Fast, Optimal Pathfinding with Compressed Path Databases

Adi Botea[*]
IBM Research, Dublin, Ireland

Introduction

Most existing pathfinding methods are based on runtime search. Numerous enhancements have been introduced in recent years, including hierarchical abstraction (Botea, Müller, and Schaeffer 2004; Sturtevant and Buro 2005), more informed heuristics (Björnsson and Halldórsson 2006; Cazenave 2006; Sturtevant et al. 2009), and symmetry reduction (Harabor and Botea 2010; Harabor and Grastien 2011). Recent work in real-time heuristic search often reduces a search to a series of fast, backtrack-free hill climbings (Bulitko, Björnsson, and Lawrence 2010). Despite such a progress, search-based methods could explore, in bad cases, large portions of a map, with a larger running time.

We take a significantly different approach from search-based methods. Given a map, our program, named Copa,[1] precomputes all-pairs shortest paths (APSP) information. A naive storage, with one entry for every start–target pair, is impractical for all but small graphs. APSP data are compressed, reducing the size dramatically (e.g., by two orders of magnitude), with no information loss. At runtime, Copa looks up optimal moves one by one until a complete (or partial, if desired) optimal path is retrieved.

Advantages of compressed path databases (CPDs) include the speed of retrieving individual moves and full paths. The first move lag is very low, as each move is retrieved independently from subsequent moves. In contrast, search-based methods might have to wait until the goal is reached, before making a move in a provably good direction. CPD paths are optimal. On the other hand, CPDs require a significant preprocessing time and memory.

CPDs require that the input graph is a spatial network (i.e., nodes have x and y coordinates). The compression exploits a powerful property of many spatial networks, previously observed and used by Sankaranarayanan, Alborzi, and Samet (2005) in their SILC framework. In a spatial network, it is often the case that the first move along an optimal path, from a start location s to any target t in a contiguous remote area, is the same. The original authors coined this property as *path coherence*. Given a pair (s, t), let the *first-move label* of t identify an optimal move to take in s towards t. Compression involves finding, for a given s, a contiguous area A where the all targets $t \in A$ have the same first move label. Then, instead of storing many identical first-move records for each (s, t) pair, store a single record for (s, A).

A significant difference between SILC and Copa is in the compression method. To identify areas such as A in the previous example, SILC performs a quad-tree map splitting. Copa uses rectangles of arbitrary sizes and locations, with more flexibility in reducing the number of rectangles and thus improving the compression (Botea 2011). Furthermore, Copa contains a number of additional enhancements, such as *list trimming*, run-length encoding, and sliding window compression (Ziv and Lempel 1977), which were added after the initial publication (Botea 2011). The latter two, which are well known compression techniques, are not included in the competition version of Copa. Part of the reason is that, while effective on larger graphs, they have no positive impact on grid maps with sizes up to 300x300. We give a very light, informal overview of our list trimming idea. A more detailed description is included in a separate report.

System Overview

Algorithm 1 Independent iterations in building a CPD.

```
for each $n \in V$ do
    $T(n) \leftarrow$ Dijkstra($n$)
    $L(n) \leftarrow$ Compress($T(n)$)
```

Let $G = (V, E)$ be an input graph. A compressed path database is an array of lists of rectangles, with up to 4 lists for each node $n \in V$. Building a compressed database requires a series of iterations, one for each node n, as illustrated in Algorithm 1. Being independent, the iterations can be run in parallel, with a speed-up linear in the number of available processors. A slight modification of the Dijkstra algorithm produces a so-called *first-move table* $T(n)$, using n as an origin point. In a first-move table, all nodes $t \neq n$ reachable from n are assigned a *first-move label* (shorter: move label) that identifies the first move from n towards t.

[*]Part of this work was performed when the author was affiliated with NICTA and The Australian National University.

[1]In previous work, we did not use a separate name for the program, using "CPD" as a generic name for multiple components of this work, such as the program and a compressed database.

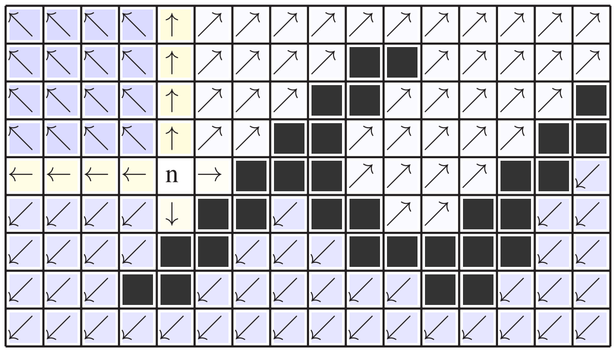

Figure 1: A first-move table on a toy grid map.

Figure 1 shows an example of a move table on a toy, 8-connected grid map, for a given origin point labelled n. Blocked cells are black. All traversable cells except for n have a move label represented in the picture with arrows with various orientations. For example, labelling the bottom-right cell of the grid with $\swarrow$ states that the first move from n towards that cell should be South-West. Notice in the picture how path coherence allows first-move data to cluster, offering an opportunity to compress the data.

The second step in an iteration in Algorithm 1 is decomposing a first-move table into disjoint *homogeneous rectangles*. A rectangle ρ is homogeneous if all nodes contained in ρ have the same move label. Homogeneous rectangles are allowed to contain blocked (or unreachable) areas. This helps obtaining a better compression, as there is no need to separate between such different types of areas.

First-move table decomposition is a recursive process. First, 4 or fewer rectangular sectors, depending on whether n is inside or on the edge of the map, are identified around n. There is one sector at the North-West (NW) of the origin, which in our example contains the upper left sector with 4 rows and 5 columns. The North-East (NE) sector has 4 rows and the 11 rightmost columns. The South-West (SW) sector has 5 rows and 5 columns, whereas the South-East (SE) one has 5 rows and 11 columns. Non-homogeneous sectors are split with a greedy procedure (Botea 2011) until all rectangles are homogeneous. For example, the NW sector is not homogeneous, as it has two types of move labels (i.e., $\nwarrow$ and $\uparrow$). In summary, the decomposition leads to up to 4 lists of rectangles associated with the origin node n. Each list, called a *sector list*, corresponds to one sector (e.g., NW).

In the example, the decomposition leads to 12 rectangles in total: 2 rectangles in the NW sector, one in the NE sector, 5 rectangles in the SE sector, and 3 in the SW sector.

Algorithm 2 Retrieving a path at runtime for an (s, t) pair.

```
$n \leftarrow s$ {initialize current node}
**while** $n \neq t$ **do**
    $e \leftarrow$ retrieveMove$(n, t)$
    $n \leftarrow$ node after move $e$
```

Algorithm 2 retrieves moves one by one in order. Method retrieveMove first identifies the sector c that contains the target by simply comparing the coordinates of n and t. Then, the corresponding list of rectangles $L_c(n)$ is parsed in order until the rectangle ρ containing t is found. The move label of ρ is precisely the move (i.e., edge) e to take from node n. In the worst-case, the entire list $L_c(n)$ has to be parsed to retrieve the move. The average case is much better, given that the rectangles in a list $L_c(n)$ are ordered decreasingly according to the number of contained reachable targets.

Finally, we illustrate list trimming, a feature that eliminates part of the rectangles in a sector list with no information loss. All rectangles removed from a given sector list have to have the same first-move label l. Assume a trimmed sector list is parsed at runtime, looking for the rectangle that contains the target (method retrieveMove, Alg. 2). If no such a rectangle is found, we know that the rectangle needed has been removed, and therefore it must have the move label l. Hence, l is the move to return by method retrieveMove.

List trimming's impact on move retrieval speed can be either positive or negative depending on the number of the removed rectangles, and their position in the sector list. It can be shown that, after selecting a move label l, an optimal trimming policy is to start removing rectangles from the end of the list, without skipping any rectangle with the move label l. If the expected move retrieval time does not increase, remove the rectangle and repeat the step. Otherwise, we have a memory vs speed trade-off, and whether to continue with the trimming or not depends on the user preferences. In our experiments, list trimming reduces the database size by a factor close to 2, with no negative impact on the speed.

References

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. In *AIIDE*, 9–14.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Dev.* 1:7–28.

Botea, A. 2011. Ultra-fast Optimal Pathfinding without Runtime Search. In *Proceedings of AIIDE-11*, 122–127.

Bulitko, V.; Björnsson, Y.; and Lawrence, R. 2010. Case-based subgoaling in real-time heuristic search for video game pathfinding. *J. Artif. Intell. Res. (JAIR)* 39:269–300.

Cazenave, T. 2006. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *CIG*, 27–33.

Harabor, D., and Botea, A. 2010. Breaking path symmetries in 4-connected grid maps. In *AIIDE-10*, 33–38.

Harabor, D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *AAAI-11*.

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

Sturtevant, N., and Buro, M. 2005. Partial Pathfinding Using Map Abstraction and Refinement. In *AAAI*, 47–52.

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *Proceedings of IJCAI*, 609–614.

Ziv, J., and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23(3):337–343.