Per-Seat, On-Demand Air Transportation
Part II: Parallel Local Search

D. Espinoza, R. Garcia, M. Goycoolea, G.L. Nemhauser, M.W.P. Savelsbergh
Georgia Institute of Technology

Abstract

The availability of relatively cheap small jet aircrafts suggests a new air transportation business: dial-a-flight, an on-demand service in which travelers call a few days in advance to schedule transportation. A successful on-demand air transportation service requires an effective scheduling system to construct minimum cost pilot and jet itineraries for a set of accepted transportation requests. In Part I, we introduced an integer multi-commodity network flow model with side constraints for the dial-a-flight problem and showed that small instances can be solved effectively. Here, we demonstrate that high-quality solutions for large-scale real-life instances can be produced efficiently by embedding the core optimization technology in a local search scheme. To achieve the desired level of performance, metrics were devised to select neighborhoods intelligently, a variety of search diversification techniques were included, and an asynchronous parallel implementation was developed.

1 Introduction

The availability of relatively cheap small jet aircrafts will revolutionize the air transportation business, as it makes operating an on-demand service in which travelers call a day or a few days in advance to schedule transportation a viable business proposition. Such dial-a-flight systems give regional travelers the option of boarding small jets flying to and from smaller, less congested airports that are closer to where they live and where they want to go. They also avoid the need to deal with packed parking lots, long lines at security checkpoints, flight delays, lost luggage, and plane changes at hubs.

To effectively manage day-to-day operations at an on-demand air transportation service, several optimization-based scheduling components need to be employed, among others: (1) an online scheduling system to quickly inform passengers if their air transportation requests can be served and at what price, and (2) an off-line scheduling system to construct minimum cost pilot and jet itineraries for the next day once the reservation deadline has passed.

In this two-part paper, we discuss the components of an off-line scheduling system developed for and in collaboration with DayJet Corporation ([4, 5]). In Part I ([7]), we introduced an integer multi-commodity network flow model with side constraints for the dial-a-flight problem, presented a variety of techniques to control the size of the network and to strengthen the quality of the linear programming relaxation, and showed that small (8 plane) instances can be solved effectively. However, DayJet expects to operate with a fleet of more than 300 hundred jets by 2009 and with a fleet of more than a thousand jets by 2011. Therefore, DayJet needs schedule optimization technology that can efficiently handle instances of the dial-a-flight problem involving hundreds of jets and thousands of requests. The integer multi-commodity network flow model with side constraints introduced in Part I cannot be used directly to satisfy their needs. We demonstrate, however, that by embedding the core optimization technology in a parallel local search scheme, it is possible to produce high-quality solutions efficiently for large-scale real-life instances.

At about midnight, when off-line schedule optimization is performed, a feasible schedule already exists. This schedule has been produced by the reservation system which sequentially processes requests for air transportation. In order to ensure that commitments to customers can be satisfied the reservation system maintains a feasible schedule of accepted transportation requests at all times. Therefore, rather than constructing an optimized schedule from scratch, we have chosen to improve the schedule produced by the reservation system using a local search scheme that relies on our core scheduling optimization technology to completely explore suitably chosen neighborhoods.

Even though a straightforward local search scheme produces high-quality schedules, it may not produce them fast enough. There is a three hour window in which no new requests are accepted and the schedule must be finalized so that flight plans can be filed with the Federal Aviation Administration (FAA) in time before the morning shift commences. To enhance the performance, we have created a set of metrics to select neighborhoods intelligently and we have devised a variety of search diversification techniques. Furthermore, we have developed a scalable asynchronous parallel implementation. The resulting parallel local search scheme has been shown to be capable of producing schedules that are within a few percent of optimality for instances with over 300 jets and over 2,800 requests.

Local search is an active area of research and our scheme has similarities to several other effective techniques. We mention a few. It has similarities to Randomized Adaptive Spatial Decoupling (RASD), introduced by Bent and Van Hentenryck ([3]), in that in each iteration of the local search a subset of the requests is selected, re-optimized, and re-introduced in the global solution. However, our scheme uses metrics to intelligently select the subset of request, whereas RASD relies on random choices. It has similarities to Adaptive Large Neighborhood Search (ALNS) proposed by Ropke and Pisinger ([10]) in that our scheme dynamically adjusts the choice of neighborhood over time to ensure that the most appropriate neighborhood is always being used. It has similarities to Large Neighborhood Search as introduced by Shaw ([11]) for solving vehicle routing problems in which constraint programming is used to re-insert a subset of extracted customers into the partial solution. Constraint programming techniques are used to re-insert customers, as opposed to using simple least-cost insertion ideas, to explore a much larger neighborhood. We use integer programming to explore large neighborhoods defined by a subset of planes and a subset of requests and consisting of all feasible flight schedules serving these requests. We have chosen to use large neighborhoods, because we have observed that small neighborhoods involving only simple changes to an existing schedule tend to converge to bad local optima as most of these simple changes are infeasible.

The remainder of Part II is organized as follows. In Section 2 we introduce the basic local search scheme. In Section 3, we present a variety of metrics designed to improve neighborhood selection. In Section 4, we consider modifications to the basic local search scheme to generate alternate neighborhoods, and we describe their use to enhance the overall performance. In Section 5, we discuss the asynchronous parallelization of the local search scheme. In Section 6, we give our recommendations for combining the ideas to yield a comprehensive scheme.

2 Optimization-Based Local Search

Local search schemes identify improved solutions in the neighborhood of the current solution. By using neighborhoods involving subsets of jets, we are able to employ our core optimization technology to completely explore the neighborhoods and therefore to identify the best schedule in a neighborhood. The use of optimization to explore (very) large neighborhoods has proven to be successful in several other transportation contexts, see for example Agarwal et al. [1], Archetti et al. [2], and De Fransesco et al. [6].

Since passengers never have to change planes, any subset of jets $J \subset \mathcal{J}$ of a given feasible schedule identifies a set of requests $R_J \subset \mathcal{R}$. Consequently, any subset $J \subset \mathcal{J}$ defines a smaller instance of the *dial-a-flight problem* (DAFP) which can be optimized using the multi-commodity network flow model described in Part I. This results in the basic local search scheme specified in Algorithm 1.

Algorithm 1 SEARCH(n, t)

```
randomly choose $J \subset \mathcal{J}$ with $|J| = n$
formulate and solve DAFP$(J)$
**if** an improved schedule is found **then**
    update master schedule
**end if**
**if** search time exceeds $t$ **then**
    stop
**end if**
```

Even though the basic local search scheme can substantially improve the schedule produced by the reservation system, the scheme can be enhanced in several ways to make it more efficient and more effective. These enhancements are detailed in the remainder of the paper.

For testing purposes, DayJet has provided us with the feasible schedules returned by their on-line reservation system for 10 simulated days of typical operation of a fleet in the southeastern United States. These schedules define instances of DAFP with 46 airports, a fleet of 312 jets and roughly $2,700$ requests. The basic statistics are shown in Table 1. For each instance we provide the number of scheduled requests (REQS), the total number of scheduled passengers (PASS), the number of legs flown (LEGS), the number of "deadheads" or empty legs flown (DHEADS), the distance required (DIST), and the flying time required (FLTIME).

Table 1: Initial on-line schedules.

DAY	REQS	PASS	LEGS	DHEADS	DIST	FLTIME
1	2,720	3,450	2,923	901	819,792	202,302
2	2,698	3,436	2,941	892	830,237	204,620
3	2,743	3,476	2,943	902	829,673	204,585
4	2,782	3,549	3,003	916	835,238	206,495
5	2,704	3,456	3,008	950	831,870	205,931
6	2,694	3,453	2,939	886	827,194	203,938
7	2,685	3,434	2,920	875	832,408	204,743
8	2,854	3,645	3,021	892	835,762	206,813
9	2,768	3,518	2,999	922	833,185	206,049
10	2,730	3,440	2,978	940	832,408	205,689

All of our neighborhoods result in a DAFP and we will use the technology described in Part I for its solution. As before, our objective is to minimize flying time. DayJet has imposed a time limit of 4 hours for off-line schedule optimization and we use this limit as the stopping criteria for all search schemes. Because we only want to spend a limited amount of time exploring each neighborhood, a time limit of 75 seconds was imposed on the solution of each integer program, and we ask the solver to return the first integer feasible solution found. Using the results in Part I to guide our selection, we use an aggregation level of $\kappa = 10$ and the most basic time discretization of $\Delta = 1440$.

The first question we ask is how does the number of jets n selected at each iteration affect the performance of Algorithm 1? Large values of n are more likely to result in larger schedule improvements, but will require more evaluation time. Given that we have only a limited amount of computation time for schedule optimization, the time required to explore a neighborhood determines the number of neighborhood searches. Smaller values of n may yield smaller improvements, but being able to perform significantly more searches may turn out to yield better results.

We ran Algorithm 1 with the settings discussed above for each of the 10 instances with different values of n. The results are shown in Table 2. The statistics presented for each n are the average percentage improvement in flight time over the initial on-line reservation schedule (IMPROV), the total number of neighborhood searches attempted (ATT), the percentage of neighborhood searches that were successful (SUCC), the average change per success in minutes (CHG/SUCC), and the average integer program solution time in seconds (TIME/IP).

Table 2: Impact of n on Algorithm 1.

n	IMPROV	ATT	SUCC	CHG/SUCC	TIME/IP
3	5.70	235,625	2.30	21.60	0.13
4	5.54	93,248	5.28	23.06	0.36
5	5.13	43,457	9.90	24.48	0.83
6	4.88	22,787	16.51	26.63	1.72
7	4.77	13,775	24.25	29.23	3.05
8	4.20	8,064	33.61	31.78	5.67

We see that even with this basic scheme, we are able to get substantial improvements within the four hour time limit. We observe that the success rate, the average change per success, and the average integer program solution time all increase with n. However, the average reduction in flight time decreases as n gets larger. The reason is that the average change per success does not increase enough to merit the extra search time needed for larger neighborhoods. This becomes clear from the number of neighborhoods we can search for the smallest values of n. The best results in this experiment are obtained for $n = 3$. However, as we will see later, when we further improve the success rate or the average change per success for the larger neighborhoods, the advantage will shift in their favor.

One question that may come to mind is why we do not continue past the first feasible solution found in the hope of improving the average change per success? We performed such an experiment, but the results were worse. The initial solution was usually the optimal one, and we spent unnecessary time trying to prove optimality and thereby reduced the total number of neighborhoods searched.

3 Metrics

The computational experiment in Section 2 demonstrates that the basic local search scheme is able to substantially improve the schedules produced by the reservation system. Next, we focus on improving its efficiency by trying to make intelligent choices of neighborhoods rather than random ones. Therefore, we defined and experimented with a variety of metrics that aim to identify a subset of jets that is (more) likely to contain an improved schedule. We created metrics by trying to correlate success (i.e., finding an improved schedule) with characteristics of the subset of jets. We distinguish two types of metrics: one-jet metrics and two-jet metrics.

A one-jet metric quantifies a particular characteristic of a single jet's current itinerary. The idea is to use a one-jet metric to select a base jet that is a good candidate for improvement. Typically, the more successful one-jet metrics are those that identify jets with "bad" itineraries. To identify bad itineraries, it is helpful to examine good itineraries. High-quality schedules are characterized by many legs with multiple passengers (i.e., satisfying several requests with one or two flights) and few legs without passengers (i.e., limited deadheading or repositioning of jets). Two of the more successful one-jet metrics were the ones that chose jets with itineraries that had a small number of legs with multiple passengers ($\frac{\text{passenger miles}}{\text{total miles}}$), or that had a large amount of deadhead time ($\frac{\text{deadhead time}}{\text{duration}}$). Other metrics chose jets that had itineraries that visited a large number of different airports, had a large number of flight legs, or served a small number of passengers.

Two-jet metrics are more complex and are used to measure the similarity between the itineraries of two jets. After we have chosen the base jet, we want to choose the remaining jets in the neighborhood so as to complement the base jet using a two-jet metric. We compare the itinerary of the base jet with the itineraries of all other jets using a metric and choose the ones that compare the most favorably. Several two-jet metrics were successful. Here we present the three most effective ones.

The first two-jet metric is based on common airports visited in the jets' itineraries. In order for the base jet to swap legs with another jet, the legs flown by the other jet must depart or arrive at airports that are close to where the base jet is usually located. A good estimator for the base jet's location is its home base. Therefore, for a base jet j^B, a jet $j \neq j^B$, and some fixed time radius t, we define the *base airport radius metric* to be

$$\textsc{radius\_metric}(j^B, j) = |A|$$

where

$$A = \{a \in \mathcal{A} : a \text{ is visited by } j \text{ and } \textsc{flight\_time}(\textsc{home\_base}(j^B), a) \leq t\}.$$

The second two-jet metric is based on the passengers serviced in the other plane's itinerary. For the base jet to be able to exchange passengers with the other plane, the passengers serviced by the other jet must have origins and destinations that are visited by the base jet. The key is to make sure that the exchange is also feasible with respect to time. For a base jet j^B and a jet $j \neq j^B$, we define the *passenger matches metric* to be

$$\textsc{matches\_metric}(j^B, j) = |O \cup D|$$

where

$$O = \{r \in \mathcal{R} : r \text{ is serviced by } j \text{ and } j^B \text{ departs from } \textsc{origin}(r) \text{ before}$$
$$\textsc{latest}(r) - \textsc{flight\_time}(\textsc{origin}(r), \textsc{destination}(r))\}$$

and

$$D = \{r \in \mathcal{R} : r \text{ is serviced by } j \text{ and } j^B \text{ arrives at } \textsc{destination}(r) \text{ after}$$
$$\textsc{earliest}(r) + \textsc{flight\_time}(\textsc{origin}(r), \textsc{destination}(r))\}.$$

The final two-jet metric is based on the distance between the base jet and the other plane throughout the day. We would like the planes to be relatively close to each other to be able to consider more exchanges. Let $t_1, \ldots, t_p \in \mathcal{T}$. For a base jet j^B and a jet $j \neq j^B$, we define the *distance metric* to be

$$\textsc{distance\_metric}(j^B, j) = \sum_{i=1}^{p} \textsc{flight\_time}(a(j^B, t_i), a(j, t_i))$$

where $a(j, t_i)$ is the closest airport to j at time t_i.

The local search scheme with metrics incorporated is presented in Algorithm 2.

Algorithm 2 $\textsc{guided\_search}(n, t)$

```
select one-jet metric
choose base jet $j^B \in \mathcal{J}$
select two-jet metric
choose $J \subset \mathcal{J}$ with $|J| = n$, $j^B \in J$, and the remainder of $J$ to be the closest $j$ to $j^B$
by the chosen metric
formulate and solve DAFP($J$)
if an improved schedule is found then
  update master schedule
end if
if search time exceeds $t$ then
  stop
end if
```

The best results were obtained when incorporating two-jet metrics. While preliminary experiments with one-jet metrics showed that they were useful on their own, we found that when combined with two-jet metrics the results were not consistent enough to justify choosing the base jet intelligently. Therefore, in the computational experiments we report, base jet j^B in Algorithm 2 is chosen randomly.

To validate the use of metrics, we reran the experiment in the previous section using Algorithm 2. In our scheme, 25% of the time we chose J randomly, and in all other iterations we selected one of the three metrics discussed above with equal probability (i.e., 25% each) and used it to choose J. The results are shown in Table 3.

Table 3: Metrics

n	IMPROV	ATT	SUCC	CHG/SUCC	TIME/IP
3	5.86	212,750	2.53	22.30	0.17
4	6.20	83,103	6.52	23.49	0.47
5	6.45	39,399	13.10	25.64	1.10
6	6.37	20,642	21.44	29.51	2.40
7	6.41	12,664	31.99	32.47	4.27
8	6.10	7,303	43.41	39.49	8.18

We first observe that for all n the improvements are better than for the random neighborhoods. The worst overall improvement in this experiment (5.86%) was better than the best improvement in the previous experiment (5.70%), and both of these occur for $n = 3$. The size that performs best in this experiment is $n = 5$, and to get a better understanding of why, we have plotted the average reduction in flight time for $n = 3, 5$ and 8 at half hour intervals throughout their respective searches in Figure 1.

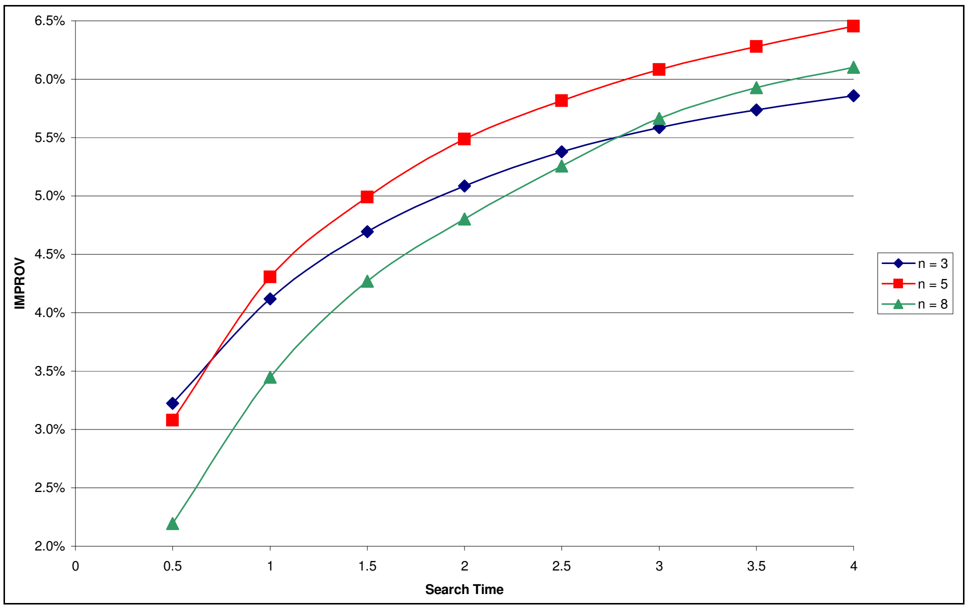

Figure 1: Tailing Off

In the first half hour, the improvement mimics what we saw in the previous experiment with worse improvement as n gets larger. But as we get to 45 minutes, $n = 3$ has been overtaken by $n = 5$, and by 2.75 hours, it has also been overtaken by $n = 8$. It is clear that smaller neighborhoods are tailing off sooner than larger neighborhoods, and had we allowed the search to continue, $n = 8$ would have eventually surpassed $n = 5$. We did not notice this phenomenon in the previous experiment because we were not getting far enough in the searches. The metrics have accelerated our searches and allowed us to achieve more within the four hour time limit.

Since $n = 5$ was most successful, we would like to gain a better understanding of what each individual metric contributed to its performance. We have divided the performance among each of the individual metrics and present them in Table 4. The distance metric contributes the most to the improvement despite possessing the worst average change per success because its success rate is the highest of all. The best average change per success belongs to the passenger matches metric, and it is not surprising that it performs just as well as the distance metric. The most important point is that all of the metrics outperform choosing the subset randomly which averaged about one-third of the improvement yielded by any of the other metrics.

Table 4: Metric contributions for $n = 5$.

METRIC	IMPROV	ATT	SUCC	CHG/SUCC	TIME/IP
RANDOM	0.69	9,736	6.26	23.35	0.83
RADIUS	1.85	9,770	15.03	25.86	1.30
MATCHES	1.95	9,838	12.98	31.37	1.09
DISTANCE	1.96	10,055	17.99	22.19	1.18

The tailing off shown in Figure 1 indicates that success rate and the average change per success are functions of search time and decrease as we get closer to optimality. When we observe the increases in average integer program solution times when comparing Table 2 and Table 3, we may be tempted to conclude that integer program solution times also get worse as the solution gets closer to optimality. But Table 4 shows that the metrics themselves are what make the integer programs slightly harder. The average solution time when choosing J randomly for $n = 5$ is still 0.83 seconds even though we have reached a much better solution. We believe that when metrics are used, the interaction of airports and passengers which are similar causes the networks to grow more rapidly as jets are added to the neighborhood, and this generates integer programs that are harder to solve.

The final question we ask in this section is whether these metrics are effective on their own or is there some synergistic effect? To answer this question, we ran Algorithm 2 for each of the 10 instances with $n = 5$ and three new schemes for selecting J. In scheme RAD (MTC, DST) we select the base airport radius (passenger matches, distance) metric 75% of the time and choose J randomly otherwise. We also include the results from

Algorithm 1 where we always chose J randomly (RND), and the results from Algorithm 2 using the equilikely distribution (ALL) above for comparison. The results are shown in Table 5.

Table 5: Performance of individual metric schemes.

SCHEME	IMPROV	ATT	SUCC	CHG/SUCC	TIME/IP
RND	5.13	43,457	9.90	24.48	0.83
RAD	4.85	34,541	11.21	25.69	1.22
MTC	5.91	42,651	9.74	29.19	1.05
DST	5.36	40,417	11.47	23.71	1.10
ALL	6.45	39,399	13.10	25.64	1.10

While two of the individual metric schemes (MTC and DST) outperformed the basic scheme from Section 2 (RND), the best improvements came from the scheme in which metrics were chosen with equal probabilities (ALL). The appropriate mix of these metrics is highly dependent on the instance characteristics. For example, if the region is small, the base airport radius metric will be much less important. While we do not claim that the equal probability scheme is optimal, it is one that has been successful for us on many large instances and is the one we use throughout the rest of the paper.

4 Diversification

We have seen that overall improvement is a balance of three important statistics: (1) the total number of neighborhood searches attempted, (2) the percentage of neighborhood searches that were successful, and (3) the average change per success. Throughout our search, if we can switch to new neighborhoods which improve at least one of the three (without significant negative effects to the others), we expect better results. The metrics discussed in Section 3 are examples of this type of diversification. They improved the success rate and the average change per success while slightly decreasing the number of neighborhood searches. In this section, we consider other ways to modify the neighborhoods and alter these statistics favorably and therefore yield greater reductions in flight time.

4.1 Discretization Levels

We have already seen the impact of switching to larger subsets on the critical statistics. Until now, we have held the discretization level fixed at $\Delta = 1440$. To measure the effects of moving to finer discretizations in our local search scheme, we ran Algorithm 2 with the equal probability metric selection scheme for each of the 10 instances with $n = 5$ and different discretization levels. These results are shown in Table 6.

Table 6: Impact of Δ for $n = 5$.

Δ	IMPROV	ATT	SUCC	CHG/SUCC	TIME/IP
10	5.08	12,949	25.99	30.94	4.08
30	6.09	24,905	18.20	27.57	1.84
60	6.27	30,771	15.55	26.85	1.44
1440	6.45	39,399	13.10	25.64	1.10

Finer discretizations have a dramatic effect on all three statistics. While the success rate nearly doubles at the finest granularity and the average change per success improves, the average integer program solution time is four times as long. Although these results appear to discourage the use of finer discretizations, we believe that moving to finer discretizations as the search progresses might be successful. The reason for this is that as we get farther along in the search, the success rate and the average change per success decrease for each n, and the smaller neighborhoods become less attractive.

To validate the use of larger n and finer discretizations later in the scheme, we have used our best solutions from $n = 5$ with $\Delta = 1440$ after 3 hours of searching and switched to $n = 6$ with varied discretizations for the final hour of our search. The results for this final hour are shown in Table 7.

Table 7: Finer discretizations in final hour.

Δ	IMPROV	ATT	SUCC	CHG/SUCC	TIME/IP
30	6.42	3,664	10.18	19.08	3.57
60	6.52	4,536	10.25	19.55	2.72
1440	6.50	5,549	9.06	17.45	2.19

The results confirm our intuition that moving to larger neighborhoods and finer discretizations later in the search can indeed improve our average reduction in flight time. The discretization that worked best in this experiment was $\Delta = 60$, but we also improved by just switching to a larger neighborhood with $\Delta = 1440$. The results also confirm that the success rate and the average change per success are functions of search time and decrease as our solution improves. Both statistics decreased dramatically in the final hour for $\Delta = 1440$ as compared to the four hour averages in Table 3, while the average integer program solution time actually improved slightly from 2.40 to 2.19 seconds.

We do not believe discretizations of $\Delta \leq 30$ will be effective, because switching to a larger neighborhood size, i.e., switching from neighborhood size n to $n + 1$, seems to produce better results. We do believe that switching to a discretization level of $\Delta = 60$ may prove beneficial as part of a transition scheme to the next larger neighborhood size.

4.2 Time Interval Focus

We know that considering neighborhoods with more jets in the later stages of our search improves the results. Next, we explore ideas that allow us to use neighborhoods involving more jets effectively earlier in the search. To do so, we need to increase the number of neighborhood searches without significantly decreasing the success rate or the average change per success.

When creating an instance of DAFP for a subset of jets J, we construct an individual jet network (V_j, E_j) for every $j \in J$. While the number of networks grows linearly with the size of the subset J, the size of each network grows at a much quicker rate. To illustrate this, we have sampled jet subsets with sizes up to 25 and plotted the average number of edges per individual jet network for three different discretization levels. The results are shown in Figure 2. The reason for the rapid growth of the number of edges is that an individual jet network grows faster than linear in the number of requests used in its creation and that there are approximately nine requests in a jet itinerary.

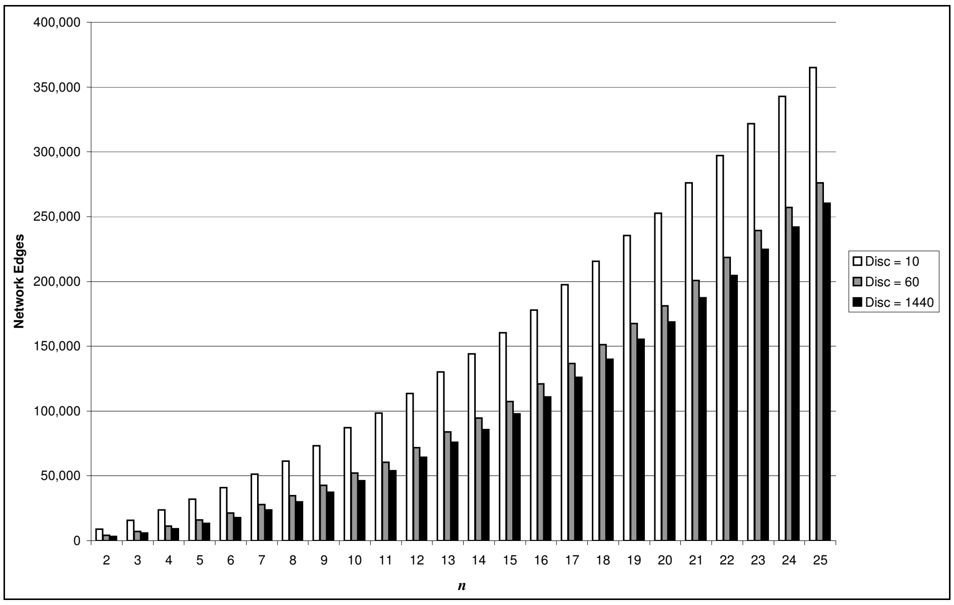

Figure 2: Network Growth

This suggests that the key to increasing the number of neighborhood searches is reducing the size of an individual jet network and that that can be accomplished by limiting the set of requests that can be served by a jet.

Let $R_{\{j\}}$ denote the set of requests served by jet j in the current schedule and let Q_j denote the set of requests that may be served by jet $j \in J$. While there are many ways to define the sets Q_j, we have chosen to do so by "freezing" the current schedule except for a certain time interval. More specifically, we choose a time interval $[e, l] \in \mathcal{T} \times \mathcal{T}$ and for all jets $j \in J$, we define Q_j to be the set of requests served by jet j in the current schedule plus all the requests that must be served in $[e, l]$, or

$$Q_j = R_{\{j\}} \cup \{r \in R_J : \text{EARLIEST}(r) \geq e,\ \text{LATEST}(r) \leq l\}.$$

As a consequence, any request in R_J that can be served outside $[e, l]$ will be served by the jet that serves it in the current schedule.

The use of the sets Q_j for $j \in J$ can easily be accommodated during network generation simply by only considering requests $r \in Q_j$ when generating the individual jet network for jet j. The metrics discussed in Section 3 can be modified appropriately when a portion of the schedule is frozen. For example, we only calculate the metrics for flights taking place within the time interval $[e, l] \in \mathcal{T} \times \mathcal{T}$.

While there are many ways to choose appropriate time intervals, the most natural thing to do is to use the schedules themselves to make this choice. We have examined the daily trends, and there are distinct morning and evening peaks each day in which an overwhelming majority of the requests are satisfied. We use two non-overlapping six-hour intervals that cover both of the peaks. The number of requests which must be serviced in each interval is roughly half of the total number of requests.

To measure the effectiveness of focusing our optimization efforts on peak time intervals, we conducted an experiment in which we randomly choose n and $[e, l] \in \mathcal{T} \times \mathcal{T}$ and use the equal probability metric selection scheme. For 80% of the neighborhoods we choose $n = 5$ and $[e, l] = [0, 1440]$. Hence most of our neighborhoods have no fixing and use the entire planning horizon. For the remaining 20% of the neighborhoods we choose $n = k$ with $k > 5$ and choose between the morning and evening time intervals with equal probability. The results, averaged over the 10 instances are presented in Table 8.

Table 8: Time Interval Focus Results

k	IMPROV	ATT	SUCC	CHG/SUCC	TIME/IP
6	6.60	40,258	12.84	26.19	1.02
7	6.74	36,374	14.11	26.93	1.09
8	6.60	31,610	15.69	27.28	1.24
9	6.67	27,571	17.53	28.30	1.39
10	6.61	23,123	19.81	29.62	1.66
12	6.35	15,911	25.21	32.48	2.53
14	5.80	10,909	29.73	36.67	4.03
16	5.05	7,532	33.15	41.49	6.14
18	4.47	5,307	35.75	48.36	8.62

When we compare the performance of the new schemes with the original scheme (shown in Table 3) we observe that the success rate and the average change per success improve for all $k > 6$ and that the improvements are getting more substantial as k gets larger. Furthermore, we see that the average integer programming solution time actually goes down for smaller values of k, which indicates an ability to solve larger subsets more rapidly.

Table 9 isolates those 20% of the searches focusing on peak time intervals. For $k = 6$,

Table 9: Isolated Time Interval Focus Results.

k	IMPROV	ATT	SUCC	CHG/SUCC	TIME/IP
6	1.03	7,450	10.54	26.97	0.54
7	1.37	6,784	14.42	28.75	0.98
8	1.68	5,866	19.54	30.11	1.71
9	1.96	5,089	24.29	32.60	2.57
10	2.18	4,306	29.42	35.34	4.03
12	2.65	2,907	44.20	42.29	8.87
14	2.63	1,888	54.71	52.14	17.87
16	2.44	1,273	58.84	66.74	30.79
18	2.27	921	55.70	90.60	44.34

we see that even though we reduce the number of requests that can be rescheduled by 50% (since only requests in the morning or the evening can be rescheduled), we get a success rate and average change per success that is comparable to the original neighborhood with $n = 5$ and we solve the integer programs in half the time. The success rate and average change per success for the largest values of k show vast improvement. The decline in overall performance at these values is due to the fact that the we use significantly more time to build graphs and the integer programs are getting harder to solve. In fact, for k the number of neighborhoods searched that ended prematurely because of the 75 second time limit was 28%.

5 Parallel Local Search

Even though the local search schemes presented in the previous sections work well, it does not appear to be the case that they converge to a local optimum within four hours. Therefore, it is natural to consider a parallel implementation to be able to search a larger portion of the search space within the four hour time limit. For discussions on the use of parallel computing in operations research see, for example, Kindervater et al. [9] and Gendron and Crainic [8]. Parallelizing the local search scheme will not be difficult because when neighborhoods are disjoint, i.e., when the sets of jets defining the neighborhoods are disjoint, we can work on them at the same time and be guaranteed that when we piece the solutions together we have a feasible schedule again.

5.1 Master/Slave Paradigm

One of the most popular paradigms used for parallel computing is that of *Master/Slave* as depicted in Figure 3. There is a single process, called the master, which functions as a task coordinator. Then there are p processes, called slaves, which communicate with the master and perform the computational tasks. Typically, the master receives the problem to be solved, intelligently partitions the work into manageable tasks, and distributes the tasks among the slaves. When the slaves complete their tasks, they report their results to the master, who decides on the next course of action based on these results.

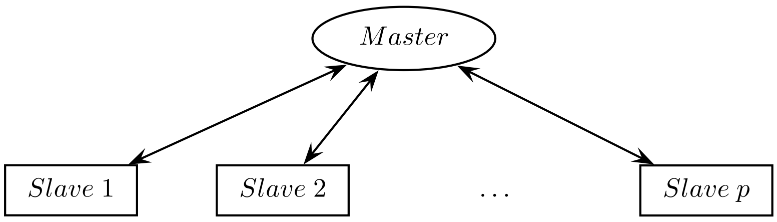

Figure 3: Master and slaves

A natural way to parallelize the local search scheme using the Master/Slave paradigm is to generate disjoint sets $J_1, \ldots, J_p \subset \mathcal{J}$ and assign each of the DAFPs defined by the sets J_i to a separate processor. Upon completion a processor sends the improved schedule back to the master. The master collects the schedules and pieces them back together. If there is time remaining, this process is repeated.

The main drawback of the parallelization scheme outlined above is its synchronous nature. To ensure feasibility, the master has to wait for all the processors to finish before generating another collection of disjoint sets $J'_1, \ldots, J'_p \subset \mathcal{J}$. This synchronization results in significant slave idle time when the solution of the integer multi-commodity network flow problem with side constraints takes much longer on one of the processors than on the others, which is rather typical. Obviously, the amount of slave idle time increases with the number of processors employed and therefore, this parallelization scheme does not scale well.

An alternative asynchronous parallelization scheme, the one we adopt, is to have each slave work on the complete schedule. In this approach, each processor executes the basic local search scheme almost independently. There are two slight modifications. First, as soon as an improved solution is found, the slave reports to the master that it has an improved schedule. If the schedule is better than the master's schedule, the processor sends its schedule to the master. On the other hand, if the master has a better schedule, it will send its schedule to the processor. Second, if after a certain time interval $\bar{t}$ (which is much smaller than the overall time limit t) the local search has not resulted in an improved schedule, the slave reports back to the master to see if the master has an improved schedule, and if so, receives that schedule. Note that the slaves always initiate communication with the master and the master's only function is to maintain the best schedule found.

Due to the asynchronous nature of this scheme, there is no slave idle time and the scheme scales without any problems. However, there is the potential for duplication of work, i.e., when slaves are working on the same set of jets and the same set of requests. We believe that especially for large instances the chance of this happening is small due to the random choices made in the local search scheme. Another potential drawback is that a slave may not be working on the best known schedule. This occurs when a slave i reports an improvement to the master while slave j is still working on its own schedule which has not yet improved. While this is unavoidable in this scheme, it's impact is minimized by keeping the time interval $\bar{t}$ small. In fact, there even seem to be benefits as this feature appears to diversify the search. We have often seen that improvements were found while working on a suboptimal schedule.

Practically, we initiate the parallel local search by starting p slave processes. Each is given the instance data and the identification of the processor where the master resides. The key to understanding the communication between the master and the slaves is to understand "awareness." The master has no idea of how many slaves there are or where they reside. All that is necessary is that the slaves know how to get in touch with the master. All communication is initiated by the slaves. When the slaves are first started, they will bombard the master processor with the message "I am available, please send me a feasible schedule." After the slave processors are started, the master is started. The master is given the instance data and the schedule produced by the reservation system, i.e., the best known schedule at the start of the solution process. When the master is contacted by a slave, it will connect with that particular slave and transfer the appropriate information based on the state of the slave.

One of the biggest benefits of this scheme is its simplicity. It is relatively easy to implement as very little synchronization and information exchange takes place. Finally, this scheme is insensitive to the number of slaves actually working on the problem. Slaves can join and leave the search without any difficulties. If a new slave is added, the master simply processes the availability message. If a slave does not answer after synchronization, the master is not affected. One can envision a grid computing like setup where all the computers available are potential slaves. If computers becomes idle, slaves are started which send availability messages to the master. If the computer is needed, the slave is just killed off with no ill affects to the overall search.

To be more precise, we will describe the communication protocol for both the master and the slaves in further detail below beginning with the slave protocol.

5.2 Slave Protocol

The slaves perform the main computational work. They are designed to coordinate themselves and they follow the basic local search scheme, choosing neighborhoods independently.

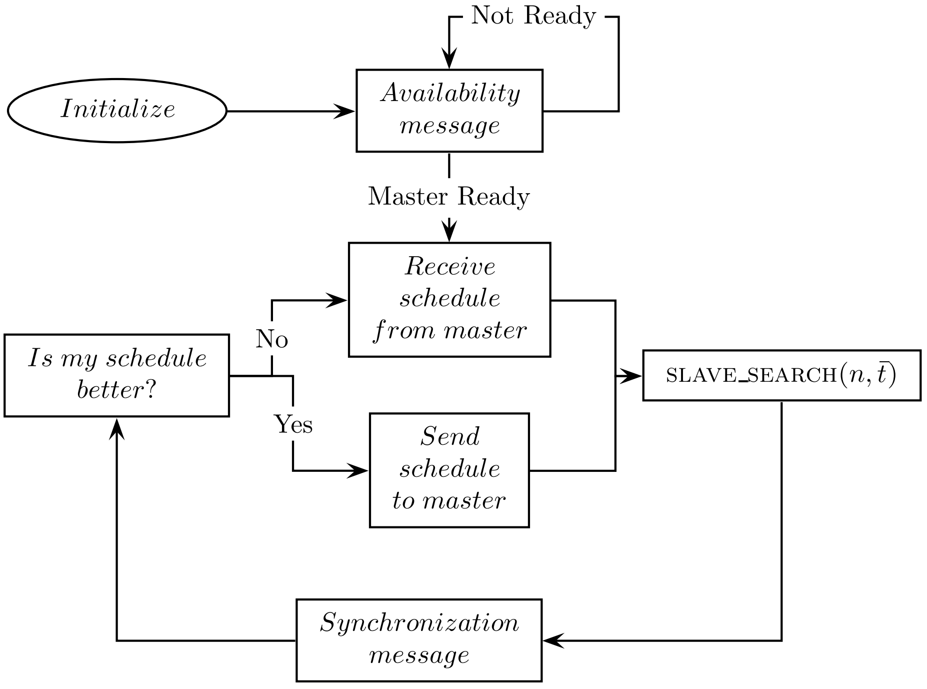

Figure 4: Slave Protocol

Figure 4 illustrates the communication protocol for the slave. Initially, the slave begins with the master location and the basic data which defines the instance, but does not possess a feasible schedule. It immediately begins broadcasting the availability message "I am available, please send me a feasible schedule." If the master has not yet been started, the slave just continues broadcasting the message over and over again until it is received by the master. Once the master receives the message and is ready to process it, the slave will connect to the master, receive the best feasible schedule, and then disconnect. Once a slave has a feasible schedule it can begin its main loop, which involves executing SLAVE_SEARCH$(n, \bar{t})$. As soon as the search finishes (either because an improved schedule has been found or the time limit has been reached) the slave broadcasts the synchronization message. Once the master receives the message and is ready to communicate with the slave, the slave will connect with the master and compare schedules. If the slave has a current schedule which is better than the best schedule possessed by the master, the master will replace its best solution with that of the slave. Otherwise the slave will replace its current schedule with that of the master. The slave will then disconnect from the master and repeat the loop indefinitely.

5.3 Master Protocol

The master serves the role of coordinator and does not perform a significant amount of computation. It will execute tasks based on messages received from the slaves. All the master does is respond to slave broadcasts and maintain the best feasible schedule found.

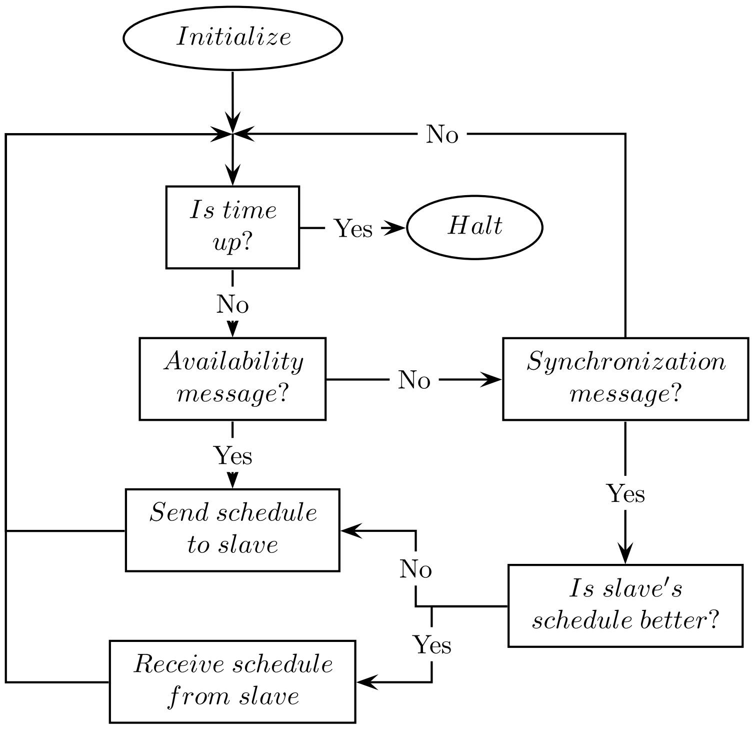

Figure 5: Master Protocol

The communication protocol for the master is illustrated in Figure 5. The master will be initialized with the basic data defining the instance and the feasible schedule from the online reservation system. The first check in the main loop is always whether the overall time limit T has elapsed. If so, the loop is halted. Otherwise, the master checks for one of the two types of broadcasts from the slaves and processes it.

The master will communicate with slaves in the order in which their broadcasts are received (first-in first-out). If the next message in the message queue is an availability message, the master will connect to the slave, send it the best feasible schedule, disconnect, and then execution will return to the top of the loop. If the next message in the queue is a synchronization message, the master will connect to the slave, compare schedules, send or receive the best feasible schedule, disconnect, and the execution will return to the top of the loop. Only improved solutions will be received from the slaves. If no messages have been received, nothing occurs and the execution returns to the top of the loop.

5.4 Value of Parallelization

We implemented the communication protocol using basic sockets in the C programming language. Our computational experiments are aimed at establishing the speed ups that can be achieved for a given local search scheme.

In the first experiment, we set $n = 5$ and use the equal probability metric selection scheme. We solved each of the 10 instances with varying numbers of processors (PROC). A synchronization interval of $\bar{t} = 200$ seconds was used for all runs. The results are presented in Table 10. We added a few new statistic: the efficiency ratio (EFF), i.e., the fraction of four hours required to achieve an improvement as least as large as the improvement obtained in four hours on a single processor, the average number of neighborhood searches a slave performed in between synchronization messages (ATT/SYNC), the average time in seconds between synchronization messages (TIME/SYNC), and the percentage of synchronization messages in which the slave had improved its solution but the master possessed a better solution (UNSUCC).

Table 10: Value of parallelization.

PROC	IMPROV	EFF	ATT	ATT/SYNC	CHG/SUCC	TIME/SYNC	UNSUCC
1	6.45	-	39,399	-	-	-	-
2	6.87	73.54	75,749	9.12	24.08	34.07	27.06
4	7.43	51.14	149,014	10.59	21.08	40.19	44.40
8	7.81	36.30	303,787	13.54	19.89	50.46	55.84
16	7.97	29.23	594,153	15.62	18.07	59.32	63.33

The value of parallelization lies in the ability to search more neighborhoods in the same amount of time. We see that we achieve an almost linear increase in the number of neighborhood searches. As the number of processors increases, we are reaching the best single processor solution faster (as indicated by the efficiency ratio) and the average reduction in flight time after four hours improves. The number of neighborhood searches and the time between successive synchronizations increase as the number of processors increases. We believe this is due to the fact that we get farther along in the search and our success rates are decreasing.

The average change per success is decreasing, which can be attributed in part to improved solutions. There is also the effect of changes to the master solution while the slaves are searching. This effect is also clearly visible in the percentage of synchronization messages in which a slave reports an improved solution, but the master already possesses a better solution. This percentage is quite high when the number of processors is large. This percentage can be controlled somewhat by keeping the time between synchronizations small.

In the second experiment, we measure the impact of the synchronization interval $\bar{t}$ on the performance of the parallel local search. We fixed the number of processors at 8 and used different synchronization intervals. The results are presented in Table 11.

Table 11: Impact of synchronization intervals.

$\bar{t}$	IMPROV	EFF	ATT	ATT/SYNC	CHG/SUCC	TIME/SYNC	UNSUCC
10	8.33	25.31	296,019	3.03	19.27	14.96	4.71
25	8.00	29.74	296,975	5.84	19.45	24.80	15.81
50	8.03	33.35	298,594	8.71	19.22	34.54	30.51
100	7.91	36.83	297,639	11.22	19.22	43.23	46.38
200	7.81	36.30	303,787	13.54	19.89	50.46	55.84

We observe that the smaller the synchronization interval, the better the improvement. With synchronization intervals less than 50 seconds, the improvement in four hours on 8 processors is greater than the improvement on 16 processors with a synchronization interval of 200 seconds (see Table 10). Note also that for $\bar{t} = 10$, we have reduced the percentage of synchronization messages in which a slave has an improved solution which is worse than the master's to 4.71%.

6 A Final Experiment

So far, we have measured the effectiveness of each of our ideas by testing its individual impact. In this final section, we use the knowledge we have accumulated to propose a comprehensive dynamic local search scheme tuned to achieve the best possible results given the four hour limit on computing time. A final computation experiment demonstrates that the scheme yields improvements of up to 11% on some instances.

First, we propose to always use metrics. They have been effective for a variety of fleet sizes, but they are most useful in large instances. If we choose two jets that are on opposite sides of the service region, the likelihood that their itineraries will allow a swap of requests is low. Hence, it makes sense to try and optimize jets which have similar itineraries and compatible requests.

Next, we propose to keep the synchronization intervals in the parallelization small.

The simplicity and flexibility of the parallel communication protocol is attractive, but allows for some duplication of work. We saw that as the number of processors increases, the likelihood of a synchronization in which a slave reports an improved solution which is worse than that of the master also increases. Keeping the synchronization interval small helps to minimize such occurrences.

We also propose a dynamic scheme in which the neighborhood size increases as the search progresses. When using metrics and parallelization, the larger neighborhoods overtake the smaller ones more quickly because their success rates and average change per success are higher. Our goal is to use, at all times, a neighborhood that produces improvements at a fast rate (see Figure 1). As improvements are harder to find later in the search, we increase the neighborhood size n later in the search. If the time out rates become too large, we increase the integer program time limit as well. The size of the largest jet subset that can be solved effectively by our core optimization technology depends on the computing environment, but we have solved instances with $n = 10$ in a reasonable amount of time.

Finally, we propose to use all diversification schemes discussed when we observe a tailing off. When that happens, we are in need of a favorable success rate and a large average change per success. Making our discretization finer and using time interval neighborhoods help us do this. It is even possible to use combinations of them to arrive at more complex neighborhood types. While these techniques allow us to work with larger subsets of jets, they can also be used to solve subsets of the same size more quickly.

We have experimented with two slightly different schemes using 16 processors. The first scheme (DYN) begins with $n = 5$ and increases the neighborhood size by one after each hour of the search. We use a discretization of $\Delta = 1440$ for the first 45 minutes of each hour and $\Delta = 60$ for the final 15 minutes. We perform neighborhood searches focusing on peak time intervals once every 15 attempts for the first two hours, and once every 10 attempts for the final two hours. In the second scheme (DYN2), we begin with $n = 3$ and increase the neighborhood size by one after each half hour of the search. A discretization of $\Delta = 1440$ for the first 15 minutes of each half hour and $\Delta = 60$ for the final 15 minutes. Neighborhoods searches focusing on peak time intervals are performed once every 4 attempts. In selecting the neighborhood size when focusing on peak time intervals, we chose the (n, k^{TW}) pairs $(3,5), (4,6), (5,7), (6,9), (7,10), (8,12), (9,13)$, and $(10,14)$ by pairing the sizes with the most similar average integer program solution times. When using pairs with $n \leq 6$, we used a synchronization interval of $\bar{t} = 10$, and $\bar{t} = 15$ was used for the larger pairs. We also doubled the integer program time limit to 150 seconds for the largest two pairs.

The results are presented in Table 12. We have also included the best results from Section 2 (RND), Section 3 (MET), and Section 5 (PAR) for comparison. Along with the overall average improvement in flight time, we present the fraction of four hours (as a percentage) required to achieve an improvement at least as large as the improvement obtained in four hours for the results in Section 2 (RND EFF) and Section 3 (MET EFF).

Table 12: Dynamic Schemes

SCHEME	IMPROV	RND EFF	MET EFF
RND	5.70	-	-
MET	6.45	59.06	-
PAR	8.33	17.67	25.31
DYN	9.42	11.63	15.70
DYN2	9.71	9.67	14.32

To illustrate the progress of each scheme, we have also plotted their average reduction in flight time at half hour intervals in Figure 6.

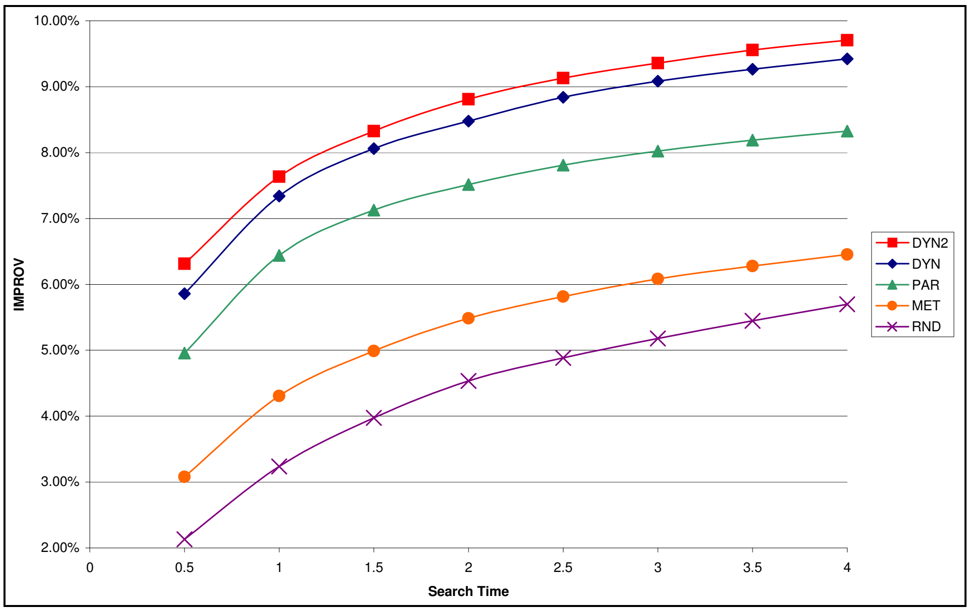

Figure 6: Progress Comparison

Both dynamic schemes have improved upon the previous results with the second scheme performing somewhat better. When comparing the results of the second dynamic scheme to the results of the basic local search scheme with $n = 3$ in Table 12, we see that we can now find the latter's solutions in a little over 23 minutes and have gained an extra 4% of overall improvement at the end of the 4 hours. If we examine the final hour in Figure 6, we see that the rates at which the dynamic searches are improving are very similar to the other searches even though we are at solutions which are much better. This is possible because we have moved to larger neighborhood sizes.

The ideas presented in Part II have helped us to obtain improvements for real world instances which are comparable to those found in Part I for much smaller fleets. We present the changes in the basic statistics for the best schedules we have found in Table 13. The overall flight time has decreased by an average of 332 hours per day, which is more than an hour per plane itinerary. Similarly, the number of legs flown has decreased by an average of 323 per day with the majority of these unnecessary legs being deadheads (96%). The most critical factor in the success of per seat, on-demand air transportation is the ability to serve multiple passengers with a single leg. Aggregating passengers is what ultimately makes this business profitable, and we have improved the average number of passengers per leg from 1.17 to 1.32.

Table 13: Change in best schedules.

DAY	LEGS	DHEADS	DIST	FLTIME
1	-342	-333	-76,750	-20,080
2	-297	-292	-80,291	-20,091
3	-348	-331	-87,527	-22,255
4	-323	-315	-73,606	-19,172
5	-359	-346	-86,338	-22,133
6	-302	-303	-74,233	-18,916
7	-273	-263	-64,617	-16,735
8	-284	-269	-66,307	-17,115
9	-336	-317	-79,099	-20,436
10	-362	-335	-85,840	-22,147

7 Conclusions

The parallel local search scheme produces schedules with good utilization factors and little deadheading and DayJet feels comfortable that with these schedules they can run a profitable business. However, the parallel local search method does not provide a performance guarantee. To address that issue, and thus to determine whether substantially better schedules may exist, we have formulated DAFP as a set partitioning problem in which rows correspond to transportation requests and columns represent feasible jet itineraries and have developed column generation techniques to solve its linear programming relaxation. Computational results on instances with up to 150 jets show that the schedules produced by the parallel local search scheme are within a few percent of the lower bound provided by the linear programming solution.

Acknowledgement

We would like to thank the DayJet corporation for financial support of this research and data, and the DayJet team consisting of Ed Iacobucci, Alex Khmelnitsky, Bruce Sawhill, Bob Spaulding, and Eugene Taits for many valuable and stimulating discussions.

References

[1] R. Agarwal, R.K. Ahuja, G. Laporte, and Z.J. Shen. 2003. A composite very large-scale neighborhood search algorithm for the vehicle routing problem. In *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, J.Y-T. Leung (ed.), Chapman & Hall CRC, 1-23.

[2] C. Archetti, M.W.P. Savelsbergh, and M.G. Speranza. *An Optimization-Based Heuristic for the Split Delivery Vehicle Routing Problem.* Submitted to Transportation Science.

[3] R. Bent and P. van Hentenryck. Randomized Adaptive Spatial Decoupling for Large-Scale Vehicle Routing Problems with Time Windows. *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI)*, 2007.

[4] DayJet Corporation. www.dayjet.com.

[5] Dayjet Corporation. 2005. How to Keep Air Transportation Moving at the Speed of Business. Whitepaper. www.dayjet.com.

[6] R. De Franceschi, M. Fischetti, P.Toth. *A New ILP-based Refinement Heuristic for Vehicle Routing Problems*, Mathematical Programming B, 105, 471-499, 2006.

[7] Espinoza, D., R. Garcia, M. Goycoolea, G.L. Nemhauser, M.W.P. Savelsbergh. *Per-Seat, On-Demand Air Transportation Part I: Problem Description and an Integer Multi-Commodity Flow Model.*

[8] B. Gendron and T.G. Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research 42*, 1042-1066, 1994.

[9] G.A.P. Kindervater, J.K. Lenstra, A.H.G. Rinnooy Kan. Perspectives on Parallel Computing. *Operations Research 37*, 985-990, 1989.

[10] S. Ropke and D. Pisinger. An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science 40*, 455472, 2006.

[11] P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP)*, 1998.