Path Planning with Compressed All-Pairs Shortest Paths Data

Adi Botea*
IBM Research, Dublin, Ireland

Daniel Harabor
NICTA and The Australian National University

Abstract

All-pairs shortest paths (APSP) can eliminate the need to search in a graph, providing optimal moves very fast. A major challenge is storing pre-computed APSP data efficiently. Recently, compression has successfully been employed to scale the use of APSP data to roadmaps and gridmaps of realistic sizes. We develop new techniques that improve the compression power of state-of-the-art methods by up to a factor of 5. We demonstrate our ideas on game gridmpaps and the roadmap of Australia. Part of our ideas have been integrated in the Copa CPD system, one of the two best optimal participants in the grid-based path planning competition GPPC.

Introduction

Many planning problems can be modelled as shortest path computation in a graph. All-pairs shortest paths (APSP) data can lead to fast path planning in road networks (Wu et al. 2012) and video game grid maps (Botea 2011). They can speed up moving target search (Botea et al. 2013), and can be used to solve temporal problems in scheduling and planning with HTNs (Planken, de Weerdt, and van der Krogt 2011). Nearest neighbor problems have been addressed by extracting shortest paths from spatial databases (Papadias, Zhang, and Mamoulis 2003; Sankaranarayanan, Samet, and Alborzi 2009). Other, potential application domains include robot planning, multi-modal journey planning in a city, and multi-agent path planning.

Computing APSP data is challenging in terms of speed and efficient storage of the results. In this work we focus on storing APSP data efficiently. Without powerful APSP compression techniques, the usefulness of APSP data and, therefore, the usefulness of algorithms for APSP computation are *limited to small graphs*. Straightforward encodings, with one entry for every start–target pair, are impractical for all but small graphs, requiring an amount of memory within $O(n^2)$, or even $O(n^2 \log n)$, for a graph with n nodes.

The SILC framework (Sankaranarayanan, Alborzi, and Samet 2005) and Compressed Path Databases (CPDs) (Botea 2011) are two recent APSP compression techniques whose impressive storage capabilities scale far beyond naive storage methods. Both methods operate on *spatial networks* (graphs where each node is labelled with x and y coordinates) and both exploit a simple but powerful feature of such domains, coined as *path coherence* (Sankaranarayanan, Alborzi, and Samet 2005). 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. SILC and CPDs both cache the first move to take for every (s, t) pair. Compression involves, for each s, partitioning the map into a set of contiguous areas (blocks) A, such that all targets t in a block A have the same first-move label. Then, instead of storing many identical records for each (s, t) pair with $t \in A$, store a single first-move record (s, A). To obtain a shortest path a lookup is performed at s to find the first-move record for the block containing t. This repeats recursively until the target is reached.

We investigate techniques that can improve the compression power significantly. We use CPDs as a baseline, motivated by an earlier favourable comparison versus SILC, reported by Botea (2011). *List trimming*, one of our contributions, eliminates, in an information lossless fashion, part of the blocks resulted from the partitioning of a map. It reduces storage costs by a factor of about 1.8 with no associated running time penalty when answering shortest-path queries. Additional enhancements, based on the well known *run length encoding* and *sliding window compression*, reduce storage costs by a further factor of 2.9, at the price of a two-fold increase in shortest path query time, as detailed later. Benchmark data includes grid maps from the *Baldur's Gate* game, and a commercially used roadmap of Australia, with 1.7 million nodes. In the case of this latter graph our enhancements, combined together, reduce the size of the compressed path database from 16.48 GB down to 3.1 GB, making it fit into the RAM of many current computers. Part of the contributions, such as list trimming, are implemented in Copa CPD, one of the two best optimal competitors in the grid-based path planning competition GPPC.[1]

Background

Let $G = (V, E)$ be an input graph for which to compute and compress APSPs. The x and y coordinates of nodes are

*This work was performed in part when the first author was affiliated with NICTA and The Australian National University.

[1] `http://movingai.com/GPPC/`

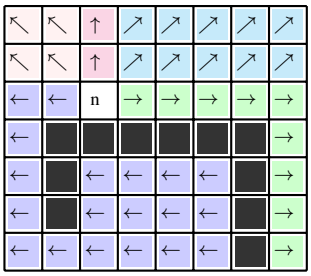

Figure 1: A first-move table on a toy grid map.

assumed to be available. The *column* (*row*) of a node n is the index of its x (y) coordinate in the ordered set of the unique x (y) values of all nodes in V.

Algorithm 1 Independent iterations in building a CPD.

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

A CPD (Botea 2011) is an array of lists of rectangles $L(n)$, with one list for each node $n \in V$. Building a CPD requires a series of independent iterations, one for each node n, as illustrated in Algorithm 1. 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 $t \neq n$ are assigned a *first-move label* (i.e., the first move from n towards t).[2]

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 $\rightarrow$ states that the first move from n towards that cell should be to the East. 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 a list of *homogeneous rectangles* $L(n)$. A rectangle ρ is homogeneous if all nodes contained in ρ have the same move label. The decomposition technique can be found in the original paper (Botea 2011).

A very simple path retrieval procedure that retrieves moves one by one in order. A method retrieveMove(n, t) involves parsing the list $L(n)$, in order, until the rectangle ρ containing t is found. The move label of ρ is precisely the move (i.e., edge) to take from node n. In the worst-case, the entire list $L(n)$ has to be parsed to retrieve the move. The average case is much better, given that the rectangles in a list $L(n)$ are ordered decreasingly according to the number of contained reachable targets. Botea (2011) has pointed out that, given a uniformly randomly selected target, the average number of rectangles to check in a list $L(n)$ of size $|L(n)| = l$, until the target is found, is $N_0 = \sum_{i=1}^{l} iw_i$. The first position in the list is indexed with 1. A value w_i is the number of contained locations of the i-th rectangle in $L(n)$, normalized such that $\sum_i w_i = 1$.

[2]For brevity, assume that all nodes are reachable from n.

Improving the compression power

In preparation for the size-reduction operations discussed in this section, each list of rectangles $L(n)$ is split into four smaller lists, called *sector lists*. For example, the NW sector list contains all rectangles placed in the North-East area of the map relative to n. The SW, SE and NW sector lists are defined similarly.[3] Sector lists allow improving the speed, as lists of rectangles to traverse at runtime become shorter.

List trimming with default moves

We present *list trimming*, a technique to eliminate rectangles from a sector list without any information loss. Assume there is a function ξ that provides a *default move* $\xi(n, t)$ for every current node n and target t. While ξ will be used only when there is a guarantee that it provides optimal and correct moves, it is not necessary that ξ returns correct answers for *all* pairs (n, t). It is desired, however, that a default move $\xi(n, t)$ is computed quickly (e.g., in constant time) and that it uses no or very little cached data.

A given rectangle ρ can be removed from a (sector) list $L(n)$ when ml(ρ), the move label of ρ, coincides with the default moves obtained with ξ: $\forall t' \in \rho, \xi(n, t') = \text{ml}(\rho)$. Consider a list $L(n)$ after removing part of its rectangles. Retrieving the move to take at node n involves parsing the list $L(n)$. If the parsing completes without finding the *target rectangle* (i.e., the rectangle containing the target t), it must be the case that the target rectangle has been removed from $L(n)$. Therefore the move label of the target rectangle is exactly $\xi(n, t)$, since this was a necessary condition to remove a rectangle. In such cases, ξ is guaranteed to provide a correct and optimal move.

A simple definition for ξ turned out to be quite effective. For each sector list, the default move is the move label associated with the largest number of rectangles in that list. Some, but not necessarily all default-move rectangles can be removed from the sector list, as discussed below.

Definition 1 *A* trimming τ *of a sector list* L *is a set of positive integers* $\{i_1, i_2, \ldots i_k\}$. *These numbers represent the original positions in* L *corresponding to the removed rectangles. All removed rectangles have the same move label.*

The impact of list trimming on the speed of retrieving a move can be either positive or negative, depending on factors such as the number, the size, and the position in the list of the removed rectangles. The N_0 formula, mentioned in the background section, quantifies the average number of rectangles to check in a list $L(n)$ in order to find the rectangle containing the target. N_0 is restricted to cases where the target does belong to one of the rectangles in $L(n)$, which is not necessarily true for trimmed lists. Hence we derive a new formula, $N(\tau)$, which covers the case of trimmed lists as well, to be able to compare trimmings in terms of speed,

[3]An artifact of the decomposition procedure (Botea 2011) is that no rectangle goes across two or more sectors.

in addition to their size reduction. As the default move function in use is very fast, consisting of a table look-up, its time cost is left out in the rest of this analysis.

Let $\tau = \{i_1, i_2, \dots i_k\}$ be an arbitrary trimming. When the target belongs to a preserved rectangle ρ, the number of rectangles to check decreases by a non-negative amount a, where $a \geq 0$ is the number of rectangles removed from the part of the list ahead of ρ. When the target belongs to a removed rectangle, the entire list, which now has $l - k$ elements, is parsed. Overall, the average number of rectangles to check is $N(\tau) = \sum_{j=0}^{k} \sum_{i=i_j+1}^{i_{j+1}-1} (i-j)w_i + (l-k)\sum_{j=1}^{k} w_{i_j}$. The first term corresponds to preserved rectangles, whereas the second term accounts for the removed ones. In this formula, $i_0 = 0$ and $i_{k+1} = l + 1$.

In the rest of this section we formally present a strategy for identifying a trimming of a given list. We introduce a non-dominance relation between trimmings and show that our resulting trimmings are non-dominated.

Definition 2 *Let τ_1 and τ_2 be trimmings of a list $L(n)$. We say that τ_1 is non-dominated by τ_2 if $N(\tau_1) \leq N(\tau_2)$ and τ_1 removes at least as many rectangles as τ_2.*

Lemma 1 *Let $\tau_1 = \{i_1, \dots i_p, \dots i_k\}$ and $\tau_2 = \{i_1, \dots j_p, \dots i_k\}$ be two trimmings that differ only on the position of p-th removed rectangle. Assume that $i_{p-1} < i_p < j_p < i_{p+1}$. Then τ_2 is non-dominated by τ_1.*

Proof: For simplicity, let us rename i_p as a and j_p as b. The only differences between $N(\tau_1)$ and $N(\tau_2)$ will occur for the terms containing $w_a, w_{a+1}, \dots, w_b$. The contribution of these terms to $N(\tau_1)$ will be $F_1 = (l-k)w_a + (a+1-p)w_{a+1} + \dots + (b-p)w_b$. For τ_2, we obtain: $F_2 = (a-p+1)w_a + \dots + (b-p)w_{b-1} + (l-k)w_b$. The difference $F_1 - F_2$ is $(l-k-a+p-1)w_a - w_{a+1} - \dots - w_{b-1} - (l-k-b+p)w_b$. Notice that, $\forall c > a$, the factor in the front of w_c is negative. Furthermore, $\forall c > a$, we have $w_a \geq w_c$, according to the ordering inside a list of rectangles. Based on these two facts, by replacing all w terms with w_a in the previous formula of $F_1 - F_2$, we obtain that $F_1 - F_2 \geq w_a(l-k-a+p-1-b+1+a-l+k+b-p) = 0$. It follows that $F_1 \geq F_2$ and hence $N(\tau_1) \geq N(\tau_2)$.

Definition 3 *A* gap *in a trimming $\tau = \{i_1, \dots i_k\}$ is a position $v > i_1$, not included in the trimming, with the property that the rectangle on position v has the same move label as the rectangles in the trimming.*

Theorem 1 *For any trimming τ with gaps, there is a trimming without gaps that is non-dominated by τ.*

Proof: Let $\tau = \{i_1, \dots i_k\}$ be a trimming with a gap v. Let i_p be the biggest position in the trimming such that $i_p < v$. Cf. Lemma 1, replacing i_p with v in the trimming leads to a new trimming that is non-dominated by τ. This change "shifts" the gap towards the beginning of the list. We repeat the process until all gaps "travelled" outside the range of indexes included in the trimming, obtaining a trimming without gaps. As the non-dominance relation is transitive, the resulting trimming is non-dominated by the original one.

Theorem 1 provides a computationally-easy strategy for building a non-dominated trimming. Starting from the end of the list, rectangles with the selected move label are added to the trimming (i.e., marked for deletion) one by one. At each step, the new trimming τ' is compared to the previous trimming τ. If τ' is non-dominated by τ, continue. Otherwise, τ' is better in terms of memory and τ is better in terms of the expected move retrieval time. Whether to continue or not depends on the user preference. For example, one can define $R(\tau) = \frac{N(\tau)}{N_0}$ and set a maximal threshold C for $R(\tau)$, thus growing the trimming τ as long as $R(\tau) \leq C$.

Compression across multiple first-move tables

Part of the ideas discussed in this section have the capability of compressing data across first-move tables, not only inside individual tables. The data (i.e., trimmed lists of rectangles) are partitioned into five subsets called *strings*. Strings are further compressed independently from each other, using two generic, widely used compression methods, slightly adaptated to suit our problem better.

The five strings separate the data according to the five fields (two rows, two columns and one move label) that compose a rectangle record: there is one string containing the left column value for every rectangle, and so on. A string contains, in order, the values of the field at hand for every rectangle record in the database. These values are seen as atomic symbols (also known as literals) in the string, rather than considering finer atoms such as digits and blanks. For each string, there is an indexing that provides constant-time access to the head of each sector list.

The first compression idea applied here is a simple variation of the well-known *run-length encoding* (RLE). A sequence containing $x > 1$ repetitions of a literal S is replaced with an encoding Sx. One bit per token is used to distinguish between actual symbols and counters. The second idea, called *sliding-window compression* (SWC), is equally simple and popular. When a substring occurs repeatedly, subsequent occurrences are replaced with a "pointer" to an earlier occurrence (Ziv and Lempel 1977). The "pointer", called a *length-distance pair*, is composed of two numbers, containing the length of the substring, and (the distance to) the starting position of an earlier occurrence of the substring. We call such a starting position to point to a *jump point*. We call a *cut* an occurrence of a substring that has been replaced with a length-distance pair.

In our problem, fast decompression is critical, as it has to be performed in real time during the retrieval of paths from the database. Two adaptations ensure that the overhead of decompression in fetching one symbol from a string is bounded by a constant factor.

Firstly, cuts that contain the head of a sector list are never considered. This guarantees that sector list heads are reached in constant time, using the available indexing. Secondly, when fetching the next symbol (e.g., the left column of the next rectangle) from a compressed string, there is at most one jump to an earlier position in the string. Unless such a precaution is taken, the number of back jumps to fetch one symbol could be up to linear in the size of the string. Consider two occurrences eo and lo of a substring $S_1 \dots S_n$, such that eo occurs earlier than lo. To allow re-

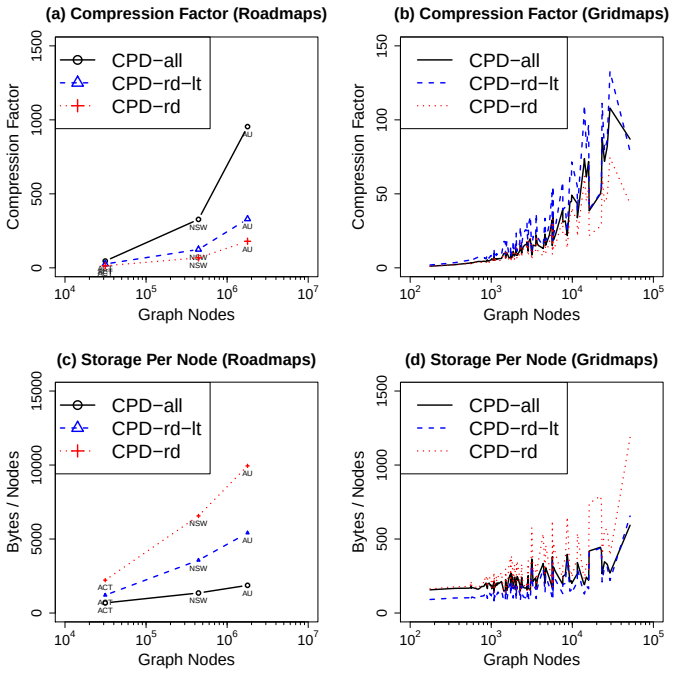

Figure 2: Top: compression. Bottom: storage per node.

placing lo with a pointer to eo, we require that the symbol at the jump point (i.e., the first position of eo) is preserved in the compressed string, rather than being cut away. This is sufficient to ensure that S_1 will be retrieved with exactly one jump. Other parts of eo can belong to cuts, giving more freedom to SWC to potentially achieve a better compression. Retrieving *all* symbols in lo could require a stack of jumps but nevertheless there is at most one jump per symbol.

In summary, parsing a (trimmed) sector list after applying RLE and SWC translates into parsing the 5 strings starting from the positions of the sector list head. Symbols are retrieved one by one from each substring, until the target rectangle is found or the list's end is reached, in which case the default move for that trimmed sector list is returned.

Experimental Results

We evaluate the impact of our new ideas on the memory and speed performance of the system. A comparison to other pathfinding is available on the competition website. Our data include 120 8-connected gridmaps (Baldur's Gate set), with sizes up to 320x320 (51,000 nodes). We also use a complete road map of Australia (AU), with 1.7 million nodes, and two subsets. The New South Wales (NSW) subset has 443,000 nodes. ACT (Australian Capital Territory) has 31,000 nodes. Our code is written in C++, and tested on a 2.6GHz Intel Core2Duo machine with 8GB RAM and Mac OSX 10.6.5.

Figure 2 shows compression data, as a multiplicative factor compared to uncompressed first-move tables. CPD-rd performs rectangle decomposition (Botea 2011). CPD-rd-lt adds list trimming (i.e., default move filtering) on top of CPD-rd. CPD-all includes RLE and SWC, on top of CPD-rd-lt. All CPD variants include sector splitting, not present

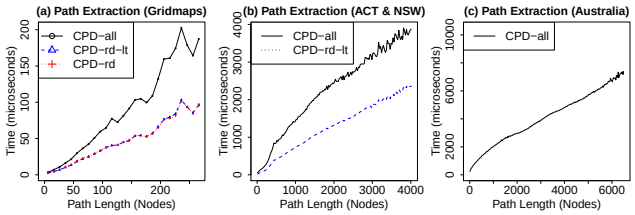

Figure 3: Time in microseconds to find shortest paths.

in previous work (Botea 2011). Figure 2 (a) shows the compression factor on road maps. The higher the compression factor, the better the compression. CPD-rd-lt is almost twice as good as CPD-rd in all cases. CPD-all improves this further on road maps. Savings grow with the map size. For the full AU map, CPD-all is 5.3x better than CPD-rd, and 950 times better than uncompressed APSP data.

Figure 2 (b) focuses on grid maps. CPD-all is generally better than CPD-rd, but weaker than CPD-rd-lt on most grid maps in our tests. The explanation is that, on the tested grid maps, CPD-rd-lt already has a good compression performance. SWC and RLE have an overhead added by the indexing that allows reaching list heads in constant time. This overhead is relatively small, being linear in the number of graph nodes. However, when the database is relatively small, the overhead becomes visible. This, plus the constant-factor slow down pointed out below, explain why RLE and SWE were not activated in the competition program.

Figures 2 (c) and (d) summarize the storage per node (SpN). As maps grow in size, the SpN increases quite slowly (note the x-axis logarithmic scale), an effect also observed in SILC (Sankaranarayanan, Alborzi, and Samet 2005). Of course, the total memory usage gets multiplied by the number of nodes, typically leading to a super-linear increase of the CPD in the size of the map.

Figure 3 presents time performance data for shortest path queries. List trimming has no negative impact on the speed, as shown in Figure 3 (a) (two curves virtually identical) and also confirmed in tests for roadmaps where CPD-rd fits in memory. CPD-all is slower by a factor close to 2. This overhead, introduced by performing jumps in SWC, is guaranteed to be bounded by a constant factor. For the whole map of Australia, CPD-all (3.1 GB in size) is the only database that fits in memory. The size of AU CPD-rd is 16.48 GB.

Conclusion

We discussed compression techniques applicable to graphs where nodes are annotated with x and y coordinates. Using a combination of rectangle list trimming with default moves, run-length encoding and sliding-window compression, we significantly improve the performance of CPD (Botea 2011), a state-of-the-art APSP compression technique. Part of our contributions, such as list trimming, are used in Copa CPD, a top competitor in the path planning competition GPPC. We have formally proved the optimality of our list trimming policy. Future work includes combining CPDs with hierarchical map decomposition for an additional compression power. We plan to apply CPDs to multi-agent path planning.

References

Botea, A.; Baier, J.; Harabor, D.; and Hernández, C. 2013. Moving target search with compressed path databases. In *Proceedings of ICAPS-13*.

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

Papadias, D.; Zhang, J.; and Mamoulis, N. 2003. Query processing in spatial network databases. In *In VLDB*, 802–813.

Planken, L.; de Weerdt, M.; and van der Krogt, R. 2011. Computing all-pairs shortest paths by leveraging low treewidth. In *Proceedings of ICAPS-11*, 170–177.

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

Sankaranarayanan, J.; Samet, H.; and Alborzi, H. 2009. Path oracles for spatial networks. In *In VLDB*, 1210–1221.

Wu, L.; Xiao, X.; Deng, D.; Cong, G.; Zhu, A. D.; and Zhou, S. 2012. Shortest path and distance queries on road networks: An experimental evaluation. In *In VLDB*, 406–417.

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