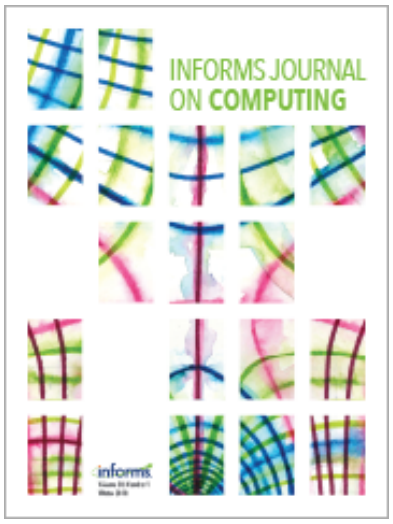

INFORMS Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Dynamic Programming-Based Column Generation on Time-Expanded Networks: Application to the Dial-a-Flight Problem

Faramroze G. Engineer, George L. Nemhauser, Martin W. P. Savelsbergh,

To cite this article:
Faramroze G. Engineer, George L. Nemhauser, Martin W. P. Savelsbergh, (2011) Dynamic Programming-Based Column Generation on Time-Expanded Networks: Application to the Dial-a-Flight Problem. INFORMS Journal on Computing 23(1):105-119. https://doi.org/10.1287/ijoc.1100.0384

INFORMS Journal on Computing
Vol. 23, No. 1, Winter 2011, pp. 105–119
ISSN 1091-9856 | EISSN 1526-5528 | 11 | 2301 | 0105

DOI 10.1287/ijoc.1100.0384

Dynamic Programming-Based Column Generation on Time-Expanded Networks: Application to the Dial-a-Flight Problem

Faramroze G. Engineer
School of Mathematical and Physical Sciences, The University of Newcastle, Callaghan, New South Wales 2308, Australia, faramroze.engineer@newcastle.edu.au

George L. Nemhauser, Martin W. P. Savelsbergh
H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30308 {george.nemhauser@isye.gatech.edu, mwps@isye.gatech.edu}

We present a relaxation-based dynamic programming algorithm for solving resource-constrained shortest-path problems (RCSPPs) in the context of column generation for the dial-a-flight problem. The resulting network formulation and pricing problem require solving RCSPPs on extremely large time-expanded networks having a huge number of local resource constraints, i.e., constraints that apply to small subnetworks. The relaxation-based dynamic programming algorithm alternates between a forward and a backward search. Each search employs bounds derived in the previous search to prune the search space. Between consecutive searches, the relaxation is tightened using a set of critical resources and a set of critical arcs over which these resources are consumed. As a result, a relatively small state space is maintained, and many paths can be pruned while guaranteeing that an optimal path is ultimately found.

Key words: dynamic programming; column generation; time-expanded networks; dial-a-flight
History: Accepted by Karen Aardal, Area Editor for Design and Analysis of Algorithms; received June 2008; revised July 2009; accepted January 2010. Published online in *Articles in Advance* May 19, 2010.

1. Introduction

Column generation is a well-known method used to solve many difficult optimization problems. As part of the column-generation pricing process, one often needs to solve a *resource-constrained shortest-path problem* (RCSPP) (Irnich and Desaulniers 2005). We consider solving the RCSPP as part of the column-generation pricing process for the *dial-a-flight problem* (DAFP). DAFP arises in the context of a per-seat, on-demand air transportation service that operates without a fixed schedule. In DAFPs, the routing of jets and passengers needs to be considered simultaneously. The time-activity networks and the associated set of resource constraints needed to represent possible jet and passenger itineraries grow rapidly with the number of airports and requests for transportation. Column generation using standard *dynamic programming* (DP) techniques for RCSPPs becomes intractable even for moderately sized instances, despite the fact that time-activity networks are acyclic and powerful dominance conditions can be developed.

In this paper, we present a novel DP-based search procedure for RCSPPs in the context of solving the pricing problem of DAFPs. The procedure exploits the network and constraint structure of the formulation to overcome some of the challenges involved with solving RCSPPs on extremely large time-expanded networks with many resource constraints. These include innovations such as the following:

- A dominance scheme that exploits the fact that certain resources are consumed only when traversing some small subset of arcs that are clustered locally within the network.
- An iterative scheme that alternates between a forward and a backward search, which gives rise to a natural bounding scheme for pruning the search, and that employs a dynamically changing and progressively tighter relaxation to control the size of the state space while ensuring that an optimal path is ultimately found.
- Preprocessing techniques that effectively take advantage of the network structure typically encountered in time-expanded networks.

A computational study using practical instances of DAFPs and the formulation proposed in Espinoza et al. (2008) demonstrates the merits of the proposed approach against standard DP techniques. Instances with up to 200 jets and 1,600 requests for transportation are considered, resulting in network represen-

tations with millions of nodes, arcs, and resource constraints. In the remainder of this paper, we formally define the problem and review relevant literature in §2, discuss the proposed solution approach in §3, and present a computational study in §4. We conclude in §5 with some general comments about the wider applicability of the proposed schemes.

2. DAFPs and Column Generation

In DAFPs, we have a set of jets $\mathcal{J}$ and a set of requests $\mathcal{R}$ for transportation from an origin airport to a destination airport. Each request represents one or more passengers. We are required to build jet and passenger itineraries that ensure that each passenger is served within a given time window, that each passenger is picked up and dropped off within a given time limit, that each passenger has at most one intermediate stop, that the seating capacity and the weight limit of each jet is never exceeded, and that the maximum flying time for each pilot during a shift is respected.

Espinoza et al. (2008) model the problem using an acyclic time-activity network $\mathcal{N} = (N, A)$, where N is the set of nodes and A is the set of arcs in which an itinerary for an individual jet, including information on the passengers served, can be represented by a path from the source node $n^s \in N$ to the sink node $n^t \in N$. Each node in the network corresponds to a point in time and space where a jet or a passenger can be located. In addition, a node may correspond to the decision to relocate the jet to another location or the decision to pick up passengers and transport them directly or indirectly to their final destination. Because the routing decisions for passengers are explicitly considered within the time-activity network, an arbitrary path from source to sink satisfies the time-window and time-duration restrictions as well as the requirement that there is at most one intermediate stop for each passenger. However, the path may violate the seating capacity and weight limit of the jet on a particular flight leg, it may exceed the maximum flying time for a pilot during a shift, and it may serve a request more than once. In the column-generation formulation of the problem, we ensure that each request is picked up exactly once in the master problem and solve RCSPPs as part of the pricing process to generate jet and passenger itineraries that are feasible with respect to the capacity, weight, and flying time restrictions and that ensure that each passenger is picked up at most once.

To account for the cost of a jet itinerary, each arc $e \in A$ is assigned a fixed cost c_e corresponding to any transportation costs that may be incurred when traversing the arc. To be able to ensure that a path defines a feasible jet itinerary, each arc $e \in A$ is also assigned integer values g_e^k $k \in \{1, \ldots, K\}$ corresponding to the consumption of various resources, such as the seating capacity and the weight limit on individual flight legs, the maximum flying time of a pilot shift, and the number of times a passenger can be picked up. The objective of the column-generation formulation is to find a minimum-cost set of paths from source to sink for each jet $j \in \mathcal{J}$, so that all requests are satisfied exactly once and the amount of resource $k \in \{1, \ldots, K\}$ consumed along each path does not exceed the amount of resource available, denoted by the integer b^k. Let $A(P)$ denote the set of arcs in path P. Then, given the set of all paths $\mathcal{P}$ from source to sink, the resulting pricing problem is

$$\min_{P \in \mathcal{P}} \ \bar{c}(P) = \sum_{e \in A(P)} \left(c_e - \sum_{r \in \mathcal{R}} \pi_r h_e^r \right) - \alpha$$
$$\text{s.t.} \ \ g^k(P) = \sum_{e \in A(P)} g_e^k \le b^k \quad \text{for } k = 1, \ldots, K.$$

Here, h_e^r is a 0–1 indicator of whether arc e corresponds to picking up the passengers associated with request r, and the dual values π_r for each request $r \in \mathcal{R}$ and α are obtained from the solution to the restricted master problem (RMP):

$$\min \ \sum_{j \in \mathcal{J}} \sum_{P \in \bar{\mathcal{P}}} c(P) \lambda_P$$
$$\text{s.t.} \ \sum_{j \in \mathcal{J}} \sum_{P \in \bar{\mathcal{P}}} h^r(P) \lambda_P = 1 \quad \forall r \in \mathcal{R},$$
$$\sum_{P \in \bar{\mathcal{P}}} \lambda_P = |\mathcal{J}|,$$
$$\lambda_P \ge 0 \quad \forall P \in \bar{\mathcal{P}}.$$

Here, $c(P) = \sum_{e \in A(P)} c_e$ is the cost of path P and $h^r(P) = \sum_{e \in A(P)} h_e^r$ is the number of times request r is satisfied along path P, and $\bar{\mathcal{P}}$ is some subset of all resource feasible paths between the source and sink. By embedding the dual costs within the arcs, we do not need to make a distinction when referring to minimum-cost and minimum-reduced-cost paths. Therefore we use these terms interchangeably throughout this paper.

Unfortunately, the size of the networks and the number of resource constraints needed to ensure feasible itineraries in this formulation grow rapidly with number of requests and airports. This growth cannot be avoided despite the fact that the networks are meticulously constructed by including only nodes and arcs that can be part of some feasible itinerary and by eliminating redundancies. Examples are including only a single representative of itineraries that differ only by departure times and including only a single representative of itineraries that differ only in the sequence in which requests are picked up for the same flight leg. This is to be expected for a time-activity network representation

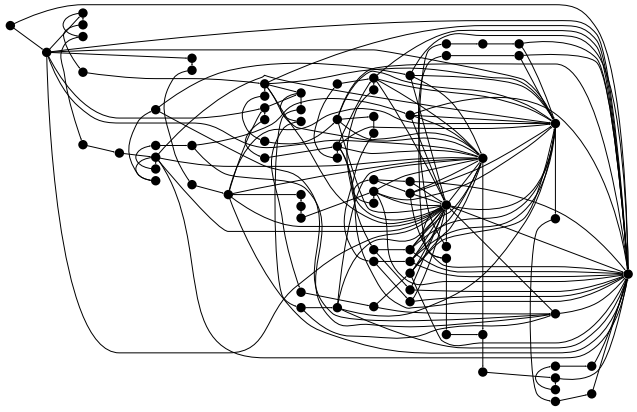

Figure 1 An Example Network Constructed for an Instance with Three Airports and Six Requests

of a nonscheduled passenger transportation system because we need to consider all possible flights and departure times as well as all possible itineraries (i.e., all routing possibilities) for each request. Figure 1 displays an example network for a single jet resulting from an instance with three airports and six requests. Even for this insignificantly small instance, the resulting network representation has 78 nodes, 167 arcs, and 45 resource constraints. Our challenge is to solve instances with 200 jets, 1,600 requests, and 40 airports.

The RCSPP is NP-hard even with acyclic networks, positive costs, and a single resource constraint with additive consumption (Garey and Johnson 1979). Several algorithms have been proposed, including approximation algorithms (e.g., Hassin 1992, Lorenz and Raz 2001), Lagrangean-based methods (e.g., Beasley and Christofides 1989, Handler and Zang 1980, Mehlhorn and Ziegelmann 2000), polyhedral approaches (e.g., Jepsen et al. 2008a, Garcia 2009), and DP algorithms such as Desrosiers et al. (1995) for modeling capacitated and time-constrained routing and Irnich and Desaulniers (2005) for general constrained shortest-path problems dealing with various types of side constraints.

DP-based algorithms are perhaps the most efficient and widely used methods for RCSPPs. Defining feasibility and dominance of partial paths forms the cornerstone of these algorithms. When resource consumption is simply additive with a single bound per resource as in DAFPs, an arbitrary path P in $\mathcal{N}$ is considered feasible if $g^k(P) \le b^k$ for $k = 1, \ldots, K$. Moreover, given two paths P_1 and P_2 from the source to node $n \in N$, P_1 dominates P_2, denoted $P_1 \preceq P_2$, if

1. $c(P_1) \le c(P_2)$, and
2. any feasible extensions of P_2 by a path from n to the sink is also a feasible extension for P_1.

Determining necessary conditions for dominance can be just as hard as solving the original problem. DP algorithms make use of reasonable sufficient conditions for dominance by identifying certain attributes of the path that together with its cost can be used to assess its quality and extensibility relative to other paths. Typically, a path is characterized by a label that contains such information as the cost of the path and the values for the chosen attributes. The attribute values associated with a path are known as a state, and the set of all possible states corresponding to feasible paths is referred to as the state space of the problem. Assuming the attribute values are discrete and bounded, maintaining only some Pareto-optimal set of nondominated labels typically leads to a pseudo-polynomial labeling algorithm for RCSPPs. Here, a feasible path P_2 is discarded if there exists an alternative feasible path P_1 that starts and ends at the same nodes and satisfies $c(P_1) \le c(P_2)$ and $g^k(P_1) \le g^k(P_2)$ for all $k \in \{1, \ldots, K\}$ because, in this case, $P_1 \preceq P_2$. We refer to Irnich and Desaulniers (2005) for details and the relative merits of various labeling schemes for RCSPPs.

Although the core components for solving constrained shortest-path problems by DP have remained mostly unchanged since the seminal work of Desrochers (1988), advances in two areas have led to significant improvements. The first of these is tighter sufficient conditions for proving dominance by exploiting specific knowledge of the problem. Examples of this are Feillet et al. (2004) for dominance conditions related to enforcing path elementarity in vehicle routing problems with time windows, Ropke and Cordeau (2006) for dominance conditions related to enforcing precedence constraints in the pickup and delivery problem with time windows, and Jepsen et al. (2008b) for dominance conditions related to integrating subset-row inequalities. The second area of improvement is augmenting the basic algorithm using various preprocessing, scaling, bounding, and search strategies. Dumitrescu and Boland (2001), for example, focus on preprocessing and label-elimination methods through resource- and cost-based bounding schemes in acyclic networks and a single resource constraint. Lübbecke (2003) also uses cost-based bounds to prune a search in the context of column generation. Another use of bounding is the bidirectional DP outlined in Righini and Salani (2006), where two concurrent searches lead to labels being pruned at some "halfway" point. Finally, Feillet et al. (2005) outline acceleration strategies that exploit information from the current solution of the column-generation relaxation to "hot start" the DP.

In the remainder of this paper, we outline a unique DP algorithm to solve RCSPPs in the context of the pricing problems encountered in DAFPs. Developments include strengthening the dominance scheme as well as various algorithmic enhancements aimed at pruning the search and maintaining a small state space.

3. Solution Approach

In this section, we present a DP-based search procedure that incorporates a number of complementary techniques for efficiently and effectively managing the size and exploration of the state space and allows the solution of RCSPPs for the large time-expanded networks encountered in DAFPs. We start by exploiting the temporal characteristics of the network structure to strengthen the dominance scheme.

3.1. Tightening Dominance

In the network representation of DAFPs, the consumption of certain resources accumulates only locally in the network, i.e., is nonzero on some subset of arcs clustered locally. Thus, using the standard sufficient conditions for dominance may lead to exploring an unnecessarily large portion of the state space. Indeed, if a given resource accumulates only locally within the network, then it is often possible to identify regions within the network such that an extension of a feasible path outside this region is guaranteed not to lead to infeasibility with respect to the given resource. In this case, one need not check dominance with respect to this resource when extending paths outside this region, which may allow elimination of paths that would otherwise not be eliminated. To capture these ideas more formally, we introduce the notion of *support* for a resource constraint.

Definition 1 (Support). For each $k = 1, \ldots, K$, we say that $N^k \subseteq N$ is a support of k in network $\mathcal{N}$ if it contains all nodes $n \in N$ for which there exists a path P_1 from the source to n and a path P_2 from n to the sink with

1. $g^k(P_1) + g^k(P_2) > b^k$, and
2. $g^k(P_1) > 0$ and $g^k(P_2) > 0$.

Thus, if $n \in N \setminus N^k$, i.e., n is not in the support of k in network $\mathcal{N}$, then for each path P_1 from the source to n and each path P_2 from n to sink, we have that either

1. $g^k(P_1) + g^k(P_2) \leq b^k$, or
2. $g^k(P_1) = 0$ or $g^k(P_2) = 0$.

We use the term *local resource constraints* when referring to resource constraints whose support is considerably smaller than the number of nodes in the network. Given a support for each resource, we can modify the criteria for identifying dominance by only considering a resource if the paths being compared end at a node in the support for that resource.

Proposition 1. *Given two feasible paths P_1 and P_2 from the source to node $n \in N$, $P_1 \preceq P_2$ whenever*

1. *$c(P_1) \leq c(P_2)$, and*
2. *$g^k(P_1) \leq g^k(P_2)$ for each $k = 1, \ldots, K$ such that $n \in N^k$.*

Because the number of nondominated feasible paths stored during the course of the DP algorithm when using the above criteria for identifying dominance depends on the number of supports containing each node, it is theoretically advantageous to compute the smallest support for each resource. This can be done for a resource $k \in \{1, \ldots, K\}$ by computing the longest path with respect to the consumption of k from the source to each node and from each node to the sink. However, it may be possible to find reasonably sized supports without having to solve auxiliary optimization problems that may become extremely time-consuming for large networks and a large number of resources. In DAFPs and the network representation proposed in Espinoza et al. (2008), for example, one can immediately determine for each node $n \in N$ and resource $k \in \{1, \ldots, K\}$ whether there exists a path P_1 from source to n and a path P_2 from n to the sink with $g^k(P_1) > 0$ and $g^k(P_2) > 0$. If so, we include n in the support of k. Because this is a weaker condition on the necessity to include a node in the support by its definition, we may not end up with the smallest support, but it may still be much smaller than $|N|$ and it is determined without much computational effort.

As a result of using a time-expanded formulation, almost all the resource constraints enforced in DAFPs are, to a certain degree, local resource constraints. Indeed, as in any time-expanded formulation, decisions at a particular time point appear contiguously along a path and decisions corresponding to different time points appear in a chronological sequence along a path. Thus, resource constraints associated with decisions at a particular time point or time window can be found clustered within the network.

In the DAFP formulation, resource constraints corresponding to seating capacity and weight limit on individual flight legs are examples of local resource constraints with very small supports. Indeed, only a small proportion of activities and associated arcs in the network correspond to transporting passengers on a specific leg given by an origin–destination pair and a departure time. Furthermore, the activities associated with transporting passengers on a specific leg appear contiguously along a path, resulting in a clustering of the arcs that contribute to the consumption of seating capacity and weight limit for a particular leg.

Resource constraints that ensure each request is satisfied at most once are also local resource constraints, albeit with much larger supports. Although any activity related to picking up a particular request contributes to the consumption of this resource, each request has a time window and, thus, the support for this resource constraint contains only nodes within the corresponding time window. Similarly, resource constraints associated with flying time restrictions on individual pilot shifts contain only the nodes corresponding to the individual shifts, with each shift being contiguous in time.

In addition to using supports to enhance dominance, we also use them to determine a sparse vector representation of resource consumption values along arcs, of the attribute values within a label, and of the bounds imposed at the nodes. This becomes crucial when dealing with the larger network formulations with millions of nodes, arcs, and resource constraints.

As a final note, we point out that using supports to enhance dominance is not the same as using resource-based bounding schemes as used, for example, in Dumitrescu and Boland (2001) to eliminate labels. Rather than exploiting the fact that a path corresponding to a particular state may not be feasibly extendable to the sink, we instead exploit the fact that a path may always be feasibly extended to the sink. In the case of resource constraints associated with satisfying requests at most once, using the respective supports to strengthen the dominance in the time-expanded network translates to not including a request as part of the dominance check if the time at the end of the path is such that the request cannot be satisfied again. This is analogous to the enhancement of the dominance scheme proposed by Feillet et al. (2004) and Ropke and Cordeau (2006) to ensure that a customer is not visited more than once in the vehicle routing problem and to enforce precedence constraints in the pickup and delivery problem.

3.2. An Arc-Based Resource Relaxation

The number of paths explored using a standard DP algorithm with the proposed dominance scheme grows rapidly with network size and number of resources and can far outpace any reasonable demand on computing power and memory, making the pricing problem intractable even when exploiting the structure provided by local resource constraints. To maintain a small state space, we incorporate relaxation ideas that are analogous in intent to the relaxation-based algorithms used for elementary shortest-path computation (Righini and Salani 2005, Boland et al. 2006).

Rather than keeping track of resource consumption over all arcs, we keep track of the consumption of a given resource only on a subset of arcs. Although this idea is related to scaling, which is a well-known technique used to project the state space of a DP onto a smaller state space, it is significantly different because the "scaling" is only performed for a subset of resources and for each of these resources only on a subset of arcs. Moreover, we use a very simple "all or nothing" scaling; namely, on a given arc, there is either no scaling of the resource consumption or the resource consumption is ignored entirely. Scaling is normally applied uniformly over all arcs (see, for example, Dumitrescu and Boland 2001). Although the reduction in the size of the state space generally leads to a reduction in the number of paths explored, this generally comes at the expense of no longer being able to guarantee feasibility of the path found. However, by judiciously choosing the resources and the arcs, we are able to reduce the size of the state space *and* ensure feasibility of the path found. We next highlight the potential benefits from using an arc-based relaxation of resource consumption by demonstrating its impact on a simple example.

In Figure 2, we attempt to construct the optimal extension of path P_0 through the subnetwork induced by the nodes in the support $N^k = \{n_1, n_2, \ldots, n_6\}$ of resource k. Alongside each arc e, we display three pieces of information: the cost c_e, the true amount g_e^k of resource k consumed when traversing the arc, and

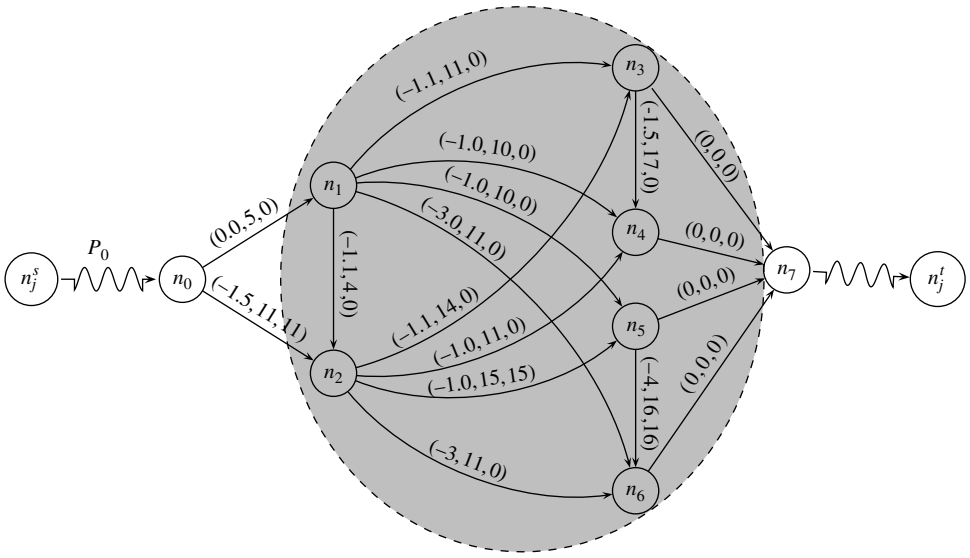

Figure 2 Reducing the Size of the State Space Using an Arc-Based Relaxation of Resource Consumption

Table 1 The Costs and Amount of Resources Consumed for All Extensions of Path P_0 Given in Figure 2 When Using Actual Values for Resource Consumption and Certain Relaxed Values

Path P	$c(P)$	$g^k(P)$	$\tilde{g}^k(P)$
$P_1 = (P_0, n_1)$	0.0	5	0
$P_2 = (P_0, n_2)$	−1.5	11	11
$P_3 = (P_0, n_1, n_2)$	−1.1	9	0
$P_4 = (P_0, n_1, n_3)$	−1.1	16	0
$P_5 = (P_0, n_2, n_3)$	−2.6	25	11
$P_6 = (P_0, n_1, n_2, n_3)$	−2.2	23	0
$P_7 = (P_0, n_1, n_4)$	−1.0	15	0
$P_8 = (P_0, n_2, n_4)$	−2.5	22	11
$P_9 = (P_0, n_1, n_2, n_4)$	−2.1	20	0
$P_{10} = (P_0, n_1, n_3, n_4)$	−2.6	33	0
$P_{11} = (P_0, n_2, n_3, n_4)$	−4.1	42	11
$P_{12} = (P_0, n_1, n_2, n_3, n_4)$	−3.7	40	0
$P_{13} = (P_0, n_1, n_5)$	−1.0	15	0
$P_{14} = (P_0, n_2, n_5)$	−2.5	26	26
$P_{15} = (P_0, n_1, n_2, n_5)$	−2.1	24	15
$P_{16} = (P_0, n_1, n_6)$	−3.0	16	0
$P_{17} = (P_0, n_2, n_6)$	−4.5	22	11
$P_{18} = (P_0, n_1, n_2, n_6)$	−4.1	20	0
$P_{19} = (P_0, n_1, n_5, n_6)$	−5.0	31	16
$P_{20} = (P_0, n_2, n_5, n_6)$	−6.5	42	42
$P_{21} = (P_0, n_1, n_2, n_5, n_6)$	−6.1	40	31
No. of feasible states (when $b^k = 40$)		19	14
No. of nondominated feasible paths		19	14

the amount $\tilde{g}_e^k$ of resource k consumed when traversing the arc in the current relaxation, i.e., g_e^k if we track resource k on arc e, and zero otherwise.

Observe that we only track resource k on the arcs in the path (n_0, n_2, n_5, n_6). Table 1 lists all possible extensions of path P_0 to nodes in N^k with their costs and the amount of resources consumed on the path. Here, we denote with $g^k(P)$ and $\tilde{g}^k(P)$ the resource consumption along path P using the actual values for resource consumption and the relaxed values for resource consumption, respectively. For simplicity, we assume that $c(P_0) = g^k(P_0) = 0$ and that the supports for any other resources do not contain nodes in N^k.

Assume that the resource limit b^k is 40. When we examine the values for resource consumption for the paths in Table 1, we see that for the relaxed resource consumption, there are five fewer states leading to five fewer nondominated paths. Moreover, the lowest cost extension, i.e., $P_{20} = (P_0, n_2, n_5, n_6)$, remains infeasible. On the other hand, any reduction in the size of the state space by scaling the resource consumption values uniformly over all arcs will lead to a relaxation in which P_{20} becomes feasible. Finally, note that the infeasible extension $P_{11} = (P_0, n_2, n_3, n_4)$ has become feasible for the given relaxation. However, because its cost is worse than the best feasible extension, the subpath (n_0, n_2, n_3, n_4) is not deemed critical for resource k.

In our example, there is only a modest reduction in the size of the state space. The reduction of the size of the state space comes primarily from paths ending at nodes n_3 and n_4. Observe that these two nodes are somewhat isolated from the arcs on the critical path (n_0, n_2, n_5, n_6). Indeed, only one of the arcs on the critical path, namely, arc (n_0, n_2), can be part of an extension of P_0 to either n_3 or n_4, resulting in only two possible values for the consumption of resource k by a path from the source to either n_3 or n_4. This suggests that a huge reduction in the size of the state space may be obtained if (1) only a relatively small number of arcs are "critical" to ensuring feasibility of the optimal path with respect to the given resource or (2) a large number of nodes in the support of a given resource are reachable from only a small proportion of these critical arcs. Thus, by only keeping track of resource consumption on arcs that are necessary to ensure that any infeasible path with cost better than an optimal solution remains infeasible, we can maximize the possibility of reducing the size of the state space while ensuring feasibility.

Given a relaxation to the problem, we next outline a DP algorithm that identifies some critical resources and arcs while producing bounds for pruning the search. An iterative scheme is then introduced that progressively tightens the relaxation and ultimately ensures that an optimal solution is found.

3.3. A Forward and Backward DP

For each arc $e \in A$ and $k = 1, \ldots, K$, let $\mathbb{I}_e^k$ be a 0–1 indicator of whether resource k is tracked in the current relaxation; i.e.,

$$\mathbb{I}_e^k = \begin{cases} 1 & \text{if we track resource } k \text{ on arc } e, \\ 0 & \text{otherwise.} \end{cases}$$

We use the terms $\mathbb{I}$-*feasible* and $\mathbb{I}$-*dominance* when referring to path feasibility and dominance when resource consumption is given by $\mathbb{I}(g_e^k) = \mathbb{I}_e^k g_e^k$ for each arc $e \in A$ and resource $k = 1, \ldots, K$. We also use the term $\mathbb{I}$-*optimal* when referring to optimality with respect to $\mathbb{I}$-feasible paths and use $P_1 \preceq_{\mathbb{I}} P_2$ to denote $\mathbb{I}$-dominance of path P_1 over P_2. Note that for each $k = 1, \ldots, K$, the support N^k constructed using actual values for resource consumption remains a support for resource k in network $\mathcal{N}$ when measuring consumption with respect to $\mathbb{I}(g_e^k)$ instead of g_e^k. Indeed, any feasible path remains feasible when relaxing resource consumption. Thus, we can use the original supports to prove $\mathbb{I}$-dominance.

Given a relaxation for RCSPPs defined by $\mathbb{I}_e^k$ for all arcs $e \in A$ and $k = 1, \ldots, K$, a DP algorithm can be used to construct an $\mathbb{I}$-optimal path from source to sink. If this path is not feasible, the relaxation can be tightened by tracking resources on additional arcs to

ensure that the same path will not be found when the DP algorithm is run again. By running the DP algorithm in the opposite direction, i.e., constructing an $\mathbb{I}$-optimal path from sink to source, and using information gathered in the forward pass, much more effective pruning can be performed. We describe the iterative process of progressively tightening the relaxation in detail in the next subsection, but we first outline the workings of the DP for a single pass and show how one can obtain bounds as a natural by-product of the relaxation and use them to significantly reduce the number of states explored.

Suppose that we have a relaxation defined by $\mathbb{I}_e^k$ for each $e \in A$ and each $k = 1, \ldots, K$, lower bounds $T(n)$ on the cost of a feasible path from node n to the sink, and an upper bound UB on the cost of a minimum-cost path from source to sink. Algorithm 1 outlines the forward DP procedure FwdDP that constructs an $\mathbb{I}$-optimal path from source to sink. The algorithm starts with the unprocessed list L containing only the trivial path rooted at the source node (denoted by (n^s)) and an empty processed list U. At each subsequent iteration, FWdDP picks a previously constructed path in L or U, it is immediately discarded. Otherwise, we remove from L and U any path that is $\mathbb{I}$-dominated by (P, e') and insert (P, e') into L for future extension. For any newly constructed path inserted into L, we update UB if the path also corresponds to a feasible (not just $\mathbb{I}$-feasible) path.

Algorithm 1 (FWdDP: A forward DP for $\mathbb{I}$-feasible RCSPPs on $\mathcal{N} = (N, A)$)

Input: $\mathbb{I}_e^k$ for all $e \in A$ and $k \in \{1, \ldots, K\}$, $T(n)$ for all $n \in N$, and UB
Initialize: $L \leftarrow \{(n^s)\}$ and $U \leftarrow \varnothing$;
while $L \neq \varnothing$ **do**
  pick $P \in L$;
  forall $e' = (\overrightarrow{n}(P), n') \in A$ **do**
    if (P, e') *is* $\mathbb{I}$*–feasible* **then**
      if $c((P, e')) + T(n') < UB$ **then**
        if $P' \not\preceq_{\mathbb{I}} (P, e')$ *for any* $P' \in L \cup U$ s.t. $\overrightarrow{n}(P') = n'$ **then**
          remove from L and U any path P' s.t. $\overrightarrow{n}(P') = n'$ and $(P, e') \preceq_{\mathbb{I}} P'$;
          $L \leftarrow L \cup \{(P, e')\}$;
          if (P, e') *is a feasible (not just* $\mathbb{I}$*-feasible) path from source to sink* **then**
            $UB \leftarrow c((P, e'))$
  $U \leftarrow U \cup \{P\}$;
Output: UB, $S(n) = \min\{c(P)\colon P \in U \text{ and } \overrightarrow{n}(P) = n\}$ for all $n \in N$ and $P^* = \arg\min\{c(P)\colon P \in U \text{ and } \overrightarrow{n}(P) = n^t\}$

For any feasible path P_2 from the source to node n such that its projected cost is less than the best feasible path found so far (i.e., such that $c(P_2) + T(n) < UB$), Algorithm 1 maintains at all times a processed list of paths U and an unprocessed list of paths L such that there exists some $\mathbb{I}$-feasible path P_1 also from source to node n such that $P_1 \preceq_{\mathbb{I}} P_2$ (P_1 may be P_2) and either

i. $P_1 \in U$, or

ii. $P_1' \in L$ and P_1 can be obtained by an extension of P_1'.

Thus the output $S(n)$ for each node $n \in N$ is a lower bound on the cost of a feasible path from source to n. Note that Algorithm 1 is similar to the one outlined in Irnich and Desaulniers (2005), with the exception that we measure feasibility and dominance with respect to the relaxation given by $\mathbb{I}_e^k$ for each k and e, and we prune the search using the bounds given by $T(n)$ for each node $n \in N$ and UB. Finally, because our networks are acyclic, we can pick unprocessed paths using a topological ordering of nodes at which they end and thus avoid the need to explicitly store and check dominance against paths that have already been processed.

FwdDP constructs paths going forward from the source node. We can similarly state a symmetrical algorithm that starts from the sink node and works its way backward, pruning the search using the bounds $S(n)$ instead of $T(n)$ for each $n \in N$. BwdDP accepts as an input lower bounds $S(n)$ on the cost of a minimum-cost path from the source to node n for each node $n \in N$ and an upper bound UB on the minimum-cost path from source to sink. BwdDP outputs the $\mathbb{I}$-optimal path P^* constructed from sink to source and bounds $T(n)$ for each node $n \in N$ for pruning the search in the opposite direction.

3.4. An Iterative DP-Based Search Procedure

Algorithm 2 describes a search procedure DPSearch that combines the relaxation described in §3.2 within an iterative DP-based search procedure with alternating search direction (i.e., alternating between FwdDP and BwdDP). In each pass of the search, we use the bounds obtained from the previous pass to prune the search in subsequent passes. Furthermore, we can ensure that the $\mathbb{I}$-optimal path P^* from source to sink, if infeasible, does not reappear in subsequent passes. Suppose P^*, the path produced by either FwdDP or BwdDP, is infeasible because $g^k(P^*) > b^k$ for some $k \in \{1, \ldots, K\}$. This can only happen if we do not track resource k on some of the arcs $e \in A(P^*)$. By tracking resource k for the appropriate arcs in P^*, we can ensure that the same infeasibility does not reappear.

Algorithm 2 (DPSearch: A DP for RCSPPs on $\mathcal{N} = (N, A)$ using FwdDP, BwdDP, and relaxation)

Initialize: $UB \leftarrow \infty$ and $LB \leftarrow -\infty$;
$S(n) \leftarrow -\infty$ and $T(n) \leftarrow -\infty$ for all $n \in N \backslash \{n^s, n^t\}$;

$S(n^s) \leftarrow 0$ and $T(n^t) \leftarrow 0$;
$\mathbb{I}_e^k \leftarrow 0$ for all $e \in A$ and $k \in \{1, \ldots, K\}$
pass $\leftarrow 0$;
while $UB - LB > \varepsilon$ **do**
  if pass *is even* **then**
    $(UB, S, P^*) = \texttt{FwdDP}(\mathbb{I}, T, UB)$;
  else
    $(UB, T, P^*) = \texttt{BwdDP}(\mathbb{I}, S, UB)$;
  $LB \leftarrow (P^*)$;
  if P^* *is infeasible* **then**
    $\mathbb{I}_e^k \leftarrow 1$ for one or more arcs $e \in A$ and one or more resources $k \in \{1, \ldots, K\}$ so that P^* is no longer $\mathbb{I}$-feasible;
  pass $\leftarrow$ pass $+ 1$;
Output: P^*

The performance of Algorithm 2, in terms of speed and memory usage, depends greatly on the initial relaxation and the degree of tightening after each pass. On the one hand, we can start with a complete state space relaxation (i.e., no tracking of any resource), and in subsequent passes track only one of the resources that caused infeasibility of P^* and only on the relevant arcs in $A(P^*)$. In this case, we increase the chance of a node in the support being isolated (reachable) from the arcs on which resources are being tracked, leading to a small state space. However, such a strategy may require a large number of passes. Furthermore, with such a strategy, the bounds available for pruning the search in consecutive passes will only improve slightly. On the other hand, we can refine the relaxation by tracking several resources in addition to all of those that caused infeasibility of P^* and on arcs other than the ones in $A(P^*)$. In this case, we are likely to need far fewer passes at the possible expense of potentially exploring a larger state space in each of them but with stronger bounds for pruning the search in each of them as well. The resources to track in subsequent passes and the arcs on which to track them have to be chosen so as to (1) ensure that the infeasible $\mathbb{I}$-optimal path of the current pass does not reappear and (2) balance the number of passes and the effort expended per pass. A judicious choice may result in faster solution times and less memory use. In our computational study, we experiment with different choices to determine the relative merits of tightening the relaxation over many resources and arcs versus tightening the relaxation more selectively over a few resources and a small set of arcs.

It is important to note that the search procedure outlined in Algorithm 2 is not a bidirectional search as described in Righini and Salani (2006). Rather than having two concurrent searches in which we can only fathom paths reaching some predefined halfway point, we alternate between forward and backward passes, allowing us to compute bounds to start pruning at any stage in the search. Also, our procedure is not simply an extension of the scaling algorithm given in Dumitrescu and Boland (2001) to multiple resources. In addition to obtaining bounds and pruning the search by alternating search directions, we are able to vary the size of the state space over the network by allowing refinement of the relaxation to specific resources and arcs. Furthermore, our bounds for pruning are obtained and strengthened as a natural by-product of the relaxation scheme and refinement process without having to solve auxiliary optimization problems.

3.5. A Path Completion Heuristic

A drawback of the scheme we just described is that we do not obtain an optimal solution until close to termination, and in general, only obtain a handful of other feasible paths in the process. With this in mind, we next describe a simple heuristic that can be used within the DP algorithm. The benefit of this heuristic is twofold: not only does it give us a simple yet efficient means of populating RMP with more than a single column per pricing iteration, it also serves to improve the pruning within FwdDP and BwdDP by improving the upper bound UB sooner.

In FwdDP, when inserting a newly constructed $\mathbb{I}$-feasible nondominated path P into the unprocessed list, we backtrack over P to check if it is feasible (not just $\mathbb{I}$-feasible) and, if so, try to extend P to the sink via a greedy depth-first search (GDFS). The greediness of the algorithm stems from our choice of the arc to explore next. Consider an iteration of GDFS in which we are attempting to extend path P. Let $\{e_1 = (\overrightarrow{n}(P), n_1), e_2 = (\overrightarrow{n}(P), n_2), \ldots, e_q = (\overrightarrow{n}(P), n_q)\}$ be the subset of outgoing arcs for which an extension of P has not yet been explored in GDFS. Of these possible choices to extend P, we pick arc e_i such that $c_{e_i} + T(n_i) \le c_{e_j} + T(n_j)$ for all $j = 1, \ldots, q$. Extending the search by outgoing arc e_i is more likely (from the point of view of the bounds) to result in a path that improves UB than any other arc e_j for $j \in \{1, \ldots, q\} \setminus \{i\}$. If the extension (P, e_i) is infeasible, then we backtrack and iterate. Similarly, if $c((P, e_i)) + T(\overrightarrow{n}((P, e_i))) > UB$, then we also backtrack. However, in this case, we can backtrack twice before continuing the search because, by our choice of arcs to explore next, we also have $c((P, e_j)) + T(\overrightarrow{n}((P, e_j))) > UB$ for all $j = i + 1, \ldots, q$.

It is impractical to perform this search for every new path inserted into the unprocessed list and continue the search exhaustively until the best possible extension is found. Instead, we first check if path P has the potential to improve UB significantly, i.e., $c(P) + T(\overrightarrow{n}(P)) < LB - \theta(UB - LB)$ for some $0 < \theta < 1$, and stop after a certain number of extensions to the

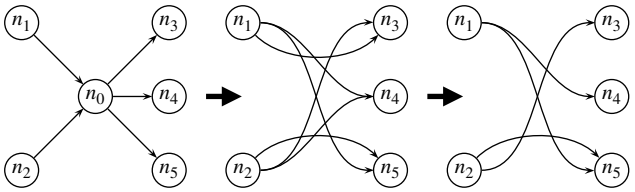

Figure 3 **The Aggregation Algorithm: Node n_0 Is Aggregated; the Resulting Paths (n_1, n_0, n_3) and (n_2, n_0, n_4) Are Infeasible and Discarded**

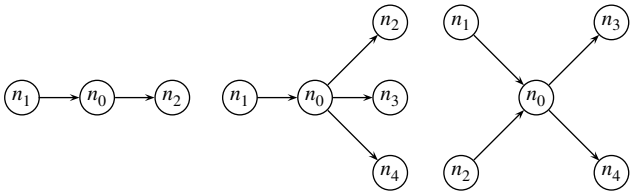

Figure 4 **Network Substructures Ideal for Aggregation**

sink or after finding an extension that significantly improves *UB*. A symmetrical procedure can be used to extend newly constructed paths to the source node in the backward pass.

3.6. Network Preprocessing

We next describe the use of aggregation, a preprocessing scheme that attempts to remove certain infeasible paths from the network. Aggregation was originally introduced in Espinoza et al. (2008) on a much smaller scale to strengthen a multicommodity flow formulation. Although aggregation does not strengthen the column-generation formulation, it has a significant impact on the tractability of the pricing problem.

The process of aggregation is quite simple: we replace a node and its incoming and outgoing arcs with arcs from the tail of each incoming arc to the head of each outgoing arc as shown in Figure 3. When aggregating a node n and replacing an incoming arc e_1 and outgoing arc e_2 with a new arc e, the resource consumption along e_1 and e_2 is simply consolidated into e, i.e., $g_e^k = g_{e_1}^k + g_{e_2}^k$ for all $k \in \{1, \ldots, K\}$. When this process is applied iteratively, the newly added arcs correspond to paths in the original network and thus may be infeasible in terms of the resources accumulated and need not be added.

Each time a node is aggregated, we remove this node from the network but may add quadratically more arcs than we remove. The key, therefore, is in the selection of the nodes to aggregate. Time-expanded networks usually contain substructures like forks, paths, and small knots that have many nodes with relatively small indegree or outdegree (see Figure 4) and thus are ideal candidates for aggregation. In our implementation of aggregation, we iterate until no such substructures can be found; i.e., a node n is aggregated if $\min\{\delta^{\text{in}}(n), \delta^{\text{out}}(n)\} \leq \gamma$, where $\gamma \geq 1$ is a parameter that controls the worst-case factor of increase in the number of arcs.

We note that reducing the number of nodes without increasing the number of arcs may in itself speed up the solution of RCSPPs, but a significant speedup only occurs when nodes that are part of a large number of supports are removed.

4. Computational Experiments

In this section, we present computational experiments to evaluate the efficiency of the proposed scheme for solving RCSPP-based column generation for DAFPs. We use practical instances provided by the DayJet® Corporation. We highlight some characteristics of the network formulation that make RCSPP-based pricing particularly difficult before moving onto the experiments that evaluate the impact of various algorithmic choices.

Test Instances and Network Characteristics. We test our algorithms on a total of 50 instances ranging from 10 jets and approximately 60 requests up to 200 jets and approximately 1,600 requests. Each instance is characterized by the number of jets, the number of requests, the number of airports, and the actual day for which the schedule is to be constructed. We denote, for example, by $(100J, 1{,}200R, 40P, 10)$ an instance for day 10 with 100 jets, 1,200 requests, and 40 airports. In each case, there is a homogeneous fleet of jets based at the same airport with seating capacity for three passengers. All time windows and flying times are stated in minutes, and each instance represents the daily operation over two overlapping shifts spanning a 16-hour day. The objective in each case is to minimize the total flying time (and thus fuel burn).

Table 2 gives the size of the network and various statistics relating to the number of resource constraints and the size of supports for some of the instances both before and after aggregation. The first column lists the instance, and the remaining columns report various statistics before and after aggregation, including the number of nodes and arcs in the network and statistics related to the different types of resource constraints. The columns labeled "Capacity/weight," "Flying time," and "Requests" correspond to the types of resource constraints within DAFPs, i.e., enforcing the jet seating capacity weight limit on some flight leg, enforcing a maximum flying time for some shift, and ensuring that some request is not picked up more than once. For each instance before and after aggregation and for each type of resource constraint, we display three pieces of information:

1. The number of constraints enforced,

Table 2 Network Characteristics for Select Instances Before and After Aggregation

	Before aggregation					After aggregation				
			Type of resource constraint					Type of resource constraint		
Instance	$\|N\|$	$\|A\|$	Capacity/ weight	Flying time	Requests	$\|N\|$	$\|A\|$	Capacity/ weight	Flying time	Requests
(10*J*, 15*P*, 60*R*, 10)	223,860	374,370	12,0972	2	60	8,925	110,907	71,951	2	60
			2.54	111,930.00	33,389.35			1.28	4,462.50	2,148.87
			1.37	1.00	8.95			10.32	1.00	14.45
(25*J*, 20*P*, 173*R*, 10)	518,197	970,492	249,318	2	173	28,449	379,524	173,554	2	173
			3.00	259,098.50	60,838.66			1.52	14,224.50	5,178.71
			1.44	1.00	20.31			9.26	1.00	31.49
(50*J*, 30*P*, 356*R*, 10)	1,238,059	2,499,362	527,273	2	356	85,044	1,142,065	418,624	2	356
			3.55	619,029.50	157,285.83			1.74	42,522.00	16,372.77
			1.51	1.00	45.23			8.55	1.00	68.54
(75*J*, 30*P*, 559*R*, 10)	1,942,265	4,109,667	663,754	2	559	145,997	2,040,478	563,717	2	559
			4.69	971,132.50	240,461.92			2.28	72,998.50	27,718.89
			1.60	1.00	69.21			8.80	1.00	106.13
(100*J*, 35*P*, 751*R*, 10)	2,788,465	6,215,328	844,351	2	751	236,137	3,330,594	756,251	2	751
			5.44	1,394,232.50	350,087.97			2.60	118,068.50	44,599.82
			1.65	1.00	94.29			8.32	1.00	141.84
(125*J*, 41*P*, 956*R*, 10)	3,769,121	8,666,698	997,749	2	956	337,430	4,892,044	937,112	2	956
			6.38	1,884,560.50	489,921.74			2.99	168,715.00	66,357.31
			1.69	1.00	124.26			8.30	1.00	188.00
(150*J*, 41*P*, 1,172*R*, 10)	4,596,596	10,804,952	1,052,891	2	1,172	408,967	6,202,019	1,002,295	2	1,172
			7.53	2,298,298.00	578,725.08			3.47	204,483.50	77,774.37
			1.73	1.00	147.59			8.51	1.00	222.88
(175*J*, 41*P*, 1,364*R*, 10)	5,563,628	13,329,286	1,112,357	2	1,364	502,505	7,754,169	1,066,226	2	1,364
			8.79	2,781,814.00	688,846.93			4.04	251,252.50	92,926.22
			1.76	1.00	168.88			8.58	1.00	252.24
(185*J*, 41*P*, 1,448*R*, 10)	5,898,199	14,279,623	1,126,225	2	1,448	544,167	8,392,791	1,083,647	2	1,448
			9.25	2,949,099.50	726,773.50			4.25	272,083.50	100,683.83
			1.77	1.00	178.42			8.47	1.00	267.91
(200*J*, 41*P*, 1,547*R*, 10)	6,404,692	15,637,938	1,141,329	2	1,547	584,782	9,234,549	1,100,796	2	1,547
			9.99	3,202,346.00	777,668.60			4.58	292,391.00	106,944.50
			1.78	1.00	187.84			8.61	1.00	282.91

2. The average size of the supports, and
3. The average overlap of supports, i.e., the average number of supports that contain the same node.

The number of constraints required to enforce the jet capacity and the weight limit is exactly the number of flight legs within the network. Although there is a huge number of these constraints (over a million for the larger instances), the average size of the supports is extremely small (fewer than 10 nodes). Furthermore, the average overlap between these supports is never more than two; i.e., requests for transportation can be accommodated by considering on average two possible flight legs for each request.

In contrast, we need only two constraints to enforce the restrictions on flying time, one for each of the two shifts. Each node in the network falls into exactly one shift and nodes belonging to the same shift are contiguous over a path within the network. Thus, the average size of these supports is half the number of nodes in the network, and the average overlap between these supports is one.

Finally, we observe that the number of constraints required to enforce that a request is not picked up more than once is exactly the number of requests. Here, a node is included as part of the support for not picking up a request more than once if the request can be part of an itinerary going through that node and still satisfy the time window of the request. The average overlap between such supports suggests that, on average, a request has the opportunity to be consolidated on the same flight leg with many other requests.

In summary, the instances and networks used to test our algorithms represent a diverse set of characteristics from network size to the number of resource constraints and the size of the respective supports.

4.1. Computational Results

Our algorithms were implemented in C using the CPLEX 9.1 barrier optimizer as the LP solver. Furthermore, all experiments were conducted on 2.4 GHz Dual AMD 250 processors with 4 GB of RAM each.

Impact of Aggregation on Network Characteristics. Figure 5 shows the network from Figure 1 after aggregation. Notice how aggregation dramatically reduces the number of nodes in this example and that the majority of resulting arcs are parallel and

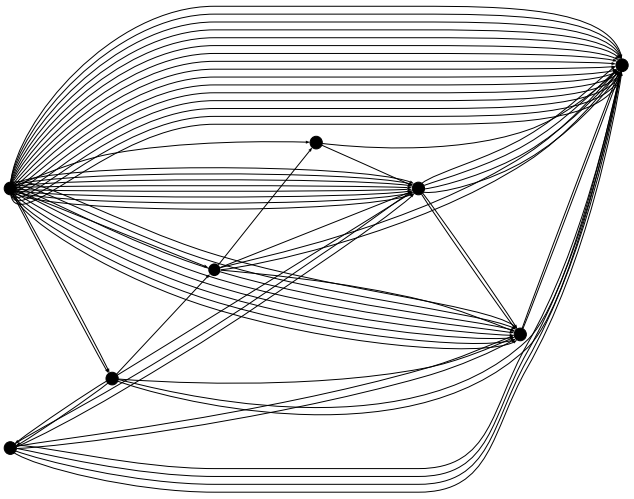

Figure 5 The Result of Aggregation on the Network Shown in Figure 1

correspond to paths in the original network. Aggregation for DAFPs can be interpreted as grouping together sequences of "likely" decisions in the form of arcs within the aggregated network. The nodes that are not aggregated correspond to crucial decision epochs at which point the course of an itinerary can change significantly. As we see next, the remaining unaggregated nodes tend to be part of many supports compared to the aggregated nodes, and thus, the reduction in network size does not directly translate to increasing problem tractability by the same amount.

Table 2 shows for select instances the network characteristics before and after aggregation. Observe that aggregation reduces the number of nodes in the network by a factor of at least 10 but the number of arcs also decreases because many sequences of arcs can be eliminated based on resource considerations. At the same time, the average overlap between supports of the capacity and weight constraints increases by a factor of about 8 for the smaller instances and a factor of about 5 for the larger instances. The average overlap between the two supports corresponding to the flying time restriction remains the same while the average overlap between supports corresponding to picking up a request no more than once increases by a factor of less than 2.

The increase in the average overlap of the capacity and weight constraints suggest that the remaining unaggregated nodes are on average contained in the supports of five to eight times more resources related to capacity and weight than the aggregated nodes. Thus although the number of nodes are reduced by more than a factor of 10, overall, aggregation has reduced the number of nodes in the supports of capacity and weight constraints by a factor of about 1.25 for smaller instances and by a factor of about 2 for the larger instances. Because the overlap for the flying time constraints remains the same after aggregation, the reduction of the number of nodes in the supports is directly correlated to the reduction in the overall number of nodes. Similarly, because the overlap between supports for constraints that ensure a request is satisfied at most once increases by less than 2, the reduction of the number of nodes in the supports is more than half the reduction in the overall number of nodes.

In summary, through aggregation alone, we can expect to see an order of magnitude decrease in the number of nondominated paths stored at any given time with a similar speedup in solution times. We quantify this gain in efficiency through aggregation in the experiments that follow.

Impact of Aggregation and the Proposed Relaxation Scheme on the Tractability of the Pricing Problem. We attempted to solve the column-generation relaxation for each of the 50 instances of DAFPs. Each instance is solved using networks with and without aggregation and using three different schemes, including two that use an arc-based relaxation of resource consumption.

The first scheme, the control experiment denoted by S^0, corresponds to a standard DP algorithm run with the proposed dominance scheme (see §3.1) but without any relaxation, thus guaranteeing the minimum-cost path to be found in a single pass.

The second scheme, denoted by S^1, corresponds to a strategy in which we start with a complete state space relaxation (i.e., we set the consumption of all resources on all arcs to zero) and we tighten the relaxation by selecting one resource k for which the $\mathbb{I}$-optimal path is infeasible. We also track the consumption of resource k on all the relevant arcs in the infeasible $\mathbb{I}$-optimal path in all subsequent passes. In fact, if resource k corresponds to a capacity, a weight, or a flying time constraint, then we strengthen the relaxation even further and track resource k including *all* other resources of the same type on *all* relevant arcs, not just those in the infeasible $\mathbb{I}$-optimal path, in all subsequent passes. The reason we do not choose to do the same for resource constraints corresponding to picking up a request at most once is that these constraints have a large overlap of supports, whereas the overlap between capacity and weight constraints is much smaller. If there are several resources for which the $\mathbb{I}$-optimal path is infeasible, then we select resource k based on the frequency with which we encounter infeasibilities of this type throughout the column-generation process. Thus, if capacity is the most frequently violated resource constraint on the $\mathbb{I}$-optimal paths encountered so far, then we always give preference to refining the relaxation with respect to capacity first. Note that in S^1, we refine

the relaxation over a large number of resources and arcs. Thus, scheme S^1 evaluates the merits of the proposed DP scheme when the mechanism for managing the number of paths being explored is biased toward finding stronger bounds for pruning the search over maintaining a small state space.

The third scheme, denoted by S^2, corresponds to a strategy in which we proceed as in scheme S^1 until we detect that the number of paths being explored is growing too rapidly during some pass (and we may thus exhaust the available memory). In this case, we undo any refinement of the relaxation done at the end of the previous pass and restart by tracking infeasibilities only on paths that provide the bounds used for pruning the search in the current pass. More precisely, given the set of processed paths U at the end of the previous pass, we identify the set of paths $\mathcal{P}' \subseteq U$ such that for all $P \in \mathcal{P}'$, $g^k(P) > b^k$ for some k, and $c(P) = S(\overrightarrow{n}(P))$ if the previous pass is FwdDP, or $c(P) = T(\overleftarrow{n}(P))$ if the previous pass is BwdDP. That is, $\mathcal{P}'$ is the set of all infeasible $\mathbb{I}$-optimal paths from either the source to all nodes or from all nodes to the sink. For each path $P \in \mathcal{P}'$, we select one resource k for which P is infeasible and track the consumption of resource k on all relevant arcs in $A(P)$ and in all subsequent passes. (Note that for a local resource constraint, the number of "relevant arcs" in $A(P)$ may be quite small.) As before, if there are several resources for which P is infeasible, then we select resource k based on the frequency with which we encounter infeasibilities of that type. In any subsequent pass, we continue to refine the relaxation by tracking the consumption of a resource on the relevant arcs of an infeasible minimum-cost path from either the source to some node or from some node to the sink.

We keep a record of the number of unprocessed nondominated paths stored at any given time within the DP and use this as an indicator of whether the number of states explored is growing too rapidly for the progress made in the DP and, if need be, switch strategies as prescribed in scheme S^2. Once the switch is made, it may require several further passes before the optimal solution is found. To avoid the algorithm stalling, we limit the total number of passes during each pricing iteration. Experiments using scheme S^2 with a limit of 10 passes produced good results. At the 10th pass, we simply revert to tracking all resources on all arcs and thus are guaranteed to find the optimal path. Of course, we still have the bounds from previous passes to prune the search. Hence, although a significant proportion of pricing iterations actually reach this limit, we find that extending this limit serves only to slow the overall pricing and does not allow us to solve larger instances. Scheme S^2 thus is designed to be a compromise between using fewer passes and providing stronger bounds for pruning the search at the possible expense of a larger state space versus having a much smaller state space at the expense of providing weaker bounds for pruning the search and potentially requiring many more passes.

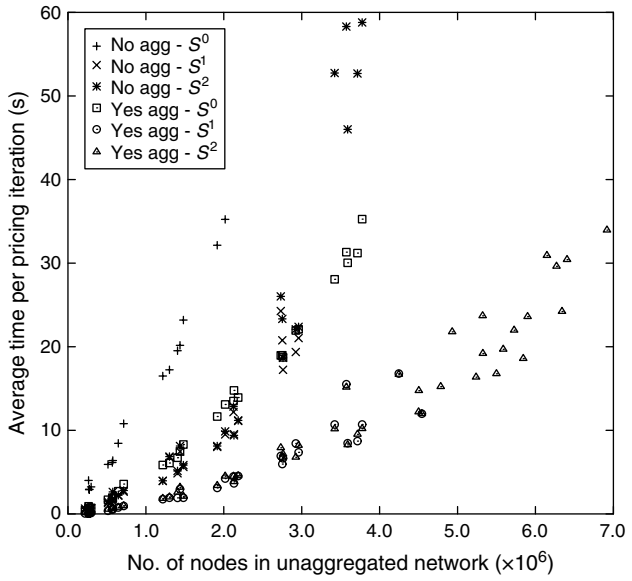

Figure 6 Average Time per Pricing Iteration Spent in Pricing Columns

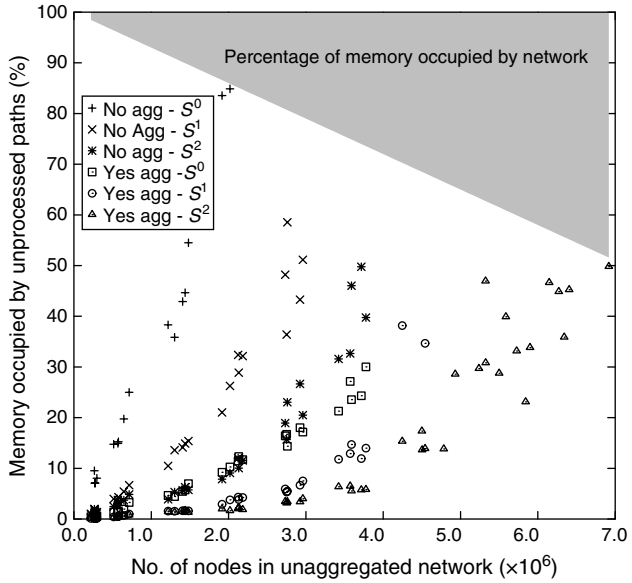

Figure 7 Percentage of Memory Used by the DP Algorithm

Figures 6 and 7 display the average time per pricing iteration (in seconds) and the percentage of available memory used during the DP algorithm as a function of the number of nodes in the original network representation for each of the 50 instances when solving the column-generation relaxation using networks with and without aggregation and using the 3 schemes just described. The shaded region in Figure 7 roughly corresponds to the amount of available memory that is occupied by the networks themselves. Both Fig-

ures 6 and 7 demonstrate the value of the two main ideas, i.e., network aggregation and arc-based relaxation. Aggregation allows us to handle much larger networks without a noticeable increase in the average time per pricing iteration (the largest instances solved with S^0 without aggregation have about 2 million nodes and about 4 million nodes with aggregation, with the average time per iteration at about 35 seconds in both cases). Careful resource relaxation allows us to handle much larger networks without exhausting the available memory. With aggregation, S^0 runs out of memory for networks with 2 million nodes, S^1 runs out of memory for networks with 4.5 million nodes, and S^2 runs out of memory for networks with 7 million nodes.

Table 3 summarizes the results over all 50 instances (10 different sizes and 5 instances per size) and the 6 experiments (the 3 schemes with and without aggregation) for each instance. In Table 3, the first column gives the number of jets in the instance, the second column gives the number of airports, the third column gives the number of requests (as a range), and the fourth column gives the number of instances with the specified number of jets, airports, and requests. The next six columns summarize the results for six experiments carried out on each of these instances. The first three experiments correspond to the three schemes described when network aggregation is not performed, whereas the last three experiments correspond to the three schemes described when network aggregation is performed. Each cell of the last six columns presents six pieces of information:

1. the number of instances that were solved,
2. the average time spent in pricing (in seconds),
3. the average number of pricing iterations,
4. the average number of passes per pricing iteration,
5. the average number of paths stored at any given time, and
6. the average of the maximum number of paths stored at any given time.

Averages are taken over the solved instances only. Note that we do not impose any limits on solution time. Thus, experiments that failed to terminate exhausted the available memory.

From Table 3, we see that using schemes S^1 and S^2 is on average at least three times faster than the standard DP scheme S^0 and that using them in conjunction with aggregation makes them two to three times faster again. The impact on the average and maximum number of paths stored at any given time follows a similar pattern. Although there is not much to separate schemes S^1 and S^2 in terms of pure speed, we notice that the average and worst-case number of paths stored at any given time is less for S^2 than for S^1 and the difference grows with problem size. The true value of scheme S^2 is not evident until we examine the larger instances. The exponential growth of the state space becomes the main obstacle to solving these instances. It is clear that not only does it pay to use a relaxation scheme and prune the search using the bounds it provides, but by judiciously selecting the arcs and resources we track once the state space hints at blowing up, we can solve much larger pricing problems.

In summary, standard DP techniques with the proposed dominance scheme allow us to consistently solve instances with up to 75 jets and 585 requests, resulting in networks with approximately 2 million nodes, 4 million arcs, and over a million local resource constraints. With the addition of aggregation, we can increase this to instances with 125 jets and 966 requests, resulting in networks with approximately 4 million nodes, 9 million arcs, and over 2 million local resource constraints. Finally, using an arc-based relaxation of resource consumption and a strategy that starts with the aim of producing strong bounds for pruning the search and later switches to a more conservative strategy in terms of choosing which arcs and resource to track, we are able to solve the column-generation relaxation for instances with up to 200 jets and 1,613 requests, resulting in networks with approximately 7 million nodes, 16 million arcs, and over 4 million local resource constraints.

The schemes described here are only a few of the strategies with which we experimented. Instead of starting with S^1 and switching to a more conservative strategy as prescribed in S^2, we could start with the conservative strategy right from the start. Of course, with such a conservative scheme one would expect many more passes and refinements per pricing iteration but fewer paths stored at any given time. However, as with S^2, we need to impose a limit on the number of passes to stop the algorithm from stalling. We found that with this extremely conservative approach, the algorithm almost always reached the limit on the number of passes even when we extended the limit to several hundred per pricing iteration. Furthermore, because the bounds available for pruning are much weaker, the algorithm actually fared much worse than schemes S^1 and S^2 with respect to the number of instances that could be solved.

We also experimented with strategies in which we kept a history of arcs and resources that often contribute to infeasibility and used them to initialize the first pass or in addition to the arcs that contribute to the infeasibility in the current pass. We also considered the idea of initializing the tracking of resources on those arcs with favorable reduced cost. In each of these cases, we noticed a moderate speedup of the pricing process but could not solve instances larger

Table 3 A Summary of the Impact of Aggregation and Using an Arc-Based Relaxation of Resource Consumption on Solution Time and Problem Tractability

				Aggregation					
				No			Yes		
				Scheme			Scheme		
No. of jets	No. of ports	No. of requests	No. of instances	S^0	S^1	S^2	S^0	S^1	S^2
10	15	[60, 80]	5	5/5	5/5	5/5	5/5	5/5	5/5
				211	36	44	43	6	9
				69	69	70	65	74	67
				1.00	2.25	3.95	1.00	2.12	3.58
				760,088	145,349	133,879	60,944	8,175	7,167
				888,736	179,598	157,874	73,846	15,216	14,577
25	20	[173, 199]	5	5/5	5/5	5/5	5/5	5/5	5/5
				1,139	287	325	355	99	113
				147	146	147	145	148	147
				1.00	2.55	4.05	1.00	2.33	3.86
				1,681,698	336,950	258,609	178,164	31,180	21,566
				2,151,918	580,111	408,219	227,436	74,947	59,541
50	30	[350, 369]	5	5/5	5/5	5/5	5/5	5/5	5/5
				4,855	1,536	1,609	1,641	524	576
				250	269	271	238	248	245
				1.00	2.78	5.09	1.00	2.48	4.74
				3,979,851	730,913	485,845	462,409	73,908	61,847
				5,188,611	1,643,206	645,692	654,087	192,071	172,599
75	30	[543, 585]	5	2/5	5/5	5/5	5/5	5/5	5/5
				12,786	4,538	4,654	5,232	1,548	1,612
				378	452	456	378	452	456
				1.00	2.88	5.23	1.00	2.75	4.97
				6,488,540	1,116,550	681,089	876,936	117,424	71,405
				10,106,232	3,377,045	1,212,930	1,319,072	459,434	234,420
100	35	[746, 779]	5	0/5	5/5	5/5	5/5	5/5	5/5
				—	14,714	16,034	9,872	3,549	3,568
				—	708	707	490	487	485
				—	2.87	5.52	1.00	2.51	5.23
				—	1,774,384	970,042	1,263,916	220,058	90,865
				—	5,700,995	2,515,464	1,983,181	742,298	416,281
125	41	[934, 966]	5	0/5	0/5	5/5	5/5	5/5	5/5
				—	—	33,882	18,618	6,474	6,315
				—	—	629	597	596	588
				—	—	6.12	1.00	2.57	5.52
				—	—	1,646,765	1,889,344	332,310	108,352
				—	—	4,794,583	3,031,568	1,567,290	718,551
150	41	[1,135, 1,182]	5	0/5	0/5	0/5	0/5	2/5	5/5
				—	—	—	—	10,511	9,935
				—	—	—	—	737	703
				—	—	—	—	2.68	5.64
				—	—	—	—	432,275	278,319
				—	—	—	—	4,370,936	1,776,275
175	41	[1,312, 1,382]	5	0/5	0/5	0/5	0/5	0/5	5/5
				—	—	—	—	—	15,778
				—	—	—	—	—	850
				—	—	—	—	—	5.97
				—	—	—	—	—	440,126
				—	—	—	—	—	3,382,715
185	41	[1,385, 1,487]	5	0/5	0/5	0/5	0/5	0/5	5/5
				—	—	—	—	—	20,456
				—	—	—	—	—	899
				—	—	—	—	—	6.31
				—	—	—	—	—	519,105
				—	—	—	—	—	4,552,260
200	41	[1,516, 1,613]	5	0/5	0/5	0/5	0/5	0/5	4/5
				—	—	—	—	—	31,382
				—	—	—	—	—	1,004
				—	—	—	—	—	6.97
				—	—	—	—	—	658,569
				—	—	—	—	—	5,596,450

than the ones already solved with the more simpler strategy S^2. Finally, we also experimented with progressively tightening the consumption of a resource along individual arcs rather than simply tracking a resource or ignoring it altogether. This had a modest impact on the size of the state space and the number of paths explored when it was applied to the weight and flying time constraints. However, again, we could not solve any larger instances.

5. Conclusions

Standard dynamic programming techniques for RCSPPs struggle when faced with the large time-activity network formulations with many resource constraints encountered in DAFPs. However, by using a relaxation-based dynamic programming algorithm that alternates between a forward and a backward search, and by carefully selecting the resources and arcs we use to refine the relaxation, we are able to maintain a small state space and prune many paths while ensuring that an optimal solution to RCSPPs is ultimately found. When tested on practical instances of DAFPs, our algorithm is several orders of magnitude faster than standard dynamic programming techniques. Perhaps more important, our algorithm can also solve much larger pricing problems than standard dynamic programming techniques, which tend to exhaust computer memory more readily. The success resulting from solving larger pricing problems translates to proving near-optimal bounds for instances of DAFPs that are otherwise impossible to obtain.

Finally, we believe that the ideas underlying the dynamic programming algorithm for RCSPPs introduced in this paper, i.e., a dominance scheme that exploits local resource consumption, a bounding scheme that iterates between a forward and backward search and employs progressively tighter relaxations, and a preprocessing scheme that takes advantage of network structure, are likely to be effective for resource-constrained shortest paths found in other time-expanded networks as well as for DAFPs.

References

Beasley, J. E., N. Christofides. 1989. An algorithm for the resource constrained shortest path problem. *Networks* **19**(4) 379–394.

Boland, N., J. Dethridge, I. Dumitrescu. 2006. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Oper. Res. Lett.* **34**(1) 58–68.

Desrochers, M. 1988. An algorithm for the shortest path problem with resource constraints. Technical Report Les Cahiers du GERAD G-88-27, University of Montréal, Montréal.

Desrosiers, J., Y. Dumas, M. Solomon, F. Soumis. 1995. Time constrained routing and scheduling. M. O. Ball, T. L. Magnanti, G. L. Nemhauser, eds. *Handbook in Operations Research and Management Science: Network Routing*, Vol. 8. Elsevier Science, Amsterdam, 35–139.

Dumitrescu, I., N. Boland. 2001. Algorithms for the weight constrained shortest path problem. *Internat. Trans. Oper. Res.* **8**(1) 15–29.

Espinoza, D., R. Garcia, M. Goycoolea, G. L. Nemhauser, M. W. P. Savelsbergh. 2008. Per-seat, on-demand air transportation part I: Problem description and an integer multicommodity flow model. *Transportation Sci.* **42**(3) 263–278.

Feillet, D., M. Gendreau, L. M. Rousseau. 2005. New refinements for the solution of vehicle routing problems with branch and price. Technical Report C7PQMRPO2005-08-X, Center for Research on Transportation, Montréal.

Feillet, D., P. Dejax, M. Gendreau, C. Gueguen. 2004. An exact algorithm for the elementary problem with resource constraints: Application to some vehicle routing problems. *Networks* **44**(3) 216–229.

Garcia, R. 2009. Resource constrained shortest paths and extensions. Ph.D. thesis, Georgia Institute of Technology, Atlanta.

Garey, M. R., D. S. Johnson. 1979. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco.

Handler, G. Y., I. Zang. 1980. A dual algorithm for the constrained shortest path problem. *Networks* **10**(4) 293–309.

Hassin, R. 1992. Approximation schemes for the restricted shortest path problem. *Math. Oper. Res.* **17**(1) 36–42.

Irnich, S., G. Desaulniers. 2005. Shortest path problems with resource constraints. G. Desaulniers, J. Desrosiers, M. M. Solomon, eds. *Column Generation*, Chapter 2. Springer, New York, 33–65.

Jepsen, M. K., B. Petersen, S. Spoorendonk. 2008a. A branch-and-cut algorithm for the elementary shortest path problem with a capacity constraint. Technical report, DIKU, University of Copenhagen, Copenhagen.

Jepsen, M. K., B. Petersen, S. Spoorendonk, D. Pisinger. 2008b. Subset-row inequalities applied to the vehicle-routing problem with time windows. *Oper. Res.* **56**(2) 497–511.

Lorenz, D. H., D. R. Raz. 2001. A simple efficient approximation scheme for the restricted shortest path problem. *Oper. Res. Lett.* **28**(5) 213–219.

Lübbecke, M. E. 2003. Dual variable based fathoming in dynamic programs for column generation. *Eur. J. Oper. Res.* **162**(1) 122–125.

Mehlhorn, K., M. Ziegelmann. 2000. Resource constrained shortest paths. M. Paterson, ed. *Proc. 7th Annual Eur. Sympos. Algorithms* (*ESA2000*). *Lecture Notes in Computer Science*, Vol. 1879. Springer, Berlin, 326–337.

Righini, G., M. Salani. 2005. New dynamic programming algorithms for the resource-constrained elementary shortest path problem. Technical report, Dipartimento di Tecnologie dell'Informazione, Universita degli Studi di Milano, Crema, Italy.

Righini, G., M. Salani. 2006. Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optim.* **3**(3) 255–273.

Ropke, S., J.-F. Cordeau. 2006. Branch-and-cut-and-price for the pickup and delivery problem with time windows. Technical Report C7PQMR PO2006-21-X, Center for Research on Transportation, Montréal.