps, homeys, gees, and other friends and acquaintances to whom praise is worthy.

\(^2\) Said distraction came in many forms including babysitting, comic relief from 3 year old Matthew, and a coyote that visited the back yard.
The Fleet Scheduling Problem

An important part of any airline operation is to take marketing analysis and turn it into a fleet schedule. A marketing department has the task of mining through gigabytes of data on past reservations and fares to determine trends. These trends are interpreted and flights are scheduled to meet the demands of the flying public.

This thesis deals with the next step. After it is determined what flights are to be flown with which equipment (type of plane), the fleet of airplanes must be dispatched to fly the schedule. Each flight in the schedule for a particular time period must be matched up with a plane in the fleet to fly it. For large airlines, this is a very large task, and finding a good routing for each plane in the fleet so that the fleet schedule as a whole is efficient is not trivial. This means that planes are scheduled so that each time a plane lands completing one scheduled flight, it is in position to fly another scheduled flight without deadheads\(^3\). It is also a large task to schedule good utilization of the planes in the fleet. This incurs keeping planes flying revenue flights while still leaving time for loading/unloading and regular scheduled maintenance.

\[^3\text{A deadheading flight is a non-revenue flight used to position a plane for a later flight.}\]
In order to find the best fleet scheduling, I have set goals of assigning all flights in the schedule with the fewest number of jets required. From hence I refer to this as the minimal fleet schedule. This would be done under the constraints that each flight has a minimum turn\(^4\) time that cannot be violated, and there are no deadheads. Another goal was to disallow what the airline industry calls a *broken flight*. A broken flight is a trip where two consecutive flights with the same flight number and equipment are not flown by the same plane.

This problem can be solved in polynomial time as I will show later. This is because the goal is to find the least number of planes required, and not the best usage of the least number of planes. Adding in requirements that the planes in the fleet scheduling have the most balanced utilization for instance causes the problem to become NP-complete. This project accepts any solution that provably requires the fewest number of planes.

It is also important that the program runs in an acceptable amount of time even for large schedules. It should not be a requirement to have an expensive, more powerful computer in order to perform the fleet scheduling task. Ideally the program will run well on an “off-the-shelf” computer.

\(^4\)A turn is simply the time a plane spends on the ground after completing a flight preparing for the next.
I chose to write the program to assign the fleet using Java. I made this decision because it has a well defined library of built in functions and data structures, and it is Object Oriented. This choice makes extending the functionality of the program easier. The built in data structures like red-black trees and resizeable vectors made implementation of the algorithms and optimization of performance troublespots much more straightforward. I used the free IBM Visual Age for Java to design and debug the system. The testbed I used was a Sony Pentium III 500 laptop with 64 Mb of RAM. I chose this system because I felt it was powerful enough to perform well, but was still well within the range of off-the-shelf, mass produced computers.
Finding the Minimal Fleet Schedule

There were many different methods that I looked at to find the smallest number of planes necessary to fly the given schedule. One solution considered a basic constrained search matching the fleet to specific flights.

To find the solution using this method, we must set up a tree. The nodes of

*Possible ways to assign each flight given previous assignments (m)*

*Figure 1: constrained search*
the tree are the individual flights that need to be assigned to a plane, and the set of
nodes at any particular depth all represent the same flight to be assigned. Arcs are
drawn from each node to the next level of nodes. An arc represents selecting a
particular plane for the preceding flight. There would be as many arcs as there are
planes available to fly that flight. A basic depth first search would be performed
through the tree. At each new node found in the tree by the search, a check
algorithm must be performed to determine if the current solution is feasible. If it is,
then continue down the tree; if it is not, then backtrack and pick a different
solution for the most recent choice. If all of the choices at a particular level lead to
unfeasible solutions, then backtrack up further to the level above this exhausted
one. If the search reaches the bottom of the tree and the final choice still yields a
consistent feasible solution, then the algorithm will report the successful schedule.

This solution would require starting with a minimal, unfeasible fleet, and
would slowly increase the fleet until the first feasible fleet is found. This solution
did not appear to be very efficient for a large real world airline schedule because it
requires many $O(b^m)$ constrained searches over the schedule for each equipment
type. The parameter $b$ refers to the number of flights that need to be assigned, and
the parameter $m$ refers to the number of planes that are available to fly an
individual leg. A typical major airline will have 2000 to 5000 flights daily, and will
have a fleet numbering between 300 and 750 planes. It becomes apparent that
simply scheduling one day is a daunting task because this search would have to be
performed hundreds of times (slowly introducing planes into the available fleet) in order to find the smallest schedule for one day alone. Creating a schedule for a week, or a month using this method does not appear to be a feasible solution.

Another solution is use a network flow model. Network Flow is a well studied field and many algorithms have been developed to work on network problems. A network in this case is a graph with nodes and arcs between the nodes. The arcs are synonymous with pipes or cables that have some capacity or bandwidth and potentially some cost to use as well. The nodes are synonymous with simple connectors to allow the contents of one pipe to progress to one or more other pipes. When we speak of network flow, we are talking synonymously to the flow of a liquid through the pipes and connectors represented by the arcs and nodes. This flow would be inherently bounded by the varying capacities and costs incurred by individual arcs.

To begin solving the minimal fleet problem using network flows, one must convert the problem into a network model. This is a two step process. First, we make a ‘feasibility’ network. This is a directed acyclic graph (DAG) to represent flights and the possible successor flights from any particular flight that can be flown with the same plane. In this network, we create nodes that represent each flight in the schedule. Arcs are drawn from node x to node y when a single plane can perform the flight represented by node x, and then continue to the flight
represented by node y. This means that flight x arrives before flight y departs, and flight x arrives at the same location that flight y departs from. With this definition, all flight pairs that meet this criteria have their representative nodes attached by a directed arc. Figures 2 and 3 depict two feasibility graphs. Feasibility graph A has 5 flights, in which flights 3 and 4 can be performed by the same plane as flight 1, flight 4 can be performed by the same plane as flight 2, et cetera. By inspection, graph A shows that at least 2 planes are required to perform the schedule, while graph B requires 3 planes.

After scanning through the schedule and creating this feasibility network, we must convert this network into one that we can place proper constraints on the flow. To do this, we take all of the flight nodes in the network and split them into two nodes. These nodes represent the origination and termination of the flight. We then place a directed arc from the origination node to the termination node. The triplet of an origination node, a termination node and the arc between them now represent the flight in the new network. If a flight node in the original network had
outgoing arcs to other nodes, these arcs are retained and run from the termination node of the first flight to the origination node of the second flight. Figure 4 shows the result of this action on feasibility graph B.
Figure 4 Splitting the flight nodes; origination nodes are dark, termination nodes are light.

Figure 5 Adding the source node, and supporting arcs

Figure 6 Adding the sink node, and supporting arcs
To complete the network, two new nodes are created. These are the source and sink nodes. The source node represents an infinite supply of planes to fly the schedule, and the sink is synonymous with a hangar where planes are taken when they are taken out of service for maintenance and such. To attach the source and sink to the network, arcs are drawn from the source to all origination nodes in the network and from all termination nodes to sink. This step is shown in Figures 5 and 6.

At this point, we are almost ready to find the minimum fleet. The algorithms we have at our disposal work with flows along the network. In order to prepare the network for the algorithms we must place values of capacity and minimum to constrain the network. We apply the constraints to all of the arcs in the network. Arcs from the source or to the sink have a minimum flow of 0 and a capacity (maximum flow) of 1. Arcs between the origination and termination nodes of a particular flight are constrained to 1 unit at all times. Finally, arcs between different flights have a minimum flow of 0 and a maximum of 1. Now the network is completely set up.

A single unit of flow from the source through the network to the sink is representative of a single plane entering the fleet, flying a set of flights, and then returning to storage. Finding the minimum amount of flow allowed by the network is synonymous with finding the minimal fleet for the schedule that the network
represents. The best of the algorithms to find the minimum flow these networks is the Preflow-Push algorithm which in its basic form runs in $O(n^3)$ time. The one problem with the Preflow-Push algorithm is that it finds the maximum flow, not the minimal. However, we can still find the minimal flow by finding any feasible flow, and then finding the maximal flow from the sink to the source. That is, find a flow that does not violate any of the constraints set, and then push back as much flow as possible switching the roles of the source and the sink. The result after finding the maximal flow from sink to source is that we have the minimal flow from source to sink. This technique for finding the minimal flow is based on an exam question in an MIT Network Optimization class.

By analyzing the minimal flow found on the network we can determine the number of planes required to fly the schedule. This is the total flow along the arcs into the sink. We can also directly determine plane routings by analyzing the flow. Figure 7 shows a possible minimum flow. The triples on the arcs, $(l,x,u)$ represent the lower flow bound, current flow, and upper flow bound on each arc. All arcs from the source or to the sink or between flights have a lower bound of 0, and upper bound of 1 and a variable flow signifying whether or not an arc is selected to show a plane performing that service or not. For instance, a plane entering service to fly a particular flight would be represented by the arc pointing to that flight having a triple $(0,1,1)$. A triple of $(0,0,1)$ shows that there is no connection in a

---

5MIT 6.855/15.082J - Network Optimization, Prof. Andreas Schulz
line of flight between the two flights at the endpoints of the arcs, or entering or leaving service via the source or sink respectively. The flow implies that three planes are necessary: One to fly flights 1 and 3, one for 2 and 4, and one for 5 and 6.

Figure 7 A possible minimum flow. Darkened arcs show 1 unit of flow, dashed lines show no units of flow.
Overview of the Preflow-Push Algorithm

The Preflow-Push algorithm finds a maximum flow through a network by potentially overloading the network, and then cleaning up the result until no constraints are violated. This is a two step process. The first stage, called preprocess, sets up the network for pushing flow. It first labels each node's distance from the sink, and the source node's label is the number of nodes in the network. This will be used later for choosing which arc to push flow along if there are multiple choices. This can be done using a simple Breadth First Search from the sink node. Next, the preprocess places as much flow as possible along all the arcs leaving the source. This creates an excess at all of the nodes reachable from the source. This excess will be corrected in the next stage.

The second stage of the algorithm loops for as long as there are active nodes to analyze. An active node is a node with excess, simply more flow coming into the node than there is flow exiting. The algorithm picks an active node and determines the amount of excess at this node. We want the flow to head towards the sink, so we try to send excess along the arc that is closest to the sink. If an arc is admissible from this node, that is it has capacity for additional flow and its end node is closer to the sink than the start node, then excess from this active node is pushed along this arc to the destination node. If the destination node is not the
sink, then it is now active because it has excess. If all of the excess was removed from the start node, then it is labeled inactive. At this point the algorithm picks another active node and repeats. Trouble happens if an active node is selected and has no admissible arcs emanating from it. In this situation, the node is relabeled to be one more than the lowest labeled node reachable from this active node. (For the purpose of the algorithm, we say that a node is reachable if a nonsaturated arc connects that node with this one.) Now that the node is relabeled, the algorithm iterates. Should the same active node be picked again, it will be allowed to send flow along any arc that reaches a lower labeled node because in terms of usable arcs, the node’s distance to the sink is the new label. The algorithm is now complete, although more explanation is necessary to show how this algorithm actually does remove all excess from the network. Because the source node is labeled higher than any other node in the network, flow will always be sent towards the sink until no more flow to the sink is feasible. At this point, flow must be sent back towards the source to remove excess. This is handled without changing the algorithm further. Because the labels are updated when no admissible arc is found, eventually, if there is remaining excess that cannot be sent to the sink without violating constraints, the label will be raised above the label of the source. When this occurs, because flow is always pushed to lower labeled nodes, flow will be pushed back to the source. The source is an infinite receptacle of flow, and so all remaining excess will be removed. At this point, with no active nodes remaining
to examine, the algorithm terminates, and a maximal flow has been determined.
Note that the solution found is not necessarily unique.

The version of the algorithm that I used for the project is slightly modified from this basic description. It allows both lower and upper bound constrained arcs in the network by searching out all arcs with lower bounds not equal to zero in the preprocess stage. If it finds one, it presets the flow on that arc to be at least the lower bound, subtracts that much flow from the start node of the arc, and adds that much excess to the end node of the arc. During the push stage, care is taken to ensure that in addition to never overloading an arc, no push ever removes flow such that the lower bound is not met.

An example run of the Preflow-Push algorithm is shown by Figure 8. This is a general case, not the specific network that will be created in the project. The triples above each arc are the same as before: lower bound, current flow, and upper bound. The $e$ and $d$ variables next to each node represent the node’s excess and distance label respectively. Four nodes are connected as seen in graph A. Node 1 is a source, and node 4 is a sink. We start by labeling each node’s distance from the sink. (B) The source node is labeled 4 because there are 4 nodes in the network. We then flood the arcs leaving the source with as much flow as they allow. This leaves excess at nodes 2 and 3. These nodes are marked active. Lower bounded arcs are also flooded to meet the constraint. (C) We arbitrarily pick an active node
to push flow from. In this example, we pick node 2. Node 2 has an excess of 4, and an admissible arc is found 2-4 because the arc is not saturated, and node 2 has a higher label than node 4. We are able to push all 4 units along the arc, so, node 2 is no longer active. (D) We again pick an active node, this time node 3. An admissible arc is found 3-4 so we push as much as we can along that arc. This time, we are not able to push the entire excess as arc 3-4 is saturated after a push of 1 unit. Therefore, node 3 remains active. At the next iteration, we see that there are no admissible arcs from node 3 because the only arc leading to a lower labeled node from 3 is saturated. Therefore, the algorithm raises the label to one more than the lowest of the reachable nodes, node 2. (E) Now, an admissible arc is found, and the remaining excess is pushed from 3 to 2. At this point, node 2 again has excess and becomes an active node. However, at distance label 1, there are no admissible arcs so, the label is updated to 3. (Not shown.) On the next iteration, one unit of flow is pushed back from 2 to 3 leaving 2 inactive, but leaving 3 with excess. The next iteration finds node 3 with no admissible arcs, and relabels it 4. The next iteration pushes the flow back to node 2, causing it to become active, and node 3 becomes inactive. At this point there are no admissible arcs, so node 2 is relabeled again as distance 5. (F) Now, there are two choices of arcs to push back flow along. We arbitrarily pick the arc 1-2 and push 1 unit of excess back to the source. This leaves node 2 inactive. The algorithm does not flag source or sink nodes as active, so with no active nodes remaining to look at, the algorithm terminates. The maximum network flow is left on the arcs.
This algorithm only requires slight modification to become a minimum flow algorithm. To do this, we run the Preflow-Push algorithm and find the maximum flow. (We actually need only find a feasible flow however, but because the maximum flow is by definition feasible, and the running time differs by a constant, I chose to use the same algorithm here.) Once a feasible flow is found, the idea is to push back as much flow as possible to the source.

To do this, I altered stage one of the Preflow-Push algorithm to label the distances to the source rather than the sink. I then skip the step of flooding the arcs from the source entirely. With the new distance labels in place, we can use the
exact same stage two as used before to push back the maximum amount of flow to
the source. Because the algorithm tries to push as much flow as possible to lower

Figure 8 Sample run of Preflow-Push algorithm
labeled nodes and the source now has a label of zero, this algorithm will correctly push back all flow possible as long as care is still taken to ensure that no arc’s lower bounds are violated. When this algorithm terminates, the minimum flow is what remains.
The version of the Preflow-Push algorithm that I used to find the minimal flow differs slightly from the version explained in this section. I used a version called Excess Scaling Preflow-Push. The difference between the generic version I described and the Excess Scaling variation is that the latter will not place more than a set amount of excess at any single node. This value is a parameter $\Delta$ that is set to be the least power of 2 greater than the highest capacity of any arc in the network. Active nodes are nodes with excess greater than or equal to $\Delta$. Again, any push of flow that occurs will be limited so that the excess at the new node does not exceed $\Delta$. When all active nodes have released excess to other nodes and
have reduced their excess to below $\Delta$, the value of $\Delta$ is halved, and the process begins anew with all nodes with excess greater than $\Delta$ becoming active.

I picked this version of Preflow-Push because it has an exceptional running time of $O(mn + n^3 U)$ where $m$ is the number of arcs in a network, $n$ is the number of nodes, and $U$ is the largest capacity of any arc in the network. Also, because excess is capped at any particular node, for the specific network that will be built in this project, this feature will limit the number of conflicting choices the algorithm could make and have to fix. Conflicts come in the form of two flights being picked the same flight as a continuation flight. This situation is represented in the network when the continuing flight’s origination node has an excess greater than 1.
The Network Model

In order to make the system more robust, I decided to make the network model independent of the rest of the system. This turned out to be a very good decision because a significant performance improvement was found by changing the network abstraction, and the rest of the system could be left unchanged. It also made the system easier to write because once the final network model was created and fully tested, I no longer had to look in the network abstraction to find the bugs that showed up.

Figure 9 Original network model

The original network model used three objects to describe a network. The Node object held information about the node's potential\(^6\) and how much excess was sitting at the node at a particular time. The arc object contained an upper

\(^6\) Potential is the distance from source or sink dependent on what the algorithm calls for.
bound (capacity), lower bound, and the current flow along it. Tying the two
together was the Network object. The network is simply a collection of nodes and
their connecting arcs. However, the data structure that is chosen to represent that
collection can have dramatic effects on the performance of the algorithms running
on the network. The representation I initially chose was the node-node adjacency
matrix. This representation is a matrix where both column and row indices
represent a particular node. The object at certain row and column is either null or
the arc that runs between the two index nodes. This method is efficient in that
every arc can be accessed in constant time by indexing the start and end node. To
give the same efficiency for the nodes, the network object also included a node
vector that would index nodes by their identification.

From this representation, I implemented the Preflow-Push algorithm and
the altered version to compute the minimum flow. I also created constructor
methods to easily generate a network. I created sample networks and tested the
algorithm for correctness.

After I was satisfied that the network implementation was correct, I went
to work on the rest of the system. At one point, I decided to see if the two main
parts could be integrated with the network model. The test was fairly simple, and
was to just create a network based on a schedule file. The test would then
terminate before any optimizations were to occur. For small schedules, the
network was created with no problem. However, the schedule did not have to
grow by much before I realized there was a major problem. When trying to create
a network model for a medium to large scale schedule, I noticed that the
performance of the system became unbearably slow. When the program reached
the network creating portion of the code, the hard disk drive would grind for long
periods of time. Eventually, the program would halt with an out of memory error.
At this point, I began suspecting the node-node adjacency matrix was taking too
much space. I analyzed the number of nodes and found that the program was
attempting to allocate memory for a matrix greater than 3000 by 3000 in some
cases. This was clearly unacceptable, so I thought of other ways to represent the
network.

I chose to use adjacency lists. In this representation, each node has record
of all arcs leaving it. This has the drawback that now it make take O(m) time to
find a particular arc where m is the total number of arcs in the network. In order to
keep this fact from causing prohibitive performance in the minflow algorithm, I had
to rewrite my implementation to keep track of all nodes that are important at a
particular time. This proved to be fairly easy. The Preflow-Push specification itself
requires the implementation to know which of the network’s nodes are active.
Because each node has access to its outgoing arcs, all of the arcs that are
potentially admissible in the push stage of the algorithm are immediately available.
In theory, the time to search the arcs leaving a single node is still O(m) but in a
typical network, each node has roughly the same number of arcs leaving it, so the number of arcs to search is reduced to no more than 16 in practice. To further aid the run time of the algorithm, I also added a field to the node object to list the arcs entering the node. This was done because during the push stage of the algorithm, flow may need to be pushed backwards along an arc. Performance would be greatly hindered if the algorithm did not have immediate access to these arcs.

Similarly, the arcs were augmented with information about the start and end nodes so that information is available in constant time. I also added code so the network object keeps track of which nodes are sources and sinks. This is important when determining the distance from the sources and sinks because these nodes can be accessed in near constant time.

This version of the network model proved to work well. Although the program still called for networks of 3000 nodes, because the adjacency list method
stores nodes in linear space, finding the space to accommodate the network was fairly easy on my conventional Pentium III testbed. In the end, creating the network was still the largest logjam but it appeared that the bottleneck is most affected by the amount of memory in the system. Most of the time in creating the network in large schedules was spent swapping data to the slower hard disk. This was not necessary for smaller schedules. This program run on a larger computer or workstation with a large memory would not see this performance problem.

The new version of the network model really shined when the minflow algorithm was run. The algorithm ran on the largest of schedules very quickly, and turned out to be one of the least time intensive portions of the whole system. This was attributed to the fact that all of the currently important variables – the active nodes and their connecting arcs – were kept track of. These variables were a subset of the whole network that was small enough to fit in memory, and possibly in the faster cache memory. Therefore, although the entire network may have portions stored on the hard disk, very little swapping was actually necessary.
The Schedule Model

The Schedule model is much more complex than the network model, however, much of its structure is simply setup for the network model. A schedule is broken up into different views on the same data. This is done in order to more quickly create a network, which itself is view of the same data. These views are
flights sorted by City, flights ordered by the plane they are assigned to, and the flights placed into the network model.

When creating a schedule from a file, each record in the file is turned into a Flight object. A flight object consists of basic information about a trip. This includes the flight number, departure time and location, arrival time and location, and which days and times of the year the flight is to be run. It does not represent a specific flight on a specific date however; this is left to the Flightnode object discussed later. The industry often flies what are known as through flights. These are flights with multiple stops that share the same flight number. In order to keep these legs together and to prevent broken flights as described earlier, all legs are stored in the same flight object. Leg Objects store the information about each leg and are stored in an array in a flight object. In cases where there are multiple records for flights of the same number in the schedule file, multiple flight objects are created. This can happen in situations such as flying a different equipment on different days of the week, or different times of the year.

When a flight object is created, it is placed into a City Schedule Object. A City Schedule is a partial view of flights in which all the flights share the same starting city for the first leg of the flights. These flights are organized in a tree structure ordered by their starting time. This is done to facilitate creating the network model of the schedule. As the network is built, connections are made
from a terminating flight to a departing flight by finding the city schedule corresponding to the location the terminating flight ended at, and searching the portion of the tree that occurs after the time this flight arrives at. All of the flights in this subtree are eligible continuation flights to be served by the same plane as the terminating flight. Lastly, the city schedule contains links to the flightnode objects representing each flight in the network for speedy access. This is discussed in greater detail in the next section.

Once the schedule file is read and flight objects are created, the network model of the schedule can be created. Separate networks are created for each equipment type in the fleet. A flight is added to the network representing the same type as the flight. To add a flight to a network, I used a similar process as outlined in Figures 4, 5, and 6. Using the node object as a model, I created an extension to link to a flight on a specific date. This extended object is a Flightnode. Because it is an extension of the node object, flightnodes can be placed in a network in the same way that node objects can. This is important as it means that no modifications are necessary to the underlying network model to implement a network of flights.

To create a network of flights for a particular equipment type, a new empty network is constructed using the constructors defined in the network model. A source node and sink node are added to the network. Now, for each flight of this
type in the schedule, flightnodes are added to represent that flight in the network. Remember that flight objects do not represent a particular flight date, but more closely a template. The actual dated flight is represented in the flightnode. There is a many to one mapping of flightnodes to flights; for each date the fleet schedule is to apply to, a new flightnode is created for each flight in the schedule if it is valid on that date. For instance, if Flight 123 is supposed to fly on Mondays, Wednesdays, and Fridays after October 5, and we are scheduling the fleet from Sunday, October 1 to Saturday, October 14, new flightnodes will be created representing Flight 123 on Friday October 6, Monday October 9, Wednesday October 11, and Friday October 13.

When a flightnode is created, it is integrated into the system, it is added with a regular node to represent the termination of the flight. To connect the pair of nodes into the network, an arc is created between the flightnode and the termination node. This arc has a lower and upper bound of 1 unit of flow. Two more arcs with lower bounds of zero and upper bounds of one are added from the source to the flightnode, and from the termination node to the sink.

To complete the network as described previously, connecting arcs must be placed in the network from each flight to its potential continuation flights. To do this, the program iterates over all flightnodes in the network one at a time. For each flightnode, the algorithm examines its terminating location and picks the city
schedule corresponding to that location. All potential continuing flights will come from this city schedule. For the minflow algorithm to find the theoretical minimum fleet, it is required to create links between all potential continuation flights. It became very apparent that for large schedules there were very many eligible continuations flights, although a large number of these candidates were long shots. These were flights that departed days after the inbound flight terminated. To prevent wasting valuable memory, I set an artificial beam to limit the number of potential continuation flight candidates that the minflow algorithm would consider. This branching factor, $b$, dramatically reduces the size of the network from $O(n^2)$ or a possible $n$ connections for each of $n$ flights, to $O(nb)$: a possible $b$ connections for each of $n$ flights. The program would begin searching for candidates in time order beginning at the time the inbound flight terminates plus an equipment-dependent minimum turn time. The program would then scan through the city schedule recording all potential continuation flights it finds until the limit $b$ connections is reached. At this point the program would begin searching for candidates for the next flightnode and would continue iterating until all flightnodes are accounted for.

When the network view of the flights is complete, the minflow algorithm is run on the network to find the minimum fleet, and aircraft routing. The results left on the network are then passed on to create the final view of the flights arranged by the planes they are assigned to. Surprisingly, the beaming of the network did
not have much effect on the minimum fleet found. I attributed this to the fact that
the minimum fleet would occur when the planes have a high utilization. If the
minflow algorithm picked a continuation flight that occurred a long time after the
inbound flight landed, that plane would not have a very high utilization. Therefore,
the minflow algorithm tends to pick flights that leave the plane on the ground less.

The last view of the flights shows them by the plane in the fleet that the
flight will be flown by arranged into lines of flight. A line of flight is an ordered list
of flights an individual plane will operate from entering until leaving service. This
view is created from the network model the program just manipulated. To make
this view, one more manipulation of the network view is required.

After the minflow algorithm is run on the network, there will be exactly
one arc with a unit flow leaving each termination node. This will either lead to
another flightnode, or to the sink. If this arc leads to another flightnode, then the
flight represented by the connecting flightnode is the next in the line of flight. In
this case, this flightnode is referred to in the next field of the previous flight’s

![Diagram](image)

**Figure 12** Partial network showing Flight B succeeds Flight A. The flightnode for
A (dark) will have its next field refer to the flightnode for B.
flightnode. (See figure.) If the arc leads to the sink, then this was the last flight on the plane's schedule before leaving service. The previous flightnode's next field is set to null in this case. To find the beginning of each line of flight, we note that by definition of flow networks, there is as much flow leaving the source as there is entering the sink. Because each unit of flow entering the sink represents the completion of a line of flight, each unit of flow leaving the source after the minimum flow is found is synonymous with a plane entering service and operating a line of flight. The arcs leaving the source with flow along them are recorded as the flightnodes they end at mark the beginning of a line of flight. It is from here that we can finally assign planes to flights and create this final view.

The array of planes is initially filled with plane records from a file. This file lists each plane in the airline's fleet by fleet number or identification. Initially these planes are unassigned. The program then begins assigning planes by matching the recorded beginnings of lines of flight to unassigned planes in the fleet of the same type. In order to determine the rest of the line of flight, the flightnodes at the end of the starter arcs are traversed like a linked list using the next field of the flightnode object until the next field is null, signifying the end of the line of flight.

At this point, we have completely assigned the planes to the fleet, solving the fleet scheduling problem. However, the results of the fleet scheduling would suggest a slightly different method of assigning the fleet.
Results and Analysis

I ran the program several times over the development period. The data that I used included a large real world schedule that has been flown by a major airline to prove the feasibility of the program. I also used smaller subsets of this schedule, as well as other specifically handcrafted schedules to test the functionality of the program and to remove all bugs. These test schedules included tests to ensure that the minflow algorithm did in fact find continuation flights and create lines of flights requiring the fewest number of planes. The small sets also tested the program to ensure that flights were only added to a line of flight if they were valid for that particular day.

The large test set includes about 20 different equipment types, 17,000 potential weekly departures\(^7\), and about 150 different destinations. The schedule also followed a basic “hub and spoke” style\(^8\). These statistics are typical for a large U.S. airline. The schedule is expected to require around 600 jets so that no flights are missed.

\(^7\)This count is skewed lower because departures which were a second, third, or fourth leg of a flight with one continuous flight number are not counted. This number is equal to the number of Flight objects created.

\(^8\)Hub and spoke is a network where many flights begin at an airport and fly to a large airport built for connecting traffic. Passengers are then able to connect to a large number of flights that have arrived at this “hub” airport and takeoff again for their destination, a “spoke” airport.
Each test was run from within the Visual Age environment so that I could track progress. The final product run standalone should run faster.

Fleet Scheduling Results

The actual number of planes in the minimal fleet varied dependent on the branch factor chosen, the date to start scheduling the fleet, and the number of days to be scheduled at a time. I ran many experiments to test the effects of these variables. I expected that the Branching Factor would have the most effect on the scheduled fleet. The reason for this idea is that the branching factor is a natural optimality limiter. A small branching factor would potentially stop the program from adding candidate successor flights to the network before the best candidate is found. If that flight is not included, the minflow algorithm does not have access to it, and as a result may pick a less desirable flight to connect into a line of flight.
I ran 8 tests of the program with the start date held to June 18, and the number of days to be scheduled held to 7. As expected, the size of the minimal fleet decreased as the branching factor was increased. At $b=5$, the minimal fleet was a very unrealistic 943. What surprised me however was that the algorithm leveled off at 576 planes when the branching factor was only 13. Values higher than 13 led to longer run times with no additional benefit. The branching factor was added to prevent the network from being excessively large. Because the branching factor’s presence gave preference to shorter ground times, and thus higher utilization of the fleet, and because a minimal fleet would have the best utilization, I concluded the addition of the branching factor was beneficial to the running time and space requirements without being a hindrance to finding the minimal fleet.
I expected that the number of days to be scheduled would have a limited effect on the size of the minimal fleet if the schedule created was to be longer than a week. For cases where the program was to schedule less than a week, I expected a slow ramp-up to a limit representing the size of the week-to-week fleet. This was because the template schedule varies little week to week. Thus if a fleet flew a particular schedule for one week, it could fly the same schedule the next week. What I found was very surprising and forced me to think differently about how the minimum fleet should be found.

As expected, the number of planes in the minimal fleet grew steadily as the number of days scheduled grew to a week. However, as you can see by the chart, the fleet size did not level off when the number of days scheduled passed a week; instead the minimal fleet continued to grow. In fact, the rate the minimal fleet grew
rose after one week. After much thought I began to realize why. I ran the program to schedule planes for a week and read through the lines of flight for many of the jets that were being scheduled. In almost all cases, the plane was assigned to fly a series of flights that left it in a different location than it began the week. Also, some planes spent longer periods of time in the air than others. Some planes finished flying their line of flight in fewer than a week, and were inactive for the weekend. While this in and of itself is not cause for concern, it showed that one of my goals was interfering with the ability to find a minimal fleet. The program was attempting to find a minimal fleet without including any dead heading flights. In my earlier simulations, I was attempting to fly a schedule with no deadheads for two weeks. Keeping this goal proved to be an impossibility because the template schedule would call for a flight on a day where there were no planes available to fly the route. The solution that the minflow algorithm would be forced to bring another plane into service in order to fly this route. The effect of this is lower utilization of the fleet and longer ground times.

These results led me to the conclusion that deadhead flights are a necessity in practice in order to keep the fleet utilization high and the minimal fleet low. A potential solution is to schedule the flights by running the algorithm week by week. To build a fleet scheduling for more than a week, the algorithm could be run for the two weeks separately, and then spliced together. To do this, a list of where each jet is located at the end of a week would be maintained. When assigning lines
of flights to planes in the second and subsequent week, as many lines of flight as possible would be matched to planes that are at the location where a line of flight begins. Any leftover lines of flight would be assigned to planes that are nearby, and are free long enough to fly between its current location and the beginning of the line of flight.

I expected that the starting date of the fleet scheduling would have a minimal effect on the number of planes found to be required in the minimum fleet. Because the schedules change little week to week, this assertion was found to be accurate.
I ran 5 tests of the program with the branching factor set to 15, and the number of days to schedule set to 7. As you can see, there was little effect that starting on a different date had. The small changes that we can see were attributed to the adding of additional flights in the summer vacation season.
Performance Results

In order to test performance fairly, I ran the program as the only program running after booting up, and took the lowest time of two tries so that cache contents would be more consistent. I hard-coded stopwatches for the schedule loading, network creation, and minflow fleet minimization portions of the program. As it would happen, it was not relevant to plot the changes in the schedule load portion because the input set was constant throughout the test sets. Each test loaded the entire schedule, and then used the portion that was relevant to that test. Differences in the timing were most likely caused by differences in the cache contents at runtime. The network creation times and fleet minimization times were much more dependent on their widely varying inputs however.

Because the network creation theoretically runs in $O(n^2)$ time where $n$ is the number of flights to be examined, I expected the network creation time to be greatly effected by the number of days to be scheduled. For each day in the schedule, the network creation algorithm has to examine the validity for each flight in the template schedule, expanding the value of $n$. 
In practice, the algorithm performed as expected, growing from barely seconds to an almost unbearable hour and a half of computation as the schedule grew from 1 day to 2 weeks. It is noteworthy that for schedules of less than 6 days, very little time if any was spent swapping memory and hard disk. Beyond 7 days however, almost all of the time was spent grinding the hard disk. Clearly, the speed at which the algorithm ran was dependent on the memory available to it. Also, if the memory is not available, having a fast hard drive is beneficial to the algorithm’s running time.
Similar results were expected of the fleet minimization time. The Excess-scaling Preflow-Push algorithm that was the bulk of the fleet minimization process runs in $O(mn + n^2U)$ time where $m$ is the number of arcs in the network, $n$ is the number of nodes, and $U$ is the maximum capacity of all arcs in the network.

Adjusting this to our more specific network, $n$ is exactly $3f + 2$, $m$ is at most $(3+b)f$, and $U$ is 1 where $f$ is the number of flights to be scheduled, and $b$ is the branching factor. Thus, the fleet minimization step should run in $O(bf^2)$ time. $f$ grows with the number of days to be scheduled and $b$ is held constant, a parabolic increase in minimization time is expected.
Again, the tests were run and as expected, the algorithm performed the fleet minimization step in parabolically increasing time. It is of note however that in this case, the computer spent nearly none of the time swapping to the hard disk. This is evidenced by the way that the algorithm works. The algorithm keeps a collection of important data points handy at all times. This data set increases and decreases incrementally as opposed to the network creation step in which most of the current data to be added is either all new, or has not been used recently. This has direct effect on the caching of data. For the fleet minimization step, the pertinent data at any step was small enough to fit in system memory so swapping was minimized. The choice of using the Preflow-Push algorithm to find the minimum flow and thus the minimal fleet proved to be a good one. As it turned out, the bottleneck in the program was not the fleet minimization, but the creation of the network the minflow algorithm was to be run on.

I expected the time required to create the network would be greatly affected by the branching factor. A small branching factor would prevent the network creating algorithm from scanning through the entire network to find candidates for continuation flights. A larger branching factor could potentially scan the entire list of flights to find candidate successor flights before reaching the limit.
After running the 8 tests based on branching factor, I was very surprised with the results. As expected, the creation time started small and increased as I increased the branching factor. However, this trend was interrupted as the creation time decreased dramatically between $b=12$ and $b=15$. The trend continued anew after $b=15$ as the time to create grew again. It is also noted that at $b=13$, the lowest minimum fleet size was found and further opening the branch factor limitation did not yield a lower fleet size. I was unable to think of a suitable explanation for this situation until I analyzed the effect of branching factor on the fleet minimization time.
The branching factor was expected to have a significant effect on the time to minimize the fleet. The reason was that there would be fewer choices for the minflow algorithm to choose amongst in order to find a minimal flow. This is supported by the theoretical running time of the Preflow-Push algorithm adapted to this network: $O(bf^3)$. Linear growth was expected.

\[
\text{Fleet Minimization Time vs. Branching Factor, Days=7}
\]

As I ran the tests, I realized my expectations were far from what would actually occur. Instead of linear growth, the algorithm spent over 11 minutes running on a smaller network, and gradually decreased to a fairly steady state when $b \geq 13$. Again, further thought was necessary. Correlating the results of this test with the Minimal Fleet vs. Branching Factor tests showed that the level off
occurred when the lowest minimal fleet was found. This fact, and a closer look at the Preflow-Push algorithm led to a justification of these results.

The Preflow-Push algorithm when run in reverse to find the minimal flow would pick a flight and try to assign it a successor flight. If this flight was a good pick, it will remain for the duration of the algorithm’s run. If it was a poor pick, it would be reassigned when a conflict occurred. A conflict occurs later in the algorithm’s run when it is determined that another flight must connect to one flight’s previously assigned successor. A flight can be the successor to at most one other flight, when this occurs, a reassignment must take place in order to continue. As a last resort, a conflict is resolved by terminating or beginning a new line of flight by pushing flow either back to the source or to the sink. Therefore, bad picks cause the algorithm to do additional work.

The schedule that the program is trying to build is based on the hub and spoke system. This means that many flights arrive at a hub near the same time, and leave again near the same time in order to offer a large number of connections to the passengers. The set of flights that arrive and depart during this period of high activity is called a bank. The network creation algorithm searches for candidate successor flights by finding flights that leave immediately after the minimum turn time expires. When a bank of flights of a particular type arrives at a hub, each of these inbound flights has the same set of outbound flights available as candidate
successor flights. Therefore, when the branching factor is too low, all of the inbound flights are assigned the same set of outbound flights as possible continuation flights. For instance, Flights 1, 2, and 3 terminate at a hub at noon. At 1pm, Flights 4, 5, and 6 depart for their destinations. All of these flights are flown by the same equipment which has a minimum turn time less than one hour. If the branching factor is only 2, then the network creation algorithm would create arcs leading to Flights 4 and 5 as continuation for all of Flights 1, 2, and 3. No arcs would be drawn to Flight 6. As a result, an additional plane would be requested to fly Flight 6, and one of Flights 1, 2, or 3 would be the last in a line of flight and subsequently pulled out of service even though it would be physically available to fly Flight 6. The figure here shows the result of situation. The grey lines are allowable network additions that the low branching factor prevented from being placed in the network. Without these arcs, the true minimal fleet could not be found. More importantly, for this test, without these arcs, a conflict would have to be corrected first. All three of the inbound flights would be assigned to one of the two outbound flights. The conflict is resolved incrementally, with eventually one flight being the termination of a line of flight, and the other flights continuing to Flight 4 or 5. These corrections take a lot of time, and slow the entire process. A larger branching factor allows more choices for the algorithm to choose from, and thus allows fewer conflicts.
These results suggest that the optimal branching factor is the largest number of flights of a particular type that enter or leave a hub within a bank. It also suggests that the optimal branching factor is dependent on the type. For instance, a large airline with multiple types will most likely have a multitude of smaller jets landing and departing at the same time, but few larger jets arriving from and departing for longer range flights. A smaller branching factor for the larger jets would not have an adverse effect on the minimal fleet schedule found.

Thinking along similar lines, I began to realize why the network creation time dipped between $b=13$ and $b=20$. I came to the conclusion that this dip was caused by the nature of the schedule and the network we were building. the network being created is based on a hub and spoke network, flights that terminate at a hub airport would have many options for continuation flights. These flights leave a short time after the terminating flights arrive at the hub. flights are stored in the City Schedule in chronological order, the network creation algorithm finds these candidate successor flights in order. Searching begins at the earliest point in the schedule after the minimum turn time has elapsed. all of the outbound candidate flights in a bank depart near the same time, they are located near each other in the City Schedule. Therefore, when the branching factor is near the size of the bank of flights, the network creation algorithm stops searching for flights when all of the flights in the bank are labeled as candidate successors for this flight, and continues with another flight. If the branching factor is much higher than the size
of a bank, searching continues uselessly the fleet minimization step will not find a better solution at this point.

This explanation however does not explain why the network creation time was higher when the branching factor was less than the bank size. Each time the test was run, it was run on the same set of flights. The only difference was the number of candidate successor flights to add to each flight. Because the same successor flights were being chosen for many flights in a bank, these flights were able to be cached. In this way, a higher branching factor would take more advantage of using the cache.

Finally I compared the times to create the network and find the minimal fleet against the date in which the fleet schedule was to start. This was to give different viewpoints of the template schedule. Because there was little change in the size of the schedules to be made, I did not expect much fluctuation in the running times. I ran 5 tests with the number of days to schedule fixed at 7, and the branching factor fixed at 15. The start date varied as shown.

As the charted results below show, the start date had no real effect on the time required to create the network, nor to find the minimal fleet on that network.
Network Creation Time
vs. Start Date, BF=15, Days=7

Fleet Minimization Time
vs. Start Date, BF=15, Days=7
Contributions

This project shows that it is feasible to use a flow model to find the minimal fleet required to fly all flights in a schedule. The project also took the next step and extrapolated lines of flight from the minimal fleet to assign to individual planes. All of the tests of this program were run on an off the shelf personal computer with a modest amount of memory and CPU power to show that excessive computation power was not necessary. This small packaging of the flow model of a schedule was possible in part by the choice of a good minimum flow algorithm in the Excess Scaling Preflow-Push algorithm, and by pruning the network of unnecessary arcs via a well-chosen branching factor.

A goal that was not met however deals with the use of deadheading flights. As the results of this project show, a fleet scheduling for a large, modern day airline must have a tradeoff between adding deadheading flights or adding additional planes. Adding the deadheading flights allows an airline to get the maximum utilization from their planes. This is key for profitability. Because a plane can only earn money when it is flying revenue flights, it is important to have planes in position to fly those revenue flights. This project however does not imply the best way to add deadheading flights. In summary, these are my contributions:
• This project finds the minimal fleet of a real world schedule using the network flow model,
• implements the network model to perform well on a small computer when the scale is great
• shows that narrowing the choices for successor flights has little to no effect on the minimal fleet

The problem of scheduling is not limited to airline scheduling. There are many other forms of the scheduling problem that are similar to the fleet scheduling problem dealt with in this project. These include job shop scheduling, and many other offline scheduling problems. With fairly minor changes, it should be possible to solve some of these problems using the flow model on a personal computer as has been done in this project. It is perhaps here where this project is more applicable. While a large airline would most likely be able to afford a more powerful computer to perform this scheduling, a smaller workshop may not. Adapting the findings in this project to that task would be very beneficial to these smaller organizations if it is still maintainable on a personal computer.
Bibliography


