Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

Bounded Suboptimal Path Planning with Compressed Path Databases

Shizhe Zhao
shizhe.zhao@monash.edu
Monash University

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

Adi Botea
adibotea@eaton.com
Eaton

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

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

Abstract

Compressed Path Databases (CPDs) are a state-of-the-art method for path planning. They record, for each start position, an optimal first move to reach any target position. Computing an optimal path with CPDs is extremely fast and requires no state-space search. The main disadvantages are overhead related: building a CPD usually involves an all-pairs precomputation, and storing the result often incurs prohibitive space overheads. Previous research has focused on reducing the size of CPDs and/or improving their online performance. In this paper we consider a new type of CPD, which can also dramatically reduce preprocessing times. Our idea involves computing first-move data for only selected target nodes; chosen in such a way as to guarantee that the cost of any extracted path is within a fixed bound of the optimal solution. Empirical results demonstrate that our new bounded suboptimal CPDs improve preprocessing times by orders of magnitude. They further reduce storage costs, and compute paths more quickly – all in exchange for only a small amount of suboptimality.

Introduction

Path planning is an important and long studied problem in AI, and it is a problem which finds applications 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).

A family of techniques known as Compressed Path Databases (CPDs) (Botea 2011; Botea and Harabor 2013) represent the state of the art in this area, for speed in computing optimal grid paths and also optimal prefixes (i.e., the first several steps of an optimal path). Each CPD is simply a data structure that tells, for any start location s and any target location t, which is the optimal first move from s towards t. Finding a shortest path using such a database is straightforward: simply look up the optimal first move toward the target and execute that move, repeating as necessary until the target is reached. Results from the 2014 GPPC (Sturtevant et al. 2015) show that CPDs are faster than other modern pathfinding approaches on grids, including Contraction Hierarchies (Geisberger et al. 2008) and Jump Point Search (Harabor and Grastien 2014). For shortest paths and prefixes on gridmaps, CPDs are also known to be faster than Hub Labels (Delling et al. 2014; Strasser, Harabor, and Botea 2014), a similarly database-driven technique but one which focuses on shortest distance queries (cf. shortest path) and which targets road networks.

The principal advantage of CPDs is speed: optimal paths can be found quickly and without any state-space search. Furthermore, optimal prefixes can be found faster still. This feature is important to reduce the *first-move lag*, where an agent needs to wait until it knows in which direction to start moving. By contrast, consider that search-based methods can only know an optimal prefix once they know the entire path. The main drawback of CPDs is build cost: each database requires an all-pairs pre-compute with space and time being worst-case quadratic in the size of the input graph; i.e., we need to run one Dijkstra search per node in the graph and then compress and store the result. Researchers in this area prioritise fast online performance and for this reason works tend to focus on reducing the size of the database so that it better fits in working memory (Strasser, Harabor, and Botea 2014; Salvetti et al. 2017; Chiari et al. 2019), or they focus on improving the lookup performance, so that moves can be extracted faster still (Salvetti et al. 2018). These efforts have improved CPD performance, for offline storage and online performance, by over one order of magnitude beyond the 2014 GPPC baseline. Despite these gains the preprocessing time required to construct each database has not substantially reduced and this aspect can be prohibitive.

In this paper we consider a new bounded-suboptimal take on CPDs which dramatically cuts the time required for precomputation. The approach can also reduce storage costs and yield faster online performance – all in exchange for a small amount of additional cost per extracted path. Our idea

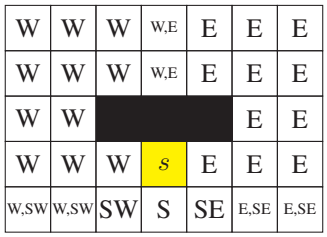

Figure 1: Optimal moves from the source cell s to each traversable cell t. Multiple optimal moves can exist in some cases (e.g., W, E on the top row, middle cell; and E, SE for the bottom-right cell).

involves selecting from the graph, induced from the input map, a subset of nodes C, which we call *centroids*, such that every node is at most a path distance δ from some centroid $c \in C$. During precomputation we store first-move data from every node in the graph to every centroid. The entire procedure requires at most $|C|$ offline instantiations of Dijkstra search. Moreover, the resulting database is sufficient to determine a bounded suboptimal path, from any node s to any other node t. We give a theoretical description of the method and show that, for a given value of δ, the cost of each extracted path is at most 2δ larger than optimal, with δ being an input parameter.

In experiments, we test this idea with different δ values and with different compression schemes. Results on standard grid benchmarks indicate order-of-magnitude reductions in preprocessing times, and up to several factors improvement in database size and lookup speed. We also show that, in most cases, the suboptimality cost per path is substantially smaller than the guaranteed upperbound, and the relative suboptimality is usually small.

Background

Gridmaps. A gridmap is a two-dimensional data structure that represents the operating environment for a mobile agent, such as a robot or a game character. Gridmaps rasterise the environment into square cells, with each cell being either traversable or blocked. Each cell has up to 8 neighbours: one in each of the four cardinal (equiv. straight) directions and one in each of the four ordinal (equiv. diagonal) directions.

When moving, an agent occupies exactly one traversable cell at a time and is allowed to step to any other adjacent traversable cell. Straight moves have a cost of 1, while diagonal moves cost $\sqrt{2}$. As in the GPPC, we enforce the *no-corner-cutting rule*, which says that diagonal moves are disallowed if the origin and destination cell share a common neighbour which is not traversable.

Given a grid cell n, we write $n.x$ and $n.y$ for the x and y coordinates of that cell. Further, let $MV = \{$N, NE, E, SE, S, SW, W, NW$\}$ be the set of 8 principal compass directions in which the agent can move. Given a move $m \in MV$ and cell n we define $m(n)$ as the cell n' where $n'.x = n.x + horiz(m)$ and $n'.y = n.y + vert(m)$ where

$$horiz(m) = \begin{cases} -1 & m \in \{\text{SW, W, NW}\} \\ 0 & m \in \{\text{N, S}\} \\ +1 & m \in \{\text{NE, E, SE}\} \end{cases}$$

$$vert(m) = \begin{cases} -1 & m \in \{\text{NW, N, NE}\} \\ 0 & m \in \{\text{E, W}\} \\ +1 & m \in \{\text{SE, S, SW}\} \end{cases}$$

Each gridmap induces an undirected graph where traversable cells are nodes, and the moves applicable in each traversable cell are edges. Consider then a weighted graph $G = (V, E)$ with vertices V and edges $E \subseteq V \times V$, and a function w such that $w(s,t)$ is the cost of edge $(s,t) \in E$. A *path* p from s to t in G is a sequence of nodes $[n_0, n_1, n_2, \ldots, n_{k-1}, n_k]$, where $k \in \mathbb{N}^+$, $n_0 = s$, $n_k = t$, and $(n_i, n_{i+1}) \in E, 0 \leq i < k$. The *length* of the path is $|p| = \sum_{i=0}^{k-1} w(n_i, n_{i+1})$. The *reverse* $rev(p)$ of the path p is the reverse sequence of its nodes. Let $sp(s,t)$ return a shortest length path in G from s to t, and let symbol $+\!\!+$ denote sequence (and thus also path) concatenation.

Compressed Path Databases (CPDs). A CPD is a data structure for encoding the first edge or move m on an optimal path, from any node s towards any node t.

For clarity, we start with defining the *first-move matrix*, a square two-dimensional array that stores optimal moves from every start to every target. The size of a first-move matrix quickly becomes prohibitive, being quadratic in the number of (traversable) cells.

A CPD is obtained by compressing the first-move matrix. Previous work compresses CPDs row by row. CPDs are constructed *offline* in a preprocessing phase that requires repeated iterations of Dijkstra search: one per graph node. With only slight modifications, this algorithm can be used to compute $T(s)$, the *first-move row* (of the matrix) corresponding to the start node s, which records *all* optimal first moves, from the source node s to any reachable target t. Once computed, $T(s)$ is compressed and stored, which concludes the iteration at hand. The compression is based on run-length encoding (RLE) (Strasser, Harabor, and Botea 2014). See Examples 1 and 2 for an illustration. As they are independent, distinct Dijkstra iterations can be run in parallel, with a speed-up linear in the number of processors.

Example 1 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 Figure 1) 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. □

Example 2 Consider the gridmap shown in Figure 1. Assume for simplicity that we order all grid cells as one single string, traversing each row from the left to the right and going row by row from the top to the bottom. We call this

Algorithm 1: Function $cpd(s,t)$ extracts at runtime an optimal path from the node s to the node t.

```
p ← [s]
while s ≠ t do
    m ← ∅
    if t is in the proximity square of s then
        m ← F_x(s,t)
    else
        m ← FirstMove(s,t)
        if m = ⓗ then
            m ← F_x(s,t)
    p ← p +₊ [m(s)]
    s ← m(s)
return p
```

the *left-right-top-bottom* ordering. The entire string is compressed into 11 runs: 1W 5E 8W 12E 15W 20E 22W 26E 29SW 32S 33SE. Obstacle cells 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. Storing all optimal moves towards a given target (e.g., E and SE for each of the last two cells at the bottom) improves run-length compression, since we can choose the symbol that would result in fewer runs (e.g., SE in that case).

The CPD size can further be reduced by replacing the left-right-top-bottom scheme with a different cell ordering. In experiments, we use the state-of-the-art DFS ordering heuristic (Strasser, Harabor, and Botea 2014). □

With a CPD in hand, we may begin the *online* phase of the algorithm. Here the objective is to compute an optimal path $cpd(s,t)$ or prefixes for any given start-target pair (s,t) (equiv. *instance*). We denote as FirstMove(s,t) the function which returns an optimal first move from s to t. The implementation of this function requires a simple binary search through a compressed string of symbols representing the first-move row $T(s)$ (Strasser, Harabor, and Botea 2014). The function can be used to extract entire optimal paths or optimal prefixes of any given size. Algorithm 1 illustrates the extraction of optimal paths. The method relies on two recent optimisations (Chiari et al. 2019) which are known to improve efficiency: Heuristic Moves and Proximity Wildcards. We describe these next.

Heuristic Moves. When pathfinding on a gridmap the first move from s to t is often the "obvious move"; e.g., the move that heads directly towards the target or the one suggested by some "default" heuristic function $F_x(s,t)$. We can add an extra "redundant" symbol ⓗ in cells where the default move is an optimal move, after the completion of Dijkstra search but before compression with RLE. These additional symbols, sometimes called *h-moves*, can be added to the first move table in linear time and they significantly improve the efficiency of CPDs; i.e. they help to reduce the total size of the database with no negative impact on query performance.

E.g., by using the default heuristic function proposed by Chiari et al. (2019), the string of symbols shown in Figure 1 can be compressed into 1ⓗ. That is, the entire compressed string has only 1 run, which is substantially shorter than options discussed in Example 2.

Exploiting h-moves during the online stage requires only a small modification to the standard path retrieval function: after extracting a ⓗ symbol from the CPD, we apply the move suggested by the default heuristic function. Note that ⓗ symbols are applicable (i.e., added to the first move table) only if the default heuristic returns just one possible move from s to t. Our implementation is based on Chiari et al.'s (2019) algorithm and uses the same heuristic.

Proximity Wildcards. Chiari et al. (2019) introduce a further use of heuristic moves. They compute the largest square around the start position s where all moves can be ⓗ, and store its size for each s. Before extracting a first move they check whether the target is within the square. If so, they simply apply the heuristic move. This speeds up the path extraction and reduces the size of the database.

Bounded Path Finding using Centroids

For some application settings, the space and time required to build a CPD could be considered too large. To address such situations, we propose to compute and compress first-move data for only a subset of grid nodes C, called *centroids*.

We associate a nearest centroid $c(t) \in C$ for every grid node t. Then, a path from any s to any t, which we call a *centroid path* (cp for short), can be built as:

$$cp(s,t) = cpd(s,c(t)) +\!\!+ rev(cpd(t,c(t)).$$

We can do better by joining the two paths at the first common point p, which may be different from $c(t)$:

$$\begin{aligned} cp(s,t) \quad = \quad & [n \mid n \in cpd(s,c(t)), n \notin cpd(t,c(t))] \\ & +\!\!+ [p] \\ & +\!\!+ [n \mid n \in rev(cpd(t,c(t)), n \notin cpd(s,c(t))]. \end{aligned}$$

Given we have chosen centroids so that no target t is more than δ from its centroid $c(t)$, we can show that the centroid path is never longer than 2δ than the optimal path.

Theorem 1. *If $|sp(t,c(t)| \leq \delta$ for all t in V, then $|cp(s,t)| \leq |sp(s,t)| + 2\delta$.*

Proof. Assume by contradiction that $|cp(s,t)| > |sp(s,t)| + 2\delta$. CPD paths $cpd(s,c(t))$ and $cpd(t,c(t))$ are optimal, and hence $|sp(s,c(t)| + |sp(t,c(t)| \geq |cp(s,t)|$. We have that $\delta \geq |sp(t,c(t)|$, as assumed in the theorem. Thus, the path $sp(s,t) +\!\!+ sp(t,c(t))$, from s to $c(t)$, has a length of at most $|sp(s,t)| + \delta$. This further leads to $|sp(s,t)| + \delta \geq |sp(s,t) +\!\!+ sp(t,c(t))| \geq |sp(s,c(t))|$. Combining these, we have:

$$\begin{aligned} |sp(s,c(t)| + |sp(t,c(t)| &\geq |cp(s,t)| \\ &> |sp(s,t)| + 2\delta \\ &\geq |sp(s,c(t))| + \delta \\ &\geq |sp(s,c(t))| + |sp(t,c(t)| \end{aligned}$$

Contradiction. □

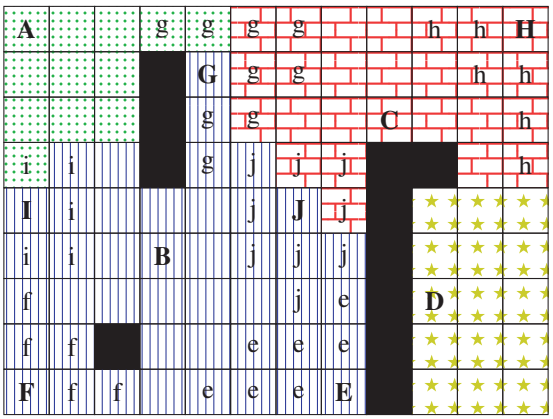

Figure 2: Centroid marking for a grid map with $\delta = 3$. **A**, **B**, **C** and **D** are centroids in the first stage, regions explored from them are filled with green dots (A), blue vertical lines (B), red bricks (C) and yellow starts (D). Centroids **E–J** are created in the second stage, and their corresponding regions are shown with lowercase letters.

Building Centroids

We propose the following algorithm which guarantees that there may be at most $\frac{2|V|}{\delta}$ centroids in each connected component, where $|V|$ is the number of traversable cells in the component. The algorithm (Algorithm 2) has two stages: the first stage makes the distance of the furthest node to a centroid in $[\delta, 2\delta]$; the second stage makes the distance of the furthest node to a centroid lower than or equal to δ. To achieve this, we prioritize grid nodes based on two criteria:

- d_c: the shortest distance to the nearest centroid, which are ∞ at the beginning and get updates during the algorithm.
- d_o: the shortest distance to the nearest obstacle, which can be precomputed by flood-fill.

In the first stage, we keep selecting node which $d_c > 2\delta$, with *minimum* d_o, and *minimum* d_c if there is a tie. For each selected node v, we run a Dijkstra search to explore neighbors of v within $2\delta + 1$ and update their d_c. In the second stage, we keep selecting node which $d_c > \delta$, with *maximum* d_c, and *minimum* d_o if there is a tie. For each selected node v, similarly to the first stage, we run a Dijkstra search to explore the neighbors of v within δ and update their d_c. Notice that, in the first stage, selecting the node with a minimum d_o is to choose centroids from the "border" to the "center" of the gridmap; selecting the node with a minimum d_c is to make a new centroid around the existing centroid regions. The motivation for such a strategy is to make more border cells covered by centroids in the first stage. We recognize there can be many other ways of selecting centroids. For this work, we choose this strategy as an our preliminary investigation showed that it performs well.

Theorem 2. *Assume that the gridmap induces a graph with only one connected component, and let $|V|$ be the number of traversable cells where $|V| \gg \delta$, then Algorithm 2 creates at most $\frac{2|V|}{\delta}$ centroids.*

Algorithm 2: Centroid creation Centroids(V,G) given a graph (V, G).

```
1  Calculate d_o[v], the shortest distance from vertex v to
     an obstacle by flood fill in;
2  d_c[v] ← ∞, v ∈ V;
3  c[v] ← ⊥, v ∈ V;
4  Q ← V;
5  while Q ≠ ∅ do
6      v ← arg min{(d_o[v], d_c[v]) | v ∈ Q};
7      Q ← Q − {v};
8      if d_c[v] > 2δ then
9          for v' ∈ V where d = |sp(v, v')| ≤ 2δ + 1 do
10             if d < d_c[v'] then
11                 d_c[v'] ← d;
12                 if d ≤ δ then
13                     c[v'] ← v
14 Q ← V;
15 while Q ≠ ∅ do
16     v ← arg max{(d_c[v], −d_o[v]) | v ∈ Q};
17     Q ← Q − {v};
18     if d_c[v] > δ then
19         for v' ∈ V where d = |sp(v, v')| ≤ δ do
20             if d < d_c[v'] then
21                 d_c[v'] ← d;
22                 c[v'] ← v
```

Proof. After the second stage, the distance between any pair of centroids is greater than δ (algorithm 2 line 18), thus for a given centroid c, all nodes $d(v, c) \leq \frac{\delta}{2}$ must belong to c, because if a node v is reassigned to c, c must be the nearest centroid to v (algorithm 2 line 20). Take a node v over the bound of the area covered by c that $d(v, c) \geq \frac{\delta}{2}$; if the $sp(v, c)$ formed by only straight moves then there are at least $\frac{\delta}{2}$ nodes covered by c; otherwise, each diagonal move on the shortest path implies that there are 3 nodes covered by c as corner-cutting is not allowed, then even more. Thus each centroid covers at least $\frac{\delta}{2}$ nodes, so that there are at most $\frac{2|V|}{\delta}$ centroids. □

Building centroid CPDs With the marking complete, we next compute a first-move matrix: from each grid node v to each centroid $c \in C$. A main objective is to reduce the number of Dijkstra searches from $|V|$ to $|C|$.

In the Dijkstra search for a node $c \in C$, c plays the role of a target. The challenge is to still be able compute forward moves from every node v towards c. On undirected graphs, which is the case with our gridmaps, this can be achieved with only small adjustments to Dijkstra's algorithm, in a fashion also used by Verzeletti, Botea, and Zanella (2019).

With this change in use, now we compute the first-move matrix in a column-by-column fashion, with all rows growing gradually. To avoid storing the full first-move matrix in memory, we compress each row on the fly: once a Dijkstra

search is done, we extend each row with the corresponding set of optimal moves, and decide whether we can continue with the current RLE run, or start building a new one. The result is equivalent to building the uncompressed row and then compressing it.

Reverse Compressed Path Databases

A standard CPD compresses first-move matrix *rows* (i.e., an array of moves from one source to many targets). We call this *forward compression*. In this section we introduce CPDs that compress *columns* (i.e., arrays with moves from many sources to one target). We call these *reverse CPDs*, and their introduction is motivated as follows. We will show that reverse CPDs can achieve better size savings than the forward ones for certain (but not all) δ values. We will further show that reverse CPDs return moves faster, due to a combination of better caching and the ability to return several moves with a single lookup. Both forward and reverse CPDs enjoy a dramatic cut in the preprocessing time, as δ grows.

Reverse CPDs are compatible with recent space-saving improvements such as heuristic moves and proximity wildcards (Chiari et al. 2019).

Example 3 Figure 3 shows reverse first-move data that we compress to 1S 3SW 5SE 6S 10W 12E 13S 15SW 16S 21SE 22E 26W 29NE 32N 33NW, which is 15 runs. Our encoding assumes a left-right-top-bottom cell ordering. Adding h-moves reduces this to 1S 3SW 5SE 6S 10W 12E 13ⓗ, which is 7 runs. For comparison, in the forward direction the first-move data for source t can be encoded with just a single run: 1ⓗ. □

"Illegal" Moves

We can improve the run length encoding of reverse CPDs by also allowing "illegal" moves to be part of the set of possible moves, which are useful if they can be unambiguously *decoded* by some function to extract a correct optimal move. A move from s is "illegal" if: (a) the move is cut by a corner or the reached node is an obstacle (see Example 4); (b) the reached node has the same set of successors as s, except s itself (see Example 5).

Example 4 Figure 4 shows reverse first-move data where S moves appear as the only symbols of three cells in column 3. These symbols force us to create one additional run per row. To improve compression we will store at each of these locations an additional symbol, SW, which can be used to reduce the database by three runs. Although following this move produces an obstacle, we can still proceed if we detect the situation at runtime. To decode the SW symbol we choose the closest move to SW which is not blocked: here, S. The definition of "illegal" move guarantees that our decoding function always returns an optimal first move. □

Example 5 Consider the example in Figure 5. We show a reverse first-move table which already includes some "illegal" moves; i.e., those that produce obstacles and which can be appropriately decoded. Further compression here is hampered by the two tiles with SE-only moves. Notice however that, while the S move from each of these locations is valid, it is not *helpful*. The only reason to apply move S from these positions is to reach the cell directly below and no other. If we identify such unhelpful moves at compression time, we can add S to the set of symbols for the currently SE-only tiles. When extracted at runtime, the "illegal" move S can be decoded to SE. With this enhancement the table compresses into 2 runs: 1S 18W. □

Preprocessing and Path Extraction

As explained earlier, building a reverse CPD for target t can be performed by a single Dijsktra search from the target and recording for each grid node s all best moves for reaching t. The resulting string of moves (column in the first-move matrix) is compressed with RLE. In the experimental section, we will show that build times for reverse CPDs and forward CPDs using the same set of centroids C are similar.

Algorithm 3 gives a modified version of FirstMove(s, t) for reverse CPDs which decodes both "illegal" and h-moves. This algorithm extracts a first move m as usual, and, if m is not legal, it finds the closest move that is legal and returns that instead. Function $legal(m, s, t)$ returns $false$ if moving in direction m from s (i) leads to a blocked square, or (ii) cuts a corner, or (iii) moves to a position $m(s) \neq t$ where all successors are better reached directly from s. In all other cases $legal(m, s, t)$ returns $true$. Function $closest(m, S)$ returns the closest move to m in set S breaking ties by moving clockwise, e.g., $closest(\text{SW}, \{\text{N}, \text{NE}, \text{E}\}) = \text{N}$ and $closest(\text{S}, \{\text{N}, \text{NE}, \text{E}\}) = \text{E}$. Note that forward CPDs can also store "illegal" moves, but only get a slight improvement.

Heuristic Path Extraction

Paths extracted with centroid CPDs, whether forward or reverse, can experience a pathological worst-case where the cost of the 2δ suboptimal detour can be substantially larger than the cost of the optimal path. Such cases occur when the start and target position are (i) in close proximity; (ii) when there exist multiple paths from s to the centroid $c(t)$ and; (iii) when the CPD stores the bad direction from s to $c(t)$ for a specific and pathological choice of t. Figure 6 gives an example. Notice that the relative suboptimality can be large in a relative sense but it is also local to the centroid. Thus we can often improve the path computation by trying a direct heuristic path, from the current node to the target, when we arrive close to the centroid. Our approach is as follows.

When the current position p in the path has the same centroid as the target $c(p) = c(t)$ we check whether following the default heuristic function $F_x(p, t)$ repeatedly leads us to the target, and instead use this path. For the example of Figure 6 starting from s and following the moves derived from the CPD we hit the region of centroid $c(t)$ at position p and find the direct path straight west to the target. We extract the direct path of length 2 by omitting overlapping paths from s to p and back.

Experiments

We run experiments on maps from the GPPC 2012 benchmarks (Sturtevant 2012), including 105 game maps and 6 ar-

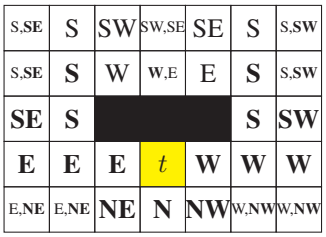

Figure 3: Optimal first moves from each grid cell to the cell marked t. Moves in bold agree with the default heuristic.

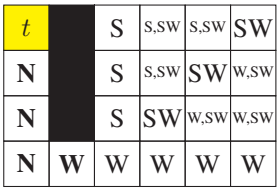

Figure 4: Optimal first moves from each grid cell to the cell marked t. Moves in bold agree with the default heuristic.

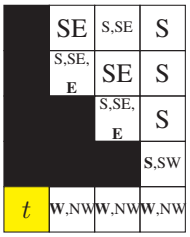

Figure 5: Optimal first moves toward the grid cell t with added "illegal" move symbols for cells adjacent to obstacles.

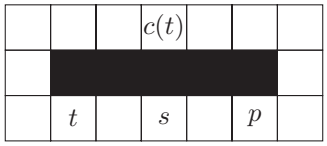

Figure 6: Worst case suboptimality for $\delta = 6$: the symbol stored for FirstMove(s,$c(t)$) is E rather than the equivalent W. The path found from s to t is length 14 rather than 2.

Algorithm 3: Function FirstMove$^r(s,\ t)$ for reverse CPDs with h-moves and "illegal" moves.

```
m ← FirstMove(s,t)
if m = ⓗ then
    m = F_x(s,t)
if ¬legal(m,s,t) then
    m = closest(m,{mv|mv ∈ MV, legal(mv,s,t)})
return m
```

tificial maps - 2 out of 9 maps each in *mazes, rooms, random*. Although all forward CPDs and most of reverse-centroid CPDs work on the rest of the 21 artificial maps, for small values of δ the size of the reverse CPD can be prohibitively large for these adversarial environments. To avoid dealing with missing values we choose to exclude these maps. All algorithms are implemented in `C++`[1] and compiled with `-O3` flag. We use the following abbreviations for convenience:

- $\texttt{fwd}_\delta$: forward CPD with centroid size δ
- $\texttt{rev}_\delta$: reverse CPD with centroid size δ.

Note that $\delta = 0$ means the full forward or reverse CPDs. Our principal point of comparison in all experiments is $\texttt{fwd}_0$. This variant includes the techniques in (Chiari et al. 2019) and represents the current state-of-the-art for CPDs, both for online performance and also offline storage costs. Our implementations builds directly on this baseline using code from the original authors. Like the previous work we employ proximity wildcards for all forward CPDs. For reverse CPDs we exclude this feature as we found that it does not lead to substantial improvement in compression. Our test machine is `Linux 4.19.45-1-MANJARO` with `i5-8600 CPU @ 3.10GHz` CPU and `15GB` memory.

Experiment 1: Preprocessing Massive preprocessing-time improvements are observed on all maps, compared to previous work, which has been limited to full CPDs. For example, Table 1 shows, for 5 representative maps, preprocessing and size statistics. Using centroids reduces the CPD construction time by up to three orders of magnitude (i.e. the number of centroids is up to thousands of times smaller than the number of cells in a full CPD). For a fixed δ, the preprocessing time is similar for forward and reverse CPDs since both are dominated by the Dijkstra search time. Forward CPDs are built slightly faster, as they skip "illegal" move processing. The savings are achieved due to the fact that the preprocessing time is proportional to the number of Dijkstra searches, which is equal to the number of centroids.

Experiment 2: CPD Size Assume that $|V|$ is the number of cells and $|C|$ is the number of centroids. A forward CPD starts from a competitive size but its size decreases at a moderate pace (smaller than linear) as the number of centroids decreases. The reason is that we always have $|V|$ compressed strings (rows) in a forward CPD, independent of $|C|$. Each compressed string compresses a row of length $|C|$. Rows in a full CPD compress well (much better than columns), and cutting symbols from a row that compresses well anyway has a moderate impact on further size reductions. By contrast, a reverse CPD starts from a much larger size (as columns compress more poorly than rows), but it decreases more aggressively, at a linear pace with the number of centroids. This is because a reverse CPD has exactly $|C|$ compressed strings (columns). Often, a reverse CPD gets smaller than a forward CPD as δ increases.

The interplay of these factors is shown in Figure 7, which shows a summary of size comparison. For $\delta \geq 8$ the size of $\texttt{fwd}_\delta$ and also $\texttt{rev}_\delta$ improves on $\texttt{fwd}_0$ in a large majority of cases. There are however 5 small game maps where $\texttt{fwd}_{64}$ CPDs are larger than $\texttt{fwd}_0$, and there are 2 large arti-

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

	δ	0		2		4		8		16		32		64	
map	stat	F	R	F	R	F	R	F	R	F	R	F	R	F	R
Aurora	C	493772		93346		22078		6777		2278		837		372	
	T	551.38	576.07	104.24	108.90	24.65	25.76	7.57	7.91	2.54	2.66	0.93	0.98	0.42	0.43
	M	**341.81**	10015.22	**320.08**	1892.17	**225.17**	448.44	**123.01**	139.63	72.66	**48.88**	51.66	**19.90**	40.20	**10.54**
orz103d	C	40392		7977		2134		759		287		105		44	
	T	2.02	2.02	0.40	0.40	0.11	0.11	0.04	0.04	0.01	0.01	0.01	0.01	0.00	0.00
	M	**1.40**	176.16	**1.51**	35.22	**1.42**	9.78	**1.37**	3.79	**1.32**	1.74	1.27	**0.94**	1.24	**0.68**
maze-400-4	C	127996		24526		7899		3687		1443		607		257	
	T	17.26	24.16	5.40	7.56	2.00	2.80	0.75	1.05	0.26	0.37	0.08	0.11	0.02	0.02
	M	**4.10**	3862.07	**4.52**	741.37	**4.35**	239.81	**4.23**	112.76	**4.05**	45.07	**3.91**	19.85	**3.81**	9.29
room-400-40	C	152811		27480		6524		2111		615		190		55	
	T	43.30	45.84	7.79	8.24	1.85	1.96	0.60	0.63	0.17	0.18	0.05	0.06	0.02	0.02
	M	**74.24**	2149.96	**69.40**	388.23	**44.11**	93.68	**25.64**	31.46	15.17	**10.38**	10.31	**4.43**	7.70	**2.60**
random-400-33	C	103535		32381		12019		4515		1566		456		105	
	T	17.26	24.16	5.40	7.56	2.00	2.80	0.75	1.05	0.26	0.37	0.08	0.11	0.02	0.02
	M	**40.77**	5208.14	**34.21**	1629.89	**27.40**	605.78	**21.19**	228.36	**15.14**	80.01	**9.52**	24.19	**5.66**	6.52

Table 1: Number of (C)entroids, building (T)ime in minutes, and (M)emory requirements in MB for (F)orward and improved (R)everse CPDs for different radii δ on five representative maps. For $\delta = 0$, C is the number of cells in the map.

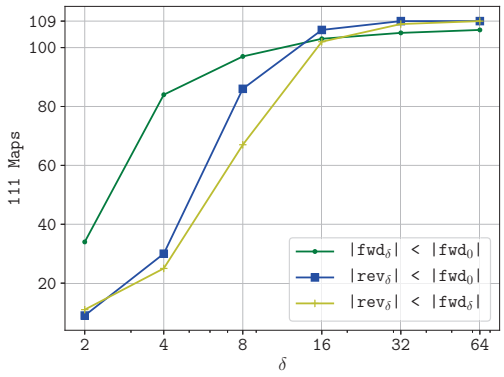

Figure 7: We show 3 comparisons: $|\texttt{fwd}_\delta|$ v.s. $|\texttt{fwd}_0|$ (the green-dot line), $|\texttt{rev}_\delta|$ v.s. $|\texttt{fwd}_0|$ (the blue-square line), and $|\texttt{rev}_\delta|$ v.s. $|\texttt{fwd}_\delta|$ (the yellow-cross line); each line shows the number of maps with smaller CPD when δ increase.

ficial maps where $\texttt{rev}_{64}$ CPDs are larger than $\texttt{fwd}_0$. Notice however that $\texttt{rev}_\delta$ outperforms $\texttt{fwd}_\delta$ on more than half of all maps when $\delta = 8$ and compression improves further beyond this point. Table 2 shows in more detail how much size reduction centroid CPDs can achieve – $\texttt{rev}_\delta$ (resp. $\texttt{fwd}_\delta$) can be up to 32.43x (resp. 9.65x) smaller than $\texttt{fwd}_0$.

Illegal moves: We can additionally report (not in Table 1) that without *illegal move* encoding, reverse CPDs can be 1.5x *larger* for maps such as room-400-40 and maze-400-4. For game maps, illegal moves provide even more benefit – up to a factor of 3x vs. no illegal moves.

Trade-offs: We have seen that as δ increases, the forward-CPD size reduction is often limited, while the reduction in size of reverse CPDs is proportional to the reduction in the number of centroids. This introduces a tradeoff as δ grows: the smaller the CPD the looser the guarantee. In the remaining experiments we focus on this tradeoff and restrict our attention to $\delta \geq 16$ where both forward and reverse CPDs improve on $\texttt{fwd}_0$ across almost all maps and where we construct corresponding databases of the smallest size.

δ	type	mean	min	25%	50%	75%	max
2	rev	0.33	0.01	0.12	0.20	0.35	1.57
	fwd	0.97	0.85	0.93	0.98	1.01	1.19
4	rev	0.82	0.02	0.45	0.72	1.05	2.34
	fwd	1.13	0.85	1.02	1.12	1.24	1.68
8	rev	1.54	0.04	1.13	1.56	1.99	3.45
	fwd	1.34	0.86	1.08	1.27	1.56	2.90
16	rev	2.59	0.09	1.87	2.48	3.12	7.39
	fwd	1.60	0.86	1.14	1.36	1.86	4.90
32	rev	3.81	0.21	2.39	3.18	4.16	17.88
	fwd	1.84	0.87	1.19	1.42	2.10	7.20
64	rev	4.93	0.44	2.70	3.64	5.12	32.43
	fwd	2.08	0.87	1.24	1.50	2.21	9.65

Table 2: CPD size reduction between centroid CPDs and $\texttt{fwd}_0$, $\texttt{ratio} = \frac{|\texttt{fwd}_0|}{|\texttt{rev}_\delta|}$ or $\frac{|\texttt{fwd}_0|}{|\texttt{fwd}_\delta|}$.

Experiment 3: Path Extraction. Reverse CPDs have a runtime advantage due to memory caching as follows. To extract a path, reverse CPDs require only a single compressed string: the one computed for the target t. Once this string is loaded into CPU cache, it resides there throughout the entirety of path extraction. In contrast, forward CPDs require a different compressed string for each move, and extracting each move incurs the possibility of a cache miss.

The second runtime advantage is that we can extract the entire run from a reverse CPD rather than just a single move. That means we can avoid additional extractions if the first move of next location $m(s)$ is compressed in the same run as s. This behavior is surprisingly frequent due to the row ordering. On average, $\texttt{rev}_0$ performs just 20% of extractions needed by $\texttt{fwd}_0$, i.e., each extracted run is used 5 times.

Table 3 examines the efficiency of forward and reverse CPDs ($\delta = 16, 32, 64$) for path extraction. Here we compare against $\texttt{fwd}_0$ as well as two other suboptimal approaches, each with characteristics similar to our work:

- TC: *Tree Cache* (Anderson 2012), a very fast *unbounded-suboptimal* algorithm based on spanning trees. We compare against this method because like our work path extraction is implemented as a series of recursive lookups. TC was the fastest suboptimal algorithm at GPPC 2014.

	mean	min	25%	50%	75%	max
$\texttt{rev}_{16}$	1.839	0.061	1.311	1.747	2.162	235.606
$\texttt{rev}_{32}$	1.738	0.031	1.194	1.666	2.091	229.882
$\texttt{rev}_{64}$	1.580	0.008	0.998	1.490	1.953	207.992
$\texttt{fwd}_{16}$	1.209	0.013	1.038	1.125	1.312	175.824
$\texttt{fwd}_{32}$	1.233	0.033	1.041	1.144	1.355	163.937
$\texttt{fwd}_{64}$	1.230	0.012	1.013	1.139	1.389	184.361
$\texttt{FS}_{32}$	0.037	0.000	0.001	0.002	0.025	18.451
$\texttt{FS}_{64}$	0.039	0.000	0.000	0.001	0.047	17.233
$\texttt{FS}_{128}$	0.041	0.000	0.000	0.002	0.051	17.431
TC	6.951	0.009	4.570	5.961	8.170	949.790

Table 3: Speedup as path extraction time divided by $\texttt{fwd}_0$ path extraction time. A value larger than 1 means faster.

- $\texttt{FS}_{\delta_2}$: *FOCAL Search* (Pearl and Kim 1982), a *bounded-suboptimal* variant of A*. Usually it is implemented with a relative suboptimality bound; i.e. the path returned is guaranteed to be no larger than some multiplicative factor $w \geq 1$. Here we adapt FOCAL search to compute solutions that are no more than an additive constant factor 2δ from optimal. This allows us to compare CPD solution quality against a competitor with equivalent guarantees.

We run each algorithm on all 105 *game maps*. We have 142,534 instances in total and we solve each one 5 times, taking the mean to eliminate random noise. Table 3 gives a summary of results, relative to $\texttt{fwd}_0$. Both forward and reverse CPDs benefit from improved cache locality due to smaller CPD size and, both improve baseline performance. The largest gains are observed for reverse CPDs, which can be almost two times faster for path extraction. The speedup decreases as the compression increases due to overheads from checking the heuristic direct path. Note that only a few factors separate CPDs and Tree Cache. TC can be understood as a performance lowerbound for CPD-like methods since the cost of path extraction for TC is only one memory operation per step. Unsurprisingly, any form of CPD is orders of magnitude faster than state-space FOCAL search.

Experiment 4: Suboptimality and Path Quality. We evaluate the (absolute) suboptimality of a path as $\mathit{diff} = l - o$, that is, the length of the extracted path l minus the length of an optimal path o. Similarly, we evaluate the (relative) quality of a path as $\frac{\mathit{diff}}{o}$. Table 4 and Table 5 shows summaries of path suboptimality and path quality for different approaches. We observe that both suboptimality and quality for the unbounded-suboptimal approach TC are prohibitively large. Meanhwhile reverse CPD paths have better suboptimality and better quality than FS for most queries, but FS performs better in the worst case.

Heuristic Extraction: We can further report that, without heuristic path extraction, the worst case behavior (as in Figure 6) can result in slightly faster path extraction but worse path quality: without this addition the mean increase for *diff* is approximately around $\delta/3$.

Overall: Based on results above, it is clear that $\texttt{rev}_\delta$ ($\delta \geq 16$) are better for path extraction in terms of tradeoff between suboptimality and extraction time. Each of our nearest competitors suffers from significant drawbacks, either in terms of a large and unbounded suboptimality (*Tree Cache*) or a much slower path extraction (*FOCAL Search*).

	mean	25%	50%	75%	99%	max
$\texttt{rev}_{16}$	0.88	0.00	0.00	1.17	7.41	26.00
$\texttt{FS}_{32}$	2.52	0.00	0.00	2.34	26.08	31.98
$\texttt{rev}_{32}$	1.55	0.00	0.00	1.66	16.68	54.59
$\texttt{FS}_{64}$	5.95	0.00	0.82	8.14	46.54	63.99
$\texttt{rev}_{64}$	4.38	0.00	0.00	3.51	50.65	113.40
$\texttt{FS}_{128}$	13.22	0.00	4.97	19.74	82.04	127.99
TC	143.10	18.97	46.53	138.12	1644.88	3194.47

Table 4: Suboptimality comparison (absolute difference).

	mean	25%	50%	75%	99%	max
$\texttt{rev}_{16}$	0.01	0.00	0.00	0.00	0.08	1.95
$\texttt{FS}_{32}$	0.01	0.00	0.00	0.01	0.15	0.79
$\texttt{rev}_{32}$	0.01	0.00	0.00	0.00	0.15	6.04
$\texttt{FS}_{64}$	0.03	0.00	0.00	0.03	0.20	0.95
$\texttt{rev}_{64}$	0.03	0.00	0.00	0.01	0.50	8.77
$\texttt{FS}_{128}$	0.04	0.00	0.02	0.07	0.26	2.40
TC	0.83	0.06	0.18	0.53	8.62	2005.07

Table 5: Quality comparison (relative difference).

Discussion

At a high level, centroid CPDs precompute information for a subset of nodes to improve path planning; such an idea also appears in Landmark (Goldberg and Harrelson 2004). However, they are very different in terms of the algorithm: (i) Landmarks improve path planning search by providing improved heuristic estimates, while CPDs are search-free path planning techniques that do not depend on heuristics; (ii) Landmark nodes usually do not appear on computed paths while in our case the target centroid almost always appears on computed paths. Recent work (Bono et al. 2019) shows CPDs can be used to derive heuristic estimates in dynamic graphs. This work compares CPD heuristics vs. Landmarks and shows that CPDs offer substantial advantages.

Conclusion and Future Work

CPDs provide a state-of-the-art solution for path planning, but they may be unattractive because of substantial overhead required to construct and store them. In this paper, we explore the use of bounded suboptimal CPDs based on centroids, that massively reduces the precomputation time. We apply the idea to both standard forward CPDs and to the reverse CPDs that we introduce in this work. As the number of centroids decreases, reverse CPDs shrink more aggressively, eventually overtaking forward CPDs in terms of size reduction. Our approach leads to faster path extraction than forward CPDs and, in practice, to a much smaller difference in path length than the suboptimality bound guarantees. Results show an excellent speed vs suboptimality tradeoff compared to other techniques from the literature.

In future work, we plan to improve the compression of reverse CPDs with new types of heuristic symbols. Bounded suboptimal reverse CPDs could also be used as an admissible heuristic in search, for an anytime behavior, in a manner related to recent work that uses standard CPDs for a similar purpose (Bono et al. 2019).

References

Anderson, K. 2012. Tree cache. In *Proceedings of the 5th Annual Symposium on Combinatorial Search (SoCS)*. Niagara Falls, Ontario, Canada: AAAI Press.

Bono, M.; Gerevini, A. E.; Harabor, D. D.; and Stuckey, P. J. 2019. Path planning with CPD heuristics. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 1199–1205. International Joint Conferences on Artificial Intelligence Organization.

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*, 293–297. AAAI Press.

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.

Chiari, M.; Zhao, S.; Botea, A.; Gerevini, A.; Harabor, D.; Saetti, A.; Salvetti, M.; and Stuckey, P. J. 2019. Cutting the size of compressed path databases with wildcards and redundant symbols. In Lipovetzky, N.; Onaindia, E.; and Smith, D., eds., *Proceedings of the 29th International Conference on Automated Planning and Scheduling*, 106–113. AAAI Press.

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)*, volume 8737 of *Lecture Notes in Computer Science*, 321–333. Springer.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, 319–333. Springer.

Goldberg, A. V., and Harrelson, C. 2004. Computing the shortest path: A* search meets graph theory. Technical Report MSR-TR-200, Microsoft Research.

Harabor, D. D., and Grastien, A. 2014. Improving Jump Point Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 128–135.

Pearl, J., and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE Trans. Pattern Anal. Mach. Intell.* 4(4):392–399.

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.; 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.

Verzeletti, M.; Botea, A.; and Zanella, M. 2019. Repairing compressed path databases on maps with dynamic changes. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, 115–124.