# Indexing Discrete Sets in a Label Setting Algorithm for Solving the Elementary Shortest Path Problem with Resource Constraints

Mateusz Polnik
Department of Mechanical and Aerospace Engineering
University of Strathclyde
Glasgow, United Kingdom
Email: mateusz.polnik at strath.ac.uk

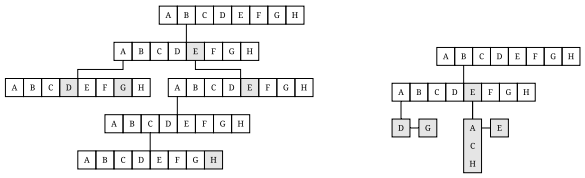
Annalisa Riccardi
Department of Mechanical and Aerospace Engineering
University of Strathclyde
Glasgow, United Kingdom
Email: annalisa.riccardi at strath.ac.uk

***Abstract*—Stopping exploration of the search space regions that can be proven to contain only inferior solutions is an important acceleration technique in optimization algorithms. This study is focused on the utility of trie-based data structures for indexing discrete sets that allow to detect such a state faster. An empirical evaluation is performed in the context of index operations executed by a label setting algorithm for solving the Elementary Shortest Path Problem with Resource Constraints. Numerical simulations are run to compare a trie with a HAT-trie, a variant of a trie, which is considered as the fastest in-memory data structure for storing text in sorted order, further optimized for efficient use of cache in modern processors. Results indicate that a HAT-trie is better suited for indexing sparse multi dimensional data, such as sets with high cardinality, offering superior performance at a lower memory footprint. Therefore, HAT-tries remain practical when tries reach their scalability limits due to an expensive memory allocation pattern. Authors leave a final note on comparing and reporting credible time benchmarks for the Elementary Shortest Path Problem with Resource Constraints.**

## I. Introduction

A trie [1], [2] is a data structure devised to efficiently store and retrieve strings built from a finite alphabet. Strings in a trie are stored as sequences of characters that correspond to nodes of a tree. Strings which have a common prefix share the initial nodes for efficient use of memory. Therefore, the total number of child nodes that a parent node may have is bounded above by the cardinality of the alphabet. For that reason the most common variant of a trie presented in textbooks stores child nodes using either lists [3] or arrays [4]. The latter data structure offers better performance in practice and simpler implementation of the $Contains$, $Insert$ and $Delete$ operations. Although, the performance advantage is at the expense of extra memory for storing unused symbols.

Tries are well known to be memory expensive due to allocation of space for storing a link to a child node for each symbol of the alphabet in every node of a trie [5]. The number of trie nodes can be reduced by compacting a path that leads to a leaf node or a branching node. This idea is neatly conceptualized in a Burst-Trie [6]. This variant of a trie has a second type of nodes that act as buckets. They have a configurable fill factor and store elements instead of pushing them down the trie thus reducing both the depth of the tree and the number of allocated nodes. After enough elements are aggregated in a bucket it is expanded to a standard trie node and its elements are dispersed down into relevant branches.




(a) Trie (b) HAT-trie

Fig. 1. Examples of trie-based data structures storing English words: bad, bag, be, beach and bee. The HAT-trie nodes are expanded into a trie node if the number of words stored in a subtree is greater than 2.

A Burst-Trie can be further tuned to exploit memory locality by using cache conscious data structures. Such a Burst-Trie is known as a HAT-trie [5]. Figure 1 illustrates structural differences between a trie and a HAT-trie. Both data structures store the the same set of strings, but a HAT-trie requires much less memory. Furthermore, a test if a word belongs to the set using a HAT-trie requires on average traversing fewer links between nodes, which reduces access time observable in practice. We refer the reader to the paper [5] for a comprehensive overview of available data structures that a HAT-trie could be built from and performance benchmarks of different bursting strategies.

Apart from typical operations on a set that can be implemented using a trie, its internal structure allows to perform efficient subset and superset queries [7]. That enabled [8] to propose a novel application of a trie as the indexing structure for labels in a label setting algorithm for solving the Elementary Shortest Path Problem with Resource Constraints (ESPPRC). The research concludes that performance benefits offered by tries are unquestionable if a problem to solve is difficult enough, for example its number of states to consider exceeds $10^6$. The overall speed up reported ranged from 4 to 20 times with respect to using lists for a label

978-1-5090-6017-7/18/$31.00 ©2018 IEEE

storage. Although, for the problems that can be solved easily an extra effort spent on building and maintaining the index compensates the obtained acceleration.

In this study we build on the work of [8] and investigate improvements by indexing labels using a HAT-trie. Our preliminary empirical analysis of a label setting algorithm, which we will present later in this paper, indicates that between 47% and 72% CPU cycles spent on computation are executing code responsible for pruning dominated labels when a trie is used for indexing. This observation motivated our further research efforts. Empirical results discussed in the paper indicate that HAT-tries are better suited for indexing labels and their memory allocation pattern is more resilient to scalability limits of tries, which may faster exhaust available virtual memory of a computing machine. Finally, we leave some critical remarks on reporting trustworthy time benchmarks for solving resource constrained shortest path problems.

The paper is organized as follows. The next section covers literature review of methods for solving shortest path problems based on dynamic programming. The optimization problem is formally stated in section III and computation results are discussed in the following section. Some conclusions are drawn in section VI.

## II. Literature Review

The Elementary Shortest Path Problem with Resource Constraints (ESPPRC) is a combinatorial optimization problem where given a graph the goal is to find a set of least cost paths that satisfy constraints expressed as resource consumption. Historically a popular variant of the problem was the Shortest Path Problem with Resource Constraints (SPPRC) which relaxes the requirement of vertices to be visited no more than once. ESPPRC appears as the sub-problem in the Column Generation method, a practical computational scheme for solving vehicle routing and crew scheduling problems [9].

The Column Generation (CG) method is designed to solve optimization problems whose large number of variables preclude an application of other methods due to memory considerations. The groundwork of CG is a decomposition of the initial problem formulation into at least two manageable problems, small enough to be solved: a restricted master problem (MP) and one or more sub-problems (SP). The method starts by solving MP using a subset of variables available in the reformulation. Information obtained from the MP solution is then transformed to SP by means of dual variables. In the next step SP is solved to find a batch of variables which have not been considered while solving MP. Dual values are unlikely to remain the same after the batch of variables is added to MP. Therefore, it is re-optimized hoping to find a better solution and update dual variables. The process is repeated in a loop until it is possible to generate new variables in SP. For a comprehensive introduction to CG, its theoretical foundation and guidelines for practical applications we refer the reader to [10], [11].

Column Generation for solving the Vehicle Routing Problem with Time Windows was introduced in [12]. The authors used the Shortest Path Problem with Time Windows (SPPTW) as the sub-problem and allowed paths to contain cycles longer than 2. A practical application in an important logistic problem attracted focus of the research community and motivated subsequent efforts to devising efficient methods for solving the shortest path problems.

Even thought SPPTW was proved to be strongly NP-hard [13] its combinatorial structure allows for an efficient exploration in a search for an exact solution by Dynamic Programming (DP). The most notable methods for solving the shortest path problems within that framework are label setting [14], [15] and label correcting [12], [16], [17], [18] algorithms. Both groups of algorithms use labels to encode partial paths and keep track of the search progress in a certain direction. During execution a label processing algorithm stores multiple labels at a time. The decision which one should be should be processed first is pivotal for the overall performance of the algorithm [19].

Label setting algorithms do not have restrictions on labels that can be processed. On the other hand, label correcting algorithms must process all labels that belong to a certain vertex in a batch, before moving to the next one. Due to this behaviour label correcting algorithms can be applied to solve shortest path problems in graphs that contain cycles and negative arc lengths. For more information on the theory and applications of label setting and label correcting algorithms we refer the reader to [20].

Necessity of tracking multiple labels imposes restrictions on the size of the shortest path problems that can be solved in practice. To partially alleviate this issue, dominance rules can be defined. Their aim is to detect labels which are certain to deliver inferior paths either with respect to their cost or due to higher resource consumption. Therefore, dominated labels can be safely discarded without affecting the quality of the final solution. The importance of having efficient dominance rules is emphasized in [16], where authors studied reduction in the duality gap obtained by solving elementary shortest path problems that provide a stronger lower bound than non-elementary ones.

A label definition and dominance rules for the Capacitated Arc-Routing Problem that allow partial enforcement of the path elementary constraint were proposed in [21]. The article demonstrates how to encode the structural constraints that prohibits cycles of a given length. Authors, equipped with such a tool, investigate the tradeoff between the strength of a dominance rule, the quality of a lower bound and computation time. Examples considered in their study provide evidence for two intuitive phenomena that can be observed while solving optimization problems. Firstly, the stronger the dominance rule is the more time is required for its validation. Secondly, weak dominance rules may be good enough to solve problems that are not overly constrained. Such rules can be executed quickly and for simple problems they do not negatively impact the quality of the lower bound. On the other hand, problems which are difficult to solve, seem to require stronger dominance rules.

Typical DP algorithms for solving the shortest path prob-

lems conduct a search by extending labels in one direction, from the source vertex to the sink vertex. Performance of such algorithms tends to deteriorate as partial paths are getting longer, because it makes finding a valid extension for a given label more difficult. The idea of Bidirectional Dynamic Programming (BDP) in the context of ESPPRC was studied by [17]. In BDP the search for the shortest paths starts simultaneously in two directions: forward from the source vertex and backwards from the sink vertex. Complete paths are then created by joining partial paths from both ends.

The concept of using DP to perform a search in multiple directions was extended in the recent work [22]. The authors devised an iterative algorithm tailored for CG, where it is enough to find columns with sufficiently small reduced cost. The algorithm is designed to be run consecutively to solve SPPRC in the same graph using different dual information. It remembers efficient label extensions from previous iterations and uses them to select labels that will be extended first. This approach successfully delays treatment of labels that are unlikely to lead to efficient paths. The algorithm may also output a complete path faster reusing a sequence of extensions from a previous iteration. Combination of both techniques allows to reduce the search of the near optimal paths to a fraction of time that a standard DP needs.

Alternative algorithms that do not bear the burden of handling labels directly have been studied recently and were proved to offer comparable performance in practice [23]. Although the article presents a promising approach for encoding state and its propagation, the empirical performance evaluation has not yet been backed up by complexity analysis and the proof of correctness of its pruning strategies.

Overall, literature on SPPRC provides a vast array of acceleration techniques. The benefits of dominance rules [16], [21] and the importance of choosing the right search direction [17], [22] have been meticulously reported. On the other hand, to the best of our knowledge except for the work [8] little research have been done on data structures that allow for an efficient storage of labels and execution of dominance rules, the gap we aim to partially fill. Our research results should be compatible with any aforementioned improvements and therefore used in conjunction to develop even faster solvers for SPPRC.

## III. Problem Statement

Let $G(V, E)$ be a directed graph with $V$ vertices and $E$ edges. Each vertex $v \in V$ and edge $e \in E$ has a vector of resources. Elements of the vector are either scalar values $r_i$ or pairs $(r_i^{begin}, r_i^{end})$. A scalar value $r_i$ encodes consumption of a resource $i$ when a vertex $v$ is visited or an edge $e$ is traversed. A pair $(r_i^{begin}, r_i^{end})$ encodes a constraint on the lower and upper bound of a resource $i$ for a vertex $v$ to be available for visiting or an edge $e$ to be admissible. Furthermore, an edge $e$ has a cost of traversal which may be negative.

The subject of ESPPRC is to find a set of paths which visit a sequence of vertices with minimal cost in such a way that along each path no vertex is visited more than once and no resource constraint is violated. All paths begin and end in a designated vertex known as the depot. For notational convenience the depot is split into two vertices referred to as the source and the sink. The term node will not be used interchangeably with a vertex to prevent the naming collision with a node that is a building block of the trie data structure.

Without the loss of generality we restrict our attention to the problem with two resources: capacity and time. For syntactic convenience resource values will be accessed via unary operators. For example, a vertex $v$ has capacity $demand(v)$, time required to be spent on servicing $time(v)$ and a time window that denotes the earliest and the latest time when servicing may begin, $time^{begin}(v)$ and $time^{end}(v)$ respectively. An edge $e$ has time $time(e)$ and the cost of traversal $cost(e)$, which is added for simplicity to the resource vector. A path may arrive at a vertex before its time window opens, but servicing cannot start earlier. There is no penalty for waiting.

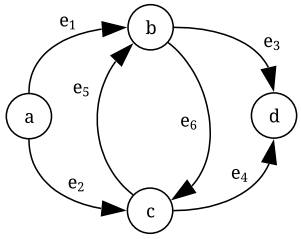
### A. Label Definition

A label encodes a partial path from the source to a given vertex, cumulative cost and resource consumption. Label processing algorithms use them to track progress of a search in a specific direction and to restore a solution after no more labels to process remain. In next paragraphs we provide the definition of a label adopted in this paper and operations on labels. The label definition and the dominance rule is due to [16].

A label is defined by the vector $[c,\ t,\ d,\ v,\ F]$ and a link to the parent label. First three scalar values are the first letters of the following operators: $c$ is the cost of the partial path calculated as the aggregate sum of the traversed edges' cost. $t$ is the total time spent on waiting, servicing vertices and traversing edges. $d$ is the aggregate sum of capacity demands of vertices visited by the partial path. $v$ is a number assigned to the vertex where the partial path ends. The set $F$ contains the vertices that cannot be visited by an extension of the partial path, because they have been visited already or visiting them would violate some resource constraints.

New labels are created by extension of existing ones. A label $l^{'}(c, t, d, v = i, F : v_i \in F \wedge v_j \notin F)$ extended along the edge $e = (v_i, v_j)$ creates a new label $l^{''}(c + cost(e), max(t + time(e), time^{begin}(v_j)) + time(v_j), d + demand(v_j), v = j, F : v_i \in F \wedge v_j \in F)$. Furthermore, the set $F$ contains all other vertices that became unreachable due to the extension. If a path arrives at a vertex $v$ before its time window is opened, extra waiting time is incurred. Otherwise, the service may start immediately after the arrival. This logic can be expressed in a compact form as $max(t+time(e), time^{begin}(v_j))+time(v_j)$ where $time(e)$, $time^{begin}(v_j)$ and $time(v_j)$ denote respectively travel time via an edge $e$, the earliest time when a service may begin and the time of servicing.

Finally, a relation of dominance can be defined between labels. A label $l_a$ dominates a label $l_b$ if the following chain of inequalities holds $c^{l_a} \leq c^{l_b} \wedge t^{l_a} \leq t^{l_b} \wedge d^{l_a} \leq d^{l_b} \wedge v^{l_a} = v^{l_b} \wedge F_b \subseteq F_a$ and at least one of the weak inequalities is strict. Intuitively it means that if no path obtained by an extension of the label $l_b$ is shorter than any path obtained by an extension of the label $l_a$ then the label $l_a$ dominates the label $l_b$. The




| Vertex | Service Time | Demand |
|---|---|---|
| a | 0 | 0 |
| b | 2 | 4 |
| c | 1 | 1 |
| d | 2 | 2 |

| Edge | Cost | Travel Time |
|---|---|---|
| $e_1$ | 1 | 1 |
| $e_2$ | 2 | 2 |
| $e_3$ | 1 | 1 |
| $e_4$ | 1 | 1 |
| $e_5$ | 1 | 1 |
| $e_6$ | 2 | 2 |

| Step | Label | Cost | Time | Demand | Vertex | Visited Vertices |
|---|---|---|---|---|---|---|
| 1 | $L_1$ | 0 | 0 | 0 | $a$ | $\{a\}$ |
| 2 | $L_1 \xrightarrow{e_1} L_2$ | $0+1$ | $0+1+2$ | $0+4$ | $b$ | $\{a\} \cup \{b\}$ |
| 3 | $L_2 \xrightarrow{e_6} L_3$ | $1+1$ | $3+1+1$ | $4+1$ | $c$ | $\{a,b\} \cup \{c\}$ |
| 4 | $L_3 \xrightarrow{e_4} L_4$ | $2+1$ | $5+1+2$ | $5+2$ | $\boxed{d}$ | $\{a,b,c\} \cup \{d\}$ |
| 5 | $L_2 \xrightarrow{e_3} L_5$ | $1+2$ | $3+2+2$ | $4+2$ | $\boxed{d}$ | $\{a,b\} \cup \{d\}$ |
| 6 | $L_1 \xrightarrow{e_2} L_6$ | $0+2$ | $0+2+1$ | $0+1$ | $c$ | $\{a\} \cup \{c\}$ |
| 7 | $L_6 \xrightarrow{e_5} L_7$ | $2+1$ | $3+1+2$ | $1+4$ | $b$ | $\{a,c\} \cup \{b\}$ |
| 8 | $L_7 \xrightarrow{e_3} L_8$ | $3+2$ | $6+2+2$ | $5+2$ | $\boxed{d}$ | $\{a,b,c\} \cup \{d\}$ |
| 9 | $L_6 \xrightarrow{e_4} L_9$ | $2+1$ | $3+1+2$ | $1+2$ | $\boxed{d}$ | $\{a,c\} \cup \{d\}$ |

Fig. 2. Steps of a label processing algorithm for finding the shortest paths with resource constraints.

correctness of the relation of dominance defined above was proven in [16].

Concepts introduced in this section are depicted in Figure 2. It presents a graph with a source depot $a$ and vertices: $b$, $c$ and a sink depot $d$. Customers have predefined capacity demands and service times. Time windows are ignored for simplicity. Edges are drawn for admissible transfer links. Each edge has cost and travel time defined.

Assume that a label setting algorithm which expands labels in a depth first search fashion is run on the graph. Following the execution steps in Figure 2, such an algorithm ran to completion creates nine labels. Four of them are assigned to the vertex $d$: $L_4$, $L_5$, $L_8$ and $L_9$. The label $L_8$ can be discarded, because having covered the same vertices as the label $L_4$ has a higher cost and requires more time.

### B. Algorithm

The ESPPRC can be solved by the label setting algorithm presented in Figure 3. The control flow resembles other label setting algorithms from literature [15]. There is no difference between a trie and a HAT-trie at this level of abstraction, thus the data structure used for indexing is being referred to simply as a trie. We start with a brief outline of the main steps of the algorithm and then focus on the operations that involve the indexing structures.

For simplicity of the exposition we split the algorithm into three stages: initialization, label processing and restoring the paths. Transitions between them are marked by comments in Figure 3.

At the beginning we create initial labels and data structures to store them. Each vertex has its own trie to index labels corresponding to partial paths terminated at the vertex. Furthermore, regardless of the final vertex, labels that are waiting to be processed are also stored in the min-priority queue which arranges them in the ascending order by the cost.

Then labels are removed from the queue sequentially and extended in all possible directions. An extension of a label creates a new label. The following invariants ensure that the process terminates. Firstly, labels that correspond to non-elementary paths are discarded immediately. Secondly, tries store only non-dominated labels. Finally, a label can be enqueued only if it is not dominated by a label encountered before. Therefore, the queue stores only labels which have not yet been processed.

After no more labels are left for processing the algorithm restores the shortest paths from labels that are indexed by the trie associated with the sink vertex.

The aforementioned algorithm uses a trie to store non-dominated labels and execute tests for dominance. An insertion to a trie is performed by the operation introduced in Figure 4. The insertion is delegated to the *trie_insert* function that depends on the trie data structure. Possible implementations of the function are explained in [1] for a trie and for a HAT-trie [5]. From control flow of the algorithm in Figure 3 it is clear that the insertion is executed only if no dominating labels were found and the label to be inserted is not stored in the trie. Thus extra integrity checks, such as a test for dominance, can be skipped. Finally, after a successful insertion the trie is cleaned up from labels that are dominated by the newly inserted one. Pruning dominated labels from a trie and testing for dominance are the subjects of the following section.

## IV. HAT-trie Operations

Both pruning of dominated labels and testing for dominance require a trie traversal, therefore they depend on the data structure definition. To make the exposition succinct we focus on HAT-tries, because a trie is a special case of the HAT-trie with all nodes expanded and relevant operations for a trie follow immediately.

Figures 5 and 6 presented in this section will use the following operations to handle trie nodes.

$is\_expanded(node)$ test if child nodes are stored in an array, otherwise they are stored in a list,

$has\_child(node, offset)$ test if a child node exists for the given offset,

$child(node, offset)$ access a child node for the given offset,

$children(node)$ enumerate child nodes stored by the node,

$erase(child, offset)$ remove a child node for the given offset,

```
procedure solve_spprc(graph, source, sink)
queue ← priority_queue() {▷ initialization}
tries ← vector(trie(), num_vertices(graph))
label_source ← label(0, 0, 0)
insert(tries[source], label_source)
add(queue, label_source)
for all vertex ∈ vertices(graph) do
  if vertex ≠ source and vertex ≠ sink then
    edge ← edge(graph, source, vertex)
    label ← extend(label_source, edge)
    insert(tries[vertex], label)
    add(queue, label)
  end if
end for
while not empty(queue) do
  label ← pop(queue) {▷ label processing}
  trie ← tries[label]
  if is_dominated(trie, label) then
    continue
  end if
  for all edge ∈ out_edges(graph, label) do
    if is_visited(label, end(edge)) then
      continue
    end if
    label_next ← extend(label, edge)
    trie_next ← tries[label_next]
    if is_dominated_or_equal(trie_next, label_next) then
      continue
    end if
    insert(trie_next, label_next)
    add(queue, label_next)
  end for
end while
return iterate(tries[sink]) {▷ restoring the paths}
```

Fig. 3. A label setting algorithm for solving ESPPRC.

```
procedure insert(node, label)
inserted ← trie_insert(node, label)
if inserted then
  prune(node, label)
end if
return inserted
```

Fig. 4. A generic operation to insert a label into a trie-based index.

$is\_empty(node)$ test if the node leads to some labels,
$labels(node)$ enumerate labels stored in the node,
$vertex(node)$ access a vertex associated with the node,
$next(iterator|offset)$ increment an offset or an iterator.

Figure 5 presents pseudocode of the operation that prunes dominated labels. The clue in understanding its control flow is the observation that a label may be dominated only if its partial path visits vertices that are also visited by the dominating label. Thus, pruning of dominated labels reduces to enumerating labels whose partial paths visit subsets of vertices

```
procedure prune(node, node_begin, node_end, label)
if is_expanded(node) then
  offset ← 0
  for it ← node_begin; it ≠ node_end; it ← next(it) do
    if it = 0 then
      if has_child(node, offset)
        and prune(child(node, offset), next(it),
      node_end, label) then
        erase(child(node, offset))
      end if
      offset ← next(offset)
      continue
    end if
    if has_child( node, offset )
      and prune( child(node, offset), next(it),
    node_end, label) then
      erase(child(node, offset))
    end if
    break
  end for
  for all label_old ∈ labels(node) do
    if is_dominated(label_old, label) then
      erase(label_old)
    end if
  end for
else
  for all label_old ∈ labels(node) do
    if is_dominated(label_old, label) then
      erase(label_old)
    end if
  end for
end if
return is_empty(node)
```

Fig. 5. A procedure for pruning of dominated labels in a HAT-trie.

which belong to the partial path of the dominating label. That operation can be implemented as a recursive procedure that traverses a trie from the top to the bottom following branches that index subsets of vertices. The procedure takes as the input a HAT-trie node, a pair of iterators to a binary vector whose elements indicate whether a vertex has been visited by the inserted label and the label itself. The procedure returns true if the node become empty after the pruning or false otherwise. The control flow of the operation depends on the type of a node that is being processed. In case of an expanded node, which is the only type of nodes in a trie, the procedure recursively crawls down to child nodes that index bigger subsets. If due to the pruning a node becomes empty its memory is released. Having traversed descendant nodes content of a parent node is processed and its dominated labels are released. On the other hand, if a node is not expanded all labels it contains are checked for dominance and erased if possible.

Figures 3 and 5 use different variants of the test for dominance, which can be either strong or weak. They are

```
procedure find(node, predicate, path)
if is_expanded(node) then
  for all label ∈ labels(node) do
    if predicate(label) then
      return true
    end if
  end for
  for all child ∈ children(node) do
    if vertex(child) ∈ path
      and find(child, predicate, path) then
      return true
    end if
  end for
else
  for all label ∈ labels(node) do
    if is_subset_or_equal(label, path)
      and predicate(label) then
      return true
    end if
  end for
end if
return false
```

Fig. 6. A template method for finding a label in a HAT-trie using a predicate.

denoted in the enclosed figures respectively as $is\_dominated$ and $is\_dominated\_or\_equal$. Both share the same logic of traversing a trie and handling various node types. To avoid repetition Figure 6 presents pseudocode of a template method that is independent of the dominance rule definition. The appropriate dominance rule is passed as a predicate. The method performs a pre-order traversal of a tree and returns true if the predicate was satisfied for any label. Otherwise, it returns false.

## V. Computation Results

### A. Experiment Design

Numerical experiments were conducted to evaluate the performance of trie and HAT-trie indexing structures used in the aforementioned label setting algorithm. Test instances were obtained from the Solomon benchmark problems [24]. The study is limited to the first series of problems with customers whose locations are either distributed randomly or grouped into clusters. The experiments were performed on instances counting 50 and 100 customers.

Solomon problems were represented as graphs that store vertex neighbors in adjacency lists. Before running experiments we removed unreachable edges from the graph and tightened time windows using the method from [12]. Then to simulate negative edge costs we followed steps explained in [16]. The cost of each edge was lowered by subtracting a random value drawn with the uniform distribution from the set $\{0, ..., 20\}$. Finally, the adjacency lists were sorted by the edge cost. The purpose of lowering the edge cost which may turn negative is to imitate the conditions characteristic to the application of ESPPRC as the sub-problem in CG. In practice values to subtract from the edge cost are the dual multipliers obtained from the MP solution.

For each instance of the benchmark problem we generated 32 graph samples and used them as the input to the algorithm. Except for the edge cost there was no other difference in graphs obtained for the same benchmark problem. The number of samples selected for the study was influenced by available computing resources. The average time required to solve a difficult instance exceeds 15 minutes, which yields at least 8 hours to evaluate all samples for a single configuration of parameters. In authors view it is very unlikely that a higher number of samples might affect the statistics of the results discussed.

Simulations were developed as a single threaded program. Trie and HAT-trie data structures were implemented by us. We used the HAT-trie variant with arrays. Therefore, we applied orders of magnitude smaller burst thresholds than the values suggested in [5] for hash arrays. The reason for using arrays as opposed to hash arrays was the need to support subset and superset queries.

The program was compiled using the GCC 6.3 compiler with the optimization flags: -O2 and -march=native. Simulations were run on a workstation with the Intel Core i7-4790 CPU and 8 GB of RAM. Dynamic CPU frequency scaling was disabled. Time of the algorithm execution was measured using the Google Benchmark library [25]. If the program exceeded maximum amount of virtual memory available on the machine the process was terminated. The same happened if the total computation time of all graph samples of a single benchmark instance exceeded 48 hours.

### B. Analysis

Figure 7 displays the time required to find all the resource constrained shortest paths. Their exact number varies between benchmark instances and depends on the final edges' costs after dual multipliers were subtracted. This subject will be further discussed later. The time is measured in milliseconds. Its mean value is aggregated over graph samples generated for the particular benchmark problem. Results were split into four charts according to a class of the problem for customers located at random or distributed between clustered and then by their count. The remainder of this section is devoted to discussion of several phenomena that appear in the charts.

The label setting algorithm that used the HAT-trie index achieved superior empirical performance except for a single instance when it was insignificantly slower. Furthermore, the HAT-trie data structure by design requires less memory per node than a trie, allows for a path compression and reduction of the total number of nodes. Our results are therefore an indicator of feasibility of using HAT-trie indices for highly dimensional structures such as sets of cardinality exceeding the number of symbols in the Latin alphabet, for which tries were excessively benchmarked.

Solving some benchmark instances is significantly harder than others. The issue has a plausible explanation. The more

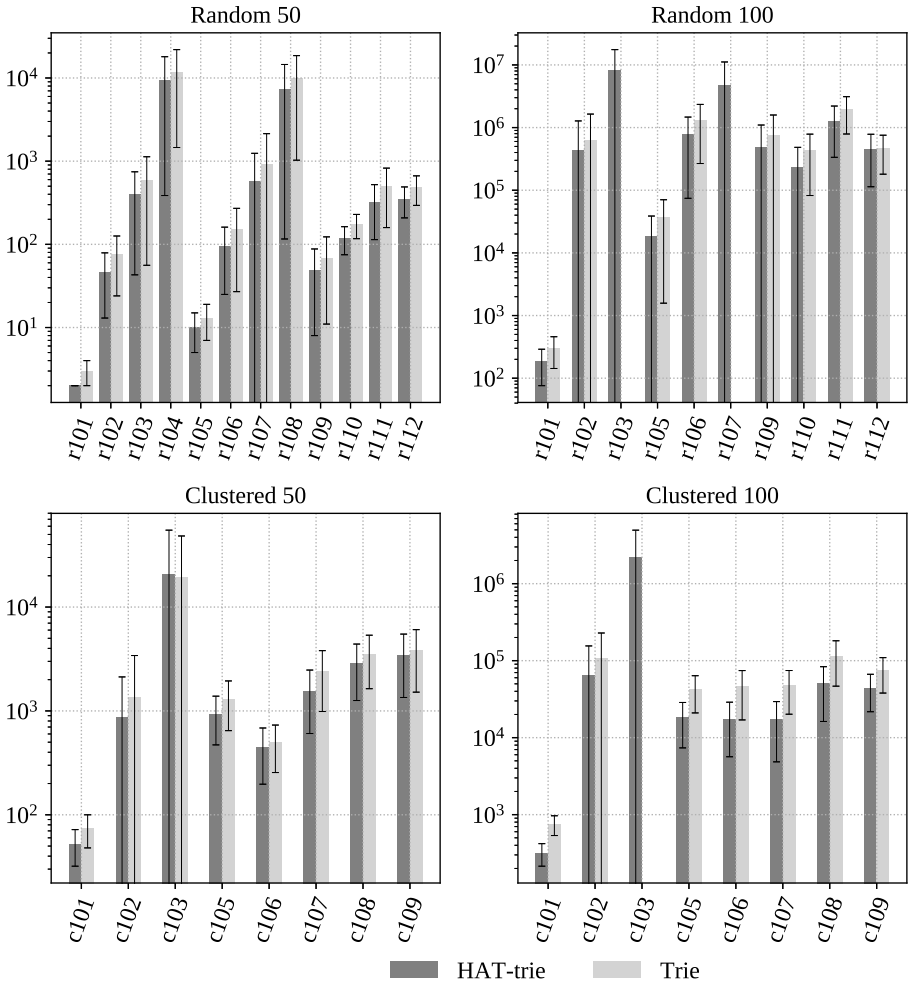



Fig. 7. Average time [ms] required for solving ESPPRC instances in the Solomon benchmark.

shortest paths a problem has the longer does it take to find all of them. Solving such a problem requires processing more labels and longer tests for dominance. Existing research also indicates that percentage of negative edges is an indicator of the problem difficulty [16], [15].

Some instances were not solved within the imposed resource limits. The label setting algorithm with the HAT-trie index failed to solve in the alloted time the problems: c104 50, c104 100 and r104 100, r108 100. Besides that, the algorithm which used a trie was terminated on the problems: c103 100, r103 100 and r107 100. In all cases of a trie the reason for termination was exceeding the memory limit. We find it improbable that a label setting algorithm without considerable improvements might solve 32 instances of such a problem in practical time on a modern workstation. It should be noted that single instances of these problems were solved by either an enhanced label correcting algorithm [26] or by the decremental state-space relaxation [18].

Standard deviation of computation time for certain problems is very high. Similar effect could be observed for other benchmark problems or may appear after the size of an instance is increased. To study this phenomenon we plotted the number of the shortest paths found in each sample of the problem c103 50 and the time required for its solution. The chart is presented in Figure 8. It shows that having network structure and resource consumption defined one can modify the cost of edges in such a way that the search for the shortest paths will finish within 1 second. However, it is also not difficult to find a sequence of edge cost reductions that makes the computation longer than 1 minute. In our view this phenomena should be taken into account while revising prior published benchmark results on SPPRC which do not provide the number of instances solved or another measure of confidence in the results presented. The notable exception is [15] where authors highlight similar issue observed while using a problem generator.

The results discussed above were obtained for the HAT-trie with burst threshold of 4. We also investigated how the value of this parameter affects the algorithm performance. 32 samples of the problem c103 50 were solved for the sequence of burst thresholds $[1, ..., 16]$. The same performance was observed for

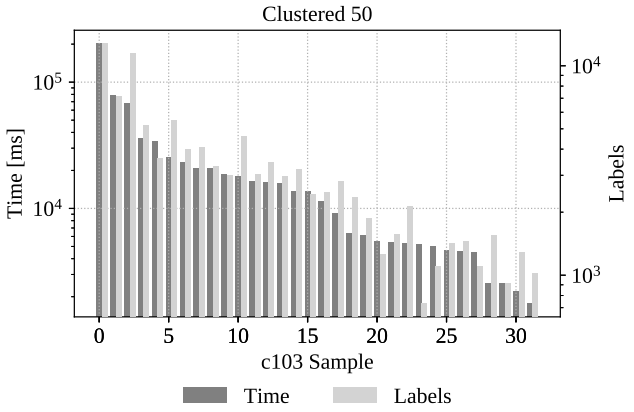



Fig. 8. Time [ms] required for computations and the number of the shortest paths with resource constrained found in samples of the benchmark instance c103 of size 50.

values ranging from 2 to 16. Understandably the performance was inferior for the burst threshold of 1 when a HAT-trie resembles a trie. Overall, this result suggests that performance advantages observed in Figure 7 were attributed to fewer branches in the tree rather than better utilization of memory locality. However, a definitive proof requires more research.

## VI. Conclusion

A HAT-trie in practical setting outperforms a trie offering superior time complexity to execute a sequence of insert, delete and look-up operations in a label setting algorithm. In addition a HAT-trie uses a more conservative memory allocation strategy which on average results in less memory allocated per node and fewer nodes in total. Therefore, results discussed in the paper could serve as an indicator of HAT-trie feasibility for indexing highly dimensional structures such as sets of cardinality exceeding the number of symbols in the Latin alphabet, which tries were benchmarked for. Meanwhile, a trie offered inferior performance and faster reached its scalability limits. Due to the excessive memory allocation pattern of a trie the algorithm that used this data structure to index labels ran out of the total available virtual memory on a computing node and it became impractical to continue simulations.

Finally, running times of algorithms for solving SPPRC observed for randomly disturbed edge cost, which is the established technique for obtaining sample problems, may significantly vary. Therefore, providing a priori the number of vertices, the network structure and the maximum value subtracted from an edge cost are not enough to reason about the computing effort required to solve the problem.

## References


[1] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[2] R. De La Briandais, "File searching using variable length keys," in *Western Joint Computer Conference*, Mar. 1959, pp. 295–298.

[3] D. E. Knuth, *The Art of Computer Programming, Volume 3: (Second Edition) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[4] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley Professional, 2011.

[5] N. Askitis and R. Sinha, "Hat-trie: A cache-conscious trie-based data structure for strings," in *Proceedings of the thirtieth Australasian conference on Computer Science*, vol. 62, 2007, pp. 97–105.

[6] S. Heinz, J. Zobel, and H. E. Williams, "Burst tries: A fast, efficient data structure for string keys," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 2, pp. 192–223, 2002.

[7] I. Savnik, "Index data structure for fast subset and superset queries," in *International Conference on Availability, Reliability, and Security*, Regensburg, Germany, Sep. 2013, pp. 134–148.

[8] T. M. Range, "Exploiting set-based structures to accelerate dynamic programming algorithms for the elementary shortest path problem with resource constraints," Department of Business and Economics, University of Southern Denmark, Tech. Rep. 17/2013, 2013.

[9] S. Irnich and G. Desaulniers, "Shortest path problems with resource constraints," in *Column Generation*, G. Desaulniers, J. Desrosiers, and M. Solomon, Eds., 2005, ch. 2, pp. 33–65.

[10] G. Desaulniers, J. Desrosiers, and M. Solomon, *Column Generation*. Springer US, 2006, vol. 5.

[11] M. Lübbecke and J. Desrosiers, "Selected topics in column generation," *Operations Research*, vol. 53, no. 6, pp. 1007–1023, 2005.

[12] M. Desrochers, J. Desrosiers, and M. Solomon, "A new optimization algorithm for the vehicle routing problem with time windows," *Operations Research*, vol. 40, no. 2, pp. 342–354, 1992.

[13] M. Dror, "Note on the complexity of the shortest path models for column generation in vrptw," *Operations Research*, vol. 42, no. 5, pp. 977–978, 1994.

[14] I. Dumitrescu and N. Boland, "Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem," *Networks*, vol. 42, no. 3, pp. 135–153, 2003.

[15] N. Boland, J. Dethridge, and I. Dumitrescu, "Accelerated label setting algorithms for the elementary resource constrained shortest path problem," *Operations Research Letters*, vol. 34, no. 1, pp. 58–68, 2006.

[16] D. Feillet, P. Dejax, M. Gendreau, and C. Gueguen, "An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems," *Networks*, vol. 44, no. 3, pp. 216–229, 2004.

[17] G. Righini and M. Salani, "Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints," *Discrete Optimization*, vol. 3, no. 3, pp. 255–273, 2006.

[18] ——, "New dynamic programming algorithms for the resource constrained elementary shortest path problem," *Networks*, vol. 51, no. 3, pp. 155–170, 2008.

[19] F. Guerriero and L. Di Puglia Pugliese, "Multi-dimensional labelling approaches to solve the linear fractional elementary shortest path problem with time windows," *Optimization Methods & Software*, vol. 26, no. 2, pp. 295–340, 2011.

[20] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.

[21] C. Bode and S. Irnich, "The shortest-path problem with resource constraints with (k, 2)-loop elimination and its application to the capacitated arc-routing problem," *European Journal of Operational Research*, vol. 238, no. 2, pp. 415–426, 2014.

[22] F. S. Ilyas Himmich, Issmail El Hallaoui, "A multidirectional dynamic programming for the shortest path problem with resource constraints," HEC Montreal, Montreal, Canada, Tech. Rep., Feb. 2018, visited on the 6th of April 2018. [Online]. Available: https://www.gerad.ca/en/papers/G-2018-05

[23] L. Lozano, D. Duque, and A. L. Medaglia, "An exact algorithm for the elementary shortest path problem with resource constraints," *Transportation Science*, vol. 50, no. 1, pp. 348–357, 2015.

[24] M. Solomon, "Algorithms for the vehicle routing and scheduling problems with time window constraints," *Operations research*, vol. 35, no. 2, pp. 254–265, 1987.

[25] D. Hamon, *A Microbenchmark Support Library*, Google, 2017. [Online]. Available: https://github.com/google/benchmark

[26] D. Feillet, M. Gendreau, and L.-M. Rousseau, "New refinements for the solution of vehicle routing problems with branch and price," *Information Systems and Operational Research*, vol. 45, no. 4, pp. 239–256, 2007.