Heuristic-Aided Compressed Distance Databases

Fan Xie
Computing Science, University of Alberta
Edmonton, Canada
fxie2@ualberta.ca

Adi Botea and **Akihiro Kishimoto**
IBM Research Ireland
Dublin, Ireland
{ADIBOTEA,AKIHIROK}@ie.ibm.com

Abstract

Answering point-to-point distance queries is important in many applications, including games, robotics and vehicle routing in operations research. Searching in a graph to answer distance queries on demand can often be too slow. An alternative strategy, taken in methods such as Transit and Hub Labels, is to pre-compute information that can help compute distances much faster. To be practical, such methods need to generate much less preprocessed data than a naive all-pairs distance table.

We present Heuristic-Aid Compressed Distance Databases (HCDs), pre-computed data structures based on the observation that heuristic distance estimations can sometimes coincide with true distances. Compared to a naive all-pairs distance table, we report compression factors of two to three orders of magnitude in a wide range of maps, reducing the memory usage to a reasonable size. Compared to compressed path databases, our approach generally generates smaller databases, and answers query distances faster.

Introduction

Answering point-to-point optimal distance queries on maps is a fundamental building block in many optimization applications, including operations research, games, robotics, and GPS itinerary planning. In vehicle routing, a fleet of vehicles might need to perform a sequence of pick-up and delivery tasks. Optimizing a schedule for these vehicles usually requires a distance matrix among all these related locations. Similarly, in video games, collaborative non-player characters (NPCs) might need to partition and share a set of tasks, taking into account pairwise distances between an NPC and a target location.

Besides the memory usage, the answer time for distance queries is also a major concern, especially when many distance queries need to be answered in real time.

Finding optimal paths with search-based methods, such as the Dijkstra's algorithm (Dijkstra 1959) and A* (Hart, Nilsson, and Raphael 1968), is one way to answer point-to-point distance queries. Despite numerous enhancements introduced in recent years, a search often ends up visiting many locations that are not on an optimal path. In effect, search-based methods can encounter a speed degradation on many maps, depending on factors such as the size and the topology of a map.

Approaches based on preprocessing and data caching can dramatically improve the response time. Fast distance oracles include Transit (Bast, Funke, and Matijevic 2006) and Hub Labels (Cohen et al. 2003; Delling et al. 2014). Transit, however, does not handle cases when the start and the destination nodes are relatively close to each other. Compressed path databases, or CPDs (Botea 2011; Strasser, Harabor, and Botea 2014), can answer move queries very fast. They can be employed to answer distance queries by fetching a full optimal path, step by step, and returning the cost of that path. However, this makes the response time for a distance query linear in the length of the path, with a corresponding performance slowdown.

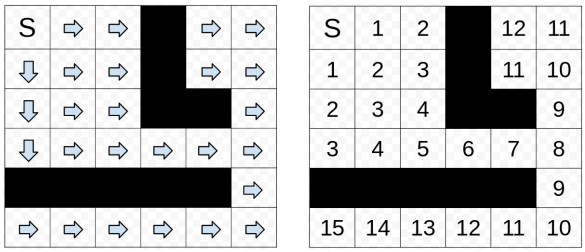
In this paper, we present Heuristic-Aided Compressed Distance Databases (HCD), a novel approach that preprocesses the distance information of a map and compresses it to a concise database with the help of a heuristic function. The resulting database can be used to answer distance queries. Unlike CPDs, that store moves, our databases store distances. Figure 1 illustrates the difference. The main intuition behind this work is the fact that sometimes a heuristic estimation of the distance can be the true distance. HCD takes use of the heuristic function to reduce the amount of cached information and the answering time to distance queries.

We use 4-connected fully-reachable grid maps and the Manhattan heuristic for illustration and evaluation purposes in this paper. HCD takes a map and a heuristic function as input. For every reachable location s, it constructs a *hierarchical anchor tree (HAT)* rooted at s. The nodes of a HAT, which are a (small) subset of the original map locations, such that for any location t on the map, there exist an anchor point p that $d(s,t) = d(s,p)+h(p,t)$, where $h(\cdot,\cdot)$ is the heuristic distance, and $d(\cdot,\cdot)$ is the true distance between two nodes. Figure 2 (b) illustrates a sample HAT that contains two anchor points $A1$ (overlapped with s) and $A2$, and for a location y, $d(s,y)$ can be calculated by $d(s,A2) + h(A2,y)$.

We evaluate HCD and show some preliminary results on 4-connected grid maps taken from Sturtevant's repository (Sturtevant 2012). Compared to a naive, uncompressed distance database, the compression factors achieved by HCDs reach two to three orders of magnitude in a wide

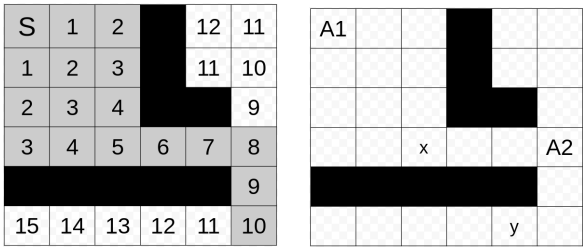
range of game maps. HCD often requires less memory than CPDs (Botea 2011) and answers distance queries faster.

Our work is relevant from an optimization point of view for several reasons. First, as mentioned at the beginning, distance queries are important in a number of optimization problems. Secondly, minimizing the size of a HCD involves a few optimization problems, such as: minimizing the number of rectangles used in a *Manhattan Neighbor* decomposition; and minimizing the size of all HATs defined for an input graph. In this work we implemented greedy approaches to both problems. Improving these further is an interesting direction.



(a) Uncompressed first-move map (b) Compressed distance map

Figure 1: Toy example with a first-move map and a distance map, for a single source location S.



(a) Manhattan neighborhood of location S, in grey.

(b) Anchor tree with two nodes, A_1 (overlapped with S) and A_2.

Figure 2: Example illustrating HCD preprocessing.

Related Work

Compressed path databases (CPDs) (Botea 2011) are oracles that return first-move queries. In other words, given any (s, t) pair of nodes, a CPD provides an optimal move from s towards t. While this is substantially different from our work, a common feature is that the idea of compactly representing *Manhattan Neighbors*, which will be discussed later, using a rectangle decomposition, is inspired from the way CPDs decompose a map (Botea 2011).

In Hub Labels (HL) (Cohen et al. 2003; Abraham et al. 2011; Delling et al. 2014), each node has a list of nodes called forward hub nodes, and a list of nodes called backward hub nodes. Exact distances to forward hub nodes, and from backward hub nodes are recorded as well. Given any two nodes s and t, the forward hub list of s and the backward hub list of t must contain a common node that belongs to a shortest path from s to t. This ensures that $d(s, t)$ can be retrieved by parsing the forward hub list of s and the backward hub list of t. HLs answer distance queries, but they can easily be adapted to both distance queries and first-move queries (Abraham et al. 2012).

Transit (Bast, Funke, and Matijevic 2006) is a distance oracle that can handle queries with the two nodes located reasonably far from each other. The original Transit method has been shown to be effective on road maps. Antsfeld et al. (2012) have observed that Transit's performance decreases on grid maps, and have presented improvements to the algorithm. Transit, however, does not handle cases when the start and the destination nodes are relatively close to each other.

Identifying regions where a simple hill-climbing, based on a given heuristic function, is sufficient for navigation within that region, has successfully been applied in the pathfinding literature (Bulitko, Björnsson, and Lawrence 2010; Lawrence and Bulitko 2013). A key difference is that we employ a similar idea to compactly store a database of exact distances, whereas previous work (Bulitko, Björnsson, and Lawrence 2010; Lawrence and Bulitko 2013) has implemented it within methods that compute full (and not necessarily optimal) paths.

Our hierarchical anchor trees have similarities to subgoal graphs (Uras, Koenig, and Hernández 2013) and jump point structures (Harabor and Grastien 2011). These have been used to compute full paths with search, providing a search space much smaller than an original grid map. Subgoal graphs contain a subset of nodes, such as nodes located at the corners of obstacles on a grid map. A subgoal graph is precomputed, and new start and destination pairs are dynamically inserted with each new pathfinding query. Jump points are a subset of the nodes of the grid map, aiming at eliminating symmetries that can artificially blow the search space. Intuitively, they are points where an optimal trajectory, not pruned as a result symmetry elimination, might include a turn. A jump-point graph can be built either on the fly, or include some preprocessing. Uras, Koenig, and Hernández (2013) discuss briefly how to adapt subgoal graphs to retrieve distances to cooperate with online search.

As the name suggests, memory-based heuristics provide (admissible) estimations of distances, not necessarily exact values, such as Landmarks based heuristics (Goldberg and Harrelson 2005; Cazenave 2006; Björnsson and Halldórsson 2006; Sturtevant et al. 2009) and pattern databases (Culberson and Schaeffer 1998). While HCD can compute the optimal distances for all pairs of locations without performing any state-space search, memory-based heuristics must be combined with runtime heuristic search to compute optimal paths.

HCD

In this paper, we only consider fully reachable maps. For maps that contains more than one disconnected components, they can be easily treated as several different maps.

Before we introduce the detailed algorithms, we formally define the following notions.

Definition 1. *Locations a and b are **Manhattan Neighbors***

(MN) iff $d(a,b) = h(a,b)$. MN_a *is used to denote the largest set in which each location* b *is a Manhattan Neighbor of* a.

Definition 2. *For a 4-connected grid map and a source location* S*, a* ***Hierarchical Anchor Tree*** *(HAT), denoted as* HAT_S*, is a tree rooted at* S*. The nodes of HAT are called anchor points which are locations on the map.* S *is called the root anchor point. An edge between a parent anchor point* p *and a child anchor point* c *has a cost that equals to* $h(p,c)$*. Let* $pc(p,q)$ *be the total cost of edges in the path from* p *to* q *in* HAT_S*.* HAT_S *must satisfy the following conditions:*

1. S *is in* HAT_S*;*
2. *For any anchor point* a*,* $d(S,a) = pc(S,a)$*;*
3. *For any location* p *in the map, there exists at least one anchor point* a *such that* a *is in* MN_p *and* $d(S,p) = d(S,a) + h(a,p)$.

See Figure 2 again as an example of MN_S and HAT_S. HAT_S consists of two anchors $A1(= S)$ and $A2$ and has only one edge (i.e., $A1 \rightarrow A2$).

There may be more than one HAT_S, and we will describe our algorithm to construct it later. The following theorem indicates how to calculate the optimal distance from any location p to S with HAT_S and MN_p.

Theorem 1. *For any accessible location* p*,* $d(S,p)$ *can be calculated by*

$$\min_{a \in HAT_S \cap MN_p} pc(S,a) + h(a,p)$$

Proof of Theorem 1. For any $a \in HAT_S \cap MN_p$, $pc(S,a) + h(a,p) \geq d(S,a)$. According to Definition 2 there exist at least one a so that $pc(S,a) + h(a,p) = d(S,a)$, so the minimum value in Theorem 1 must equal to $d(S,a)$. □

$HAT_S \cap MN_x$ and $HAT_S \cap MN_y$ are respectively $\{A1, A2\}$ and $\{A2\}$ in Figure 2(b). $d(S,x)$ and $d(S,y)$ are therefore calculated as follows:

$$\begin{aligned} d(S,x) &= \min(pc(A1,A1) + h(A1,x), \\ &\qquad pc(A1,A2) + h(A1,x)) \\ &= \min(5, 8+3) = 5, \\ d(S,y) &= pc(A1,A2) + h(A2,y) \\ &= 8 + 3 = 11. \end{aligned}$$

Calculating $d(S,p)$ needs to compute $I = HAT_S \cap MN_p$ by checking if each anchor point in HAT_S is in MN_p as well as $pc(S,a)$ by traversing a path from S to a in HAT_S. This computational overhead is much smaller than CPD checking all the locations along the optimal path, because HAT_S and I contain a very small number of anchor points in practice.

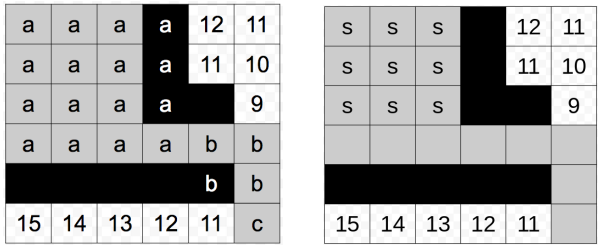
Computing and Compressing MN

Algorithm 1 outlines the procedure of computing and compressing MN_n for each reachable location n. After the algorithm calculates the optimal distance for all reachable locations from n by using Dijkstra's algorithm, it calculates MN_n by comparing $d(p,n)$ with $h(p,n)$ for each reachable location p. The algorithm then compresses MN_n either by using either rectangle regions or previously compressed MN_m for other location m.

Below is the definition of *Manhattan Neighbor Map (MNMap)* that our algorithm uses to compress MN.

Definition 3. *For a 4-connected grid map and a source location* S*, a* ***Manhattan Neighbor Map*** *for* S*, denoted as* $MNMap_S$*, is a set of a rectangles, so that all the locations in* MN_S *are covered and none of the other reachable locations outside* MN_S *is covered.*

Figure 3(a) illustrates the basic idea of compressing based on rectangle regions that can cover a MN colored by grey: the MN is divided into a number of different rectangles for each of which only the left-top and the right-bottom locations are stored. In Figure 3(a), 3 rectangles marked by a, b and c are needed to construct a MNMap. Obstacle locations can be included in rectangle regions.



(a) Manhattan Neighbors To Rectangles (b) Reuse Rectangles

Figure 3: Example of compressing MN

Algorithm 1 build a compressed MN database

```
1: for each reachable location n do
2:     D(n) ← Dijkstra(n)
3:     T(n) ← ComputeMN(D(n))
4:     L(n) ← CompressMN(T(n))
```

Our algorithm currently uses the following greedy strategy to generate rectangles to cover MN_S:

1. Select the source location S and set the current expanding rectangle r to include S.
2. Keep expanding r in the direction of the longest[1] edge of the enclosed rectangle until r cannot be expanded any more. Note, obstacle locations are allowed in r, but locations already covered by previous rectangles are not allowed in r.

[1] Although any tie-breaking rule can be applied here, our current implementation breaks ties in order of the top, right, bottom and left directions.

3. Select a new location $t \in MN_S$ that is uncovered by any rectangle and closest to source location S, replace r with t, and then go to step 2, until all locations are in $MNMap_S$.

MN_a and MN_b are often identical even if $a \neq b$. For example, in Figure 3(b), all locations marked by "s" have the same MN illustrated by grey tiles. In compressing MN, our algorithm checks if there is an identical MN already compressed as a MNMap. If this is the case, the algorithm reuses that MNMap to further reduce the memory usage.

Algorithm 2 build a compressed HAT database

Input Manhattan Neighbor Database $MNdb$

1: **for** each reachable location n **do**
2: $T(n) \leftarrow Dijkstra(n)$
3: $L(n) \leftarrow ComputeHAT(T(n), MNdb)$

Computing and Compressing HAT

Algorithm 2 outlines the basic procedure of computing HATs. Since computing HATs needs to determine whether two locations are MN or not, it needs a precomputed MN database as input. In order to compute HAT_S, the following algorithm is performed:

1. Put all reachable locations into a list L, in ascending order of $d(S, \cdot)$ calculated by Dijkstra's algorithm.
2. Set S to the root anchor of HAT_S. Set the current working anchor point $a = S$;
3. Remove all locations in MN_a from L;
4. Dequeue a top element p from L. If L is empty, terminate;
5. Find a location q among the direct four neighbors of p that satisfies the following conditions:
   - q is not in L;
   - $d(S, p) = d(S, q) + 1$;
   - An anchor point a' in current HAT_S such that q is in $MN_{a'}$ and $d(S, q) = d(S, a') + h(a', q)$.

   Update the current working anchor point $a = q$, and insert q into HAT_S with an edge of $a' \rightarrow q$. Go back to step 3.

Theorem 2. *The algorithm defined above generates a HAT_S that satisfies Definition 2.*

Proof Sketch of Theorem 2. We give a proof sketch of an important property that there exits at least one qualified location q given such a location p in step 5 above. Assume a location p' is the direct predecessor of p in one optimal path from S to p, so $d(S, p) = d(S, p') + 1$. Because p is the closest location to S in L, p' is not in L. Because p' is not in L, there must exist a location a' in current HAT_S such that p' is in $MN_{a'}$ and $d(S, p') = d(S, a') + h(a', p')$. p' is a qualified q. □

Figure 4 illustrates an example of computing a HAT. After $S = A1$ (the source location) is added to HAT_S, all locations with light grey are removed from L. Later, the dark grey location with $d(A1, p) = 9$ is selected as p in step 4, followed by $A2$ and $A1$ being selected as a qualified q and a', respectively.

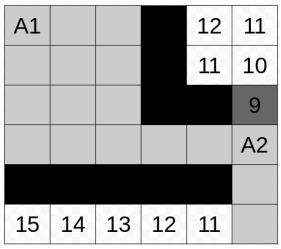


Figure 4: Compute HAT.

Similar to MN, many different source locations can reuse HATs generated previously, thus enabling to decrease the memory usage. More specifically, before the algorithm computes HAT_S, it checks if there is any HAT_T such that a tree constructed from HAT_T by replacing only T by S is consistent with the definition of HAT_S. In this case, instead of generating HAT_S, HCD remembers that HAT_T can be reused for S. HAT_S is dynamically generated during a distance query starting from S.

Experiments

We evaluate empirically two versions of our approach, HCD-noreuse and HCD-reuse. HCD-noreuse builds a separate HAT for every location s, as well as a separate MNMap. In contrast, HCD-reuse identifies cases where the same HAT, except for its root, can be re-used across multiple starting locations. It also reuses MNMaps across locations. We use compressed path databases as the baseline, which is implemented in the Copa program (Botea 2012) and can be downloaded from the website of the GPPC 2012[2]. The comparison also includes a naive way of storing all-pairs distances without any compression.

Experiments are performed on a set of game maps from three games: *Dragon Age: Origins*, *Baldurs Gate II* and *Warcraft III*. The maps are downloaded from Sturtevant's repository (Sturtevant 2012).

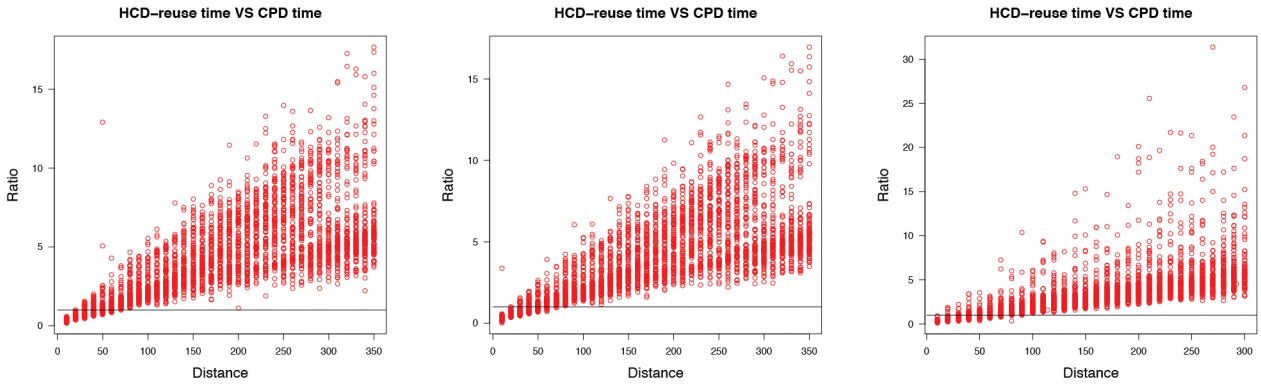
Table 1 compares the size of databases generated with CPDs, HCD-noreuse and HCD-reuse. HCD-reuse generally generates the smallest database, except for the den401d map. In den401d, previously generated HATs can not be reused, while HATs are frequently reused in other maps. The reason of this behavior on map den401d is still unclear, which would be explored later.

In our implementation for 4-connected grid maps, the naive uncompressed distance databases (UDDs), store every distance as a 32-bit integer. The last three columns in Table 1 show the compressing factor, computed as the size of the UDD divided by the size of the corresponding compressed database. HCD-reuse achieve compressing factors from 79 to 4,280. Generally, HCD-reuse tends to have larger compressing factors for larger maps, but the map topologies might also change the compressing factor significantly. For

[2]http://movingai.com/GPPC/

		Size of Database in MB				Improving Factor over UDD		
Map	# of states	UDD	CPD	HCD-noreuse	HCD-reuse	CPD	HCD-noreuse	HCD-reuse
Baldurs Gate II								
AR0018SR	2860	33.0	**0.37**	1.95	**0.37**	88	17	88
AR0012SR	6175	148.8	**0.81**	4.30	**0.81**	184	35	183
AR0412SR	7626	227.8	1.00	2.70	**0.31**	228	84	735
AR0603SR	13764	725.4	1.60	11.75	**0.68**	453	62	1064
Dragon Age: Origins								
den403d	2036	16.1	0.26	0.35	**0.20**	63	46	79
isound1	2976	34.6	0.36	0.19	**0.04**	95	182	961
den401d	11456	504.4	**1.30**	7.95	5.43	388	63	93
arena2	24311	2256.0	5.30	14.90	**0.67**	426	151	3352
Warcraft III								
deadwaterdrop	76028	22258.0	29.00	62.50	**5.2**	768	356	4280

Table 1: Size comparison for CPD, HCD-noreuse and HCD-reuse, and uncompressed distance databases (UDD).



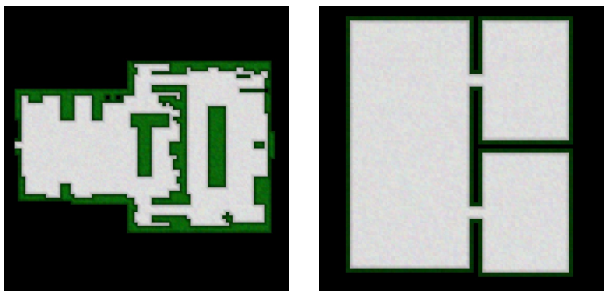
(a) AR0603SR (b) arena2 (c) deadwaterdrop

Figure 5: Compare the time usage of CPD and HCD-reuse in returning the optimal distance between two locations. The extra horizontal line represents y=1. There are 100 instances for every distance.

example, den403d and isound1 are close in the number of states, but they have quite different compression factors with HCD-reuse. Figure 6 shows the two maps. As the map isound1 is topologically much simpler than the den403d map, it generates smaller HATs and also allows to reuse HATs more frequently.

Figure 5 compares the CPD and HCD-reuse in terms of the response time to distance queries. As the response times of HCD-noreuse and HCD-reuse are very similar, we only plot HCD-reuse time data. Every point represents a distance query. The x-axis represents the exact distance, and the y-axis represents the ratio between the CPD response time and the HCD-reuse response time, which is computed as the CPD response time divided by the HCD-reuse response time. There is one horizontal line with y equals to 1 in each figure. All points above this line indicate that HCD-reuse is faster. The ratios in Figure 5 roughly increase linearly with the distance.

Using the AR0603SR map, Figure 7 illustrates why this tendency occurs. The CPD response time is linear over the distance. CPDs need to retrieve all moves step by step in one optimal path, and report the distance at the end. In contrast to CPDs, the HCD-reuse response times remain relatively stable as the distance grows. It is because, for HCD-reuse, we only need to calculate the equation in Theorem 1.

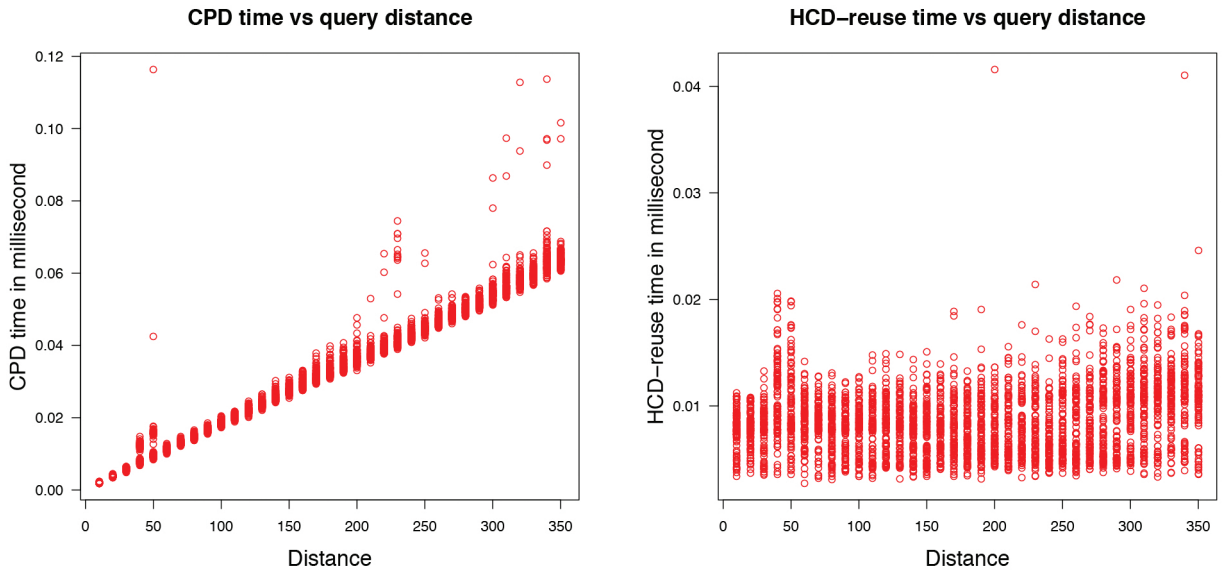


(a) den403d (b) isound1

Figure 6: The den403d and isound1 maps.

Conclusion and Future Work

In this preliminary work, we introduce HCDs, which make use of the Manhattan distance heuristic to compress size of the optimal distance database for 4-connected grid maps. Compared to compressed path databases, preliminary re-



(a) time usage of CPD

(b) time usage of HCD-reuse

Figure 7: Time usage of CPD and HCD-reuse over query distance on the same 3500 pairs of locations (100 pairs for each distance) on the AR0603SR map.

sults show that our approach generally generates smaller databases, which also reply distance queries faster.

Future work includes a more detailed empirical analysis, including benchmark methods such as Hub Labels, and extending HCD to 8-connected grid maps. We plan to further reduce the size of HCDs, taking advantage of work in the area of covering polygons with rectangles (Franzblau and Kleitman 1984). Reducing the size of the union of HATs is another promising future direction.

References

Abraham, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F. 2011. A hub-based labeling algorithm for shortest paths on road networks. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, 230–241. Springer.

Abraham, I.; Delling, D.; Fiat, A.; Goldberg, A. V.; and Werneck, R. F. 2012. HLDB: Location-based services in databases. In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, 339–348. ACM Press. Best Paper Award.

Antsfeld, L.; Harabor, D.; Kilby, P.; and Walsh, T. 2012. Transit routing on video game maps. In *AIIDE*.

Bast, H.; Funke, S.; and Matijevic, D. 2006. Transit – ultrafast shortest-path queries with linear-time preprocessing. In *In 9th DIMACS Implementation Challenge*.

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

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 2011, October 10-14, 2011, Stanford, California, USA*.

Botea, A. 2012. Fast, optimal pathfinding with compressed path databases. In *Proceedings of the Symposium on Combinatorial Search SoCS*.

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.

Cohen, E.; Halperin, E.; Kaplan, H.; and Zwick, U. 2003. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing* 32(5):1338–1355.

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

Delling, D.; Goldberg, A. V.; Pajor, T.; and Werneck, R. F. 2014. Robust distance queries on massive networks. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, Lecture Notes in Computer Science. Springer. to appear.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Franzblau, D., and Kleitman, D. 1984. An algorithm for covering polygons with rectangles. *Information and Control* 63(3):164 – 189.

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*.

Harabor, D. D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*.

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

Lawrence, R., and Bulitko, V. 2013. Database-driven real-time heuristic search in video-game pathfinding. *IEEE Trans. Comput. Intellig. and AI in Games* 227–241.

Strasser, B.; Harabor, D.; and Botea, A. 2014. Fast first-move queries through run-length encoding. In *Seventh Annual Symposium on Combinatorial Search*.

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.

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

Uras, T.; Koenig, S.; and Hernández, C. 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *ICAPS*.