Improving Time-dependent Contraction Hierarchies

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

Computing time-optimal shortest paths, in road networks, is one of the most popular applications of Artificial Intelligence. This problem is tricky to solve because road congestion affects travel times. The state-of-the-art in this area is an algorithm called Time-dependent Contraction Hierarchies (TCH). Although fast and optimal, TCH still suffers from two main drawbacks: (1) the usual query process uses bi-directional Dijsktra search to find the shortest path, which can be time-consuming; and (2) the TCH is constructed w.r.t. the entire time domain T, which complicates the search process for queries q that start and finish in a smaller time period $T_q \subset T$. In this work, we improve TCH by making use of time-independent heuristics, which speed up optimal search, and by computing TCHs for different subsets of the time domain, which further reduces the size of the search space. We give a full description of these methods and discuss their optimality-preserving characteristics. We report significant query time improvements against a baseline implementation of TCH.

Introduction

Recent years have seen substantial progress for computing shortest paths in static road network. Leading algorithms in this area, such as Contraction Hierarchies (CHs) (Dibbelt, Strasser, and Wagner 2016) and CH-based Compressed Path Databases (Shen et al. 2021), can solve a given query in just micro seconds. However static networks can only roughly approximate actual road conditions, since traffic congestion (among other issues) can affect travel times throughout the day. How to compute more accurate solutions in these settings has become an important topic, and is a necessary enabling technology for reliable route planning software (Delling et al. 2017), such as Bing Maps and Google Maps. One way to address this problem is to model the road network in a time-dependent way, where the cost of each edge is determined by a piecewise linear function that changes depending on the time of day. Because the time-dependent model can take into account expected traffic congestion, the computed shortest paths more closely approximate actual road conditions.

The state-of-art algorithm in this area is Time-dependent Contraction Hierarchies (Batz et al. 2013, 2009) : a family of successful speed up techniques that embeds the road network into a hierarchical graph. There are however two drawbacks: (i) TCH inherits the bidirectional Dijkstra search from static Contraction Hierarchies (Dibbelt, Strasser, and Wagner 2016). This approach does not rely on any lower-bounding heuristics for guidance, although such methods are known to improve performance. (ii) In time-dependent road networks, TCH is built by considering the entire time domain T, in order to answer all queries $q \in T$. However, each individual query only corresponds to a trip within a limited time period T_q, such that $T_q \subset T$. Embedding the travel time metric for the entire time domain T can increase the size of TCH search space, which again affects query performance.

In this work, we investigate how to improve the TCH algorithm. First, we adapt two admissible heuristic functions from the static network literature: landmarks (Goldberg and Harrelson 2005) and Path Databases (Bono et al. 2019). We show that both approaches can be easily integrated with TCH and both can substantially improve performance. Our second approach involves building a set of smaller TCHs, each of which focuses on a subset of the time domain. By choosing the appropriate TCH for each query $q \in T$ we can retain the optimality guarantees of the original algorithm while substantially improving search performance.

We give a complete description of the new algorithms, and evaluate them on a range of road networks, including real-world as well as synthetic datasets. Results show substantial improvement over the baseline TCH method.

Preliminaries

Let $G = (V, E, F, T)$ be a directed graph, with nodes V, edges $E \subseteq V \times V$ and $f \in F$ maps each edge $e \in E$ to a Travel Time Function (TTF) which returns the non-negative travel time $f(t)$ needed to travel through the edge e for a given specific start time t in the time domain T. Each directed edge $e_{v_i v_j} \in E$ with its corresponding TTF $f_{v_i v_j}$ represents the edge that connects node v_i to $v_i j$. In a road network, we naturally assume that the network G satisfies the FIFO property (i.e., $f_{v_i v_j}(t') + t' \geq f_{v_i v_j}(t) + t \mid \forall e_{v_i v_j} \in E$ and $\forall t' > t \in T$), that is departing later or waiting at an intermediate node cannot result in arriving earlier. Similar to many existing works (Batz et al. 2009, 2013), we model

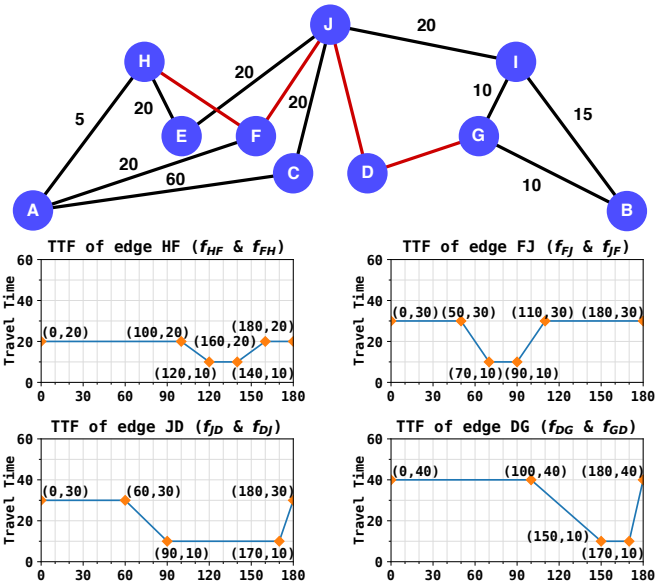

Figure 1: An example of an undirected time-dependent graph. TTFs of the red edges are shown below the graph, and the travel cost of the other edges are constant.

the TTF as a continuous piece-wise linear function with the time domain of 24 hours. Figure 1 shows an example of a time-dependent network. For exposition only, we assume the graph G is undirected and only the edges that are highlighted in red have non-constant TTF with $T = [0, 180)$. Next, we explain some of the important operations of TTF.

Evaluation of $f(t) \in F$ normally requires a binary search over the sorted array of interpolate points, and runs in $O(log(|f|))$ where $|f|$ denotes the number of interpolate points of f. However, we use the bucket-implementation (Batz et al. 2013), which evaluates $f(t)$ by scanning only the interpolate points inside the bucket corresponding to t.

Chaining computes the TTF of a path $e_{v_iv_j} \rightarrow e_{v_jv_k}$ as $f_{v_iv_k}(t) = f_{v_jv_k}(t + f_{v_iv_j}(t))$. We use $f_{v_iv_k} = f_{v_iv_j} \circ f_{v_jv_k}$ to denote the chaining. Since concatenating two piece-wise linear functions can only result in a piece-wise linear function, the operation can be computed in linear time (i.e., $O(|f_{v_iv_j}| + |f_{v_jv_k}|)$). The resultant function $f_{v_iv_k}$ has the number of interpolate points $|f_{v_iv_k}| \leq |f_{v_iv_j}| + |f_{v_jv_k}|$ with the lower-bound $min(f_{v_iv_k}) \geq min(f_{v_iv_j}) + min(f_{v_jv_k})$.

Merging minimizes the TTFs $f'_{v_iv_j}$ and $f''_{v_iv_j}$ on two parallel edges of $e_{v_iv_j}$ while preserving all the shortest paths in G. The operation $f_{v_iv_j} = min(f'_{v_iv_j}, f''_{v_iv_j})$ is defined as $f_{v_iv_j}(t) = min\{f'_{v_iv_j}(t), f''_{v_iv_j}(t)\} \mid \forall t \in T$. Similar to chaining, this operation also runs in $O(|f'_{v_iv_j}| + |f''_{v_iv_j}|)$ and results in a piece-wise linear TTF.

A *path* $\mathcal{P}$ from source (s) and destination (d) is a sequences of nodes $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_k \rangle$, where $k \in \mathbb{N}^+$, $v_0 = s$, $v_k = d$, and $e_{v_iv_{i+1}} \in E$ for $0 \leq i < k$. In time-dependent road network, the length (travel cost) of path $|\mathcal{P}|$ depends on the departure time $t \in T$ and $|\mathcal{P}| = f_{v_0v_k}(t)$, where $f_{v_0v_k}(t) = f_{v_0v_1} \circ f_{v_1v_2} \ldots \circ f_{v_{k-1}v_k}$. Given a start time t, $sp(s, d, t)$ denotes the *shortest path* from s to d.

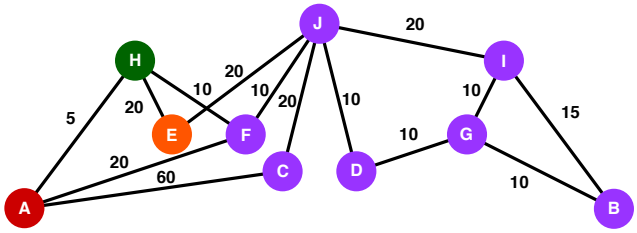

Figure 2: From the source node H, the first move on the optimal path to any node are A (red), E(orange) and F (purple).

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

Table 1: First moves for H, A and J for the example of Fig. 2

Compressed Path Database (CPD): is the state-of-the-art approach (Botea 2011) for extracting shortest path in static road networks (i.e., the cost of each edge is a constant value). In a CPD, for each $s \in V$, a row $R(s)$ is computed which stores, for every $d \in V$, the first move (i.e., the first vertex) on the shortest path from s to d. Each row $R(s)$ is compressed using run-length encoding (RLE), thus extracting first move $fm[s, d]$ only requires a simple binary search through the compressed RLE encoded string. The shortest path from s to any d (i.e., CPD(s, d)) can then be obtained by recursively finding the first moves $fm[s, d]$.

Example 1. *Consider the graph shown in Figure 2. Table 1 shows the first move rows for nodes H, A and J, where "*" is a wildcard (i.e., don't care) symbol that indicates s and d are the same, thus never need to lookup this symbol. The RLE compresses a string of symbols into representative sub-strings, each sub-strings has two values: a start index and a first move symbol, e.g., the compressed RLE strings for H, A and J are [(1,A), (3,F), (10,E)], [(1,H)] and [(1,F), (5,I), (6,D), (9,C), (10,E)], respectively. Note that the symbol "*" can be combined with any other symbols, and the effectiveness of RLE compression depends on the way nodes in each row are ordered, e.g., the order of columns is a Depth First Search order (from H) as suggested by Strasser, Harabor, and Botea (2014).*

Reversed Path Database (RPD): stores, for each source node $s \in V$, a reverse row $RR(s)$ which records, for every $d \in V$, the first move on the shortest path from d to s (Mahéo et al. 2021). Unlike CPD, the compression on $RR(s)$ is not effective. Therefore, RPDs are not compressed. This allows accessing the first move in $O(1)$. The shortest path from s to d can be efficiently obtained by recursively obtaining the $fm[s, d]$ using $RR(d)$, i.e., the shortest path can be extracted using a single row. Note that both CPD and RPD runs Dijkstra search to compute $R(s)$ or $RR(s)$ on each source node s, and can be paralleled linear in number of processors.

The CPD and RPD are state-of-the-art heuristics in dynamic environment settings too (Bono et al. 2019). Similarly, given a time-dependent graph G, we build a CPD or RPD by considering the minimal cost on each edge of G. Since the optimal path extracted from database is no larger than the shortest path on G (i.e., $|\text{CPD}(s,d)| \leq |sp(s,d,t)|$ for $\forall(s,d) \in G$ and $\forall t \in T$), the first-move oracle defines an *admissible* heuristic.

Example 2. *Figure 2 shows an example, where we construct a CPD on a graph with minimal edge weight shown in Figure 1. The shortest path extracted from CPD between* H *to* J *is a valid lower bound, i.e.,* $(|CPD(\texttt{H},\texttt{J})| = 20) \leq (\{sp(\texttt{H},\texttt{J},t) \mid \forall t \in T\} = [30,40])$.

Landmarks: are another method for generating admissible heuristics in time-dependent road network (Delling and Wagner 2007). For each landmark $l \in L$, we use the minimum travel time of each edge to compute an array that records a pair $(d(l,v), d(v,l))$ on each $v \in V$, where $d(l,v)$ denotes the shortest travel time from l to v. Due to the triangle inequality, the array of cost is exploited to be the lower-bound of $sp(v_i,v_j,t)$ for $\forall t \in T$, from any v_i to any v_j: $landmark(v_i,v_j) = \max_{l \in L}\{max(d(v_i,l) - d(v_j,l), d(l,v_i) - d(l,v_j))\}$. The effectiveness of lower-bound depends on the distribution of landmarks, thus we select L on the borders of the graph following the same procedure explained in (Sturtevant et al. 2009).

Time-Dependent Contraction Hierarchy (TCH): is a speedup technique that exploits the hierarchical nature of the real-world transportation networks (Batz et al. 2009, 2013). Given a graph G, a TCH can be built by repeatedly applying a contraction operation to $v \in V$. In broad strokes:

1. Apply a total lex order $\mathcal{L}$ to the nodes V of G.
2. W.r.t. $\mathcal{L}$, choose the least node $v \in V$ that has not been previously contracted.
3. (Contraction) Add to G a shortcut edge e_{uw} between *each* pair of in-neighbour u and out-neighbour w of v for which: 1) the lex order u and w are larger than v; and 2) $\langle u,v,w\rangle$ is the shortest path between u and w at *some time* in T. When adding the shortcut edge e_{uw}, the TTF is computed as $f_{uw} = f_{uv} \circ f_{vw}$. However, the parallel edges can exist, and in this case, we merge existing TTF f'_{uw} as $f_{uw} = min(f'_{uw}, f_{uv} \circ f_{vw})$ and maintain a *middle node profile* to track the intermediate node for corresponding interval in TTF.

Fewer shortcuts improve query performance, but computing a lex order $\mathcal{L}$ that minimizes the number shortcuts is NP-hard (Bauer et al. 2010). Thus, we use the heuristic order suggested in (Batz et al. 2013). Note that the contraction operation requires verifying local optimality and can be costly in time-dependent scenario, therefore the steps 2-3 are parallelized. We refer the reader to the paper (Batz et al. 2013) for more details of the parallelization.

Example 3. *In Figure 3, we contract the time-dependent graph shown in Figure 1 in alphabetical lex order. The TTF of the shortcut edge* $e_{\texttt{HJ}}$ *is computed as* $f_{\texttt{HJ}} = min(f_{\texttt{HE}} \circ f_{\texttt{EJ}}, f_{\texttt{HF}} \circ f_{\texttt{FJ}})$ *and the corresponding middle node profile is* $\{(0,\texttt{E}),(40,\texttt{F}),(80,\texttt{E})\}$ *which indicates the middle nodes are* E, F *and* E *for the time period* $[0,40),[40,80)$ *and* $[80,180)$, *respectively. Also note, without shortcut edges (dashed), the optimal path from* A *to* B *at time 150 has 6 edges:* $\langle \texttt{A},\texttt{H},\texttt{E},\texttt{J},\texttt{D},\texttt{G},\texttt{B}\rangle$. *However, with shortcut edges, we traverse only four edges:* $\langle \texttt{A},\texttt{H},\texttt{J},\texttt{G},\texttt{B}\rangle$.

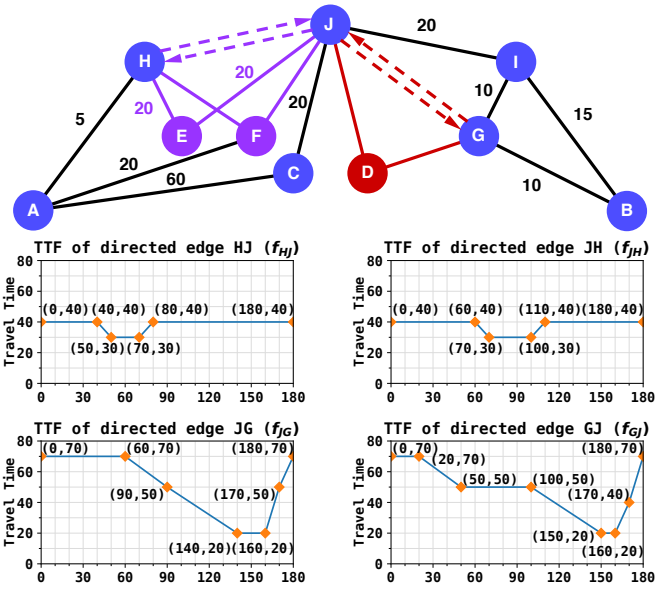

Figure 3: We show the result for contracting nodes E and F in purple, and D in red. Dashed edge are the shortcut edges and their corresponding TTFs are shown in the figure below.

Recall that TCH adds a shortcut edge e_{uw} iff both u and w are lexically larger than the intermediate node v, and e_{uw} is optimal and equivalent to the path $\langle u,v,w\rangle$. Therefore, for every pair of edges (e_{uv}, e_{vw}), there must exist a cost equivalent *up edge* $e_{uw} \uparrow$ (i.e, $u <_{\mathcal{L}} w$) or *down edge* $e_{uw} \downarrow$ (i.e, $u >_{\mathcal{L}} w$), if $\langle u,v,w\rangle$ is $sp(u,w,t)$ for $t \in T$. Thus, we have the following:

Lemma 1. ***(tch-path)****: For every optimal path* $sp(s,d,t)$ *in* G, *there is a cost equivalent tch-path* $\langle s,\ldots k\ldots d\rangle$ *whose prefix* $\langle s,\ldots k\rangle$ *is an up path (i.e.,* $s <_{\mathcal{L}} s+1 <_{\mathcal{L}} k$ *and suffix* $\langle k\ldots d\rangle$ *is a down path (i.e.,* $k >_{\mathcal{L}} k+1 >_{\mathcal{L}} d$) □

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

The key idea of TCH is that the shortcut edges can bypass one or more intermediate nodes in a single step. To achieve a speedup, authors in (Batz et al. 2013) develop the Bidirectional TCH search (BTCH) to efficiently find the tch-path, following Lemma 1. To support the bi-directional search, BTCH divides the set of edges E into two as follows:

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

Given a source-destination pair (s,d) and departure time t, the main challenge of the bidirectional search is that the

backward search is prohibitive in time-dependent scenario (i.e., we can not search backward without knowing the arrival time t'). Therefore, BTCH runs in two phases:

(i) **Bi-directional Search:** In the forward direction, BTCH runs a time-dependent Dijkstra search from s, considering only the outgoing edges in $E_{\uparrow}$. Similar to the static graph, this Dijkstra search differs mainly in that it considers the TTFs when it generates successors. In the reverse direction, BTCH performs the backward exploration from d using a static Dijkstra search, and considers only the lower-bound edge weights of the incoming edges in $E_{\downarrow}$. We also maintain U (initially infinity) which is the smallest upper bound distance of any path from s to d seen so far by the algorithm. During the backward Dijkstra search, we store each traversed edge in E_{trv} and compute the upper bound by taking the maximal value on each edge in $E_{\downarrow}$. Whenever the two searches meet at an apex node k, we: (i) update U if the upper bound distance from s to d via k is smaller than current U; and (ii) obtain a lower-bound $L(s, k, d)$ for the path from s to d via k. If $L(s, k, d) \geq U$, we discontinue the search on both sides from k. Every time the search meets, we also record the apex node k in K. The bi-directional search continues until the minimum f-value on both open lists are no less than U (or when both lists are empty, if no such path).

(ii) **Forward Search:** Once the bi-directional search stops, for each recorded apex node $k \in K$ and $L(s, k, d) \leq U$, we continue a forward time-dependent Dijkstra search on the edges E_{trv} by iteratively inserting k into the queue. This forward search is funnelled into d and considers only the edges that are previously traversed by the backward search. The search terminates if d is expanded.

Finally, BTCH unpacks the tch-path using the *middle node profile* maintained on the corresponding shortcut edges, and the optimal path $sp(s, d, t)$ is returned. BTCH remains state-of-the-art for time-dependent routing (Strasser, Wagner, and Zeitz 2021).

Improving Search on TCH

Given a time dependent graph G, we construct the Time-dependent Contraction Hierarchy. For a given pair of source and destination and its departure time t, BTCH is a successful and efficient methods to solve the pathfinding problem. However, we show the search algorithm can be further improved (i) by combining the BTCH with landmark heuristics; (ii) by changing the search to a forward search and combining with a more advanced CPD-based heuristic: TCPD heuristic; and (iii) by splitting the time domain T into multiple T' (i.e., $T' \subseteq T$), such that each query in T' can be efficiently solved by TCH constructed for T' only.

Combining BTCH with Landmarks

In time-dependent road networks, landmarks have been widely used in many existing techniques, such as bi-directional A* search (Nannicini et al. 2012), core routing (Delling and Nannicini 2012) and SHARC routing (Delling 2011). Following the success of these algorithms, we extend BTCH using landmarks:

(i) During the *Bi-directional Search*: We employ the bi-directional A* search instead of bi-directional Dijkstra search, and the search on each side is guided by the landmark heuristic, i.e., the f-value for a node v is $g(s, v) + landmark(v, d)$ with $g(s, v) \geq sp(s, v, t)$ being a tentative upper-bound for the shortest path from s to v. We also apply the pruning rule used in the BTCH and mark the edges that traversed by the backward search as E_{trv}. The search terminates when the minimum f-value for both A* searches are larger or equal to the smallest upper-bound U so far.

(ii) During the *Forward Search*: Recall that the backward search in the bi-directional phase records the edges that have been traversed as E_{trv}. We argue that the lower-bound $L(v, d)$ computed on each node v can be directly used as a heuristic, because $L(v, d) \leq sp(v, d, t)$ for $\forall t \in T$. $L(v, d)$ is also more effective than the landmarks as this lower-bound computes the true minimum distance in $E_{\downarrow}$ (i.e., $L(v, d) \geq landmark(v, d)$).

Combining the landmarks heuristic with bi-directional search speeds up the search in both directions, and results in a smaller subset E_{trv} compared with the Dijkstra search. Although the forward search is funnelled into the direction of d and is usually cheap to run, directly reusing the previously computed lower-bound further improves the query performance as we show later in the experiments section.

Forward TCH Search with CPD-based Heuristic

Due to the fact that the travel cost on a TTF can not be computed without a known departure time t, forward search seems to be a natural way to solve the pathfinding problem in a time-dependent road network. This motivates us to revise the search on TCH to be a forward search. In addition to landmarks, we consider a more sophisticated goal-directed heuristic, called the TCPD heuristic, and we show this heuristic (i) requires less first move extractions; and (ii) provides a more effective lower-bound compared to the CPD heuristic. To further improve the query performance, we also propose several pruning rules and optimizations.

Forward TCH Search (F-TCH): For a given source s and destination d, suppose we expand a search node n with its predecessor $p(n)$, a time-dependent Dijkstra search on the TCH typically generate successors $s(n)$, which fall into one of the following types:

1. Up-Up successors: $p(n) <_{\mathcal{L}} n$ and $n <_{\mathcal{L}} s(n)$;
2. Up-Down successors: $p(n) <_{\mathcal{L}} n$ and $n >_{\mathcal{L}} s(n)$;
3. Down-Down successors: $p(n) >_{\mathcal{L}} n$ and $n >_{\mathcal{L}} s(n)$;
4. Down-Up successors: $p(n) >_{\mathcal{L}} n$ and $n <_{\mathcal{L}} s(n)$;

Given a departure time t, although this uni-directional Dijkstra search finds the shortest path $sp(s, d, t)$, the search is unlikely to be efficient without avoiding the non tch-paths. Therefore, we modify the search to consider only the tch-paths by a simple pruning rule called UTD (Up-Then-Down) (Harabor and Stuckey 2018). Recall that a tch-path is always an UP-Down path (i.e., Lemma 1) with an apex node which is lexically larger than all the other nodes on the path. The F-TCH needs to generate (i) the type 1 and 3 successors, which covers the Up or Down tch-paths (i.e., s or d is the apex node) by continuously moving up or down; (ii) the type 2 successors, which covers the Up-Down tch-paths

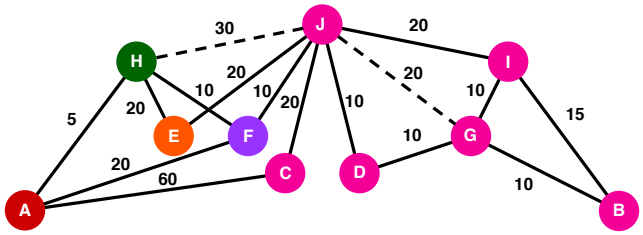

Figure 4: From the source node H, the optimal first move to any node are A (red), E(orange), F (purple) and J (pink)

(i.e., an intermediate node k is an apex node and $k >_{\mathcal{L}} v$ for $v \in \langle s, \dots k-1\rangle \mid \langle k+1, \dots d\rangle$) by switching the direction at the apex node. Therefore, the only type of successors that is disallowed by UTD is the Down-Up successors which leads to a non tch-path.

TCH-CPD (TCPD) heuristic: Given a time-dependent graph G, we construct a CPD using the minimal edge cost of each edge, and use this CPD in a forward TCH search as a heuristic to further improve the search. However, path extraction for CPD requires a number of lookups (i.e.,$fm[s,d]$) equal to the number of edges on the optimal path, thus can be costly when used as a heuristic. To mitigate such defects, we propose to combine the CPD with TCH. Our idea is motivated by a recent paper (Shen et al. 2021) that shows combining CPD with contraction hierarchies in a static graph reduces the first move extractions significantly. However, combining CPD with TCH is more complicated.

Given a TCH constructed on a time-dependent graph G, we obtain a contracted graph by taking the lower-bound value of each TTF. In order to build CPD, we further modify the F-TCH to compute the first moves on the optimal tch-path, from each source node s toward $\forall d \in V$. For a given source node s, we divide the search node v with its parents $p(v)$ into two types: (i) Up-reach search node v_u: $p(v_u) <_{\mathcal{L}} v_u$. (ii) Down-reach search node v_d: $p(v_d) >_{\mathcal{L}} v_d$. At each search node v, we allow the search to independently expand each type of search nodes once with $g(s, v_u)$ and $g(s, v_d)$ being the best tentative distance respectively. The search terminates when the open list becomes empty, and for each node v, we have (i) $g(s, v_u)$ which computes the shortest up tch-paths; and (ii) $g(s, v_d)$ which covers all the shortest up-down or down tch-paths. Therefore, the first move from s to any v on the optimal tch-paths can be easily obtained from the path $min(g(s, v_u), g(s, v_d))$.

Example 4. *In Figure 4, the contracted graph is taken from the TCH in Figure 3. Assume we compute the first move row on the source node* A*. A simple F-TCH would falsely prune the successor* J *when expanding search node* F *because the best tentative solution from* A *to* F *is* $\langle$A, H, F$\rangle$ *and the successor* J *is a down-up successor (i.e.,* $H >_{\mathcal{L}} F <_{\mathcal{L}} J$*). However, the modified F-TCH computes the optimal path* $\langle$A, F, J$\rangle$ *correctly as it manages up and down search nodes separately, i.e., the down-up successor pruning is only performed when expanding a down-reachable search node. Also note the first moves from source node* H *are now on the optimal tch-path.*

With the modified F-TCH search, we now construct the TCPD following the same general procedures already described. Given a source and destination, we denote the path extracted from TCPD as TCPD(s, d).

Theorem 1. *Given a pair of* (s, d) *and departure time* $t \in T$ *in a TCH,* $|TCPD(s, d)|$ *is a lower-bound for* $|sp(s, d, t)|$.

Proof. Given a TCH constructed in G, there must exists a cost equivalent tch-path for every $sp(s, d, t)$ and $t \in T$. In a lower-bound graph of TCH, TCPD computes and encode the first move on the optimal path, from any s to any d by examining all possible (i) up; (ii) up-down; and (iii) down tch-path. Thus $|\text{TCPD}(s, d)|$ defines a lower-bound, i.e., $|\text{TCPD}(s, d)| \leq |sp(s, d, t)|$ for $\forall t \in T$. □

Due to Theorem 1, $|\text{TCPD}(s, d)|$ defines an admissible heuristic and can be easily combined with forward TCH search. Recall that the lower-bound on a shortcut edge $min(f_{uw}) \geq min(f_{uv}) + min(f_{vw})$, thus $|\text{TCPD}(s, d)|$ is a tighter lower bound, i.e., $|\text{TCPD}(s, d)| \geq |\text{CPD}(s, d)|$. Next we introduce several optimization techniques to further improve the query performance.

Downward Successors Pruning: Although the F-TCH prunes the up-successors, when expanding the node n, it has to generate every down-successor v no matter whether $p(n) >_{\mathcal{L}} n$ or $p(n) <_{\mathcal{L}} n$. However, not every down-successor v can lead to a path $\langle n, \dots d\rangle$ that reaches the destination d from the apex node n. Therefore, we reuse the concept of CPD and propose a *reachabilty oracle*, called $Reach$, that tells whether there is a down path $\langle s, \dots d\rangle$, from a given node s to any node d. Whenever the F-TCH generates the down-successors n, we use $Reach(n, d)$ to prune non-reachable successors. Eliminating the down-successors early helps the search to reduce the branching factor. Next, we discuss the construction of $Reach$.

To build $Reach$, we first compute a down-reachability table (DT) by running a Depth First Search (DFS) on each source node s that only generates the down successors (i.e., $s(n) >_{\mathcal{L}} n$). Although DT is a simple truth table, without a better ordering, it does not compress well. But note that the TCH is a hierarchical data structure that is rooted from one or more important nodes, and these nodes are assigned to the largest lex order possible. Therefore, we run a DFS from the node that has the largest lex order. Similarly, the search only visits the down-successors, not earlier expanded by a node with larger lex order. We order the columns of DT according to the order of nodes accessed by this DFS, and compress it using RLE. With this novel ordering, $Reach$ can be compressed effectively (e.g., in a graph with millions of nodes, $Reach$ only needs a few MB to store), and each binary search $Reach(s, d)$ runs in near constant time.

Cost Caching: When TCPD is used as a heuristic, it needs to continuously extract the path TCPD(s, d) at each node expansion of s. To further reduce the number of $fm[s, d]$, we apply a simple cost caching strategy. When we extract a path from s to d, we cache distance $dist(v, d) = |\text{TCPD}(v, d)|$ for each vertex v on the extracted path TCPD(s, d). Later

on, for a subsequent path extraction TCPD(s', d), we terminate early if the path extraction reaches the vertex v such that $dist(v, d)$ is cached, and we use the cached distance to get the path length from s' to d. Similar cost caching strategies have also appeared in CPD-based search (Shen et al. 2020), and CPD-heuristics (Bono et al. 2019), since they significantly reduce the first move extractions.

Reverse TCH Path Database: Another way to improve the path extraction of TCPD(s, d) is to speed up each $fm[s, d]$ by building a reversed tch path database (RTPD) that is similar to the RPD as discussed earlier. Recall that a RPD computes a reverse first move table RR that records the first move on optimal path from any s to a given target node d. Similarly, on each source node s, we compute $RR(s)$ in TCH using a modified F-TCH which considers only the incoming edges of TCH. RTPD and TCPD essentially compute the same first moves, but only store in a different way, thus all the properties of TCPD preserved.

Splitting the Time-domain

In order to handle any arbitrary query that is issued at $t \in T$, a TCH is usually built by considering the entire time domain T. However, each individual query only requires us to consider a much shorter time period $T_q = [t, t']$, where t' is the optimal arrival time when traveling from s to d at t. Therefore, the TCH has two drawbacks: (i) In each edge, the TTF stores all interpolate points w.r.t T, thus requires more time to evaluate the travel time for t, and results in a looser lower and upper bounds than T_q. (ii) For each shortcut edge e_{uw}, e_{uw} is added if $\exists T' \subseteq T$ such that $\langle u, v, w\rangle$ is an optimal path from u to w during T', such a shortcut edge may be unnecessary for T_q. To mitigate these disadvantages of TCH, we propose to split T into n number of equal-sized time buckets T_i (i.e., $T_i = [t_i, t_{i+1}] = |T|/n$ for $0 \leq i \leq n-1$) for processing start times $t \in [t_i, t_{i+1})$. For each bucket T_i, we build a TCH to cover all $|sp(s, d, t)| \leq t_u$ by adding an upper-bound t_u, and building TCH$_i(T)$ over time range $[t_i, t_{i+1} + t_u]$. First, we show how to choose the upper-bound t_u to form a single-layer TCH, denoted as STCH, such that $\exists$ TCH$_i \in$ STCH and TCH$_i(T) \supseteq T_q$ for all queries $q \in T$. We then describe a multi-layer TCH which combines STCHs with customized t_u. For each individual query $q \in T$, we show q can be improved using TCH built in STCH with minimal t_u.

Single-layer TCH In the time-dependent shortest path problem, highly accurate solutions are needed, especially when a user needs to plan a short-term trip. Therefore, our main focus is on city-sized road networks. From point to point in a city-sized graph, the shortest path $sp(s, d, t)$ does not typically take a large amount of time (e.g., travel within Melbourne almost always requires less than four hours). Suppose $|sp(s, d, t)| < t_u$ for every possible start s, destination d and start time t. We build a TCH for each start time bucket $T_i = [t_i, t_{i+1}]$ which only considers travel in the time $[t_i, t_{i+1}+t_u]$. Then we can answer a shortest path query $sp(s, d, t)$ correctly by using TCH$_i$ built for time bucket T_i where $t \in T_i$ if travel time is no more than t_u.

Example 5. *Consider the example in Figure 3. Assume we split the time domain $T = [0, 180]$ into 6 time buckets, i.e., $|T_i| = 30$ for $0 \leq i \leq 5$. Since $\forall sp(s, d, t) \leq 90$ for $t \in T$ and $(s, d) \in G$, we set the upper bound $t_u = 90$. Notice that contracting the node D for T_0 (i.e., $T_0 = [0, 120]$) cannot add the shortcut e_{JG}, as $\langle J, I, G\rangle$ is a shorter path than $\langle J, D, G\rangle$ (i.e., $20 + 10 \leq min(f_{JD} \circ f_{DG})$).*

Multi-layer TCH Although we can predict an upper-bound t_u that is large enough for solving all queries, the upper-bound t_u may not be efficient, because many, if not most travel time of queries $q \in T$ may be much smaller than t_u. In order to solve q using a TCH such that TCH(T) is as small as possible, we propose to form multiple STCHs with different t_u into a Multi-layer TCH (MTCH). In the *offline* phase, we build MTCH as following:

1. At the root R of MTCH, we construct a TCH w.r.t the entire time domain T. In addition, we build a TCPD as discussed earlier.
2. For each lower layer j of R, we build STCH$_j$ by splitting T into n time buckets T_i, and the upper-bound t_u can be set to any customized value, but only needs to be less than t'_u, where t'_u is the upper-bound used in STCH$_{j-1}$.

During the *online* phase of MTCH, given a query $sp(s, d, t)$ with $t \in T_i$, we first obtain an upper-bound $U(s, d)$ using the maximal travel cost for the path extracted from TCPD between s and d. Therefore, we know the query can be solved in $T_q = [t, t + U(s, d)]$. For TCHs built in time bucket T_i, we check each layer of MTCH via a top-down scan from STCH$_0$ to STCH$_j$. The scan terminates when TCH$_i(T) \not\supseteq T_q$, and returns TCH$_i$ with minimal TCH$_i(T)$ that covers T_q. If TCH$_i$ is found, we answer the query using the TCH$_i$. Otherwise, we answer the query by running a forward TCH search in R, with TCPD guided as a heuristic. MTCH is guaranteed to solve all queries $q \in T$ in R, and each $q \in T_i$ is safely improved by using the TCHs maintained in lower layers.

Experiments

We test our proposed algorithms against the baseline implementation of Time-dependent Contraction Hierarchies (**TCH**), taken from the repository[1] of original authors (Batz et al. 2013). The implementation is also known as KaTCH. In a recent study (Strasser, Wagner, and Zeitz 2021), it was shown that KaTCH is the state-of-the-art algorithm and outperforms a range of optimal algorithms, including CATCHUp (Strasser, Wagner, and Zeitz 2021), TD-CALT (Delling and Nannicini 2012), and TD-SHARC (Delling 2011). Meanwhile, our approaches: (i) the Single-layer TCH (**STCH**) is built by splitting the time domain into 24 hourly buckets and the upper-bound t_u is set to 4 hours as no trip in our tested maps exceeds this limit. (ii) The Multi-layer TCH (**MTCH**) is built by adding three layers of STCH under the root R. For each layer, we maintain 24 hourly time buckets and set t_u to 4h, 2h and 1h correspondingly.

[1] https://github.com/GVeitBatz/KaTCH

Map	#V	#E	Build Time (Mins)								Memory (MB)							
			TCH				STCH		MTCH		TCH				STCH		MTCH	
			-	CPD	TCPD	RTPD	-	TCPD	-	TCPD	-	CPD	TCPD	RTPD	-	TCPD	-	TCPD
NY	96k	260k	1.72	2.92	2.94	3.21	2.99	31.87	9.04	95.12	269	346	353	9596	1279	3286	3193	9198
GC	39k	81k	0.03	0.18	0.17	0.18	0.25	3.56	2.51	12.46	32	36	36	1614	231	334	829	1138
SYD	192k	421k	0.37	5.17	4.99	6.19	1.88	111.99	11.01	341.03	214	290	294	37168	1421	3254	3711	9147
MEL	314k	690k	0.62	15.55	14.65	17.93	3.50	342.74	24.52	1030.67	417	587	609	99237	2710	7132	6815	19927
NY-5	96k	260k	30.76	31.94	31.96	32.30	22.15	50.68	37.18	122.48	4242	4319	4325	13569	12254	14262	23289	29297

Table 2: Total number of vertices (#V) and edges (#E) in maps, build time in Mins, and memory in MB for TCH-based data structure with different heuristics.

For the heuristics, we further compare our approaches with Compressed Path Databases (Strasser, Harabor, and Botea 2014) and take the implementation from the publicly available repository.[2] By **CPD**, we mean the Compressed Path Databases that is built on the contracted graph of TCH. On the other hand, our approach **L12** indicates using 12 landmarks for travel time estimation (we varied the number of landmarks from 4 to 16, and 12 appears to be the best). **TCPD** and **RTPD** refer to TCH-based Compressed Path Databases and Reverse TCH Path Database respectively. We also use the letter **B** and **F** to denote the Bidirectional search and Forward search respectively. For example, **B-TCH(L12)** denotes our algorithm bidirectional TCH search with landmarks heuristic, and **F-TCH(L12)** denotes the forward TCH search with landmarks heuristic, while applying all optimizations introduced.

Dataset: For experiments, we consider the real-world dataset taken from the public repository.[3] The dataset contains the road network for New York (NY) and the historical travel time that is estimated every hour during the entire 2013 year. In order to compute TTF for each edge, we take the travel time data from Tuesday to Thursday following (Batz et al. 2013) and average them for each hour after filtering out the data by two standard deviation. Overall, the NY dataset has 12.59% of edges that are time-dependent with time domain $[0h, 24h)$. Since there are not many time-dependent datasets available online, we also create a few synthetic datasets to evaluate our algorithms: (i) In order to simulate the data on other cities, we take the road networks for Gold Coast (GC), Sydney (SYD), and Melbourne (MEL) from the OpenStreetMap[4] and use the traffic pattern taken from NY dataset, to assign each type of road the same percentage of time-dependent edges. (ii) In order to simulate a more accurate TTF, we change the TTF of NY dataset by taking the 5 mins data points on a cubic spline created using the original data. We denote this dataset as NY-5.

Queries: We generate queries following the same method as in (Wu et al. 2012). For each road network, we discretise the map into a 1024×1024 grid with cell side length l. Then, we randomly generate ten groups of queries such that i-th group contains 1000 (s, d) pairs with Euclidean distance between them within $2^{i-1} \times l$ to $2^i \times l$, i.e., 10,000 queries in total. During each hour in the time-domain $[0h, 24h)$, we report the performance for each algorithm to determine the length of the shortest path, without outputting the complete description of the paths. Individual queries are run 10 times; we omit the best and worst runs and average the rest.

All algorithms are implemented in C++ and compiled with -O3 flag. We use a 2.6 GHz Intel Core i7 machine with 16GB of RAM and running OSX 10.14.6. For reproducibility, all of our implementations are available online.[5]

Preprocessing Cost and Space

All indexes were built on a 32 core Nectar research cloud with 128GB of RAM running Ubuntu 18.04.4 LTS (Bionic Beaver). Table 2 shows the build time and memory required for TCH, STCH and MTCH, as well as constructing heuristic CPD, TCPD and RTPD. Note that for STCH and MTCH, TCPDs are constructed on top of TCH in each time bucket T_i. All CPD-based heuristics include the costs for constructing and storing the underlying hierarchical data structure and the reachabilty oracle $Reach$ (for all datasets, $Reach$ takes $<$10 secs to build and $<$5MB to store).

For heuristics of the TCH, our proposed TCPDs are built in similar time to CPDs, but take slightly more space to store. On the other hand, RTPDs shows higher build time and space-consumption than both CPDs and TCPDs. Comparing with TCH, both STCH and MTCH are generally more expensive to build and store. However, interestingly, we see STCH is cheaper to compute than TCH in the NY-5 dataset. This is because the TTF of NY-5 has more interpolate points (i.e., every 5 mins) than the other datasets. During TCH construction, adding shortcut edges requires search to verify the local optimality which can adversely affect the performance when considering the entire time domain T. The memory of STCH and MTCH can be large after including the TCPDs heuristics. However, during the query time of entire day, STCH and MTCH roll over the TCHs built for each hourly bucket, so the actual memory required in RAM is only 1/24 of the memory reported in Table 2.

Query Processing Time

In this experiment, we compare the average query processing time for our algorithms against the baseline implementation of TCH. During the entire time domain [0h, 24h), we

[2]https://bitbucket.org/dharabor/pathfinding

[3]https://uofi.app.box.com/v/NYC-traffic-estimates

[4]https://www.openstreetmap.org

[5]https://github.com/bshen95/Improving-TCH

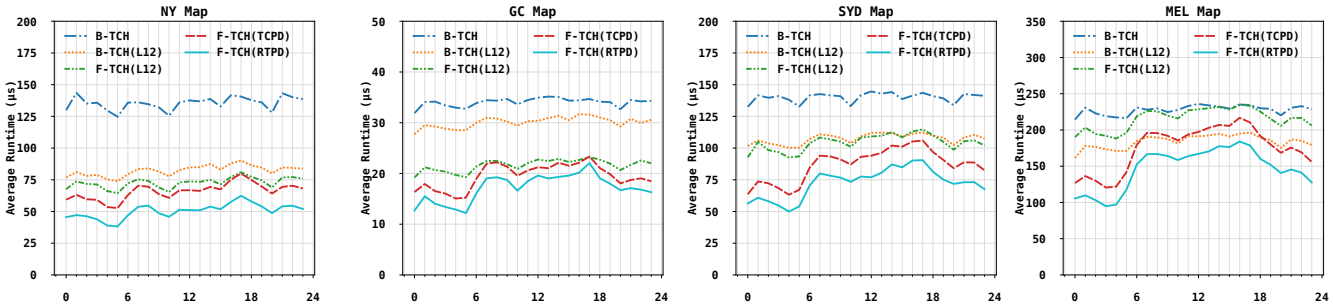

Figure 5: Runtime comparisons between TCH and different heuristic searches on TCH. The x-axis represents every hour during the time domain [0h, 24h). The y-axis shows the average runtime of different algorithms in μs.

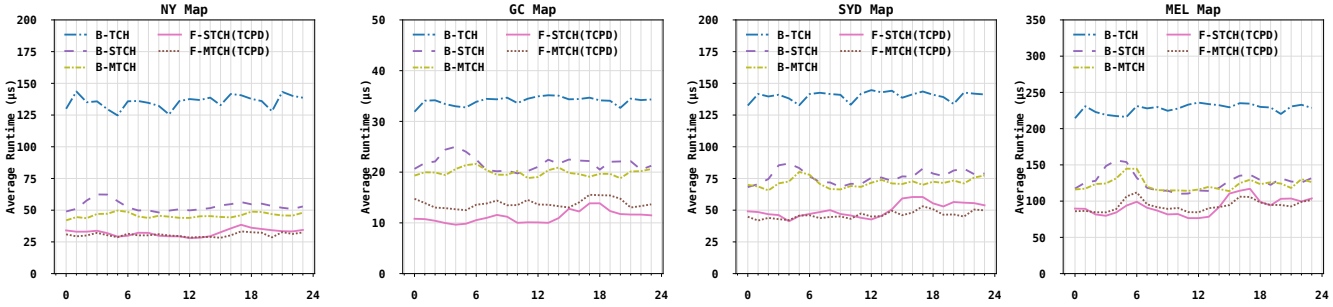

Figure 6: Runtime comparisons between the baseline implementation of TCH, and our algorithms STCH and MTCH with-/without TCPD heuristics. The x-axis and y-axis are same as in Figure 5.

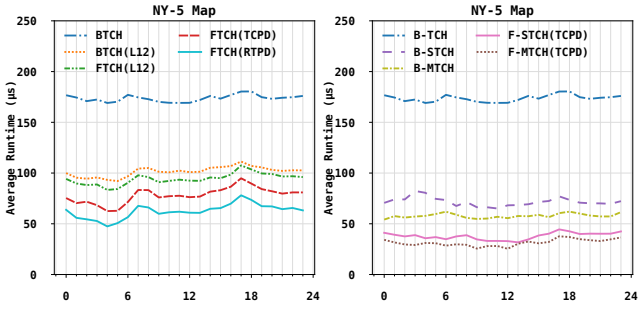

Figure 7: Runtime comparisons on NY-5 datasets, we show the results for different heuristics on TCH (left), STCH and MTCH with/without TCPD heuristics (right).

evaluate all queries by setting the departure time t at each hour. We report the average runtime of each algorithm.

Heuristic search: Figure 5 compares the query performance between the bidirectional TCH search (B-TCH) and our proposed heuristic searches on TCH. Examining the results, we see that all heuristic searches are substantially faster than the baseline implementation of TCH. For bidirectional TCH search, our proposed algorithm B-TCH(L12) improves B-TCH by around 30% of time. Moving to forward TCH search, the landmark heuristic F-TCH(L12) becomes faster than B-TCH(L12) for most datasets (e.g., NY, GC and SYD). In addition, the TCPD heuristic outperforms the landmark heuristics for forward TCH search in all datasets. Using RTPD heuristics dominates all methods. We also compared against CPD heuristics, the performance of F-TCH(CPD) is however significantly slower than other approaches (see Table 3).

Splitting the time domain: Figure 6 compares B-TCH against our algorithms. From the results, it is clear that B-STCH significantly outperforms B-TCH, and B-MTCH further improves the query performance. We also see that TCPD heuristics improve both algorithms, but F-MTCH(TCPD) costs additional time to switch to the corresponding TCH, thus only shows competitive results with F-STCH(TCPD). One can build RTPDs in STCH and MTCH by storing each of RTPD with same amount of space as shown in Table 2. However, the improvement is only a few microseconds, thus omitted. We also remark that the MTCH can be extended to speed up only certain period of the time domain (e.g., during the peak hours: 6 - 10 AM or 16 - 19 PM) or further improved by adding more layers with smaller t_u. However, how to efficiently choose upper-bound t_u and split the time domain is left as future work.

Tolerance of more accurate TTF: In Figure 7, we reproduce the experiments on NY-5 datasets, where the travel times are evaluated every 5 mins. Clearly, our proposed algorithms are tolerant to the more accurate TTF. In addition, unlike the NY dataset, both F-MTCH and F-MTCH(TCPD) outperform the STCH approaches. This is because the TCH constructed for $t_u = 1h$ and $2h$ in MTCH can be more efficiently evaluated than STCH (i.e., $t_u = 4h$).

Map	Stat	B-TCH		F-TCH				B-STCH	F-STCH	B-MTCH	F-MTCH
		-	L12	L12	CPD	TCPD	RTPD	-	TCPD	-	TCPD
NY	#Generated	409.030	238.549	194.885	146.910	128.333		231.202	73.821	191.762	60.718
	#Expanded	188.190	87.927	52.343	33.512	26.761		127.227	16.281	111.311	13.217
	#FirstMove	-	-	-	437.900	104.098		-	72.327	8.354	71.835
	#ReachTest	-	-	469.593	320.587	263.588		-	155.519	-	126.272
	Runtime	135.462	82.297	72.773	143.242	65.966	50.428	53.158	32.372	45.806	30.484
GC	#Generated	157.091	125.512	89.900	75.883	68.356		115.409	48.613	102.002	41.311
	#Expanded	98.240	72.178	43.035	33.852	28.400		78.561	19.883	71.528	16.536
	#FirstMove	-	-	-	287.394	61.682		-	49.488	10.170	54.765
	#ReachTest	-	-	265.317	202.391	168.863		-	113.855	-	91.703
	Runtime	34.048	30.106	21.637	36.205	19.509	17.188	21.747	11.182	19.940	13.822
SYD	#Generated	396.275	283.144	243.200	200.175	163.915		258.679	104.310	219.491	84.920
	#Expanded	203.671	123.877	81.389	60.884	43.175		151.042	28.513	134.567	22.991
	#FirstMove	-	-	-	857.854	130.283		-	98.494	11.558	97.879
	#ReachTest	-	-	823.346	624.071	447.878		-	274.051	-	218.711
	Runtime	140.094	107.453	104.462	235.888	86.982	72.609	76.488	50.362	71.604	46.152
MEL	#Generated	558.484	420.292	394.894	333.293	262.745		374.386	161.660	322.278	133.948
	#Expanded	293.440	189.117	139.651	108.987	73.811		219.413	43.616	196.001	36.044
	#FirstMove	-	-	-	1630.162	193.499		-	143.565	13.761	139.118
	#ReachTest	-	-	1592.497	1283.166	909.075		-	520.653	-	428.705
	Runtime	227.828	184.182	215.427	570.812	175.058	146.730	126.825	92.807	122.892	93.952
NY-5	#Generated	427.587	248.241	201.013	151.863	133.485		238.443	75.512	195.478	61.123
	#Expanded	191.516	88.954	51.879	33.236	26.944		129.906	16.548	112.625	13.321
	#FirstMove	-	-	-	354.327	106.424		-	73.479	8.428	72.187
	#ReachTest	-	-	475.553	325.475	270.539		-	157.568	-	126.871
	Runtime	173.661	101.496	94.273	148.735	78.185	62.465	71.651	37.887	57.855	31.426

Table 3: Average runtime in μs and number of nodes #Generated and #Expanded by each algorithm. For forward search, we report the number of reachability tests (#ReachTest) and #FirstMove extractions performed using $Reach$ and different CPDs.

Query statistics: Table 3 provides more insights. For the searches that are conducted on TCH, B-TCH(L12) significantly improves B-TCH due to the smaller number of nodes generated and expanded using landmarks heuristic. Although F-TCH(CPD) requires smaller number of nodes generated and expanded, it still performs worse than B-TCH as the number of first move extractions is large, even after applying the cost caching discussed earlier. On the other hand, F-TCH(TCPD) retrieves the first moves on the optimal tch-paths which significantly reduces #FirstMove. Also, #Generated and #Expanded are also reduced because TCPD computes tighter lower-bounds. Overall, F-TCH(TCPD) outperforms the other heuristics in terms of average runtime for all maps, and RTPD further improves the performance by extracting each first move in constant time. For the searches that are conducted on STCH and MTCH, both B-STCH and B-MTCH outperform B-TCH, because building the TCHs for small time period (i) has smaller number of shortcut edges; (ii) stores tighter lower and upper bound; and (iii) maintains much smaller number of interpolate points on each TTF. In addition, TCPD heuristic further improves the performance for STCH and MTCH, and achieves speed up against baseline algorithm B-TCH by 3-6 factors.

Conclusion and Future Work

We show how search in time-dependent road networks can be substantially improved by applying several heuristics. Forward search using heuristics based on landmarks or CPDs is able to improve upon the usual bi-directional search. We also show that we can improve TCHs by building a set of TCHs to be used depending on the start time, and a given upper bound on the shortest path. The resulting TCHs allow for faster query processing, using a heuristic search to quickly find an upper bound, and then choosing the appropriate TCH to answer the query.

Regarding bounded-suboptimal search, there exist many techniques that compute solutions faster at the cost of slightly suboptimal paths, such as TD-CRP (Baum et al. 2016), TD-CFLAT (Kontogiannis et al. 2017), and ATCH (Batz et al. 2013). The CPDs can also be used to find bounded-suboptimal paths (Bono et al. 2019), e.g., whenever we compute a lower-bound using CPDs, we can find a solution by following the path extracted from CPDs. We believe the TCPDs and RTPDs should have the same advantage and can be easily extended for bounded suboptimal search, which we leave as future work.

Acknowledgements

This research was partially supported by the Australian Research Council under grants FT180100140, DP190100013, DP200100025, and also by a gift from Amazon.

References

Batz, G. V.; Delling, D.; Sanders, P.; and Vetter, C. 2009. Time-Dependent Contraction Hierarchies. In *Proceedings of the Eleventh Workshop on Algorithm Engineering and Ex-*

periments, ALENEX 2009, New York, New York, USA, January 3, 2009, 97–105. SIAM.

Batz, G. V.; Geisberger, R.; Sanders, P.; and Vetter, C. 2013. Minimum time-dependent travel times with contraction hierarchies. *ACM J. Exp. Algorithmics*, 18.

Bauer, R.; Columbus, T.; Katz, B.; Krug, M.; and Wagner, D. 2010. Preprocessing Speed-Up Techniques Is Hard. In *Algorithms and Complexity, 7th International Conference, CIAC 2010, Rome, Italy, May 26-28, 2010. Proceedings*, volume 6078 of *Lecture Notes in Computer Science*, 359–370. Springer.

Baum, M.; Dibbelt, J.; Pajor, T.; and Wagner, D. 2016. Dynamic Time-Dependent Route Planning in Road Networks with User Preferences. In *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*.

Bono, M.; Gerevini, A. E.; Harabor, D. D.; and Stuckey, P. J. 2019. Path Planning with CPD Heuristics. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 1199–1205. ijcai.org.

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.

Delling, D. 2011. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1): 60–94.

Delling, D.; Goldberg, A. V.; Pajor, T.; and Werneck, R. F. 2017. Customizable Route Planning in Road Networks. *Transportation Science*, 51(2): 566–591.

Delling, D.; and Nannicini, G. 2012. Core Routing on Dynamic Time-Dependent Road Networks. *INFORMS J. Comput.*, 24(2): 187–201.

Delling, D.; and Wagner, D. 2007. Landmark-Based Routing in Dynamic Graphs. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, volume 4525 of *Lecture Notes in Computer Science*, 52–65. Springer.

Dibbelt, J.; Strasser, B.; and Wagner, D. 2016. Customizable Contraction Hierarchies. *Journal of Experimental Algorithmics (JEA)*, 21: 1–49.

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 Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*, 55–62. AAAI Press.

Kontogiannis, S. C.; Papastavrou, G.; Paraskevopoulos, A.; Wagner, D.; and Zaroliagis, C. D. 2017. Improved Oracles for Time-Dependent Road Networks. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2017, September 7-8, 2017, Vienna, Austria*.

Mahéo, A.; Zhao, S.; Afzaal, H.; Harabor, D.; Stuckey, P. J.; and Wallace, M. 2021. Customised Shortest Paths Using a Distributed Reverse Oracle. In *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*, 79–87. AAAI Press.

Nannicini, G.; Delling, D.; Schultes, D.; and Liberti, L. 2012. Bidirectional *A** search on time-dependent road networks. *Networks*, 59(2): 240–251.

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.

Shen, B.; Cheema, M. A.; Harabor, D. D.; and Stuckey, P. J. 2021. Contracting and Compressing Shortest Path Databases. In *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, 322–330. AAAI Press.

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*, 157–165.

Strasser, B.; Wagner, D.; and Zeitz, T. 2021. Space-Efficient, Fast and Exact Routing in Time-Dependent Road Networks. *Algorithms*, 14(3): 90.

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.

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. *Proceedings of the VLDB Endowment*, 5(5): 406–417.