Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Operations Research Letters 34 (2006) 58–68

Operations Research Letters

www.elsevier.com/locate/orl

Accelerated label setting algorithms for the elementary resource constrained shortest path problem

Natashia Boland[a], John Dethridge[a], Irina Dumitrescu[b,*]

[a]Department of Mathematics and Statistics, The University of Melbourne, Australia
[b]Canada Research Chair in Distribution Management, HEC Montréal, CRT, C.P. 6128, Succursale Centre-ville, Canada H3C 3J7

Received 4 August 2003; accepted 13 November 2004
Available online 4 March 2005

Abstract

A label setting algorithm for solving the Elementary Resource Constrained Shortest Path Problem, using node resources to forbid repetition of nodes on the path, is implemented. A state-space augmenting approach for accelerating run times is considered. Several augmentation strategies are suggested and compared numerically.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Shortest paths; Networks; Dynamic programming; Column generation

1. Introduction

Given a directed graph which has a (possibly negative) cost associated with each arc, a number of resources, each of them having a specified limit, and a nonnegative quantity of each resource consumed on each arc, the Elementary Resource Constrained Shortest Path Problem (ERCSPP) is to find the least cost elementary path (with no repeated nodes) between two specified nodes such that the accumulated quantity of each resource consumed on all arcs in the path does not exceed its limit. The ERCSPP is NP-hard in the strong sense (simple transformation from the longest path problem; alternatively, see [6]).

Let $G=(V,A)$ be the directed graph, where V is the set of nodes, $|V|=n$, and A is the set of arcs, $|A|=m$. We consider R resources, where $R \geqslant 1$. We denote the resource limit vector by $\mathscr{W}=(\mathscr{W}_1,\ldots,\mathscr{W}_R)\in\mathbb{Z}_+^R$, so $\mathscr{W}_r$ is the limit for resource r. For each arc $(i,j)\in A$ we denote the arc cost by $c_{ij}\in\mathbb{R}$ and the resource quantity consumed along the arc by $w_{ij}=(w_{ij}^1,\ldots,w_{ij}^R)\in\mathbb{Z}_+^R$, where w_{ij}^r represents the amount of resource r which is consumed when arc (i,j) is used. Throughout this paper we assume that for all arcs $(i,j)\in A$ there exists a resource $r\in\{1,\ldots,R\}$ such that $w_{ij}^r>0$, i.e., $w_{ij}^r\neq 0$ (we already assume that $w_{ij}^r\geqslant 0$).

For any path p in the graph, we denote the set of arcs that lie along p by $A(p)$. The cost of the path p is $c(p)=\sum_{(i,j)\in A(p)}c_{ij}$ and the resource consumed along p is $w(p)=(w^1(p),\ldots,w^R(p))$, where

* Corresponding author. Tel.: 1514 343 6111, extension 15479; fax: 1514 343 7121.
E-mail address: irina@crt.umontreal.ca (I. Dumitrescu).

0167-6377/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.orl.2004.11.011

$w^r(p) = \sum_{(i,j)\in A(p)} w^r_{ij}$, for all $r = 1, \ldots, R$. The ERCSPP is the problem of finding an elementary path p in G from s to t which minimizes $c(p)$ subject to the constraint that $w^r(p) \leqslant \mathscr{W}_r$ for all $r = 1, \ldots, R$. We say that a path p that satisfies $w^r(p) \leqslant \mathscr{W}_r$ for all $r = 1, \ldots, R$ is *resource feasible*. Note that although we use a simple, additive, notion of path feasibility, it is not hard to adapt the approach we adopt for more general resources constraints, such as time windows. Furthermore, this model can easily be adapted to resources that are consumed on nodes, rather than on arcs. Such a resource can be modelled as a resource consumed on arcs by applying the quantity of resource consumed on the node to all arcs entering the node. It is important to note that our path costs c are *not* restricted in any way and the graph G is not assumed to be acyclic, so there may be negative cost cycles, indeed it is precisely this case that we wish to address.

Our motivation for studying the ERCSPP comes from its application in column generation methods. For example, the subproblem for a column generation approach to the short-haul aircraft rotation problem [2] seeks paths in the flight connection network (in which nodes represent flights) that have negative reduced cost and total flying time no more than that allowed for an aircraft between maintenance stops. Other criteria, such as the number of days, or number of flights, may also be limited. This can be modelled as an ERCSPP in which arc costs model reduced costs and resources are used to model flying time, elapsed time, or number of flights. Since the schedule period is typically a day, which is much shorter than the maximum interval between maintenance, negative cost cycles may appear in the graph. Another example is the subproblem for the vehicle routing problem with time windows (VRPTW), which consists of finding a negative reduced cost vehicle route that respects vehicle capacity and time window constraints (see for example [5]). This can be modelled as a general resource constrained form of the ERCSPP in which the costs model the reduced costs and the resources model the customer demands and travel times.

In practice, the non-elementary relaxation is solved, i.e., routes that visit customers more than once are allowed (although cycles of length 2 in the routes are usually prohibited). Branch-and-bound is used to eliminate such routes. Other examples of successful column generation approaches that use the relaxation of the ERCSPP as a subproblem are those of Lavoie et al. [12] for air crew pairing or Graves et al. [8] for flight crew scheduling.

While several papers in the literature address the problem of eliminating cycles of length 2 in the optimal solution of the non-elementary relaxation of the ERCSPP, not much work investigates the elimination of *all* cycles. Eliminating cycles of length 2 in the context of shortest path algorithms was first addressed by Houck et al. [9] and Christofides et al. [4]. Their method has been the standard procedure used to eliminate 2-cycles since.

Beasley and Christofides [3] gave a zero-one integer programming formulation of the ERCSPP in which the prohibition on visiting nodes more than once in the paths is modelled via an additional resource for each node, which we refer to as a *node resource*. Each of these resource has a resource limit of one. When a path visits a node, one unit of the resource corresponding to that node is consumed and the resource limit is reached. Thus the path cannot visit the node again. The result is a Resource Constrained Shortest Path (RCSPP) formulation, with n additional (node) resources, each having a resource limit of one. We refer to this RCSPP as the *RCSPP with node resources*. Standard dynamic programming approaches to solve this RCSPP will have running time exponential in the number of node resources, and so will be exponential in n. Beasley and Christofides discarded this formulation as unlikely to solve the problem in reasonable time and did not conduct any computational tests.

The idea of associating a resource with each node was later developed by Kohl [10]. Kohl suggested using a state space relaxation of a dynamic programming algorithm for the RCSPP with node resources, where node resources are only included for *some* nodes. This will yield a relaxation of the ERCSPP, which could be solved to yield an optimal path which may contain cycles. A resource associated with some node that is repeated along the optimal solution could be added to the set of node resources included and the RCSPP with node resources re-solved. If the new solution obtained still contained cycles, the procedure could be repeated; an extra resource associated with another repeated node could be added. Kohl also proposed an acceleration of this method making use of the observation that if two nodes cannot be on the same path, the *same* resource can be associated with both nodes.

As in the case of Beasley and Christofides the ideas were not implemented and therefore no numerical experiments were conducted.

The same idea was taken up recently by Feillet et al. [7]. They introduced the notion of strong dominance for enhancing the procedure, which appears to be equivalent to the acceleration idea of Kohl [10]. The difference between the two is that Kohl uses it in order to limit the size of the state space, while Feillet et al. maintain a full-dimensional state space and try to accelerate the method from the point of view of the running time. Feillet et al. present results for a number of problems obtained from Solomon's data set [13]. Results are presented for the method when no acceleration is used and for the method that takes advantage of the acceleration idea, with the latter solving more problems. Feillet et al. [7] also solve the LP relaxation of the VRPTW column generation formulation, with (general constrained) ERCSPP subproblems, and compare the bounds obtained with those of Kohl et al. [11], who also used column generation, but with a non-elementary relaxation subproblem. The results showed fairly minor bound improvements, although in some cases these were enough to yield an integer LP solution.

Feillet et al. [7] also suggested an extension of the method: instead of having one resource associated with each node, resources are associated with subsets of nodes; no pair of nodes contained in the same subset can appear on the same path. Once a subset is visited, the resource corresponding to it is consumed. However this extension was not tested numerically.

In this paper we use the idea of node resources and employ the label setting method for solving RCSPPs with node resources. We adopt the suggestion of Kohl [10] and include node resources for only *some* nodes. Our main innovation is to propose, implement, and test several strategies for selecting which node resources to include and for iteratively augmenting the set of node resources included. As far as we are aware, this is the first time any method with iterative augmentation of the set of node resources has been implemented and tested numerically.

2. Algorithms for forbidding repetition of nodes

In this section we consider a relaxed form of the ERCSPP, in which we seek a resource feasible path of minimum cost, such that no node in a given set $S \subseteq V$ is visited more than once, i.e. is repeated. We describe a modification of the label setting algorithm for the non-elementary form of the problem to solve the relaxed problem, where each node in S is treated as a resource, with a resource limit of one. We discuss preprocessing steps, give a general form of the label setting algorithm, and then define the resources and the procedure for creating new labels.

Definition 2.1. Given a path $p=(i_0, i_1, \ldots, i_l)$ in the graph $G=(V, A)$ and given a node $j \in V$, we define the *multiplicity of j in p*, denoted $M_p(j)$, to be the number of times node j appears in p, i.e., $M_p(j)=|\{k : 0 \leqslant k \leqslant l,\ i_k = j\}|$.

The relaxed form of the ERCSPP, which we denote by S-ERCSPP for given set $S \subseteq V$, may be defined as the problem of finding a path p in G from s to t which is resource feasible, minimizes $c(p)$, such that each node $j \in S$ has multiplicity no more than one in p, i.e., such that $M_p(j) \leqslant 1$ for all $j \in S$. Clearly if $S=\emptyset$ this is simply the non-elementary form of the resource constrained shortest path problem, whilst if $S=V$ this is just ERCSPP.

2.1. Preprocessing of the ERCSPP

Our preprocessing procedure consists of two main steps. The first follows closely the method of Aneja et al. [1], reducing the number of nodes and arcs in the graph by considering least resource paths from the path start node to each node in the graph and from each node in the graph to the path destination node, for each resource. Any node or arc which cannot be used to complete a path from the start to the destination node without violating a resource limit can be deleted from the network. The procedure is repeated until no node or arc can be deleted. Our second step is to solve the all-pairs shortest path problem on the graph, with lengths set to w^r, for each resource r, so the problem is solved R times altogether. We use $\underline{w}^r_{ij}$ to denote the quantity of resource r consumed along the least resource path from node i to node j, for all nodes $i, j \in V$ and all resources $r = 1, \ldots, R$.

2.2. *A general label setting algorithm*

In this section we describe the label setting method of Desrochers et al. [5] with a slight modification to make use of information obtained from preprocessing.

The algorithm uses a set of *labels* for each node. Each label on a node represents a different path from node s to that node and consists of a vector representing the cost and resource usage of the path. For each label on a node, any other label on that node with lower cost must have at least one greater resource element. In what follows, we will want to use extra resources (in particular, node resources), to model the multiplicity constraints, so in this section we consider resource vectors of length R', where $R' \geqslant R$. We formalise these ideas below, where I_i is the index set of labels on node i and for each $k \in I_i$ there is a corresponding path P_i^k from s to i having rth resource W_i^{rk}, for $r = 1, \dots, R'$, and cost C_i^k. Writing W_i^k to denote the vector of all resources, $(W_i^{1k}, \dots, W_i^{R'k})$, we use (W_i^k, C_i^k) to denote the *label* and refer to P_i^k as the *corresponding path*.

For (W_i^k, C_i^k) and (W_i^l, C_i^l) two labels corresponding to two different paths, P_i^k and P_i^l, from s to node i, we say that (W_i^k, C_i^k) *dominates* (W_i^l, C_i^l) if and only if $W_i^k \leqslant W_i^l$, (i.e., $W_i^{rk} \leqslant W_i^{rl}$ for all $r = 1, \dots, R'$), $C_i^k \leqslant C_i^l$, and the labels are not equal. A label (W_i^k, C_i^k) is said to be *efficient* if it is not dominated by any other label at node i, i.e., if $(w(P), c(P))$ does not dominate (W_i^k, C_i^k) for all paths P from s to i, where $w(P) = (w^1(P), \dots, w^{R'}(P))$. A path is said to be efficient if the label it corresponds to is efficient.

The label setting algorithm, given below in Algorithm 2.1, finds all efficient labels on every node. Starting with no labels on any node, except for the label $(0, 0)$ on node s, the algorithm extends the set of all labels by *treating* an existing label on a node, that is, by extending the corresponding path along all outgoing arcs. We use L_i to denote the set of labels on node i and $T_i \subseteq I_i$ to index the labels on node i which have been treated. The label chosen to be treated is the lexicographically minimal untreated label, where we say a vector of resources w_1 is lexicographically less than vector w_2 if for some $r' \in \{1, \dots, R'\}$, $w_1^r = w_2^r$ for all $r = 1, \dots, r' - 1$ and $w_1^{r'} < w_2^{r'}$. The algorithm proceeds until all labels have been treated. Note that it is assumed that the preprocessing steps have already been performed, so that $\underline{w}$ is available to the algorithm.

Algorithm 2.1. *The general label setting algorithm (GLSA)*

```
Step 0: Initialization
    Set $L_s = (0, 0)$ and $L_i = \emptyset$ for all $i \in V \backslash \{s\}$.
    Initialize $I_i$ accordingly and set $T_i = \emptyset$ for each $i \in V$.
Step 1: Selection of the label to be treated
    if $\bigcup_{i \in V} (I_i \backslash T_i) = \emptyset$ then
        stop; all efficient labels have been generated,
        return $\mathscr{P}^* = \{P_t^k : k \in I_t\}$, the set of paths corresponding to labels on node $t$
    else choose $i \in V$ and $k \in I_i \backslash T_i$ so that $W_i^k$ is lexicographically minimal.
Step 2: Treatment of label $(W_i^k, C_i^k)$
    for all $(i, j) \in A$
        if create_new_label($i$,$(W_i^k, C_i^k)$, $j$,$(W', C')$) then
            if $(W', C')$ not dominated by any label in $L_j$ then
                set $L_j = L_j \cup \{(W', C')\}$, remove any dominated labels from $L_j$, and update $I_j$.
    Set $T_i := T_i \cup \{k\}$.
    goto Step 1
```

In the treatment of a label, we must create new labels resulting from the extension of the corresponding path along all outgoing arcs. We take account of preprocessing in doing so. To treat a label (W, C) on node i, we consider the extension to each node j with $(i, j) \in A$. Unless $W + w_{ij} + \underline{w}_{jt} \leqslant \mathscr{W}$, the extension cannot be extended to a feasible path from s to t, in which case there is no point considering it. Otherwise, we may create a new label $(W + w_{ij}, C + c_{ij})$ on node j. Formally, the procedure for creating a new label in the GLSA is as follows:

```
create_new_label($i$, $(W, C)$, $j$, $(W', C')$)
    if $W^r + w_{ij}^r + \underline{w}_{jt}^r > \mathscr{W}_r$ for some $r \in \{1, \dots, R\}$ then return FALSE
    else set $W' = W + w_{ij}$ and $C' = C + c_{ij}$ and return TRUE.
```

2.3. Resources and new label creation

To solve the S-ERCSPP, we define $|S|$ new resources, one for each node in S, resulting in $R'=R+|S|$ resources in total. The limits for the new resources are all one, so we extend $\mathscr{W}_r=1$ for all $r=R+1,\ldots,R'$. We can also extend each resource vector on an arc $(i,j)\in A$ according to

$$w_{ij}^r=\begin{cases}1, & \text{if node } j \text{ is the } (r-R)\text{th node in } S,\\ & \text{or if } i=s \text{ and } s \text{ is the } (r-R)\text{th}\\ & \text{node in } S,\\ 0, & \text{otherwise}\end{cases}$$

for $r=R+1,\ldots,R'$, where now we must order S. Thus for path p originating at node s, $w^{R+r}(p)$ is the multiplicity of the rth node in S in the path p.

We modify the procedure `create_new_label` described in Section 2.2 in order to make use of the *strong dominance* introduced by Feillet et al. [7]. The idea is to set the "node-resource" component of the resource vector in the new label to one for each node in S that cannot be reached from the current node, due to original resource constraints. Implicitly, it is as if the corresponding path has used these nodes, since they, like the nodes in S which have already been visited, cannot be visited by extensions of the path. With this modification the new label is more likely to be dominated. The procedure for creating a new label is given in detail below, where e_r denotes the rth standard basis vector in $\mathbb{R}^{R'}$.

```
create_new_label($i$, $(W, C)$, $j$, $(W', C')$)
  if $W^r + w_{ij}^r + \underline{w}_{jt}^r > \mathscr{W}_r$ for some $r \in \{1,\ldots,R\}$ then return FALSE
  else
    set $W' = W + w_{ij}$ and $C' = C + c_{ij}$
    for each $l \in S$ such that $W'^r + \underline{w}_{jl}^r > \mathscr{W}_r$ for some $r \in \{1,\ldots,R\}$
      Let $q$ be such that $l$ is the $q$th node in $S$.
      if $W'_{R+q} = 0$ then set $W' = W' + e_{R+q}$.
    return TRUE
```

We name the general label setting algorithm with resources and label creation procedure as defined above `GLSA`(S). It is not hard to see that `GLSA`(S) solves the S-ERCSPP. Note that if $S=V$ then `GLSA`(S) is just the method of Feillet et al. [7], with a small modification to make use of preprocessing information, and if $S=\emptyset$ then `GLSA`(S) is simply the usual label setting method for solving the non-elementary form of the problem. It is not hard to see that with the appropriate data structures, the worst-case complexity of `GLSA`(S) is $\mathcal{O}(|A|\prod_{r=1}^{R}(\mathscr{W}_r+1)2^{|S|})$. So clearly, the complexity increases exponentially with the size of S. Therefore finding small sets S is desirable. In what follows, we use $\mathscr{P}^*(S)$ to denote the set of paths returned by `GLSA`(S).

2.4. State space augmenting algorithms

We now consider several methods for solving ERCSPP, making use of the algorithm `GLSA`(S) for solving the S-ERCSPP. It is important to observe that the value of S-ERCSPP is a lower bound on the value of ERCSPP, since S-ERCSPP is identical to ERCSPP, except that the multiplicity constraints on the nodes in $V\backslash S$ are relaxed. Thus if some minimum cost path found by `GLSA`(S) is elementary, it must be an optimal solution to the ERCSPP. Our hope is that for some S consisting of relatively few of the elements of V, the solution to the S-ERCSPP will in fact be elementary.

In all cases, we begin by solving the non-elementary problem, i.e., we set $S=\emptyset$ and solve the S-ERCSPP using `GLSA`(S). We then examine the paths generated (those returned in $\mathscr{P}^*(S)$) and based on these, determine a set of nodes to add to S. We repeat the procedure until a minimum cost path found by `GLSA`(S) is elementary, in which case we may stop, assured that this path is an optimal solution to the ERCSPP. In all cases the nodes selected appear in at least one cycle in some path in $\mathscr{P}^*(S)$. If the method is applied to solve column generation subproblems, in which a negative cost elementary path is sought, the method may actually be terminated earlier, as soon as such a path, (not necessarily the least cost path), is found by `GLSA`(S). In what follows, we use p_S^* to denote a cheapest, i.e., least cost, path in $\mathscr{P}^*(S)$. The general state space augmenting algorithm is given in Algorithm 2.2.

Algorithm 2.2. *The general state space augmenting algorithm* (*GSSAA*)

```
Perform preprocessing to reduce the network and find $\underline{w}$.
Set $S = \emptyset$.
Run GLSA($S$) to obtain $\mathscr{P}^*(S)$ and hence $p_S^*$.
while $p_S^*$ is not elementary do
```

```
        update_set_S($S, \mathscr{P}^*(S)$)
        Run GLSA($S$) to obtain $\mathscr{P}^*(S)$ and hence $p_S^*$.
    Return $p_S^*$.
```

In what follows we give several alternative approaches to augmenting S. The first algorithm we consider increases the set S by one node, the node with highest multiplicity in a cheapest path in $\mathscr{P}^*(S)$, i.e., in p_S^*. We note that if there are several cheapest paths, the first one found will be used, and if there are several nodes with highest multiplicity in this path, the first one found will be added. The procedure for updating the set S, i.e. the content of update_set_S($S, \mathscr{P}^*(S)$), is given in detail below, for each approach.

```
Highest multiplicity on the optimal path (HMO):
    Find $i \in V$ that maximizes $M_{p_S^*}(i)$.
    Set $S = S \cup \{i\}$.
```

The next algorithm we consider uses a similar procedure for updating the set S; the only difference is that *all* nodes with highest multiplicity in the cheapest path are added to the set S, instead of just one. Thus the cardinality of S increases faster with each iteration.

```
Highest multiplicity on the optimal path—all
nodes (HMO-All):
    Find $i \in V$ that maximizes $M_{p_S^*}(i)$.
    Set $S = S \cup \{j \in V : M_{p_S^*}(j) = M_{p_S^*}(i)\}$.
```

Our third proposal of updating the set S is by adding to the set S all the nodes with multiplicity greater than one in the cheapest path.

```
Multiplicity greater than one on the optimal
path—all nodes (MO-All):
    Set $S = S \cup \{i \in V : M_{p_S^*}(i) > 1\}$.
```

The next algorithm uses an updating procedure that adds all nodes which have multiplicity greater than one in *some* path in $\mathscr{P}^*(S)$.

```
Multiplicity greater than one on some
path—all nodes (M-All):
    Set $S = S \cup \{i \in V : M_p(i) > 1, \quad \exists p \in \mathscr{P}^*(S)\}$.
```

It should be clear that the algorithms progress from "most conservative" to "least conservative" in terms of the rate at which nodes are added to S: the algorithm that uses the *HMO* procedure has the slowest rate while the algorithm that uses the *M-All* procedure has the highest rate.

In addition to the state space augmenting algorithms, we also implemented the algorithm with S initialized to be the set of all vertices, $S = V$, which is very like the approach of Feillet et al. [7]. In this case, the procedure of updating S will be run only once. We denote this procedure by *All*.

3. Numerical experiments

We tested all algorithms on randomly generated test problems, all having a single resource ($R = 1$). The problem generator we used is a modified version of the generator provided by Cherkassky et al. in SPLIB [14] for the Shortest Path Problem. Our generator does not allow parallel arcs and produces problems that have the randomly generated resource associated with each arc inversely proportional to its cost (modulo randomization). Our generater is parameterized by the number of nodes in the network, the arc density of the network, the proportion of arcs having negative cost, a parameter $\alpha \in (0, 1)$ that controls how arc resources are generated, and a parameter $\beta \in \mathbb{Z}_+$ that controls the resource limit. Initially, the cost c_a of each arc $a \in A$ is set to be a randomly generated integer between 0 and 101. After these values are generated, the resource w_a associated with each arc a is generated according to the formula

$$w_a = 1 + \left\lfloor \frac{\alpha C}{c_a + 1} \right\rfloor + nrand\left(\left\lfloor \frac{C}{c_a + 1} \frac{1 - \alpha^2}{\alpha} \right\rfloor\right),$$

where $nrand(l)$ is the function which randomly generates an integer between 0 and l written by Cherkassky and Goldberg [14] and C is the maximum cost of an arc over all arcs in the graph. The resource limit W is generated by taking the sum of all arc resources, dividing by 2β and rounding down. Finally, we calculate an integer to subtract off all arc costs to ensure that the desired proportion of arcs have negative cost and set the final cost of each arc to be its initial cost minus this integer.

After extensive numerical testing, we found that the difficulty of the problems generated in this manner is quite sensitive to the choice of α and β. The parameter α affects the degree of "variability" in the arc weights. Values of α closer to one will yield problems with arc resources drawn from a very narrow range of values. As α approaches zero, the range of resource values broadens rapidly. The parameter β affects the number of arcs that can appear in feasible paths, i.e., feasible path *length*; higher values of β yield shorter feasible paths. For all experiments reported in this paper, we used $\alpha = 0.163$ and chose β so that "average" feasible path length was about 10. Decreasing α for the same value of β increases problem solution time roughly proportionately (dividing α by 10 increases solution times 10-fold). Decreasing β for the same value of α increases problem solution times by a factor of roughly 50 for every five-node increase in average path length. Increasing either α or β for these values, for the range of other problem parameters reported in this paper, produces problems very easily solved by the best method. So, as one might expect, problem difficulty increases with increasing variability in arc resource values and with increasing feasible path length. Our choice of α and β to report here was made to reflect the kinds of values one might see in realistic problems and to permit detailed comparisons to be made across all algorithms; lower values lead to greatly inflated run times for the worst algorithms, while higher values lead to very short run times for all algorithms, making meaningful comparison difficult.

We report numerical results (algorithm run times and memory usage) for a range of the other problem generator parameters: number of nodes, arc density, and negative cost arc proportion. For each choice of parameters we generated one hundred problem instances. One factor that makes reporting these results difficult is the high degree of variability in run times (and memory usage) observed across randomly generated problems generated with identical parameters. Even with one hundred problem instances, we find 90% confidence intervals about the mean run times are quite wide, with widths about a half to two-thirds the value of the mean. Looking at histograms of run times, on an exponential scale (e.g. with bars of the histogram representing the number of problems solved in less than 1, 2, 4, 8 s, and so on), we find that the bars are almost level, with a small peak at smaller run times. Similar observations can be made about memory usage. This must be kept in mind when reading our reports of average absolute run times and memory usage. However *relative* run times (and relative memory usage measures) were quite a bit more reliable, having much narrower 90% confidence interval widths about the mean. Thus our reports of average *relative* algorithm performance provide a reliable indication of the relative performance across all problems.

For each algorithm, we record the number of iterations, i.e., the number of times the "while loop" is repeated, which is equivalent to the number of times $\texttt{GLSA}(S)$ is run with non-empty S. We also record the largest number of labels used in running $\texttt{GLSA}(S)$ over all iterations: this is our measure of memory usage. We record the number of nodes in S at the end of the algorithm and the total time taken in CPU seconds. In Tables 1–5 we report summary results on problems having a range of characteristics. Each line of a table represents an average result, averaged over a group of one hundred different randomly generated problems, generated using identical parameters as input to the random problem generator. We report the number of nodes in each problem in the group, the number of arcs, and the percentage of arcs which have a negative cost, in case these values are not the same for the whole set of results presented in a table. If some of the values are constant for all groups, we do not mention them in the tables (only in the table caption).

The first set of results is split into two tables: Tables 1 and 2. Table 1 contains results on the memory and time usage of the algorithms, while Table 2 contains results on the number of iterations and the size of the set S. In Table 1, for the algorithm that uses the *HMO* updating procedure, GSSAA (*HMO*), we report the average of the maximum number of labels used in solving each of the problems in the group in the column labeled $\hat{L}$. For each of the other algorithms, we report the average ratio of the number of labels required by the algorithm to solve the problem divided by the number required by GSSAA (*HMO*), over all problems in the group. To be more precise, if l_k^{proc} denotes the maximum number of labels used over each iteration of the algorithm that uses the updating procedure *proc* to solve the kth problem in the group, $\hat{L} = \frac{1}{100}\sum_{k=1}^{100} l_k^{HMO}$, and the entry recorded in the column labeled $L/\hat{L}$ for GSSAA (*proc*) is given by $\frac{1}{100}\sum_{k=1}^{100}(l_k^{\text{proc}}/l_k^{HMO})$, where $proc \in \{HMO-All, MO-All, M-All, All\}$. Similarly, we record the average CPU time in seconds needed to run GSSAA (*HMO*) in the column labeled $\hat{T}$ and for each of the other algorithms, in the column labeled $T/\hat{T}$, we report the average ratio of the time required by the

Table 1
Memory and time usage for all algorithms

Problem group characteristics			Algorithm									
			HMO		*HMO-All*		*MO-All*		*M-All*		*All*	
$\|V\|$	$\|A\|$	% neg. cost arcs	$\hat{L}$	$\hat{T}$	$L/\hat{L}$	$T/\hat{T}$	$L/\hat{L}$	$T/\hat{T}$	$L/\hat{L}$	$T/\hat{T}$	$L/\hat{L}$	$T/\hat{T}$
30	435	20	2298	0.02	1.48	1.30	1.67	1.48	2.12	1.67	85.00^{1}	10872^{1}
30	435	25	13636	1.99	1.62	2.04	2.32	8.15	3.40	23.09	131.55^{77}	3955.49^{77}
50	1225	15	2507	0.02	1.42	1.31	1.55	1.40	1.97	1.84	162.78^{10}	36468.86^{10}
150	16762	10	15172	0.98	1.40	1.27	1.53	1.59	2.85	4.76	*	*
30	435	30	60904	49.77	2.05^{3}	7.05^{3}	2.84^{7}	12.59^{7}	5.49^{6}	82.21^{6}	*	*
50	1225	20	20640	3.37	1.74	2.81	2.61	9.17	4.25	23.37	*	*
100	4950	15	26872	2.87	1.74	3.44	2.09	5.90	3.91^{1}	16.99^{1}	*	*
100	9900	15	169210^{16}	268.38^{16}	2.21^{42}	6.95^{42}	2.81^{49}	12.49^{49}	4.85^{69}	55.25^{69}	*	*

Each line in the table represents average results for each problem group. Each group contains 100 problems generated with identical parameters. The average result for labels is rounded down. A superscript indicates the number of problems in the group that were *not* solved within 30 min. If a superscript appears, then the result reported is an average over the remaining problems (the problems that were solved). A "*" indicates the fact that the corresponding algorithm was not run for that group.

algorithm to solve the problem divided by the time required by GSSAA (*HMO*).

In Table 2 we report on the number of iterations and the number nodes which needed to be added to S, to solve the problem. For GSSAA (*HMO*) we report in the column labelled $|\hat{S}|$ (%) the average $|S|$ at the end of the algorithm as a percentage of the number of nodes in the network, $|V|$. We also report the average of the number of iterations required to solve each of the problems in the group, in the column labeled $\hat{N} = |\hat{S}|$ (in the *HMO* algorithm, $|S|$ increases by one at each iteration, so the final $|S|$ is equal to the number of iterations in all cases). For each of the other algorithms, we report the average ratio of the number of iterations required by the algorithm to solve the problem divided by the number required by GSSAA (*HMO*). This is recorded in the column labeled $N/\hat{N}$ for each of these algorithms. We also report, in the column labeled $|S|/|\hat{S}|$, the average ratio of the $|S|$ required by the algorithm to solve the problem divided by the $|S|$ required by GSSAA (*HMO*), over all problems in the group. We do not report such results for the last algorithm considered, GSSAA (*All*), as there are no "iterations" for this algorithm and $|S| = |V|$ in this case. For simplicity, we label the columns corresponding to the algorithms by their updating procedure.

Tables 3–5 present the results on memory, time, number of iterations, and size of S together, for the GSSAA (*HMO*) algorithm.

All algorithms are programmed in C, use common subroutines, and are run on an Intel Xeon Pentium (3.2 GHz) with 512 KB Cache and 1 GB RAM. We finally mention that we considered a time limit of 30 min for all algorithms.

It is clear from Table 1 that in terms of both time and memory GSSAA (*HMO*) is the best, (all ratios for the other algorithms are at least one), and GSSAA (*All*) is by far the worst. We see that the maximum number of labels required is generally correlated with the total time required in all algorithms. What is interesting is that there is a clear degradation in the algorithm performance as the algorithms become more aggressive in their state space augmentation, with time and memory increasing markedly across the table. We see quite dramatically in this table how the state space augmentation approaches need orders of magnitude less time and memory to solve the problem, compared with the approach of GSSAA (*All*).

Table 2
Number of iterations and final $|S|$ for the first four algorithms. (Recall that GSSAA (*All*) only performs one iteration and in this case $|S| = |V|$)

Problem group characteristics			Algorithm							
$\|V\|$	$\|A\|$	% neg. cost arcs	*HMO*		*HMO-All*		*MO-All*		*M-All*	
			$\|\hat{S}\|$ (%)	$\hat{N} = \|\hat{S}\|$	$N/\hat{N}$	$\|S\|/\|\hat{S}\|$	$N/\hat{N}$	$\|S\|/\|\hat{S}\|$	$N/\hat{N}$	$\|S\|/\|\hat{S}\|$
30	435	20	13.14	3.94	0.72	1.60	0.64	1.86	0.46	2.29
30	435	25	20.53	6.16	0.63	1.44	0.51	1.71	0.36	2.05
50	1225	15	5.37	2.68	0.84	1.92	0.81	2.27	0.64	2.99
150	16762	10	2.99	1.99	0.80	2.41	0.74	2.94	0.60	5.11
30	435	30	27.00	8.10	0.57^{3}	1.39^{3}	0.45^{7}	1.60^{7}	0.33^{6}	1.97^{6}
50	1225	20	10.71	5.36	0.66	1.75	0.57	2.24	0.43	2.80
100	4950	15	4.55	4.55	0.67	1.94	0.56	2.31	0.43^{1}	3.40^{1}
100	9900	15	7.23^{16}	7.23^{16}	0.61^{42}	1.98^{42}	0.48^{49}	2.51^{49}	0.35^{69}	3.67^{69}

Each line in the table represents average results for each problem group. Each group contains 100 problems that were generated using identical parameters. A superscript indicates the number of problems in the group that were *not* solved within 30 min. If a superscript appears, then the result reported is an average over the remaining problems (the problems that were solved).

Table 3
Memory, time usage, and number of iterations for GSSAA (*HMO*) for problems with 50% arc density, for which the number of nodes increases

Problem group characteristics (20% neg. arcs)		Algorithm			Problem group characteristics (10% neg. arcs)		Algorithm		
$\|V\|$	$\|A\|$	*HMO*			$\|V\|$	$\|A\|$	*HMO*		
		$\hat{L}$	$\hat{T}$	$\hat{N} = \|\hat{S}\|$			$\hat{L}$	$\hat{T}$	$\hat{N} = \|\hat{S}\|$
30	435	2298	0.02	3.94	100	4950	3146	0.06	1.87
40	780	7036	0.24	4.36	150	11175	8428	0.32	2.41
50	1225	20640	3.37	5.36	200	19900	14065	0.81	2.36
60	1770	46201^{3}	26.02^{3}	6.35^{3}	250	31125	23715	2.05	2.77

The percentage of negative cost arcs is kept constant. The average result for labels is rounded down. The left side of the table presents results for problems with 20% negative cost arcs and the right side of the table presents results with 10% negative cost arcs. A superscript indicates the number of problems in the group that were *not* solved within 30 min. If a superscript appears, then the result reported is an average over the remaining problems (the problems that were solved).

In Table 2, we can see that the final $|S|$ for GSSAA (*HMO*) is substantially less than $|V|$, as we had hoped. The average $|S|$ ranges from 2.99% to 27% of $|V|$. For the more aggressive augmentation algorithms the proportion becomes larger, in some cases reaching 50%.

In the rest of the paper we investigate the difficulty of the ERCSPP. For simplicity, we present only results obtained for the best algorithm, GSSAA (*HMO*).

The size of the problem is a key factor in determining the difficulty of an ERCSPP. We tested GSSAA (*HMO*) on several sets of problems with the same arc density and percentage of negative cost arcs, but varying numbers of nodes. The results are presented in Table 3. We see that the number of labels created increases significantly with increase in the number of nodes (and implicitly arcs). A dramatic increase in computational time is also recorded. We note that three

Table 4
Memory, time usage, and number of iterations for GSSAA (*HMO*) for problems with 50 nodes and 15% negative cost arcs for which the arc density increases

Problem group characteristics	Algorithm *HMO*		
Arc density (%)	$\hat{L}$	$\hat{T}$	$\hat{N} = \lvert\hat{S}\rvert$
40	1714	0.02	2.21
60	4392	0.07	3.29
80	9927	0.46	4.64
90	15075	1.86	5.11
100	19188	3.33	5.32

The average result for labels is rounded down. A superscript indicates the number of problems in the group that were *not* solved within 30 min. If a superscript appears, then the result reported is an average over the remaining problems (the problems that were solved).

Table 5
Memory, time usage, and number of iterations for GSSAA (*HMO*) for problems with 30 nodes and 435 arcs for which the percentage of negative cost arcs increases

Problem group characteristics	Algorithm *HMO*		
% neg. cost arcs	$\hat{L}$	$\hat{T}$	$\hat{N} = \lvert\hat{S}\rvert$
20	2298	0.02	3.94
25	13636	1.99	6.16
30	60904	49.77	8.10
35	115789^{35}	192.20^{35}	9.35^{35}

The average result for labels is rounded down. Each group contains 100 problems generated with identical parameters. A superscript indicates the number of problems in the group that were *not* solved within 30 min. If a superscript appears, then the result reported is an average over the remaining problems (the problems that were solved).

problems from the last group with 20% negative cost arcs could not be solved in less than 30 min. The size of the set S also increased with problem size.

The difficulty of the ERCSPP also increases with the arc density of the problem. We tested the algorithms on several groups of problems for which the number of nodes and percentage of negative cost arcs was kept unchanged, along with all other parameters, while the arc density in the graphs was increased. The results in terms of running time and memory used are presented in Table 4. As expected, GSSAA (*HMO*) took more time and used more memory to solve the problems when the size of the graphs increased. We see that $|S|$ too increased with arc density.

Another important factor that determines the difficulty of an ERCSPP is the proportion of negative cost arcs in the graph. We tested the algorithms on several sets of problems that were generated using the same initial parameters with the exception of the percentage of the negative cost arcs in the graph. The results for GSSAA (*HMO*) are presented in Table 5. We see that with every increase in the proportion of negative cost arcs, the run time increased by at least an order of magnitude, and the maximum number of labels created also increased dramatically. We see that $|S|$ too increased substantially with the percentage of negative cost arcs.

4. Conclusions and future work

State space augmenting strategies can obtain optimal solutions in significantly less computation time than if the full state space is used from the outset. More conservative strategies are generally better, and with these strategies much harder problems can be solved in modest computing time. In the future we intend to apply this method within column generation, to consider Lagrangian relaxation of some of the node multiplicity constraints to improve lower bounds earlier, and to consider removing node resources at each iteration, as well as adding them.

References

[1] Y.P. Aneja, V. Aggarwal, K.P.K. Nair, Shortest chain subject to side constraints, Networks 13 (1983) 295–302.

[2] C. Barnhart, N. Boland, L. Clarke, E.L. Johnson, G.L. Nemhauser, R.G. Shenoi, Flight string models for aircraft fleeting and routing, Transport. Sci. 32 (1998) 208–220.

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

[4] N. Christofides, A. Mingozzi, P. Toth, Exact algorithms for the vehicle routing problem based on spanning tree and shortest path relaxations, Math. Program. 20 (1981) 255–282.

[5] M. Desrochers, J. Desrosiers, M. Solomon, A new optimization algorithm for the vehicle routing problem with time windows, Oper. Res. 40 (1992) 342–354.

[6] M. Dror, Note on the complexity of the shortest path models for column generation in VRPTW, Oper. Res. 42 (5) (1994) 977–978.

[7] D. Feillet, P. Dejax, M. Gendreau, C. Gueguen, An exact algorithm for the elementary shortest path problem with resource constraints: application to some vehicle routing problems, Networks 43 (3) (2004) 216–229.

[8] G.W. Graves, R.D. McBride, I. Gershkoff, D. Anderson, D. Mahidhara, Flight crew scheduling, Manage. Sci. 39 (6) (1993) 657–682.

[9] D.J. Houck Jr., J.-C. Picard, M. Queyranne, R.R. Vemuganti, The travelling salesman problem as a constrained shortest path problem: theory and computational experience, Oper. Res. 17 (1980) 93–109.

[10] N. Kohl, Exact methods for time constrained routing and related scheduling problems, Ph.D. Thesis, Institute of Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark, Ph.D. Dissertation no. 16, 1995.

[11] N. Kohl, J. Desrosiers, O.B.G. Madsen, M. Solomon, F. Soumis, 2-path cuts for the vehicle routing problem with time windows, Transport. Sci. 33 (1999) 101–116.

[12] S. Lavoie, M. Minoux, E. Ordier, A new approach for crew pairing problems by column generation with an application to air transportation, Eur. J. Oper. Res. 35 (1988) 45–58.

[13] M.M. Solomon, Vehicle routing and scheduling with time window constraints: models and algorithms, Ph.D. Thesis, Department of Decision Sciences, University of Pennsylvania, 1983.

[14] SPLIB, http://www.star-lab.com/goldberg/soft.html, 1994.