Cutting the Size of Compressed Path Databases With Wildcards and Redundant Symbols

Mattia Chiari
m.chiari017@studenti.unibs.it
University of Brescia

Shizhe Zhao
shizhe.zhao@monash.edu
Monash University

Adi Botea
adibotea@ie.ibm.com
IBM Research, Ireland

Alfonso E. Gerevini
alfonso.gerevini@unibs.it
University of Brescia

Daniel Harabor
daniel.harabor@monash.edu
Monash University

Alessandro Saetti
alessandro.saetti@unibs.it
University of Brescia

Matteo Salvetti
m.salvetti014@studenti.unibs.it
University of Brescia

Peter J. Stuckey
peter.stuckey@monash.edu
Monash University

Abstract

Path planning on gridmaps is a common problem in AI and a popular topic in application areas such as computer games. Compressed Path Databases (CPDs) represent a state-of-the-art approach to the problem, in terms of the speed of computing full optimal paths and also individual optimal moves. Despite significant improvements in recent years, the memory required to store a CPD can still be a bottleneck for large game maps. In this work we present a new compression approach that can reduce the size of CPDs. Our approach uses an extended notion of wildcards and a novel concept called a redundant symbol. We implement our ideas on top of a state-of-the-art CPD system and, in a range of experiments, we demonstrate a substantial reduction in the size of CPDs.

1 Introduction

Path planning is an important and long studied problem in AI, and it is a problem which finds common application in different real-world settings such as robotics and computer games. When the problem appears in practice, it is often assumed that the input environment can be modelled as a two-dimensional *gridmap* that is made up of traversable and non-traversable cells. Despite significant improvements in the recent literature, path planning on gridmaps remains an active area of research. This is demonstrated, for instance, by the interest shown in the Grid-based Path Planning Competition GPPC (Sturtevant et al. 2015).

Compressed Path Databases (CPDs) (Botea 2011) are a state-of-the-art approach (in terms of speed) for optimal pathfinding on gridmaps (Sturtevant et al. 2015; Salvetti et al. 2018). Each CPD is simply a data structure that provides an optimal first move: from any cell s towards any cell t of the gridmap. Created during an offline preprocessing step, and exploited during a subsequent online phase, CPDs can be used to compute shortest paths quickly. This is done by looking up which is the next optimal first move on the way to the target and executing that move; a process which is repeated until the target is reached. CPDs can also be used to quickly provide any prefix of an optimal path. This is important to reduce the so-called first-move lag, where an agent needs to wait until it knows in which direction to move. Many other pathfinding techniques for example, including search-based methods derived from A* (Hart, Nilsson, and Raphael 1968) know the first optimal move only when they know the entire solution. By contrast, CPDs identify such moves faster and independently from the rest of the path.

A main drawback of CPDs is that preprocessed data can sometimes be prohibitively large — even after considering recent efforts to reduce its size; e.g. (Strasser, Harabor, and Botea 2014; Salvetti et al. 2017). In this work we seek to address the issue with two new compression ideas that can substantially reduce the size of existing CPDs:

- *Redundant symbols* are a generic concept that allow us to represent a single first-move with two or more symbols. The additional flexibility allows us to more effectively compress first-move data.
- *Proximity wildcards* are "don't care" symbols which allow us to avoid storing first-move data in cases where such data can be efficiently re-computed online.

We undertake a detailed evaluation on gridmaps from Sturtevant's well known benchmark sets (Sturtevant 2012). Results indicate substantial reductions CPD size, from several factors to over one order of magnitude. Because CPD sizes are much smaller, we also report a speedup in the time required to extract optimal shortest paths.

The rest of this paper is organised as follows. In the next section, we present background material on CPDs and summarise existing work in the area. In Section 3, we introduce the redundant heuristic symbol and show how this can help compress CPDs. In Section 4, we show how we can use proximity wild cards to help compress CPDs. In Section 5, we explain the experimental setup and give experimental results. Finally, in Section 6 we give the conclusions and mention future work.

2 Background

We present background information on gridmaps, compressed path databases and wildcards, that are necessary to understand the new contributions of this paper.

Gridmaps. A gridmap is a rectangular grid where each cell is either fully traversable or fully blocked. Gridmaps represent a popular way to represent the navigation environment of a mobile agent, such as a robot or a game character.

The agent can occupy exactly one traversable cell at a time. The time is discretised and, at a given time step, the agent can move to an adjacent traversable cell. 4-connected grids and 8-connected grids correspond to two common ways to define neighbouring relations. In 4-connected grids, a traversable cell has (at most) 4 neighbours: North, South, East and West. The cost of each such a transition is 1. 8-connected grids, also known as Octile gridmaps, allow diagonal transitions (NE, SE, SW, NW), with a cost of $\sqrt{2}$ each, in addition to the 4 straight transitions mentioned. In this work, we assume that diagonal moves are disallowed if they would touch an obstacle cell. We use the major compass directions (N, S, SW, SE etc.) to refer to the corresponding transitions from a given cell. Given a cell n in a gridmap, we write $n.x$ and $n.y$ for the x and y coordinates of that cell.

Each gridmap induces a graph where traversable cells are nodes, and neighbouring relations are edges. Consider a weighted graph $G = (V, E)$ with vertices V and edges $E \subseteq V \times V$, and edge length function w such that $w(s, t)$ is the weight of edge $(s, t) \in E$. A *path* from s to t in G is a sequence of edges (n_0, n_1), (n_1, n_2), $\ldots$, (n_{k-1}, n_k), where $k \in \mathbb{N}^+$, $n_0 = s$, and $n_k = t$. The length of the path is $\sum_{i=0}^{k-1} w(n_i, n_{i+1})$.

Compressed Path Databases (CPDs). A CPD (Botea 2011) encodes optimal first edges from any node s towards any node t on a graph. More formally, we say that a "move" from s towards t is the first edge of a path from s to t. An optimal move is the first edge of an optimal path. In this paper, we use the terms "move" and "edge interchangeably. Likewise for "cell" and "node" and also "gridmap" and "graph".

Building a compressed database requires a series of iterations. Each iteration runs the Dijkstra algorithm using a distinct node s as a source (root of the search). A slight modification of the Dijkstra algorithm produces a so-called first-move array $T(s)$, using s as source. In a first-move array, all nodes t reachable from s are assigned a first-move label that identifies all moves leaving s that start a shortest path towards t. The first-move array $T(s)$ is compressed, which concludes the iteration at hand. Being independent, the iterations can be run in parallel, with a speed-up linear in the number of available processors.

Example 1. Consider the gridmap shown in Figure 1. We illustrate an iteration of the preprocessing that builds the CPD. In our example, Dijkstra is run from the source node s. For each traversable cell we now specify all the optimal first moves, from s to the node at hand. For example, two optimal first moves exist from s towards the bottom-left corner: W and SW. Compressing the set of first-moves requires a fixed ordering of the graph nodes. In our example we assume that the nodes are ordered left to right and top to bottom.

Following the approach of Strasser, Harabor, and Botea (2014), the string of symbols is encoded with run-length encoding (RLE). RLE compresses a string of symbols by representing more compactly substrings, called runs, consisting of repetitions of the same symbol. E.g., the substring W; W; W; (W,E); E; E; E (the first row in the figure) can have two runs, namely WWWW, and EEE. We replace each such run by a pair of values: one value indicates the starting index (where the run begins) and the other value stores the associated symbol. With RLE the example substring can be represented more efficiently as 1W; 5E. Observe how we are free to choose any symbol from a non-singleton list such as (W,E). Overall, the entire string is compressed into 11 runs: 1W 5E 8W 12E 15W 20E 22W 26E 29SW 32S 33SE. Note that obstacle nodes and the source are assigned wildcard symbols "$*$"; i.e., "don't care" symbols, because we never need to look up a move from s to any of these. This concludes one iteration of CPD preprocessing.

W	W	W	W,E	E	E	E
W	W	W	W,E	E	E	E
W	W				E	E
W	W	W	s	E	E	E
W,SW	W,SW	SW	S	SE	E,SE	E,SE

Figure 1: The optimal first moves to each cell of the gridmap from the cell marked s. Black cells indicate obstacles.

Once preprocessing is complete the resulting CPD can be used to look up optimal moves by performing a binary search on the compressed string of any given source node. E.g., to look up the optimal move from s to (6,2), which is at index 13, we start with end pointers $(l, u) = (1, 11)$ and look up middle point $m = 6$, which is the entry 20E. As $13 < 20$, the binary search continues to the left, setting $u = 5$. Next we look up $m = 3$ (the 8W entry in the compressed string) and since $13 > 8$, we continue to the right, setting $l = 4$. Next we look up $m = 4$ (entry 12E) and set $l = 5$. Next we look up $m = 5$ (the 15W entry) and set $u = 4$. The search ends because $u < l$. For the final middle point $m = 5$, the entry is 15W, and since $13 < 15$ the search returns move E of the fourth entry 12E.

We denote as $\mathsf{CPD}(s,t)$ a function which returns a first move from s which starts some optimal shortest path from s to t. As shown in the previous example, the function requires a binary search through a compressed string of symbols. We can retrieve a shortest path by simply repeatedly applying the CPD to find the next move to make. The procedure for the extraction of the shortest path is shown in Algorithm 1.

Algorithm 1: Unidirectional path extraction at runtime for an (s, t) pair. Symbol $+$ denotes concatenation.

```
p ← []
while s ≠ t do
    (s, n) ← CPD(s, t)
    p ← p + [(s, n)]
    s ← n
return p
```

In practice, the size of the CPD can be reduced by choos-

Algorithm 2: Bidirectional path extraction at runtime for an (s, t) pair

```
prefix ← []
suffix ← []
while s ≠ t do
    if (s, t) ∈ R then
        (s, n) ← CPD(s, t)
        prefix ← prefix + [(s, n)]
        s ← n
    else
        (t, n) ← CPD(t,s)
        suffix ← [(n, t)] + suffix
        t ← n
return prefix + suffix
```

ing a *column ordering*, that is the order in which nodes appear in the run length encoding. For simplicity, in the running examples we will use a default order (top to bottom, left to right). However, in the experiments we use state-of-the-art heuristic orderings introduced in previous work (Strasser, Harabor, and Botea 2014). These have been shown to outperform naive orderings in terms of memory by up to a factor of 10 (Strasser, Botea, and Harabor 2015).

Bidirectional Wildcards. Salvetti et al. (2017) show how we can improve the compression in a CPD for undirected graphs by taking into account the duplicate information. In order to find a shortest path from s to t we only need to know either (a) the first move from s towards t, or (b) the first move from t towards s. We do not need both.

Assume a realised relation $R \subseteq V \times V$ which defines which pairs of (s, t) are realised correctly in the CPD. For the completeness of path finding we simply require that $\forall s \in V, t \in V, s \neq t \rightarrow ((s, t) \in R \vee (t, s) \in R)$.

Given this information we can extract a shortest path from s to t using a bidirectional search shown in Algorithm 2.

In practice the realised relation R is based on a node ordering $\prec$, so that $(s, t) \in R \Leftrightarrow s \prec t$.

The advantage of the realised relation is that if $(s, t) \notin R$ then we will never access the t entry in $L(n)$. Hence we can replace the entry in T the first moves array for s, $T(t)$ with wildcard entries "$*$". This allows to create more compressed run length encodings.

3 Heuristic Redundant Symbols

In many cases in navigating a grid map, the first move to be taken from a start s to target t is the "obvious move" that heads in the direction towards the target. Rather than record the first optimal move in the CPD, we can just record that the move is "obvious". To do so we introduce a new *heuristic redundant symbol* ⓗ to represent such moves.

Given nodes s and t we can define a *default move* from s to t, $d(s, t)$ as follows. Let $A = [NW, N, NE, W, \mathit{null}, E, SW, S, SE]$ be the array of directions indexed from 1 to 9. Let $\sigma x = \text{sign}(t.x - s.x)$, and $\sigma y = \text{sign}(t.y - s.y)$, where $\text{sign}(x)$ is -1 if $x < 0$, 0 if $x = 0$, and $+1$ if $x > 0$; then $d(s, t) = A[\sigma x + 3 \times \sigma y + 5]$. Essentially, the default move is just the move that leads us closest to the target, according to a heuristic distance function like, e.g., the Euclidean or the octile distance.

NW	NW	NW	N	NE	NE	NE
NW	NW	NW	N	NE	NE	NE
NW	NW	■	■	■	NE	NE
W	**W**	**W**	s	**E**	**E**	**E**
SW	**SW**	**SW**	**S**	**SE**	**SE**	**SE**

Figure 2: The default first move $d(s, t)$ to each cell of the gridmap. Bold if it appears in $T(t)$ for the cell marked s.

When the default move from s to some node t is one of the optimal first moves from s to t, i.e., if $d(s, t) \in T(t)$ where T is the first move array for s, then we can encode this information in the CPD using the new symbol ⓗ which represents that we follow the heuristic of default moves. Specifically, we add to $T(t)$ the new symbol ⓗ, in addition to the existing contents of $T(t)$. This way, the compression step will have more options to choose what symbol to keep from $T(t)$ in the compressed string, with a potentially better compression in the end.

Example 2. Consider the gridmap shown in Figure 2. For each node t, we show the default move bolded if it appears in $T(t)$ for s, that is the set of the optimal first moves sketched in Figure 1. As we can see, almost half of the graph can be encoded using the default moves. Once we add the heuristic move symbol ⓗ to each entry $T(t)$ where the move is bolded, we can perform a better run length encoding. The encoding using the new heuristic move symbol is 1W 5E 8W 12E 15W 20E 22ⓗ, which is smaller than without using ⓗ (i.e., 7 runs in the new encoding, compared to 11 in Example 1).

The heuristic move symbol is able to encode large parts of the graph which are close to the source, and many other parts as well.

3.1 Distance Functions

The default move is quite basic, it takes no account of information about the surroundings of s, and indeed in many cases the direction returned may not even be possible. We can improve the use of the heuristic move symbol ⓗ by considering the relationship between s and t in more detail.

Let s be the source, t be a target, let $f_x(s, t)$ be a predefined heuristic distance function. We assume $f_x(s, t)$ is simple to calculate. For our examples we will use the octile distance function

$$f_o(s,t) = \sqrt{2} \times c + |s.x - t.x| - c + |s.y - t.y| - c \quad \text{where } c = \min(|s.x - t.x|, |s.y - t.y|)$$

or the Euclidean distance function

W	**W**	**W**	**E**	**E**	**E**	**E**
W	**W**	**W**	**E**	**E**	**E**	**E**
W	**W**	**W**	**E**	**E**	**E**	**E**
W	**W**				**E**	**E**
W	**W**	**W**	s	**E**	**E**	**E**
SW	**SW**	**SW**	**S**	**SE**	**SE**	**SE**

Figure 3: The heuristic first move $F_o(s,t)$ to each cell of the gridmap from the cell marked s. All symbols are in bold, since each of them belongs to the corresponding set $T(t)$ for the marked symbol s.

$$f_e(s,t) = \sqrt{(s.x - t.x)^2 + (s.y - t.y)^2}.$$

We define the move chosen by the distance function $F_x(s,t)$ from s to t as the move leaving s to n such that the estimated path distance through n, $w(s,n) + f_x(n,t)$, is minimised, that is

$$F_x(s,t) = \arg\min_{(s,n)\in E}\{w(s,n) + f_x(n,t)\}.$$

In order for ⓗ to be used in the CPD, it must be unambiguous, hence we assume a total order on all moves leaving s, and assume the $\arg\min$ in F returns the least move in this order that leads to the minimal distance. For gridmaps our default ordering is NE, NW, SE, SW, N, S, E, W. Effectively this tries to take diagonal moves first when there is a tie.

The move chosen by the distance function is immediately better than the default move approach because it can never choose invalid grid moves. Just as before, if $F_x(s,t)$ returns a first move that corresponds with the start of the shortest path from s to t, then we can add the symbol ⓗ to the set $T(t)$ of possible first moves for s.

Example 3. Consider the gridmap in Figure 1. Figure 3 shows the heuristic move to each node in the graph underlying the gridmap. Since for each target node t, $F_o(s,t) \in T(t)$ for s, we can add a heuristic move symbol ⓗ to each entry in $T(t)$. Now when we perform run length encoding we get the result 1ⓗ. That is, the entire compressed string has only 1 run, which is substantially shorter than options discussed in Examples 1 and 2.

The revised CPD lookup function is shown in Algorithm 3. It simply looks up the CPD as usual but, if it finds a symbol ⓗ, it returns the heuristic move $F_x(s,t)$, for whichever version we choose to use.

3.2 Improving the Tie-breaking

The heuristic move must be unambiguous, hence when there are moves that look equally good we must choose one of them unambiguously. This choice can be different for each source node s. While the diagonal first ordering is a good default, we can make use of the relative positions of s and t to have better tie breaking.

Algorithm 3: Get next move algorithm for CPDs with heuristic moves.

```
1 CPDH(s, t)
2 m ← CPD(s,t)
3 if m = ⓗ then
4 |   return F(s, t)
5 else
6 |   return m
```

s	E	E	E	E	E
S	SE	SE		SE	SE
S	SE		SE	SE	SE
S	SE	SE	SE	SE	SE
S	SE	SE	SE	SE	SE

(a)

s	**E**	**E**	**E**	**E**	**E**
S	**SE**	**E,SE**		E	E
S	**S,SE**		**S,SE**	E	E
S	**S,SE**	**S,SE**	**S,SE**	**S,SE**	**S,SE**
S	**S,SE**	**S,SE**	**S,SE**	**S,SE**	**S,SE**

(b)

Figure 4: (a) The heuristic first move $F_o(s,t)$ to each cell of the gridmap from the cell marked s and (b) the optimal first moves to each cell from s, with the heuristic moves in bold.

Example 4. Consider the gridmap in Figure 4. We show the heuristic move $F_o(s,t)$ to each cell of the gridmap in Figure 4(a), and the optimal first moves in Figure 4(b). Clearly all nodes can make use of a heuristic move, except those marked E on the second and third row. The resulting encoding is 1ⓗ 12E 13ⓗ 17E 19ⓗ. This pattern is common where some part of the graph is blocked. We want to have a better tie breaker for such circumstances.

When we have blockages in the graph it can often be the case that the heuristic best move is not the diagonal move. We improve our heuristic tie breaking by breaking the graph into 8 quadrants around s, corresponding to the 8 directions, and break ties by choosing the closest direction to the straight line direction from s to t. In order to avoid complex trigonometric operations we use simple approximations: if $|s.x - t.x| \geq 2|s.y - t.y|$ then the straight line from s to t is close to the horizontal line, so we favour E or W, similarly if $|s.y - t.y| \geq 2|s.x - t.x|$ then the straight line from s to t is close to vertical so we favour N or S. In other cases we favour the closest diagonal direction. We denote the heuristic move with directional tie breaking by $F^d_x(s,t)$ where x denotes the distance function.

Example 5. Consider the gridmap in Figure 4 once more. Using direction-based tie breaking the heuristic moves are given in Figure 5. Note that now each heuristic move appears in the optimal first moves (Figure 4(b)), and hence we can encode all the first moves as 1ⓗ, that is much shorter than without using directional-based tie breaking.

4 Proximity Wildcards

In the previous section, we have used heuristic moves to introduce redundant symbols in CPDs, for an improved com-

s	E	E	E	E	E
S	SE	E	■	E	E
S	S	■	SE	E	E
S	S	SE	SE	SE	SE
S	S	S	SE	SE	SE

Figure 5: The heuristic first move $F_o^d(s,t)$ to each cell of the gridmap from the cell marked s using directional tie breaking.

pression. In this section we further exploit the potential of heuristic moves by focusing on proximity areas around source nodes s. Often, within an open area around a node s, all the nodes will be optimally reachable using the heuristic move as a first move. To take advantage of this we introduce a new level of compression, using *proximity wildcards*.

Definition 1. *Given a node s and a function $F_x(s,n)$, the* proximity distance $\mathtt{pd}(s)$ *is the lowest value $d \in \mathbb{N}$ such that there exists a cell n for which $|s.x - n.x| \leq d+1$ or $|s.y - n.y| \leq d+1$, and $F_x(s,n) \notin T(n)$ for s, where T is the first-move array for s.*

In other words, for all cells n such that $|s.x - n.x| \leq \mathtt{pd}(s)$ and $|s.y - n.y| \leq \mathtt{pd}(s)$, we have that ⓗ $\in T(n)$ for s. For simplicity, in the rest of the paper the proximity wildcards are determined by using $F_x = f_o$, i.e., using the octile distance function.

Definition 2. *The* proximity square *of a node s is the square centred in s and with the edge size equal to $2 \cdot \mathtt{pd}(s) + 1$.*

That is, the proximity square is the largest square centred on s such that the cells of the gridmap inside the square can optimally be reached from s by the heuristic move ⓗ.*

We can then replace the entries in T for the nodes inside a proximity square with a wildcard symbol "$*$", and use the distance $\mathtt{pd}(s)$ instead to determine the move for nodes n within the proximity square. Wildcards obtained this way are called proximity wildcards. To be able to use proximity wildcards in the CPD, we change the lookup function to always use the heuristic move when the target is within the proximity square of s.

Example 6. Consider the gridmap shown in Figure 6, with 8 cells labelled from a to h. Figure 7 illustrates, for each source cell (node), the areas on the map where heuristic moves coincide with optimal moves. As mentioned, heuristic moves are computed with $f_o(s,t)$.

Using Definitions 1 and 2, it follows that, in Figure 7, the proximity square of a source node may contain the source cell, cells in bold, obstacles, and areas outside the map. The proximity square cannot contain (white) cells in normal font. This remark allows us to calculate, for each source node s, the value $\mathtt{pd}(s)$. We show the values in Table 1.

*We can keep the definition simple and still use a square in those cases when s is close to the border of the gridmap, by simply allowing the proximity square to partly go outside the gridmap.

a	b	c
d	e	■
f	g	h

Figure 6: A small gridmap to illustrate proximity wildcards.

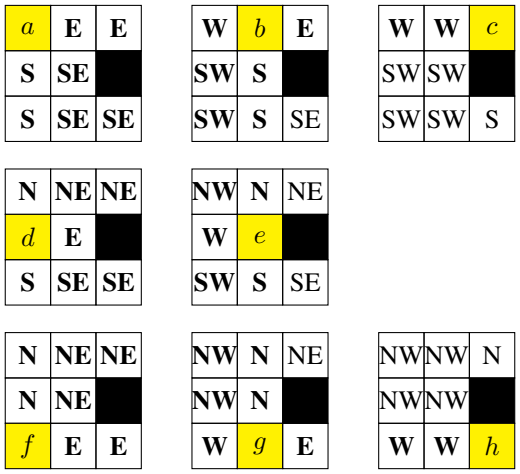

Figure 7: Heuristic moves, for each source cell. Source cells are shown in yellow. In each case, heuristic moves that coincide with optimal moves are shown in bold.

So far we have computed the proximity squares of each node. Next we illustrate how these are used to introduce proximity wildcards in the first-move table. Table 2 shows the first-move table of our running example before adding the proximity wildcards. This is compressed into 1ⓗ (first row); 1ⓗ 9S (second row); 1W (third row); 1ⓗ (fourth row); 1ⓗ 3N 4ⓗ 9S (fifth row); 1ⓗ (row 6); 1N 4ⓗ (row 7); 1W (last row). This sums up to 13 RLE runs.

Then, for each row, we replace all entries within the proximity square of the source node at hand with wildcards, since we do not need to use the CPD to look up their first move. The resulting first-move table is shown in Table 3. The resulting run length encodings are: 1*; 1S; 1W; 1*; 1ⓗ 3N 4ⓗ 9S; 1*; 1N; and 1W. This gives a total of 11 RLE runs. This concludes our example.

The revised CPD lookup function using proximity squares is shown in Algorithm 4. The first advantage is that for lookups close to the source s we may not need to perform logarithmic run length decoding. The second advantage is that since many more wildcards are added to the first move array, the run length encoding can be much smaller.

Node s	a	b	c	d	e	f	g	h
$\mathtt{pd}(s)$	2	1	0	2	0	2	1	0

Table 1: Proximity distances for the nodes in the gridmap in Figure 6.

	a	b	c	d	e		f	g	h
a	$*$	ⓗ, E	ⓗ, E	ⓗ, S	ⓗ, SE	$*$	ⓗ, S	ⓗ, S, SE	ⓗ, S, SE
b	ⓗ, W	$*$	ⓗ, E	ⓗ, SW	ⓗ, S	$*$	ⓗ, S, SW	ⓗ, S	S
c	ⓗ, W	ⓗ, W	$*$	W	W	$*$	W	W	W
d	ⓗ, N	ⓗ, NE	ⓗ, NE	$*$	ⓗ, E	$*$	ⓗ, S	ⓗ, SE	ⓗ, SE
e	ⓗ, NW	ⓗ, N	N	ⓗ, W	$*$	$*$	ⓗ, SW	ⓗ, S	S
f	ⓗ, N	ⓗ, N, NE	ⓗ, N, NE	ⓗ, N	ⓗ, NE	$*$	$*$	ⓗ, E	ⓗ, E
g	ⓗ, N, NW	ⓗ, N	N	ⓗ, NW	ⓗ, N	$*$	ⓗ, W	$*$	ⓗ, E
h	W	W	W	W	W	$*$	ⓗ, W	ⓗ, W	$*$

Table 2: First move table with no proximity wildcards, for the example shown in Figure 6. In each cell, we show all the optimal moves, and the ⓗ symbol, if the heuristic move coincides with an optimal move.

	a	b	c	d	e		f	g	h
a	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
b	$*$	$*$	$*$	$*$	$*$	$*$	ⓗ, S, SW	ⓗ, S	S
c	ⓗ, W	ⓗ, W	$*$	W	W	$*$	W	W	W
d	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
e	ⓗ, NW	ⓗ, N	N	ⓗ, W	$*$	$*$	ⓗ, SW	ⓗ, S	S
f	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
g	ⓗ, N, NW	ⓗ, N	N	$*$	$*$	$*$	$*$	$*$	$*$
h	W	W	W	W	W	$*$	ⓗ, W	ⓗ, W	$*$

Table 3: First move table with proximity wildcards.

Algorithm 4: Get next move algorithm for CPDs using proximity wildcards and heuristic moves.

```
1  CPDHP(s, t)
2  d ← pd(s)
3  if |s.x − t.x| ≤ d ∧ |s.y − t.y| ≤ d then
4      return F(s, t)
5  else
6      m ← CPD(s,t)
7      if m = ⓗ then
8          return F(s, t)
9      else
10         return m
```

5 Experiments

We run experiments on two sets of maps: *Dragon Age: Origins* (*DAO*) and a set of 9 large maps from three different games. For the first benchmark, we used 155 maps from the game *Dragon Age: Origins* with the number of nodes ranging from about 2,000 to 100,000. For the second benchmark we used 3 maps from *Dragon Age: Origins*, 3 from *StarCraft* and 3 form *Baldur's Gate II* with a number of nodes ranging from about 100,000 to 300,000.

As a baseline we compare against *Single Row Compression* (*SRC*) with DFS ordering (Strasser, Harabor, and Botea 2014), the fastest optimal solver at the last edition of GPPC (Sturtevant 2014). We further test three variants of *SRC*, each enhanced with additional compression. These algorithms are denoted as follows:

- Algorithm *h*, which implements the heuristic symbol enhancements described in Section 3;
- Algorithm *w*, which implements proximity wildcards, described in Section 4, combined with bidirectional wildcards (Salvetti et al. 2017);
- Algorithm *hw*, which is the combination of the previous two methods, *h* and *w*.

	mean	std	min	25%	50%	75%	max
SRC	6.52	11.2	0.011	0.41	2.37	7.25	68.6
h	1.85	3.82	**0.003**	0.09	0.70	1.87	29.6
w	1.93	3.54	0.005	0.12	0.74	2.04	24.3
hw	**1.48**	**2.82**	0.004	**0.08**	**0.52**	**1.57**	**21.7**

Table 4: Size of CPDs of all maps from DAO (MB).

In the comparison, we examine the improved size, the preprocessing time and the path extraction time.

As mentioned, in our experiments, inside setting *w* we have implemented our proximity wildcards on top of a different type of wildcards from the literature, namely bidirectional wildcards (Salvetti et al. 2017). The relation is that both types of wildcards replace actual move symbols with don't care symbols, improving the compression. The difference is that they are computed significantly differently, exploiting different properties of the problem. See Section 2 for a brief description of bidirectional wildcards. Given the relation between both types of wildcards, we wanted to evaluate how they work together and what are the gains of proximity wildcards on top of Salvetti et al.'s (2017) wildcards.

All algorithms are implemented in C++ and compiled with *clang-902.0.39.1* using *-O3* flag, under *x86_64-apple-*

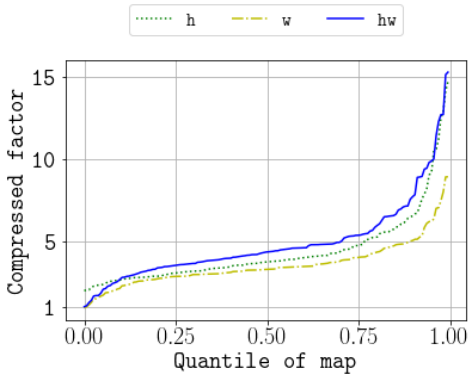

Figure 8: Distribution of the compression factor in the maps from DAO. The compression factor is the CPD size produced by *SRC* enhanced with the setting at hand (*h*, *w*, *hw*) divided by the CPD size produced by *SRC*. Maps are ordered by the *SRC* CPD size.

darwin17.5.0 platform with *2.5 GHz Intel Core i7* Processor and *16 GB 1600 MHz DDR3* Memory. All benchmarks[†] and our implementations[‡] are available online.

5.1 Preprocessing Results

From Table 4, we can notice that for the maps from DAO the sizes of CPDs are quite small. Figure 8 shows the distribution of the compression factor of the proposed methods. From the plot we can see that both proposed methods achieve better compression in 99% of maps, and *hw* tends to dominate other methods in most of maps. We also notice *w* and *hw* produce larger size ($\approx 7\%$) in 1% of maps. The reason is that these maps are small, while *w* needs to compute and store extra information (the proximity distances), and such an extra cost can sometimes be larger than the benefit in such cases.

On the DAO maps, a closer look inside the *w* setting reveals the following: CPDs computed with SRC (Strasser, Harabor, and Botea 2014) enhanced with bidirectional wildcards (Salvetti et al. 2017) have a minimum size of 0.006 MB, an average size of 2.292 MB and a maximum size of 26.526 MB. I.e., the usage of bidirectional wildcards leads to a reduction of the size of the CPDs by 41% to 86%, with an average size reduction of 61%. Neverthless, adding proximity wildcards on top of this (i.e., running the *w* setting) is even stronger: *w* further improves the size of the CPDs, compared to SRC with bidirectional wildcards, by 6.33% to 43.68%, to an average improvement of 24.86%.

We also examine the compression performance on larger maps, with results shown in Table 5. Notice that even in larger maps, the final CPD size produced by our methods are still small. For example, map *AR0044SR* has 231469 nodes; an uncompressed first move matrix needs 231469^2 Bytes $\approx$ 53.5GB; for this map, *SRC* requires 507 MB, while *hw* requires only 9.1MB, which is more than 50 times smaller. In fact, *h* alone leads to a reduction by a factor of 25.46 on this map.

Figure 9 shows the rate between the preprocessing time of *SRC* using methods *h*, *w*, *h*w and the preprocessing time of *SRC* for all maps (including the DAO set and the large maps). From the figure we can see that, on preprocessing, *h*, *w*, and *hw* require more time than *SRC*. The worst-case preprocessing slow-down does not exceed a factor of 4. In 80% of the cases, the preprocessing slow-down does not exceed a factor of 1.5.

[†]https://movingai.com/benchmarks/grids.html

[‡]https://github.com/eggeek/CPD-Hsymbol-Wildcard

		The size of the CPD in MB			
Maps	#cells	*SRC*	*h*	*w*	*hw*
AR0044SR	231,469	507.1	14.3	34.8	**9.1**
AR0605SR	140,922	179.4	25.4	23.0	**14.2**
AR0700SR	131,852	69.0	37.8	23.3	**20.3**
Aftershock	166,076	88.0	8.9	14.7	**7.6**
DarkContinent	285,669	213.8	70.5	50.6	**40.4**
TheatreofWar	220,816	170.1	53.9	42.2	**36.6**
hrt000d	106,608	60.8	11.0	11.4	**6.8**
ost000a	130,478	44.1	15.9	13.4	**9.7**
ost000t	105,707	38.0	13.4	11.2	**8.0**

	The number of the RLE runs $\times 10^3$			
Maps	*SRC*	*h*	*w*	*hw*
AR0044SR	126,086	2,887	7,549	**1,128**
AR0605SR	44,418	5,917	5,042	**2,854**
AR0700SR	16,856	9,050	5,166	**4,422**
Aftershock	21,512	1,730	2,846	**1,057**
DarkContinent	52,594	16,767	11,226	**8,678**
TheatreofWar	41,852	12,803	9,437	**8,045**
hrt000d	14,878	2,418	2,325	**1,175**
ost000a	10,644	3,574	2,695	**1,763**
ost000t	9,191	3,036	2,267	**1,460**

Table 5: CPD size statistics for the large maps used in experiments. Top: the number of cells per map, and the CPD size in MB. Bottom: the number of RLE runs per map in thousands.

	Full path extraction					
	std dev.	min	25%	50%	75%	max
h	0.43	0.23	1.03	1.16	1.33	8.44
w	0.53	0.28	1.10	1.28	1.52	9.68
hw	0.64	0.22	1.03	1.24	1.50	11.05

Table 6: Speed up factor when extracting a full path measured as the time cost of *SRC* divided by the time cost of the proposed algorithms.

5.2 Path Extraction Results

Our proposed methods achieve impressive compression performance without paying any cost in path extraction. Table 6 shows the speed up factor for the full path extraction. From the table, we can see that the proposed algorithms tend to be faster and data shown in Table 7 helps understand why: i) *h* runs the same number of binary searches as *SRC*, but the former has smaller CPD size, so that the binary search is faster;

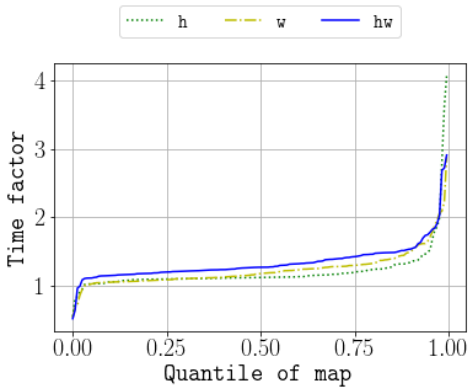

Figure 9: Distribution of the preprocessing-time factor over all maps. The preprocessing-time factor is the preprocessing time cost of the proposed algorithm at hand (*h*, *w*, *hw*) divided by the preprocessing time cost of *SRC*. Maps are ordered by *SRC* size.

#op	mean	std	min	25%	50%	75%	max
l_{src}	382.6	383.0	0.0	109	268	525	2563
c_{src}	0.00	0.00	0.0	0.0	0.0	0.0	0.0
l_h	382.6	383.0	0.0	109	268	525	2563
c_h	0.00	0.00	0.0	0.0	0.0	0.0	0.0
l_w	365.6	383.9	0.0	91	251	508	2563
c_w	17.04	22.49	0.0	5.0	11.0	22.0	428.0
l_{hw}	368.5	385.2	0.0	93	254	510	2563
c_{hw}	14.14	20.75	0.0	2.0	9.0	18.0	428.0

Table 7: Number of operations in path extraction. Here l represents the number of binary searches, and c represents the number of times the target is inside the source's proximity square.

also, smaller CPDs could improve the rate of cache hits; ii) using *w* if a target is inside the source's proximity square can avoid looking for the optimal first move in the CPD, via a binary search, and compute it directly in constant time instead; iii) *hw* can benefit from both.

6 Conclusion

In this paper, we have shown how to substantially reduce the size required to store compressed path databases (CPDs) for a grid map, by using a heuristic move symbol, and introducing wildcard symbols which we store for nodes in the proximity to the source. Using these methods together we can reduce the size of a CPD significantly. The reduction in CPD size also leads to an improvement in lookup time, even though the lookup function is slightly more complex. With these improvements we define a new state of the art for CPDs.

While the paper has focused on grid maps, most of what we have talked about can be extended to more general graphs. Given a distance function we can apply the distance function approach to any graph. We can extend directional tie-breaking to any graph embedded in the Euclidean plane. Proximity wildcards can also be extended to this case by discovering the largest radius within which the move to all target nodes follows the distance function heuristic.

Interesting directions for the future work include extending the area of the proximity wildcards (e.g., with rectangles instead of squares), evaluating the effectiveness of the proposed techniques for the multi-row compression of the CPD (Strasser, Harabor, and Botea 2014), and investigating the usage of CPDs and default moves on road maps.

References

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.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Sciences and Cybernetics* 4(2):100–107.

Salvetti, M.; Botea, A.; Saetti, A.; and Gerevini, A. E. 2017. Compressed path databases with ordered wildcard substitutions. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, (ICAPS-17)*, 250–258.

Salvetti, M.; Botea, A.; Gerevini, A. E.; Harabor, D.; and Saetti, A. 2018. Two-oracle optimal path planning on grid maps. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, (ICAPS-18)*, 227–231.

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 Seventh Annual 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.

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