Exploiting Set-Based Structures to Accelerate Dynamic Programming Algorithms for the Elementary Shortest Path Problem with Resource Constraints

by

Troels Martin Range

Discussion Papers on Business and Economics
No. 17/2013

FURTHER INFORMATION
Department of Business and Economics
Faculty of Business and Social Sciences
University of Southern Denmark
Campusvej 55
DK-5230 Odense M
Denmark

Tel.: +45 6550 3271
Fax: +45 6550 3237
E-mail: lho@sam.sdu.dk
http://www.sdu.dk/ivoe

Exploiting Set-Based Structures to Accelerate Dynamic Programming Algorithms for the Elementary Shortest Path Problem with Resource Constraints

Troels Martin Range*

Department of Business and Economics, and COHERE, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark

October 16, 2013

Abstract

In this paper we consider a label-setting dynamic-programming algorithm for the Elementary Shortest Path Problem with Resource Constraints (ESPPRC). We use a pseudo resource to guarantee that labels are permanent. We observe that storing the states based on the subset of nodes visited by the associated path can improve the performance of the algorithm significantly. To this end we use a variant of a prefix tree to store the states and show by computational experiments that the performance of the dynamic programming algorithm is improved significantly when the number of undominated states is large.

Keywords: Elementary Shortest Path Problem, Resource Constraints, Dynamic Programming, Prefix Tree
JEL Code: C61 **MSC Code:** 90C39, 90B06, 68P05

1 Introduction

The Elementary Shortest Path Problem with Resource Constraints (ESPPRC) is the problem of identifying a minimum cost non-cyclic path through a network having possibly negative arc costs, such that the path satisfies a set of resource constraints. This problem, also known as the pricing problem, arises when column generation is applied to the Vehicle Routing Problem with Resource Constraints (see Desrochers et al. (1992)). Dror (1994) shows that the ESPPRC is $\mathcal{NP}$-hard in a strong sense for the special case where the resource constraints are time windows.

The prevalent approach for solving ESPPRC has been Dynamic Programming (DP), which is a general group of techniques where the solution process is divided into a sequence of stages such that each stage can be derived from the previous stages. Within each stage a set of possible states is constructed to keep track of the possible solutions of the stages. A state corresponds to a partially constructed solution, and it has to be recursively extended

*E-mail: tra@sam.sdu.dk, Phone: +45 6550 3685, Fax: +45 6550 3237

to other states in later stages to complete the solution. DP suffers from the problem that the number of states within a stage may explode if it is not possible to eliminate some of the states. To handle this, simple sufficient dominance criteria are often given, which stipulate that one state will always yield a better final solution than the other state.

Among the key factors to construct a good DP algorithm for a problem are:

1. the strength of the sufficient dominance criteria. In some cases using more computation time may enable a dominance criterion which eliminates more states. Whether or not this is worthwhile depends on the problem type or the problem instance.

2. the order of extending the states, i.e., how the states are sorted into stages. If this is not done carefully, then some states already extended may be dominated and thereby eliminated. As a consequence, all the extensions are also dominated and should be eliminated. Hence, these extensions have wasted computational time.

3. the search method when trying to identify dominant or dominated states. When the number of states in a stage grows large, then it is important to be able to quickly identify the subset of states which may be dominated or dominant.

While we acknowledge that the first factor is important, we focus solely on the second and the third factors in the present paper. Especially the third factor has not received much attention in the literature, and we will show that it is important to efficiently identify dominant and dominated states.

The main contribution of this paper is the development of a set-based data structure – a so-called prefix tree – which facilitates fast dominance checks excluding a large number of unnecessary dominance checks. We show that when the problems have a large number of states, then the set-based data structure significantly speeds up the solution process. A secondary contribution is to develop a permanent label setting algorithm for any kind of resource extension function. In ESPPRC this is based on the observation that paths cannot be extended to already visited nodes, and therefore we can use the number of unreachable nodes as a pseudo resource to guarantee that states with fewer unreachable nodes will never be eliminated.

The paper is organized as follows: A brief review of related literature is given in section 2. This is followed, in section 3, by a formal description of the problem and the related notation. Then a description of the dynamic programming procedure is given in section 4. In section 5 we discuss how to store the states of the ESPPRC efficiently. A series of computational experiments have been conducted, and they are described and discussed in section 6. Finally, section 7 concludes the paper.

2 Related literature

Resources in shortest path problems have been used in many variants. Examples are capacity constraints, time windows, and follower constraints in routing problems. Desaulniers et al. (1998) introduce the definition of a resource extension function, which is a generalization of the examples of resource constraints given above. They describe several types of resource constraints. A further discussion of resource constraints, as well as shortest path problems with resource constraints in general, is given by Irnich and Desaulniers (2005).

A distinction has to be made between methods having both non-negative costs and non-negative resource consumption and methods for problems not satisfying this requirement. We are mainly interested in the latter case. The case having non-negative costs and

non-negative resource consumption is investigated by Mehlhorn and Ziegelmann (2000), Dumitrescu and Boland (2003), and Santos et al. (2007).

Desrochers and Soumis (1988) construct a dynamic programming algorithm for the (non-elementary) shortest path problem with resource constraints based on the assumption that a monotone positive resource extension function exists. They use generalized buckets for this monotone positive resource extension function to constitute the stages of the dynamic programming. Powell and Chen (1998) construct an algorithm for the (non-elementary) shortest path problem with resource constraints based on lexicographic ordering of states and choose only states for extension which are below a certain threshold.

Desrochers et al. (1992) apply a relaxation of the ESPPRC to solve the Vehichle Routing Problem with Time Windows. They relax the elementarity of the paths and replace this requirement with the requirement that no path can contain a 2-cycle. Dror (1994) shows, as a comment on the paper by Desrochers et al. (1992), that ESPPRC is $\mathcal{NP}$-hard in a strong sense for the special case where the resource constraints are time windows. Irnich and Villeneuve (2006) construct a k-cycle free algorithm for the resource constrained shortest path problem. In principle, this algorithm can be used to solve ESPPRC by setting k sufficiently large, but the complexity of the suggested algorithm increases exponentially with k.

Beasley and Christofides (1989) note that in order to guarantee elementarity of a path it is sufficient to add an extra resource for each node indicating whether or not the node has been visited on the path. This resource is upper bounded by one, thereby prohibiting the path to reenter previously visited nodes. Feillet et al. (2004) enhance this idea by observing that some nodes are not reachable due to the resource constraints and that node resources for the elementarity can be incremented for these nodes without the path having visited them. They also observe that it is possible to count the number of unreachable nodes for a path and use this number as a pseudo resource to speed up the dominance check, i.e., if a state has larger value in the pseudo resource than another state, then it can never dominate the other state. In all, as it is possible to model the elementarity by resources, it is possible to use standard SPPRC algorithms for the ESPPRC. Chabrier (2006) applies a stronger dominance criterion than Feillet et al. (2004) to eliminate more states. This is based on the observation that if one state has sufficiently less cost compared to another state, then some of the nodes not reachable for the first state but reachable for the second state will never be reached by a lower cost extension of the second state.

Given that the resource extension functions of the shortest path problem have inverse functions, Righini and Salani (2006) show that a significant reduction in the number of states can be obtained by solving the problem using bidirectional dynamic programming. Irnich (2008) gives a thorough treatment of inverse resource extension functions.

Kohl (1995) describes a state space relaxation of the ESPPRC, where the problem is relaxed by removing the requirement that each node must be visited at most once. After the relaxation has been solved the requirement is reintroduced for the nodes which are visited more than once on the best path. This process is repeated until an elementary path is found. Boland et al. (2006) and Righini and Salani (2008) implement this strategy and show that in many cases it is worthwhile to use this approach.

Ibrahim et al. (2009) set up a multi-commodity flow formulation for the elementary shortest path problem in a digraph and show that the LP relaxation of this is stronger than an arc-flow formulation. Drexl and Irnich (2012) follow up on this and show that from an integer point of view the arc-flow formulations along with subtour elimination constraints are superior to the multi-commodity flow base formulations. Furthermore, Drexl and Ir-

nich (2012) argue that neither mathematical formulations-based approaches nor dynamic programming-based methods are able to fully handle large scale elementary shortest path problems.

The ESPPRC can be interpreted as a prize collecting problem, where a connected path has to be selected such that the total prize obtained from the arcs is maximized. The prize for each arc will then correspond to the negated cost of the arc. The selective traveling salesman problem (STSP) described by Gendreau et al. (1998) is closely related to the ESPPRC, but with two main differences: The STSP does not have resources constraints, and it has to visit a pre-specified subset of the nodes of the network. Gendreau et al. (1998) solve STSP by branch-and-cut.

Gualandi and Malucelli (2012) provide a constraint programming approach for solving resource constrained shortest path problems with super additive cost functions. Their definition of resources differs from the one we use in the sense that in their context resources are additive and are allowed to vary freely as long as the path at termination has resource values within a specific resource window. Hence, they only have a single resource window for each resource, whereas we have resource windows for each node and each resource.

3 Problem Description

Let $D(\mathcal{V}, \mathcal{A})$ be a directed graph with nodeset $\mathcal{V}$ and arcset $\mathcal{A} \subseteq \{(i,j)|i,j \in \mathcal{V}, i \neq j\}$. The set of nodes is divided into the set $\mathcal{C}$ of intermediate nodes which is sometimes referred to as the customer nodes, the origin node o, and the destination node d, i.e., $\mathcal{V} = \mathcal{C} \cup \{o, d\}$.

We denote a path visiting nodes $v_0, \ldots, v_p$ in sequence for $P = (v_0, \ldots, v_p)$. The length of path P is p corresponding to the number of arcs used along the path. We say that a node $v \in \mathcal{V}$ is on the path if $v \in \{v_0, \ldots, v_p\}$, and we adopt the notation $v \in P$. For convenience we let $P_q = (v_0, \ldots, v_q)$, for $0 \leq q \leq p$, be the subpath of P visiting the first $q+1$ nodes of P. If all the nodes $v_0, \ldots, v_p$ are pairwise distinct, then the path is said to be elementary.

A real valued cost c_{ij} is associated with each arc $(i,j) \in \mathcal{A}$. The cost of a path, P, is then $C(P) = \sum_{q=1}^{p} c_{v_{q-1}v_q}$. The cost can also be calculated recursively by

$$\begin{array}{lcll} C(P_0) & = & 0 & \\ C(P_q) & = & C(P_{q-1}) + c_{v_{q-1}v_q}, & 1 \leq q \leq p \end{array} \tag{1}$$

The recursive calculation is typically used within dynamic programming as we extend the paths one node at a time.

An index set of resources $\mathcal{R} = \{1, \ldots, R\}$ is given, and for each node $i \in \mathcal{V}$ and resource $r \in \mathcal{R}$ a resource window $[a_i^r, b_i^r]$ is present. With each arc $(i,j) \in A$ and resource $r \in \mathcal{R}$ a resource extension function (REF) $f_{ij}^r : \mathbb{R}^R \to \mathbb{R}$ is defined, stating the amount, $f_{ij}^r(T)$, of resource r is used when traversing arc (i,j) starting from i with resource vector $T \in \mathbb{R}^R$. Given a path $P = (v_0, \ldots, v_p)$, the resource consumption when entering node v_p is denoted $T(P) = (T^1(P), \ldots, T^R(P))$. Typically, the resource consumption along a path is calculated by the recursion

$$\begin{array}{ll} T^r(P_0) & = a_{v_0}^r \qquad\qquad\qquad\qquad\qquad\quad , r \in R \\ T^r(P_q) & = \max\left\{a_{v_q}^r, f_{v_{q-1},v_q}^r(T(P_{q-1}))\right\} \quad , r \in R, 1 \leq q \leq p \end{array} \tag{2}$$

Clearly, this depends on the types of resources in the problem, and the reader is referred to Irnich and Desaulniers (2005) for a more elaborate description of these.

The ESPPRC is then the problem of identifying an elementary path $P = (v_0, \ldots, v_p)$ from $v_0 = o$ to $v_p = d$ such that the cost is minimized and $T^r(P_q) \in [a_{v_q}, b_{v_q}]$ for all $v_q \in P$.

Even though the results of this paper can be applied using more general REFs, we limit our attention to separable non-decreasing REFs in order to make the exposition more clear. Furthermore, note that our algorithm does not require, that an inverse REF exists, and it can therefore be applied to a wider variety of problems compared to the bidirectional approach described by Righini and Salani (2006).

4 Dynamic Programming Algorithm

The ESPPRC can be solved by dynamic programming, which we describe in this section. Our algorithm is a slight modification of the label correcting approach described by Feillet et al. (2004). The main difference between our approach and the one presented by Feillet et al. (2004) is that by selecting states for extension carefully, we obtain a label setting algorithm rather than a label correcting algorithm and consequently minimize the number of extensions performed.

We will denote $\mathcal{L}(P) = (C(P), T(P))$ the state (or the label) of a path P. When extending path P to node v_{p+1}, we obtain the path $Q = (v_0, \ldots, v_p, v_{p+1})$, and we can obtain $\mathcal{L}(Q) = (C(Q), T(Q))$, where $C(Q)$ and $T(Q)$ are calculated by the recursion given in (1) and (2), respectively.

To maintain elementarity it is prohibited to extend a path to an already visited node. Hence, the nodes already visited are unreachable for any extension of the path. Furthermore, some nodes may be unreachable due to resource bounds, as described by Feillet et al. (2004). Therefore, we let $\mathcal{U}(P) \subseteq \mathcal{V}$ be the subset of nodes which are unreachable for any extension of P.

4.1 Dominance

We will distinguish between resource dominance between states and elementary dominance. In resource dominance it is only the resources and the costs which are taken into account, whereas for elementary dominance the unreachable set of nodes is also considered.

Dominance 1 (Resource dominance)**.** *Given two paths $P_1 = (v_0^1, \ldots, v_p^2)$ and $P_2 = (v_0^2, \ldots, v_q^2)$ with $v_p^1 = v_q^2$. Then P_1 resource dominates P_2 if*

1. *$C(P_1) \leq C(P_2)$,*
2. *$T^r(P_1) \leq T^r(P_2)$ for all $r \in \mathcal{R}$,*
3. *$\mathcal{L}(P_1) \neq \mathcal{L}(P_2)$.*

When P_1 resource dominates P_2, we write $\mathcal{L}(P_1) \prec_r \mathcal{L}(P_2)$.

If a path is resource dominated by another path but can still visit some of the unreachable nodes from the other path, then it has the potential to be extended to a lower cost path by visiting the nodes unreachable by the other path. Hence, eliminating the corresponding state based on resource dominance may prohibit the identification of an optimal solution. This leads to a sufficient condition for elementary dominance:

Dominance 2 (Elementary dominance)**.** *Given two paths $P_1 = (v_0^1, \ldots, v_p^2)$ and $P_2 = (v_0^2, \ldots, v_q^2)$ with $v_p^1 = v_q^2$. Then P_1 dominates P_2 if*

1. $\mathcal{L}(P_1) \prec_r \mathcal{L}(P_2)$,

2. $\mathcal{U}(P_1) \subseteq \mathcal{U}(P_2)$.

When P_1 dominates P_2, we write $\mathcal{L}(P_1) \prec \mathcal{L}(P_2)$.

Dominance 2 corresponds to the sufficient condition for dominance described by Feillet et al. (2004), though it does not state the node resources explicitly. It requires the set of unreachable nodes of the dominant path to be a subset of the set of unreachable nodes of the dominated path. While seemingly innocent, this is highly prohibitive for the dynamic programming approaches. The reason is that it is not possible to dominate cost-wise bad paths which can reach some of the unreachable nodes of the resource dominant paths. To overcome this, Chabrier (2006) constructs a stronger dominance relation, where P_1 may dominate P_2 if $|\mathcal{U}(P_1) \setminus \mathcal{U}(P_2)| \leq K$ and the cost of P_1 is sufficiently less than the cost of P_2. For $K \leq 2$ the author demonstrates that this sufficient dominance condition enables elimination of more states than Dominance 2 without impeding the performance of the overall method. While we do not use the dominance criterion described by Chabrier (2006) in the present paper, we comment on how to combine it with our approach in section 5.2.5.

Computationally it is much cheaper to just check resource dominance than to check elementary dominance. Part 2 of Dominance 2 has a linear worst case complexity in the length of the paths on top of the time it takes to evaluate resource dominance. This, along with the increasing number of states, makes the ESPPRC difficult.

4.2 Algorithm

Observe the following: If we extend P to a node $v \in \mathcal{V} \setminus \mathcal{U}(P)$, then the resulting path Q can never reach the nodes in $\mathcal{U}(P)$ nor the node v. As a result, the set of unreachable nodes strictly increases when extending a path. More precisely, $\mathcal{U}(P) \subset \mathcal{U}(P) \cup \{v\} \subseteq \mathcal{U}(Q)$ and consequently $|\mathcal{U}(P)| < |\mathcal{U}(Q)|$. We will exploit this in Algorithm 1.

Below we describe a permanent label setting algorithm for ESPPRC. It exploits the strictly increasing number of unreachable nodes when extending a state. The algorithm we suggest uses the following elements:

- Δ_i is the container for efficient states resident in node $i \in \mathcal{V}$.
- Δ_i^d is the subset of Δ_i, where the paths associated with the states have d unreachable nodes.
- $E_d \subseteq \mathcal{V}$ is the nodes for which unprocessed states with paths having d unreachable nodes may exist.
- **create_init_state**() is a function constructing the initial state. Typically, this is a state for the simple path $P = (o)$ having cost $C(P) = 0$ using the $T^r(P) = a_o^r$ as resource consumption for each resource $r \in \mathcal{R}$ and having the origin node as an unreachable node, i.e., $\mathcal{U}(P) = \{o\}$.
- **succ**(v) is the set of successors of v.
- **extendable**$(\mathcal{L}, v)$ is a function returning true if node v is reachable by extending state $\mathcal{L}$ and false otherwise.
- **extend**$(\mathcal{L}, v)$ creates the extension of state $\mathcal{L}$ to node v.

- **dominated**$(\mathcal{L}, \Delta_j)$ checks whether or not a state within Δ_j dominates $\mathcal{L}$.
- **eliminate_inefficient**$(\mathcal{L}, \Delta_j)$ searches Δ_j for states which $\mathcal{L}$ dominates and eliminates these states.
- **insert**$(\mathcal{L}, \Delta_j\ )$ simply inserts $\mathcal{L}$ into Δ_j.
- **num_unreach**$(\mathcal{L})$ gives the number of unreachable nodes for the path P associated with state $\mathcal{L}$, i.e., $|\mathcal{U}(P)|$.

Special notice should be given to the three methods **dominated**, **eliminate_inefficient**, and **insert**. The efficiency of these depend on the way the states are stored in the state container Δ_j. We will discuss this in detail in section 5.

We suggest the label setting algorithm given in Algorithm 1. It extends states in increasing order of number of unreachable nodes. As we know that the number of unreachable nodes is strictly increasing when extending a state – because of the requirement of elementarity – the state extended will never be dominated using Dominance 2 later in the algorithm. Consequently, this state is considered permanent.

Algorithm 1: Label setting dynamic programming algorithm

```
1  Function EspprcDynProg
2      L^init = create_init_states ()
3      insert(Δ_o, L^init )
4      for d = 1 to d = |V| do
5          E_d = ∅
6      end
7      E_1 = {o}
8      for d = 1 to d = |V| do
9          while E_d ≠ ∅ do
10             Choose v_i ∈ E_d
11             forall the L ∈ Δ_i^d do
12                 forall the v_j ∈ succ (v_i) do
13                     if extendable (L, v_j) then
14                         L' = extend (L, v_j)
15                         if not dominated ( L', Δ_j) then
16                             eliminate_inefficient (L', Δ_j)
17                             insert ( L', Δ_j )
18                             u = num_unreach ( L' )
19                             E_u ← E_u ∪ {v_j}
20                         end
21                     end
22                 end
23             end
24             E_d ← E_d \ {v_i}
25         end
26     end
27 end
```

The algorithm first initializes the necessary elements in lines 2-7. That is, it creates an initial state and inserts it into the state container for the origin node. Then each of the sets of nodes having unprocessed states with d unreachable nodes is set up to be empty for each d. The initialization ends with stating that the origin node has an unprocessed state with one unreachable node.

The main part of the algorithm is given in lines 8-26. The algorithm iterates increasingly through the possible number of unreachable nodes. For each of these iterations the nodes having states with the corresponding number of unreachable nodes are investigated. The set of states having the corresponding number of unreachable nodes are, if possible, extended to the possible successors. If the extension is not dominated by already inserted states, then elimination of already inserted states are commenced and when this is finished, the new state is inserted into the container. Finally, the set of nodes which has unprocessed states with a number of unreachable nodes corresponding to the new state's unreachable nodes, is updated. When all possible extensions have been made from the node it is removed from the set of nodes under consideration. This is continued until no more nodes are present in the set E_d.

In principle, lines 9-25 is a label correcting algorithm, which is run $|\mathcal{V}|$ times. But as an extension will always increase the number of unreachable nodes, a node will only be processed once in this loop. However, this makes it possible to use the algorithm in a state-space relaxation of the problem, where we only have to iterate through the number of non-relaxed nodes, instead of $|\mathcal{V}|$. The loop in lines 9-25 may then be executed more than $|\mathcal{V}|$ times as a result.

5 Storing and updating the efficient states

The set of efficient states, Δ_i, for node i can be stored in different ways. Each way of storing the efficient states has consequences for the efficiency of the insertion, dominance check, and elimination of states. The simplest way is to store the states in a single linked list and then, when extending a state to node i, pass through the list to check whether or not the new state is dominated by another state. If not, then a second pass through the list determines which states in the list are dominated by the newly extended state. If the number of states for the node is small, then this approach fast and easy to implement. However, for harder instances of ESPPRC the number of states tends to increase drastically, and consequently two linear searches through the states yield a corresponding increase in the running time. We will refer to this approach as the single linked list storage (SLLS).

In this section we will discuss two alternative strategies for storing the states. In 5.1 the states are stored in a set of lists exploiting the number of unreachable nodes for the state. In contrast to this, subsection 5.2 gives a tree-based data structure efficiently storing the states using the actual unreachable sets instead of just the number of unreachable nodes. In the tree-based storage we observe that it is not necessary to store $\mathcal{U}(P)$ explicitly for each state, and by implicitly checking part 2 of Dominance 2 it is in fact only necessary to check Dominance 1 explicitly.

5.1 List-based storage

The states in Δ_i can be stored in a single linked list, as described above. However, we can easily exploit additional information on the states to improve the performance of a list-based container of these states. In particular, we can use the pseudo resource described by Feillet et al. (2004) in the following way. Instead of having a single list a set of $|\mathcal{V}|+1$ lists are used – one for each possible number of unreachable nodes for the states. Hence, each state is inserted into the list corresponding to the unreachable nodes of the states. This corresponds to a bucket-based sorting, where each list is a bucket containing the states having a number

of unreachable nodes matching the bucket index. We will refer to this approach as the multiple linked list storage (MLLS).

Then, when checking whether or not a state is dominated by any of the states in the container, only the states in the lists corresponding to the number of unreachable nodes or less may dominate the given state, while the remaining states will never dominate the given state. Therefore, the lists corresponding to a larger number of unreachable nodes than the number of unreachable nodes for the state are never checked.

If the state is not dominated by any of the states in the container, then it may dominate some of the already inserted states. To check this, it is only necessary to search through the lists corresponding to the number of unreachable nodes or higher. The remaining states have less unreachable nodes and can therefore never be dominated by the state at hand.

Whereas sorting the states by number of unreachable nodes decreases the number of dominance checks conducted, each dominance check still has to verify both part 1 and part 2 of Dominance 2 explicitly. If each of the lists is single linked lists, then the insertion of a state has constant worst case complexity.

5.2 Tree-based storage

The list-based storage described in section 5.1 to some extent takes the set-based dominance into account by using the cardinality of the sets to limit the number of dominance checks. In this section we introduce a tree-based data structure which exploits the set-based structure to the full extent. That is, we can identify all the states which have subsets of unreachable nodes of a given set and, likewise, identify all the states which have supersets of unreachable nodes of a given set.

Savnik (2012) discusses how to use a variant of a prefix tree (also known as a Trie) to store sets such that the operations of checking the existence of subsets and supersets as well as enumeration of all subsets and supersets can be done efficiently. The author investigates the existence-checking operations in detail, stating that enumeration operations are extensions of these. In the present paper we develop a variant of the enumeration operations.

Below we describe the basic elements of the variant of a prefix tree we use. This will be followed by a description of how we relate states with the tree. The necessary operations of 1) inserting a state into the tree, 2) checking whether or not a state is dominated by another state in the tree, and 3) eliminating states already inserted into the tree based on a given state is then described. Finally, we make observations related to the prefix tree.

5.2.1 Prefix tree

A prefix key $(\pi_1, \ldots, \pi_p)$ is an increasing sequence of key values, where the possible key values are bounded. We intend to build a tree storing states based on the corresponding prefix key. The idea is that each prefix key corresponds to a unique path from the root node to a specific node in the tree, where the nodes in the tree have key values equal to the element of the prefix key. Suppose that we have a tree with $\mathcal{N} = \{0, 1, \ldots, N\}$ nodes. With each node, $n \in \mathcal{N}$, we associate:

- $key(n) \in \{1, \ldots, |\mathcal{V}|\} \cup \{-1\}$, the key of the node n. The node n has $key(n) = -1$ if and only if it is the root node.
- $par(n) \in \mathcal{N} \cup \{-1\}$, the parent node of node n. If the node is the root node, the parent is set to -1.

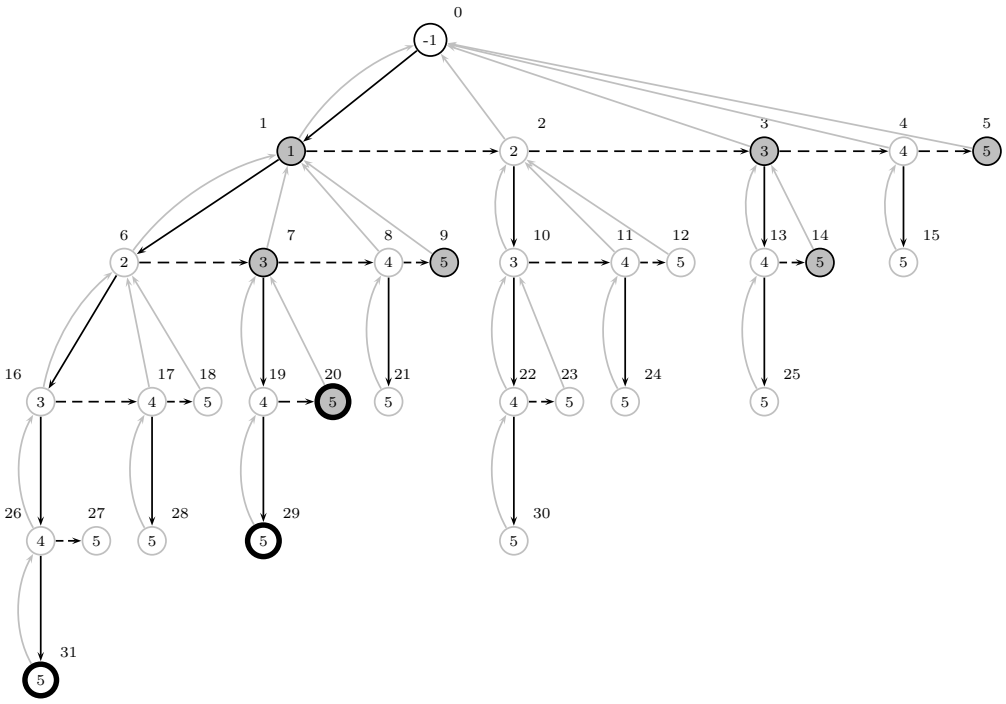

Figure 1: A full prefix tree for all subsets of $\{1, \ldots, 5\}$. The full black arcs correspond to the first child, the dashed black arcs correspond to right sibling and the gray arcs correspond to the parent. The filled gray nodes are the nodes corresponding to subsets of the set $\{1, 3, 5\}$, and the nodes having thick black borders are the nodes corresponding to supersets of $\{1, 3, 5\}$.

- $fc(n) \in \mathcal{N} \cup \{-1\}$, the first child of node n. If the node has no children, then this is -1. The first child should be the child node with the lowest key value.
- $rs(n) \in \mathcal{N} \cup \{-1\}$, the right sibling of node n. If the node has no right sibling, then this is -1.
- $\mathcal{S}(n)$, the set of states contained in the node. This is stored as a single linked list.

These parameters constitute the tree and make the traversals we need possible. For simplicity we let node 0 be the root node of the prefix tree. The tree maintains the following invariants:

1. For any node n with $rs(n) \neq -1$: $par(n) = par(rs(n))$.
2. For any non-root node n: $key(n) > key(par(n))$.
3. For any node n with $rs(n) \neq -1$: $key(n) < key(rs(n))$.

The first part of the tree invariant states that a node has to have the same parent as its right sibling. The second tree invariant states that any child has to have larger key value than

its parent node. Consequently, a path from the root node to a leaf node will correspond to a strictly increasing sequence of keys. Finally, the third invariant states that moving from left to right through the siblings also yields a strictly increasing sequence of keys.

Figure 1 is an example of a full prefix tree for the set $\{1, \ldots, 5\}$ having a node for each possible subset and where the root node corresponds to the empty set. The numbers above the nodes are the node indices from $\mathcal{N}$, whereas the numbers given within the nodes are the keys of the nodes. The full black arcs correspond to the first child of the relation of the tail node, and the dashed black arcs indicate the right sibling of the tail node. The gray arcs point from a node to its parent node. We have only included arcs where both head and tail are within $\mathcal{N}$.

A path from the root node using only arcs corresponding to the first-child and right-sibling relations corresponds to a search for a specific set. If the first-child relation is used, then the key of the tail node is included into the set, except if the tail node is the root, whereas if the right-sibling relation is used, then the value is excluded from the set. In the example a path $(0, 1, 6, 16, 17, 28)$ corresponds to the set $\{1, 2, 4, 5\}$. Each of the nodes corresponds to a specific subset of $\{1, \ldots, |\mathcal{V}|\}$. Using the parent relation, we can obtain the corresponding subset directly by the arcs traversed. Again in the example, the path from node 22 to the root node 0 using the parent relation is $(22, 10, 2, 0)$ yielding the set $\{2, 3, 4\}$.

Identifying all subsets of a given set requires traversal of parts of the tree. Intuitively, if a node in the tree has a key which is not in the subset, then neither the node nor any of the nodes in the subtree can be part of a subset of the set we are searching for. In Figure 1 the nodes corresponding to subsets of the set $\{1, 3, 5\}$ are indicated by filled gray nodes. The traversal skips any node and subtree of the node, having a key equal to either two or four.

Likewise, all supersets of a given set can be identified by traversal of the tree. Here it is the skipping of nodes which reduces the number of nodes traversed. That is, we will never go to the right sibling of a node having a key which is included in the set, as this corresponds to excluding the key from the sets identified. In the example of Figure 1 the nodes corresponding to supersets of $\{1, 3, 5\}$ are the nodes with thick black borders. Here it is not possible to use the right-sibling relationship for nodes with keys one, three, or five. For instance, we will never use the arc from node 1 to node 2, as it corresponds to skipping key one from node 1. Thereby, half of the tree is excluded from the traversal. Note that, if we identify a path to a node including all the keys from the set, then all of the nodes in the subtree will correspond to supersets of the set we are searching for.

If the root has depth zero, then the depth of the nodes correspond to the number of included keys in the set. The maximal depth is equal to the number of possible keys. Furthermore, as the keys are strictly increasing both when selecting the first child and when selecting the right sibling we have that the longest path in the tree uses no more arcs than the number of possible keys.

If all possible subsets are represented by a node in the prefix tree, then the cardinality of $\mathcal{N}$ will be $2^{|\mathcal{V}|}$. Consequently, we are interested in not having a node unless it is necessary. That is, we only include a node in the tree if a set of unreachable nodes corresponds to either the node or a node in the subtree. Hence, when adding states to the prefix tree we also add the nodes necessary to represent the sets of unreachable nodes for the states, thus gradually building the tree.

To relate the set of unreachable nodes with the prefix tree described above, we will use an auxiliary function giving each node in $\mathcal{V}$ a unique number and then state the set as an increasing sequence of the numbers of the unreachable nodes. The formal definition is:

Definition 1. *Given a path P and a bijective function $\Pi : \mathcal{V} \to \{1, \ldots, |\mathcal{V}|\}$, let $\mathcal{U}(P) =$*

$\{v^1, \ldots, v^u\}$ *be ordered such that* $\Pi(v^d) < \Pi(v^{d+1})$ *for* $d = 1, \ldots, u-1$ *and denote* $\pi_d = \Pi(v^d)$. *Then a prefix key for the path* P *is the increasing sequence* $\pi = (\pi_1, \ldots, \pi_u)$.

Note that, as Π is bijective, not only are the elements of the prefix key unique but the inverse of these elements is also unique, i.e., $\Pi^{-1}(\pi_d) = v^d$. Hence, we can easily obtain the original set of unreachable nodes when we know the prefix key π. A simple way to construct Π would be to use the predetermined ordering of the nodes given by the problem instance and thereby maintain the relative ordering of the nodes. As we will discuss below, this is not always the best choice, but it is an intuitive starting point.

5.2.2 Inserting a state into the tree

A state is only inserted into the state container if it is not dominated by other states in the container. Insertion of a state is based on a simple recursive algorithm, which is stated in Algorithm 2. It has to search for the node corresponding to the prefix key of the state. Algorithm 2 is divided into two functions. The first function, **insert**$(\mathcal{L}, \Delta)$, retrieves the prefix key, π, from the state, $\mathcal{L}$, and determines the root node of Δ. Then the second function, **recursive_insert**$(\mathcal{L}, \pi, n)$, is called with the root-node as n. The second function is stated in lines 6-23. It recursively calls itself until it reaches the depth of the node which is searched for. The depth of the node is exactly equal to the length of π. When the node is found, the state is inserted into the state set $\mathcal{S}(n)$ in line 21. On the other hand, if the

Algorithm 2: Prefix tree-based insertion of a state

```
1  Function insert (L,Δ)
2      π =get_prefix_key (L)
3      root = get_root (Δ)
4      recursive_insert (L,π,root)
5  end
6  Function recursive_insert (L,π,n)
7      if depth(n) < len(π) then
8          d = depth(n) + 1
9          c = fc(n) // child
10         pc = −1 // previous child
11         while c ≠ −1 and key(c) < π_d do
12             pc = c
13             c = rs(c)
14         end
15         if c = −1 or key(c) > π_d then
16             c = add_child (n, π_d, pc) // add a right sibling to pc
17         end
18         recursive_insert (L,π,c)
19     end
20     else
21         S(n) = S(n) ∪ {L}
22     end
23 end
```

depth of the node has not yet been reached, then we search for the node among the children having a key equal to π_d, where d is the depth of the child nodes. This search is conducted in lines 8-18 by iteratively passing through the right siblings of the first child. We either identify a child node having a key equal to π_d, in which case we found the intended node at that depth, or we reach a child node which either has a larger key or does not exist. In the

latter case a new node is added with key π_d. At this point we have identified the correct child node and the function is called recursively.

The insertion function only searches through at most $|\mathcal{V}|$ nodes before identifying the node, n, in which the state $\mathcal{L}$ can be inserted. This is due to the strictly increasing keys of the first child and the right sibling. The addition of a new node can be performed in $O(1)$ time complexity, and the insertion therefore requires at most $O(|\mathcal{V}|)$ time. This is clearly slower than the SLLS and the MLLS approaches, which had $O(1)$ insertion time.

5.2.3 Checking dominance

Each time a state is extended, it has to be checked whether or not the new state is dominated by another already inserted state. It is therefore the most frequently used dominance operation, and we describe it in Algorithm 3.

Algorithm 3: Prefix tree-based dominance check

```
1  Function dominated (L,Δ)
2      π =get_prefix_key (L)
3      root = get_root (Δ)
4      return recursive_dominated (L, π, root, 0)
5  end
6  Function recursive_dominated (L,π,n, s)
7      dom = false
8      d = depth(n) + s + 1
9      if d ≤ len(π) then
10         c = fc(n) // child
11         t = 0
12         while c ≠ −1 and not dom and d + t ≤ len(π) do
13             if key(c) = π_{d+t} then
14                 dom = recursive_dominated (L,π,c, s + t )
15                 if not dom then
16                     foreach L' ∈ S(child) do
17                         if L' ≺_r L then return true
18                     end
19                 end
20             end
21             else if key(c) > π_{d+t} then t = t + 1
22             else c = rs(c)
23         end
24     end
25     return dom
26 end
```

Like the insertion described above, Algorithm 3 is split into two function: one which initializes the dominance check and one which recursively explores the tree. The initialization, in line 2-3 is done in the same way as the initialization of the insertion. Then the recursive dominance check is called from the root node.

Recall that the depth of a node corresponds to how many of the keys from the prefix key are included in the specific set related to the node. We use the input parameter s to indicate how many keys are skipped from the prefix key in order to reach the node. Consequently, $d = depth(n) + s + 1$ indicates the position in the prefix key with the value π_d we intend to search for among the child nodes. In lines 12-23 we search through the children for the key-values of the prefix key. During this search we may skip keys from the prefix key, and

t counts the number of times we have skipped such a key value. If a child have key value π_{d+t}, then the method is called recursively, and we check whether or not any of the states included in the child node dominates the input state. If the child node has a larger key value than π_{d+t}, then we have skipped the current key, and t is incremented. Finally, if the key value is less than π_{d+t}, then we have not yet found the key value we are searching for, and we proceed to the right sibling.

In the worst case, we have to traverse the whole tree and check each of the nodes for dominant states. In this case, the PFTS is not better than the SLLS, as the same number of states have to be checked. However, remember that part 2 of Dominance 2 is automatically checked by the traversal of the tree, which may improve the performance of PFTS over SLLS. Next, if only a few of the nodes in the tree have non-empty sets $\mathcal{S}(n)$ for $n \in \mathcal{N}$, then PFTS's dominated method has to do more work to identify these nodes than the work required for the corresponding method for SLLS. In the case where it is not necessary to traverse the whole tree and many or most of the nodes $n \in \mathcal{N}$ have non-empty sets $\mathcal{S}(n)$ PFTS can be considerably faster, as it may exclude a large portion of Δ_i from the dominance check.

5.2.4 Elimination of inefficient states

Algorithm 4: Prefix tree-based elimination of inefficient states

```
1  Function eliminate_inefficient (L,Δ)
2      π =get_prefix_key (L)
3      root =get_root (Δ)
4      recursive_eliminate_inefficient (L,π,root,0)
5  end
6  Function recursive_eliminate_inefficient (L,π,n,e)
7      d = depth(n) − e
8      c = fc(n)
9      if d < len(π) then
10         while c ≠ −1 and key(c) ≤ π_d do
11             if key(c) = π_d then recursive_eliminate_inefficient (L,π,c,e)
12             else recursive_eliminate_inefficient (L,π,c,e + 1)
13             c = rs(c)
14         end
15     end
16     else
17         while c ≠ −1 do
18             recursive_eliminate_inefficient (L,π,c,e + 1)
19             c = rs(c)
20         end
21         foreach L' ∈ S(n) do
22             if L ≺_r L' then
23                 S(n) = S(n) \ {L'}
24             end
25         end
26     end
27 end
```

When it has been determined that a newly extended state is not dominated by any state in the container Δ, then it can be used to eliminate states which it dominates in Δ. We provide a recursive method for this in Algorithm 4.

As for the insertion and the dominated check we have split this into two parts: one which initializes the necessary components (lines 1-5), and one which is recursively called to traverse the tree (lines 6-27). This initialization is equivalent to the one of Algorithms 2 and 3 except for the call to the **recursive_eliminate_inefficient** method.

Again, the depth of the node corresponds to the number of keys included in the set corresponding to the node n. Hence, we need to keep track of which keys are actually part of the prefix key π and which are extra keys inserted because we are searching for supersets. We let e be the number of extra keys used in the path from the root to node n compared to the position in the prefix key. Then we can calculate the position in the prefix key which we are searching for as $d = depth(n) - e$.

If $d < len(\pi)$, then we still have to add elements from the prefix key to the set in order to obtain a superset. We do this in lines 9-15. In this case, each of the children (with keys no larger than π_d) is checked to ascertain if they have a matching key. If the key is not matched, then the key of the child node is an extra key, and the method is called recursively stating that we have included an extra key. If it is a match, then the method is called recursively with the same number of extra keys. Note that we will never skip a key π_d because otherwise we would not obtain a superset.

On the other hand, if $d \geq len(\pi)$, then all elements of the prefix have been included in the set, and all nodes in the subtree correspond to supersets of π. This is described in lines 16-26. Clearly all states included in the superset nodes have to be dominance checked.

If the set $\mathcal{U}(P)$ is small, then it may have many supersets in the tree, and we may have to traverse most of the tree. In this case the SLLS may be more efficient compared to the PFTS. This is also true if the number of nodes in the tree having non-empty $\mathcal{S}(n)$ sets is small, as we then have to traverse many unnecessary nodes. However, during the course of the algorithm the number of unreachable nodes will increase and consequently we have to check relatively fewer nodes compared to the case where we had only a few unreachable nodes. Furthermore, having long prefix keys will in general result in cutting off large portions of the tree, because some of the key values are not present in these parts. Again, when the number of nodes having non-empty sets $\mathcal{S}(n)$ is high, then more states will not be checked, and the PFTS will become faster than SLLS and MLLS.

5.2.5 Remarks

For each realization of subsets $\mathcal{U} \subseteq \mathcal{V}$ it can be observed that there are many paths leading to the set of unreachable nodes corresponding to $\mathcal{U}$. Therefore the number of states contained in node n may be large. The number of undominated states will tend to increase if all resources are arc-based instead of node-based. This is because node-based resources will tend to have the same value if having visited the same set of node.[1] Thus, it is expected that it is more difficult to solve problems where resources are arc-based rather than node-based.[2]

A state will be stored at the same depth of the tree regardless of the selection of Π. This is due to the fact that the depth corresponds to the number of unreachable nodes for the state which is independent of the selection of Π. This could lead one to think that arbitrary

[1] Note that due to other resource constrains we may have some nodes are unreachable and therefore the corresponding node-based resource is not accumulated for these. Hence, even though the set of unreachable nodes is the same for two paths the amount of accumulated node-based resource may not be the same for the two paths.

[2] Clearly it is possible to transform node-based resources into arc-based resources, so the more distinct the arc resource consumption is for different arcs into the same node, the more difficult the problem will tend to be.

choices of Π are equally good. This is not true, however. Suppose that we have a single node v' which is never reachable for any extension and $\Pi(v') = |\mathcal{V}|$. Then any prefix key will have $|\mathcal{V}|$ as the largest value. As a consequence, each prefix key will end in a leaf node having key $|\mathcal{V}|$ i.e. the number of nodes in the tree will be equal to at least the number of different prefix keys. On the other hand, if we change Π such that $\Pi(v') = 1$, then only a single node, as a child of the root, is added to the tree having a key equal to 1. This yields a huge memory saving. Intuitively, the more likely it is for a node to be unreachable the smaller value it should have in the Π-function.

Two states, $\mathcal{L}^1$ and $\mathcal{L}^2$, the first resident in node n, and the second resident in $par(n)$ will always have the relation $\mathcal{L}^2 \not\prec_r \mathcal{L}^1$, i.e., the state in the parent node will never resource dominate the state in the child node. If the state in the parent node did in fact resource dominate the state in the child node, then the state in the child node would not only be resource dominated but also dominated according to Dominance 2, and could thereby be eliminated. This is clearly transitive, and therefore we have that no state in the subtree of a given node will ever be resource dominated by a state from that node.

It is possible to extend the prefix tree-based storage to accommodate the dominance relation given by Chabrier (2006). Remember that the requirement is $|U(P_1) \setminus U(P_2)| \leq K$ for $\mathcal{L}(P_1)$ to have the chance of dominating $\mathcal{L}(P_2)$. Thus, when we search for states which may dominate a given state and having found a node corresponding to a subset, the part of the subtree of this node having depth no deeper than the node's depth plus K has to be included in the dominance check. In this way, we include up to K additional keys into the set we are searching for. Likewise, when searching for supersets, we may exclude up to K keys from the superset, which is done by using the right-sibling arcs in the tree. That is, we are allowed to use up to K right sibling arcs having tails in nodes with keys included in the set for which we are trying to identify supersets.

If all sets are represented by nodes in the prefix tree, then an alternative view of the tree is as a balanced binary search tree. In each node we have the choice between including the key in the set or excluding the key from the set. This corresponds to selecting the first child or the right sibling in the prefix tree. However, this is not directly possible if some of the nodes are left out of the prefix tree.

If the bidirectional approach described by Righini and Salani (2006) is applied, it is possible to exploit the PFTS to enhance the merging of states necessary. Recall that the bidirectional approach uses a forward pass, where only states below a certain threshold are extended, and a backward pass, where states are only extended if they, too, are below a specific threshold. After this, the states from the two passes have to be merged. Here Righini and Salani (2006) suggest to run through the lists of states for pairs of nodes. This can be enhanced by the PFTS where a state $\mathcal{L}(P^1)$ from Δ_i in the forward pass can be merged with a state $\mathcal{L}(P^2)$ from Δ_j from the backward pass if $\mathcal{U}(P^1) \cap \mathcal{U}(P^2) = \emptyset$. Hence, we select each state $\mathcal{L}(P^1)$ from Δ_i and search for states $\mathcal{L}(P^2)$ in Δ_j from the backward pass having $\mathcal{U}(P^2) \subseteq \mathcal{V} \setminus \mathcal{U}(P^1)$. This search yields all the pairs of states which are elementary compatible, and a check of whether or not they are resource compatible then has to be commenced.

6 Computational Experiments

We have conducted a set of experiments based on an implementation of the approach described in the previous sections. The main purpose of these experiments is to observe the

behavior of the dynamic programming algorithm when we change the storage method. The performance of the storage methods depends on the number of states generated and we therefore construct instances which have very few states as well as instance which result in a large number of states.

In 6.1 we give further details on the actual implementation. A set of test instances has been generated and we describe these in 6.2. Finally, the computational set-up as well as the results of the experiments are described in section 6.3.

6.1 Implementational details

With each state it is necessary to be able to obtain $\mathcal{U}(P)$. This can be done by either identifying it each time it is necessary or by storing it after it has been identified the first time. This is the classical trade-off between time consumption and memory consumption. For LSSL and MSSL we choose to store $\mathcal{U}(P)$ after it is obtained the first time because it is necessary each time Dominance 2 has to be checked. But as part 2 of Dominance 2 is not checked explicitly when using PFTS, we do not store $\mathcal{U}(P)$ for PFTS.

Each extended state has a counter on the number of successful extensions it has which are not eliminated. When a state is eliminated, then the counter of the predecessor state is decremented. If this counter becomes zero, then all of the extensions of the state have been dominated, and it is not necessary to keep this state any more. Consequently, it can be eliminated, too. Clearly, this is recursive. In practice we mark these as dominated and eliminate them prior to extending states from the corresponding resident node.

When having eliminated inefficient states in the tree-based approach, we may have removed the last state resident in the node of the tree. The node is not explicitly removed from the tree, however. Instead, we have a recursive procedure which eliminates all nodes having no states in the corresponding subtree. It is executed prior to extending states from the container.

For PFTS, deriving the set Δ_i^d is easily done by traversing the tree associated with the container Δ_i and selecting the states in the nodes having depth d. This, of course, becomes increasingly expensive throughout the algorithm as a larger part of the tree has to be traversed. This is in contrast to the list-based approach, MLLS, where Δ_i^d corresponds to the list of states having d unreachable nodes.

As we are using instances based on the Solomon instances (see Section 6.2), we know both traveling time t_{ij} along arcs (i, j) and the time window $[a_j; b_j]$. For a given node i the latest departure time at which we can reach the node j is $b_j - t_{ij}$, after which it is not possible to reach node j anymore. Therefore, we set up Π in increasing order of latest departure time and thereby have the nodes which are unreachable at the earliest time at the front of the prefix key, whereas the nodes which become unreachable latest are placed at the end of the prefix key.

6.2 Test instances

Solomon (1987) constructs six sets of benchmark instances for the Vehicle Routing Problem with Resource Constraints. These are subdivided into three pairs of instance sets: one which has clustered customers (C), one which has randomly dispersed customers (R), and one having a combination of the two previous (RC). These subsets of instances are then subdivided into instances having narrow resource windows and instances having wide resource windows. The sets of instances having narrow resource windows are referred to as

the 1-instances and those having wide resource windows are referred to as the 2-instances. Each of the instances has 100 customers, but are often trunkated to the 25 first customers or 50 first customers.

The distances between the nodes are trunkated euclidian distances, i.e.,

$$d_{ij} = \frac{1}{10} \left\lfloor 10\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right\rfloor \tag{3}$$

where node i has coordinated (x_i, y_i). The time for traveling between customers is equal to the distance. Hence the time resource is highly correlated with the distance traveled. Furthermore, the instances have a specified demand and a capacity for demand covered for each of the homogeneous vehicles. Consequently, the problem has two resources. One which is node-based and one which is highly correlated with the cost function.

We have constructed 54 test instances based on the Solomon benchmark instances having wide resource windows. 27 of the instances have 50 customers, and 27 have 100 customers. Typically, the more negative cost arcs there are in the instance the harder the problem becomes to solve for the dynamic programming algorithms. This is due to the time resource becoming negatively correlated with the cost, and therefore it is not possible to dominate as many states. As we are interested in having some easy instances and some hard instances, we subtract a virtual profit from each arc which is partly based on the head node and partly based on the arc itself. For each node $i \in \mathcal{C}$ let X_i be drawn from the uniform distribution $U(0; M_1)$, and for each arc $(i, j) \in \mathcal{A}$ we let Y_{ij} be drawn from the uniform distribution $U(0; M_2)$. Then we use the cost $c_{ij} = d_{ij} - X_j - Y_{ij}$. For the instances with 50 customers we let $M_1 = M_2 = 10.0$, and for the instances with 100 customers we let $M_1 = M_2 = 5.0$. The reason for the different selection of parameters is that these balance the number of easy with the number of hard instances for each number of customers.

6.3 Results

We have conducted experiments on the instances described in Section 6.2. The dynamic programming algorithm has been implemented using C++ and compiled with GNU g++ 4.4.7 compiler using the -O3 flag. The experiments have been performed on a computer equipped with an Intel(R) Xeon(R) E5-1620 CPU and 24Gb RAM using a Linux operating system. We have imposed a two-hour time limit on the execution.[3]

In tables 1 and 2 we report the results of the experiments. Here the instance (I) is given along with the percentage of negative arcs (% neg.). Then for each of the three storage methods, SLLS, MLLS, and PFTS, the time (Time), the stage (St.), and the number of undominated states at termination (U.S.) are given. If the time is above 7200 seconds, then the instance is terminated prematurely due to the time limit. The stage then corresponds to the stage reached on termination. Note that we record the last stage where extensions have actually taken place and none of the remaining stages have produced any states inserted into the containers. The time taken from the last active stage to stage $|\mathcal{V}|$ is negligible.

For the instances with 50 customer nodes we see that SLLS is at least as fast as the other storage methods in seven out of the 27 instances, whereas MLLS is at least as fast as SLLS and PFTS in four out of the 27 instances. The PFTS storage method is no slower than the other methods in 13 out of 27 instances.[4] As soon as the number of states becomes large

[3]The check of the time limit is carried out each time a new successor node is selected, i.e. in line 12 of algorithm 1. Hence, we always allow finishing all extensions from one node to another.

[4]In some cases the running times are identical, and these instances may be counted more than once.

		SLLS			MLLS			PFTS		
I	% neg.	Time	St.	U.S.	Time	St.	U.S:	Time	St.	U.S.
c201.50	9.56	0.06	49	1,217	0.06	49	1,217	0.07	49	1,217
c202.50	9.21	68.66	51	93,742	56.97	51	93,742	9.12	51	93,742
c203.50	9.10	7415.83	22	1,394,556	7223.84	22	1,607,783	7200.08	36	13,129,556
c204.50	9.94	7738.78	8	2,669,316	7268.46	8	2,746,553	7212.52	10	39,219,765
c205.50	13.96	0.16	51	2,703	0.18	51	2,703	0.18	51	2,703
c206.50	12.78	0.43	50	8,083	0.44	50	8,083	0.4	50	8,083
c207.50	13.18	33.29	51	74,414	29.67	51	74,414	7.44	51	74,414
c208.50	12.30	1.23	50	14,363	1.17	50	14,363	1.01	50	14,363
r201.50	5.27	0.04	49	260	0.04	49	260	0.04	49	260
r202.50	6.25	4.31	46	30,398	3.87	46	30,398	3.27	46	30,398
r203.50	5.97	7213.24	30	1,845,503	7202.88	31	2,159,993	1660.92	50	3,354,928
r204.50	5.41	7200.13	12	2,905,738	7241.26	12	3,310,501	7201.14	16	25,021,429
r205.50	5.27	0.16	44	1,237	0.16	44	1,237	0.18	44	1,237
r206.50	6.55	316.58	49	386,622	258.34	49	386,622	86.75	49	386,622
r207.50	3.57	31.81	47	83,811	26.8	47	83,811	20.66	47	83,811
r208.50	5.94	7201.23	10	2,873,925	7359.8	10	3,265,207	7223.65	12	35,146,200
r209.50	6.51	5.15	47	30,604	4.74	47	30,604	5.82	47	30,604
r210.50	4.71	42.43	44	112,463	34.19	44	112,463	28.16	44	112,463
r211.50	6.70	7253.74	14	2,718,784	7200.12	14	2,745,058	7203.57	17	4,769,108
rc201.50	13.85	0.05	49	607	0.05	49	607	0.04	49	607
rc202.50	11.81	0.37	46	4,477	0.38	46	4,477	0.44	46	4,477
rc203.50	12.17	7285.63	31	1,840,503	7282.85	31	2,002,558	5936.03	51	9,820,636
rc204.50	12.46	7288.17	11	2,399,211	7201.83	11	2,552,646	7225.61	15	37,763,645
rc205.50	13.52	0.28	50	3,399	0.3	50	3,399	0.32	50	3,399
rc206.50	12.69	0.17	45	2,069	0.18	45	2,069	0.2	45	2,069
rc207.50	13.35	144.55	47	211,225	127.96	47	211,225	51.94	47	211,225
rc208.50	12.32	7221.80	12	2,815,911	7213.51	12	3,005,654	7200.23	14	12,173,535

Table 1: Results for 50 instances

		SLLS			MLLS			PFTS		
I	% neg.	Time	St.	U.S.	Time	St.	U.S.	Time	St.	U.S.
c201.100	1.72	0.14	100	877	0.17	100	877	0.18	100	877
c202.100	1.64	8.37	98	23,820	8.01	98	23,820	6.5	98	23,820
c203.100	1.63	1277.37	100	358,009	1031.86	100	358,009	200.52	100	358,009
c204.100	1.90	7743.77	16	1,883,482	7581.88	16	2,047,535	7202.7	21	15,548,964
c205.100	2.79	0.43	99	2,430	0.5	99	2,430	0.52	99	2,430
c206.100	2.82	0.71	97	3,665	0.81	97	3,665	0.9	97	3,665
c207.100	2.43	2.08	96	14,066	2.36	96	14,066	2.56	96	14,066
c208.100	2.44	1.25	97	6,784	1.4	97	6,784	1.62	97	6,784
r201.100	1.54	0.16	94	661	0.19	94	661	0.2	94	661
r202.100	1.91	28.66	96	67,930	28.06	96	67,930	25.95	96	67,930
r203.100	1.60	7200.28	18	2,338,512	7210.83	18	2,641,092	7201.33	33	6,406,248
r204.100	1.96	7223.65	10	2,676,779	7232.27	10	2,753,918	7201.65	12	27,377,153
r205.100	1.57	0.69	88	2,298	0.78	88	2,298	0.89	88	2,298
r206.100	1.80	7203.31	73	1,260,741	7214.41	75	1,368,715	2385.57	101	1,443,978
r207.100	1.48	7274.79	16	1,782,125	7203.79	17	1,886,187	7200.01	30	6,438,110
r208.100	1.51	7710.53	10	2,573,874	7202.99	10	2,687,185	7209.96	12	18,827,208
r209.100	2.03	21.53	86	54,702	21.68	86	54,702	41.74	86	54,702
r210.100	1.60	243.63	91	241,325	215.81	91	241,325	208.76	91	241,325
r211.100	1.68	7200.94	12	3,372,368	7223.24	12	3,361,297	7200.08	14	6,934,562
rc201.100	1.91	0.16	99	553	0.19	99	553	0.19	99	553
rc202.100	1.47	0.88	92	2,780	0.96	92	2,780	1.09	92	2,780
rc203.100	1.60	8.16	95	21,747	8.54	95	21,747	9.91	95	21,747
rc204.100	1.46	7200.47	35	1,387,056	7201.38	42	1,591,111	2239.72	97	1,533,980
rc205.100	1.91	0.62	96	2,363	0.72	96	2,363	0.79	96	2,363
rc206.100	1.49	0.66	88	2,206	0.72	88	2,206	0.82	88	2,206
rc207.100	1.79	2.93	81	9,655	3.04	81	9,655	4.73	81	9,655
rc208.100	1.74	2258.09	75	647,951	2101.28	75	647,951	5871.32	75	647,951

Table 2: Results for 100 instances

(in our case above 30604 states), we observe that PFTS is consistently the fastest method. Furthermore, we observe that for the instances which are not solvable within the time limit

using any of the methods, the PFTS handles far more undominated states compared to the two other methods.

When turning to the instances with 100 customer nodes we observe that SLLS is the fastest method for 14 out of the 27 instances. These 14 instances are, however, among the easiest of the instances. In contrast, the PFTS is the fastest method for six of the 27 instances. Most of these are among the hardest of the instances. The MLLS is only fastest for a single instance. Like for the instances having only 50 nodes we observe that for those which cannot be solved within the time limit a stage is reached where significantly more undominated states exist for the PFTS than for SLLS and MLLS.

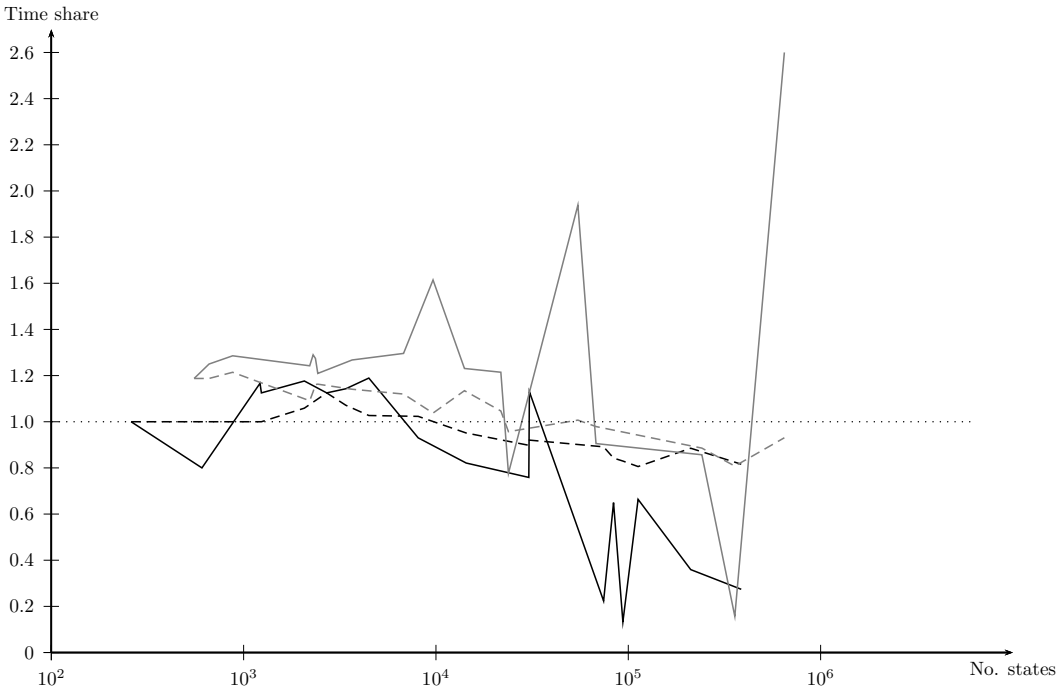

Figure 2: Relative performance for number of states for instances having less than a million states at termination. The dotted line is the baseline corresponding to the time used by the SLLS approach. The dashed lines are the time use by the MLLS divided by the base time. The full lines are the PFTS time divided by the base time. The black curves correspond to the 50 customer instances, and the gray curves correspond to the 100 customer instances.

We are interested in measuring the relative performance of the different ways of storing the states. As we impose a time limit, the same stage may not be reached by all of the methods. Hence, to compare the performance we identify the point in the algorithm in which the slowest method terminates and record the time it took for the other methods to reach the exact same point. We choose the time it takes for SLLS to reach this point as a base time and divide the time it takes for MLLS and PFTS, respectively, by this base time. The value of this is given in Figures 2 and 3, where it is stated as a function of the number of states generated when reaching the stage. Figure 2 shows this for less than a million states, whereas Figure 3 illustrates the cases having more than a million states.

For the 50 customer instances we see that when the number of states increase, then the

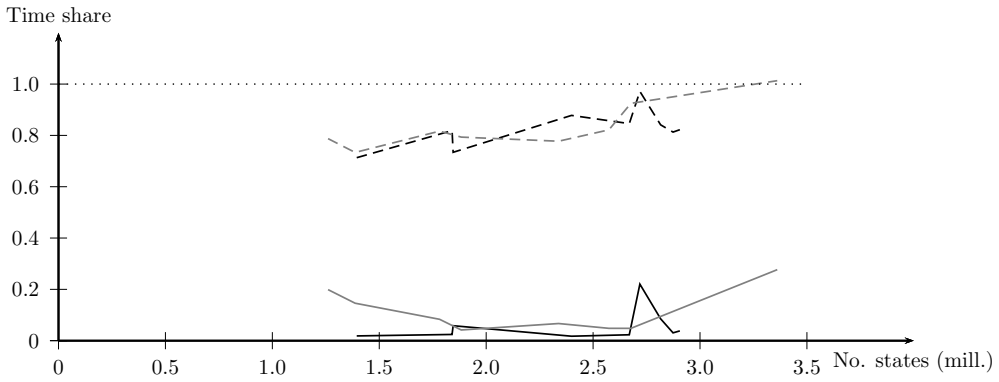

Figure 3: Relative performance for number of states for instances having more than a million states at termination. The dotted line is the baseline corresponding to the time used by the SLLS approach. The dashed lines are the time use by the MLLS divided by the base time. The full lines are the PFTS time divided by the base time. The black curves correspond to the 50 customer instances, and the gray curves correspond to the 100 customer instances.

MLLS gets slightly faster than the SLLS. The PFTS gets significantly faster, however. Note that nine of these instances have more than a million states for SLLS on termination, and PFTS uses at most a quarter of the time to reach the same point in the DP. Furthermore, in some cases PFTS is more than 50 times faster at reaching the point where SLLS terminates.

To some extent we can make the same observations for the 100-customer instances as for the 50-customer instances. However, the number of states needed for the PFTS to become more efficient than the two other methods has increased. Eight instances have more than a million states for SLLS on termination. For these eight instances the PFTS reaches the same point in the DP between four and 20 times faster.

When the number of states is not large, then the cost of insertion and the traversal time of the tree outweighs the gain from the reduced number of dominance checks. When we increase the number of customers, then we will have an increase in the depth of the tree for PFTS, and therefore the gain from using the tree-based storage only becomes clear when the number of states increases beyond a million states. This is exemplified by the tests for r209.100 and rc208.100.

7 Conclusion and further research

In this paper we have demonstrated an approach for efficiently handling a large number of states in dynamic programming approaches for ESPPRC. It is based on a prefix tree data structure and makes it possible to check dominance only for states having sub- or supersets of unreachable nodes. We show that when the number of states grows large, then it is worthwhile to use this approach, while it is better just to use a single linked list when the number of states is small. We have also demonstrated a dynamic programming approach leading to a label setting algorithm rather than a label correcting algorithm.

In our implementation of the PFTS we have not made any attempts to compress the tree. That is, we observe the number of times each key is used and then update the function Π to have the most used keys closest to the root of the tree. Furthermore, if some nodes

from $\mathcal{V}$ can never be reached, then they could be removed altogether and thereby decrease the traversal time in the tree.

We have tested the PFTS on a mono-directional dynamic programming algorithm. As discussed in section 5 it is possible to use the PFTS in the bi-directional dynamic programming algorithm. Furthermore, it can easily accommodate the state space relaxation-based algorithms. We will leave this for future research, however.

References

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

Boland, N., Dethridge, J., and Dumitrescu, I. (2006). Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34(1):58–68.

Chabrier, A. (2006). Vehicle routing problem with elementary shortest path based column generation. *Computers & Operations Research*, 33(10):2972–2990.

Desaulniers, G., Desrosiers, J., Ioachim, I., Solomon, M. M., Soumis, F., and Villneuve, D. (1998). A unified framework for deterministic time constrained vehicle routing and crew scheduling problems. In Crainic, T. G. and Laporte, G., editors, *Fleet Management and Logistics*, pages 57–93. Kluwer.

Desrochers, M., Desrosiers, J., and Solomon, M. (1992). A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2):342–354.

Desrochers, M. and Soumis, F. (1988). A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR*, 26(3):191–212.

Drexl, M. and Irnich, S. (2012). Solving elementary shortest-path problems as mixed-integer programs. *OR Spectrum*, pages 1–16.

Dror, M. (1994). Note on the complexity of the shortest path models for column generation in vrptw. *Operations Research*, 42(5):977–978.

Dumitrescu, I. and Boland, N. (2003). Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, 42(3):135–153.

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

Gendreau, M., Laporte, G., and Semet, F. (1998). A branch-and-cut algorithm for the undirected selective traveling salesman problem. *Networks*, 32(4):263–273.

Gualandi, S. and Malucelli, F. (2012). Resource constrained shortest paths with a super additive objective function. In Milano, M., editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 299–315. Springer Berlin Heidelberg.

Ibrahim, M. S., Maculan, N., and Minoux, M. (2009). A strong flow-based formulation for the shortest path problem in digraphs with negative cycles. *International Transactions in Operational Research*, 16(3):361–369.

Irnich, S. (2008). Resource extension functions: properties, inversion, and generalization to segments. *OR Spectrum*, 30(1):113–148.

Irnich, S. and Desaulniers, G. (2005). Shortest path problems with resource constraints. In Desaulniers, G., Desrosiers, J., and Solomon, M., editors, *Column Generation*, pages 33–65. Springer US.

Irnich, S. and Villeneuve, D. (2006). The shortest-path problem with resource constraints and k-cycle elimination for $k \geq 3$. *INFORMS J. on Computing*, 18(3):391–406.

Kohl, N. (1995). *Exact Methods for Time Constrained Routing and Related Scheduling Problems*. PhD thesis, Technical University of Denmark.

Mehlhorn, K. and Ziegelmann, M. (2000). Resource constrained shortest paths. In Paterson, M., editor, *Algorithms - ESA 2000*, volume 1879 of *Lecture Notes in Computer Science*, pages 326–337. Springer Berlin Heidelberg.

Powell, W. B. and Chen, Z.-L. (1998). A generalized threshold algorithm for the shortest path problem with time windows. In Pardalos, P. M. and Du, D., editors, *Network Design: Connectivity and Facilities Location*, volume 40 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 303–318. American Mathematical Society.

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

Righini, G. and Salani, M. (2008). New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Networks*, 51(3):155–170.

Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2007). An improved solution algorithm for the constrained shortest path problem. *Transportation Research Part B: Methodological*, 41(7):756–771.

Savnik, I. (2012). Efficient subset and superset queries. In Čaplinskas, A., Dzemyda, G., Lupeikienė, A., and Vasilecas, O., editors, *Databases and Information Systems, Tenth International Baltic Conference on Databases and Information Systems*, volume 924 of *CEUR Workshop Proceedings*, pages 45–57.

Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265.