Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)

Compressed Path Databases with Ordered Wildcard Substitutions

Matteo Salvetti
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Brescia, Italy

Adi Botea
IBM Research
Ireland

Alessandro Saetti, Alfonso Emilio Gerevini
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Brescia, Italy

Abstract

Compressed path databases (CPDs) are a state-of-the-art approach to path planning, a core AI problem. In the Grid-based Path Planning Competition, the CPD-based SRC path planning system was the fastest competitor with respect to *both* computing full optimal paths and computing the first moves of an optimal path. However, on large maps, CPDs can require a significant amount of memory, which can be a serious practical bottleneck. We present an approach that significantly reduces the size of a CPD. Our approach replaces part of the data encoded in a CPD with wildcards ("don't care" symbols), maintaining the ability to compute optimal paths for all pairs of nodes of an undirected graph. We show that using wildcards in a way that maximizes the memory savings is NP-hard. We consider heuristics that achieve a good performance in practice. We implement our ideas on top of SRC and provide a detailed empirical analysis. Average memory savings can reach a factor of 2. Our first-k-moves lag (i.e., the time before knowing the first k optimal forward moves) increases, but it can be kept within competitive values. The speed of computing full optimal paths improves slightly.

1 Introduction

A compressed path database (CPD) is a data structure that encodes optimal first moves from any node s towards any other node t in a graph. In optimal path planning, CPDs have been shown to achieve a state-of-the-art speed performance. CPD-based systems such as Copa (Botea 2012) and SRC (Strasser, Harabor, and Botea 2014) have been top performers in two different editions of the Grid-based Path Planning Competition.

A CPD system precomputes and stores the first move of all-pairs shortest paths (APSPs) for the input graph. When a new query is addressed, the moves of an optimal path are retrieved one by one from the CPD, as opposed to performing a more expensive graph search. As a naive encoding of APSP data would be prohibitively large, CPDs compress their data. Despite impressive compression results (Botea and Harabor 2013; Strasser, Harabor, and Botea 2014), CPDs can still be very large when the input graph is large, which can make the approach infeasible.

We present an approach to reducing the size of CPDs. In this work we focus on undirected graphs. The undirectedness assumption is very common in gridmap path planning.

The intuition behind our work is simple to understand. Given any two nodes s and t, a standard CPD encodes both an optimal first move from s towards t, *and* an optimal first move from t towards s. However, it is sufficient to require that, for every pair (s, t), at least one of these moves is available. This allows to reconstruct a full optimal path between any two nodes (or any fragment of an optimal path). For instance if a first move from s towards t is not available, get t', the resulting node after the first optimal move from t towards s, and continue recursively with the pair (s, t').

The observation that one out of the two moves is sufficient allows to replace up to about half of the symbols encoded in a standard CPD with wildcards (i.e., "don't care" symbols). Wildcards allow to improve the compression of a CPD, with a corresponding reduction of the CPD size.

We develop our ideas on top of the state-of-the-art system SRC (Strasser, Harabor, and Botea 2014). SRC represents APSP data as a so-called first-move matrix $\mathbf{m}$ where an entry $\mathbf{m}[u, v]$ is an optimal first move from node u towards node v. Then, each matrix row is compressed with run-length encoding (RLE). Consider the string $aaabaaaabaaa$. RLE encodes it more efficiently as $a, 1; b, 4; a, 5; b, 9; a, 10$ (i.e., there is a solid block of one or more a symbols, starting at position 1, followed by a solid block of one or more b symbols, starting at position 4, and so on). Each solid block is called a run, and our sample string is split into five runs.

Imagine now that we replace each b with a wildcard. Now the whole string becomes $aaa{*}aaaa{*}aaa$, and it can be represented much more compactly with only one run, namely $(a, 1)$, since the wildcard is a "don't care" symbol. This illustrates how wildcards can reduce the size of a CPD.

In deciding what subset of entries in the first-move matrix should be replaced with wildcards, our approach uses an ordering of the graph nodes that we call the *wildcard node ordering* (shorter, *wildcard ordering*). Put it simply, if a node v comes before a node u in the wildcard ordering, then, in the first-move matrix $\mathbf{m}$, we can replace the symbol $\mathbf{m}[u, v]$ with a wildcard. All wildcard orderings are correct, but they differ in terms of their compression power. In the previous example, replacing both b symbols led to a good additional compression, reducing the number of runs from 5 to 1. On

the other hand, replacing any two a symbols would have no impact in the number of runs.

We perform a theoretical analysis, showing that finding a wildcard order that maximizes the memory savings is NP-hard. Due to this theoretical result, we consider a range of heuristic wildcard orderings.

While reducing the CPD size, wildcards could increase the first-k-moves lag, which is the time before the first k optimal forward moves are known. The reason is that we build an optimal path as a combination of "forward" moves (i.e., moves from the current source towards the current target) and "backwards" moves (i.e., from the target towards the source). Some backwards moves could be needed before the first k forward moves. We present heuristics focused on how many wildcards should be considered to balance well the trade-off between the CPD size and the first-k-moves lag.

We perform a detailed empirical analysis on a large set of game maps from Sturtevant's online repository (Sturtevant 2012), a well known standard testbed in path planning research. In domains where the full path is important but the first-k-moves lag is less important, wildcard ordering heuristics focusing only on the size can reduce the CPD size by a factor of two on average. On the other hand, when the first-k-moves lag is also important, we can tune our heuristics, thanks to their parameters, to balance the trade-off between the CPD size and the first-move lag. The time to compute full optimal paths can improve slightly.

2 Related Work

We start with techniques that precompute and compress APSP data. In the preprocessing, methods such as SILC (Sankaranarayanan, Alborzi, and Samet 2005), Copa (Botea 2011; 2012; Botea and Harabor 2013) and SRC/MRC (Strasser, Harabor, and Botea 2014; Strasser, Botea, and Harabor 2015) iterate through all nodes of the input graph. At one iteration, the node at hand s plays the role of a current node (source). All other nodes are potential targets. A Dijkstra search rooted at s assigns to target nodes t labels representing a first optimal move from s towards t.

At each iteration, the set of first-move labels is compressed. For this purpose, SILC splits the map using quadtrees, requiring that all targets included in one square of the quadtree have the same first-move label. Copa splits the map using rectangles of arbitrary sizes and positions, obtaining a better compression than quadtrees. SRC takes a different approach. Instead of compressing a bi-dimensional structure such as the input map, SRC represents all nodes as a one-dimensional string, using a fixed ordering of the nodes. The string is compressed with run-length encoding (RLE), as mentioned in the introduction and discussed in more detail in subsequent sections.

Methods such as Copa and SRC/MRC make use of wildcards in a limited way. Only moves for same-node pairs (s, s) are treated as a wildcard ("don't care" symbol). As such, the impact of using wildcards is limited, as there are only n such same-node pairs, out of n^2 node pairs in total. In contrast, our method allows to use wildcards for more than half of the n^2 node pairs, with a substantial reduction of the CPD size, as shown in the experimental evaluation.

When a shortest path query is posed, systems such as those mentioned earlier do not perform any search in the problem graph. Instead, they retrieve optimal moves one by one and chain them into an optimal solution. In contrast, other modern pathfinding approaches, such as searching in subgoal graphs (Uras, Koenig, and Hernández 2013), Jump Point Search (JPS) (Harabor and Grastien 2011), and JPS with bounding boxes (Rabin and Sturtevant 2016) perform a highly optimized search in a graph obtained from the input gridmap. In systems that perform graph search, the first-k-moves lag is equal to the time to find the entire path. The reason is that a graph search has to be completed all the way to the target before knowing even the first optimal move. In contrast, CPD-based systems can provide the first moves along a path much earlier than the target is reached.

SRC (Strasser, Harabor, and Botea 2014; Strasser, Botea, and Harabor 2015) was shown in previous work and in the Grid-based Path Planning Competition to be a state-of-the-art approach, exceeding the speed performance of other systems discussed in this section. This is why we implement our ideas on top of SRC, and use SRC as a benchmark algorithm in our empirical evaluation.

3 Background

Given a graph $G = (V, E)$, a first-move matrix $\mathbf{m}$ is a square, $|V| \times |V|$ matrix, with the property that $\mathbf{m}[i, j]$ (i.e., the cell on row i and column j) is an optimal first move from node i towards node j. The size of a naive encoding of a first-move matrix can be within $O(|V|^2)$ but, as shown in the related work section, the literature shows methods to compress a first-move matrix.

Strasser, Harabor, and Botea (2014) use RLE to compress each matrix row. RLE compresses a string of symbols by representing more compactly substrings, called *runs*, consisting of repetitions of the same symbol. E.g., the string $\sigma = aabbbaaa$ 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 string is represented more efficiently as a,1; b,3; a,6; we call these *sequential runs*.

Cyclic runs (Botea, Strasser, and Harabor 2015) are obtained by putting the two ends of the string next to each other, as if the string was cyclic, and then counting the runs. Consider again the string $\sigma = aabbbaaa$. There are three sequential runs, as shown earlier. At the same time, σ has two cyclic runs. The reason is that the first and the last sequential runs have the same symbol, namely a. When the two ends of the string are put next to each other, all a symbols become one single solid block. When the first and the last symbols of a string are different, sequential and cyclic runs are equivalent. When the first and the last symbols are identical, the number of cyclic runs is smaller by 1. In the example above, the sequential runs are $a, 1; b, 3; a, 6$. The cyclic runs are $b, 3; a, 6$. Cyclic runs are sufficient to reconstruct the original string (Botea, Strasser, and Harabor 2015).

Cyclic runs could save some additional memory in comparison to serial runs, but in practice the differences often are very small. However, cyclic runs are important because

they allow an easier theoretical analysis. Similarly to Botea, Strasser, and Harabor (2015), we use cyclic runs in the theoretical analysis presented in this paper.

In this work, we assume that the ordering of the columns of the first-move matrix is fixed. Column orderings have been studied in previous work (Strasser, Harabor, and Botea 2014; Botea, Strasser, and Harabor 2015), and reordering the columns is beyond our focus in this work.

Given an input graph, we assume that we index locally the outgoing edges of every node i with numbers from 0 to $od(i) - 1$, where $od(i)$ is the out-degree of the node i. In the case of an undirected graph, all edges that are adjacent to i are both incoming and outgoing edges. We encode an optimal first move from i towards j as the local index of the outgoing edge of i that represents the first step on an optimal path to j. On a sparse graph, locally indexing edges has the advantage that each local index fits into a small number of bits, with a corresponding benefit in the size of a compressed path database. Notice that graphs used in path planning, such as 8-connected gridmaps, are sparse, with nodes having no more than 8 adjacent edges each.

4 Ordered Wildcard Substitutions

A wildcard is a "don't care" symbol that can improve the effectiveness of RLE compression. Consider the string $\sigma = aabbcbb$. This string has 4 runs (either sequential or cyclic). If we replace the c symbol with a wildcard, the number of runs reduces to two. Wildcards can be used to greatly reduce the size of a first-move matrix compressed with run-length encoding. As shown next, CPDs with wildcards allow to perform optimal pathfinding.

To introduce our wildcard-based approach informally, consider a total ordering of the graph nodes, $o = i_1, i_2 \dots i_n$. We call this a wildcard ordering. In $\mathbf{m}$, replace the symbol on row r and column c with a wildcard if and only if c come before r in the wildcard ordering. Despite removing some information, the remaining information (not replaced with wildcards) is sufficient to reconstruct an optimal path from any node i to any node j, with $i \neq j$. Indeed, exactly one of $\mathbf{m}[i, j]$ and $\mathbf{m}[j, i]$ is preserved in the matrix, while the other one is replaced with a wildcard. If $\mathbf{m}[i, j]$ is preserved, take a move from i towards j and continue recursively. Otherwise, take a move from j towards i and continue recursively. See an example later in this section. Before the example, we formalize this intuitive idea.

Definition 1 (Basic wildcard range, BWR)**.** *Let $i_1, i_2 \dots i_n$ be any reordering of the numbers from 1 to n. The basic wildcard range $\lambda(i_k)$ of i_k is defined as $\lambda(i_k) = \{i_1, i_2 \dots i_k\}$, for $1 \leq k \leq n$.*

BWR defines the symbols to replace with a wildcard on each row, as described in the previous paragraph. Notice that each row i also includes $\mathbf{m}[i, i]$, which is never needed for pathfinding purposes anyways. I.e., we never need a move from the node i to itself.

Definition 2 (Basic ordered wildcard substitution, BOWS)**.** *Given a square matrix $\mathbf{m}$, let $o = i_1 \dots i_n$ be a wildcard ordering. A basic ordered wildcard substitution replaces some symbols with wildcards in the matrix as follows. In the k-th row in the ordering o (i.e., the row with the index i_k), replace with wildcards all symbols $\mathbf{m}[i_k, l]$, where $l \in \lambda(i_k)$.*

Note that the total number of wildcards is $\frac{n^2+n}{2}$, thus exceeding half of first-move matrix cells.

The following example illustrates our idea. Consider the toy 8-connected grid map shown in Figure 1 (a). Its first-move matrix graph is shown in Figure 1 (b). Figure 1 (c) shows the first-move matrix after applying our ordered wildcard substitution with B, E, F, A, C, G, D as wildcard ordering. In the matrices, optimal moves are represented as numbers as follows: 0 = East, 1 = South-East, 2 = South, ..., 7 = North-East. For instance, $m[A, G]$ = 2 because going South is an optimal move from A towards G.

In the preprocessing, Strasser, Harabor, and Botea (2014) compress with RLE the matrix shown in Figure 1 (b). In contrast, we compress the matrix shown in Figure 1 (c). In this toy example, the former matrix has 16 serial runs, and the latter has 12 serial runs. The savings can be much greater on large maps, as shown in the experiments.

After the preprocessing is completed and the system starts answering shortest-path queries, assume that we need to compute an optimal path from C to E, using a CPD after applying our wildcards. As $C > E$ in the wildcard ordering in use, it follows that we have a backwards move available from E to C, namely move 6 (i.e., North). Indeed, in Figure 1 (c), $\mathbf{m}[E, C] = 6$ (as opposed to that cell being a wildcard). The result of that move is D. We continue recursively with the pair (C, D). $C < D$ in the wildcard ordering, which means we have a move available from C towards D (i.e., move 4 = West, since $\mathbf{m}[C, D] = 4$ in Figure 1 (c)). The result of this forward move is B. We continue recursively with pair (B, D). $B < D$ in the wildcard ordering, so a forward move from B towards D is available (i.e., 3 = South-West). After this move, the two segments of the optimal path, namely the forward segment C, B, D and the backward segment E, D have met, and the optimal path is complete. Notice that, before being able to make the first move forward, we had to make one move backwards. In general, 0 or more backwards moves may be needed before making the first k moves forward. This is why the first-k-move lag can be longer than k in our approach.

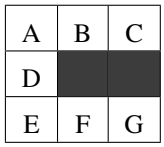
5 Theoretical Analysis

We provide an NP-completeness result about computing a wildcard ordering that minimizes the number of runs of a matrix. We introduce a few definitions and assumptions, for an easier formal analysis.

Definition 3 (Wildcard range with one swap, WR1)**.** *Let $i_1, i_2 \dots i_n$ be any reordering of the numbers from 1 to n. The range $\lambda_1(i_k)$ of i_k is defined as follows:*

- $\lambda_1(i_1) = \{i_1, i_n\}$
- $\lambda_1(i_k) = \{i_1, i_2 \dots i_k\}$, *for* $2 \leq k \leq n - 1$
- $\lambda_1(i_n) = \{i_2, i_3 \dots i_n\}$

WR1 is a slight variant of BWR, introduced in Definition 1. Compared to BWR, WR1 moves the location of one wildcard symbol from $\mathbf{m}[i_n, i_1]$ to $\mathbf{m}[i_1, i_n]$. We call this change a swap. This swapping preserves the key property



(a)

$$\mathbf{m} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \end{array} \begin{array}{c} \begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\ \begin{bmatrix} * & 0 & 0 & 2 & 2 & 2 & 2 \\ 4 & * & 0 & 3 & 3 & 3 & 3 \\ 4 & 4 & * & 4 & 4 & 4 & 4 \\ 6 & 7 & 7 & * & 2 & 1 & 1 \\ 6 & 6 & 6 & 6 & * & 0 & 0 \\ 5 & 5 & 5 & 5 & 4 & * & 0 \\ 4 & 4 & 4 & 4 & 4 & 4 & * \end{bmatrix} \end{array}$$

(b)

$$\mathbf{m} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \end{array} \begin{array}{c} \begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\ \begin{bmatrix} * & * & 0 & 2 & * & * & 2 \\ 4 & * & 0 & 3 & 3 & 3 & 3 \\ * & * & * & 4 & * & * & 4 \\ * & * & * & * & * & * & * \\ 6 & * & 6 & 6 & * & 0 & 0 \\ 5 & * & 5 & 5 & * & * & 0 \\ * & * & * & 4 & * & * & * \end{bmatrix} \end{array}$$

(c)

Figure 1: Left: a small grid map. The black cells are not traversable. Middle: uncompressed first-move matrix. Right: (uncompressed) first-move matrix with BOWS, using *B, E, F, A, C, G, D* as a wildcard ordering. In the matrices, numbers represent optimal moves; the symbol * represents a wildcard.

that, given any pair of distinct nodes s and t, the database provides either an optimal move from s to t, or an optimal move from t to s. WR1 is introduced because the technical details of the formal analysis presented in the remaining part of this section are easier to perform with this variant in use instead of BWR. In terms of empirical performance, the difference between WR1 and BWR is negligible, as only one wildcard moves its position, out of a large number of wildcards used on a map.

Definition 4 (Ordered wildcard substitution with one swap, OWS-1)**.** *Given a square matrix* $\mathbf{m}$*, assume that the ordering of columns is fixed, and let* $o = i_1 \dots i_n$ *be an ordering of its rows. OWS-1 replaces some symbols with wildcards in the matrix as follows. In the* k*-th row in the ordering* o *(i.e., the row with the index* i_k*), replace with wildcards all symbols* $\mathbf{m}[i_k, l]$*, where* $l \in \lambda_1(i_k)$*.*

Figure 2 (a) shows a matrix used as a running example, and Figure 2 (b) illustrates applying OWS-1.

Given a wildcard ordering $i_1, i_2 \dots i_n$, we define the *next cyclic position* $nc(i_k)$ as i_{k+1} if $k < n$ and as i_1 if $k = n$.

Definition 5. *The* nc wildcard substitution rule *replaces with a wildcard elements* $\mathbf{m}[i_k, nc(i_k)]$*, for* $1 \leq k \leq n$*.*

Definition 6 (NOWS-1: OWS-1 plus the nc rule)**.** *NOWS-1 is a substitution method combining OWS-1 and the* nc *rule.*

Figure 2 (c) illustrates applying NOWS-1.

Remark 1. *For* $n \geq 3$*, positions replaced with wildcards when applying OWS-1 are disjoint from positions replaced with wildcards when applying* nc *substitutions.*

See Figures 2 (b) and (c) for an illustration of this remark.

OWS-1 is a lossless technique, in the sense that it allows reconstructing a full optimal path (or any fragment) for any pair of nodes. The interesting discussion is whether the nc rule is a loseless rule. In general, adding the nc substitution rule on top of OWS-1 could result into a lossy method, due to the removal (replacement with wildcards) of symbols $\mathbf{m}[i_k, nc(i_k)]$ in the matrix. Indeed, if we want either an optimal move from node i_k to node $nc(i_k)$ or an optimal move from node $nc(i_k)$ to i_k, the matrix has neither of them after performing the nc-based substitutions.

However, the information $\mathbf{m}[i_k, nc(i_k)]$ can be handled separately. With the local indexing of edges in use, presented in the background section, a very simple trick ensures that actually there is *no information loss*. After a wildcard ordering is selected (and thus the nc function is known), reindex the outgoing edges of each graph node such that the local index of the first-move from i_k towards $nc(i_k)$ is the same for every node i_k. With no generality loss, assume that such a common first-move index is 0. With this re-indexing in place, if such a move is needed, return 0.

For example, for a given node i_k, assume that the first move from i_k towards the node $nc(i_k)$ has the index l. If $l > 0$, swap the indexes of the two outgoing edges of i_k initially indexed as 0 and l.

In summary, the nc rule is a lossless method since, in the common implementation of graphs, every node has its adjacent edges indexed locally, as shown above.

Definition 7 (Optimal wildcard ordering for NOWS-1, OPT-NOWS-1)**.** *Input: a square matrix* $\mathbf{m}$ *and an integer* k*. Question: Is there a wildcard ordering* o *such that, after applying NOWS-1 on that ordering, the number of cyclic runs summed up across all rows is at most* k*?*

Theorem 1. *OPT-NOWS-1 is NP-complete.*

Proof. Clearly, the problem is within NP, as solutions can be guessed and verified in polynomial time. The hardness is shown with a reduction from the problem of the existence of a Hamiltonian cycle in a 3-regular, undirected, bipartite graph, 3REG-BIP-HAM, a known NP-complete problem (Takanori, Takao, and Nobuji 1980). Let $G = (V, E)$ be an arbitrary 3-regular undirected bipartite graph, and let $n = |V|$. Figure 3 shows a sample 3-regular undirected bipartite graph. Given an instance of 3REG-BIP-HAM, i.e., $G = (V, E)$, we construct an instance of OPT-NOWS-1. Specifically, we build a matrix $\mathbf{m}$ of size $n \times n$, defined as follows: $\mathbf{m}[k, l] = l$ if $(k, l) \in E$, and $\mathbf{m}[k, l] = 0$ if $(k, l) \notin E$. This construction can be done in time that is polynomial in the size of the 3REG-BIP-HAM instance. The matrix shown in Figure 2 (a) is precisely the matrix $\mathbf{m}$ corresponding to our sample graph.

Note that G has $\frac{3n}{2}$ edges, and $\mathbf{m}$ has exactly $3n$ positive symbols, 3 per row. We claim that G has a Hamiltonian cycle if and only if $\mathbf{m}$ ends up with $\frac{3n}{2}$ cyclic runs.

The "only if" part: Assume that G has a Hamiltonian cycle, and pick a wildcard ordering o identical to the ordering of the nodes in a randomly chosen Hamiltonian path

$$\mathbf{m} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 4 & 5 & 6 \\ 0 & 0 & 0 & 4 & 5 & 6 \\ 0 & 0 & 0 & 4 & 5 & 6 \\ 1 & 2 & 3 & 0 & 0 & 0 \\ 1 & 2 & 3 & 0 & 0 & 0 \\ 1 & 2 & 3 & 0 & 0 & 0 \end{bmatrix} \end{array} \quad \mathbf{m} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} * & 0 & 0 & 4 & 5 & * \\ * & * & 0 & * & 5 & 6 \\ * & * & * & * & * & 6 \\ * & 2 & 3 & * & 0 & 0 \\ * & * & 3 & * & * & 0 \\ 1 & * & * & * & * & * \end{bmatrix} \end{array} \quad \mathbf{m} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} * & 0 & 0 & \mathbf{*} & 5 & * \\ * & * & 0 & * & \mathbf{*} & 6 \\ * & * & * & * & * & \mathbf{*} \\ * & \mathbf{*} & 3 & * & 0 & 0 \\ * & * & \mathbf{*} & * & * & 0 \\ \mathbf{*} & * & * & * & * & * \end{bmatrix} \end{array}$$

(a) (b) (c)

Figure 2: Left: A sample matrix. Middle: The matrix after applying OWS-1, using the wildcard ordering $1, 4, 2, 5, 3, 6$. Right: The matrix after applying NOWS-1 with the same wildcard ordering. The symbol * represents a wildcard. At the right, wildcards corresponding to the nc rule are shown in bold.

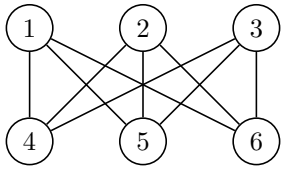


Figure 3: A 3-regular undirected bipartite graph.

stemming from the Hamiltonian cycle at hand. (I.e., obtain a Hamiltonian path by disregarding one edge of the cycle.)

Recall that NOWS-1 is OWS-1 plus an additional replacement of symbols of the form $\mathbf{m}[i_k, nc(i_k)]$. Regardless of the wildcard ordering, OWS-1 replaces exactly half of the positive symbols. Indeed, observe that, for any two distinct indexes i_l and i_k, $\mathbf{m}[i_k, i_l] > 0$ iff $\mathbf{m}[i_l, i_k] > 0$. OWS-1 will remove exactly one of these two since exactly one of the following two relations holds: either $i_l \in \lambda_1(i_k)$ or $i_k \in \lambda_1(i_l)$. See Figure 2 (b) for an example.

In addition to OWS-1, NOWS-1 substitutes n symbols of the form $\mathbf{m}[i_k, nc(i_k)]$. See Figure 2 (c) for an example. Since the ordering at hand corresponds to a Hamiltonian cycle, all the symbols replaced with wildcards using the nc rule are positive symbols, corresponding to edges included in the Hamiltonian cycle. It follows that there are $3n - \frac{3n}{2} - n = \frac{n}{2}$ positive symbols left in the matrix.

Furthermore, there is at most one positive symbol per row. Indeed, every row initially had 3 positive symbols, and the two of them corresponding to adjacencies included in the Hamiltonian path are replaced for sure.

By construction, using NOWS-1, rows i_{n-1} and i_n contain only wildcards, row i_{n-2} contains exactly one non-wildcard symbol, and every other row contains at least two non-wildcard symbols. In row i_{n-2}, the non-wildcard symbol is $\mathbf{m}[i_{n-2}, i_n]$. In Figure 2 (c), this is $\mathbf{m}[5, 6]$. We claim that the non-wildcard symbol on that row is always 0. Indeed, the Hamiltonian cycle implies the existence of edges (i_{n-2}, i_{n-1}) and (i_{n-1}, i_n). A positive value $\mathbf{m}[i_{n-2}, i_n]$ would imply the existence of an edge (i_{n-2}, i_n), leading to a cycle of length 3, namely (i_{n-2}, i_{n-1}, i_n). However, odd-length cycles cannot exist in a bipartite graph.

It follows that each of the $n/2$ positive symbols occur on rows with at least 2 non-wildcard symbols. As remarked earlier, there is at most one positive symbol per row, which implies that there is at least one zero symbol in each of these rows. In summary, these rows have one positive symbol, one or more zero symbols, and some wildcards. Every row like this (i.e., row with a positive symbol) has two cyclic runs. All other rows (i.e., rows with only zeros and/or wildcards) have one run each. These add up to $3n/2$ runs.

The "if" part: Assume that the total number of runs N_t is at most $3n/2$ for a wildcard ordering $i_1, \ldots, i_n$. After applying NOWS-1, rows with at least one 0 are called *0-populated rows*. A 0-populated row with p positive symbols (and any number of wildcards) has at least $p+1$ runs. A row with no zeros has $p \geq 1$ positive symbols, and any number of wildcards has p runs. We call these *solid-positive rows*. Thus, if a matrix has z positive elements distributed across b_0 0-populated rows and b_1 solid-positive rows, we have $N_t \geq n + z - b_1$ (n is there because each row has at least one run, and positive symbols add at least $z - b_1$ runs). Note that $b_0 + b_1 = n - 2$ as rows i_{n-1} and i_n, with only wildcards, are neither 0-populated nor solid-positive.

In the rest of this section, every time we write $\mathbf{m_0}[i, j]$ we refer to the value *before applying NOWS-1*. We still write $\mathbf{m}[i, j]$ when the value is not impacted by NOWS-1.

Let w be the number of cells $\mathbf{m_0}[i_k, nc(i_k)]$ equal to 0. Recall that OWS-1 leaves $3n/2$ positive symbols in place, and nc replaces one more symbol (either 0 or positive) per row. It follows that $z = n/2 + w$ and thus $N_t \geq 3n/2 + w - b_1$. As $N_t \leq 3n/2$, it follows that $w \leq b_1$.

For $n \geq 8$, rows $i_1, \ldots, i_{n-4}$ are 0-populated.* Hence at most two rows (i_{n-2} and i_{n-3}) could be solid-positive (i.e., $b_1 \leq 2$). It follows $w \leq b_1 \leq 2$, and hence $w \leq 2$. We call this last relation *the α property*.

In the rest of the proof we consider the following four cases. We show that in Cases (1) and (4.1) the graph admits

*By construction, each row $i_1, \ldots, i_{n-5}$ is 0-populated, having at least 4 non-wildcard symbols, out of each at most 3 are positive. Thus $b_0 \geq n - 5$ and $b_1 = n - 2 - b_0 \leq 3$. Assume by contradiction that row i_{n-4} is solid-positive (i.e., $\mathbf{m}[i_{n-4}, i_{n-2}]$, $\mathbf{m}[i_{n-4}, i_{n-1}]$, $\mathbf{m}[i_{n-4}, i_n]$ are positive). Then $\mathbf{m_0}[i_{n-5}, i_{n-4}] = \mathbf{m_0}[i_{n-4}, i_{n-3}] = 0$, as the degree of node i_{n-4} is exactly 3; $\mathbf{m_0}[i_{n-2}, i_{n-1}] = 0$ to avoid the odd-length cycle $(i_{n-4}, i_{n-2}, i_{n-1})$; $\mathbf{m_0}[i_{n-1}, i_n] = 0$ to avoid cycle (i_{n-4}, i_{n-1}, i_n). Thus, $w \geq 4$ and $4 \leq w \leq b_1 \leq 3$. Contradiction.

a Hamiltonian cycle, while all other cases are impossible.

Case 1: $\mathbf{m}[i_{n-2}, i_n] = 0$ and $\mathbf{m_0}[i_{n-3}, i_{n-2}] > 0$. Row i_{n-2} is 0-populated as $\mathbf{m}[i_{n-2}, i_n] = 0$. We show that also row i_{n-3} is 0-populated. Assume by contradiction that $\mathbf{m}[i_{n-3}, i_{n-1}] > 0$ and $\mathbf{m}[i_{n-3}, i_n] > 0$. Then $\mathbf{m_0}[i_{n-4}, i_{n-3}] = 0$, since the degree of node i_{n-3} is exactly 3. Furthermore, $\mathbf{m_0}[i_{n-2}, i_{n-1}] = 0$ to avoid the odd-length cycle $(i_{n-3}, i_{n-2}, i_{n-1})$, and $\mathbf{m_0}[i_{n-1}, i_n] = 0$ to avoid the cycle (i_{n-3}, i_{n-1}, i_n). It follows that $w \geq 3$, but this contradicts the α property. Thus, all rows that could have positive symbols are 0-populated (i.e., $b_1 = 0$). It follows that $w = 0$, i.e., $i_1, \ldots, i_n, i_1$ is a Hamiltonian cycle.

Case 2: $\mathbf{m}[i_{n-2}, i_n] = \mathbf{m_0}[i_{n-3}, i_{n-2}] = 0$. Case 2.1: row i_{n-3} is 0-populated. It follows that $b_1 = 0$. As $\mathbf{m_0}[i_{n-3}, i_{n-2}] = 0$, $w \geq 1$. Contradiction with $w \leq b_1$. Case 2.2: row i_{n-3} is solid-positive (i.e., $\mathbf{m}[i_{n-3}, i_{n-1}] > 0$ and $\mathbf{m}[i_{n-3}, i_n] > 0$) and hence $b_1 = 1$. It follows that $\mathbf{m_0}[i_{n-1}, i_n] = 0$, to avoid the odd-length cycle (i_{n-3}, i_{n-1}, i_n). Thus $w \geq 2$. Contradiction with $w \leq b_1$. Thus Case 2 is impossible.

Case 3: $\mathbf{m}[i_{n-2}, i_n] > 0$ and $\mathbf{m_0}[i_{n-3}, i_{n-2}] > 0$. As $\mathbf{m}[i_{n-3}, i_n] = 0$ (to avoid the odd-length cycle (i_{n-3}, i_{n-2}, i_n)) row i_{n-3} is 0-populated and thus the only solid-positive row is i_{n-2} (i.e., $b_1 = 1$). Thus $w \leq 1$ (i.e., at most one cell $\mathbf{m_0}[i_k, nc(i_k)]$ is equal to 0). If either $\mathbf{m_0}[i_{n-2}, i_{n-1}] = 0$ or $\mathbf{m_0}[i_{n-1}, i_n] = 0$, it follows that $i_1, i_2 \ldots i_{n-2}, i_n, i_1$ is a cycle of length $n-1$, which is an odd length. Contradiction. Otherwise, (i_{n-2}, i_{n-1}, i_n) is a cycle of length 3, which is a contradiction again. Thus Case 3 is impossible.

Case 4: $\mathbf{m}[i_{n-2}, i_n] > 0$ and $\mathbf{m_0}[i_{n-3}, i_{n-2}] = 0$. Case 4.1: $\mathbf{m}[i_{n-3}, i_{n-1}] > 0$ and $\mathbf{m}[i_{n-3}, i_n] > 0$. It follows that $\mathbf{m_0}[i_{n-1}, i_n] = 0$, to avoid the odd-length cycle (i_{n-3}, i_{n-1}, i_n). As $\mathbf{m_0}[i_{n-3}, i_{n-2}] = \mathbf{m_0}[i_{n-1}, i_n] = 0$, all other elements $\mathbf{m_0}[i_k, nc(i_k)]$ are positive (cf. property α). Thus $i_1, i_2, \ldots i_{n-3}, i_{n-1}, i_{n-2}, i_n, i_1$ is a Hamiltonian cycle. Case 4.2: At least one of $\mathbf{m}[i_{n-3}, i_{n-1}]$ and $\mathbf{m}[i_{n-3}, i_n]$ is equal to 0 and thus row i_{n-3} is 0-populated. Hence i_{n-2} is the only solid-positive row, and $b_1 = 1$. This further implies that $w \leq 1$, which means that at most one element $\mathbf{m_0}[i_k, nc(i_k)]$ is equal to 0. As $\mathbf{m_0}[i_{n-3}, i_{n-2}] = 0$, according to the assumption made for this case, all other elements $\mathbf{m_0}[i_k, nc(i_k)]$ are positive. It follows that $\mathbf{m_0}[i_{n-2}, i_{n-1}] > 0$ and $\mathbf{m_0}[i_{n-1}, i_n] > 0$, leading to the odd-length cycle (i_{n-2}, i_{n-1}, i_n). Contradiction. Thus Case 4.2 is impossible. □

Note that the matrix $\mathbf{m}$ used in Definition 7 and constructed in the previous proof is not necessarily a first-move matrix. Investigating whether Theorem 1 holds under the additional condition that $\mathbf{m}$ is a first-move matrix is left as future work.

6 Heuristics

When applying our method, there is a trade-off between the size of the resulting CPD and the first-k-moves lag, as mentioned earlier. We present heuristics aimed at balancing and controlling the trade-off. The desired behavior depends in part on the application domain. For instance, in domains where full paths are required and the first-k-moves lag is less important, one could aim at minimizing the CPD size.

We discuss two types of heuristics, which are orthogonal and can be combined. The first type refers to heuristic node orderings to be used as a wildcard ordering. Heuristics in the second category are about using only a subset of the available wildcards, to reduce the first-k-moves lag.

Heuristic wildcard orderings. New heuristics used in our experiments are an enhancement of the *CUT node ordering heuristic* and the *DFS node ordering heuristic*, which have been introduced by Strasser, Harabor, and Botea (2014). In that previous work, the CUT and the DFS heuristic are used to order the columns of the first-move matrix. In contrast, in our work we evaluate CUT and DFS as wildcard ordering heuristics. Column orderings and our new wildcard orderings are two independent and orthogonal enhancements to CPDs.

CUT and DFS are designed to capture the intuition that nodes close to each other in the graph should come close to each other in the heuristic ordering. The CUT heuristic computes the ordering recursively. It divides the graph into two subgraphs, and requires that all nodes in one subgraph come in the heuristic ordering before all nodes in the other subgraph. The process continues recursively. DFS orders the graph nodes in the order in which the nodes are visited for the first time in a depth-first traversal of the graph.

As shown in the experiments section, when used as wildcard orderings, the CUT and the DFS heuristics achieve a good performance in terms of CPD size. However, the first-move lag increases on average. A closer look at these heuristics explains this behavior. Consider a current source node s and a current target t with the property that $s > t$ in the wildcard ordering. This means that a backwards move is available. Let t' be the new target after performing the backwards move. Recall that the CUT and DFS heuristics capture the intuitive idea that nodes close to each other in the graph should be close to each other in the heuristic ordering. Thus, as t and t' are neighbors in the graph, chances are they are close to each other in the heuristic ordering. When $s > t$ and t and t' are close to each other in the heuristic ordering, chances are that $s > t'$, which means that a backwards move is available from t' towards s. Applying this argument recursively, it could be the case that a relatively long series of backwards moves are obtained before getting the first move forward. This explains why DFS and CUT tend to have an increased first-move lag.

Part of the heuristics presented in the rest of this section aim at reducing the first-move lag of CUT and DFS. The *combined CUT-random heuristic* CUT-rnd(k) starts from the CUT ordering and swaps the ordering of k pairs of nodes picked at random, where k is a parameter. The *combined DFS-random heuristic* DFS-rnd(k) is similar, except it is based on DFS. We have also used an *uniformly random wildcard ordering*, as a baseline approach.

Varying the number of wildcards in use. As argued earlier in this section, the CUT and the DFS node orderings could lead to long chains of backwards moves, increasing the first-move lag. For this reason we introduce the second type of heuristics, which control the number of used wild-

Algorithm 1: TK-heuristic

```
input: s source, t target, r threshold, k K-factor
if t before s in the wildcard ordering then
    d ← 100 × |s − t|/n;
    if d > r and t is multiple of k then
        encode real move
    else
        encode wildcard
else
    encode real move
```

cards. Their purpose is to break a chain of backwards moves as early as possible, decreasing the first-k-moves lag.

The idea is to skip the application of some wildcards, leaving in place some pairs (s, t) with moves available in both directions. In such cases, we are free to prefer the forward move, reducing the first-k-moves lag.

Our *TK-heuristic* TK(r, k) is used to decide what wildcards should be skipped. The name comes from its two parameters, the threshold r and the K-factor k. The threshold r is given as a percentage of the number of nodes, and k is an integer. Consider an entry $\mathbf{m}[s, t]$ that would normally be replaced with a wildcard, according to a wildcard ordering. If, in the wildcard ordering, the difference between the indexes of a current source node s and a target node t exceeds r, and the index of t is a multiple of k, then do not replace the current first-move symbol with a wildcard. Otherwise, apply the method as before (i.e., replace the symbol with a wildcard). See Algorithm 1 for the pseudocode.

The two parameters control how many wildcards to skip. The larger r, the fewer wildcards are skipped. In particular, when r grows to 100%, no wildcard is skipped, and thus the TK-heuristic reduces to our original wildcard substitution approach. Likewise, the larger k, the fewer wildcards are skipped. When $k > n$, no wildcards are skipped.

7 Experiments

In this section, we present the results of our experimental analysis with the aim of evaluating the effectiveness of using wildcard substitutions. The test data consist of 8-connected gridmaps ranging from about 2,000 to 750,000 nodes (Sturtevant 2012). We used 185 real game maps, from games such as *Dragon Age: Origins*, *Starcraft* and *Baldur's Gate II*. Each map comes with a set of shortest-path queries. Maps are undirected. Straight edges have a weight of 1 and diagonal edges have a weight of $\sqrt{2}$.

The experiments were conducted using an Intel Xeon® 6-core 3.46GHz CPU running Red Hat 6.8. The original SRC code is provided by its authors. All algorithms are compiled using g++ 5.4.0 with -O3.

Table 1 compares the performance of the heuristic wildcard orderings with respect to using no wildcard substitutions. We evaluate the CPD size, the first-move lag, the first-20-moves lag, and the speed to compute a full path. The first-20-moves lag is borrowed from the Grid-based Path Planning Competition (GPPC). Our method has virtually no overhead in terms of preprocessing time on top of SRC.

CUT Column Ordering Heuristic				
Wildcard heuristic	Size [%]	Lookups for 1 fwd move	Lookups for 20 fwd moves	Full path time [μs]
None	100	1	20	17.03
CUT	49.56	84.37	98.32	14.61
CUT + TK(10, 2)	60.40	5.51	22.65	16.60
CUT + TK(10, 5)	57.97	5.95	23.79	16.38
CUT + TK(10, 10)	56.52	8.12	25.67	16.19
CUT + TK(0, 5)	77.92	2.78	20.11	17.68
CUT + TK(0, 10)	71.74	4.87	21.93	17.26
CUT + TK(0, 100)	57.88	27.27	42.82	16.00
CUT-Rnd(5)	59.09	35.76	64.22	16.09
CUT-Rnd(10)	62.04	25.65	60.29	16.57
CUT-Rnd(100)	70.72	7.59	54.14	18.60
Random	71.55	5.15	53.54	16.56
DFS Column Ordering Heuristic				
Wildcard heuristic	Size [%]	Lookups for 1 fwd move	Lookups for 20 fwd moves	Full path time [μs]
None	100	1	20	12.64
DFS	51.93	84.10	98.11	10.43
DFS + TK(10, 2)	66.64	5.51	23.65	11.99
DFS + TK(10, 5)	64.70	6.80	24.72	11.93
DFS + TK(10, 10)	63.19	9.02	26.65	11.84
DFS + TK(0, 5)	82.93	2.72	20.06	12.76
DFS + TK(0, 10)	78.04	4.74	21.85	12.68
DFS + TK(0, 100)	61.27	29.39	45.59	14.67
DFS-Rnd(5)	62.02	36.22	64.32	12.33
DFS-Rnd(10)	65.78	25.58	60.10	12.66
DFS-Rnd(100)	75.53	7.42	53.10	13.60
Random	76.93	5.27	52.60	13.21

Table 1: Results averaged over all maps.

The entry with no wildcard heuristic (label "None") stands for the original system SRC (Strasser, Harabor, and Botea 2014). Every other entry represents our system, implemented on top of SRC, with a given wildcard heuristic. In the top half of the table, the CUT heuristic is used to order the columns of the first move matrix. In the bottom half, the column ordering heuristic is DFS.

Column 2 in Table 1 shows the average CPD size as a percentage of the original average CPD. Wildcard heuristics such as CUT and DFS reduce the size by a factor of 2 in average, a significant memory improvement. On the other hand, the measured first-k-moves lags (i.e., $k = 1$ and $k = 20$) increase significantly for CUT and DFS (columns 3 and 4). Using the TK heuristic helps balance this tradeoff. Take for instance CUT + TK(10,5). It cuts the CPD size by about 42%, and reduces the first-k-moves lags significantly in comparison to just CUT.

In the rest of the section, TK(r, k) is a shortcut for CUT + TK(r, k) and DFS + TK(r, k). When two TK combinations are directly compared, they are both from the same half of the table (top or bottom half). TK$(r, *)$ denotes all the TK heuristics tested using threshold r and any value of K factor.

Overall, CUT, DFS, and TK$(10, *)$ appear to be among the best heuristics that we tested. We call these the *star heuristics*. When minimizing the CPD size is more important, use CUT or DFS as wildcard ordering heuristics.

When the first-k-moves lags matter too, heuristic TK$(10, *)$ achieves a good performance, reducing the CPD size significantly, and maintaining the lags within competitive limits. For TK$(10, *)$, the first-20-moves lag reduces to values slightly larger than the 20 optimal value. The first-move lag stays within less than 10 moves before the first forward move is available. This too is a very good value in comparison to modern systems based on graph search. For instance, the GPPC results show that SRC (called SRC-dfs-i in the competition report (Sturtevant et al. 2015)) is orders of magnitude faster in terms of the first-20-moves time and the max segment time (max time per move) than other optimal solvers (Sturtevant et al. 2015). As explained in the related work section, the reason is that solvers based on graph search need to reach the target (i.e., complete the search) before knowing the first optimal move. On the other hand, the overheads of our TK(10,*) heuristic are much smaller, being less than a factor of 1.4 for the first-20-moves lag and less than a factor of 10 for the first-move lag (cf. Table 1).

Figure 4 illustrates the CPD size reduction for a representative subset of maps (*Dragon Age: Origins* maps), to avoid clutter. Figure 5 illustrates the first-move lag on all maps. Notice how TK improves the first-move lag of CUT, reducing it from increasingly large values to small and stable values. To save room and avoid clutter, we plotted a few heuristics from the top half of Table 1 (i.e., programs using the CUT column ordering.) Using DFS for column ordering leads to a similar behavior.

Interestingly, using our wildcard technique can slightly improve the full path time (column 5 in Table 1). In particular, this is true about the star heuristics. The reason is that, with a smaller CPD in use, the rate of CPU cache misses could be smaller. Also, as compressed matrix rows are shorter, the binary search needed to extract an optimal move could be slightly faster.

MRC (Strasser, Harabor, and Botea 2014) can also achieve some memory savings on top of SRC. However, MRC's savings are more modest, being smaller than 15%, as shown in the original work. Compared to SRC, MRC increases the time per move and the full path time by 25% or more (Strasser, Harabor, and Botea 2014).

Star heuristics dominate other heuristics evaluated in our tests. For example, TK$(10, 5)$ dominates the CUT-Rnd heuristics in terms of both size reduction and first-move(s) lag. TK$(10, 5)$ achieves similar size reductions as DFS-Rnd, but the former has better first-move(s) lag performance. TK$(10, 5)$ and TK$(10, 2)$ are similar to Random in terms of first-move(s) lags, but the latter creates larger CPDs. In terms of CPD size, TK$(10, 5)$ is similar to TK$(0, 100)$, but the former has a better first-move(s) lag performance. Size-wise, TK$(10, 5)$ is much better than TK$(0, 5)$ and TK$(0, 10)$, while the first-move(s) lags are comparable.

Table 1 shows that the baseline program SRC is slower on average with the CUT column ordering heuristic, as opposed to the DFS heuristic. A closer look at the data shows that this is due to the Starcraft domain. In the other domains SRC with CUT is generally faster than SRC with DFS. A deeper analysis is beyond our focus, as this refers to the behaviour of the baseline program SRC.

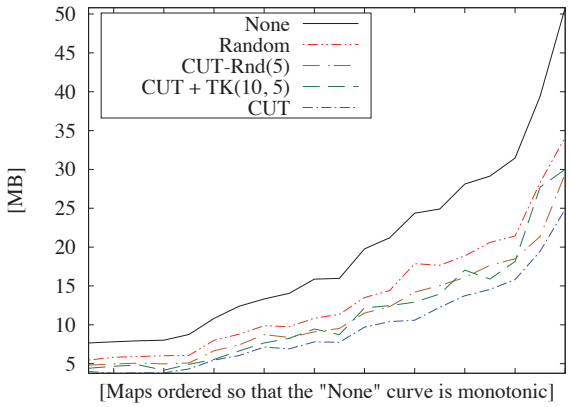


Figure 4: Impact on the CPD size.

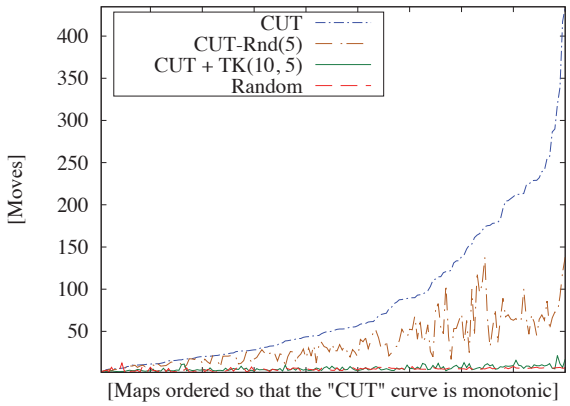


Figure 5: Impact on the first-move lag.

8 Conclusion

Path planning is a core AI problem, with applications in domains such as robotics and games. CPD-based path planning systems, such as SRC (Strasser, Harabor, and Botea 2014), achieve an impressive speed performance, in terms of computing both full optimal paths and a first fragment of an optimal path. However, large maps can lead to large CPDs, which can represent a performance bottleneck.

In this work we have focused on reducing the size of a CPD. Our contributions are algorithmic (a new technique), theoretical and experimental. As the main theoretical contribution, we proved that applying our wildcard idea in a way that maximizes the memory savings is NP complete. We implemented our ideas on top of the state-of-the-art SRC system. The experiments demonstrate the benefits of our technique. We reduce the CPD size significantly. The time to compute full optimal paths can improve slightly. The first-k-moves lag is kept within competitive values.

In future work, we would consider graphs with mixed directed and undirected edges. We plan to investigate new heuristics to further improve the performance and the scalability to larger maps. As mentioned earlier, extending our theoretical analysis is another interesting direction for future work.

9 Acknowledgment

This work has been performed when the first author was an intern at IBM Research, Ireland. We thank Jakub Marecek and Jussi Rintanen for their comments on parts of this work.

References

Botea, A., and Harabor, D. 2013. Path planning with compressed all-pairs shortest paths data. In *Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS-13*, 288–292.

Botea, A.; Strasser, B.; and Harabor, D. 2015. Complexity Results for Compressing Optimal Paths. In *Proceedings of the 29th National Conference on AI, AAAI-15*, 1100–1106.

Botea, A. 2011. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the 7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 122–127.

Botea, A. 2012. Fast, optimal pathfinding with compressed path databases. In *Proceedings of the Symposium on Combinatorial Search, SoCS-12*, 204–205.

Harabor, D. D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-11*, 1114–1119. AAAI Press.

Rabin, S., and Sturtevant, N. 2016. Combining bounding boxes and jps to prune grid pathfinding. In *Proceedings of the 30-th AAAI Conference on Artificial Intelligence, AAAI-16*, 746–752.

Sankaranarayanan, J.; Alborzi, H.; and Samet, H. 2005. Efficient query processing on spatial networks. In *ACM workshop on Geographic information systems*, 200–209.

Strasser, B.; Botea, A.; and Harabor, D. 2015. Compressing optimal paths with run length encoding. *Journal of Artificial Intelligence Research, JAIR* 54:593–629.

Strasser, B.; Harabor, D.; and Botea, A. 2014. Fast First-Move Queries through Run Length Encoding. In *Proceedings of the 5th International Symposium on Combinatorial Search, SoCS-14*, 157–165.

Sturtevant, N. R.; Traish, J. M.; Tulip, J. R.; Uras, T.; Koenig, S.; Strasser, B.; Botea, A.; Harabor, D.; and Rabin, S. 2015. The grid-based path planning competition: 2014 entries and results. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS-15*, 241–251.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144–148.

Takanori, A.; Takao, N.; and Nobuji, S. 1980. NP-Completeness of the Hamiltonian Cycle Problem for Bipartite Graphs. *Journal of information processing* 3(2):73–76.

Uras, T.; Koenig, S.; and Hernández, C. 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the 23-rd International Conference on Automated Planning and Scheduling, ICAPS-13*, 224–232.