Project Number 288094

eCOMPASS

eCO-friendly urban Multi-modal route PlAnning Services for mobile uSers

STREP
Funded by EC, INFSO-G4(ICT for Transport) under FP7

eCOMPASS – TR – 061

Fast First-Move Queries through Run-Length Encoding

Ben Strasser, Daniel Harabor, Adi Botea

July 2014

Fast First-Move Queries through Run-Length Encoding *

Ben Strasser
Karlsruhe Institute of Technology
Germany

Daniel Harabor
NICTA
Australia

Adi Botea
IBM Research
Ireland

Abstract

We introduce a novel preprocessing-based algorithm to solve the problem of determining the first arc of a shortest path in sparse graphs. 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.

Introduction

A compressed path database (or CPD) is a shortest path oracle. Given a weighted directed input graph $G = (V, E)$ and a pair of vertices $s, t \in V$ (resp. the source and target nodes) a CPD is able to return the first edge (i.e. the first move) on the preferred path from s to t. Our algorithm works in two phases: (i) a preprocessing phase at which only the graph is known; (ii) a query phase that has s and t as input and should output the first arc on a shortest st-path. The preprocessing phase may use a lot of computing time and generates data (i.e. the CPD) that is accessible in the query phase and used to accelerate the first move computation. The motivation for this setup is that in many scenarios the graph is static but many shortest path queries are executed on the same graph. CPDs have been used in a variety of contexts including fast path finding for video games (Botea 2011; 2012), path finding on road networks (Botea and Harabor 2013) and for improving hunter performance when chasing a mobile prey (Botea et al. 2013). Consider for example games on a map with obstacles. Units need to navigate on the map and maneuver around these obstacles to get to their destination. Note that it is not necessary to extract the whole path at the moment that the unit starts moving. For each unit it is sufficient to know the next edge to take and once it reached the next node a new query can be performed. This approach balances the computational load over time reducing the lag spike incurred when many units start moving at the same time. It is often unclear whether a unit will reach its destination. For example the player may change his orders midway resulting in a new destination. In this situation, the approach of extracting first moves when needed avoids discarding a computed path. Another scenario where the destination rapidly changes is moving-target search (Ishida and Korf 1991). Suppose that the hunter unit knows the exact position of the prey-unit. At each step the hunter can use the prey's position as target. As our queries are independent it does not matter if in the next step the prey will already have moved or not. The computational costs to navigate to a moving target are the same as for a static target.

Another scenario in which fast shortest path computations are needed is to plan routes in road networks. For example an online navigation service (for example Google, Bing, OpenStreet, Apple Maps) have many visitors that run queries on the same static road graph. A vast amount of techniques tailored for this specific scenario has been developed in the last decade. We refer the interested reader to a recent survey article (Bast et al. 2014). The usual approach taken on roads differs from the one presented in this paper. The setup includes the same two phases (i.e., preprocessing and online querying) but the query phase should compute the distance of a shortest st-path instead of a first move. Determining a shortest path is viewed as extension of the basic distance algorithm and is usually not performed by computing independent queries per arc. Most of the algorithms compute some form of coarsened path along the distance computation and unpack coarsened subpaths as needed. Unpacking is usually significantly faster than performing an independent query for each edge.

The simplest encoding for a CPD is an all-pairs matrix where each entry represents the id of the first arc on the preferred path between a pair of nodes. Computing such a matrix is a challenging research problem in its own right and the topic of many papers from the literature (e.g. (Knopp et al. 2007; Planken, de Weerdt, and van der Krogt 2011; Delling et al. 2013)). Once the matrix is computed, storage quickly becomes the most immediate practical consideration, as the size of the re-

*Partial support by DFG grant WA654/16-2 and EU grant 288094 (eCOMPASS) and Google Focused Research Award.

sulting data structure is quadratic in the size of the input graph. Efficient random access to individual entries is another another important consideration as the entirety of the matrix may not fit into RAM.

In this paper we study approaches for compactly encoding such APSP matrices in order to derive efficient and practical CPDs. Our main result is therefore about compression, not computation. The main idea of our algorithm is simple: we compute an order for the nodes in the input graph such that nodes which are located in close proximity 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 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 the compressed result. As part of our contribution we undertake a detailed empirical analysis including comparisons of our techniques with Copa and Hub-Labeling. Copa is a recent and very fast CPD oracle which was among the joint winners at the 2012 Grid-based Path Planning Competition. Using a variety of benchmarks from the competition we show that our approaches improve on Copa, both in terms of storage and query time. Hub-Labeling is a technique developed to speedup queries on roads but has been shown to work on different types of input. Hub-Labeling is to the best of our knowledge the fastest technique known on roads.

Related Work

Many techniques from the literature can be employed in order to compute the first arc on an optimal path. Standard examples include optimal graph search techniques such A* and Dijkstra's Algorithm. Significant improvements over these methods can be achieved by preprocessing the input graph. A common approach consists of adding online pruning rules to Dijkstra's Algorithms, 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 a recent survey article (Bast et al. 2014).

SILC (Sankaranarayanan, Alborzi, and Samet 2005) and Copa (Botea and Harabor 2013) are compressed path databases with similarities to our work. Like our work both SILC and Copa can be described as oracles for spatial networks. Given a start and target vertex from the input graph, both algorithms return the first arc on the optimal path. SILC employs a recursive quad-tree mechanism for compression while Copa uses a simpler and more effective (Botea 2011) decomposition based on rectangles. As one of the joint winners of the 2012 Grid-based Path Planning competition, we regard Copa as the current state-of-the-art for efficiently encoding path databases. We compare against Copa in the experimental evaluation.

Hub Labels (HL) were initially introduced by (Cohen et al. 2003). However, for several years there was only little research on the topic until Abraham et al. (2011) discovered that road graphs have tiny hub labels compared to their number of nodes. Since then there has been a multitude of papers on the topic (Abraham et al. 2012b; Delling, Goldberg, and Werneck 2013; Abraham et al. 2012a; Akiba, Iwata, and Yoshida 2013). We compare against the most recent paper (Delling et al. 2014). The algorithm to compute the labels is called SamPG. These labels are then compressed using two schemes: RXL and CRXL. Rather than storing a compressed database of first-moves, HL stores a database of distances. In particular, for each node there is a forward and backward label containing 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 meeting hub can be determined using a simultaneous scan over both labels. As hinted in (Abraham et al. 2012a) it is easy to extend hub labels to solve first move queries. To achieve this, the entries in the forward and backward labels are extended with a third component: a first move arc ID. If h is a forward hub of s then the corresponding entry is extended using the first arc ID of a shortest sh-path. If h is backward hub of t then the entry is extended with the first arc 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 arc 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 the distance values are needed even if one only wishes to compute first-moves.

Another speedup technique with low average query times is Transit Node Routing (TNR) (Bast, Funke, and Matijevic 2009; Bast et al. 2007; Antsfeld et al. 2012). However, two independent studies (Abraham et al. 2011; Arz, Luxen, and 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. In the motivating moving target scenario the prey-unit is often close to the hunter-unit, thus making the technique ineffective.

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 and de Moor 2011) is one such example. Like our work the authors precompute an APSP 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-arc.

Preliminaries

We consider sparse directed graphs $G = (V, A)$ consisting of the *node set* V and the *arc set* $A \subseteq V \times V$. Every arc (u, v) has a *tail* u and a *head* v. All graphs have positive *weights* w, no reflexive-arcs and no multi-arcs. A *node order* $o : V \to [1, |V|]$ assigns to every node v a unique *node ID* $o(v)$. We denote by $\deg(u)$ the number of outgoing arcs of u. The maximum out-degree is denoted by Δ. The out-arcs (u, v) of every node u are ordered in an arbitrary but fixed order and their position is referred to as their *out-arc ID*. We assume that node IDs can be encoded within 28-bit integers. Further we assume throughout the paper that $\Delta \leq 15$ which allows us to encode an out-arc ID within a 4-bit integer and allows us to use 15 to indicate an invalid arc. Note that the concatenation of a node ID and an out-arc ID fits into a single 32-bit machine word.

As mentioned in the introduction, we build a $|V| \times |V|$ all-pairs matrix $\mathbf{m}$. Given a certain ordering of the rows and the columns, an entry $\mathbf{m}[i, j]$ is the id of an outgoing arc from node i contained in an optimal path from i to j. The rows of the matrix are compressed with RLE as discussed later.

Node Order

The node order is essential to the performance of our algorithm. Informally, we need an ordering that assigns close IDs to close nodes. Notice that the Bandwidth problem, which is one way of formalizing this fuzzy criterion, is NP-complete (Garey and Johnson 1979). We present two heuristics to compute a node ordering.

Depth First Search Node Order We can traverse graphs using a depth first search and order the nodes in the order that they are first reached (i.e. a preorder traversal). This order has the property that for many nodes v the parent node of v will have ID $o(i) - 1$ and the first child node of v will have the ID $o(i) + 1$. This motivates the proposed ordering: Start at a random node and perform a depth first search. Assign IDs to the nodes in the order that they are visited. If after the search unvisited nodes remain then perform another depth first search from a random unvisited node scanning only unvisited nodes. Repeat until all nodes are visited and have a unique ID. If the graph is sufficiently sparse then many nodes are a first child and therefore many neighboring nodes will have neighborng IDs.

Cut Node Order The required ordering property 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 $\ell(v) - h(v)$.

Compression

Let $a_1 \dots a_n$ denote an uncompressed row. Every row corresponds to a source node s and every entry a_t is the first arc of a shortest st-path (or an invalid value if there is no path). We refer to a sequence of identical a_i as a *run*, to the value of the a_i in a run as the *run's value* and to the first node ID of the run as the *run's start*. Instead of storing the uncompressed row we store a list of runs ordered by their start. Each run is represented by its start and its value. 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.

Memory Consumption Recall that we required node IDs to be encodable in 28 bits and out-arc 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.

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 the next paragraph. However, depending on the graph it is possible that shortest paths are not unique and may differ in their first arc. 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 $a_1 \ldots a_n$ we modify it to compute sets $A_1 \ldots A_n$ of valid first move arcs. We require that for each $a_t \in A_t$ a shortest st-path must exist that uses a_t as its first arc. Our algorithm maintains alongside the tentative distance array $d(t)$ for each node t a set of valid first move arcs A_t. If the algorithm relaxes an arc (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.

Compressing Rows with Run Length Encoding For every target the compression method is given a set of valid first move arcs and may pick the one that minimizes the compressed size. We formalize this subproblem as following: Given a sequence of sets $A_1 \ldots A_n$ find a sequence $a_1 \ldots a_n$ 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 $A_1 \ldots A_i A_{i+1}$ until the intersection is empty, i.e., $\bigcap_{j\in[1,i]} A_j \neq \emptyset$ 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 $a_1 \ldots a_i$. 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.

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 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 R_s the set of runs in the row of s. Instead of storing for each row s the whole set R_s we store for each group h the intersection of all rows, i.e., we store $R'_h = \bigcap_{i\in h} R_i$. For each row s we store $R'_s = R_s \setminus R_{g(s)}$. Recall that a query with target t consists of finding $\max\{x \in R_s \mid x < t'\}$ (where $t' = 15t + 16$). Notice that this formula can be rewritten using basic set logic as $\max\{\max\{x \in R'_s \mid x < t'\}, \max\{x \in R'_{g(s)} \mid x < t'\}\}$ which can be implemented using two binary searches if all R'_i are stored as ordered arrays. Note that we need a second index array to lookup the R'_g for the groups g.

Computing Row Groups By design close source nodes have close node IDs and thus neighbouring rows. This motivates restricting ourselves to *row-run groupings*, i.e., 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 $S(n)$ maximum number of runs saved compared to using no group-compression restricted to the first n rows. Notice that $S(1) = 0$. Given $S(1) \ldots S(n)$ 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 $S(n + 1 - \ell) + (\ell - 1) \cdot \bigcap_{i\in[n+1-\ell,n+1]} R_i$ 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.

Queries

Given a source node s and a target node t (with $s \neq t$) the algorithm determines the first arc 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-arc 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.

Experimental Setup

To evaluate our work we consider two types of graphs: road graphs and grid-based graphs. In the case of grid graphs we have chosen two benchmark problem sets drawn from real computer games and which have appeared in the 2012 Grid-based Path Planning Competition. In addition to the maps used in the competition we also pick the two largest maps (in terms of nodes) for both games available at (Sturtevant 2012). Note that these two maps were not used in the competition. All maps in each benchmark set are undirected and feature two types of arcs: straight arcs which have a weight of 1.0 and diagonal arcs which have a weight of $\sqrt{2}$.

27 maps come from the game `Dragon Age Origins`. These maps have 16K nodes and 119K edges, on average. The largest map called `ost100d` with 137K nodes and 1.1M edges is evaluated separatately. 11 maps come from the game `StarCraft`. These maps have 288K nodes and 2.24M edges, on average. The largest map called `FrozenSea` and has 754K nodes and 5.8M edges is evaluated separatately.

In the case of road graphs we have chosen several benchmarks made available during the 9th DIMACS challenge (Demetrescu, Goldberg, and Johnson 2009). The `New York City` map (henceforth, NY) has 264K nodes and 730K arcs. The `San Francisco Bay Area` (henceforth, BAY) has 321K nodes and 800K arcs. Finally, `the State of Colorado` (COL) has 436K nodes and 1M arcs. For all three graphs travel time weights (denoted using a `-t` suffix) and geographic distance weights (denoted using `-d`) are available.

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.

Results

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 of these approaches with Copa and RXL. Note that there are 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 authors' original C++ implementation which is available from the competition repository.[1] The second variant, which we denote Copa-R, is optimised for road graphs. This algorithm is described in (Botea and Harabor 2013); we used the authors' original C++ implementation.

RXL is the newest version of the Hub-Labeling algorithm. We asked the original authors to run the experiments for us presented below. As a result the RXL experiments were run on a different machine. They were run on a Xeon E5-2690 @ 2.90 GHz. The reported query times are therefore scaled by a factor of 2.90/3.40 = 85% to adjust for the lower clock speed. Note, however that their implementation computes path distances instead of first-moves. As detailed in the related work section, 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 into the compression schemes presented in (Delling et al. 2014). The reported RXL-sizes are therefore only (probably very tight) lower bounds.

Avg. Preprocessing Time (seconds)			
	Order Time	SRC Time	MRC Time
Dragon Age 2			
+cut	2	32	33
+dfs	<1	35	36
+input	0	38	40
DIMACS			
+cut	16	1950	1953
+dfs	<1	1982	1985
+input	0	2111	2125
StarCraft			
+cut	18	1979	1993
+dfs	<1	2181	2195
+input	0	2539	2574

Table 1: Preprocessing time on each of our four benchmarks. 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.

Preprocessing Time Table 1 gives the average preprocessing time for SRC and MRC on the 6 road graphs and the two competition sets. Each variant is distinguished by suffix. The suffix +cut indicates a node ordering based on the balanced edge-separators graph cutting technique described earlier. The suffix +dfs indicates a node ordering based on depth-first search traversal, again as described earlier. The suffix +input indicates the order of the nodes is taken from the associated input file. For map graphs we ordered the fields lexicographically first by y- and then by x-coordinates.

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 computation. Our preprocessing approach involves executing Dijkstra's algorithm repeatedly; the time required could likely be significantly improved (at least for roads with unique shortest paths) using modern APSP techniques (e.g. (Delling et al. 2013)).

Creating a node order is fast; +dfs requires only fractions of a second. Even the +cut order requires not more than 18 seconds on average using METIS (Karypis and 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 overhead was on the order of seconds.

[1] https://code.google.com/p/gppc-2012/

Map	$\lvert V\rvert$	$\frac{\lvert E\rvert}{\lvert V\rvert}$	DB Size (MB)							Query Time (nanos)						
			Copa-G	MRC			SRC			Copa-G	MRC			SRC		
				+cut	+dfs	+input	+cut	+dfs	+input		+cut	+dfs	+input	+cut	+dfs	+input
Dragon Age: Origins (27 maps)																
Min	244	7	< 1	< 1	< 1	< 1	< 1	< 1	< 1	34	19	26	26	14	19	18
Q1	1828	7.2	< 1	< 1	< 1	< 1	< 1	< 1	< 1	63	22	31	35	16	22	26
Med	5341	7.4	1	< 1	1	1	< 1	1	2	81	30	44	54	20	31	38
Avg	30740	7.4	12	5	7	23	6	8	53	156	34	50	72	25	36	54
Q3	52050	7.6	18	6	10	29	7	12	65	266	36	62	106	28	45	82
Max	99630	7.7	75	31	39	106	35	44	349	316	95	116	176	67	78	138
StarCraft (11 maps)																
Min	104900	7.7	60	20	35	89	25	42	187	304	63	93	130	47	63	88
Q1	172600	7.7	128	28	61	144	33	71	281	324	70	103	142	51	69	102
Med	273500	7.8	183	69	111	393	83	126	956	334	95	121	187	66	77	133
Avg	288200	7.8	351	148	203	444	172	222	983	358	105	130	195	66	82	132
Q3	396100	7.8	510	189	282	621	222	308	1318	396	126	146	226	72	90	156
Max	493700	7.8	934	549	626	1245	630	660	2947	436	197	195	311	108	118	208

Table 2: Comparative performance of both Copa-G and a range of SRC and MRC variants. We test each one on two sets of grid graphs which have appeared in the 2012 GPPC. We measure (i) the size of the compressed database (in MB) and; (ii) the time needed to extract a first query (in nanos). We give results for both grid graphs and road graphs. All values are rounded to the nearest whole number (either MB or nano, respectively).

Graph	$\lvert V\rvert$	$\frac{\lvert E\rvert}{\lvert V\rvert}$	DB Size (MB)							Query Time (nanos)						
			Copa-R	Hub Labels		MRC		SRC		Copa-R	Hub Labels		MRC		SRC	
				RXL	CRXL	+cut	+dfs	+cut	+dfs		RXL	CRXL	+cut	+dfs	+cut	+dfs
BAY-d	321270	2.5	317	90	19	141	129	160	144	527	488	3133	89	100	62	69
BAY-t	321270	2.5	248	65	17	102	95	117	107	469	371	1873	74	87	52	60
COL-d	435666	2.4	586	138	24	228	206	268	240	677	564	3867	125	111	68	85
COL-t	435666	2.4	503	90	22	162	150	192	175	571	390	2131	88	97	58	65
NY-d	264346	2.8	363	99	21	226	207	252	229	617	621	4498	112	122	75	83
NY-t	264346	2.8	342	66	18	192	177	217	198	528	425	2529	98	111	67	75

Table 3: Comparative performance of SRC, MRC, Copa-R and two recent Hub Labeling algorithms. 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). All values are rounded to the nearest whole number (MB or nano, respectively).

Compression and Query Performance In Table 2 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 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.

In Table 3 we look at 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 speed. RXL wins in terms of CPD-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 4 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 FrozenSea 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.

To investigate the behaviors of the SamPG (i.e. RXL without the label compression) and SRC algorithms in greater detail, we report in Table 5 the average number of hubs per label and the average number of runs

Map	$\lvert V\rvert$	$\frac{\lvert E\rvert}{\lvert V\rvert}$	**DB Size (MB)**						**Query Time (nanos)**					
			Hub Labels		MRC		SRC		Hub Labels		MRC		SRC	
			RXL	CRXL	+cut	+dfs	+cut	+dfs	RXL	CRXL	+cut	+dfs	+cut	+dfs
ost100d	137375	7.7	62	24	39	50	49	57	598	5501	89	110	58	71
FrozenSea	754195	7.7	429	135	576	634	753	740	814	9411	176	192	104	109

Table 4: Comparative performance of SRC, MRC, RXL, and CRXL. We test two large grid graphs. TheFrozenSea is drawn from the game StarCraft; ost100d is drawn from the game Dragon Age Origins. We measure (i) database sizes in MB; (ii) the time needed to extract a first query (in nanos). All 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 5: We report the average number of hubs per label (length), number of runs per row (length), and average space usage per node. The reported hub label lengths accomodate only for one direction, i.e., we need two labels per node. SamPG is the label creation algorithm used by RXL without the encoding. The Plain encoding assumes 8 bytes per entry. The memory reported for SRC assumes 4 bytes per run.

per row using SRC. Using the straightforward encoding SamPG needs to store for each hub a 32-bit distance value, a 28-bit node ID and a 4-bit out-arc ID, whereas a compressed SRC-run consists of only a 28-bit node ID and a 4-bit out-arc ID. SRC-runs are therefore more compact as we do not have to store the 32-bit distance value. On the other hand, in general there are more runs than hubs. It is surprising that SamPG+Plain labels consistently occupies more bytes than SRC rows, even though the experiments to far suggest that RXL is more compact. The explanation is that RXL (in contrast to SamPG+Plain) applies some additional compression compared to SamPG+Plain, such as for example delta-compression, while SRC does not.

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.

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.

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 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-arc 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-arc exists, SRC and MRC select an arc that will improve the compression, whereas Copa sticks with one arbitrary optimal out-arc. 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 and Harabor 2013).

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, Copa stores a 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 (cf. (Botea 2011; Botea and Harabor 2013)).

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-sides 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.

Conclusion

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. We develop two novel approaches: SRC, using a simple run-length encoding scheme, and MRC, which improves compression by identifying 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 and Harabor 2013), and RXL. Copa is a state-of-the-art CPD method and one of the joint winners at the 2012 Grid-based Path Planning Competition. RXL is the state-of-the-art for distance oracles on 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 show that SRC and MRC outperform RXL in terms of query times.

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.[2] In multi-agent pathfinding for example it is sometimes useful to guarantee properties like “there must always be a local detour available” (e.g. as in (Wang and Botea 2011)). Another example are turn-costs in road graphs. A possible direction for future work is creating CPDs that store only paths satisfying such constraints.

The MRC compression relies on the 32-bit patterns being exactly the same. The intuition we want to exploit is that the upper 28 bits coding the run starts are similar. However, there is no real reason why the lower 4 bits encoding the out-arc ID should be the same. In our experiments arranging the out-arc IDs in the same order as the arcs appear in the input resulted in the best MRC compression. Devising an algorithm that optimizes the out-arc ID assignment would directly translate in smaller MRC space consumption.

Acknowledgment

We would like to thank Daniel Delling & Thomas Pajor for running some experiments for us.

References

Abraham, I.; Delling, D.; Goldberg, A. V.; and 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)*, volume 6630 of *Lecture Notes in Computer Science*, 230–241. Springer.

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

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

Akiba, T.; Iwata, Y.; and 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)*. ACM Press.

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

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

Bast, H.; Funke, S.; Matijevic, D.; Sanders, P.; and 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)*, 46–59. SIAM.

Bast, H.; Delling, D.; Goldberg, A. V.; Müller–Hannemann, M.; Pajor, T.; Sanders, P.; Wagner, D.; and Werneck, R. F. 2014. Route planning in transportation networks. Technical Report MSR-TR-2014-4, Microsoft Research.

Bast, H.; Funke, S.; and Matijevic, D. 2009. Ultrafast shortest-path queries via transit nodes. In *The Shortest*

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

Path Problem: Ninth DIMACS Implementation Challenge, volume 74 of *DIMACS Book*. American Mathematical Society. 175–192.

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

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)*, 122–127. AAAI Press.

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

Cohen, E.; Halperin, E.; Kaplan, H.; and Zwick, U. 2003. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing* 32(5):1338–1355.

Delling, D.; Goldberg, A. V.; Nowatzyk, A.; and 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.; and Werneck, R. F. 2014. Robust distance queries on massive networks. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, Lecture Notes in Computer Science. Springer. to appear.

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

Demetrescu, C.; Goldberg, A. V.; and Johnson, D. S., eds. 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company.

Ishida, T., and Korf, R. E. 1991. Moving Target Search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 204–210.

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

Knopp, S.; Sanders, P.; Schultes, D.; Schulz, F.; and Wagner, D. 2007. Computing many-to-many shortest paths using highway hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, 36–45. SIAM.

Planken, L.; de Weerdt, M.; and van der Krogt, R. 2011. Computing all-pairs shortest paths by leveraging low treewidth. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Sankaranarayanan, J.; Alborzi, H.; and Samet, H. 2005. Efficient query processing on spatial networks. In *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems (GIS)*, 200–209.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*.

van Schaik, S. J., and 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, 913–924. New York, NY, USA: ACM.

Wang, K.-H. C., and Botea, A. 2011. Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *J. Artif. Intell. Res. (JAIR)* 42:55–90.