Journal of Artificial Intelligence Research 54 (2015) 593-629 Submitted 07/15; published 12/15

Compressing Optimal Paths with Run Length Encoding

Ben Strasser STRASSER@KIT.EDU
Karlsruhe Institute of Technology
Karlsruhe, Germany

Adi Botea ADIBOTEA@IE.IBM.COM
IBM Research
Dublin, Ireland

Daniel Harabor DANIEL.HARABOR@NICTA.COM.AU
NICTA
Sydney, Australia

Abstract

We introduce a novel approach to Compressed Path Databases, space efficient oracles used to very quickly identify the first edge on a shortest path. Our algorithm achieves query running times on the 100 nanosecond scale, being significantly faster than state-of-the-art first-move oracles from the literature. Space consumption is competitive, due to a compression approach that rearranges rows and columns in a first-move matrix and then performs run length encoding (RLE) on the contents of the matrix. One variant of our implemented system was, by a convincing margin, the fastest entry in the 2014 Grid-Based Path Planning Competition.

We give a first tractability analysis for the compression scheme used by our algorithm. We study the complexity of computing a database of minimum size for general directed and undirected graphs. We find that in both cases the problem is NP-complete. We also show that, for graphs which can be decomposed along articulation points, the problem can be decomposed into independent parts, with a corresponding reduction in its level of difficulty. In particular, this leads to simple and tractable algorithms with linear running time which yield optimal compression results for trees.

1. Introduction

A Compressed Path Database (CPD) is an index-based data-structure for graphs that is used to very quickly answer first-move queries. Such a query takes as input a pair of nodes, namely a source node s and a target node t, and asks for the first edge on a shortest st-path (i.e., a path from s to t). CPDs have successfully been applied in a number of contexts important to AI. For instance, Copa (Botea, 2012), a CPD-based pathfinding algorithm, was one of the joint winners in the 2012 edition of the Grid-Based Path Planning Competition, or shorter GPPC (Sturtevant, 2012b). A related algorithm, MtsCopa, is a fast method for moving target search over known and partially known terrain (Botea, Baier, Harabor, & Hernández, 2013; Baier, Botea, Harabor, & Hernández, 2014).

Given a graph $G = (V, E)$, a trivial CPD consists of a square matrix $\mathbf{m}$ with dimensions $|V| \times |V|$. The matrix $\mathbf{m}$, constructed during a precomputation step, stores in each cell $\mathbf{m}[s, t]$ the identity of a first edge on a shortest st-path. We call this a *first-move matrix*. By convention we say that rows of $\mathbf{m}$ correspond to fixed source nodes and the columns to fixed target nodes. This is optimal in terms of query time but the $O(|V|^2)$ space consumption quickly becomes prohibitive for larger graphs. The challenge is to design a compact representation of $\mathbf{m}$ that trades a small increase in query times for a large decrease in space consumption.

A number of different techniques to compress a first-move matrix have been suggested for this purpose (Sankaranarayanan, Alborzi, & Samet, 2005; Botea, 2011; Botea & Harabor, 2013a). In each case the objective is to conserve space by grouping together entries of $\mathbf{m}$ which all share a common source node and which all store the same first-edge information.

In this work we present the Single-Row-Compression (SRC) and the Multi-Row-Compression (MRC) indexing algorithms for compressing all-pairs shortest paths. In 2014's GPPC, SRC outperformed all competitors in terms of query running time. The contributions presented in this article go in three main directions: a new approach to compressing a first-move matrix; experiments that demonstrate advancing state-of-the-art in terms of both response time and memory consumption; and a thorough theoretical analysis, discussing NP-hardness results and islands of tractability.

We introduce a new matrix compression technique based on run-length encoding (RLE). The main idea of our algorithm is simple: we compute an order for the nodes in the input graph and assign numeric IDs to nodes (e.g., from 1 to $|V|$) in this order. The purpose of the ordering is that nodes which are located in close proximity in the graph have a small ID difference. This ordering is used to order the rows and the columns of a first-move matrix, which is also computed during preprocessing. Then, we apply run-length encoding (RLE) to each row of the first-move matrix. We study three types of heuristic orderings: graph-cut order, depth-first order and input-graph order. We also study two types of run-length encoding. The first involves a straightforward application of the algorithm to each row. The second type is a more sophisticated multi-row scheme that eliminates redundancies between adjacent RLE-compressed rows. To answer first-move queries we employ a binary search on a fragment on the compressed result.

We undertake a detailed empirical analysis including comparisons of our techniques with state-of-the-art variants of CPDs (Botea, 2012), and Hub-Labeling (Delling, Goldberg, Pajor, & Werneck, 2014). Copa is a recent and very fast CPD oracle which was among the joint winners at the 2012 International Grid-Based Path Planning Competition (GPPC). Using a variety of benchmarks from the competition we show that our techniques improve on Copa, *both* in terms of storage and query time. Hub-Labeling is a technique initially developed to speedup queries on roads, but they also work on other graphs, such as gridmaps. Hub-Labeling is to the best of our knowledge the fastest technique known on roads. In experiments, we show that our approach leads to better query times than Hub-Labeling on graphs where we can reasonably compute $\mathbf{m}$.

As our technique relies on an all-pairs-shortest-path pre-computation, it plays a tradeoff between its query-response speed, the preprocessing time and the memory required to store the compressed path database. Thus, our algorithm is faster, but it also requires a larger preprocessing time and more memory than some of the other techniques from the literature. In other words, when memory and preprocessing time are available, our technique will provide a state-of-the-art speed performance. On the other hand, when larger and larger graphs create a memory and preprocessing time bottleneck, other techniques should be considered. See a detailed comparison in the experiments section.

In the theoretical analysis, we formally define and study optimal RLE-compression of first-move matrices produced from input graphs. We consider the case of directed input graphs and the case of undirected weighted input graphs. We show that both versions are NP-complete. Focusing on such distinct types of graphs, each result brings something new compared to the other. Related (Kou, 1977; Oswald & Reinelt, 2009) and weaker, less specific (Mohapatra, 2009) results on RLE-based matrix compression are available in the literature. However, as known, the NP-hardness of a class of problems does not necessarily imply the NP-hardness of a subset of the class. Thus, despite

previous related results (Mohapatra, 2009), it has been an open question whether the optimal RLE-compression of a *first-move matrix computed from an input graph* is tractable.

We also show that, for graphs which can be decomposed along articulation points, the problem can be decomposed into independent subproblems. If optimal orderings are available for the subproblems, a global optimal ordering can easily be obtained. In particular, a depth-first preorder is optimal on trees, and the general ordering problem is fixed-parameter tractable in the size of the largest 2-connected component.

Our approach and part of the evaluation have previously been reported in a shorter conference paper (Strasser, Harabor, & Botea, 2014). The theoretical analysis was the topic of another conference paper (Botea, Strasser, & Harabor, 2015). Putting these together in the current submission provides a unique source that describes our method, its performance and its theoretical properties. Compared to the previous conference papers, we now provide complete proofs for all the theoretical results. We have included more details and more examples in the presentation, for a better clarity. We report additional results, such as the performance in the pathfinding competition GPPC 2014, which were originally published in a paper about the competition (Sturtevant, Traish, Tulip, Uras, Koenig, Strasser, Botea, Harabor, & Rabin, 2015).

2. Related Work

Many techniques from the literature can be employed in order to quickly answer first-move queries. Standard examples include optimal graph search techniques such as Dijkstra's algorithm (Dijkstra, 1959) and A* (Hart, Nilsson, & Raphael, 1968). Significant improvements over these methods can be achieved by preprocessing the input graph, as done in CPDs, for instance. As shortest paths have numerous applications in various fields, a plethora of different preprocessing-based algorithms have been proposed. For an overview, we refer the interested reader to a recent survey article (Bast, Delling, Goldberg, Müller–Hannemann, Pajor, Sanders, Wagner, & Werneck, 2015). A common approach consists of adding online pruning rules to Dijkstra's algorithm, which rely on data computed in the preprocessing phase, significantly reducing the explored graph's size. As this approach significantly differs from the technique described in this paper, we omit the details and refer the interested reader to the aforementioned survey article.

SILC (Sankaranarayanan et al., 2005) and Copa (Botea & Harabor, 2013a) are CPD-based techniques for a fast first-move computation. SILC employs a recursive quad-tree mechanism for compression while Copa uses a simpler and more effective (Botea, 2011) decomposition with rectangles.

Hub Labels (HL) were initially introduced as 2-Hop Labels (Cohen, Halperin, Kaplan, & Zwick, 2002). For nearly a decade there has not been much research on the topic, until Abraham, Delling, Goldberg, and Werneck (2011) showed that the technique is practical on huge road networks, and coined the term Hub Labels. This realization drastically increased the interest in HL and thus spawned numerous follow up works, such as (Abraham, Delling, Goldberg, & Werneck, 2012; Delling, Goldberg, & Werneck, 2013; Abraham, Delling, Fiat, Goldberg, & Werneck, 2012; Akiba, Iwata, & Yoshida, 2013). In our context, the most relevant one is probably RXL (Delling et al., 2014), which is a HL variant. The authors show that their algorithm works well not only on road graphs but on a variety of graphs from different sources including graphs derived from maps used during GPPC. We compare our algorithm against RXL.

The HL index consists of a forward and backward label for each node, that contains a list of *hub nodes* and the exact distances to them. For each st-pair there must exist a *meeting hub* h that is a

forward hub of s and a backward hub of t and is on a shortest st-path. A shortest distance query from a node s to a node t is answered by enumerating all common hubs of s and t. A labeling is "good" if all labels only contain very few hubs. Computing a labeling minimizing the index size is NP-hard (Babenko, Goldberg, Kaplan, Savchenko, & Weller, 2015).

Most works do not consider HL in its most general form, but consider a more restrictive variant called Hierarchical Hub Labels (HHL). This term was introduced by Abraham et al. (2012) but labels used in previous work (Abraham et al., 2011) were already hierarchical. A labeling is called hierarchical if an ordering of the vertices exists, such that every hub h of a vertex v comes after v in the order. Given a fixed node order, an optimal labeling can be computed efficiently (Abraham et al., 2012). The difficult task with HHL consists of computing the node order. Computing a node order minimizing the index size is also an NP-hard task (Babenko et al., 2015).

HHL is deeply coupled with a different popular speedup technique for shortest path computations called Contraction Hierarchies (CH) (Geisberger, Sanders, Schultes, & Delling, 2008). CH does not achieve the query speeds of HHL but has significantly smaller index sizes. However, for most applications even CH query times are already faster than necessary, which makes CH a very strong competitor. CH iteratively contracts nodes while inserting shortcuts to maintain all shortest path distances in the remaining graph. By following the so inserted shortcuts only a small fraction of the graph needs to be explored from every node. A node order is "good" for CH if the search spaces of every node is small. Again, computing an optimal order is NP-hard (Bauer, Columbus, Katz, Krug, & Wagner, 2010). The first HL paper on road graphs (Abraham et al., 2011) computed the label of v by explicitly storing all nodes reachable from v in the CH search space and then applying some pruning rules. Later papers have refined these rules, but every hierarchical label can be viewed as an explicitly stored and pruned CH search space. A consequence is that node orders that are "good" for CH are also "good" for HHL and vice versa, even though the formal optimization criteria differ and therefore an optimal order for one of them with respect some criterion can be slightly suboptimal for the other.

The node orders used for HHL and the original CH depend on the weights on the input graph. Substantial changes to the weights requires recomputing the node ordering. More recent work (Bauer, Columbus, Rutter, & Wagner, 2013; Dibbelt, Strasser, & Wagner, 2014) has introduced Customizable Contraction Hierarchies (CCH) and shown that node orders exist and work well that only depend on the structure of the input graph. These node orders exploit that the input graph has small balanced node-separators or has a comparative small treewidth.

In our paper we also consider two types of node orders. The first is a depth first search preorder and the second is based on small balanced edge-cuts. They are thus also independent of the input graph's weights. However, do not confuse our orders with the CCH node orders. They are not interchangeable. Using a CCH ordering will result in bad performance with our technique, just as using one of our node orders with CCH will not work well. In fact, using a preorder with CCH maximizes the maximum search space in terms of vertices instead of minimizing it. That is, an order that works well with our technique is a CCH worst case node order. Further, our orders can also not be interchanged with the weight-dependent orders needed for HHL and CH.

As described in the literature, HL answers distance queries. However, as hinted by Abraham et al. (2012), it is easy to extend hub labels to first move queries. To achieve this, the entries in the forward and backward labels are extended with a third component: a first move edge ID. If h is a forward hub of s then the corresponding entry is extended using the first edge ID of a shortest sh-path. If h is backward hub of t then the entry is extended with the first edge of a shortest ht-path.

For a st-query first the corresponding meeting hub h is determined. If $s \neq h$ then the first move is the edge ID stored in the forward label s and otherwise the first move is contained in the backward label of t. This slightly increases memory consumption but should have a negligible impact on performance. Note that distance values are needed even if one only wishes to compute first-moves as we need the distances to determine the right hub if s and t have several hubs in common.

In the context of program analysis it is sometimes desirable to construct an oracle that determines if a particular section of code can ever be reached. PWAH (van Schaik & de Moor, 2011) is one such example. Similarly to our work, the authors precompute a quadratic matrix and employ a compression scheme based on run-length encoding. The main difference is that such reachability oracles only return a yes-no answer for every query rather than the identity of a first-edge.

Another speedup technique with low average query times is Transit Node Routing (TNR) (Bast, Funke, & Matijevic, 2009; Bast, Funke, Matijevic, Sanders, & Schultes, 2007; Antsfeld, Harabor, Kilby, & Walsh, 2012). However, two independent studies (Abraham et al., 2011; Arz, Luxen, & Sanders, 2013) have come to the conclusion that (at least on roads) TNR is dominated by HL in terms of query time. Further, TNR does not optimize short range queries. A scenario that often arises is a unit that chases another unit. In most situations both units tend to be very close, which results in many short range queries. TNR is rather ineffective in this scenario.

Bulitko, Björnsson, and Lawrence (2010) present a subgoal-based approach to pathfinding. Similarities to our work include a preprocessing stage where paths on a map are precomputed, after which the results are compressed and stored into a database. The database is used to speed up the response time when a path query is posed to the system. There are substantial differences between the two approaches as well. Our method precomputes all-pairs shortest paths, eliminating graph search entirely in the "production" mode (i.e., the stage where the system is queried to provide full shortest paths or fragments of shortest paths). In contrast, Bulitko et al. restrict their precomputed database to a subset of nodes, which in turn requires some additional search in the production mode. The compression method is different in each case. Our system provides optimal paths, which is not guaranteed in the case of Bulitko et al.'s method. Besides Bulitko et al. (2010) work, pathfinding with sub-goals turned out to be a popular and successful idea in more recent work (Hernández & Baier, 2011; Bulitko, Rayner, & Lawrence, 2012; Lawrence & Bulitko, 2013; Uras, Koenig, & Hernández, 2013).

Pattern databases (PDBs) (Culberson & Schaeffer, 1998) are lookup tables that provide heuristic estimations of the true distance from a search node to a goal state. They are obtained by abstracting an original search space into a smaller space. Optimal distances in the abstracted space, from every state to a pre-established goal, are precomputed and stored into the pattern database as estimations of the distances in the original space. As such, both techniques are memory-based enhancements in problems where the solution can be represented as a path in a graph. There are several key distinctions between PDBs and CPDs. PDBs are lossy abstractions, and they are specific to a goal or subset of goals. CPDs are lossless compressions, and encode shortest paths for *every* start–target pair. Given their lossy nature, PDBs need to be used as heuristic within in a search algorithm, such for example as A*, as opposed to a complete and optimal method on its own. PDBs are commonly used in large graphs, such as implicitly defined search spaces, where exploring the entire graph in the preprocessing is impractical. In PDBs, the coarseness of the abstraction impacts the accurracy of heuristic estimations. A finer abstraction has a better quality, but it can also result in a larger PDB. Work on addressing this bottleneck include compressing pattern databases (Felner, Korf, Meshulam,

& Holte, 2007; Samadi, Siabani, Felner, & Holte, 2008). In contrast, CPDs compress all-pairs shortest paths.

3. Preliminaries

We denote by $G = (V, E)$ a graph with node set V and edge[1] set $E \subseteq V \times V$. We denote by $\deg(u)$ the number of outgoing edges of u.[2] The maximum out-degree is denoted by Δ. A *node order* $o : V \to [1, |V|]$ assigns to every node v a unique *node ID* $o(v)$. The out-going edges of every node are ordered in an arbitrary but fixed order and their position (index in the ordering) is referred to as their *out-edge ID*.

Further, there is a weight function $w : E \to \mathbb{R}_{>0}$[3]. An st-path is a sequence of edges $a_1 \dots a_k$ such that a_1 starts at s and a_k ends at t and for every i the edge a_i ends at the node where a_{i+1} starts. The weight (or cost) of a path is $\sum_i w(a_i)$. An st-path is shortest if no other st-path exists with a strictly smaller weight. The distance between two nodes s and t is the weight of a shortest st-path, if one exists. If no st-path exists, the distance is ∞. Notice that there may be multiple shortest st-paths but all of them have the same weight.

Without loss of generality we can assume that no duplicate edges (multi-edges) exist in our graphs, as if there were, we could just drop all but a shortest edge, as the other edges are not used by any shortest path. Further, using a similar argument, we can assume without loss of generality that no reflexive loops exist.

For $s \neq t$, an st-first-move is the first edge of a shortest st-path. If there are multiple shortest st-paths, there may also be multiple st-first-moves. If no st-path exists, no st-first-move exists. The formal problem we consider is the following: Given a pair of nodes s and t, find an st-first-move. If there are several valid first-moves, the algorithm can freely choose which to return.

Given a oracle that answers first move queries, we can easily extract shortest paths. Compute an st-first move a. In other words, a is the first edge of the shortest path. Next, set s to the end of a. As long as $s \neq t$, apply this procedure iteratively. Notice, that this only works as edge weights are guaranteed to be non-zero. If we allowed zero-weights, we could run into an infinite-loop problem, as the following example illustrates: Consider a graph G with two nodes x and y connected by edges xy and yx with weights zero. Denote by t some other node in G. A valid xt-first-move is using xy. Further a valid yt-first-move is using yx. If the oracle always returned these two first-moves, our path extraction algorithm would oscillate between x and y and would not terminate.

A depth first search (DFS) is a way of traversing a graph and constructing a special sort of spanning tree using backtracking. A depth-first preorder is a node order that orders nodes in the way that a DFS first sees them. The search is parameterized on the root node and on the order in which the neighbors of each node are visited. In this work we will regularly refer to depth-first preorders without stating these parameters. We always implicitly assume that the root is some arbitrary node and that the neighbors are visited in some arbitrary order.

1. The term "arc" is also used in the literature. Sometimes, the distinction is made on whether the graph is directed (in which case some authors prefer to say "arcs") or undirected. In this paper, we stick with the term "edge" in all cases.
2. In a directed graph, every ordered pair $(u, v) \in E$ is an outgoing edge of u. In an undirected graph, every edge incident to u is an outgoing edge of u.
3. We assume that this is a function $E \to \mathbb{R}_{>0}$ to be able to apply Dijkstra's algorithm during the preprocessing phase. However, one could consider arbitrary weights without negative cycles and replace every occurrence of Dijkstra's algorithm with the algorithm of Bellman and Ford (Bellman, 1958; Ford, 1956).

Run length encoding (RLE) compresses a string of symbols by representing more compactly substrings, called *runs*, consisting of repetitions of the same symbol. For instance, string $aabbbaaa$ has three runs, namely aa, bbb, and aaa. A run is replaced with a pair that contains the *start* and the *value* of the run. The start is the index of the first element in the substring, whereas the value is the symbol contained in the substring. In our example, the first run aa has the start 1 and the value a. Run bbb has the start 3 and the value b, whereas the last run has the start 6 and the value a.[4]

When the first and the last run have the same value, there is no need to encode both. The first run can easily be reconstructed in constant time in this case. First, decide whether the first run has been removed or not, this can be done by checking if the first run among the preserved ones has the start equal to 1. Secondly, if needed, reconstruct the first run, using 1 as a start position and a value equal to the value of the last encoded run. Another way of looking at this is that, if the first and the last run have the same value, we can allow them to merge, as if we wrapped around the string to form a cycle. When we allow this, we say we are using *cyclic runs*. Otherwise (never consider merging the ends of a string), we say we use *sequential runs*. See Example 1 below.

Given an ordered sequence of elements (string), we say that two positions are: *adjacent* if they are next to each other; *cyclic-adjacent* if they are adjacent or one is the first and the other is the last position in the ordering; *separated* otherwise.

Let σ be an ordered sequence of elements (symbols) over a dictionary (or alphabet) Σ. Given a symbol $\alpha \in \Sigma$, let an α*-run* be an RLE run containing symbol α. For every string σ, we denote by $N_\alpha(\sigma)$ the total number of occurrences of symbol α in σ. Further, the number of sequential α-runs is denoted by $R^s_\alpha(\sigma)$ and the number of of cyclic by $R^c_\alpha(\sigma)$. Notice that $0 \leq R^s_\alpha(\sigma) - R^c_\alpha(\sigma) \leq 1$. In other words, the number of sequential runs and the number of cyclic runs never differ by more than 1. Finally, we denote by $R^s_*(\sigma)$ the total number of sequential runs and by $R^c_*(\sigma)$ the total number of cyclic runs. In this paper, we assume that first-move compression uses cyclic runs, unless we explicitly say otherwise.

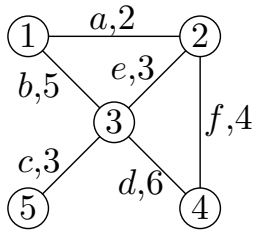
Example 1. Consider again the string $\sigma = aabbbaaa$. Compressing σ yields 1, a; 3, b; 6, a. This means that after position 1 the string consists of as. Similarily after position 3 there are bs and finally after position 6 all elements are as until the string ends. We have $N_a(\sigma) = 5$ and $N_b(\sigma) = 3$. There are three sequential runs, namely aa, bbb and aaa. The first and the third ones are a-runs, whereas the middle one is a b-run. Thus, we have $R^s_a(\sigma) = 2$, $R^s_b(\sigma) = 1$, and $R^s_*(\sigma) = 2 + 1 = 3$.

At the same time, σ has one cyclic a-run. Indeed, if we put next to each other the two ends of the string, as if the string were cyclic, all occurrences of a in the string become one solid block (i.e., one cyclic a-run). Thus, $R^c_a(\sigma) = 1$, $R^c_b(\sigma) = 1$, and $R^c_*(\sigma) = 1 + 1 = 2$.

4. Basic Idea

As mentioned in the introduction, our algorithm starts by building a $|V| \times |V|$ all-pairs first-move matrix $\mathbf{m}$. The entry at position $\mathbf{m}[i, j]$ is an ij-first-move. The central idea of our algorithm is to compress each row of $\mathbf{m}$ using RLE. The compression is performed gradually, as the matrix rows are being computed, so that the uncompressed matrix does not have to be kept in memory. To answer an st-first-move query, we run a binary search for t in the row of s. However, to achieve a good compression ratio, we first reorder the columns of $\mathbf{m}$ to decrease the total number of runs. As the columns correspond to nodes, we can regard the problem of reordering the columns as a problem

4. Alternative encodings exist, such as the value followed by the run length. E.g., $a, 2; b, 3; a, 3$ in the example.



(a) Input

s \ t	1	2	3	4	5
1	−	a	a	a	a
2	a	−	e	f	e
3	e	e	−	d	c
4	f	f	d	−	d
5	c	c	c	c	−

(b) First-Move Matrix

1	$1/a$
2	$1/a$ $3/e$ $4/f$ $5/e$
3	$1/e$ $4/d$ $5/c$
4	$1/f$ $3/d$
5	$1/c$

(c) Compressed Path Database

Figure 1: A toy example of our algorithm

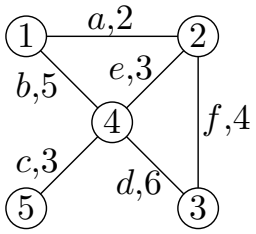
of computing a good node order. Computing an optimal node order that minimizes the number of runs is NP-hard, as we show in our theoretical analysis. Fortunately, a simple depth-first preorder works well in practice.

Sometimes, in a formal analysis, some technical details are "annoying" in the sense that they can make the presentation somewhat more complicated. The question of what symbol we should use for $\mathbf{m}[i, i]$ is such an example. In our practical implementation, we say that we do not care about the symbol, as we never query for it. To reduce the number of runs we therefore assign it either the value of $\mathbf{m}[i-1, i]$ or $\mathbf{m}[i+1, i]$. In our theoretical analysis, we make a similar assumption (i.e., a "don't care" symbol) in Sections 5 and 6. As we will state in Section 7, the assumption there is that $m[i, i]$ is a symbol different from any edge symbol. In every case, our assumptions have the purpose of keeping the analysis as simple as possible.

Example 2. Figure 1a shows a toy weighted and undirected graph, with 5 nodes and 6 edges. For each edge, we show its weight (cost), as a number, and a unique label, as a letter. A first-move matrix $\mathbf{m}$ of this graph, corresponding to the node ordering 1, 2, 3, 4, 5, is shown in Figure 1b. Recall that the entry $\mathbf{m}[r, c]$, where r is a row and c is a column, is the id of the first move of a shortest path from node r to node c. For example, $\mathbf{m}[3, 1] = e$ because e is the first step of ea, an optimal path from node 3 to node 1. Another optimal path would be the single-step path b, as both ea and b have an optimal weight (cost) of 5. Thus, we are free to choose between $\mathbf{m}[3, 1] = e$ and $\mathbf{m}[3, 1] = b$. We prefer e because this leads to a better compression of row 3 in $\mathbf{m}$, since the first two symbols of the third row, being identical, will be part of the same RLE run. We will show in Section 9 that breaking such ties in an optimal way is feasible and computationally easy. The compression of $\mathbf{m}$ for the given node ordering (or equivalently, matrix column ordering) is shown in Figure 1c.

Notice that the ordering of the nodes impacts the size of a compressed matrix. In Example 2, swapping nodes 3 and 4, as illustrated in Figure 2, would further reduce the number of RLE runs on row 2, as the two e symbols will become adjacent. The total number of runs decreases from 11 runs to 10 runs. Thus, the challenge is to find an optimal or at least a "good enough" node ordering, where the objective function is the size of the compressed first-move matrix.

The compression strategy with RLE illustrated in Example 2 is a key component of our approach. We study it theoretically in the next three sections, showing that computing an optimal node ordering is NP-hard in general, and identifying tractability islands. We present a number of effective heuristic node orderings in Section 8. A variant of our implemented method, called SRC,



(a) Input

s \ t	1	2	3	4	5
1	−	a	a	a	a
2	a	−	f	e	e
3	f	f	−	d	d
4	e	e	d	−	c
5	c	c	c	c	−

(b) First-Move Matrix

1	1/a			
2	1/a	3/f	4/e	
3	1/f	3/d		
4	1/e	3/d	4/c	
5	1/c			

(c) Compressed Path Database

Figure 2: The same toy example as in Figure 1 but with a different node ordering (i.e., nodes 3 and 4 swapped).

performs the compression as illustrated in the example. Another version of our program, called MRC, goes beyond the idea of compressing each row independently, implementing a multi-row compression strategy. These are discussed in Section 9 and evaluated empirically in Section 12.

5. First-Move Compression for Directed Graphs

Recall that the ordering of the columns of a first-move matrix $\mathbf{m}$ affects the number of RLE runs in the matrix. In this section we show that obtaining an optimal ordering is intractable in general when the input graph is directed. Our construction works with uniform edge weights. For simplicitly we therefore omit the weights in this section.

Definition 1. The *FMComp-d* (First Move Compression–Directed) problem:
Input: A directed graph $G = (V, E)$; a matrix $\mathbf{m}$ of size $|V| \times |V|$ where each cell $\mathbf{m}[i, j]$ encodes the first move on an optimal path from node i to node j; an integer k.
Question: Is there an ordering of the columns of $\mathbf{m}$ such that, if we apply RLE on each row, the total number of cyclic RLE runs summed up for all rows is k?

Theorem 1. The FMComp-d problem is NP-complete.

Proof. It is easy to see that the problem belongs to NP, as a solution can be guessed and verified in polynomial time.

The NP-hardness is shown as a reduction from the Hamiltonian Path Problem (HPP) in an undirected graph. Let $G_H = (V_H, E_H)$ be an arbitrary undirected graph, and define $n = |V_H|$ and $e = |E_H|$. Starting from G_H, we build an instance of the FMComp-d problem. According to Definition 1, such an instance includes a directed graph, which we call G_F, the first-move matrix $\mathbf{m}$ of G_F, and a number.

$G_F = (V_F, E_F)$ is defined as follows. For each node $u \in V_H$, define a node in V_F. We call these nodes in V_F type-n nodes, to indicate they are created from original nodes in V_H. For each edge $(u, v) \in E_H$, define a new node $n_{uv} \in V_F$ (type-e nodes). For each new node n_{uv}, define two edges in E_F, one from n_{uv} to u and one from n_{uv} and v. There are no other edges in E_F. See Figure 3 for an example.

Table 1 shows the first-move matrix of the running example. Given a type-n node u, all other nodes are unreachable from u in the graph G_F. Thus, the matrix row corresponding to u has only

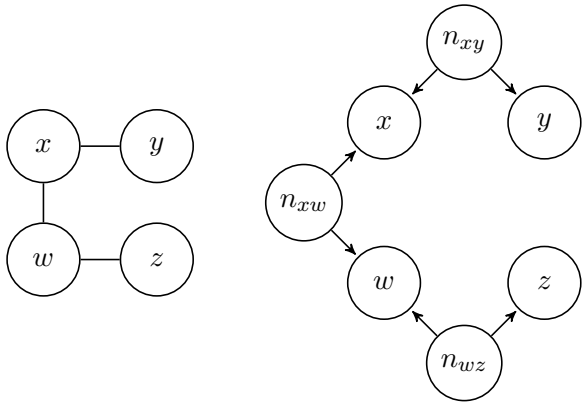


Figure 3: Left: sample graph G_H. Right: G_F built from G_H. In G_F, x, y, w, z are type-n nodes. Nodes n_{ij} have type e.

	x	y	w	z	n_{xy}	n_{xw}	n_{wz}	Nr. cyclic runs
x	-	2	2	2	2	2	2	1
y	2	-	2	2	2	2	2	1
w	2	2	-	2	2	2	2	1
z	2	2	2	-	2	2	2	1
n_{xy}	0	1	2	2	-	2	2	3
n_{xw}	0	2	1	2	2	-	2	4
n_{wz}	2	2	0	1	2	2	-	3

Table 1: First-move matrix for the running example. Both rows and columns follow the node ordering $x, y, w, z, n_{xy}, n_{xw}, n_{wz}$.

one non-trivial symbol,[5] which we chose to be symbol 2, and which denotes that a node is not reachable. Such rows have one RLE run each, regardless of the node ordering.

A matrix row corresponding to a type-e node n_{uv} has three distinct (non-trivial) symbols in total: one symbol for the edge to node u, another symbol for the edge to node v, and the "non-reachable" symbol 2 for every other node. Without any generality loss, we use symbol 0 for the edge to u, and symbol 1 for the edge to v. It is easy to see that, when nodes u and v are cyclic-adjacent in a given ordering, the n_{uv}'s row has 3 RLE runs. When u and v are separated, the row will have 4 RLE runs. See Table 1 for a few sample orderings.

We claim that HPP has a solution iff FMComp-d has a solution with $4e + 1$ RLE runs. Let $v_{i_1}, v_{i_2} \ldots, v_{i_n}$ be a solution of HPP (i.e., a Hamiltonian path in G_H), and let $P \subseteq E_H$ be the set of all edges included in this solution. We show that the node ordering in V_F starting with $v_{i_1}, \ldots, v_{i_n}$, followed by the type-e nodes in an arbitrary order, will result in $4e+1 = 3(n-1)+4(e-n+1)+n$ runs, with $3n - 3$ runs in total for the type-e rows[6] corresponding to edges in P; $4(e - n + 1)$ runs in total for the remaining type-e rows; and n runs in total for the type-n rows.

5. By trivial symbol we mean the that "don't care" symbol $-$. Recall that it has no impact on the number of runs. For simplicity, we can safely ignore this symbol in our discussion.

6. We say that a row has a type-n (or type-e) iff its associated node has that type.

Indeed, for each edge $(u, v) \in P$, the type-e row in $\mathbf{m}$ corresponding to node $n_{uv} \in V_F$ will have 3 RLE runs, since u and v are adjacent in the ordering. There are $n-1$ edges in a Hamiltonian path, with a total number of RLE runs of $3(n-1)$ for all these rows.

For an edge $(u, v) \notin P$, the two nodes are separated and therefore the corresponding matrix row will have 4 runs. This sums up to $4(e-n+1)$ RLE runs for all rows corresponding to edges not included in the Hamiltonian path.

Conversely, consider a node ordering that creates $4e+1 = 3(n-1)+4(e-n+1)+n$ RLE runs in total. We show that the ordering has all type-n nodes as a contiguous block,[7] and that their ordering is a Hamiltonian path in G_H. This is equivalent to saying that there exist $n-1$ pairs of type-n nodes u and v such that u and v are cyclic-adjacent in the ordering, and $(u, v) \in E_H$.

Here is a proof by contradiction. Assume there are only $p < n-1$ pairs of type-n nodes u and v such that u and v are cyclic-adjacent in the ordering, and (u, v) is an edge in E_H. For each of these p pairs, the row corresponding to the type-e node n_{uv} will have 3 RLE runs. The remaining $e-p$ type-e rows will be 4 RLE runs each. As mentioned earlier, the type-n rows have n runs in total, regardless of the ordering. Thus, the total number of RLE runs is $3p+4(e-p)+n = 4e-p+n > 4e-(n-1)+n = 4e+1$. Contradiction. □

6. Compression for Undirected Weighted Graphs

We turn our attention to undirected weighted graphs, showing that computing an optimal ordering is NP-complete.

Definition 2. The *FMComp-uw* problem (First Move Compression–Undirected, Weighted) is defined as follows.
Input: An undirected weighted graph $G = (V, E)$; a matrix $\mathbf{m}$ of size $|V| \times |V|$ where a cell $\mathbf{m}[i, j]$ stores the first move on an optimal path from node i to node j; an integer k.
Question: Is there an ordering of $\mathbf{m}$'s columns such that, if we apply run length encoding (RLE) on each row, the total number of cyclic RLE runs in the matrix is at most k?

As a stepping stone in proving the NP-hardness of FMComp-uw, we introduce a problem that we call SimMini1Runs (Definition 3), and prove its NP-completeness. SimMini1Runs is inspired by the work of Oswald and Reinelt (2009), who have studied the complexity of a problem involving the so-called *k-augmented simultaneous consecutive ones* property ($C1S_k$) for a 0/1 matrix (i.e., a matrix with only two symbols, 0 and 1). By definition, a 0/1 matrix has the $C1S_k$ property if, after replacing at most k 1s with 0s, the columns and the rows of the matrix can be ordered so that, for each row and for each column, all 1s on that row or column come as one contiguous block. Oswald and Reinelt (2009) have proven that checking whether a 0/1 matrix has the $C1S_k$ property is NP-complete. Our proof for SimMini1Runs is related, as we point out later in the proof.

Given a 0/1 matrix $\mathbf{o}$, an ordering of its columns, and an ordering of its rows, let the *global sequential 1-runs count* $G_1^s(\mathbf{o})$ be the number of sequential 1-runs summed over all rows and all columns. That is,

$$G_1^s(\mathbf{o}) = \sum_{\sigma} R_1^s(\sigma),$$

7. Here, the notion of a contiguous block allows the case when part of the block is at the end of the sequence, and the other part is at the beginning, as if the sequence were cyclic.

$$\mathbf{o} = \begin{array}{c} \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \left[\begin{array}{ccc} 0 & 1 & 1 \\ 1 & 0 & 1 \end{array} \right] \end{array}$$

Figure 4: Running example 0/1 matrix $\mathbf{o}$. Rows are labelled as r_i, whereas c_j represent column labels.

where σ is iterated through $\mathbf{o}$'s rows and columns. For instance, $G_1^s(\mathbf{o}) = 6$ for the matrix $\mathbf{o}$ shown in Figure 4.

Definition 3. The *Simultaneous Mini 1-Runs (SimMini1Runs)* problem is defined as follows.
Input: A 0/1 matrix $\mathbf{o}$ so that every row and column contain at least one value of 1; an integer k.
Question: Is there an ordering of the columns, and an ordering of the rows, so that $G_1^s(\mathbf{o}) \leq k$?

Theorem 2. SimMini1Runs is NP-complete.

The proof is available in Appendix A.

Lemma 1. Let σ be a 0/1 string so that it starts with a 0, or it ends with a 0, or both. Then $R_1^s(\sigma) = R_0^c(\sigma)$.

Proof. Case (i): σ starts with a 0 and ends with a 1. As the two end symbols are different, sequential runs and cyclic runs are identical. As 0-runs and 1-runs alternate, their numbers are identical. Case (ii), when σ starts with a 1 and ends with a 0, is similar to the previous one.

Case (iii): σ has a 0 at both ends. As 0-runs and 1-runs alternate, and we have 0-runs at both ends, it follows that $R_1^s(\sigma) = R_0^s(\sigma) - 1 = R_0^c(\sigma)$. □

Theorem 3. FMComp-uw is NP-complete.

Proof. The NP-hardness is shown with a reduction from SimMini1Runs. Consider an arbitrary SimMini1Runs instance $\mathbf{o}$ with m rows and n columns. Figure 4 shows a running example. We build an undirected weighted graph $G = (V, E)$ as follows. V has 3 types of nodes, to a total of $m + n + 1$ nodes. Each *column* of $\mathbf{o}$ generates one node in V. We call these c-nodes. Each row generates one node as well (r-nodes). There is an extra node p called the hub node.

One r-node r_i and one c-node c_j are connected through a unit-cost edge iff $\mathbf{o}[r_i, c_j] = 1$. In addition, there is an edge with a weight of 0.75 between p and every other node. No other edges exist in graph G. See Figure 5 for an example.

Let $\mathbf{m}$ be the first-move matrix of G. The row of p has a fixed number of runs, namely $m + n$,[8] regardless of the ordering of $\mathbf{m}$'s columns. Let v be a c-node or r-node. Apart from v's adjacent nodes, all other nodes are reached through a shortest path of cost 1.5 whose first move is the edge (v, p). The matrix $\mathbf{m}$ for the running example is shown in Figure 6.

Let T_1 be the total number of occurrences of symbol 1 in matrix $\mathbf{o}$. We claim that there is an ordering of $\mathbf{o}$'s rows and columns that results in at most k sequential 1-runs (summed up for all rows and all columns) iff there is an ordering of the columns of $\mathbf{m}$ resulting in at most $k + 2T_1 + m + n$

8. Recall that we can ignore the "don't care" symbol $\mathbf{m}[p, p] = -$, which has no impact on the number of RLE runs.

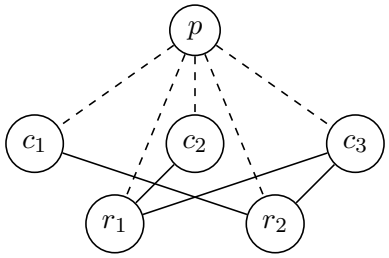


Figure 5: Graph in the running example. Dashed edges have a weight of .75, whereas solid lines are unit-cost edges.

$$\mathbf{m} = \begin{array}{c} \\ r_1 \\ r_2 \\ c_1 \\ c_2 \\ c_3 \\ p \end{array} \begin{array}{c} \begin{array}{cccccc} r_1 & r_2 & c_1 & c_2 & c_3 & p \end{array} \\ \begin{bmatrix} - & 0 & 0 & 1 & 2 & 0 \\ 0 & - & 1 & 0 & 2 & 0 \\ 0 & 1 & - & 0 & 0 & 0 \\ 1 & 0 & 0 & - & 0 & 0 \\ 1 & 2 & 0 & 0 & - & 0 \\ 1 & 2 & 3 & 4 & 5 & - \end{bmatrix} \end{array}$$

Figure 6: The first-move matrix for the running example. Without any generality loss, 0 is the move towards p. The incident edges of a given node are counted starting from 1.

cyclic RLE runs in total (summed up for all rows). Thus all rows of $\mathbf{m}$, except for p's row, have at most $k + 2T_1$ runs in total.

Let $r_{i_1}, \ldots r_{i_m}$ and $c_{j_1}, \ldots c_{j_n}$ be the row and column orderings in $\mathbf{o}$ that result in at most k sequential RLE runs on all rows and columns. We show that the ordering $r_{i_1}, \ldots r_{i_m}, c_{j_1}, \ldots c_{j_n}, p$ of $\mathbf{m}$'s columns generates at most $k + 2T_1 + m + n$ cyclic runs. Clearly, for every *row or column* σ in $\mathbf{o}$, there is a corresponding *row* σ' in $\mathbf{m}$ (see again Figures 4 and 6 for an example). According to the steps explained earlier and illustrated in Figures 4 and 6, σ' is obtained from σ as follows. All original 0s are preserved. All original 1s are replaced with distinct consecutive integers starting from 1. In addition, σ' is padded with 0s at one or both of its ends. Since σ' has 0s at one or both its ends, it follows that $R^s_1(\sigma) = R^c_0(\sigma')$.[9] It follows that $R^c_*(\sigma') = R^c_0(\sigma') + N_1(\sigma) = R^s_1(\sigma) + N_1(\sigma)$. Summing up $R^c_*(\sigma')$ over all rows σ' of $\mathbf{m}$, except for p's row, we obtain

$$\sum_{\sigma' \in \rho(\mathbf{m}) \setminus \{p\}} R^c_*(\sigma') = \sum_{\sigma \in \beta(\mathbf{o})} R^s_1(\sigma) + \sum_{\sigma \in \beta(\mathbf{o})} N_1(\sigma) \leq k + 2T_1,$$

where ρ denotes the set of rows of a matrix, κ is the set of columns, and $\beta = \rho \cup \kappa$. It follows that $\mathbf{m}$'s rows have at most $k + 2T_1 + m + n$ cyclic RLE runs in total (that is, summed up for all rows).

Conversely, assume an ordering of $\mathbf{m}$'s columns with at most $k + 2T_1 + m + n$ cyclic RLE runs in total (for all rows). This means that summing up the runs of all rows of $\mathbf{m}$, except for node p's row, results in at most $k + 2T_1$ runs. As there are exactly $2T_1$ distinct runs different from 0-runs, it

9. $R^s_1(\sigma) = R^c_0(\sigma)$ from Lemma 1, and $R^c_0(\sigma) = R^c_0(\sigma')$ by construction.

follows that there are at most k 0-runs in total:

$$\sum_{\sigma' \in \rho(\mathbf{m}) \setminus \{p\}} R_0^c(\sigma') \leq k.$$

Let $r_{i_1}, \ldots r_{i_m}, c_{j_1}, \ldots c_{j_n}, p$ be a re-arragement of $\mathbf{m}$'s columns so that: all r-nodes come in one contiguous block, and their relative ordering is preserved; all c-nodes are one contiguous block, and their relative ordering is preserved.

Since G restricted to c-nodes and r-nodes is bi-partite, this rearrangement cannot possibly increase the number of RLE runs. (If anything, it could eliminate some 0-runs). This is not hard to prove. For example, if the current matrix row corresponds to an r-node a as a source node, we have that $\mathbf{m}[a, b] = 0$ for every other r-node b, since a, p, b is an optimal path from a to b. Also, $\mathbf{m}[a, p] = 0$. Our rearrangement moves all nodes b into a block that is cyclic-adjacent to p, which does not create any new run. The case with a c-node source is similar.

We order $\mathbf{o}$'s columns as $c_{j_1}, \ldots c_{j_n}$, and $\mathbf{o}$'s rows as $r_{i_1}, \ldots r_{i_m}$. With these orderings, the relation between a row or column σ of $\mathbf{o}$ and its corresponding row σ' in $\mathbf{m}$ is as follows. All non-zero values in σ' are converted into 1s in σ. Some of σ' 0s from one or both of its ends are cut away in σ. Since σ' contains some 0s at one or both ends, $R_1^s(\sigma) = R_0^c(\sigma')$, according to Lemma 1. It follows that

$$\sum_{\sigma \in \rho(\mathbf{o}) \cup \kappa(\mathbf{o})} R_1^s(\sigma) = \sum_{\sigma' \in \rho(\mathbf{m}) \setminus \{p\}} R_0^c(\sigma') \leq k.$$

□

7. Fighting Complexity with Decomposition

So far all our results have been negative. We have shown that computing an optimal order on a large class of graphs is NP-hard. In this section we identify tractability islands. We show that the problem can be decomposed along articulation points (which are related to cuts of size 1). In particular, this implies (as shown in this section) that a depth-first preorder is an optimal node ordering on trees. Further we are able to construct optimal orders efficiently on a broader class of graphs than trees: We show that the problem is fixed-parameter tractable in the size of the largest component of the graph that has no articulation points.

Definition 4. We say that a node x of a graph G is an *articulation point* if removing x and its adjacent edges from G would split the graph into two or more disjoint connected subgraphs $G_1 \ldots G_n$.

Figure 7 shows an example. In the rest of this section we focus on graphs G with articulation points x. We consider cyclic runs. In previous sections, we treated $\mathbf{m}[s, s]$ as a "don't care" symbol, with no impact on the number of runs. In this section, we make a different assumption. Every cell $\mathbf{m}[s, s]$ gets its own distinct symbol, called the s-singleton, which always creates its own run, and which can not be merged with adjacent symbols into a common run. This makes the proofs easier and clearly does not have a significant impact on the number of runs.

Definition 5. We call an *x-block ordering* any node ordering where x comes first, the nodes of G_1 come next as a contiguous block, and so on all the way to block G_n.

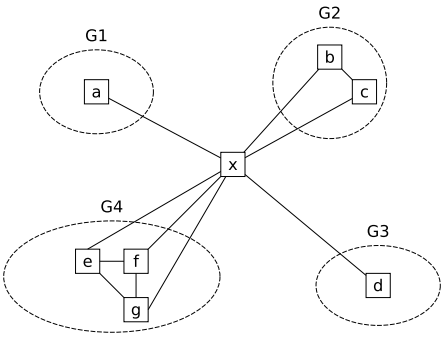


Figure 7: A graph with an articulation point x. Removing x would decompose the graph into four disjoint components, depicted as G_1 to G_4.

In the example shown in Figure 7, the ordering $o = x, a, b, c, d, e, f, g$ is an example of an x-block ordering.

We use $o|_{G'}$ to denote the projection of the node ordering o to a subset of nodes corresponding to a subgraph G' of G. We use Γ_i to denote the subgraph induced by the nodes from $G_i \cup \{x\}$.[10] We say that an order o is a rotation of another order o' when o is obtained from o' by taking a block of o''s elements from the beginning and appending it to the end. For instance, d, e, f, g, x, a, b, c is a rotation of x, a, b, c, d, e, f, g. More formally, o is a rotation of o' when two sub-orders α and β exist such that $o' = \alpha, \beta$ and $o = \beta, \alpha$.

Lemma 2. Let x be an articulation point of a graph G. Every node order o can be rearranged into an x-block ordering o' without increasing the number of runs on any row.

Given a graph G, a node ordering o and a row subset S, let $N(o, G, S)$ be the number of runs restricted to subset S. Clearly, $N(o, G, G)$ is the total number of runs.

Lemma 3. Given any x-block ordering o, we have that:

1. $N(o, G, G_i) = N(o|_{\Gamma_i}, \Gamma_i, G_i)$; and
2. $N(o, G, \{x\}) = 1 - n + \sum_i N(o|_{\Gamma_i}, \Gamma_i, \{x\})$; and
3. $N(o, G, G) = 1 - n + \sum_i N(o|_{\Gamma_i}, \Gamma_i, \Gamma_i)$.

The proofs to Lemmas 2 and 3 are available in Appendix B.

Theorem 4. Given an optimal order o_i for every subgraph induced by Γ_i, we can construct an optimal global ordering o for G as following. Obtain new orderings o'_i by rotating o_i such that x comes first, and then removing x. Then, $o = x, o'_1, \ldots, o'_n$ is optimal.

Proof. We show, by contradiction, that the global ordering o is optimal. Notice that $o|_{\Gamma_i}$ is optimal for Γ_i. Assume there is a strictly better ordering o'. According to Lemma 2, there exists an x-block

10. A subgraph induced by a subset of nodes S contains all nodes from S and all edges whose both ends belong to S.

ordering o'' at least as good as o'. We have

$$\begin{aligned} N(o,G,G) &= 1-n+\sum_i N(o|_{\Gamma_i},\Gamma_i,\Gamma_i) \\ &\leq 1-n+\sum_i N(o''|_{\Gamma_i},\Gamma_i,\Gamma_i) \\ &= N(o'',G,G) \leq N(o',G,G) \end{aligned}$$

which is a contradiction with o' being strictly better (i.e., $N(o',G,G) < N(o,G,G)$). □

Lemma 4. If G is a tree and o is a depth-first preorder of G (with arbitrary root) then o is a rotated x-block order for every node x.

Proof. Every preorder induces a rooted tree. With respect to this root every node x (except the root) has a parent p and a possibly empty sequence of direct children $c_1 \dots c_n$ ordered in the way that the depth-first search visited them. When removing x, G is decomposed into the subgraphs G_p, $G_{c_1} \dots G_{c_n}$. If x is the root then G_p is the empty graph. The order o has the following structure: some nodes of G_p, x, all nodes of G_{c_1} ... all nodes of G_{c_n}, the remaining nodes of G_p. Clearly this is a rotated x-block ordering. □

Theorem 5. If $G = (V, E)$ is a tree and o a depth-first preorder of G then $N(o, G, G) = 3|V| - 2$.

Proof. A direct consequence of Lemma 4 is that every node v has as many runs as $d(v) + 1$, where $d(v)$ is the degree of the node. The $+1$ comes from the v-singleton. We have thus

$$N(o,G,G) = \sum_{v \in V} (d(v)+1) = 2|E| + |V| = 3|V| - 2.$$

□

Theorem 6. Computing an optimal order for graph G is fixed-parameter tractable in the size of largest two-connected component of G (i.e., the largest component with no articulation points).

Proof. Recursively decompose G at articulation points until only two-connected parts are left. As the size of the parts does not depend on the size of G we can enumerate all orders and pick the best one. Given optimal orders for every part we can use Theorem 4 to construct an optimal global order. □

Being able to decompose graphs along articulation points is useful on real-world road networks. These graphs tend to have a large two-connected component with many small trees attached. For example the Europe graph made available during the 9th DIMACS challenge (Demetrescu, Goldberg, & Johnson, 2009) has about 18M nodes in total of which only 11.8M are within the largest two-connected component. Our result allows us to position 6.2M nodes in the order fast and optimally using only local information.

8. Heuristic Node Orderings

In Sections 5 and 6 we have shown that computing an optimal order is NP-hard in theory. Fortunately, NP-hardness does not rule out the existence of "good" heuristic orderings that can be computed quickly. Indeed, a simple depth-first preorder works very well in practice. This observation can partially be explained by the fact that, as shown in Section 7, a depth-first preorder is optimal for trees. However, we can also explain it using more informal and intuitive terms.

An ordering is "good" if neighboring nodes in the graph are assigned neighboring IDs. This is consistent with the previous observation (Sankaranarayanan et al., 2005; Botea, 2011) that, if two target nodes are close to each other, chances are that the first move from a current node towards any of these targets is the same. A depth-first preorder achieves the goal of assigning close IDs to neighboring nodes in low degree graphs. A node is either an interior node, the root, or a leaf in the DFS tree. Most of the nodes of a graph tend to be interior nodes. For these, a depth-first preorder will assign to two neighboring nodes adjacent IDs. Denote by v an internal node, by p its parent and by c the first child of v. The ID of p will be the ID of v minus 1, whereas the ID of c is the ID of v plus one. We can guarantee nothing for other children. However, if the average node degree is low, as is the case in for example road graphs, then there just are not that many other children.

Besides using depth-first preorders, we also propose another heuristic that is based on the intuition of assigning close IDs to close nodes. It is based on cuts. The above formulated intuitive optimization criterion can also be formulated as following: For every edge, both endpoints should have a close ID. Obviously this can not be fulfilled for all edges at once. For this reason the proposed ordering tries to identify a small set of edges for which this property may be violated. It does this using balanced edge cuts. Given a graph with n nodes we want to assign IDs in the range $[1, n]$ using recursive bisection. In the first step our algorithm bisects the graph into two parts of nearly equal node counts and a small edge cut size. It then divides the ID range in the middle and assigns the lower IDs to one part and the upper IDs to the other part. It continues by recursively bisecting the parts and further dividing the associated ID ranges until only parts of constant size are left. As described so far the algorithm is free to decide to which part it assigns the lower and to which the upper ID ranges. For this reason we augment it by tracking for every node v two counters $h(v)$ and $\ell(v)$ representing the number of neighbors with guaranteed higher and a lower IDs. Initially these counters are zero. At every bisection after the ranges are assigned the algorithm iterates over the edge cut increasing the counters for the border nodes. When deciding which of two parts p and q gets which ranges it uses these counters to estimate the ID distance of both parts to the nodes around them. It evaluates

$$\sum_{v \in q} h(v) - \sum_{v \in q} \ell(v) < \sum_{v \in p} h(v) - \sum_{v \in p} \ell(v)$$

and assigns the higher IDs to p if the condition holds. When the algorithm encounters a part that is too small to be bisected it assigns the IDs ordered by $\ell(v) - h(v)$.

9. Compression

Let $a_1 \dots a_n$ denote an uncompressed row in the first-move matrix. As stated previously, SRC compresses it as a list of runs ordered by their start. As the compressed rows vary in size, we need an additional *index array* that maps each source node s onto the memory offset of the first run in the

row corresponding to s. We arrange the rows consecutively in memory and therefore the end of s's row is also the start of $s+1$'s row. We therefore do not need to store the row ends.

9.1 Memory Consumption

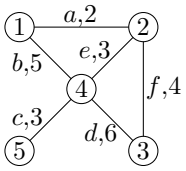
We required node IDs to be encodable in 28 bits and out-edge IDs in 4 bits. We encode each run's start in the upper 28 bits of a 32-bit machine word and its value in the lower 4 bits. The total memory consumption is therefore $4 \cdot (|V|+1+r)$ bytes where r is the total number of runs over all rows and $|V|+1$ is the number of offsets in the index array. Notice that, in our implementation, we assume that 4 bytes per index entry are sufficient, which is equivalent to saying that $r < 2^{32}$. The formula can easily be adapted to other sizes (i.e., number of bits) for node IDs, edge IDs, and index entries. For instance, when the sum of one node ID and one edge ID is K bytes, and J bytes are sufficient to encode the index of any run (in other words, the number r fits into J bytes), the formula becomes $J \cdot (|V|+1) + K \cdot r$ bytes.

9.2 Computing Rows

Rows are computed individually by running a variant of Dijkstra's one-to-all algorithm for every source node s and then compressed as described in detail in Section 9.3. However, depending on the graph it is possible that shortest paths are not unique and may differ in their first edge. It is therefore possible that multiple valid uncompressed rows exist that tie-break paths differently. These rows may also differ in their number of runs and therefore have different compressed sizes. To minimize the compressed size of a row, instead of using Dijkstra's algorithm to compute one specific row $a_1 \dots a_n$ we modify it to compute sets $A_1 \dots A_n$ of valid first move edges. We require that for each $a_t \in A_t$ a shortest st-path must exist that uses a_t as its first edge. Our algorithm maintains alongside the tentative distance array $d(t)$ for each node t a set of valid first move edges A_t. If the algorithm relaxes an edge (u, v) decreasing $d(v)$ it performs $A_v \leftarrow A_u$. If $d(u) + w(u, v) = d(v)$ then it performs $A_v \leftarrow A_v \cup A_u$. As we restricted the out-degree of each node to 15 we can store the A_t sets as 16-bit bitfields. Set union is performed using a bitwise-or operation.

9.3 Compressing Rows with Run Length Encoding

For every target the compression method is given a set of valid first move edges and may pick the one that minimizes the compressed size. We formalize this subproblem as following: Given a sequence of sets $A_1 \dots A_n$ find a sequence $a_1 \dots a_n$ with $a_i \in A_i$ that minimizes the number of runs. We show that this subproblem can be solved optimally using a greedy algorithm. Our algorithm begins by determining the longest run that includes a_1. This is done by scanning over the $A_1 \dots A_i A_{i+1}$ until the intersection is empty: $\bigcap_{j \in [1,i]} A_j \neq \emptyset$ but $\bigcap_{j \in [1,i+1]} A_j = \emptyset$. The algorithm then chooses a value from the intersection (it does not matter which) and assigns it to $a_1 \dots a_i$. It continues by determining the longest run that starts at and contains a_{i+1} in the same way. This procedure is iterated until the row's end is reached. This approach is optimal because we can show that an optimal solution with a longest first run exists. No valid solution can have a longer first run. An optimal solution with a shorter first run can be transformed by increasing the first run's length and decreasing the second one's without modifying their values. As subsequences can be exchanged without affecting their surroundings we can conclude that the greedy strategy is optimal.



(a) Input

1	$1/a$
2	$1/a$ $3/f$ $4/e$
3	$1/f$ $3/d$
4	$1/e$ $3/d$ $4/c$
5	$1/c$

(b) SRC

1	X
2	X $3/f$ $4/e$
3	Y $1/f$
4	Y $1/e$ $4/c$
5	Z

(c) MRC per row info.

X	$1/a$
Y	$3/d$
Z	$1/c$

(d) MRC per group info.

Figure 8: MRC applied to the toy graph from Figure 2, reproduced here for convenience, at the left. Part (b) illustrates the SRC input and the full runs $\mathbf{R}_s$ of every row. Part (c) show the groups (X, Y, or Z) of each row and the row-specific runs $\mathbf{R'}_s$. Finally, part (d) depicts the runs $\mathbf{R'}_g$ shared by the rows in each group.

9.4 Merging Rows using Groups

To compress individual rows we have exploited that shortest paths from s to t_1 and t_2 often have the same first move if t_1 and t_2 are close. A similar observation can be made for close source nodes s_1 and s_2. Their compressed rows tend to resemble each other. We want to further compress the data by exploiting this redundancy. We call this technique multi-row compression (MRC) and we illustrate it in Figure 8. We partition the nodes into groups and store for each group the information shared by all nodes in the group. At each row we only store the information unique to it. Denote by $g(s)$ the unique group of node s. Two runs in different rows with the same start and same value will have the same 32-bit pattern. Denote by $\mathbf{R}_s$ the set of runs in the row of s. Instead of storing for each row s the whole set $\mathbf{R}_s$ we store for each group h the intersection of all rows. That is, we store $\mathbf{R'}_h = \bigcap_{i \in h} \mathbf{R}_i$. For each row s we store $\mathbf{R'}_s = \mathbf{R}_s \setminus \mathbf{R}_{g(s)}$. Recall that a query with target t consists of finding $\max\{x \in \mathbf{R}_s \mid x < t'\}$ (where $t' = 15t + 16$). Notice that this formula can be rewritten using basic set logic as $\max\{\max\{x \in \mathbf{R'}_s \mid x < t'\}, \max\{x \in \mathbf{R'}_{g(s)} \mid x < t'\}\}$ which can be implemented using two binary searches if all $\mathbf{R'}_i$ are stored as ordered arrays. Note that we need a second index array to lookup the $\mathbf{R'}_g$ for the groups g.

9.5 Computing Row Groups

By design close source nodes have close node IDs and thus neighbouring rows. This motivates restricting ourselves to *row-run groupings*. That is, for each group h there are rows i and j such that all rows in $[i, j]$ belong to the group. An optimal row-run grouping can be computed using dynamic programming. Denote by $S(n)$ the maximum number of runs saved compared to using no group-compression restricted to the first n rows. Notice that $S(1) = 0$. Given $S(1) \dots S(n)$ we want to compute $S(n + 1)$. Obviously the $n + 1$'s row must be part of the last group. Suppose that the last group has length ℓ then we save in total $S(n + 1 - \ell) + (\ell - 1) \cdot |\bigcap_{i \in [n+1-\ell, n+1]} \mathbf{R}_i|$ runs. As there are only n different values for ℓ we can enumerate, with brute force, all possible values, resulting in an algorithm with a running time in $\Theta(n^2)$. We observe that the intersection of large groups often seems to be nearly empty and therefore we only test values for $\ell \leq 100$ resulting in a $\Theta(n)$ heuristic.

10. Queries

Given a source node s and a target node t (with $s \neq t$) the algorithm determines the first edge of a shortest st-path. It does this by first determining the start and the end of the compressed row of s using the index array. It then runs a binary search to determine the run containing t and the corresponding out-edge ID. More precisely the algorithm searches for the run with the largest start that is still smaller or equal to t. Recall that we encode each run in a single 32-bit machine word with the higher 28 bits being the run's start. We can reinterpret these 32-bits as unsigned integers. The algorithm then consists of a binary search in an ordered 32-bit integer for the largest element not larger than $16t + 15$ (i.e., t in the higher 28 bits and all 4 lower bits set).

Extracting a path using CPDs is an extremely simple recursive procedure: beginning at the start node we extract the first move toward the target. We follow the resultant edge to a neighbouring node and repeat the process until the target is reached.

11. Experimental Setup

To evaluate our work we consider two types of graphs: road graphs and grid-based graphs. In all cases we assume that node IDs can be encoded within 28-bit integers. Further we assume that $\Delta \leq 15$, and that we use a distinct value (15) to indicate an invalid edge. This allows us to encode out-edge IDs within 4 bits. Note that the concatenation of a node ID and an out-edge ID fits into a single 32-bit machine word.

Our experiments were performed on a quad core i7-3770 CPU @ 3.40GHz with 8MB of combined cache, 8GB of RAM running Ubuntu 13.10. All algorithms were compiled using g++ 4.8.1 with -O3. All reported query times use a single core.

11.1 Grid Graphs

We have chosen three benchmark problem sets drawn from real computer games. The first two sets of benchmark instances have appeared in the 2012 Grid-Based Path Planning Competition. The third benchmark set consists of two worst case maps in terms of size. These two maps are available as part of Nathan Sturtevant's extended problem repository at `http://movingai.com/benchmarks/` but are not part of the 2012 competition set.

- The first benchmark set features 27 maps that come from the game `Dragon Age Origins`. These maps have 16K nodes and 119K edges, on average.
- The second benchmark set features 11 maps that come from the game `StarCraft`. These maps have 288K nodes and 2.24M edges, on average.
- The third benchmark set comprises two large grids which we evaluate separately. These are the largest maps available for the two games. From the extended `Dragon Age Origins` problem set we choose the map called `ost100d`. It has 137K nodes and 1.1M edges. From the extended `StarCraft` problem set we choose the map called `TheFrozenSea`. It has 754K nodes and 5.8M edges. Note that ost100d, while being the largest `Dragon Age Origins` map, is smaller than the average `StarCraft` map.

All grid maps under evaluation are undirected and feature two types of edges: straight edges which have a weight of 1.0 and diagonal edges which have a weight of $\sqrt{2}$.

11.2 Road Graphs

In the case of road graphs we have chosen several smaller benchmarks made available during the 9th DIMACS challenge (Demetrescu et al., 2009).

- The `New York City` map (henceforth, NY) has 264K nodes and 730K edges.
- The `San Francisco Bay Area` (henceforth, BAY) has 321K nodes and 800K edges.
- Finally, `the State of Colorado` (henceforth, COL) has 436K nodes and 1M edges.

For all three graphs travel time weights (denoted using a `-t` suffix) and geographic distance weights (denoted using `-d`) are available.

11.3 Comparisons

We implemented our algorithm in two variants: single-row-compression (SRC) not using the row merging optimization, and multi-row-compression (MRC), using this optimization. We compare both these approaches with two recent and state-of-the-art methods: Copa (Botea & Harabor, 2013b) and RXL (Delling et al., 2014). We evaluate two variants of Copa. The first variant, which we denote Copa-G, appeared at the 2012 GPPC and is optimised for grid-graphs. We use the original C++ implementation which is available from the competition repository (Sturtevant, 2012a). The second variant, which we denote Copa-R, is optimised for road graphs. This algorithm is described in (Botea & Harabor, 2013a); we used the original C++ implementation of this program version as well.

RXL is the newest version of the Hub-Labeling algorithm. We asked the original authors to run the experiments for us presented below. These experiments were carried out on a Xeon E5-2690 @ 2.90 GHz. To compensate for the lower clock speed, when compared to our test machine, we scale the query times of RXL by a factor of 2.90/3.40 = 85%. It is important to note that this implementation of RXL computes path distances instead of first-moves. As discussed in Section 2 this should not make a significant difference for query times. However it is unclear to us whether it is possible to incorporate the additional data needed for first-move computation into the compression schemes presented by Delling et al. (2014). The reported RXL database sizes should therefore be regarded as lower bounds.

12. Results

We evaluate our two algorithms (SRC and MRC) in terms of preprocessing time, compression performance and query performance. We also study the impact of a range of heuristic node orderings using these same metrics. There are three such variants, each distinguished by suffix. The suffix +cut indicates a node ordering based on the balanced edge-separators graph cutting technique described in Section 8. The suffix +dfs indicates a node ordering based on depth-first search traversal, again as described in Section 8. The suffix +input (or shorter +inp) indicates the order of the nodes is taken from the associated input file. In the case of grid graphs we ordered nodes lexicographically, first by y- and then by x-coordinates. Where applicable we compare our work against the state-of-the-art first-move algorithms Copa-R and Copa-G. We also compare against the very recent hub labeling technique known as RXL and a more space efficient (but not as fast) variant called CRXL.

Benchmark	Average Preprocessing Time (seconds)								
	Compute Order			Single Row Compression			Multi Row Compression		
	+cut	+dfs	+input	+cut	+dfs	+input	+cut	+dfs	+input
DIMACS	16	< 1	0	1950	1982	2111	1953	1985	2125
Dragon Age Origins	2	< 1	0	32	35	38	33	36	40
StarCraft	18	< 1	0	1979	2181	2539	1993	2195	2574
ost100d	19	< 1	n/m	101	100	n/m	104	105	n/m
TheFrozenSea	110	< 1	n/m	3038	3605	n/m	3133	3690	n/m

Table 2: Preprocessing time for road and grid graphs. We give results for (i) the average time required to compute each node ordering; (ii) the total time required to compute the entire database for each of SRC and MRC. Values are given to the nearest second. The ost100d and TheFrozenSea preprocessing experiments were run on an AMD Opteron 6172 with 48 cores @ 2.1 GHz to accelerate the APSP computation. The experiments on the smaller graphs clearly show that the input order is fully dominated. We therefore omit the numbers for the two larger test graphs. "n/m" stands for "not measured".

	Graph		DB Size (MB)								Query Time (nano seconds)						
	$\|V\|$	$\frac{\|E\|}{\|V\|}$	CopaG	MRC			SRC			UM	CopaG	MRC			SRC		
				+cut	+dfs	+inp	+cut	+dfs	+inp			+cut	+dfs	+inp	+cut	+dfs	+inp
Dragon Age: Origins (27 maps)																	
Min	< 1K	7	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	34	19	26	26	14	19	18
Q1	2K	7.2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	2	63	22	31	35	16	22	26
Med	5K	7.4	1	< 1	1	1	< 1	1	2	12.5	81	30	44	54	20	31	38
Avg	31K	7.4	12	5	7	23	6	8	53	480.5	156	34	50	72	25	36	54
Q3	52K	7.6	18	6	10	29	7	12	65	1352	266	36	62	106	28	45	82
Max	100K	7.7	75	31	39	106	35	44	349	5000	316	95	116	176	67	78	138
StarCraft (11 maps)																	
Min	105K	7.7	60	20	35	89	25	42	187	5512.5	304	63	93	130	47	63	88
Q1	173K	7.7	128	28	61	144	33	71	281	14964	324	70	103	142	51	69	102
Med	274K	7.8	183	69	111	393	83	126	956	37538	334	95	121	187	66	77	133
Avg	288K	7.8	351	148	203	444	172	222	983	41472	358	105	130	195	66	82	132
Q3	396K	7.8	510	189	282	621	222	308	1318	78408	396	126	146	226	72	90	156
Max	494K	7.8	934	549	626	1245	630	660	2947	122018	436	197	195	311	108	118	208

Table 3: Performance of SRC and MRC on grid graphs. We use two problem sets taken from the 2012 GPPC and compare against Copa-G, one of the winners of that competition. We measure (i) the size of the compressed database (in MB) and; (ii) the time needed to extract a first query (in nanos). All values are rounded to the nearest whole number (either MB or nano, respectively). As a baseline, column UM shows the size of a naive, non-compressed first-move matrix.

12.1 Preprocessing Time

Table 2 gives the average preprocessing time for SRC and MRC on the 6 road graphs and the two competition sets. The time in each case is dominated by the need to compute a full APSP table. As we have previously commented, APSP compression is the central point of our work; not the APSP-computation. Our preprocessing approach involves executing Dijkstra's algorithm repeatedly resulting in a total running time of $O(n^2 \log n)$ on sparse graphs with non-negative weights; using modern APSP techniques (e.g., Delling, Goldberg, Nowatzyk, and Werneck 2013) succeeded in significantly reducing the hidden constants behind the big-O and are able to exploit more specific graphs structures (e.g., road graphs) to get the running time down. However, these techniques do not give a benefit over repeatedly running Dijkstra's algorithm in the asymptotic worst-case.

Graph			DB Size (MB)								Query Time (nano seconds)						
Name	$\|V\|$	$\frac{\|E\|}{\|V\|}$	Copa -R	Hub Labels RXL	Hub Labels CRXL	MRC +cut	MRC +dfs	SRC +cut	SRC +dfs	UM	Copa -R	Hub Labels RXL	Hub Labels CRXL	MRC +cut	MRC +dfs	SRC +cut	SRC +dfs
BAY-d	321K	2.5	317	90	19	141	129	160	144	51521	527	488	3133	89	100	62	69
BAY-t	321K	2.5	248	65	17	102	95	117	107	51521	469	371	1873	74	87	52	60
COL-d	436K	2.4	586	138	24	228	206	268	240	95048	677	564	3867	125	111	68	85
COL-t	436K	2.4	503	90	22	162	150	192	175	95048	571	390	2131	88	97	58	65
NY-d	264K	2.8	363	99	21	226	207	252	229	34848	617	621	4498	112	122	75	83
NY-t	264K	2.8	342	66	18	192	177	217	198	34848	528	425	2529	98	111	67	75

Table 4: Comparative performance of SRC, MRC, Copa-R and two recent Hub Labeling algorithms. We also report the size UM of an uncompressed matrix. We test each one on six graphs from the 9th DIMACS challenge. We measure (i) database sizes (in MB); (ii) the time needed to extract a first query (in nanos). Values are rounded to the nearest whole number. Graph sizes are rounded to the nearest thousand nodes.

Creating a node order is fast; +dfs requires only fractions of a second. Even the +cut order requires no more than 18 seconds on average using METIS (Karypis & Kumar, 1998). Meanwhile, the difference between the running times of SRC and MRC indicate that multi-row compression does not add more than a small overhead to the total time. For most of our test instances the recorded preprocessing overhead was on the order of seconds.

12.2 Compression and Query Performance

In Table 3 we give an overview of the compression and query time performance for both Copa-G and a range of SRC and MRC variants on the competition benchmark sets. To measure the query performance we run 10^8 random queries with source and target nodes picked uniformly at random and average their running times.

MRC outperforms SRC in terms of compression but at the expense of query time. Node orders significantly impact the performance of SRC and MRC. In most cases +cut yields a smaller database and faster queries. SRC and MRC using +cut and +dfs convincingly outperform Copa-G on the majority of test maps, both in terms of space consumption and query time.

A naive, non-compressed first-move matrix is impractical due to the very large memory requirements. The size an uncompressed matrix would have is $4 \cdot |V|^2$ bits, reflecting our assumption that an outgoing edge can be stored in 4 bits. For Tables 3, 4, and 5 we specify in column UM the memory consumption of an uncompressed matrix. For example, for the ost100d game graph with 137K, a non-compressed matrix requires a bit more than 9GB of memory, which is more than two orders of magnitude higher than the 49MB, respectively 39MB that SRC+cut respectively MRC+cut need. For larger graphs, the difference is more striking. For example TheFrozenSea, our largest game graph, with 754K nodes leads to a 576MB MRC+cut database compared to 284GB in a non-compressed matrix. On the other hand, for the smaller graphs where the matrix would fit in memory, fetching up moves would be extremely fast, being one table lookup per move. For comparison, to fetch one move, our method performs a binary search in a compressed string whose length is no larger, and usually much smaller than $|V|$.

In Table 4 we look at the performance for the 6 road graphs and compare Copa-R, SRC, MRC, RXL and CRXL. The main observations are that on road graphs +dfs leads to smaller CPDs than +cut. Surprisingly, the lower average row lengths do not yield faster query times. Copa-R is dominated by RXL, SRC, and MRC. SRC+cut outperforms all competitors by several factors in terms of

Graph			DB Size (MB)							Query Time (nano seconds)					
Name	$\|V\|$	$\frac{\|E\|}{\|V\|}$	Hub Labels		MRC		SRC		UM	Hub Labels		MRC		SRC	
			RXL	CRXL	+cut	+dfs	+cut	+dfs		RXL	CRXL	+cut	+dfs	+cut	+dfs
ost100d	137K	7.7	62	24	39	50	49	57	9436	598	5501	89	110	58	71
FrozenSea	754K	7.7	429	135	576	634	753	740	284405	814	9411	176	192	104	109

Table 5: Performance of SRC and MRC on large grid graphs from Nathan Sturtevant's extended repository. We compare against the Hub Labeling methods RXL, and CRXL. We also report the size UM of an uncompressed matrix. We run tests on TheFrozenSea, drawn from the game StarCraft, and ost100d, which comes from the game Dragon Age Origins. We measure (i) database sizes (in MB); (ii) the time needed to extract a first query (in nanos). Values are rounded to the nearest whole number. RXL & CRXL exploit that the graphs are undirected while SRC & MRC do not. For directed graphs the space consumption of RXL would double.

Graph	Average Row and Label					
	Length			Space (Bytes)		
	SamPG	SRC		SamPG	SRC	
		+cut	+dfs	+ Plain	+cut	+dfs
BAY-d	51	129	108	816	516	432
BAY-t	34	94	79	544	376	316
COL-d	59	160	131	944	640	524
COL-t	35	114	96	560	456	384
NY-d	70	248	203	1120	992	812
NY-t	44	214	175	704	856	700
FrozenSea	92	260	256	1472	1040	1024
ost100d	80	91	108	1280	364	432

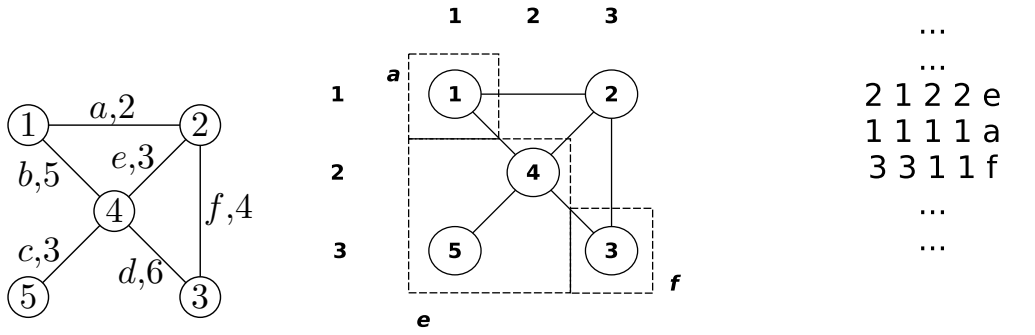
Table 6: We report the average number of hubs per label (length), number of runs per row (length), and average space usage per node for SamPG+Plain and SRC.

speed. RXL wins in terms of the database size. However the factor gained in space is smaller than the factor lost in query time compared to SRC. CRXL clearly wins in terms of space but is up to two orders of magnitude slower than the competition. On road graphs distance weights are harder than travel time weights. This was already known for algorithms that exploit similar graph features as RXL. However, it is interesting that seemingly unrelated first-move compression based algorithms incur the same penalties.

In Table 5 we evaluate the performance of SRC, MRC, RXL and CRXL on the larger game maps. We dropped Copa-R because from the experiments on the smaller graphs it is clear that it is fully dominated. Other than on road graphs, the space consumption for SRC and MRC is lower for the +cut order than for +dfs. As a result the +cut order is clearly superior to +dfs on game maps. On ost100d both SRC and MRC beat RXL in terms of query time and of space consumption. On TheFrozenSea RXL needs less space than SRC and MRC. However, note that on game maps RXL gains a factor of 2 by exploiting that the graphs are undirected which SRC and MRC do not.

CRXL employs powerful compression techniques specific to shortest paths. RXL does not use them but it is not uncompressed as it uses some basic encoding techniques such as delta encoding. A basic HL variant that stores all nodes and distances explicitly needs more memory. We refer to this basic variant as SamPG+Plain. SamPG is the ordering algorithm used in RXL[11] and Plain refers

11. The RXL-paper describes several node orders. However, SamPG is the order they suggest using. All other RXL and CRXL numbers in our paper use SamPG.



(a) Input Graph (b) Computing rectangles (c) List of rectangles

Figure 9: Copa's rectangle decomposition on the toy example from Figures 2 and 8 for the first moves from source node 2. Similar decompositions are needed for every other source node.

to an elementary HL encoding with 4 bytes per hub ID and 4 bytes per distance value. We want to comparse SRC with SamPG+Plain. We therefore report in Table 6 the average number of hubs per label and the average number of runs per row using SRC. The reported number of hubs is per label. Note that on directed graphs every node needs two labels: a forward and a backward label. On undirected graphs the two labels coincide and only one has to be stored. This contrasts with SRC which cannot exploit that the input graph is undirected. The numbers in the table therefore assume that two HL-labels are needed per node for better comparability. For HL we need to store a 32-bit distance value, a 28-bit node ID and a 4-bit out-edge ID and this times 2 because of there being two labels per node. The total space consumption is thus $16h$ bytes where h is the average number of hubs per label. For SRC we need to store a 28-bit node ID and a 4-bit out-edge ID per run. This results in $4r$ bytes where r is the average number of runs per row. For Table 6 it can be seen that SamPG+Plain consistently occupies more space than SRC, even though the experiments thus far suggest that RXL is more compact. The basic compression techniques in RXL are therefore important and are enough to make the performance ordering of the algorithms tip with respect to space consumption.

RXL has some advantages not visible in the tables. For example it does not require computing an APSP in the preprocessing step significantly reducing preprocessing time. Further it computes besides the first move also the shortest path distance.

12.3 Discussion

We compared SRC and MRC with Copa and RXL. Copa is a recent and successful technique for creating compressed path databases. As one of the joint winners at the 2012 Grid-Based Path Planning Competition (GPPC-12) we regard Copa as the current state-of-the-art for a range of pathfinding problems including the efficient storage and extraction of optimal first-moves. RXL is the newest version of the Hub-Labeling algorithm and to our knowledge the state-of-the-art in terms of minimizing query times on road graphs.

SRC and MRC have been illustrated in Figures 1, 2 and 8. Figure 9 illustrates how Copa works in preprocessing, for a better understanding of this section, without any intention of a fully detailed description. Copa assumes that every node in the graph is labelled with x, y coordinates. Our toy example has 3 rows and 3 columns, as shown in Figure 9 (a). Copa iterates through all nodes in the graph. Let n be the current node ("source node") at a given iteration. Copa splits the map into rectangles, and labels each rectangle with the id of an outgoing edge from n. When a target t belongs to a given rectangle, the optimal move from n towards t is precisely the label of that rectangle. Figure 9 (a) shows the map decomposition for the source node 2, with rectangles depicted with a dashed line. There are three rectangles constructed in the figure: one at the bottom left, with size 2×2 and label e; one at the top left, with size 1×1 and label a; and one at the bottom right, having size 1×1 and label f. In part (b), each rectangle is represented with 5 symbols each: upper row, left column, width, height, label. Here we show just the rectangles for source node 2, but this is concatenated with the lists of all other nodes. Some of the rectangles can safely be removed ("list trimming") but we skip this in our example. Then, each of the 5 columns in Figure 9 (a) is treated as a separate string, and it is further compressed with sliding window compression and run-length encoding.

We performed experiments on a large number of realistic grid-graphs used at GPPC-12 and find that both SRC and MRC significantly improve on both the query time and compression power of Copa. For a large number of experiments and on a broad range of input maps we were able to extract a first move in just tens or hundreds of nano-seconds (a factor of 3 to 5 faster than Copa). There are two main reasons why SRC and MRC are performant vs. Copa: Our approach uses less memory and our query running time is logarithmic (cf. linear) in the label size.

Our approach requires less memory than Copa. Part of the explanation stems from the differences between the sizes of the "building blocks" in each approach. In SRC and MRC, the "building block" is an RLE run represented with two numbers: the start of the run, which is a node id and thus requires $\log_2(|V|)$ bits, and the value of the run, which is an out-edge id and requires $\log_2(\Delta)$ bits. In Copa, a building block is a rectangle that requires $2\log_2(|V|) + \log_2(\Delta)$ bits. In the actual implementations, both SRC and MRC store only a single 32-bit machine word per run, which allows for graphs with up to 2^{28} nodes. The Copa code used in the 2012 Grid-Based Path Planning Competition stores a rectangle on 48 bits, corresponding to a max node count of 2^{22}.

Clearly, the size of the building blocks is not the only reason for the different compression results. The number of RLE runs in SRC or MRC can differ from the total number of rectangles in Copa. When more than one optimal out-edge exists, SRC and MRC select an edge that will improve the compression, whereas Copa sticks with one arbitrary optimal out-edge. On the other hand, besides rectangle decomposition, Copa implements additional compression methods, such as list trimming, run length encoding and sliding window compression, all performed on top of the original rectangle decomposition (Botea & Harabor, 2013a).

Our approach has an asymptotic query time of $O(\log_2(k))$ where k is the number of compressed labels that must be searched. By comparison, given a source node, Copa stores its corresponding list of rectangles in the decreasing order of their size. Rectangles are checked in order. While, in the worst-case, the total number of rectangle checks is linear in the size of the list, the average number is much improved due to the ordering mentioned (Botea, 2011; Botea & Harabor, 2013a).

The reason why a CPD is faster than RXL is due to the basic query algorithm. The algorithm underlying RXL consists of a merge-sort like merge of two integer arrays formed by the forward label of s and the backward label of t. This is a fast and cache friendly operation but needs to look

at each entry resulting in an inherently linear time operation. SRC on the other hand builds upon a binary search which is slightly less cache friendly as memory accesses are not sequential it but has a logarithmic running time.

One can regard the compressed SRC rows as one-sided labels. For each st-pair the first move can be determined using only the label of s. HL on the other hand needs the forward label of s and the backward label of t. HL-labels tend to have less entries than the SRC labels. However, each HL-entry needs more space as they need to store the distance values in addition to node-IDs.

13. Results from the 2014 Grid-Based Path Planning Competition

We have recently submitted the algorithms SRC+cut and SRC+dfs to the 2014 edition of the Grid-Based Path Planning Competition GPPC (Sturtevant, 2014). In this section we give a brief overview of the competition and a short summary of results. A full description of the methodology employed by the organisers, as well as a full account of the results, is given in (Sturtevant et al., 2015).

13.1 Competition Setup

The Grid-Based Path Planning Competition features over one hundred grid maps from which more than three hundred thousand distinct problem instances are drawn. Individual maps differ in size, ranging from several thousand to several million nodes. The topography of the maps is also varied with many maps originating from the computer games StarCraft, Dragon Age: Origins and Dragon Age 2. Other maps appearing as part of the competition are synthetically generated grids; mazes, rooms and randomly placed obstacles of varying density. In the 2014 edition of the competition there were a total 14 different entries, submitted by 6 different teams. Several entries are variants of the same algorithm and are submitted by the same team.

- 6 entries employ symmetry breaking to speed up search. The entries BLJPS, BLJPS2, JPS+ and JPS+Bucket can roughly be described as extensions of Jump Point Search (Harabor & Grastien, 2011) and JPS+ (Harabor & Grastien, 2014). The entry NSubgoal makes use of multi-level Subgoal Graphs (Uras & Koenig, 2014). Finally the entry named BLJPS_Sub is a hybrid algorithm that makes use of both Jump Point Search and Subgoal Graphs.
- 1 entry (CH) employs a variation on Contraction Hierarchies (Dibbelt et al., 2014).
- 3 entries directly improve the performance of the A* algorithm; either through the use of faster priority queues (A* Bucket) or by trading optimality for speed (RA* and RA*-Subgoal).
- 4 entries use Compressed Path Databases. Two of these (SRC+dfs-i and SRC+cut-i) are incremental algorithms that return the optimal path one segment at a time; that is, they must be called repeatedly until the target location is returned. The other two algorithms (SRC+dfs and SRC+cut) are non-incremental and when queried return a complete path.

Unfortunately, 3 entries contained some bugs and therefore did not finish on all instances. SRC+cut was one of them. We therefore omit them from the tables and the discussion.

13.2 Results

A summary of results from the competition is given in Table 7. We observe the following:

Entry	Averaged Query Time over All Test Paths (μs)			Preprocessing Requirements	
	Slowest Move in Path	First 20 Moves of path	Full Path Extraction	DB Size (MB)	Time (Minutes)
†**RA***	282 995	282 995	282 995	**0**	**0.0**
BLJPS	14 453	14 453	14 453	20	0.2
JPS+	7 732	7 732	7 732	947	1.0
BLJPS2	7 444	7 444	7 444	47	0.2
†**RA*-Subgoal**	1 688	1 688	1 688	264	0.2
JPS+ Bucket	1 616	1 616	1 616	947	1.0
BLJPS2_Sub	1 571	1 571	1 571	524	0.2
NSubgoal	773	773	773	293	2.6
CH	362	362	362	2 400	968.8
SRC-dfs	145	145	**145**	28 000	11649.5
SRC-dfs-i	**1**	**4**	189	28 000	11649.5

Table 7: Results from the 2014 Grid-Based Path Planning Competition. Figures are summarised from the official competition results, which appear in (Sturtevant et al., 2015). Entries denoted by † indicate approximate algorithms that are not guaranteed to always find the shortest path. The measurments in bold indicate which entry performed best with regard to this single criterion. The entries whose name are in bold are those that are not fully Pareto-dominated with respect to every criterion. The preprocessing running times are the time needed to process all 132 test maps.

1. Our CPD-based entries are the fastest methods at the competition across all query time metrics. This includes the fastest (average) time to required to extract a complete optimal path, the fastest (average) time to extract the first 20 steps of an optimal path and the fastest (average) time required to extract any single step of an optimal path.

2. Performing a first move query with our algorithm is *faster than the resolution of the competition's microsecond timer*. Even iteratively extracting 20 edges of a path is only barely measurable without a finer timer resolution. During testing we observed that a significant amount of the query running time was spent within the benchmarking code provided by the competition. We therefore opted to submit two variants. SRC+dfs extracts the path as a whole. The benchmarking code is thus only run once per path. On the other hand SRC+dfs-i extracts the path one edge at a time. This allows measuring the time needed by an individual first move query. Unfortunately, it also requires executing the benchmarking code once per edge. The difference in path extraction running times between SRC-dfs and SRC-dfs-i (i.e., 44μs) is the time spent in the benchmarking code.

3. Our algorithm is the only competitor that is able to answer first-move queries faster than a full path extraction.

4. Because our entries are database driven they require generous amounts of preprocessing time and storage space. SRC+dfs therefore had the largest total preprocessing time and the largest total storage cost of all entries at the competition.

14. Conclusion and Future Work

We study the problem of creating an efficient compressed path database (CPD): a shortest path oracle which, given two nodes in a weighted directed graph, always returns the first-move of an

optimal path connecting them. Starting with an all-pairs first-move matrix, which we assume is given, we create oracles that are more compact and that can answer arbitrary first-move queries optimally and many orders faster than is otherwise possible using conventional online graph search techniques. We employ a run-length encoding (RLE) compression scheme throughout and analyse the problem from both a theoretical perspective and an empirical one. Our main idea is simple: we look to re-order the nodes (i.e., columns) of the input first-move matrix in a "good" way such that its RLE-compressed size is subsequently reduced.

On the theoretical side we show that the problem of finding an optimal node ordering, in the general case for both directed and directed graphs, is NP-complete. In more specific cases, such as graphs that can be decomposed along articulation points, the same problem can be efficiently tackled by solving a series of independent sub-problems. In particular we show that a depth-first traversal of a tree provides an optimal node ordering. These results give a first theoretical underpinning to the problem of creating space-efficient CPDs using RLE. Our work also extends theoretical results from the areas of information processing and databases (Oswald & Reinelt, 2009; Mohapatra, 2009).

On the empirical side we study the efficacy of three heuristic node orderings: (i) a depth-first ordering; (ii) a graph-cut ordering based on balanced edge separators; (iii) a naive baseline given by the ordering specified by the input graph. Given an ordering and a first-move matrix, we describe two novel approaches for creating CPDs. The first of these, SRC, uses simple run-length encoding to compress the individual rows of the matrix. The second approach, MRC, is more sophisticated and identifies commonalities between sets of labels compressed by SRC.

In a range of experiments we show that SRC and MRC can compress the APSP matrix for graphs with hundreds of thousands of nodes in as little as 1-200MB. Associated query times regularly require less than 100 nanoseconds. We also compare our approaches with Copa (Botea, 2012; Botea & Harabor, 2013a), and RXL (Delling et al., 2014). In a range of experiments on both grid and road graphs we show that SRC and MRC are not only competitive with Copa but often several factors better, both in terms of compression and query times. We also show that SRC and MRC outperform RXL in terms of query time. We also summarise results from the 2014 Grid-Based Path Planning Competition. In particular we can report that SRC was the fastest method at the competition across all query-time metrics and that SRC performed better than the resolution of the competition's microsecond timer.

There appear several promising directions in which the current work could be extended. One immediate possibility is to harness the available results on efficient appproximate TSP algorithms in order to compute better and more space-efficient node orderings. Another immediate possibility is to improve the current MRC compression scheme by devising an algorithm that optimizes the assignment of first-move IDs.

Looking more broadly, a strength that all CPDs have, in addition to fast move extraction, is that they can compress any kind of path – not just those that are network-distance optimal.[12] In multi-agent pathfinding for example it is sometimes useful to guarantee properties like "there must always be a local detour available" (Wang & Botea, 2011). Another example are turn-costs in road graphs. Thus one possible a possible direction for future work is to create CPDs that store only paths satisfying such constraints.

A weakness of our approach is that in the preprocessing an APSP-computation is required. While Delling et al. (2013) have shown that an APSP can sometimes be computed reasonably fast

12. Suboptimal paths, however, introduce the additional challenge of avoiding infinite loops when extracting such a path from a CPD.

on graphs with many nodes, APSP remains inherently quadratic in the number of nodes on any graph class as its output size is quadratic. Our approach would therefore hugely profit from an algorithm that can directly compute the compressed CPD without first computing the first-move matrix in an intermediate step.

15. Acknowledgments

We thank Patrik Haslum, Akihiro Kishimoto, Jakub Marecek, Anika Schumman and Jussi Rintanen for feedback on earlier versions of parts of this work. We would like to thank Daniel Delling & Thomas Pajor for running the Hub-Labeling experiments for us.

Appendix A. Proof to Theorem 2

Theorem 2. SimMini1Runs is NP-complete.

Proof. The membership to NP is straightforward. The hardness proof uses a reduction from the Hamiltonian Path Problem (HPP) in an undirected graph. Let $G = (V, E)$ be an arbitrary undirected graph, without duplicate edges, and define $n = |V|$ and $e = |E|$. Figure 10 shows a toy graph used as a running example.

Starting from G, we build a SimMini1Runs instance as follows. We define a 0/1 matrix $\mathbf{m}$ with e rows and n columns. Let r be the row corresponding to an edge (u, v), and let c_u and c_v be the columns associated with nodes u and v. Then $\mathbf{m}[r, c_v] = \mathbf{m}[r, c_u] = 1$ and $\mathbf{m}[r, c] = 0$ for all other columns. Notice that $\mathbf{m}$ has at least a value of 1 on every row and column. Figure 11 shows the matrix in the running example.

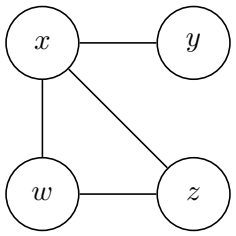


Figure 10: Sample graph G.

Let r be the matrix row corresponding to an edge (u, v). It is easy to see that, when a given ordering of the columns (nodes) makes the two nodes u and v adjacent, the number of sequential

	x	y	w	z
(x, y)	1	1	0	0
(x, w)	1	0	1	0
(x, z)	1	0	0	1
(w, z)	0	0	1	1

Figure 11: Matrix $\mathbf{m}$ built for G.

	y	x	w	z
(x, z)	0	**0**	0	1
(w, z)	0	0	1	1
(x, w)	0	1	1	0
(x, y)	1	1	0	0

Figure 12: The matrix after: i) converting some 1s into 0s (shown in bold); ii) re-ordering columns on the Hamiltonian path; and iii) re-ordering rows lexicographically.

	y	x	w	z
(x, z)	0	**1**	0	1
(w, z)	0	0	1	1
(x, w)	0	1	1	0
(x, y)	1	1	0	0

Figure 13: Matrix after restoring back the previously replaced 1s (shown in bold).

RLE 1-runs[13] for row r is 1. If the nodes are not adjacent, the number of sequential RLE 1-runs for row r is 2.

We claim the HPP has a solution iff SimMini1Runs has a solution with $t = 3e - n + 2$ RLE 1-runs. Let $v_{i_1}, \ldots, v_{i_n}$ be a solution of HPP (i.e., a Hamiltonian path), and let P be the set of all edges included in this solution. In the running example, let P contain (y, x), (x, w) and (w, z).

In every row corresponding to an edge not contained in P, switch one of its two 1-entries to 0. Then, order the columns with respect to the sequence of the nodes in the Hamiltonian path and rearrange the rows in lexicographical order. Figure 12 illustrates these changes.

The construction of the matrix, the trick of converting some of the 1s into 0s, and the ordering of the rows and columns are reused from Oswald and Reinelt's proof for the hardness of deciding whether a 0/1 matrix has the $C1S_k$ property (Oswald & Reinelt, 2009). The rest of the proof, coming below, is significantly different.

Now, restore the previously replaced 1s, as shown in Figure 13. Each of the $e - n + 1$ 1s that were replaced and restored have no adjacent 1s in the matrix.[14] As such, each of them counts as two 1-runs, one horizontal and one vertical. This sums up to a total of $2(e-n+1)$ 1-runs corresponding to the 1s replaced and restored. In addition, each row and column has one more 1-run. It follows that the matrix has $3e - n + 2$ 1-runs.

Conversely, consider a row and column ordering that creates $3e-n+2$ RLE 1-runs in total. We show that the matrix has at least $e + 1$ vertical 1-runs, regardless of the row ordering. Consider the rows, in order, starting from the top. The first row introduces exactly 2 vertical 1-runs, one for each column where it contains a value of 1. Each subsequent row introduces at least one more vertical 1-run. Otherwise, the new row would be identical to the previous one, which contradicts the fact that the graph has no duplicate edges.

13. All runs used in this proof are sequential.
14. If they had an adjacent 1 on a column, this would imply that we have two identical rows, which would further mean that G has duplicate edges. If they had an adjacent 1 on a row, it would mean that the edge at hand belongs to the Hamiltonian path, which contradicts the fact that 1s were replaced and restored on the complementary set of edges.

As there are at least $e + 1$ vertical 1-runs, the number of horizontal 1-runs will be at most $(3e - n + 2) - (e + 1) = 2e - n + 1$. We show that the column ordering is a Hamiltonian path. Assuming the contrary, there are only $p < n - 1$ edges such that their nodes are adjacent in the ordering. It follows that the number of horizontal 1-runs is $p + 2(e - p) = 2e - p > 2e - n + 1$. Contradiction. □

Appendix B. Proofs to Lemma 2 and Lemma 3

We start by pointing out two simple but important properties stemming from the notion of an articulation point:

Remark 1. Given a graph G, let x be an articulation point, and let $G_1 \dots G_n$ be the corresponding connected components obtained after removing x.

1. Given a source node $s \in G_i$, the first optimal move from s towards anywhere outside G_i is the same.[15]
2. Given two distinct components G_i and G_j, the first optimal move from x towards anywhere in G_i is different from the first optimal move from x towards anywhere in G_j.

Remark 1 follows easily from the obvious observation that there is no way of going from one subgraph G_i to another subgraph G_j other than passing through x. See Figure 7 for an illustration.

Lemma 2. Let x be an articulation point of a graph G. Every node order o can be rearranged into an x-block ordering o' without increasing the number of runs on any row.

Proof. We construct the desired ordering o' by applying the following steps:

1. Rotate o so that x comes first.
2. For every $i \in \{1 \dots n\}$ project the resulting order onto G_i, obtaining a suborder o'_i.
3. Define o' as: $o' = x, o'_1, \dots, o'_n$.

It is clear by construction that the nodes in every subgraph G_i are consecutive in o'. It remains to show that the number of runs per row does not grow.

Denote by s a source node. We distinguish between two cases:

- Case when $s \in G_i$ for some i. Being just a rotation, step 1 has no impact on the number of cyclic runs. Steps 2 and 3 take all nodes from $\bigcup_{k \neq i} G_k$ and put them into one or two blocks that are cyclic adjacent to x. We know from Remark 1, point 1 that $\mathbf{m}[s, x] = \mathbf{m}[s, n]$ for all nodes n in $\bigcup_{k \neq i} G_k$. Thus, the re-arrangement brings next to x nodes n with the same first-move symbol as x. Clearly, this does not increase the number of runs.
- Case when $s = x$. As in the previous case, the rotation performed at step 1 does not increase the number of cyclic runs. After step 1, cyclic runs and sequential runs are equivalent, since the first position contains a distinct symbol, namely the x-singleton. Steps 2 and 3 separate

15. Assuming that we split the ties among optimal paths in a consistent manner, which is easy to ensure.

each G_i into a contiguous block. This does not increase the number of sequential runs since, according to Remark 1, point 2, every two blocks corresponding to G_i and G_j, with $i \neq j$, have no common symbol. It follows that the number of cyclic runs does not increase either.

□

Lemma 3. Given any x-block ordering o, we have that:

1. $N(o, G, G_i) = N(o|_{\Gamma_i}, \Gamma_i, G_i)$; and
2. $N(o, G, \{x\}) = 1 - n + \sum_i N(o|_{\Gamma_i}, \Gamma_i, \{x\})$; and
3. $N(o, G, G) = 1 - n + \sum_i N(o|_{\Gamma_i}, \Gamma_i, \Gamma_i)$.

Proof. We prove each point as follows.

1. As o is an x-block ordering, the nodes come in the order $x, G_1, \ldots G_i, \ldots G_n$. Consider a node $s \in G_i$ and its corresponding row in the first-move matrix. As pointed out in Remark 1, every path from s to a node outside G_i has to pass through x, and therefore the first move from s to anywhere outside G_i is the same. It follows that all nodes in the sequence $x, G_1, \ldots G_{i-1}$, together with all nodes in the sequence $G_{i+1}, \ldots G_n$, form one cyclic run. In effect, removing from consideration all nodes contained in $\bigcup_{k \neq i} G_k$ leaves the number of runs unchanged, which completes the proof of this case.

2. This is the case focused on x as a start node. According to Remark 1, if $G_i \neq G_j$, then the first move from x towards anywhere in G_i is different from the first move from x towards anywhere in G_j. It follows that two runs from two adjacent subsets G_i and G_{i+1} never merge into one run. Thus,

$$\begin{aligned} N(o, G, \{x\}) &= 1 + \sum_i (N(o|_{\Gamma_i}, \Gamma_i, \{x\}) - 1) \\ &= 1 - n + \sum_i N(o|_{\Gamma_i}, \Gamma_i, \{x\}). \end{aligned}$$

3. This case follows from the previous two, with standard arithmetic manipulation.

$$\begin{aligned} N(o, G, G) &= N(o, G, \{x\}) + \sum_i N(o, G, G_i) \\ &= 1 - n + \sum_i N(o|_{\Gamma_i}, \Gamma_i, \{x\}) + \sum_i N(o|_{\Gamma_i}, \Gamma_i, G_i) \\ &= 1 - n + \sum_i (N(o|_{\Gamma_i}, \Gamma_i, \{x\}) + N(o|_{\Gamma_i}, \Gamma_i, G_i)) \\ &= 1 - n + \sum_i (N(o|_{\Gamma_i}, \Gamma_i, \{x\} \cup G_i) \\ &= 1 - n + \sum_i N(o|_{\Gamma_i}, \Gamma_i, \Gamma_i). \end{aligned}$$

□

References

Abraham, I., Delling, D., Fiat, A., Goldberg, A. V., & Werneck, R. F. (2012). HLDB: Location-based services in databases. In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, pp. 339–348. ACM Press. Best Paper Award.

Abraham, I., Delling, D., Goldberg, A. V., & Werneck, R. F. (2011). A hub-based labeling algorithm for shortest paths on road networks. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, Vol. 6630 of *Lecture Notes in Computer Science*, pp. 230–241. Springer.

Abraham, I., Delling, D., Goldberg, A. V., & Werneck, R. F. (2012). Hierarchical hub labelings for shortest paths. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, Vol. 7501 of *Lecture Notes in Computer Science*, pp. 24–35. Springer.

Akiba, T., Iwata, Y., & Yoshida, Y. (2013). Fast exact shortest-path distance queries on large networks by pruned landmark labeling.. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, pp. 349–360. ACM Press.

Antsfeld, L., Harabor, D., Kilby, P., & Walsh, T. (2012). Transit routing on video game maps.. In *AIIDE*.

Arz, J., Luxen, D., & Sanders, P. (2013). Transit node routing reconsidered. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, Vol. 7933 of *Lecture Notes in Computer Science*, pp. 55–66. Springer.

Babenko, M., Goldberg, A. V., Kaplan, H., Savchenko, R., & Weller, M. (2015). On the complexity of hub labeling. In *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS'15)*, Lecture Notes in Computer Science. Springer.

Baier, J., Botea, A., Harabor, D., & Hernández, C. (2014). A fast algorithm for catching a prey quickly in known and partially known game maps. *Computational Intelligence and AI in Games, IEEE Transactions on*, *PP*(99).

Bast, H., Delling, D., Goldberg, A. V., Müller–Hannemann, M., Pajor, T., Sanders, P., Wagner, D., & Werneck, R. F. (2015). Route planning in transportation networks. Tech. rep. abs/1504.05140, ArXiv e-prints.

Bast, H., Funke, S., & Matijevic, D. (2009). Ultrafast shortest-path queries via transit nodes. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, Vol. 74 of *DIMACS Book*, pp. 175–192. American Mathematical Society.

Bast, H., Funke, S., Matijevic, D., Sanders, P., & Schultes, D. (2007). In transit to constant shortest-path queries in road networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pp. 46–59. SIAM.

Bauer, R., Columbus, T., Katz, B., Krug, M., & Wagner, D. (2010). Preprocessing speed-up techniques is hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*, Vol. 6078 of *Lecture Notes in Computer Science*, pp. 359–370. Springer.

Bauer, R., Columbus, T., Rutter, I., & Wagner, D. (2013). Search-space size in contraction hierarchies. In *Proceedings of the 40th International Colloquium on Automata, Languages, and*

Programming (ICALP'13), Vol. 7965 of *Lecture Notes in Computer Science*, pp. 93–104. Springer.

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, *16*, 87–90.

Botea, A. (2011). Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'11)*, pp. 122–127. AAAI Press.

Botea, A. (2012). Fast, optimal pathfinding with compressed path databases. In *Proceedings of the Symposium on Combinatorial Search (SoCS'12)*.

Botea, A., Baier, J. A., Harabor, D., & Hernández, C. (2013). Moving target search with compressed path databases. In *Proceedings of the International Conference on Automated Planning and Scheduling ICAPS*.

Botea, A., & Harabor, D. (2013a). Path planning with compressed all-pairs shortest paths data. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling*. AAAI Press.

Botea, A., & Harabor, D. (2013b). Path planning with compressed all-pairs shortest paths data. In *Proceedings of the International Conference on Automated Planning and Scheduling ICAPS*.

Botea, A., Strasser, B., & Harabor, D. (2015). Complexity Results for Compressing Optimal Paths. In *Proceedings of the National Conference on AI (AAAI'15)*.

Bulitko, V., Björnsson, Y., & Lawrence, R. (2010). Case-based subgoaling in real-time heuristic search for video game pathfinding. *J. Artif. Intell. Res. (JAIR)*, *39*, 269–300.

Bulitko, V., Rayner, D. C., & Lawrence, R. (2012). On case base formation in real-time heuristic search. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-12, Stanford, California, October 8-12, 2012*.

Cohen, E., Halperin, E., Kaplan, H., & Zwick, U. (2002). Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pp. 937–946, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Delling, D., Goldberg, A. V., Nowatzyk, A., & Werneck, R. F. (2013). PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, *73*(7), 940–952.

Delling, D., Goldberg, A. V., Pajor, T., & Werneck, R. F. (2014). Robust distance queries on massive networks. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, Vol. 8737 of *Lecture Notes in Computer Science*, pp. 321–333. Springer.

Delling, D., Goldberg, A. V., & Werneck, R. F. (2013). Hub label compression. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, Vol. 7933 of *Lecture Notes in Computer Science*, pp. 18–29. Springer.

Demetrescu, C., Goldberg, A. V., & Johnson, D. S. (Eds.). (2009). *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, Vol. 74 of *DIMACS Book*. American Mathematical Society.

Dibbelt, J., Strasser, B., & Wagner, D. (2014). Customizable contraction hierarchies. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*, Vol. 8504 of *Lecture Notes in Computer Science*, pp. 271–282. Springer.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Felner, A., Korf, R. E., Meshulam, R., & Holte, R. C. (2007). Compressed pattern databases.. *J. Artif. Intell. Res. (JAIR)*, *30*, 213–247.

Ford, Jr., L. R. (1956). Network flow theory. Tech. rep. P-923, Rand Corporation, Santa Monica, California.

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms (WEA'08)*, pp. 319–333.

Harabor, D. D., & Grastien, A. (2011). Online graph pruning for pathfinding on grid maps. In Burgard, W., & Roth, D. (Eds.), *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press.

Harabor, D. D., & Grastien, A. (2014). Improving jump point search. In Chien, S., Do, M. B., Fern, A., & Ruml, W. (Eds.), *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. AAAI.

Hart, P. E., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*, 100–107.

Hernández, C., & Baier, J. A. (2011). Fast subgoaling for pathfinding via real-time search.. In *Proceedings of the International Conference on Automated Planning and Scheduling ICAPS-11*.

Karypis, G., & Kumar, V. (1998). Metis, a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0..

Kou, L. T. (1977). Polynomial complete consecutive information retrieval problems. *SIAM Journal on Computing*, *6*(1), 67–75.

Lawrence, R., & Bulitko, V. (2013). Database-driven real-time heuristic search in video-game pathfinding. *Computational Intelligence and AI in Games, IEEE Transactions on*, *5*(3), 227–241.

Mohapatra, A. (2009). Optimal Sort Ordering in Column Stores is NP-Complete. Tech. rep., Stanford University.

Oswald, M., & Reinelt, G. (2009). The simultaneous consecutive ones problem. *Theoretical Computer Science*, *410*(21-23), 1986–1992.

Samadi, M., Siabani, M., Felner, A., & Holte, R. (2008). Compressing pattern databases with learning. In *Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, pp. 495–499, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Sankaranarayanan, J., Alborzi, H., & Samet, H. (2005). Efficient query processing on spatial networks. In *Proceedings of the 13th Annual ACM International Workshop on Geographic Information Systems (GIS'05)*, pp. 200–209.

Strasser, B., Harabor, D., & Botea, A. (2014). Fast First-Move Queries through Run Length Encoding. In *Proceedings of the Symposium on Combinatorial Search (SoCS'14)*.

Sturtevant, N. (2012a). 2012 Grid-Based Path Planning Competition. `https://code.google.com/p/gppc-2012/`.

Sturtevant, N. (2012b). The Website of the Grid-Based Path Planning Competition 2012. `http://movingai.com/GPPC/`.

Sturtevant, N. (2014). The Website of the Grid-Based Path Planning Competition 2014. `http://movingai.com/GPPC/`.

Sturtevant, N., Traish, J., Tulip, J., Uras, T., Koenig, S., Strasser, B., Botea, A., Harabor, D., & Rabin, S. (2015). The grid-based path planning competition: 2014 entries and results. In *Proceedings of the 6th International Symposium on Combinatorial Search (SoCS'15)*. AAAI Press.

Uras, T., & Koenig, S. (2014). Identifying hierarchies for fast optimal search. In Brodley, C. E., & Stone, P. (Eds.), *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pp. 878–884. AAAI Press.

Uras, T., Koenig, S., & Hernández, C. (2013). Subgoal graphs for optimal pathfinding in eight-neighbor grids.. In *Proceedings of the International Conference on Automated Planning and Scheduling ICAPS-13*.

van Schaik, S. J., & de Moor, O. (2011). A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pp. 913–924, New York, NY, USA. ACM.

Wang, K.-H. C., & Botea, A. (2011). MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *Journal of Artificial Intelligence Research (JAIR)*, *42*, 55–90.