Proceedings of the Twelfth International
Symposium on Combinatorial Search (SoCS 2019)

Repairing Compressed Path Databases on Maps with Dynamic Changes

Marco Verzeletti
Department of Information Engineering
University of Brescia, Italy

Adi Botea
IBM Research
Ireland

Marina Zanella
Department of Information Engineering
University of Brescia, Italy

Abstract

Single-agent pathfinding on grid maps can exploit online compiled knowledge produced offline and saved as a Compressed Path Database (CPD). Such a knowledge is distilled by performing repeated searches in a graph, where each node corresponds to a distinct grid cell, typically by algorithms such as Dijkstra's. All-pairs shortest paths (APSPs) are computed, and the first move along a shortest path is persistently stored in the CPD. This way, an optimal move can efficiently be retrieved for any pair of source and target cells that is considered while the agent is navigating. However, a CPD supports a static grid, that is, a grid where each cell is permanently either traversable or non-traversable. Our work instead assumes that the cells in the map can undergo dynamic changes. Reasoning about the altered map would require a new CPD. As creating it from scratch is computationally expensive, we present techniques to repair an existing CPD. We prove that using our technique leads to correct and optimal solutions. Experiments demonstrate the benefits of our approach. When a single obstacle of a given size is added or removed, the repair costs often are a small fraction of a recomputation from scratch.

Introduction

Pathfinding (or *path planning*) in a plane is a core AI task, with several application domains, such as robotics (Lee and Yu 2009; Algfoor, Sunar, and Kolivand 2015), and computer games (Sturtevant 2007; Botea et al. 2013; Algfoor, Sunar, and Kolivand 2015). The problem to be solved is to guide a single-agent navigation from a source to a target on a fully known, static map. Such a map is often discretized into a *grid map*: a partitioning into atomic square *cells*, each of which is either *traversable* or *non-traversable*. In common grid pathfinding the agent can move only along 4 or 8 directions, that is, it can turn either at modulo 90 degrees or 45 degrees. A grid map can be represented as a weighted undirected graph, where each node corresponds to a traversable cell of the grid, and each edge connects two adjacent traversable cells.

Computing optimal (length-minimal) paths and computing the first moves of a minimal path are challenges faced in the Grid-based Path Planning Competition[1] (GPPC) (Sturtevant 2014).

Compressed Path Databases (CPDs) (Botea 2011; Strasser, Harabor, and Botea 2014; Strasser, Botea, and Harabor 2015; Salvetti et al. 2018) achieve a state-of-the-art speed in optimal pathfinding on gridmaps with an approach that avoids graph searches during the computation of a path. In a preprocessing stage, all-pairs-shortest-paths (APSPs) data are computed with repeated calls to Dijkstra's algorithm (Dijkstra 1959), one for each node on the map. The APSP data are compressed into a CPD. After the precomputation, a CPD provides fast an optimal move from any node s towards any target node t.

However, a CPD supports a static grid, that is, a grid where each cell is permanently either traversable or non-traversable. When obstacles on a map change dynamically, a standard CPD precomputation needs to start from scratch. This is slow, especially on large maps, as preprocessing involves many calls to Dijkstra's algorithm. Preprocessing can be parallelized, but the number of CPU cores available often is very limited.

We introduce an approach to dynamically repair a CPD when obstacles appear or disappear from the map. The key idea is to identify a subset of nodes around the changed cells, with the property that running Dijkstra's algorithm for each of those nodes as a source is sufficient to repair the CPD. Often, such a subset of nodes is much smaller than the entire graph, resulting in important speedups compared to a recomputation from scratch. As our technique relies on separate Dijkstra runs, it can also benefit from parallelization.

As mentioned earlier, we assume that the grid maps are undirected (i.e., traveling is allowed on both directions along a given edge). This is a very common assumption in the literature. It allows us to compute, with one single Dijkstra search, two types of moves: the first optimal move from the source node to every other node (target), and the first optimal move from each target towards the source node. The ability to compute both types of moves is key in reducing the CPD repair effort, as illustrated in the fourth section.

The contribution of the paper is threefold. Firstly, we present a study of CPD repairing to react to dynamic changes in the map. To the best of our knowledge, this topic

[1] http://movingai.com/GPPC/index.html

has never been investigated before in the literature. Secondly, we formally prove sufficient conditions that guarantee that a partial repair results in an optimal and correct CPD. That is, our repaired CPDs are free from faults such as providing suboptimal paths, running into an obstacle and, very importantly, running into infinite loops. Finally, we provide experimental evidence that CPD repairing can convincingly outperform a recomputation from scratch.

Related Work

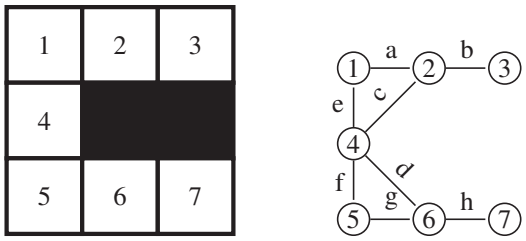
In gridmap pathfinding, many approaches rely on a graph search performed online, while looking for a path. Graph search, however, can be expensive, visiting many more nodes than a set of nodes placed on an optimal path. Research efforts have been spent in identifying better heuristics to guide the search (Sturtevant et al. 2009), which is usually performed with A* (Hart, Nilsson, and Raphael 1968) or variants of it, and reducing the size of the searched portion of the graph (Harabor and Botea 2010; Harabor and Grastien 2011; Uras, Koenig, and Hernández 2013; Rabin and Sturtevant 2016), while preserving the optimality of the resulting path.

Dynamic map changes are related to moving obstacles on maps. Jaillet and Simeon (2004) construct a cycle-free map for the static part of the environment. If, in a query, an edge crucial for finding a path intersects with a moving obstacle, a variant of the Rapidly-exploring Random Trees (RRT) approach (Lavalle and Kuffner 2000) is used to reconnect the vertices of this edge. If this procedure fails, an attempt is made to reconnect the two disconnected components of the map by additional global sampling. The method aims at solving the problem in the query phase rather than in the preprocessing phase.

Nieuwenhuisen, van den Berg, and Overmars (2007) present an algorithm that creates a robust map in the preprocessing phase. Such an algorithm is based on the observation that the motion of the moving obstacles is often restricted to some confined area. Examples of such obstacles are a door that can be opened or closed, or a chair whose position is bounded to a room. van den Berg et al. (2005) introduce an algorithm that assumes that a moving obstacle has a predefined set of potential placements. The work presented in this paper does not make any assumption about the positions where an obstacle can be added or removed.

A common approach used for efficient path planning in dynamic environments involves modeling moving obstacles as static objects with a small window of high cost around the beginning of their projected trajectories (Likhachev and Ferguson 2009). By avoiding the additional time dimension, these approaches can efficiently find paths that do not collide with any obstacles in the near future. However, they suffer from severe sub-optimality or even incompleteness due to the uncertainty of moving obstacles in the future.

Sturtevant (2011) uses incremental repairing for a sparse representation of three-dimensional grids. Repairing preprocessed data, in response to dynamic changes, has also been addressed in transportation networks. See (Bast et al. 2015) for a survey.



(a) 8-connected grid map (b) Graph

Figure 1: Grid map and the corresponding graph.

Vemula, Muelling, and Oh (2016) successfully use the notion of adaptive dimensionality in high-dimensional motion planning, while such a notion is not adopted in the context of path planning in dynamic environments.

Background

The result of dividing a plane navigation environment into $N \times M$ square cells is called a *grid map*. Each cell must be either fully traversable or fully non-traversable. If the agent is able to move only along four straight directions (North, East, South, West), the grid map is *4-connected*; if instead the agent can move also along the diagonal directions (North-East, North-West, South-East, South-West), the grid map is *8-connected*. Moves involving non-traversable cells are disallowed.

A grid map has an associated graph, where each traversable cell is a node of the graph, and adjacent cells are adjacent nodes, connected with edges. For a 4-connected map, the out-degree of a node in the corresponding graph is up to 4, while it is up to 8 in case of an 8-connected map. Figure 1a shows a small grid map, where the black cells are non-traversable, and Figure 1b the graph obtained after its conversion.

Consider a 2×2 area on an 8-connected grid. If two diagonal cells are traversable, but the other two are blocked, no diagonal move is defined for the traversable cells. For example, in Figure 1a, if cell 5 were black, no diagonal move could exist between cells 4 and 6, as an agent would have no room to squeeze through two diagonally-adjacent obstacles.

Observe that Figure 1 allows a diagonal move such as 4 to 6, despite the fact that one nearby cell (a cell adjacent to both cells 4 and 6) is blocked. Some works in the literature allow such a diagonal move near an obstacle, and some do not. Our approach works with either assumption.

The grid maps often are assumed to have uniform costs:[2] the cost of every straight move from a (traversable) cell to a neighbor (traversable) cell is 1, whereas the cost of every diagonal move is $\sqrt{2}$. Such costs become the weight of the edge of the undirected graph corresponding to the map. Once such a graph G has been built, the preprocessing phase, meant to offline knowledge compilation, is carried out. The construction of the CPD is performed in two steps repeated for each node, encompassed by Algorithm 1: shortest-path computations, and data compression.

[2]CPDs work both with and without such an assumption.

	1	2	3	4	5	6	7
1	*	e	e	s	s	s	s
2	w	*	e	sw	sw	sw	sw
3	w	w	*	w	w	w	w
4	n	ne	ne	*	s	se	se
5	n	n	n	n	*	e	e
6	nw	nw	nw	nw	w	*	e
7	w	w	w	w	w	w	*

(a) Uncompressed

1	1/e	4/s		
2	1/w	3/e	4/sw	
3	1/w			
4	1/n	2/ne	5/s	6/se
5	1/n	6/e		
6	1/nw	5/w	7/e	
7	1/w			

(b) Compressed with RLE

Figure 2: First-move matrix.

Algorithm 1 CPD, preprocessing phase

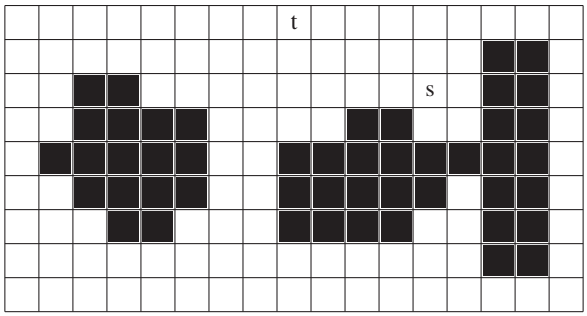
1: **procedure** CPD($G = (V, E)$)
2: **for each** n $\in V$ **do**
3: $R(n) \leftarrow$ Dijkstra(n)
4: $C(n) \leftarrow Compress(R(n))$

Repeated calls to Dijkstra's algorithm (line 3 of Algorithm 1) is used for computing APSPs, where the source node ranges over all the nodes in the graph (line 2). The original Dijkstra's algorithm returns the distance from the source to every target. The version of Dijkstra's algorithm exploited for building the CPD, introduced in (Botea 2011), not only returns the distance, but also provides all the first (optimal) moves on the shortest paths from the source to any target. The first move on a shortest path from a source node n to any target node in graph $G = (V, E)$ is stored in a single row $R(n)$ of a $|V| \times |V|$ matrix, called a *first-move matrix*. After that, row $R(n)$ is compressed (line 4), by adopting, for instance, the Run-Length-Encoding (RLE) technique, which allows to compactly represent strings (Strasser, Harabor, and Botea 2014; Strasser, Botea, and Harabor 2015).

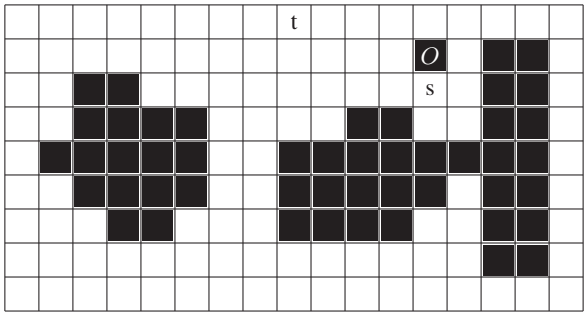
Given a string of symbols, a run consists of a maximal solid block repetition of the same symbol. String $\alpha = aaabbaaccc$, for instance, has four runs: aaa, bb, aa and ccc. RLE encodes it as $1/a$, $4/b$, $6/a$ and $8/c$. Each compressed run indicates the starting position and the symbol contained in the run.

Example Consider the grid map shown in Figure 1a, where $\mid V \mid = 7$. After having built the search graph relevant to this map, depicted in Figure 1b, the first-move matrix m is computed (Figure 2a). Optimal moves are represented as follows: e = East, se = South-East, s = South, ... ne = North-East. For instance, $m[1, 4] =$ s because going South is an optimal move from 1 towards 4. All elements $m = [i, i]$ are "don't care" symbols $*$, since no move is needed to go from a cell to itself. The (uncompressed) matrix has $7 \times 7 = 49$ elements, 7 of which are "don't care" symbols $*$.

Each $*$ symbol is replaced with an element that is suitable to achieve a better compression. For example, in row 3 of m, by replacing the asterisk with w we obtain only one run. The compressed first-move matrix, shown in Figure 2b, includes 16 runs altogether, a number which is smaller than the size of the original first-move matrix. Each run indicates the starting node and the first move. For instance, run $1/e$ in the first row refers to the starting node 1 and its first move, East. The



(a) Grid map



(b) The same grid with an added obstacle

Figure 3: Two grid maps differing for an obstacle cell.

compressed first-move matrix is precisely the CPD.

Definition 1. *(Distance) The distance between nodes A and B is the cost of the shortest path from A to B (or, dually, from B to A) in the considered graph, denoted $d(A, B)$ or, indifferently, $d(B, A)$.*

We recall here two well-known properties of graphs.

Property 1. *(Triangle inequality) In a graph G where the cost of every edge is strictly positive, for every triple of nodes (A, B, C), the sum of the distances between any two pairs of nodes is greater than or equal to the distance of the remaining pair, namely $d(A, B) + d(B, C) \geq d(A, C)$.*

Property 2. *(Acyclicity of optimal paths) In a graph where the cost of every edge is strictly positive, every optimal path is cycle-free.*

Repairing CPDs

In the following, suffix *old* will be used to denote an entity (grid, graph, CPD, distance, etc.) before any dynamic change, and suffix *new* to denote the same entity after the dynamic change has occurred. In this work we focus on dynamic changes of two types: *adding an obstacle* and *removing an obstacle* of a given size.

Consider, for example, the 4-connected grid map in Figure 3a, where the cells marked with s and t are the source and the target, respectively. The optimal moves from s to t are North and East, and either of them can be chosen to be stored in the CPD. In our example, we assume that the chosen move is North.

After an obstacle has been added onto the map (it is the single cell marked with O in Figure 3b), the move North

from s to t is not available any more. The example shows that, after a change in the map, some repairs are needed in order for CPD$_{new}$ to guarantee the optimality of every path in the new graph.

We aim at finding all the nodes s for which there exists some pair (s, t) whose first move in CPD$_{old}$ is not optimal any more, and apply to such nodes what we call a CPD repair. For each and every one of such nodes, a CPD repair consists in replacing the moves, that were optimal in the old graph, with moves that are optimal in the new graph.

Given a set of nodes $\mathcal{X}$, this being a subset of the nodes of the old graph, the CPD is repaired by Algorithm 2.

Algorithm 2 Repair CPD, given a list of nodes $\mathcal{X}$

```
1: procedure REPAIRCPD(X)
2:     for each node x ∈ X do
3:         Dijkstra(x)
4:         SingleSourceRepair(x)
```

In line 3, Dijkstra's algorithm is run for a source node x. The implemented version of Dijkstra's algorithm returns not only the distance between x and all the other nodes, but also the first optimal moves.

In a standard CPD, a Dijkstra run is used to get first moves from the source towards any target, but *not* first moves from the targets towards the source at hand.

The version of Dijkstra's algorithm implemented in our repair approach retrieves first optimal moves in *both* directions: from the source towards the targets, and from the targets towards the source. This is a key idea that allows us in the end to repair a CDP with only a small number of calls to Dijkstra's algorithm. An optimal first move from a target towards the source is obtained as follows: we keep track of the last move along an optimal path from the source to the target, and take its opposite (e.g., if the last move is North, the opposite move is South).

Given an undirected graph, the opposite of the last move of an optimal path from s to t is guaranteed to be the first move of an optimal path from t to s. This is why we rely on the assumption that the input gridmap is undirected. As said in the introduction, this is a very common property of gridmaps considered in the literature and in practice.

When several optimal moves exist from one node towards another (i.e., from the current node towards a target, or from a target towards the current node), the Dijkstra algorithm can record them all. Then, in the compression stage, we can choose a move that would better help with the compression.

Algorithm 3 Repair all CPD entries relevant to node $x \in \mathcal{X}$

```
1: procedure SINGLESOURCEREPAIR(x)
2:     for each node v ∈ V do
3:         update CPD[x, v]
4:         update CPD[v, x]
```

Algorithm 3 repairs the entries relevant to a single node in the CPD. Let CPD$[a, b]$ represent the first optimal move from node a towards node b. In line 3, the old first move from node x towards v is replaced with the new optimal first move. In line 4, the old first move from node v towards x is overwritten with the new optimal first move.

CPD$_{old}$

	·	s	·	t	·
·	·	·	·	·	·
s	·	·	·	north	·
·	·	·	·	·	·
t	·	east	·	·	·

CPD$_{new}$

	·	s	·	t	·
·	·	·	·	·	·
s	·	·	·	**west**	·
·	·	·	·	·	·
t	·	east	·	·	·

Figure 4: CPD repair example for the (s, t) pair in Figure 3.

In the example in Figure 3, the CPD entries relevant to source node s have to be repaired. Figure 4 shows an excerpt of CPD$_{old}$ and CPD$_{new}$. To easily follow the example, the figure illustrates uncompressed first-move matrices. Furthermore, we show the moves only for the two nodes at hand, s and t. Adding an obstacle cell has been depicted as the transition from the map in Figure 3a to the one in Figure 3b. Removing an obstacle is the dual change, which can be depicted as the transition from Figure 3b to Figure 3a. It is quite intuitive to accept the following property.

Property 3. *If a traversable cell in a map becomes an obstacle, then the distance between any two cells cannot decrease.*

Property 3 dually means that, if an obstacle cell becomes traversable, then the distance between any two cells cannot increase. We present a method to find the set of nodes $\mathcal{X}$: the $\mathcal{P}$ list is a concrete instantiation of $\mathcal{X}$, being a superset of the nodes that need a repair. In other words, $\mathcal{P}$ is a subset of nodes of the old graph that is sufficient to guarantee the correctness and the optimality of a repaired CPD, when $\mathcal{P}$ is used instead of $\mathcal{X}$ in Algorithm 2.

Dynamically Adding Obstacles

Before defining list $\mathcal{P}$, we introduce a notion that will be exploited in the definition of the list $\mathcal{P}$.

Definition 2. *(Border) The border $\mathcal{B}$ of a connected sub-graph $\mathcal{M}$ of a graph $G = (V, E)$ is*

$$\mathcal{B} = \{m \in \mathcal{M} \mid \exists v \in V \backslash \mathcal{M} \text{ s.t. } (m, v) \in E\}.$$

Notice that, if graph G corresponds to a grid map, then a connected sub-graph $\mathcal{M}$ represents a region of the map, where each cell in the region is reachable starting from any other cell belonging to the same region. The border $\mathcal{B}$ of the sub-graph corresponds to the cells in the region that have a neighbor cell that does not fall in the region.

Now we can switch back to the introduction of a $\mathcal{P}$ list.

Definition 3. *($\mathcal{P}$ list) Let $G_{old} = (V, E)$ be the graph corresponding to a grid map, and G_{new} be the graph corresponding to the same grid map once it has been changed. We call a $\mathcal{P}$ list a set of nodes in G_{new}, whose border is denoted as $\mathcal{B}$, such that, for every pair of nodes including a node $b \in \mathcal{B}$ in the border and a node $v \in V \backslash \mathcal{P}$ that does not belong to $\mathcal{P}$, the distance between them in G_{new} is equal to the distance in G_{old}, namely $d_{new}(b, v) = d_{old}(b, v)$.*

Algorithm 4 shows a method to build a $\mathcal{P}$ list, after one cell has become an obstacle. Let O be the node in G_{old} corresponding to that cell. First, the region is initialized with the neighbors of O.[3] For each node $p \in \mathcal{P}$, the distance in G_{new} between p and any node in $V \backslash \mathcal{P}$ is computed: if the distance is the same as in G_{old}, DISTUNCHANGED(p) (line 6) returns *true*, hence node p is added to border set $\mathcal{B}$ (line 7). Otherwise, each neighbor of node p is added to list $\mathcal{P}$, unless already there (line 10).

Algorithm 4 Computing a $\mathcal{P}$ list

```
1:  procedure BUILDPLIST(N_O)
2:      P ← N_O          ▷ Initialize P to all neighbors of the
    obstacle, and mark the neighbors as unprocessed
3:      while P has unprocessed elements do
4:          p ← getUnprocElem(P)          ▷ Return an
    unprocessed elem. from P, without removing it from P
5:          mark p as processed
6:          if DISTUNCHANGED(p) then
7:              add(B, p)                 ▷ add node p in B
8:          else
9:              for each neighbor n of p do
10:                 add(P, n)   ▷ Add node n to P, unless
    already there. Every newly added node to P is marked
    as unprocessed.
11:     return P
```

Algorithm 5 Find out whether distances from a node p to all nodes in $V \backslash \mathcal{P}$ have changed

```
1: procedure DISTUNCHANGED(p)
2:     Dijkstra_old(p)        ▷ Dijkstra run in the old graph
3:     Dijkstra_new(p)        ▷ Dijkstra run in the new graph
4:     for each node v ∈ V\P do
5:         if d_new(p, v) ≠ d_old(p, v) then
6:             return false
7:     return true
```

With reference to the example in Figure 3, the nodes marked with $+$ in Figure 5 represent region $\mathcal{P}$, which is small compared to the entire map.

We prove that it is sufficient to repair (by exploiting Algorithms 2 and 3) all nodes that belong to $\mathcal{P}$ in order to guarantee that CPD$_{new}$ gives an optimal path between any pair of nodes in the new graph.

Theorem 1. *For every source node s and target t, CPD$_{new}$ gives an optimal path from s to t in the new graph after an obstacle cell has been added.*

Proof. Proving that CPD$_{new}$ gives an optimal path for all pairs of nodes (s, t) amounts to proving that CPD$_{new}[s, t]$ is optimal in the new graph $G_{new} = (V, E)$. The following two cases cover all situations.

Case A: $t \in \mathcal{P}$.

[3] O implicitly falls in the region; however, as it is now an obstacle, it is not appended to list $\mathcal{P}$ since it does not have to be repaired.

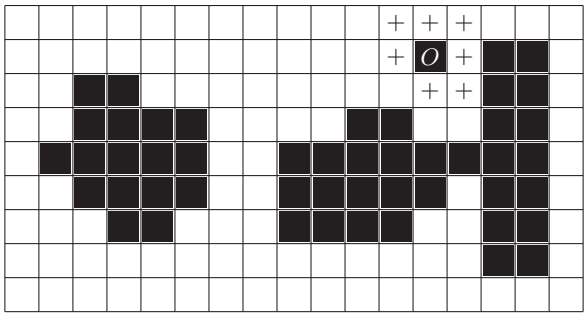


Figure 5: Region $\mathcal{P}$ for added obstacle O.

List $\mathcal{P}$ is the input parameter of Algorithm 2. Hence, Dijkstra's algorithm is run for the new graph, assuming t as the source. Dijkstra's algorithm computes all the minimal paths from t to whichever node in V, then also the minimal paths from t to s. This guarantees that both CPD$_{new}[t, s]$ and CPD$_{new}[s, t]$, as repaired by the call of Algorithm 3, are optimal.

Case B: $t \notin \mathcal{P}$.

B_1: For all nodes h on the path from s to t given by CPD$_{old}$, condition $h \notin \mathcal{P}$ holds. It follows that CPD$_{new}$ gives exactly the same moves as CPD$_{old}$ for this path, and that the length of the path in the new graph is the same as in the old graph. From Property 3 it follows that the path is optimal in the new graph, hence move CPD$_{new}[s, t]$ is optimal too.

B_2: There exists a node $h \in \mathcal{P}$ on the path from s to t given by CPD$_{old}$. Let h be the first node that belongs to $\mathcal{P}$ along the path. Let $\pi = \langle s, ..., h, ..., t \rangle$ be such a path. This means that $h \in \mathcal{B}$, where $\mathcal{B}$ is the border of $\mathcal{P}$. Based on Definition 3, the distance between s and h in the new graph is the same as the distance in the old graph. Also the distance between h and t in the new graph is the same as the distance in the old graph, because $h \in \mathcal{B}$ and $t \notin \mathcal{P}$. Namely $d_{old}(s, h) = d_{new}(s, h)$ and $d_{old}(h, t) = d_{new}(h, t)$.
The following conclusions can be drawn:

1. From node s towards h, CPD$_{new}$ gives the same first moves for every node as CPD$_{old}$ along this path. Since no repair has been done for this path, it follows that CPD$_{new}$ gives an optimal path from s to h too.
2. From node h towards t, the proof follows directly from Case A above since $h \in \mathcal{P}$.

□

The $\mathcal{P}$ list is related to the concept of a *swamp* (Pochter et al. 2010). A swamp is a region on a map (or, more generally, a subgraph of a graph) with the property that any two nodes that do not belong to the swamp can be connected with an optimal path that does not intersect the swamp. The definition suggests that swamps could possibly be used to speedup our repair process. For example, we could identify swamps on a map in a preprocessing step. Then, if an obstacle is dynamically added inside a swamp, we could exploit the fact that distances between nodes that do not belong to

the swamp remain unchanged. We leave the exploration of this idea as future work.

Dynamically Removing Obstacles

The same definition given for list $\mathcal{P}$, when an obstacle cell is added in the map, can be used when a non-traversable cell becomes traversable. Also, $\mathcal{P}$ is built as in the case of an added obstacle, the only difference being that, when removing obstacles, the cells that have become traversable are added to the $\mathcal{P}$ set. The first optimal moves from the new traversable cell F to any other traversable cell and, vice versa, from any other traversable cell to F have to be computed and saved in CPD$_{new}$. Hence the new node F has to be added in list $\mathcal{P}$, and the CPD has to be repaired for all the nodes in $\mathcal{P}$.

Theorem 2. *For every source node s and target t, CPD$_{new}$ gives an optimal path from s to t in the new graph once an obstacle cell has been removed.*

Proof. The following two cases cover all situations.
Case A: $t \in \mathcal{P}$.

The proof is the same as for Case A of Theorem 1.

Case B: $t \notin \mathcal{P}$.

B_1: For all nodes x on an optimal path from s to t in the new graph, condition $x \notin \mathcal{P}$ holds. Let π be an optimal path in the new graph from s to t. It follows that path π does not pass through the new node F. Let π_1 be an optimal path in the old graph from s to t. According to Property 3, π_1 cannot be shorter than π because the old graph has one more obstacle than the new graph. Hence an optimal path in the old graph has the same length as an optimal path in the new graph. It follows that the path given by CPD$_{old}$ has the same length as the path given by CPD$_{new}$. No node in the path belongs to $\mathcal{P}$, so no repair is needed for the nodes in this path. It follows that CPD$_{new}$ gives exactly the same moves as CPD$_{old}$, and the path is optimal in the new graph.

B_2: There exists a node $h \in \mathcal{P}$ on the path from s to t given by CPD$_{new}$. Let h be the first node that belongs to $\mathcal{P}$ along the path. Let $\pi = \langle s, ..., h, ..., t \rangle$ be such a path. This means that $h \in \mathcal{B}$, where $\mathcal{B}$ is the border of $\mathcal{P}$. Based on Definition 3, the distance between s and h in the new graph is the same as the distance in the old graph. Also the distance between h and t in the new graph is the same as the distance in the old graph, because $h \in \mathcal{B}$ and $t \notin \mathcal{P}$. Namely $d_{old}(s,h) = d_{new}(s,h)$ and $d_{old}(h,t) = d_{new}(h,t)$.
The following conclusions can be drawn:

1. Along the path from node s towards h, CPD$_{new}$ gives the same first moves for every node as CPD$_{old}$. Since no repair has been done for this path, it follows that CPD$_{new}$ gives an optimal path from s to h too.
2. For the nodes from h towards t along the path, the proof follows directly from Case A above since $h \in \mathcal{P}$.

□

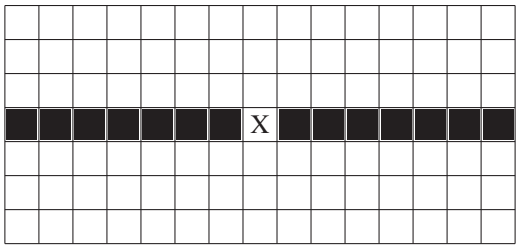


Figure 6: An example where $\mathcal{P}$ can cover the entire map, if the cell marked with X changes its status from traversable to blocked, or from blocked to traversable.

Worst Case Behavior

In the worst case, the $\mathcal{P}$ list, as defined and computed in this paper, grows linearly with the map size, and can even cover the entire map. Think, for instance, of a two-room map, where the rooms communicates through a door, as illustrated in Figure 6. An added obstacle closes such a door (location X on the map). Or, dually, removing an obstacle from X opens a door between the two rooms. In such cases, $\mathcal{P}$ includes all the traversable cells in the grid (and the border $\mathcal{B}$ is empty).

This example also suggests that, in certain particular cases, one could bypass the computation of the $\mathcal{P}$ list. For example, when cell X becomes blocked in Figure 6, it creates two disjoint connected components. Basically, there is no need to repair the CPD, as all moves inside one connected component remain correct and optimal.[4] This topic, however, is beyond the focus of this work. We chose this example to illustrate a case where $\mathcal{P}$ grows large, and we preferred it to other examples for its simplicity.

Experimental Results

In this section we outline the experimental setup, followed by an analysis of the results when obstacles are dynamically added, and the results when obstacles are dynamically removed from the map.

Experimental Setup

Three different types of maps, including real commercial game maps and artificial benchmarks, have been used to evaluate the performances of the proposed CPD repairing technique. The maps, all downloaded from http://movingai.com/benchmarks/ (Sturtevant 2012), were selected from *Games* (*Dragon Age: Origins, Warcraft III*), *Mazes* and *Rooms*. Specifically, four game maps were used, ranging in size from 40,392 to 115,010 nodes (traversable cells). Two room maps were considered, with room sizes of 8×8 and 32×32 cells. They have 51,553 and 60,645 nodes, respectively. The test maps further include two mazes: one with corridors of width 1, and 32,707 traversable cells, and one with corridors of width 8, with 58,264 traversable cells.

[4] It's easy to ensure that the CPD is never queried for a disconnected pair of nodes, by checking first if they belong to the same connected component.

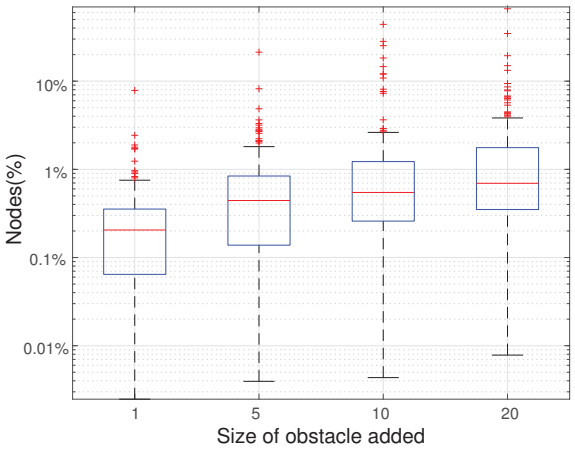


Figure 7: Size of list $\mathcal{P}$ for an added obstacle in game maps.

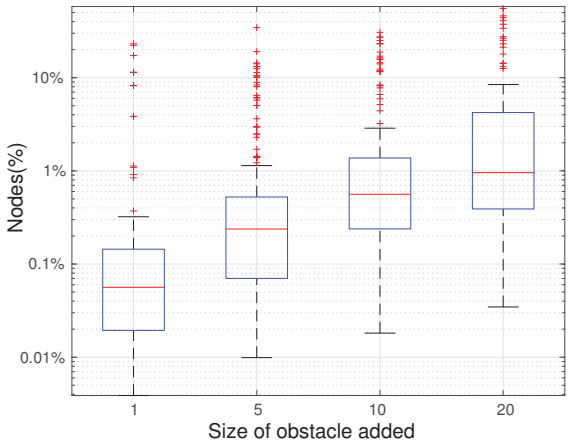


Figure 8: Size of list $\mathcal{P}$ for an added obstacle in room maps.

Every obstacle was added to or removed from each map 50 times in a random position, to evaluate the performance.

Experimental results are displayed as box-plots. A box-plot depicts groups of numerical data through their quartiles. In each box, the central mark indicates the median, and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively. The whiskers extend to the extreme data points that are not considered as outliers, while the outliers are plotted individually using symbol $+$.

Adding an Obstacle

Figures 7 and 8 show the size of list $\mathcal{P}$ when an obstacle has been added to the game maps and room maps, respectively. We consider obstacles of 4 distinct sizes: 1, 5, 10, and 20 cells respectively. The x axis shows the obstacle size. The y axis shows the size of the list $\mathcal{P}$, as a percentage of the total number of nodes. Notice the logarithmic scale on the y axis. The median number never exceeds 1%, regardless of the obstacle size, and often is significantly lower than 1%. As expected, the size of list $\mathcal{P}$ increases with the size of the obstacle. Figures 7 and 8 show results for 4-connected grids.

Table 1: Summary statistics for the size of the list $\mathcal{P}$, in the case of added obstacles.

	4-connected		8-connected	
Maps	Min	Mean	Min	Mean
Games	1 node	1.20 %	1 node	2.60 %
Rooms	1 node	2.68 %	1 node	5.46%
Mazes	1 node	2.11 %	1 node	2.52%

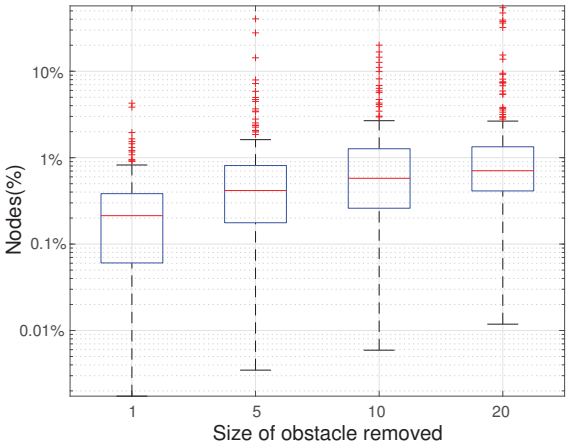


Figure 9: Size of list $\mathcal{P}$ for a removed obstacle in game maps.

Results on 8-connected grids lead to a similar conclusion.

Table 1 summarizes the results of the tests with added obstacles. For each combination of a type of maps (Games, Rooms, Mazes) and a connectivity style (4-connected, 8-connected), we show the minimum and the average size of $\mathcal{P}$, across all maps in that category, and all obstacle placements on each map. The best case is when $\mathcal{P}$ contains only one node, which occurs when the added obstacle has only one neighbor. Mean values are low single-digit percentages, with values slightly larger in the case of 8-connected maps.

In summary, as displayed in Figures 7 and 8, and in Table 1, $\mathcal{P}$ often is much smaller than the set of nodes V. This shows that the proposed repairing process can conveniently replace the recomputation of the CPD.

Removing an Obstacle

Figures 9 and 10 plot the size of the list $\mathcal{P}$ when an obstacle has been removed from games maps and room maps, respectively. As in the case of adding obstacles, the median values do not exceed 1% and often are much smaller, especially for smaller obstacle sizes. Figure 11 plots *mean* values nodes in list $\mathcal{P}$ for games maps and room maps. The figures show that the $\mathcal{P}$ list is a small percentage of the entire graph. The percentage increases with the size of the obstacle removed. It grows up to 10% in the figure (Rooms, 20-cell obstacles), and it is significantly smaller in games maps (all obstacles sizes) and in Rooms for smaller obstacles. (Note the logarithmic scale in the figure.)

Table 2 displays summary statistics in a similar fashion to Table 1, but this time for removed obstacles instead of added obstacles. In the best case, which occurs when the re-

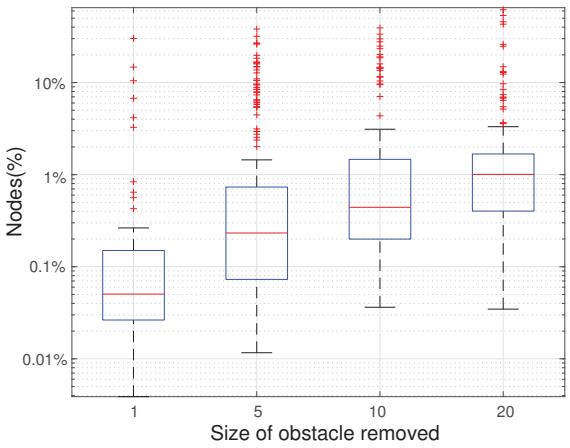


Figure 10: Size of list $\mathcal{P}$ for a removed obstacle in room maps.

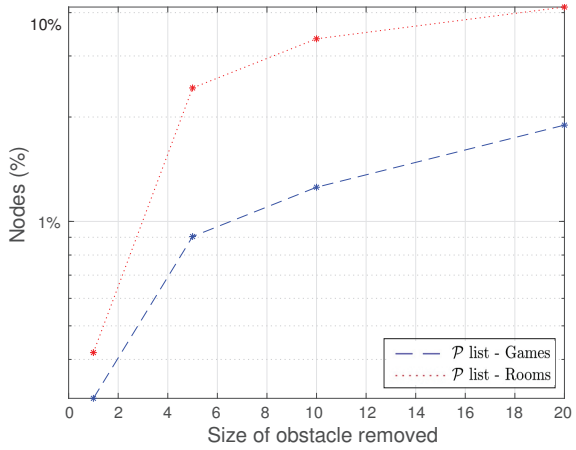


Figure 11: Mean size of list $\mathcal{P}$ for a removed obstacle in game maps and room maps.

moved obstacle has only one neighbour, the list $\mathcal{P}$ contains 2 nodes: the new added node and its neighbour. The table too shows that the average case is a small percentage of the total number of nodes. It varies from 1.33% (4-connected games maps) to 7.40% (8-connected mazes). Interestingly, the results are somewhat stronger for games maps as compared to synthetic maps such as rooms and mazes. This is a tendency observed, for example, in Tables 1 and 2 and in Figure 11.

A straightforward implementation of our repair technique requires running Dijkstra's algorithm twice per repaired node, once in the old graph and once in the new graph (see Algorithm 5). The Dijkstra run on the new graph is also reused for the repair of the node at hand. Thus, CPU time ratios are doubled compared to the $\mathcal{P}$ percentages reported in this section. In future work, part of the Dijkstra search effort could be avoided (e.g., return from Algorithm 5 as soon as a difference is encountered).

Table 2: Summary statistics with the average size of list $\mathcal{P}$, in the case of removed obstacles.

	4-connected		**8-connected**	
Maps	Min	Mean	Min	Mean
Games	2 nodes	1.33 %	2 nodes	2.54 %
Rooms	2 nodes	2.70 %	2 nodes	4.10%
Mazes	2 nodes	3.36 %	2 nodes	7.40%

Conclusions

CPDs are a state-of-the-art approach to *pathfinding*, a core AI problem. A software system based on CPDs requires a preprocessing phase to compute and compress APSPs: the preprocessing time can be heavy, especially on large maps. While very successful in many applications, doing most of the work in the offline knowledge compilation phase restricts the environment to be static.

In this work we have focused on repairing an existing CPD once a change has occurred in the original map. The changes on which our attention has focused are the addition and the removal of one obstacle. A technique for repairing the first moves in the CPD is proposed, which relies on the creation of a list of nodes (called list $\mathcal{P}$) whose first moves, stored in the CPD, might not be optimal after the occurred change. Experimental evidence demonstrates that the proposed repairing technique can be significantly faster than a recomputation of the CPD from scratch, above all in large maps and when small changes have occurred.

Future work includes algorithmic improvements (e.g., the integration of swamps, mentioned earlier), and an extended evaluation. One experiment would focus on dynamic changes at cells with a high (vertex) reach (Gutman 2004), to evaluate if turning their status from traversable to blocked and vice versa involves a larger number of cells whose relevant entries in the CPD need to be repaired.

So far, approaches to pathfinding that adopt a CPD to build an optimal path have been regarded as antithetic with respect to approaches that search for an optimal path online only, without any support from compiled knowledge produced offline. See (Baier et al. 2015) for a notable exception. A challenge for future research is making online search cooperate with (online) CPD building and/or repairing. A whole spectrum of combined approaches can be envisaged: for instance, the information produced by online searches for optimal paths could be exploited to incrementally update the CPD content, and, as soon as a change has dynamically occurred in a map, the CPD could be (partially) repaired.

Acknowledgments

This work has been performed in part when the main author was an intern at IBM Research, Ireland. We thank the anonymous reviewers for their insightful comments.

References

Algfoor, Z. A.; Sunar, M. S.; and Kolivand, H. 2015. A comprehensive study on pathfinding techniques for robotics

and video games. *International Journal of Computer Games Technology*.

Baier, J.; Botea, A.; Harabor, D.; and Hernandez, C. 2015. A fast algorithm for catching a prey quickly in known and partially known game maps. *Computational Intelligence and AI in Games, IEEE Transactions on* 7(2).

Bast, H.; Delling, D.; Goldberg, A. V.; Müller–Hannemann, M.; Pajor, T.; Sanders, P.; Wagner, D.; and Werneck, R. F. 2015. Route planning in transportation networks. Technical Report abs/1504.05140, ArXiv e-prints.

Botea, A.; Bouzy, B.; Buro, M.; Bauckhage, C.; and Nau, D. 2013. Pathfinding in Games. In Lucas, S. M.; Mateas, M.; Preuss, M.; Spronck, P.; and Togelius, J., eds., *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 21–31.

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

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

Gutman, R. J. 2004. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, 100–111. SIAM.

Harabor, D., and Botea, A. 2010. Breaking path symmetries on 4-connected grid maps. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 33–38. Palo Alto, California: AAAI Press.

Harabor, D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 1114–1119. San Francisco, California: AAAI Press.

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

Jaillet, L., and Simeon, T. 2004. A prm-based motion planner for dynamically changing environments. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 2, 1606–1611 vol.2.

Lavalle, S. M., and Kuffner, J. J. 2000. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, 293–308.

Lee, J., and Yu, W. 2009. A coarse-to-fine approach for fast path finding for mobile robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5414–5419. St. Louis, Missouri: IEEE.

Likhachev, M., and Ferguson, D. 2009. Planning long dynamically feasible maneuvers for autonomous vehicles. *Int. J. Rob. Res.* 28(8):933–945.

Nieuwenhuisen, D.; van den Berg, J.; and Overmars, M. 2007. Efficient path planning in changing environments. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3295–3301.

Pochter, N.; Zohar, A.; Rosenschein, J. S.; and Felner, A. 2010. Search space reduction using swamp hierarchies. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Rabin, S., and Sturtevant, N. R. 2016. Combining bounding boxes and JPS to prune grid pathfinding. In *Proceedings of the Thirtieh AAAI Conference on Artificial Intelligence*, 746–752. Phoenix, Arizona: AAAI Press.

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 2018, Delft, The Netherlands, June 24-29, 2018.*, 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 Symposium on Combinatorial Search (SoCS'14)*.

Sturtevant, N. R.; Felner, A.; Barer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, 609–614. Pasadena, California: AAAI Press.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *Proceedings of the Third AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 31–36. Stanford, California: AAAI Press.

Sturtevant, N. 2011. A sparse grid representation for dynamic three-dimensional worlds. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 73–78.

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

Sturtevant, N. R. 2014. The grid-based path-planning competition. *AI Magazine* 35 (3):66–69.

Uras, T.; Koenig, S.; and Hernández, C. 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the Twenty-Third International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'13, 224–232. AAAI Press.

van den Berg, J. P.; Nieuwenhuisen, D.; Jaillet, L.; and Overmars, M. H. 2005. Creating robust roadmaps for motion planning in changing environments. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1053–1059.

Vemula, A.; Muelling, K.; and Oh, J. 2016. Path Planning in Dynamic Environments with Adaptive Dimensionality. *ArXiv e-prints*.