Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment

Ultra-Fast Optimal Pathfinding without Runtime Search

Adi Botea
NICTA* and the Australian National University

Abstract

Pathfinding is important in many applications, including games, robotics and GPS itinerary planning. In games, most pathfinding methods rely on runtime search. Despite numerous enhancements introduced in recent years, runtime search has the disadvantage that, in bad cases, most parts of a map need to be explored, causing a time performance degradation. In this work we explore a significantly different approach to pathfinding, eliminating the need for runtime search. Optimal paths between all pairs of locations are pre-computed. Since straightforward ways to store pre-computed paths are prohibitively expensive even for maps of moderate size, pre-computed data are compressed, reducing the memory requirements dramatically. At runtime, pathfinding is very fast, as it requires visiting only the locations on an optimal path. In each location, a quick computation provides the next move along the optimal path. We demonstrate the effectiveness of this approach on Baldur's Gate game maps. The compression factor reaches two orders of magnitude, bringing the memory requirements down to reasonable values. Compared to A* search, the runtime speedup reaches and even exceeds two orders of magnitude. When averaged over paths of similar cost, the speedup reaches a value of 700 in our experiments.

Introduction

Pathfinding has many applications, including games, robotics and GPS itinerary planning. Runtime search is used in most pathfinding techniques. One popular method to obtain a search space is to discretize an initial map into a grid map. Then, a search algorithm, such as A*, enhanced with an admissible heuristic function, such as the Manhattan or the Octile heuristic, can be used for pathfinding.

Numerous enhancements to pathfinding (e.g., on grid maps) have been introduced in recent years. These include reducing the size of the search space with hierarchical abstraction (Botea, Müller, and Schaeffer 2004; Sturtevant and Buro 2005), exploiting symmetry (Harabor and Botea 2010; Harabor and Grastien 2011), and building accurate heuristics at the cost of using extra-memory (Björnsson and Halldórsson 2006; Cazenave 2006; Sturtevant et al. 2009). Despite the success of such enhancements, it is often the case that solving a pathfinding instance requires visiting many locations on a map, resulting in a speed degradation.

We introduce *compressed path databases* (CPDs), a pathfinding approach significantly different from traditional approaches based on runtime search. Our goal is to improve the speed dramatically using pre-processing and additional memory to efficiently store the pre-processing results. While many games feature dynamic environments, and mobile units with different sizes and terrain traversal capabilities, we focus on static environments and single-agent pathfinding. We discuss briefly the case of dynamic obstacles (other mobile units or changes in environment) as well.

During pre-processing, optimal paths between any two locations on a map are computed. If pre-computed data are stored in a naive way, the memory requirements are quadratic in the size of the original map, being prohibitively large even for maps of moderate size. To address this, we introduce a technique for compressing path information, reducing memory requirements dramatically.

With CPDs in use, runtime pathfinding is very fast. It only requires visiting each location along an optimal path from the start to the target location. In each location, a quick computation provides the next location where to move along an optimal path. The first-move lag, which measures the time needed before making the first move on a path, is extremely small, since deciding the next move is independent of deciding the rest of the moves. Computing the next move is faster even than in real-time search, as the latter relies on a small local search to compute a move.

Similarly to Sankaranarayanan, Alborzi, and Samet (2005), our work exploits the idea that, often, the shortest paths from a current node to any target in a remote area share a common first move. These authors call that property *path coherence*. There are significant differences between our work and the approach taken by Sankaranarayanan, Alborzi, and Samet (2005). First off, Sankaranarayanan, Alborzi, and Samet compress a map using a quad-tree decomposition. Our method decomposes a map into a list of rectangles that can have arbitrary sizes and placements, which can potentially require significantly fewer rectangular blocks to cover the map. We will show that, in our experiments, CPDs are significantly more

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

memory efficient than quad-tree decompositions. Secondly, Sankaranarayanan, Alborzi, and Samet (2005) applied their ideas to road maps. We use game maps as a test bed.

We run experiments on 120 maps from the Baldur's Gate video game, with sizes of up to 320x320 tiles. When the speed-up is averaged over paths of similar length, the largest values are within two orders of magnitude, being up to 700 for our test data. To our knowledge, this is by more than one order of magnitude higher than the best speedups seen in the literature on the same, frequently used dataset (e.g., (Harabor and Grastien 2011; Botea, Müller, and Schaeffer 2004; Sturtevant and Buro 2005)).

The compression factor achieved with CPDs reaches two orders of magnitude. This means that, on the larger maps, our compressed path databases require less than 1% of the memory required by an uncompressed path database.

Our results on static grid maps demonstrate that a reasonable amount of extra-memory can eliminate the need for runtime search in pathfinding. This ensures that all pathfinding queries are answered very fast, visiting only locations along an optimal path, whereas methods based on runtime search can potentially explore the whole map in bad cases. The total pre-processing time is about 40 minutes on the largest test map and less than 8 minutes on all other maps. Since preprocessing is composed of many independent iterations, the pre-processing time can trivially be reduced further by running different iterations on different CPUs, with a speed-up that is linear in the number of processors available.

Related Work

The closest related work is due to Sankaranarayanan, Alborzi, and Samet (2005). A brief comparison has been provided in the introduction. An empirical comparison between CPDs and quad-trees in terms of memory efficiency is provided in the experiments section.

Many of the existing memory-based admissible heuristics designed for pathfinding on grid maps use a pre-computation step which has similarities with our approach. A major difference is in the way that memory bottlenecks are avoided. In CPDs, we pre-compute full path information (i.e., enough information to quickly retrieve optimal paths between any two locations) and then compress the data to make it fit into the memory. On the other hand, the information provided by memory-based heuristics is more limited, having to be combined with runtime search in order to compute a path.

Using one or several *landmarks* is a key idea in computing memory-based heuristics. In pathfinding, a landmark l is a location with the property that distances $d(l, a)$ from the landmark to all other locations a on the map are available. Storing all distances $d(l, a)$ for a fixed landmark l requires a table of the size of the (traversable area of the) map. Given a landmark l and two arbitrary locations a and b, $h_L(a, b) = |d(l, a) - d(l, b)|$ is an admissible heuristic for the true distance $d(a, b)$ (Goldberg and Harrelson 2005).

Several extensions of the h_L heuristic have been introduced in recent years. For example, the so-called ALTBest heuristic allows defining more than one landmark (Cazenave 2006). ALTBest is equal to the maximum of the heuristic estimations provided by each landmark. Relatively few landmarks could typically be used, as each landmark requires to store a table of the size of the map. Other related techniques, such as the gateway heuristic (Björnsson and Halldórsson 2006), the canonical heuristic (Sturtevant et al. 2009), and the portal-based heuristic (Goldenberg et al. 2010) rely on storing perfect distance information for a subset of selected pairs of locations on a map.

Kring, Champandard, and Samarin (2010) cache full path information for small parts of a grid map, such as 10x10 areas called clusters. This allows to traverse a cluster between predefined entrance points without search, but it does not avoid the need for search at a global, map-level scale. Our new compression technique allows us to scale the caching of path information to full maps.

In pattern databases (PDBs) (Culberson and Schaeffer 1998), an original, very large search space is abstracted into a smaller space that fits into the main memory and that can be searched exhaustively. Distances in the abstract space are cached and used as an estimator of distances in the original state space. Differences between a PDB and a CPD can be summarized as follows. PDBs are designed for very large search spaces, which can't be searched exhaustively. Typically pathfinding does not belong to this category. PDBs are usually built for a fixed goal or set of goals and store imperfect distance information. CPDs store perfect information about optimal paths between any two nodes. They work on search spaces that can be searched exhaustively during preprocessing, such as pathfinding maps.

In two-player board games, move tables such as opening books and end-game databases are common enhancements to game playing programs.

Introducing Compressed Path Databases

Similarly to Sankaranarayanan, Alborzi, and Samet (2005), our work exploits a feature, called by these authors *path coherence*, that many search maps exhibit.

Consider we are in Montreal, Canada and want to travel on an optimal route to San Diego, California, intersection Broadway – 10th Avenue. Assume we have a pathfinding system that tells, in each current location (such as a street intersection), what move to make next (e.g., what intersection to reach next). When we are in a new current location, the pathfinding system once again will tell what intersection to reach next along an optimal path to the destination.

Pre-computing and storing all this information about what move to make next would normally require huge memory resources, as we would need one first-move record for each pair of locations (e.g., street intersections) on the map. Fortunately, path coherence offers a massive potential for compressing the information. Consider again the previous example. It is very likely that the first move along an optimal path from the current location in Montreal towards *any* target location in San Diego is *the same*. Take things to a larger scale: it could actually be the case that the first move on an optimal path towards any destination in the USA is the same.

The example just presented suggests the following data compression strategy, which we implement in compressed

path databases. When the first move on an optimal path from a current location n to any target t in a contiguous rectangular area R is the same, we store only one move record, corresponding to (n, R), instead of storing the same move record many times, once for each pair (n, t). In this way, for each current location n, the map is decomposed into a set of rectangles. Each rectangle R has the property that the first move from n on an optimal path to any target location inside R is the same (we call these *homogeneous rectangles*). The fewer such rectangles, the better the compression.

The following section describes our method in detail. The discussion and the experiments will use grid maps, which are very relevant to the games community, even though the method is more generally applicable (e.g., to road maps).

Computing CPDs as Preprocessing

Algorithm 1 outlines the procedure that computes compressed path databases. It iterates through all traversable tiles n on the map. Each iteration has two steps. First, compute a bi-dimensional move table $T(n)$, having the size of the map. Each entry $T(n)[c, r]$ contains the first move on an optimal path from tile n to the tile whose column–row coordinates are (c, r).[1] We call this the *move label* (or the *color*) of tile (c, r). Computing $T(n)$ is performed with a slightly modified Dijkstra algorithm. A standard Dijkstra implementation outputs the distance from the origin to any target. Our implementation provides the first move on an optimal path from the origin to any target.

Figure 1 shows an example of a move table on a toy map. The white cell is the current location n. Black cells represent obstacles. The label in a cell l is the first move on an optimal path from n to l. For clarity, each label has its own color.

There is an important detail about breaking ties in a move table. When two distinct moves are optimal in a given location, we prefer the one that favours obtaining straighter borders between clusters. Straight borders allow building fewer rectangles in the decomposition step. For example, in Figure 1, consider l, the NW location above the leftmost We location. Location l could be labelled either NW or We. The reason is that there are several optimal paths from the origin to l, and some of the paths start with an NW move, whereas others start with a We move. The NW label is preferred to build a straighter cluster border.

The second step in an iteration compresses the move table $T(n)$ into a list $L(n)$ of homogeneous rectangles ordered decreasingly according to their traversable area. We say that a rectangle is homogeneous if all traversable tiles have the same move label. A homogeneous rectangle may contain blocked tiles. For each homogeneous rectangle in a list we store both its coordinates (i.e., upper-left tile and bottom-right tile) and the unique move label of its traversable tiles.

The rectangles in $L(n)$ are disjoint and cover the whole map[2] except for the origin tile n.

[1]Entries $T(n)[c, r]$ corresponding to a blocked tile (c, r) are never needed, so such entries need no initialization.

[2]However, homogeneous rectangles with only blocked tiles (if any) are never needed and thus they can be dropped from $L(n)$.

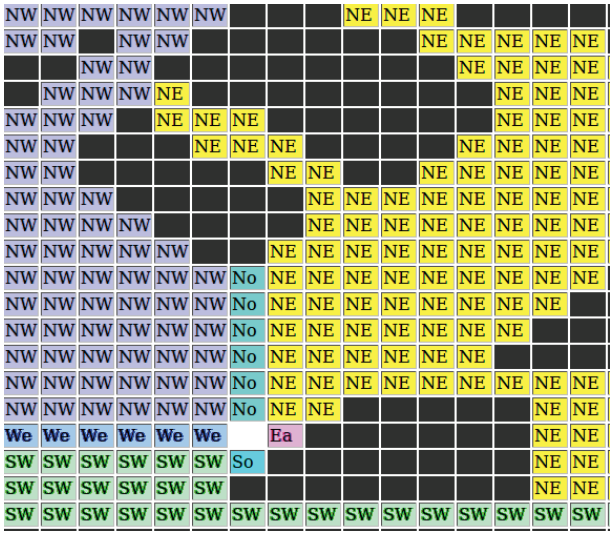

Figure 1: A move table on a toy map.

In the following sections, the coordinates of a rectangle ρ are represented as (c_1, r_1, c_2, r_2), where c_1 is the first column that has common tiles with ρ, c_2 is the last column that has common tiles with ρ, r_1 is the first row that has common tiles with ρ, r_2 is the last row that has common tiles with ρ.

Algorithm 1 Building a compressed path database

for each traversable tile n **do**
  $T(n) \leftarrow$ Dijkstra(n)
  $L(n) \leftarrow$ Compress$(T(n))$

The compression of a move table $T(n)$ is a recursive process. First, depending on the placement of the origin tile n on the map (corner, edge, or interior tile), eight or fewer rectangles are identified around n. In the most general case of an interior tile n, there are one rectangle at the northwest of n: $(0, 0, \text{col}(n) - 1, \text{row}(n) - 1)$; one narrow, width-one rectangle at the north: $(\text{col}(n), 0, \text{col}(n), \text{row}(n) - 1)$; one rectangle at the northeast of the origin: $(\text{col}(n) + 1, 0, w - 1, \text{row}(n) - 1)$ (where w is the map width); one narrow, height-one rectangle at the east: $(\text{col}(n) + 1, \text{row}(n), w - 1, \text{row}(n))$; and so on.

The example shown in Figure 1 features three homogeneous rectangles after the first decomposition step: the one at the north east, the one at the south west, and the one at the west of the origin. All others rectangles are decomposed further as follows.

Each non-homogeneous rectangle is decomposed further recursively, with one or more *decomposition steps*, until all rectangles are homogeneous. A decomposition step splits a given rectangle ρ into two smaller rectangles ρ' and ρ''. To determine where to split ρ, the algorithm considers four options first: 1) split ρ vertically such that the rectangle at the left is homogeneous, has width at least 1, and is as wide as possible; 2) split ρ vertically such that the rectangle at the right is homogeneous, has width at least 1, and is as wide

as possible; 3) split ρ horizontally such that the rectangle at the top is homogeneous, has height at least 1, and is as high as possible; 4) split ρ horizontally such that the rectangle at the bottom is homogeneous, has height at least 1, and is as high as possible. If at least one of the four options is possible, choose the one that provides the homogeneous rectangle with the largest traversable area. Notice that, in such a case, at least one of the two obtained rectangles, ρ' and ρ'', is homogeneous and therefore requires no further decomposition.

On the other hand, if none of the four options is possible, split ρ in two parts. If width(ρ) $>$ height(ρ), split it with a vertical line as evenly as possible. (If the width is odd, we assign one extra column to the rectangle at right.) Otherwise, split ρ with a horizontal line as evenly as possible.

The procedure to compress one move table outlined above has a $O(N \log N)$ time complexity, where N is the number of tiles on the map. Therefore, the time complexity of building a compressed path database for a map is $O(N^2 \log N)$.

If desired, the preprocessing time can be reduced by performing different iterations of Algorithm 1 on different processors. Such iterations are independent from each other and hence it is trivial to parallelize the process, obtaining a speed-up that is linear in the number of available processors.

Runtime Pathfinding

Algorithm 2 Runtime pathfinding

```
z ← s {initialize current location}
while z ≠ t do
    m ← getMove(L(c), t) {retrieve next move info}
    z ← γ(z, m) {simulate move to update z}
    π ← π ⊕ m {append move to solution}
return π
```

Retrieving a path at runtime is a straightforward procedure. As all path information is pre-computed, no runtime search is needed. Instead, runtime pathfinding is a simple and fast process that retrieves path information and returns an optimal path as a sequence of moves.

Algorithm 2 illustrates the runtime process. The current location z is initialized to the starting position s. At each iteration, z moves to the next location along the optimal path, until the target location t is reached. The move to take next is retrieved as a result of a computation that iterates through the rectangles of the list $L(z)$ until the rectangle ρ that contains the target location t is found. The move to take at location z is precisely the move label associated with ρ. Checking whether a given location t belongs to a given rectangle ρ is called a *rectangle check*. It is a constant-time operation that compares the coordinates of t with the coordinates of ρ.

As said earlier, ordering the list $L(z)$ decreasingly, according to the traversable area of each rectangle, reduces the number of rectangle checks. Let w_i be the area of rectangle ρ_i, normalized such that $\sum_{i=1}^{l} w_i = 1$, where $l = |L(z)|$.

Proposition 1 *Given a list of homogeneous rectangles $L(z)$, and a location t that is chosen uniformly randomly, the average number of rectangle checks necessary to detect the rectangle that contains t is $\sum_{i=1}^{l} i w_i$, where $l = |L(z)|$.*

Proof: It is easy to see that the number of rectangle checks can be modelled as a discrete random variable v that takes the values 1, 2, ..., l with the probabilities $w_1, w_2, ..., w_l$. The mean value of v is precisely $\sum_{i=1}^{l} i w_i$.

CPDs and Dynamic Environments

Even though this paper focuses on static environments, we comment briefly on using CPDs in dynamically changing environments. Here we present a strategy that combines CPD-based pathfinding with online search. The latter is used only when CPD paths run into (recently appeared) obstacles, aiming at keeping runtime search effort low. Let $l_0, l_1, \ldots, l_m$ be a CPD-provided path. Assume that the unit is at location l_i and that location l_b, with $b > i$, becomes blocked, invalidating the path. We launch an online search, starting from l_i, and stop as soon as we reach any location l_j, $j > b$. Repair the path by replacing the original $l_i \ldots l_j$ segment with the new one.

Small variations of this strategy might prove worthwhile. For example, instead of starting searching from location l_i, advance to location l_{b-1} and re-plan from there. Clearly, if location l_b becomes unblocked quickly, there is no need to repair the original path. In the case of more permanent obstacles (e.g., a bomb destroying a bridge), incrementally repairing a CPD is a promising idea. This is beyond the focus of this paper, being left as future work.

Experimental Results

We have run experiments on maps obtained from the Baldur's Gate game. This frequently used benchmark contains 120 maps varying in size from 50x50 to 320x320.

After building a compressed path database for each map, the performance is analyzed in terms of preprocessing time, memory requirements, and the runtime speedup over conventional A*. We generated for each map 100 instances with the start and the target selected randomly in such a way that a path exists between them. In each instance, we compare CPD pathfinding and A* pathfinding in terms of time and nodes. The environment is assumed to be static.

Our program is written in C++ from scratch. For comparison, we used the A* implementation available in the open-source HOG library (http://webdocs.cs.ualberta.ca/~nathanst/hog.html). All tests have been run on a Linux system with a 3.2GHz processor.

Figure 2 illustrates how the average speedup of CPDs over A* search evolves as path cost increases. The speedup reaches two orders of magnitude even for relatively short paths, having a cost of 100. It increases steadily as the path cost increases, reaching a maximum average value of 700. Other speedups over A* previously reported in the pathfinding literature are much smaller, as mentioned earlier.

The explanation for CPD's very large speedup can be summarized as follows. Firstly, there is the difference in the number of processed nodes, which we present in Figure 3. CPDs require visiting only the nodes along an optimal path. In contrast, in search-based methods the number

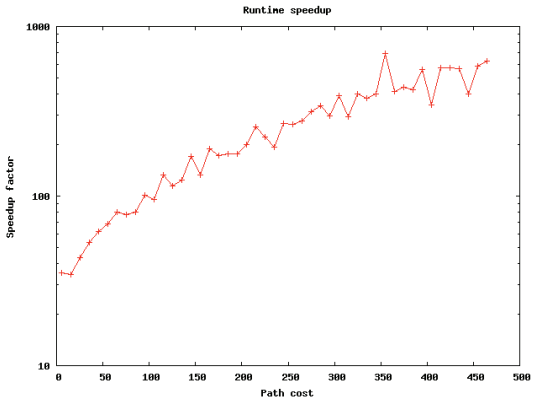

Figure 2: Average speedup of CPD over A* search

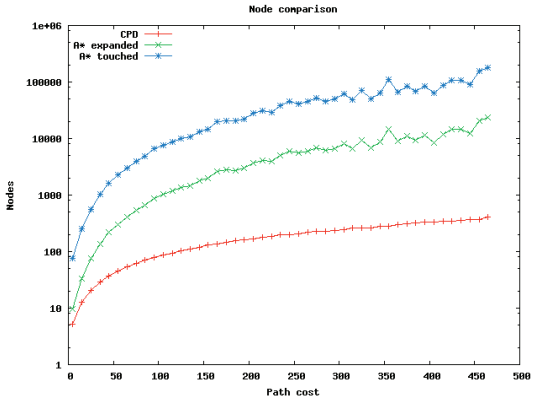

Figure 3: Nodes

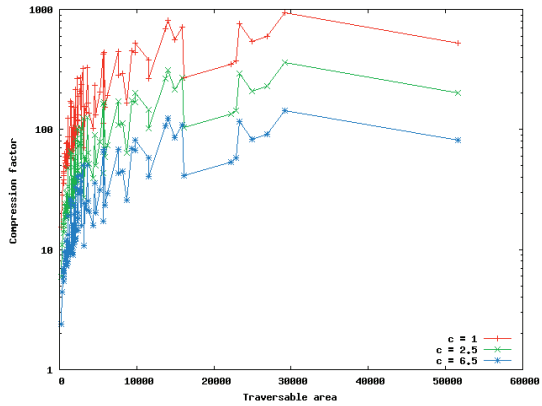

Figure 4: Compression factor

of nodes along a path is just a lower bound for the number of searched nodes. As shown in Figure 3, the number of expanded nodes in A* can be two orders of magnitude larger than the number of nodes visited in CPD. The number of visited nodes is even larger. Secondly, the cost per node is lower in CPDs. The only processing needed per node is iterating through a list of rectangles until a rectangle that contains the a given tile (the target) is found. The largest average number of rectangle checks per node is 6.2 (computed as in Proposition 1) and corresponds to the largest map, which has more than 50,000 traversable tiles. As mentioned earlier, ordering the rectangles in the list decreasingly according to their traversable area minimizes the number of average rectangles checks per node. On the other hand, expanding a node in A* requires generating up to 8 successors and performing operations on a priority list (the Open list) whose complexity is often (depending on the implementation (Cazenave 2006)) logarithmic on the size of the list.

Next we analyze the memory requirements and the compression power of CPDs by comparing CPDs with uncompressed path databases (UPDs). A UPD is a collection of move tables before these are compressed into lists of rectangles. UPDs can provide the next move in constant time, but their memory requirements are prohibitive for all but very small maps or submaps. Figure 4 plots the compression factor, computed as $\frac{1}{c} \times \frac{A^2}{R}$, which is the memory required by a UPD divided by the memory required by a CPD. A is the number of traversable tiles. A UPD stores A^2 records, one for each pair of traversable locations (m, n). Each UPD record contains two moves: the first move from m to n and the first move in the other direction. R is the number of rectangles contained in a CPD and c is a constant that reflects the size difference between a UPD record and a CPD record. In a CPD, one rectangle record stores the coordinates of the upper-left and the bottom-right corners, and the move label (five numbers per rectangle record). In our implementation, we represent all data members as integers, with no attempt to optimize the size of each record. In such a case, the constant c is $5/2 = 2.5$. However, the record size could be optimized as follows. We need three bits to represent each of the 8 moves possible on grid maps. When the map size doesn't exceed 512×512, we need at most 4×9 bits to represent the coordinates of a rectangle. Therefore, since we would need $3 + 3 = 6$ bits for a UPD record and $3 + 36 = 39$ bits for a CPD record, the constant c would be $39/6 = 6.5$.

In Figure 4 we show the compression factor for three sample values of c: 1, 2.5 and 6.5. Even when scaled down by the largest constant, the compression factor approaches and even exceeds two orders of magnitude.

Figure 5 compares the number of rectangles in a CPD and the number of square blocks in a quad-tree decomposition. CPDs require significantly fewer decomposition blocks. Factors that explain the differences include the following. CPDs decompose a move table around the origin point of that table. As illustrated in Figure 1, with a proper tie breaking when more than one color can be assigned to one location, clusters tend to have straight borders around the origin. CPDs split a rectangle in two at a time, whereas quadtrees split a square block into four. CPDs allow arbitrary sizes and locations of rectangles, allowing more flexibility in covering a map with a (small) set of rectangles.

For the largest map, with more than 50,000 traversable tiles (51,586, to be precise), the CPD requires about 5 mil-

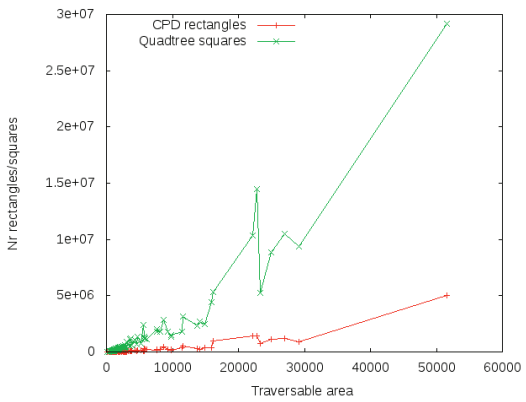

Figure 5: Number of CPD rectangles and quad-tree squares

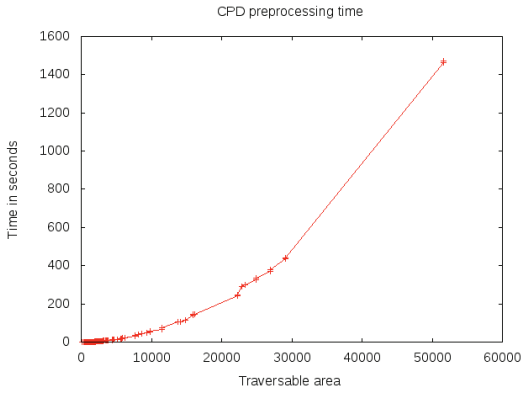

Figure 6: Preprocessing time needed to build CPDs

lion rectangle records. With our implementation, that does not optimize the size of a record, 5 million times 5 integers times 4 bytes per integer sums up to roughly 100 MB. While far from negligeable, it fits easily into the RAM of today's computers. With an implementation with 39-bit ($\approx$ 5-byte) per record, the memory requirements drop to 25 MB. In contrast, a UPD with an optimized record size would require about 2 GB. An unoptimized UPD would require about 20 GB, which is beyond the reach of most today's computers.

Obviously, the memory requirements are much smaller for smaller maps, as Figure 5 shows. For example, a map with 9,836 traversable tiles requires a CPD with 184,034 rectangles. This sums up to about 3.6 MB even with an unoptimized record encoding, being acceptable even for small, mobile gaming devices. An implementation with 39-bit ($\approx$ 5-byte) records would require less than 1 MB.

Finally, Figure 6 plots the time needed to build CPDs offline. The times needed to build the quat-tree decompositions are similar. The largest preprocessing time is about 40 minutes. All other maps require less than 8 minutes each. As mentioned earlier, these times can be reduced by running different iterations of Algorithm 1 on different CPUs.

Conclusion

We introduced compressed path databases (CPDs), a pathfinding method based on pre-computing and compressing all-pair shortest path data. Compared to a standard, A*-based pathfinding system, we report an average speed-up of up to 700 on realistic game maps from Baldur's Gate.

We believe that CPDs open many new research directions. Future work includes applying CPDs to more classes of pathfinding problems, such as moving target search (Ishida and Korf 1992). Existing multi-agent pathfinding algorithms, such as MAPP (Wang and Botea 2009), could possibly be combined with CPDs to reduce the runtime search efforts. Our compression technique exploits redundancies inside individual move tables, but ignores redundancies across different move tables. Exploiting these could result in a further reduction of the memory requirements. In domains where units have different sizes and terrain traversal capabilities, one could start by computing one CPD for each unit type. Then, combine all CPDs together, exploiting redundancies across CPDs to reduce the memory needs.

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.

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

Culberson, J., and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence* 14(4):318–334.

Goldberg, A. V., and Harrelson, C. 2005. Computing the Shortest Path: A* Search Meets Graph Theory. In *ACM-SIAM Symposium on Discrete Algorithms SODA-05*.

Goldenberg, M.; Felner, Ar.; Sturtevant, N.; and Schaeffer, J. 2010. Portal-based true-distance heuristics for path finding. In *Symposium on Combinatorial Search*, 39–45.

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*.

Ishida, T., and Korf, R. E. 1992. Moving Target Search. In *IJCAI*, 204–210.

Kring, A.; Champandard, A. J.; and Samarin, N. 2010. DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds. In *AIIDE-10*, 39–44.

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.

Wang, K.-H. C., and Botea, A. 2009. Tractable Multi-Agent Path Planning on Grid Maps. In *IJCAI*, 1870–1875.