# Contracting and Compressing Shortest Path Databases

**Bojie Shen, Muhammad Aamir Cheema, Daniel D. Harabor, Peter J. Stuckey**
Faculty of Information Technology, Monash University, Melbourne, Australia
{bojie.shen, aamir.cheema, daniel.harabor, peter.stuckey}@monash.edu

## Abstract

Compressed Path Databases (CPD) are powerful database driven methods for shortest path extraction in grids and in spatial networks. Yet CPDs have two main drawbacks: (1) constructing the database requires an offline all-pairs precompute, which can sometimes be prohibitive and; (2) extracting a path requires a number of database lookups equal to its number of edges, which can be costly in terms of time. In this work, we consider how CPD methods can be improved and enhanced by: (i) *contracting* the input graph before preprocessing and; (ii) limiting the preprocessing step to only a selected subset of graph nodes. We also describe a new bi-directional path extraction algorithm which we call CH-CPD. In a range of experiments on road networks, we show that CH-CPD substantially improves on conventional CPDs in terms of preprocessing costs and online performance. We also report convincing query time improvements against a range of methods from the recent literature.

## Introduction

Shortest path queries are one of the most ubiquitous practical uses of computing. This query is the underlying computation in route planning software (Delling et al. 2017), such as Bing and Google Maps. Shortest paths are also used as input for higher-level planning tasks, such as ride sharing (Alonso-Mora et al. 2017), traffic assignment (Luxen and Sanders 2011) and for computing collision-free plans for teams of moving agents (Stern et al. 2019).

In each application setting, shortest path problems are myriad and solutions are needed extremely fast. Among the leading methods in this area are Compressed Path Databases (Botea 2011; Strasser, Harabor, and Botea 2014): a family of techniques that forgo conventional state-space search and instead *extract* shortest paths using precomputed *first-move* data. A CPD can be understood as an oracle `CPD[s, t]` which, given a start-target pair, resp. $s$ and $t$, tells the identity of the first edge on the optimal path: from $s$ toward $t$. Using a simple recursive procedure, CPDs can extract entire paths at ultra-fast speed (Sturtevant et al. 2015). There are however two drawbacks: (1) the time complexity for building the database is quadratic in the size of the input graph, which can be prohibitive in some cases; (2) the query time performance grows supra-linearly with the number of edges in the shortest path,[1] which affects performance when there are many edges to extract.

To mitigate the disadvantages of CPDs we investigate connections with another family of successful, but search-based, speedup techniques called Contraction Hierarchies (CH) (Geisberger et al. 2008, 2012). The CH method can be understood as a type of embedded graph abstraction (Holte et al. 1996). During preprocessing, additional "shortcut" edges are added to the graph. During online search, these shortcuts help the search to bypass many "unimportant" nodes which would otherwise need to be expanded. Moreover, the total number of edges on a path, from start to target, is reduced. For this reason we consider contracted graphs in combination with CPDs. First, CH graphs help to improve CPD online performance, by reducing the number of lookups we need to perform when extracting a path. Second, CH graphs help to reduce CPD offline costs, by allowing us to speed up the many Dijkstra searches required to construct first-move data. In broad strokes, our strategy is as follows:

- We compute *distance tables* for a small number of important CH nodes. We show that these tables can be used to provide bounds for, and therefore can help to speed up, each of the many Dijkstra searches necessary for computing first-move data.
- We also compute first-move data for only a selected subset of CH nodes. This further reduces the overall time needed for CPD precomputation and also lowers the storage cost.
- We develop a new bi-directional query algorithm, which combines online search in a CH with CPD path extraction. This allows us to compute shortest paths substantially faster than either CH or CPD.

We compare our approach on several well known road network benchmarks. Our principal points of comparison are SHP (Li et al. 2017) and PHL (Akiba et al. 2014): two recent and also database-driven query algorithms. We show that, for computing shortest paths, CH-CPD offers substantially better performance and has overall smaller storage costs.



[1] Complexity per lookup is logarithmic in the size of the compressed data w.r.t the corresponding row.

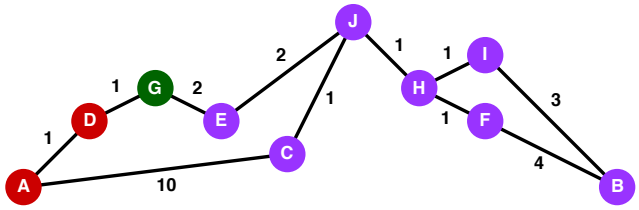


Figure 1: From the source node G, the optimal first move to any node colored red (resp. purple) is D (resp. E).

| Ordering | G | D | A | C | J | E | H | F | B | I |
|---|---|---|---|---|---|---|---|---|---|---|
| G | * | D | D | E | E | E | E | E | E | E |
| J | E | E | E | C | * | E | H | H | H | H |
| I | H | H | H | H | H | H | H | H | B | * |

Table 1: First moves for G, J and I for the example of Fig. 1

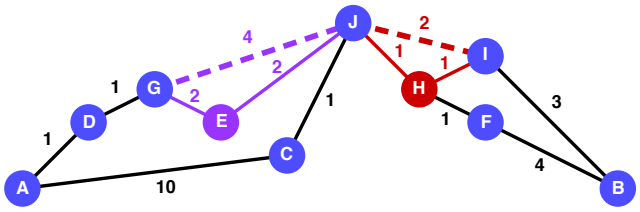


Figure 2: We show the result of contracting E (resp. H) in purple (resp. red). Dashed edges indicate shortcut edges.

| Ordering | G | D | A | C | J | E | H | F | B | I |
|---|---|---|---|---|---|---|---|---|---|---|
| $d(\mathtt{J}, \_)$ | 4 | 5 | 6 | 1 | 0 | 2 | 1 | 2 | 5 | 2 |
| $d(\mathtt{A}, \_)$ | 2 | 1 | 0 | 7 | 6 | 4 | 7 | 8 | 11 | 8 |
| $d(\mathtt{B}, \_)$ | 9 | 10 | 11 | 6 | 5 | 7 | 4 | 4 | 0 | 3 |

Table 2: Arrays of costs for nodes J, A and B shown in Fig. 1

## Preliminaries

Let $G = (V, E, w)$ be a (directed or undirected) graph, with nodes $V$, edges $E \subseteq V \times V$ and $w : E \to \mathbb{R}^+$ a weight function that maps each edge $e \in E$ to a non-negative weight $w(e)$. A *path* $p$ from $s$ to $t$ in $G$ is a sequence of nodes $\langle n_0, n_1, n_2, \ldots, n_{k-1}, n_k \rangle$, where $k \in \mathbb{N}^+$, $n_0 = s$, $n_k = t$, and $(n_i, n_{i+1}) \in E, 0 \leq i < k$. The *length* of the path is $|p| = \sum_{i=0}^{k-1} w(n_i, n_{i+1})$ and $d(s, t)$ denotes the length of the *shortest path*, from $s$ to $t$ in $G$. For exposition only, we will assume $G$ is undirected.

**Compressed Path Database:** A CPD is a data structure that specifies the first edge (equiv. *move*) $m$ on an optimal path, from any node $s$ towards any node $t$ (Botea 2011). We often use the endpoint $m$ to refer to a first-move edge $(s, m)$, with the current node $s$ being clear from context. Each CPD is constructed *offline* in a preprocessing phase. For each source node $s \in V$, we run a complete Dijkstra search. The result is a *first move table*, $T(s)$, which records *all* optimal first moves; from $s$ to any reachable target $t \in V$. The table is compressed via Run-Length Encoding (RLE) (Strasser, Harabor, and Botea 2014) and stored to the disk, concluding one iteration. Being independent, the iterations can be run in parallel, with a speed-up linear in the number of processors. For the effectiveness of RLE compression, the columns of table are ordered by Depth First Search (DFS) as suggested by Strasser, Botea, and Harabor (2015).

**Example 1.** *Consider the graph shown in Figure 1. The first move tables for nodes G, J and I are shown in Table 1, where * is a wildcard symbol that can be combined with any run. The compressed RLE string of each row is respectively [(1,D),(4,E)], [(1,E),(4,C),(6,E),(7,H)] and [(1,H),(9,B)].*

With a CPD in hand, we begin the *online* phase of the algorithm. Given a start-target pair $(s, t)$ (equiv. *instance*). our task is to extract an optimal path $\mathsf{CPDPath}(s, t)$. The implementation of this function requires a simple binary search through a compressed string of symbols representing the first-move row $T(s)$ (Strasser, Harabor, and Botea 2014). Once a move is extracted it can be immediately followed, giving rise to a simple recursive procedure that terminates after reaching the target (or after $k$ steps for a partial path).

**Contraction Hierarchies:** A CH is an augmented multi-level graph that can be exploited to speed up pathfinding search. Building a CH is a simple process requiring only the repeated application of a **contraction** operation to the nodes of the input graph $G$. In broad strokes:

1. Apply a total lex order $\mathcal{L}$ to the nodes of $G$. We use the ND order suggested in (Dibbelt, Strasser, and Wagner 2016).
2. W.r.t. $\mathcal{L}$, choose the least node $v$ from the graph that has not been previously selected.
3. (Contraction) Add to $G$ a shortcut edge $(u, w)$ between *each* pair of in-neighbour $u$ and out-neighbour $w$ of $v$ for which: 1) $u$ and $w$ are both lexically larger than $v$; and 2) the shortest path from $u$ to $w$ passes through $v$. To reduce the number of shortcuts added in $G$, the subpath $\langle u, v, w \rangle$ should be both unique and optimal. Fewer shortcuts improve query performance but verifying local optimality requires additional pre-processing time.

**Example 2.** *In Figure 2, we contract a toy graph in alphabetical lex order. Note how shortcut edges (dashed) can connect start and target faster than would otherwise be possible. Without shortcuts, the optimal path from A to B has 7 edges: $\langle$A, D, G, E, J, H, I, B$\rangle$. An equivalent-cost path, with shortcuts, traverses only five edges: $\langle$A, D, G, J, I, B$\rangle$.*

A core idea of contraction hierarchies is that shortcut edges can bypass one or more intermediate nodes in a single step. However, for each shortcut edge $(u, w)$ and each intermediate node $v$ we have $f(v) \leq f(w)$; i.e., given a monotonically increasing cost function $f$, a simple best-first search will usually expand $v$ before $w$ in order to compute an optimal path. To achieve a speedup, authors in (Geisberger et al. 2008) divide the set of edges $E$ into two as follows:

- $E_\uparrow = \{(u, v) \in E \mid u <_\mathcal{L} v\}$
  (i.e., the set of all "up" edges); and
- $E_\downarrow = \{(u, v) \in E \mid u >_\mathcal{L} v\}$
  (i.e., the set of all "down" edges).

The following results, paraphrased here, are due to (Geisberger et al. 2008).

**Lemma 1.** ***(ch-path)****: For every optimal path $\pi^*_{s,t}$ in $E$, there is a cost equivalent ch-path $\pi'_{s,t}$ whose prefix $\langle s, \dots k \rangle$ is found in $E_\uparrow$ and whose suffix $\langle k \dots t \rangle$ is found in $E_\downarrow$.* □

**Corollary 1.** ***(apex node)****: Every ch-path has a node $k$ which is lexically largest among all nodes on the path.* □

Following Lemma 1, a natural decomposition of the shortest path problem in a contraction hierarchy is the following: first compute a subpath $\langle s, \dots, k \rangle$ in $E_\uparrow$; next, compute a second subpath $\langle k, \dots, t \rangle$ in $E_\downarrow$. All that remains is to identify a suitable node $k$ which minimises the total distance. BCH is a variation on bi-directional Dijkstra search that was developed specifically for solving such problems.

In forward direction, BCH considers only the outgoing edges in $E_\uparrow$. In reverse direction, BCH considers only the incoming edges in $E_\downarrow$.[2] Each meeting point of the two search frontiers corresponds to a tentative shortest path. Unlike standard bi-directional Dijkstra search, which can be terminated as soon as the sum of the minimum $f$-values on open lists for both directions is no less than the length of best candidate path, BCH continues until it can prove the meeting point $k$ minimises the total distance between $s$ and $t$, i.e., BCH stops when the minimum $f$-value on both open lists are at least as large as the best candidate path found so far (or when both lists are empty, if there is no such path). Though simple, BCH remains state of the art for pathfinding on road networks with millions of nodes (Geisberger et al. 2012).

**Landmarks:** Landmarks (Goldberg and Harrelson 2005) is a method for generating admissible estimates in shortest path search. We use landmarks to further improve the performance of the BCH query algorithm. For each landmark $l \in L$, we select $l$ on the border of the graph following the same procedure as Sturtevant et al. (2009), and compute an array that records the distance to every other node. The arrays of costs are exploited to lower-bound the true distance, from any node $u$ to any other node $v$:

$$landmark(u, v) = \max\{|d(u, l) - d(v, l)| \mid l \in L\}$$

**Example 3.** *Consider Table 2 that shows arrays of costs for three landmarks. The lower-bound distance from* `G` *to* `F` *is* $\max\{|4-2|, |2-8|, |9-4|\} = 6$ *(i.e., the true distance).*

## Combining CH and CPD

Given a weighted graph $G$, we first construct a Contraction Hierarchy. With this contracted graph in hand, a CPD can now be constructed by following the same general procedures already described. However, we introduce three changes: (i) we modify the successor generating function of the Dijkstra algorithm so that every node is reached along a ch-path; (ii) we enhance the basic Dijkstra algorithm using precomputed *distance tables*, which speeds up the computation of first-move data; (iii) we store compressed data for only a subset of all graph nodes.

[2]Other edges from $E$, such as incoming up-edges and outgoing down-edges are safely discarded by BCH to save space.

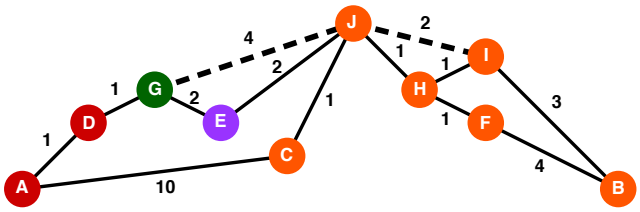


Figure 3: Constructing CPD on top of CH. The source node `G` is highlighted as green. The first move on the optimal path from source node to any node are `D`, `E` and `J` shown as red, purple and orange, respectively.

**CH-Paths in Dijkstra Search:** Recall a ch-path always has an apex node which is lexically larger than all the other nodes on the path. For a given source $s$ and target $t$, deconstructing the ch-path gives following three cases:

1. Up ch-path: $t$ is an apex node (i.e., $t >_\mathcal{L} v$ for $v \in \langle s, \dots t-1 \rangle$)
2. Up-Down ch-path: an intermediate node $k$ is an apex node (i.e., $k >_\mathcal{L} v$ for $v \in \langle s, \dots k-1 \rangle \mid \langle k+1, \dots t \rangle$)
3. Down ch-path: $s$ is an apex node (i.e., $s >_\mathcal{L} v$ for $v \in \langle s+1, \dots t \rangle$)

In other words, before the apex node, every subsequent node on a ch-path is lexically larger than the previous. After the apex, every subsequent node on a ch-path is lexically smaller than the previous. We modify Dijkstra search to only consider ch-paths by way of a simple neighbour pruning rule called UTD (Up-Then-Down) (Harabor and Stuckey 2018). The idea is simple: (i) if the predecessor of the current node is lexically larger than the current node we prune all up successors (this covers case 3); (ii) if the predecessor is lexically smaller than the current node we generate all successors, up or down (this covers case 1 and 2). The only paths disallowed by UTD involve edges to lexically smaller nodes followed by edges to lexically larger nodes; i.e., non ch-paths.

**Distance Tables Enhancement:** In a CH, nodes with high lex values appear often as the apex node along a great many shortest ch-paths. We exploit this observation to reduce the cost of first-move preprocessing, as follows:

**Caching:** For a given contracted graph $G$, we first select a set of nodes $C$ that have the largest lex values. For each node $v_c \in C$, we run our modified single-source Dijkstra search and store not only a first move table but also a table of distances, from $v_c$ to every other node in the graph.

**Pruning:** For each remaining node $v \notin C$, we start a single-source Dijkstra search as usual but we never generate any successors for any cached node $v_c \in C$. When expanding a cached node we instead refer to the cached distances and attempt to relax the tentative estimate $g(v, v')$ for each node $v' \in G$. We perform the relaxation if $g(v, v') > d(v, v_c) + d(v_c, v')$ where $d(v_c, v')$ is the cached distance. The first move table of $v$ is also relaxed accordingly using the first move on the current optimal path $\mathcal{P}(v, v_c)$. Simultaneously relaxing all tentative distances gives tight upper-bounds sooner and helps the search to terminate faster. First,

we never expand successors for any node $v_c \in C$. Second, all distance information usually propagated by such nodes is copied into the first move table of $v$. That means we also never expand any node $n \notin C$ where the apex of the optimal ch-path from $v$ resides in $C$. It is easy to see this approach limits the search space and allows for faster termination.

**On Demand Reading:** Storing all distance tables in RAM requires $O(|C||V|)$ space and can be prohibitive as $|C|$ grows large. We thus store the tables to disk and retrieve them on demand: whenever the search expands a node $v_c \in C$. When loading, distance data is mapped into virtual memory at once which avoids unnecessary I/O operations.

**Example 4.** *Consider building the CPD for the graph of Figure 3. Assume we have already constructed $C = \{\texttt{J}\}$. We begin a Dijkstra search from* `G` *and update tentative distances and first move tables when expanding* `J`*. The search terminates after expanding* `D`*,* `A`*,* `E` *and* `J`*, without exploring the rest of the graph. The first moves to* `C`*,* `H`*,* `I`*,* `F`*, and* `B` *are the same as for* `J` *since the shortest ch-path from* `G` *to each of these nodes is via* `J`*.*

**Partial CH-CPD:** CH and cost caching help to speed up each Dijkstra search, which means computing first-move data is faster in practice compared to the original baseline. However the worst-case time complexity is unchanged: $O(|V||E| + |V|^2 \log |V|)$. To further improve preprocessing costs, we propose to compute and store first-move data for only a selected subset of the full CH graph (i.e., $T \subseteq G$).

There are many ways to choose $T$ but an important requirement is that the subgraph is closed, in other words, for each pair of nodes, $s, t \in T$, the shortest ch-path must also belong to $T$. Since every ch-path is always an Up-Down path, we can therefore select any subset of vertices which is lexicographically upwards closed, i.e., $T(v_l) = \{v_l \in V \mid v_l >_{\mathcal{L}} v \text{ for } v \in V \setminus T\}$. Clearly all Up-Down paths between vertices in $T(v_l)$ only make use of vertices in $T(v_l)$.

With $T$ selected, we now compute a partial CPD for some upper part of the contraction hierarchy. However we also need to define a new shortest path algorithm, to support queries for arbitrary pairs of start $s$ and target $t$. Here we combine the BCH query method with CPD path extraction. The approach is similar to End Point Search (EPS) (Shen et al. 2020) in that, when a node $v \in T$ is expanded, we use the CPD to extract candidate shortest paths: from $v$ to all other nodes $v' \in T$ that have been expanded in the opposite direction. We give a detailed description in the next section.

## Bi-directional CPD Search

Our search algorithm is similar to BCH but with some fundamental differences. Firstly, we employ bi-directional A* search instead of bi-directional Dijkstra. In particular, for each $s$ and $t$ query, our search is guided by a landmark heuristic; i.e., the $f$-value for a node $v$ is $g(s, v) + landmark(v, t)$ with $g(s, v) \geq d(s, v)$ being a tentative upper bound for the optimal distance from $s$ to $v$. Secondly, when the search expands a CPD node (i.e., a node that exists in the partial CPD), we do not generate any successors. Instead, the partial CPD is used to extract paths/distances to all CPD nodes expanded in the other direction. We also reduce the number of first move extractions using pruning rules discussed later. Similar to BCH, our approach considers only "up" edges and uses *stall-on-demand*: a well known and performance improving technique (Geisberger et al. 2008).

**Algorithm 1:** Bi-directional CPD Search

**Input:** $s$: start, $t$: target, CPD: for vertices in the set $T$
**Output:** an optimal path from $s$ to $t$
**Init:** $V_s = \emptyset, V_t = \emptyset, p = \langle\rangle, |p| = \infty, R_s = \emptyset, R_t = \emptyset$

```
1  cur = s; opp = t;
2  Q_s = {s}; Q_t = {t}
3  while both A* searches are not exhausted do
4      v = pop(Q_cur);
5      if v ∈ T then // v is a CPD node
6          V_cur ← V_cur ∪ {v}
7          for each v' ∈ V_opp do
8              if g(cur, v) + d(v, v') + g(v', opp) < |p| then
9                  p ← ⟨cur, v, v', opp⟩
10     else
11         if v ∈ R_opp then // v reached by opp search
12             if g(cur, v) + g(v, opp) < |p| then
13                 p ← ⟨cur, v, opp⟩
14         A*Expand(v, Q_cur, R_cur)
15     if Q_opp is not empty then
16         cur, opp ← opp, cur;
17 return p after unpacking it;
```

The pseudo-code of our approach is shown in Algorithm 1. We start the bi-directional search from $s$ and $t$ with separate queues $Q_s$ and $Q_t$. We use $cur$ (resp. $opp$) to denote the current (resp. opposite) direction in which the search is expanding; i.e., if $cur$ is start then $opp$ is target and vice versa. The optimal path $p$ and the optimal path length $|p|$ are initialized to be empty and infinity, respectively.

In each iteration, we pop the node $v$ with the smallest $f$-value from the current queue $Q_{cur}$ to expand. If this node is in the CPD (i.e., $v \in T$), we add this to $V_{cur}$ to record that this CPD node is expanded by the search from $cur$. Then, we use the CPD to efficiently compute the shortest path/distance from $v$ to each CPD node $v' \in V_{opp}$ found by the search from the opposite side $opp$. If, for any $v' \in V_{opp}$, $g(cur, v) + d(v, v') + g(v', opp)$ is smaller than $|p|$ (length of the current optimal path $p$), we update the optimal path to be $\langle cur, v, v', opp\rangle$. Note that this only records two intermediate CPD nodes on the path $p$. The complete optimal path is recovered once at the end of the algorithm. Also, note that $g(cur, v)$ and $g(v', opp)$ are already known due to the two A* searches from $cur$ and $opp$, respectively, whereas $d(v, v')$ is efficiently extracted using the CPD.

If the node $v$ is not a CPD node, we follow bi-directional search. For node $v$ that we have met from the other direction (i.e., $v \in R_{opp}$), we calculate the path length from $s$ to $t$ via $v$ and replace $p$ with this path if it is better than the current $p$. We then expand $v$, adding its neighbours to the "reached nodes" from the current direction $R_{cur}$ and the priority queue $Q_{cur}$, pruning as appropriate using the current incumbent path length. Finally, we swap the search and proceed in the other direction (assuming $opp$ is not already exhausted). Note that the search never expands beyond a CPD

node, i.e., only non-CPD nodes can generate successors.

The loop terminates when both of the A* searches exhaust. We say that an A* search exhausts if either the queue becomes empty or the top node $v$ has $f$-value at least equal to $|p|$. When the loop terminates, we unpack $p$ to obtain the complete path and return it. First, the Up-Down path in the contraction hierarchy is recovered from $p$ by using the predecessor for each node (recorded either during the two A* searches or the CPD path extraction as discussed shortly). Then, the shortcuts in this Up-Down path are unpacked to obtain the optimal path in the original graph. Next, we discuss pruning to speed up the algorithm.

**Pruning:** Recall, at line 8, we compute $d(v, v')$ for every $v' \in V_{opp}$ using the CPD. This involves recursively obtaining first moves to identify a complete path from $v$ to $v'$. In some cases, we can avoid computing $d(v, v')$ by checking if $g(cur, v) + landmark(v, v') + g(v', opp) \geq |p|$. For each pair $v, v'$ that cannot be pruned in this way, we employ a caching scheme that can reduce unnecessary first move extractions. Specifically, whenever we extract a first move $u$ on the shortest path from $v$ to $v'$, we also record $g(cur, u)$ which corresponds to the shortest distance from $cur$ to $u$ seen so far. We also maintain the predecessor of $u$ which helps in path recovery as discussed earlier. Later, when extracting a path from any $v \in V_{cur}$ to any $v' \in V_{opp}$, we can terminate the recursion early if we reach a node $u$ for which $g(cur, u) < g(cur, v) + d(v, u)$. This is because we already have explored a path to $u$ which is shorter than the current path to $u$ via $v$. Notice that our caching strategy maintains only tentative distances. As such it requires no additional memory overheads beyond what is typically allocated for bi-directional search. Though simple, this approach substantially improves the performance of CH-CPD search.

**Theorem 1.** *Algorithm 1 returns an optimal path.*

*Proof.* (Sketch) Clearly Algorithm 1 examines all the paths that either meet, or connect via a pair of CPD vertices $(v_s, v_t)$. But avoids exploring the vertices that have the $f$-values bigger than the cost of current shortest path $|p|$ (thus can never be part of optimal path), and vertices pairs $(v_s, v_t)$ of CPD where $d(s, v_s) + d(v_s, v_t) + d(v_t, t) \geq |p|$. □

**Example 5.** *Consider the example in Figure 4 and assume that* D *to* J *are CPD nodes (shown in red) and* C *is a landmark (shown in blue). The A* search from start* A *first expands the CPD node* D*. Then, the A* search from target* B *expands the CPD node* I*. CPD is used to extract the path from* I *to* D *and the distances from* B *to each node on the path are cached. The optimal path $p$ is updated to be $\langle$B, I, D, A$\rangle$ with length 11. A* search from* A *prunes the node* C *using the landmark heuristic because $g(A, C)+landmark(C, B) = 10 + 6 = 16 > 11$. This A* search exhausts. The A* search from the target expands* F*. It needs to extract a path from* F *to* D *using the CPD. First moves are extracted and when the node* J *is reached, the path extraction stops. This is because the cached distance $g(B, J) = 5$ is smaller than $g(B, F) + d(F, J) = 6$. A* search from the target is also exhausted. The path $\langle$B, I, D, A$\rangle$ is unpacked and returned.*

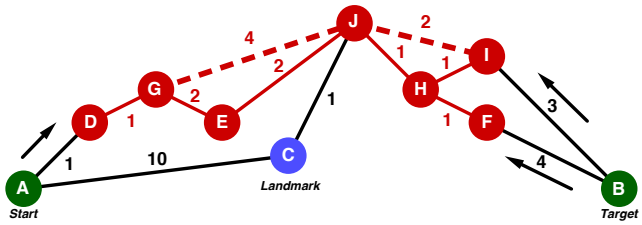


Figure 4: A* search from start (resp. target) expands D (resp. I and F). C is pruned by landmark heuristic.

## Experiments

We test our proposed algorithms against baseline implementations of our two main ingredients, CH and CPD, and against other state-of-the-art methods from the recent literature. By **CPD** we refer to Compressed Path Databases, as represented by SRC (Strasser, Harabor, and Botea 2014) and implemented by original authors.[3] **CH** refers to Contraction Hierarchies (Geisberger et al. 2008, 2012), as implemented in RoutingKit.[4] **CH+L** (CH + Landmarks) is a customised variant similar to CH where we replace bidirectional Dijkstra search with bidirectional A* and Landmark heuristics. We show CH+L performs significantly better than CH. Our approach meanwhile is denoted **CH-CPD (x%)** where $x\%$ means that a CPD is constructed for top $x\%$ of the nodes in the contraction hierarchy. Both CH-CPD and CH+L use 4 landmarks for distance estimates (adding more did not improve performance).

For further comparison we also consider two recent hub-labeling algorithms, PHL and SHP, as described in (Li et al. 2017) and using implementations from those authors[5]. **PHL** (Pruned Highway labeling) (Akiba et al. 2014) is a popular and efficient hub labeling method for shortest distance queries on road networks. While most hub labeling algorithms use nodes as hubs, PHL differs mainly in that it uses highway paths as hubs and the distances are maintained to these highways. **SHP** (Significant path based Hub Pushing) (Li et al. 2017) is another state-of-the-art hub labeling approach for road networks. It employs ideas similar to PHL but considers the vertices on "significant paths" as hubs. Such vertices are simply ranked by the multiplication of vertex degree and descendant size difference. To efficiently recover shortest paths, both PHL and SHP store predecessor nodes along with each hub label as suggested in (Li et al. 2017). While this increases the index size, it significantly speeds up the shortest path recovery time.

**Queries:** For experiments, we consider a variety of road networks taken from the 9th DIMACS challenge.[6] We generate queries as suggested in (Wu et al. 2012): each road network is discretised into a 1024 × 1024 grid with cell side length $l$. We randomly generate ten groups of queries such that $i$-th group contains 1000 $(s, t)$ pairs with Euclidean distance between them within $2^{i-1} \times l$ to $2^i \times l$, thus

[3] https://bitbucket.org/dharabor/pathfinding
[4] https://github.com/RoutingKit/RoutingKit
[5] http://degroup.cis.umac.mo/sspexp
[6] http://users.diag.uniroma1.it/challenge9/

| Type | Name | #V | #E | Build Time (Mins) | | | | | | | | | Memory (MB) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | CH-CPD | | | | | Competitors | | | | CH-CPD | | | | | Competitors | | | |
| | | | | 20% | 40% | 60% | 80% | 100% | CPD | CH | PHL | SHP | 20% | 40% | 60% | 80% | 100% | CPD | CH | PHL | SHP |
| Distance | NY | 264k | 733k | 0.361 | 0.732 | 1.177 | 1.867 | 2.945 | 8.763 | **0.238** | 0.601 | 0.441 | 70 | 104 | 183 | 271 | 338 | 219 | **29** | 411 | 449 |
| | BAY | 321k | 800k | 0.226 | 0.470 | 0.862 | 1.728 | 3.131 | 13.365 | **0.139** | 0.348 | 0.307 | 60 | 80 | 132 | 183 | 213 | 144 | **29** | 302 | 359 |
| | COL | 435k | 1057k | 0.331 | 0.793 | 2.206 | 4.381 | 7.460 | 23.072 | **0.186** | 0.705 | 0.589 | 86 | 155 | 217 | 268 | 337 | 239 | **38** | 495 | 593 |
| | FLA | 1070k | 2712k | 3.114 | 8.408 | 19.777 | 36.005 | 56.241 | 148.572 | **0.435** | 1.850 | 1.713 | 189 | 328 | 542 | 755 | 984 | 692 | **97** | 1325 | 1586 |
| | NW | 1207k | 2840k | 2.004 | 10.535 | 25.104 | 46.132 | 73.629 | 204.249 | **0.433** | 2.621 | 2.548 | 205 | 424 | 665 | 868 | 1100 | 818 | **100** | 1515 | 2008 |
| | NE | 1524k | 3897k | 5.275 | 24.142 | 56.436 | 108.295 | 167.397 | 342.805 | **1.179** | 7.892 | 7.049 | 436 | 836 | 1438 | 2160 | 2708 | 1998 | **149** | 3453 | 4434 |
| Travel Time | NY | 264k | 733k | 0.274 | 0.964 | 1.789 | 2.241 | 3.000 | 11.027 | **0.157** | 0.184 | 0.169 | 63 | 88 | 156 | 222 | 277 | 188 | **28** | 161 | 198 |
| | BAY | 321k | 800k | 0.152 | 0.303 | 0.609 | 1.096 | 1.831 | 13.843 | **0.090** | 0.117 | 0.141 | 50 | 63 | 102 | 137 | 157 | 106 | **28** | 116 | 180 |
| | COL | 435k | 1057k | 0.207 | 0.496 | 1.259 | 2.511 | 4.243 | 25.378 | **0.103** | 0.193 | 0.219 | 67 | 117 | 158 | 193 | 241 | 174 | **36** | 180 | 250 |
| | FLA | 1070k | 2712k | 2.405 | 5.100 | 10.880 | 21.383 | 33.971 | 166.378 | **0.289** | 0.709 | 0.751 | 152 | 261 | 405 | 567 | 706 | 514 | **92** | 526 | 727 |
| | NW | 1207k | 2840k | 1.201 | 5.551 | 13.804 | 26.454 | 42.560 | 259.305 | **0.319** | 0.714 | 0.860 | 168 | 346 | 509 | 631 | 789 | 586 | **98** | 568 | 835 |
| | NE | 1524k | 3897k | 3.211 | 12.595 | 27.293 | 52.753 | 85.102 | 436.746 | **0.587** | 1.436 | 1.510 | 325 | 602 | 1037 | 1599 | 2028 | 1597 | **141** | 995 | 1350 |

Table 3: Number of vertices (#V) and edges (#E) in maps, build time in Mins, and memory in MB for CH-CPD and competitors.

| Type | Name | Stat | CH-CPD (20%) | | | | CH-CPD (40%) | | | | CH-CPD (60%) | | | | CH-CPD (80%) | | | | CH-CPD (100%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | #C | 0% | 0.25% | 0.5% | 1% | 0% | 0.25% | 0.5% | 1% | 0% | 0.25% | 0.5% | 1% | 0% | 0.25% | 0.5% | 1% | 0% | 0.25% | 0.5% | 1% |
| Distance | NY | #CE | 0 | 13.481 | 12.833 | 12.603 | 0 | 12.353 | 12.238 | 12.196 | 0 | 12.702 | 12.623 | 12.653 | 0 | 13.220 | 13.201 | 13.240 | 0 | 12.446 | 12.451 | 12.487 |
| | | TC | 0 | 0.002 | 0.005 | 0.008 | 0 | 0.009 | 0.018 | 0.029 | 0 | 0.018 | 0.030 | 0.063 | 0 | 0.031 | 0.059 | 0.123 | 0 | 0.050 | 0.093 | 0.185 |
| | | TT | 0.384 | 0.121 | 0.123 | 0.120 | 1.506 | 0.512 | 0.494 | 0.556 | 3.188 | 0.939 | 0.939 | 0.999 | 5.963 | 1.570 | 1.629 | 1.707 | 9.912 | 2.561 | 2.707 | 2.812 |
| | NE | #CE | 0 | 15.642 | 15.219 | 15.092 | 0 | 17.898 | 17.699 | 17.687 | 0 | 16.581 | 16.550 | 16.594 | 0 | 16.271 | 16.260 | 16.309 | 0 | 16.279 | 16.278 | 16.319 |
| | | TC | 0 | 0.057 | 0.118 | 0.219 | 0 | 0.276 | 0.399 | 0.654 | 0 | 0.492 | 0.931 | 1.642 | 0 | 0.893 | 1.770 | 3.328 | 0 | 1.327 | 2.791 | 5.065 |
| | | TT | 11.457 | 4.393 | 4.096 | 4.416 | 49.095 | 26.081 | 22.963 | 24.443 | 125.745 | 57.157 | 55.256 | 56.816 | 255.027 | 98.660 | 107.115 | 99.024 | 382.009 | 150.067 | 166.218 | 148.021 |
| Travel Time | NY | #CE | 0 | 4.967 | 5.050 | 4.969 | 0 | 6.043 | 6.005 | 6.004 | 0 | 5.884 | 5.871 | 5.877 | 0 | 5.384 | 5.381 | 5.385 | 0 | 5.258 | 5.259 | 5.263 |
| | | TC | 0 | 0.003 | 0.005 | 0.008 | 0 | 0.009 | 0.015 | 0.031 | 0 | 0.020 | 0.036 | 0.075 | 0 | 0.035 | 0.066 | 0.125 | 0 | 0.052 | 0.105 | 0.207 |
| | | TT | 0.392 | 0.115 | 0.117 | 0.129 | 1.615 | 0.801 | 0.806 | 0.780 | 3.993 | 1.562 | 1.632 | 1.670 | 7.435 | 1.977 | 2.084 | 2.223 | 12.335 | 2.676 | 2.843 | 2.907 |
| | NE | #CE | 0 | 6.035 | 5.999 | 5.986 | 0 | 5.857 | 5.850 | 5.758 | 0 | 5.613 | 5.545 | 5.545 | 0 | 5.607 | 5.553 | 5.555 | 0 | 5.577 | 5.576 | 5.573 |
| | | TC | 0 | 0.064 | 0.129 | 0.269 | 0 | 0.298 | 0.506 | 0.791 | 0 | 0.563 | 1.053 | 1.940 | 0 | 1.002 | 1.968 | 3.738 | 0 | 1.713 | 3.019 | 6.202 |
| | | TT | 13.433 | 2.374 | 2.624 | 2.516 | 60.091 | 12.010 | 12.007 | 12.027 | 155.598 | 24.821 | 26.705 | 28.022 | 306.740 | 48.411 | 52.166 | 52.877 | 488.504 | 81.968 | 84.515 | 82.648 |

Table 4: The average number of cache nodes expanded (#CE), the time in minutes for building cache (TC) and total building time (TT) for constructing CH-CPD. We show the number of cached nodes (#C) in range of 0 to 1% of the total nodes.

10,000 queries in total. In discussions, we distinguish between *path queries*, which asks for a shortest (uncontracted) path from start to target, and *distance queries*, which ask only for the length. Individual queries are run 10 times; we omit the best and worst run and average all the rest.

All algorithms are implemented in C++ and compiled with -O3 flag. We use a 32 cores Nectar research cloud with 64GB of RAM and Ubuntu 18.04 LTS (Bionic) amd64.

## Preprocessing Cost and Space

We first compare various methods for shortest path query processing in terms of the preprocessing time required and the size of the data structures required to support the method. Table 3 compares the CH-CPD approaches where the CPD is produced on the top 20%, 40%, 60%, 80%, or all the nodes versus other techniques. Note that the costs for CH-CPD include all the costs for constructing and storing the contraction hierarchy. We make use of the distance tables enhancement introduced earlier to speed up the CH-CPD preprocessing by caching the top 0.5% of the nodes.

We use road maps from the DIMACS challenge using either the distance weights or the travel time weights (which we will see are surprisingly different). Unsurprisingly, the contraction hierarchy is the cheapest approach to both compute and store. Both PHL and SHP are more expensive to compute, but only by a few factors. The CPD approaches are the most expensive to compute; what is interesting is that the CPD on the original graph is *more expensive* to compute than on the contraction hierarchy, since for the CH-CPD we can restrict to Up-Down paths and make use of caching in the preprocessing. The full CH-CPD is around 50% larger than the CPD on the original graph even though it is cheaper to compute. Partial CH-CPDs are significantly cheaper to both compute and store than the full CH-CPD, reaching the same ballpark compute times as PHL and SHP at 20%. What is somewhat surprising is that the full CH-CPD storage costs are significantly smaller than both PHL and SHP on the Distance maps, while in the Travel Time maps they tend to be larger than PHL and around the same size as SHP.

Overall, we see that the storage requirements for CH-CPDs are not overwhelming, and indeed can be smaller than the competitors. The use of partial CH-CPDs means we can trade off storage requirements with query time.

Table 4 shows some results about the effectiveness of the

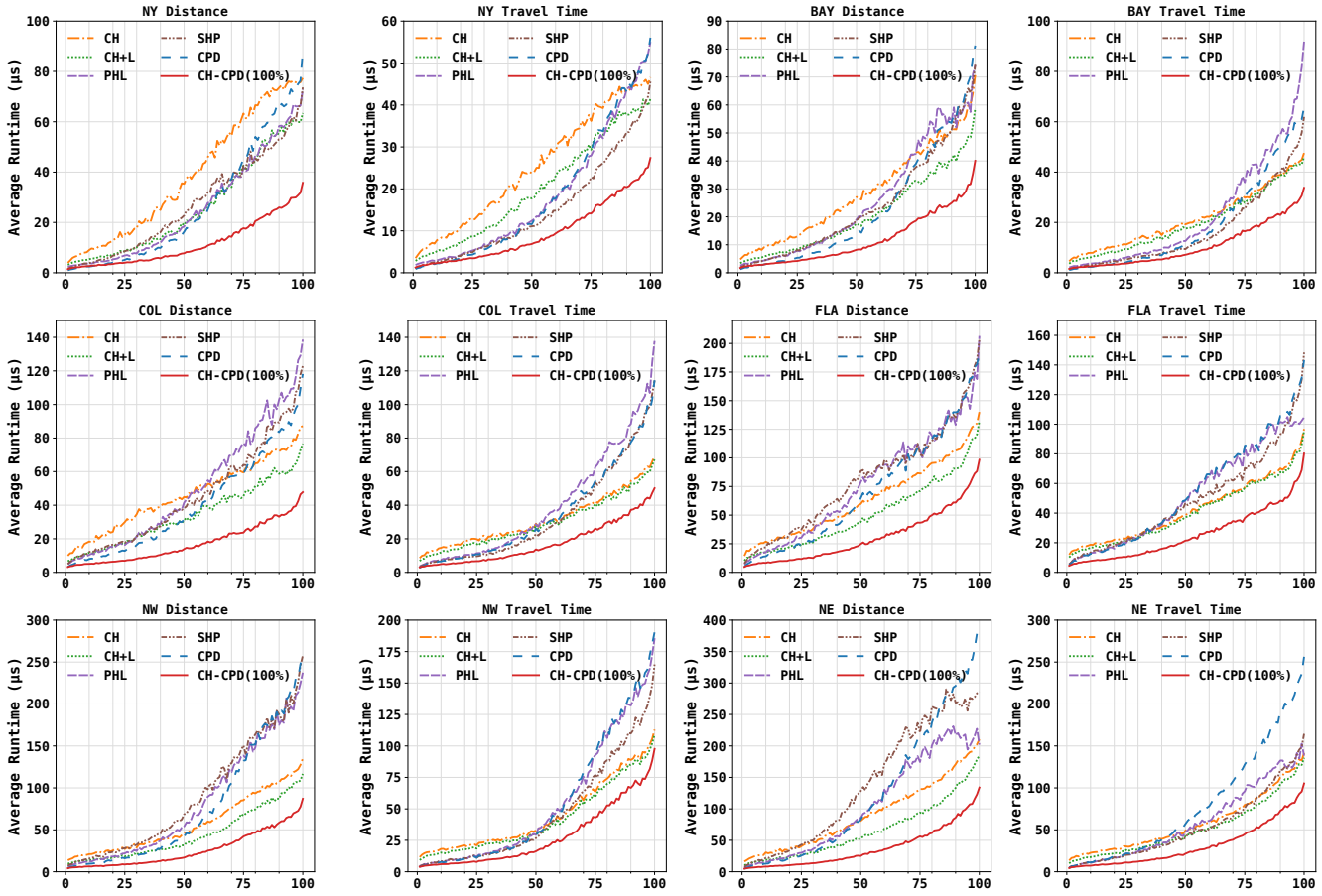


Figure 5: Runtime comparisons on the six road network Distance and Travel Time graphs. The x-axis shows the percentile ranks of path queries sorted based on actual distances between start and target.

caching preprocessing enhancement. The table shows the average number of cached nodes expanded when building CH-CPDs for various proportions of the contraction hierarchy, as well as the time to build the cached data, and the total build time to build the (partial) CH-CPD, for the smallest and largest map (not including the CH building time as in Table 3). We vary the number of cached nodes to be 0% (no distance tables enhancement), and 0.25%, 0.5% and 1% of the total number of nodes in the map. Clearly the caching preprocessing enhancement is highly effective, reducing construction times to one third, and by more on larger maps. The percentage of cached nodes does not make that much difference, as clearly the number of expanded cached nodes hardly changes. The results on the other maps are similar. Note that this approach can be adapted to improve the preprocessing of traditional CPDs as well. Although not reported here, we observed improvement by a few factors for constructing traditional CPDs by caching the distances on the same top-n% of nodes. But the choice of which nodes to cache is important to the success of the approach. With randomly chosen cache nodes, we observed a slowdown as it incurs a large number of distance updates in the order of $|CE| \times |V|$. Our optimisation is not limited by certain ordering and we believe that other intuitive lexical orderings (e.g., nodes having high "reach") may further improve preprocessing of traditional CPDs.

## Query Processing Time

Next we compare path query processing times of the various methods. Figure 5 shows cactus plots for each competing algorithm on 12 different graphs. Note that for each CH-based method the path query time includes path unpacking. Examining the results we can see that in terms of worst case performance contraction hierarchies are particularly important. The top three methods in the larger maps are CH-CPD, CH+L and CH. Other methods can be faster for some queries, since they avoid path unpacking, but by the time we reach 100% of queries solved, these methods dominate. Interestingly, Landmarks are significantly beneficial for CH on the Distance maps while making almost no difference on the Travel Time maps. Basic CPD is very good for queries with short paths but becomes less competitive as more extractions are required. CH-CPD is faster than all other approaches on all maps; the combination of no search together with few lookups finds optimal paths very quickly.

In Table 5, we extend the comparison to consider short-

| Type | Name | Distance Queries | | | | | | | | | Path Queries | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | CH-CPD | | | | | Competitors | | | | CH-CPD | | | | | Competitors | | | | |
| | | 20% | 40% | 60% | 80% | 100% | CH | CH+L | PHL | SHP | 20% | 40% | 60% | 80% | 100% | CPD | CH | CH+L | PHL | SHP |
| Distance | NY | 12.826 | 10.992 | 7.568 | 5.323 | 3.230 | 28.330 | 15.643 | **0.919** | 1.013 | 23.165 | 20.831 | 17.186 | 13.667 | **11.419** | 26.375 | 38.641 | 25.581 | 25.359 | 26.759 |
| | BAY | 10.176 | 9.006 | 6.692 | 4.496 | 2.792 | 17.992 | 11.592 | **0.692** | 0.722 | 19.775 | 18.900 | 16.674 | 13.751 | **11.784** | 22.852 | 28.686 | 20.850 | 25.876 | 23.895 |
| | COL | 12.958 | 10.183 | 8.428 | 5.803 | 3.640 | 29.426 | 17.277 | 0.940 | **0.873** | 29.060 | 26.332 | 23.371 | 20.183 | **17.014** | 39.751 | 45.998 | 33.960 | 49.809 | 45.146 |
| | FLA | 18.909 | 15.655 | 11.686 | 7.911 | 5.020 | 31.099 | 21.501 | 1.187 | **1.038** | 49.122 | 44.655 | 39.606 | 34.627 | **30.619** | 70.860 | 63.634 | 50.677 | 76.544 | 80.532 |
| | NW | 17.527 | 14.867 | 11.408 | 7.681 | 4.991 | 30.898 | 19.445 | **0.980** | 1.052 | 41.866 | 38.819 | 35.052 | 29.844 | **26.280** | 76.620 | 57.423 | 44.283 | 81.401 | 89.203 |
| | NE | 25.971 | 19.060 | 14.462 | 8.910 | 6.436 | 53.536 | 30.971 | 1.820 | **1.762** | 62.335 | 51.880 | 47.270 | 40.313 | **37.697** | 124.367 | 90.845 | 65.569 | 105.197 | 138.857 |
| Travel Time | NY | 10.525 | 9.772 | 6.810 | 4.720 | 2.551 | 16.527 | 11.832 | **0.553** | 0.565 | 19.158 | 18.818 | 14.830 | 12.845 | **9.526** | 18.234 | 25.265 | 20.064 | 18.315 | 14.926 |
| | BAY | 9.147 | 7.785 | 5.772 | 3.750 | 2.306 | 12.232 | 10.547 | 0.499 | **0.485** | 19.058 | 17.360 | 15.106 | 12.540 | **10.611** | 19.147 | 21.636 | 19.970 | 22.861 | 16.786 |
| | COL | 10.282 | 8.556 | 6.047 | 4.888 | 3.008 | 14.631 | 12.500 | 0.540 | **0.505** | 26.221 | 24.483 | 20.850 | 19.967 | **17.261** | 34.758 | 31.544 | 29.430 | 39.852 | 33.473 |
| | FLA | 14.311 | 12.518 | 10.148 | 6.777 | 4.187 | 17.430 | 15.327 | 0.592 | **0.578** | 38.402 | 35.681 | 32.733 | 29.112 | **25.523** | 54.153 | 42.209 | 40.444 | 51.897 | 49.206 |
| | NW | 15.828 | 13.337 | 10.289 | 7.247 | 4.192 | 18.788 | 16.581 | 0.569 | **0.550** | 41.388 | 39.318 | 36.031 | 32.223 | **27.949** | 55.378 | 44.970 | 41.713 | 54.710 | 46.612 |
| | NE | 19.594 | 15.389 | 11.328 | 7.631 | 5.506 | 27.098 | 21.652 | 0.698 | **0.646** | 49.708 | 43.221 | 37.278 | 32.087 | **31.222** | 79.938 | 58.037 | 51.549 | 59.335 | 53.164 |

Table 5: Running time comparison for distance and path queries, we report average time (µs) between CH-CPD and competitors.

| Type | | Distance | | | | | | | | Travel Time | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Stat | CH-CPD | | | | | Competitors | | | CH-CPD | | | | | Competitors | | |
| | | 20% | 40% | 60% | 80% | 100% | CPD | CH | CH+L | 20% | 40% | 60% | 80% | 100% | CPD | CH | CH+L |
| NY | #Generated | 62.630 | 52.101 | 18.985 | 6.978 | - | - | 301.529 | 123.441 | 57.173 | 45.269 | 20.219 | 7.464 | - | - | 161.916 | 93.249 |
| | #Expanded | 22.049 | 17.855 | 7.415 | 3.009 | - | - | 101.302 | 41.469 | 24.079 | 18.549 | 8.809 | 3.563 | - | - | 71.815 | 40.289 |
| | CPD Usage | 68.41% | 74.97% | 90.57% | 95.50% | 100% | 100% | - | - | 66.31% | 74.66% | 89.87% | 95.09% | 100% | 100% | - | - |
| | $\|V_s\|$ | 3.304 | 2.585 | 2.136 | 1.384 | 1 | 1 | - | - | 2.512 | 2.087 | 1.999 | 1.320 | 1 | 1 | - | - |
| | $\|V_t\|$ | 3.290 | 2.567 | 2.088 | 1.393 | 1 | 1 | - | - | 2.481 | 2.062 | 1.973 | 1.342 | 1 | 1 | - | - |
| | #Path | 2.910 | 1.960 | 1.539 | 1.147 | 1 | 1 | - | - | 2.051 | 1.652 | 1.577 | 1.173 | 1 | 1 | - | - |
| | #FirstMove | 25.530 | 18.555 | 15.464 | 11.163 | 9.592 | 191.742 | - | - | 15.330 | 14.364 | 15.042 | 11.517 | 9.947 | 168.589 | - | - |
| NE | #Generated | 87.953 | 51.536 | 21.333 | 1.158 | - | - | 503.155 | 208.474 | 73.962 | 42.742 | 19.217 | 1.154 | - | - | 197.155 | 121.689 |
| | #Expanded | 29.722 | 18.135 | 7.843 | 0.628 | - | - | 149.765 | 62.799 | 32.072 | 19.020 | 8.714 | 0.637 | - | - | 89.644 | 54.048 |
| | CPD Usage | 81.49% | 89.97% | 96.93% | 99.98% | 100% | 100% | - | - | 77.89% | 89.45% | 96.24% | 99.98% | 100% | 100% | - | - |
| | $\|V_s\|$ | 4.317 | 3.169 | 2.282 | 1.209 | 1 | 1 | - | - | 2.740 | 2.353 | 1.878 | 1.219 | 1 | 1 | - | - |
| | $\|V_t\|$ | 4.374 | 3.187 | 2.240 | 1.230 | 1 | 1 | - | - | 2.761 | 2.351 | 1.879 | 1.232 | 1 | 1 | - | - |
| | #Path | 4.001 | 2.438 | 1.647 | 1.090 | 1 | 1 | - | - | 2.288 | 1.841 | 1.480 | 1.119 | 1 | 1 | - | - |
| | #FirstMove | 41.307 | 26.506 | 19.670 | 13.548 | 12.426 | 607.931 | - | - | 20.560 | 19.469 | 17.757 | 14.442 | 12.964 | 520.366 | - | - |

Table 6: Average number of nodes #Generated and #Expanded by each algorithm. CPD Usage corresponds to % of queries for which both A* searches expand at least one CPD node (and hence end up using CPD). For queries that use CPD, we report average $|V_s|$ (resp. $|V_t|$) that denote # of CPD nodes expanded from $s$ (resp. $t$), and average #Path and #Firstmove extractions.

est distance queries and also include the partial CH-CPDs. Unsurprisingly the distance based methods PHL and SHP are far superior for distance queries, since the other methods essentially find the shortest path and then calculate its length (although they can avoid path unpacking). We can see that the partial CH-CPDs roughly double the query time when moving to a 20% CH-CPD, where the query times are roughly still slightly ahead of CH+L and still significantly better than PHL and SHP for path retrieval.

Table 6 provides more insights. CH+L performs significantly better than CH due to the smaller number of nodes generated and expanded using the landmark heuristic. As shown by CPD Usage, CH-CPD does not always need to use the CPD (when the optimal path does not pass through CPD nodes). For such queries, CH-CPD essentially is the same as CH+L. Note that the numbers of CPD nodes expanded from start ($|V_s|$) and target ($|V_t|$) are pretty small. Furthermore, the number of paths extracted using CPD is also significantly smaller than $|V_s| \times |V_t|$ which shows the effectiveness of our pruning rules that also help significantly reduce the number of first move extractions. Compared with the CPD on the original graph, the full CH-CPD requires a significantly smaller number of first move extractions which explains its superior performance.

## Conclusion

We show how to use Compressed Path Databases and Contraction Hierarchies to generate the fastest shortest path query retrieval method we are aware of. The use of Contraction Hierarchies also allows us to cache information for the CPD construction that actually makes CPD construction significantly faster. We also show how we can tradeoff preprocessing time and space with path retrieval time by building partial CPDs. While path retrieval now requires search it is still highly competitive with other methods.

## Acknowledgements

Research at Monash University was partially supported by the Australian Research Council under grants FT180100140, DP180103411, DP190100013, DP200100025, and also by a gift from Amazon.

## References

Akiba, T.; Iwata, Y.; Kawarabayashi, K.; and Kawata, Y. 2014. Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA, January 5, 2014*, 147–154. SIAM.

Alonso-Mora, J.; Samaranayake, S.; Wallar, A.; Frazzoli, E.; and Rus, D. 2017. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proceedings of the National Academy of Sciences* 114(3): 462–467.

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, 122-127*. The AAAI Press.

Delling, D.; Goldberg, A. V.; Pajor, T.; and Werneck, R. F. 2017. Customizable route planning in road networks. *Transportation Science* 51(2): 566–591.

Dibbelt, J.; Strasser, B.; and Wagner, D. 2016. Customizable Contraction Hierarchies. *ACM J. Exp. Algorithmics* 21(1): 1.5:1–1.5:49.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038, 319–333. Springer.

Geisberger, R.; Sanders, P.; Schultes, D.; and Vetter, C. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transp. Sci.* 46(3): 388–404.

Goldberg, A. V.; and Harrelson, C. 2005. Computing the shortest path: *A** search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, 156–165. SIAM.

Harabor, D. D.; and Stuckey, P. J. 2018. Forward Search in Contraction Hierarchies. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 55–62.

Holte, R.; Holte, R. C.; Perez, M.; Perez, M. B.; Zimmer, R. M.; Zimmer, R. M.; MacDonald, A.; and Macdonald, A. J. 1996. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *In Proceedings of the National Conference on Artificial Intelligence*, 530–535.

Li, Y.; U, L. H.; Yiu, M. L.; and Kou, N. M. 2017. An Experimental Study on Hub Labeling based Shortest Path Algorithms. *Proc. VLDB Endow.* 11(4): 445–457.

Luxen, D.; and Sanders, P. 2011. Hierarchy decomposition for faster user equilibria on road networks. In *International Symposium on Experimental Algorithms*, 242–253. Springer.

Shen, B.; Cheema, M. A.; Harabor, D.; and Stuckey, P. J. 2020. Euclidean Pathfinding with Compressed Path Databases. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 4229–4235. ijcai.org.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Barták, R.; and Boyarski, E. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, 151–159. AAAI Press.

Strasser, B.; Botea, A.; and Harabor, D. 2015. Compressing Optimal Paths with Run Length Encoding. *Journal of Artificial Intelligence Research* 54: 593–629.

Strasser, B.; Harabor, D.; and Botea, A. 2014. Fast First-Move Queries through Run-Length Encoding. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*. AAAI Press.

Sturtevant, N. R.; Felner, A.; Barer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-Based Heuristics for Explicit State Spaces. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 609–614.

Sturtevant, N. R.; Traish, J. M.; Tulip, J. R.; Uras, T.; Koenig, S.; Strasser, B.; Botea, A.; Harabor, D.; and Rabin, S. 2015. The Grid-based Path Planning Competition: 2014 Entries and Results. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 241–251.

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. *Proc. VLDB Endow.* 5(5): 406–417.