Fast A* on Road Networks Using a Scalable Separator-Based Heuristic

Renjie Chen
University of Science and Technology of China
Heifei, China
renjie.c@gmail.com

Craig Gotsman
New Jersey Institute of Technology
Newark, NJ, USA
gotsman@njit.edu

ABSTRACT

Fastest-path queries between two points in a very large road map is an increasingly important primitive in modern transportation and navigation systems, thus very efficient computation of these paths is critical for system performance and throughput.

We present a novel method to compute an effective admissible heuristic for the fastest-path travel time between two points on a road map, which can be used to significantly accelerate the classical A* algorithm when computing fastest paths. Our basic method - called the Hierarchical Separator Heuristic (HSH) - is based on a hierarchical set of linear separators of the map represented by a binary tree, where all the separators are parallel lines in a specific direction. A preprocessing step computes a short vector of values per road junction based on the separator tree, which is then stored with the map and used to efficiently compute the heuristic at the online query stage. We demonstrate experimentally that this method scales well to any map size, providing a high-quality heuristic, thus very efficient A* search, for fastest-path queries between points at all distances - especially small and medium range. We show how to significantly improve the basic HSH method by combining separator hierarchies in multiple directions and by partitioning the linear separators. Experimental results on real-world road maps show that HSH achieves accuracy above 95% in estimating the true travel time between two junctions at the price of storing approximately 25 values per junction.

CCS CONCEPTS

• **Theory of computation** → **Design and analysis of algorithms** → **Graph algorithms analysis** → Shortest paths;

KEYWORDS

Road map, fastest-path, heuristic, A* search

ACM Reference format:

Renjie Chen and Craig Gotsman. 2020. Fast A* on Road Networks Using a Scalable Separator-Based Heuristic. In *13th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'20), November 3, 2020, Seattle, WA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3423457.3429363

1 INTRODUCTION

Fastest-path queries between two points in a very large road map is an increasingly important primitive in modern transportation and navigation systems, thus very efficient computation of these paths is critical for system performance and throughput. The road map is modeled as a network - a mathematical graph - the vertices representing road junctions and the edges representing road segments between the junctions. In the simplest scenario, the graph is almost planar, the vertices are embedded in the plane, namely have (x, y) coordinates, and each edge is assigned a positive weight measuring its "travel time", which is the segment's physical length divided by the maximal speed possible on that segment, depending on the category of the road. Highway segments typically have shorter travel times, as they allow higher speeds. The fastest-path query between two points (s, t) on a road map calls for determining the fastest possible route along road segments between two points, and the resulting path is called the "fastest path". The more realistic variant of this problem is when the graph edges are directed, namely the travel time along an edge may depend on the direction of the edge. In the special case of a one-way road, the edge exists in just one direction (or, equivalently, the travel time in the opposite direction is infinite).

Many methods have been proposed to compute fastest paths between two given points on a map, but the importance of this problem motivates the ongoing quest to provide even more efficient methods. The subject of computing minimal-cost paths in graphs has been studied for decades and the literature on this topic is vast. Rather than surveying all this here, we refer the interested reader to the recent survey by Bast et al [1]. We will just mention the Contraction Hierarchies (CH) family of algorithms of Geisberger et al. [9], which relies on a *hierarchical simplification* of the road map, with an Open Source software implementation available [5,13] and the family of methods due to Delling et al. [4] relying on *partition-based overlay graphs*, implemented in Bing Maps. Although both families of methods are quite complex to implement, they achieve very good performance and are considered state-of-the-art in the literature.

1.1 Heuristic A* Search

Our contribution focuses on improving one of the most fundamental and classic approaches to solving the fastest-path problem – the heuristic A* algorithm, which was designed for computing minimal-cost paths in graphs, of which the fast-path problem is just one instance.

Let $G =< V, E, c >$ be an undirected graph with vertex set V, edge set $E \subset V \times V$, and a positive *cost function* on the edges $c: E \to \mathbb{R}^+$. The minimal-cost path problem on G is as follows: Given a pair of query vertices $s, t \in V$, find a path of edges in G from s to t, such that the sum of the costs of these edges is minimal among all possible paths. This is called a *minimal-cost path* and using this we can generalize the function c to express the minimal cost between *all* pairs of vertices: $c: V \times V \to \mathbb{R}^+$.

The A* algorithm [11] is a generalization of the classical (and most basic) Dijkstra algorithm [6] to computing minimal-cost paths, which is notoriously slow. In search of the minimal-cost path from s to t, the Dijkstra algorithm traverses many more graph edges and vertices than those on the actual path. In fact, the complexity of computing the minimal cost from s to the single vertex t is essentially the same as computing the minimal cost from s to *all* other vertices in the graph. To wit, the time complexity of the Dijkstra algorithm, even after some optimization [8], is $O(m + n \log n)$, where m is the number of edges and n the number of vertices in the graph. A* improves on this with the help of an *admissible* heuristic function $h: V \times V \to R^+$, such that $\forall s, t: h(s, t) \leq c(s, t)$, namely h is a *lower bound* on c. The closer h is to c, the faster A* runs, reducing the number of graph edges and vertices traversed during the search for the minimal-cost path. For the trivial *uninformed* $h \equiv 0$, A* reduces to the Dijkstra algorithm, and for the *perfectly informed* $h = c$, A* performs gradient descent on c directly from s to t, with no unnecessary search overhead. Ideally, A* coupled with a good heuristic should reduce the overhead of the search to some constant multiplier of the number of vertices along the optimal path.

Much effort has been invested in designing good heuristics for A*. A complete account would be lengthy, and much of it is domain-dependent, so we discuss here just the most generic method. All heuristics require preprocessing of the graph, in which some auxiliary information is computed and stored, to be referred to later during the running of online minimal-cost queries. The name of the game is, therefore, to optimize the tradeoff between 1) the preprocessing time complexity; 2) the space complexity to store the results of the preprocessing; 3) the time complexity of computing the heuristic based on the stored information when A* runs on a given (s, t) query, and 4) the quality (i.e. accuracy) of the resulting heuristic.

1.2 The Differential Heuristic ALT

A very simple, but surprisingly effective *differential heuristic*, was proposed by Goldberg et al. [10] (who called it *ALT*) and independently by Chow [3]. A small number (usually $k \leq 10$) "landmark" vertices (also called anchors/pivots/centers) $l_1, .., l_k$ are chosen from $V(G)$. In a preprocessing step, for each vertex $v \in V(G)$, the vector of minimal costs $mc(v) = \big(c(l_1, v), .., c(l_k, v)\big)$ is computed and stored. Then, at the online computation of the minimal-cost path from s to t, the heuristic

$$h(s, t) = \|mc(s) - mc(t)\|_\infty$$

is used, which is proven to be admissible and consistent based on the simple triangle inequality. This heuristic requires $O\big(k(m + n \log n)\big)$ preprocessing time and $O(kn)$ space to store. Given s and t, $h(s, t)$ can be computed online in $O(k)$ time.

The degrees of freedom in the ALT heuristic are the choice of the landmark vertices. Goldberg et al. [6] and Efentakis et al. [7] show how to optimize these, concluding that a good choice are landmarks which cover the graph well. In the special case of a plane (or close to plane) graph, a good choice are vertices covering the boundary. However, as the size of the road map increases, more and more landmarks are required, incurring more storage costs. So it would seem that ALT does not scale well.

1.3 The Separator Heuristic

Since each landmark employed by ALT induces a *cost field* on the graph vertices, where every vertex is assigned the value of the minimal cost of a path between vertex and the landmark, we first observe that this concept may be easily generalized. Instead of a landmark being a mere *single* vertex, it may be a *set* of vertices $S \subset V(G)$, and the minimal cost of a vertex v (relative to S) is defined as:

$$mc(v, S) = \min_{u \in S} mc(v, u)$$

This defines a more complicated distance field per landmark, to which the triangle inequality may be applied to obtain an analogous differential heuristic. Unfortunately, in practice this generalization does not add much power to that heuristic.

In an unpublished preprint of ours [2], we first observed that a heuristic significantly more powerful than ALT can be obtained if the vertex set S is a *separator* of the graph, namely its removal (along with the edges incident on the removed vertices) results in V being partitioned into three sets U_1, S and $U_2 = V - U_1 - S$, such that there exist no edges between U_1 and U_2. This means that S *separates* between U_1 and U_2 and the separated graph contains at least two connected components, none of them mixing U_1 and U_2. We say that S *separates* s and t if $s \in U_1$ and $t \in U_2$ or vice-versa. We showed [2] that the following *separator heuristic* is admissible:

$$h_S(s, t) = \begin{cases} c(s, S) + c(t, S) & \text{if } S \text{ separates } s \text{ and } t \\ |c(s, S) - c(t, S)| & \text{otherwise} \end{cases}$$

Essentially the first case means that if S separates s and t, then the minimal cost between s and t is at least the minimal cost from s to S + the minimal cost from t to S, since any path from s to t *must* cross S. In the second case, that S does not separate s from t, we fall back onto the ALT heuristic, which is typically much weaker.

Generating a separator can be quite simple. If the graph is embedded in the plane, namely, each vertex is assigned (x, y) coordinates, and every edge (u, w) drawn as a straight line segment

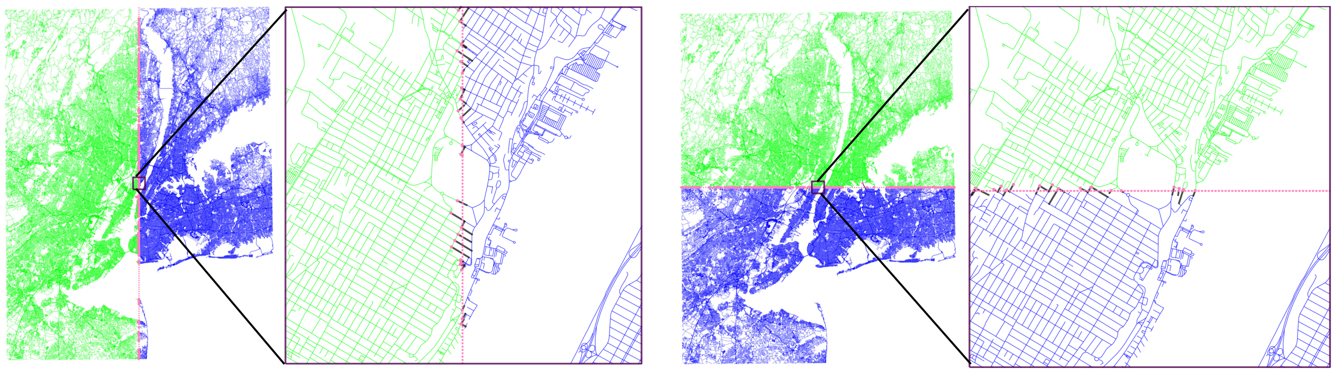

Figure 1: Separators of road map of New York City region generated by vertical or horizontal lines. Green vertices are component U_1, blue vertices are component U_2, and large pink vertices are the separator vertices S. The vertices of S are the left (top, resp.) endpoint of the black edges intersected by the vertical (horizontal, resp.) dotted pink line in the plane.

between the position of u and the position of w, it is possible to generate a separator of the graph by "drawing" a line (or polyline) L on the plane through the graph. L will intersect a subset of the edges $F \subset E$. It is straightforward to verify that the set S defined as the vertices obtained by taking (either) one of the two vertex endpoints of all edges in (a non-empty) F is a separator of G. See Fig. 1 for examples.

Using the separator heuristic based on S means storing the positive value $c(v, S)$ for each $v \in G$. In practice, a number of independent separators $S_i, i = 1, .., k$ are employed, computing and storing in a preprocessing stage *all* the k cost values $c(v, S_i)$ for each vertex $v \in G$. Each results in a different heuristic value $h_{S_i}(s, t)$. Since all the heuristics are admissible, we can take the final heuristic value to be:

$$h(s, t) = \max_{i=1,..,k} h_{S_i}(s, t)$$

To be effective, the set of k separators should cover the map well. This could possibly be a set of equally-spaced horizontal and/or vertical separators. The heuristic power depends critically on the separation property. If s and t are separated by some S, $h_S(s, t)$ will typically be a good estimate of $c(s, t)$, albeit this also depends on the "angle" of the separator relative to s and t. If s and t are not separated by any of the k separators, $h(s, t)$ will be computed using ALT, typically a weak estimate of $c(s, t)$. This implies that if the map is very large, a significant number of separators will be required to ensure that most pairs of query points (s, t) are separated by at least one separator, especially if s is close to t (relative to the size of the map), as is typical in navigation applications. Unfortunately, using a large number k of separators could be unpractical, as it would require the storage of a large number of values per graph vertex, which could be prohibitive. Thus, in its simplest form, it would seem that the separator heuristic is not scalable to very large road maps, just like ALT.

The objective of this paper is to describe a *scalable* method to deploy the separator heuristic of [2] on large roadmaps. We describe an admissible heuristic using a *binary tree* of parallel separators, parameterized by a *tree depth* k, such that the preprocessing time complexity is $O(2^k n \log n)$, the storage space complexity is $O(kn)$, the heuristic computation time complexity is $O(k)$ and the heuristic quality increases with k. In practice, k is taken to be $O(\log n)$, thus the time complexity of preprocessing, storage space complexity of the result, and heuristic computation time complexity become $O(n^2 \log n), O(n \log n), O(\log n)$, respectively. Experimentally we have observed that the overhead of computing the minimal-cost path using this heuristic is a small constant factor of the number of vertices along the path.

2 THE HIERARCHICAL SEPARATOR HEURISTIC (HSH)

We now describe the main contribution of this paper: a more sophisticated version of the basic separator heuristic, one that guarantees that any pair of query vertices (s, t), even those close to each other, will be separated with high probability, and the separator angle (the angle formed by the separator and the fastest path between the query vertices) will not be too small. This relies on a recursive binary subdivision of the map.

If G is a graph representing a road network, and B its bounding box in the plane, then an obvious separator S is a vertical line through the center of the box. The use of this separator requires the storage of the values $c(v, S)$ for each $v \in V$.

Given query vertices (s, t), if indeed S separates them - we may finish the heuristic computation here. Alas, with just one separator, the chances of separating an arbitrary pair of vertices which are close to each other is very small. Thus we continue to recursively separate the two subgraph components generated by S (by a vertical separator through the middle of each component) and store the minimal costs to these separators for each vertex in each relevant subgraph. After d levels of recursion, each vertex will have a vector of d costs, albeit each vertex will have a *different* set of d costs, depending on its x coordinate. We say that a cost entry of s *is common to* a cost entry of t if they are derived from the same separator. In general, a pair of vertices which are close to each other will have more costs in common than a pair of vertices which are distant from each other. These common costs will be all the costs computed at

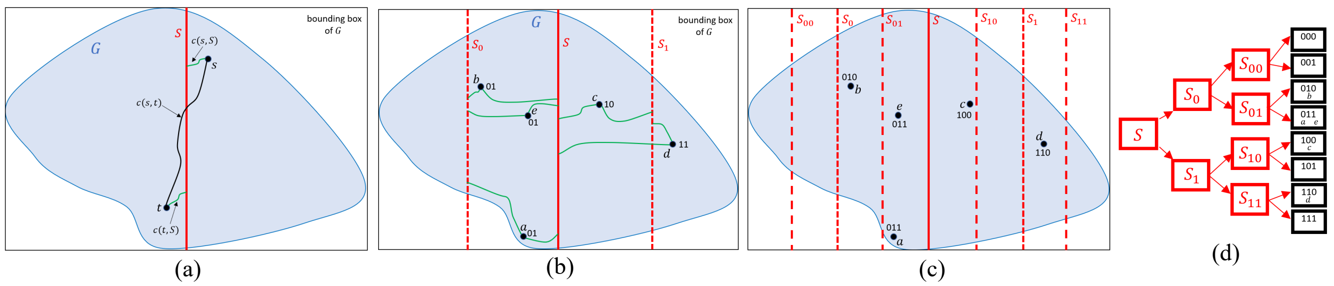

Figure 2: The HSH: (a) Top level of the local separator heuristic. The plane embedded graph is separated by a vertical line S through the middle of the horizontal extent of the graph, separating s from t. The (black) minimal-cost path between s and t has cost $c(s,t)$. The heuristic using S is defined by the (green) "costs" to S: $h(s,t) = c(s,S) + c(t,S)$. (b) Two level separator heuristic. Two new vertical separators S_0 and S_1 are added to the system, subdividing the two horizontal extents defined by S. The binary subscript of the separator indicates in which extent it lies. Any vertex of G can be labeled with a two-bit binary code derived from its x coordinate, indicating in which of the four cells the vertex lies. Each vertex has two (green) "costs", measuring the minimal cost from the vertex to the relevant separators. Since a is separated from c and d immediately at the top level by S, then they have only one cost in common, and the heuristic value is determined by S. Vertices c and d are separated only by S_1 at the second level, so have two costs in common and the cost to S_1 may be used to determine the heuristic value: $h(c,d) = c(c,S_1) + c(d,S_1)$. Vertices a, b and e are not separated at all. (c) Three level separator heuristic. Each vertex has two three-bit codes. $h(a,c)$ is identical to the one-level case, and $h(c,d)$ to the two-level case, but b and e are now separated by S_{01}, so $h(b,e) = c(b,S_{01}) + c(e,S_{01})$. a and e are still not separated. (d) The binary tree corresponding to the three-level vertical separator system.

all the levels higher than the level at which separation occurs. See the example in Fig. 2.

2.1 Computing HSH

Assuming d costs per vertex, the HSH heuristic is computed in two parts: In an offline preprocessing stage, an abstract binary tree of depth d is constructed recursively while generating vertical separators. During the recursive process, a d-bit binary code and a positive real d-vector of costs are assigned to each vertex, using the x coordinates of the vertices. See the pseudo-code in Fig. 3.

In the online query stage, computing the HSH $h(s,t)$ involves comparing the binary codes of s and t from left to right. If the corresponding bits are equal, this indicates a common cost to which ALT may be applied. If the corresponding bits are different, this indicates a separator between s and t and the separator heuristic may be applied. The procedure terminates when either the codes are exhausted, thus identical, namely no separation is ultimately achieved, or when the first nonequal bit is discovered, indicating separation at that level. The complexity of computing $h(s,t)$ is thus $O(d)$. The chances of separating s and t increases exponentially with depth, so this will happen even when they are quite close to each other, given a reasonable depth. Essentially high probability of separation is achieved for $d = O(\log n)$, where n is the number of vertices in the graph. See the pseudo-code in Fig. 3.

2.2 Generalizations of HSH

There are two ways to generalize HSH, improving its heuristic power at the expense of extra storage costs.

Partitioning separators: It is possible to partition each separator S into $k \geq 1$ segments: $S^1, .., S^k$ such that $S = \cup_{i=1}^{k} S^i$. In this case the heuristic for this separator becomes:

$$hp_S(s,t) = \min_{i=1,..,k} h_{S^i}(s,t)$$

and requires storing the k values $c(v, S^i)$ for each vertex $v \in V$. It is not difficult to see that the partitioned separator heuristic can only improve the non-partitioned separator heuristic, while still being admissible:

$$hp_S(s,t) \geq h_S(s,t)$$

There are many ways to partition a separator. A particularly effective strategy is when the separator is crossed by k highways. In this case it is best to partition the separator such that each segment contains one such highway crossing. In practice, however, it may be simplest just to uniformly partition the linear separators into k equal-length segments. Note that if ALT is applied to a partitioned line (e.g. when it does not separate the two vertices), the max operator is applied to the values obtained from the individual segments.

Multiple orientations: Another way to generalize HSH is to employ multiple separator sets, each a binary tree of parallel separators at a distinct angle $\alpha \in \left[0, \frac{\pi}{2}\right]$ relative to the north direction. The vertical separators described above are the special case $\alpha = 0$, but since vertical separators do not always capture well the minimal cost paths, other orientations, especially horizontal separators, complement it well. In practice k equi-distant orientations are employed $\left\{\alpha_i = \frac{\pi}{k}(i-1) : i = 1, .., k\right\}$. The resulting heuristic is just:

$$ho(s,t) = \max_{i=1,..,k} h_{\alpha_i}(s,t)$$

which obviously can only improve the heuristic as k increases.

3 IMPLEMENTATION DETAILS

Since efficiency of the fast-path computation is critical, we list here a number of implementation details which have a significant effect on this:

Avoiding the A* priority queue: It is sometimes possible to completely avoid operating on the priority queue. When the heuristic is almost perfect (close to the true minimum cost), the A* search

```
Preprocess($\boldsymbol{G}, \boldsymbol{k}$)
 for each $v \in G$
        $code(v) := ()$;        // empty binary code
        $cost(v) := ()$;        // empty cost vector
 end
 Code($G, 1, k, x_{min}, x_{max}$);  // $x_{min}$ and $x_{max}$ are the $x$-extents of $G$
```

```
Code($\boldsymbol{G}, \boldsymbol{d}, \boldsymbol{d_{max}}, \boldsymbol{x_1}, \boldsymbol{x_2}$)    // generate codes and costs for
                                   // each vertex of $G$ up to depth $d_{max}$
 if $d \leq d_{max}$
     $x_c = \frac{(x_1+x_2)}{2}$;
     $F :=$ edges of $E$ cut by the line $x = x_c$;
     $S :=$ right endpoints of edges in $F$ ;     // $S$ is separator
     partition $G$ to $(L, S, R)$;
     for each $v \in G$
         $code(v) :=$ concat($code(v), x(v) \leq x_c$ ? 0 : 1);
         $cost(v) :=$ concat($cost(v), c(v, S)$);
     end
     Code($L, d+1, d_{max}, x_1, x_c$);
     Code($R \cup S, d+1, d_{max}, x_c, x_2$);
 end
```

```
HSH($\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{d}$)
$h := 0$;
for $i := 1$ to $d$
    if $code(s, i) == code(t, i)$
        // $code(s, i)$ is the $i$'th bit of $code(s)$
        $h := \max(h, |cost(s, i) - cost(t, i)|)$;
    else
        $h := \max(h, cost(s, i) + cost(t, i))$;
         return $h$;
    end
    return $h$;
end
```

Figure 3: (left) Pseudo-code for the preprocessing stage: constructing binary codes and cost vectors for all vertices of graph G based on x-coordinates of the vertices. (right) Pseudo-code of online computation of the HSH heuristic estimate of the minimal cost between two vertices $s, t \in G$ based on the top $d \leq k$ levels of a separator tree of depth k.

will be close to gradient descent from source to target, and the next node expanded at any given step will be one of the neighbors of the current minimal element of the priority queue. In this case, there is no need to insert this neighboring node in the priority queue only to immediately delete it. A simple comparison of the neighbors to the minimal element of the priority queue is sufficient to determine if this is the case, and then just insert all neighbors which are not the one to be expanded (the one with the smallest cost). This "short cut" has proven to be quite effective.

Efficient implementation of the A* "CLOSED" list: The A* algorithm also uses the so-called "CLOSED" list to determine which network nodes have already been visited. This may be implemented efficiently using a bit-array whose length is the number of nodes in the network. All operations on this data structure are constant time.

Parallelizing the offline pre-computation: The minimal-cost values associated with a separator tree (and stored with the map) are computed in a preprocessing step. The time required for this preprocessing is not critical, but should be accelerated if possible, since these values may need to be recomputed periodically in the case of a dynamic network (e.g. the travel times on road segments are changing due to evolving traffic conditions). Fortunately, due to the hierarchical subdivision nature of the separators, the values are computed independently on strips of network nodes. These may be computed in parallel in a multi-core computing environment, significantly speeding up the preprocessing phase.

4 EXPERIMENTAL RESULTS

We used the datasets used by Chen and Gotsman [2], who extracted directed graphs on the portions of New York (NY), Colorado (COL) and the San Francisco Bay Area (BAY) from OpenStreetMap [12] and removed all vertices of degree 2, a common simplification useful for fast-path computations. We also used a much larger road map of Europe. Table 1 shows the specs of those graphs. We used undirected versions of these graphs, where the edges of the graphs were weighted by the minimal travel time along the two directed edges, which was computed as the Euclidean length of the edge (using the UTM coordinates computed from the latitude and longitude information per vertex) divided by the maximal speed on that edge, as extracted from OpenStreetMap. In our experiments, we distinguished between pairs of points based on the (Euclidean) distance between them. We randomly selected 3,000 pairs of points in "bins" of distances, e.g.: 1-5 km, 5-10 km, 10-20 km, 20-50 km, 50-100 km, 100-200 km, 200-400 km and 400-750 km. For COL, for example, which is approximately a rectangle of size 610×450 km, this covers all possible cases. The (s, t) pairs in each bin were chosen with uniform distribution over area, using the following method for the bin $[a, b]$: s was chosen at random uniformly within the bounding box of the graph, and then "snapped" to the closest map vertex, as long as the snap was not too far. t was then chosen also at random within an annular region centered at s with inner radius a and outer radius b and snapped to the closest vertex as long as the snap was not too far and $\|s - t\| \in [a, b]$. We

Roadmap	**Europe (EUR)**		**Colorado (COL)**		**Bay Area (BAY)**		**New York (NY)**	
Physical dimensions (km)	4,063 x 3,567		623 x 450		178 x 225		85 x 112	
	original	**culled**	**original**	**culled**	**original**	**culled**	**original**	**culled**
Vertices	18,010,173	15,634,580	5,157,103	705,676	3,094,277	743,974	1,582,249	387,139
Edges	42,560,279	39,463,514	5,404,002	1,850,567	3,354,097	1,955,307	1,747,827	1,082,194

Table 1: Specs of the roadmap graphs used in our experiments. Graphs were extracted from OpenStreetMap and degree-2 vertices culled.

then compared the true fastest travel time $c(s,t)$ with the heuristic $h(s,t)$.

For each pair of vertices (s,t), we measure the relative *quality* of the heuristic:

$$\text{qual}(s,t) = \frac{h(s,t)}{c(s,t)}$$

which is a value in [0,1] reflecting how accurate the heuristic is. The closer this number is to 1, the more informed the heuristic is. The quality of the heuristic is the mean of this quantity over all possible pairs (s,t) in the experiment.

The *efficiency* of a heuristic in conjunction with A* is measured as the number of vertices on the fastest path divided by the total number of vertices traversed by A*:

$$\text{eff}(s,t) = \frac{\#vertices\big(\text{fastest\_path}(s,t)\big)}{\#vertices(\text{A*\_traversal}(s,t))}$$

The closer this number is to 1 – the more efficient the heuristic is. The efficiency of the heuristic is the mean of this quantity over all possible pairs (s,t) in the experiment. The best possible efficiency on a road network is typically 40%-50%, since any variant of A* must traverse at least the fastest path vertices and also their immediate neighbors. When an uninformed heuristic is used, the efficiency can sometimes drop dramatically to the vicinity of 1%, meaning 100 vertices of the graph are explored for every vertex along the fastest path.

4.1 HSH vs. FSH

In our first set of experiments we compared the simple Flat Separator Heuristic (FSH) of Chen and Gotsman [2], which uses just a simple set of k parallel separators covering the map to the Hierarchical Separator Heuristic (HSH) based on a binary tree of depth k, described in this paper. Fig. 4 shows the qualities and efficiencies of HSH vs. FSH obtained in the experiments we performed on the road maps of EUR, COL and BAY, one graph per distance bin, as a function of the number of separators used per vertex, which for HSH is the single vertical separator tree depth and for FSH is the number of vertical separators uniformly covering the horizontal extent of the map. Each distance bin is color-coded separately, the dashed line for FSH and the solid line for HSH.

The results confirm that HSH outperforms FSH on most distances, especially the shorter ones, as HSH is able to position a separator between two vertices, even when they are very close to each other. For example, on the map of EUR, in the 1-5km range of distances, HSH obtains up to 87% quality vs. 63% for FSH at a depth of 12. This difference may be even more significant if a deeper tree is used. In the 5-10km range, HSH levels off at depth 11, obtaining 85% vs. 60% for FSH. HSH continues to exhibit a significant advantage at much longer ranges. However, for very long distances (400-750km), FSH starts to supply more than one separator between two vertices, so is more effective than HSH. The efficiency measure behaves in the same way.

Similarly, on the map of COL, in the 1-5km range, HSH obtains up to 85% quality vs. 73% for FSH at depth 9. In the 5-10km range, HSH obtains 80% vs. 73% for FSH, even at tree depth of 7, and still exhibits an advantage at the 10-20km range. Similar results are obtained for the map of BAY. Since this map spans an area smaller than EUR or COL, the first bin was designed to cover a smaller range of distances: 1-2 km, for which a separator tree of depth 9 is needed, and the last bin was 100-200 km. For 1-2 km, HSH obtained on BAY a quality of 83% and efficiency of 41%, vs. 70% and 36% for FSH.

4.2 Multiple Segments vs. Multiple Orientations

The quality of FSH may be improved, at the expense of more preprocessing and storage costs, as described in Section 2.2, by either partitioning each separator into a number of segments, or simply using more separator trees at different orientations. Partitioning each separator into r segments would increase the storage costs by the same factor, as would using r separator trees. Fig. 5 shows the improvement obtained as more segments or tree orientations are used in HSH for queries in the range 5-10km. As expected, increasing r improves both the quality and efficiency of HSH, although it seems it is far more beneficial to use multiple separator trees than to partition the separators. For example, increasing r from 1 to 2 segments increases the heuristic quality from 80% to 82% in COL and from 69% to 78% in NY whereas increasing the number of trees from 1 to 2 orientations increases the heuristic quality from 80% to 91% in COL and from 69% to 87% in NY. Obviously adding more costs per vertex will have diminishing returns, but it seems that using 4 separators trees, spanning the angles $\left\{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}\right\}$ would be the most cost-effective, resulting in heuristic quality of 95% and efficiency of 47% for COL and heuristic quality of 93% and efficiency of 27% for NY.

5 SUMMARY AND DISCUSSION

We have described a separator-based admissible heuristic for A*, which is based on a binary tree of separators, thus scales well to provide an informed estimate of the fastest travel time between two vertices in a road map, at short and long distances. The shorter

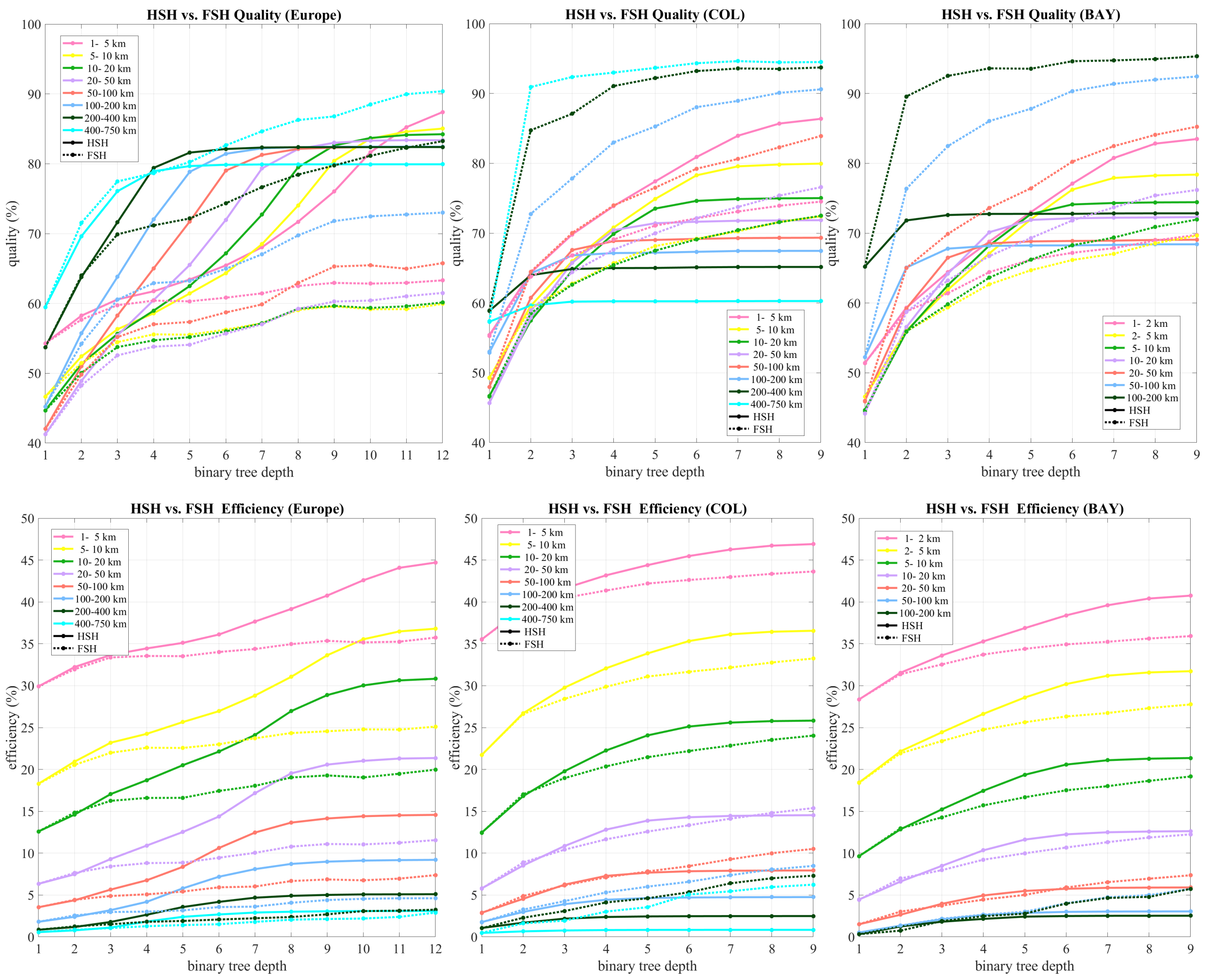

Figure 4: HSH vs. FSH heuristic qualities (top row) and efficiencies (bottom row) on the roadmaps of Europe, COL and BAY for (s, t) pairs of different distances, as a function of the single vertical separator tree depth. For the hierarchical separator heuristic (HSH), marked by solid curves, quality monotonically increases with tree depth. For the flat separator heuristic (FSH), marked by dashed curves, quality tends to increase with the number of separators (equivalent to the HSH tree depth), but at a much lower rate. Large distances require shallower trees than short distances, which benefit much more from tree depth.

distances are captured well by the deeper levels of the tree. We have shown that this approach, which we call the Hierarchical Separator Heuristic (HSH), easily outperforms the simpler Flat Separator Heuristic (FSH), where the map is partitioned by a fixed number of separators, simply because FSH does not scale well to cover large road maps. Loosely speaking, to be effective, the required number of costs stored per vertex is $O(n)$ for FSH vs. $O(\log n)$ for HSH, where n is the number of vertices in the graph.

Our HSH heuristic has been formulated in this paper and experimented with for undirected graphs, but is immediately applicable also to the more realistic case of directed graphs (especially when the graph models a road network). As shown by Chen and Gotsman [2], dealing with directed graphs merely doubles the number of costs stored per graph vertex. They also show that the quality and efficiency of separator-based heuristics is similar for both types of graphs.

In practice, to obtain superior results, a single binary separator tree is not sufficient, since it has a specific orientation, thus does not cover well all possible orientations. We found that, in practice, it is most cost-effective to use four trees, at orientations of multiples of $\pi/4$. Using these trees at depth 7, namely storing 28 values per map vertex.

One aspect of our work that could be improved is the time complexity of the preprocessing stage. Currently it is $O(n^2 \log n)$,

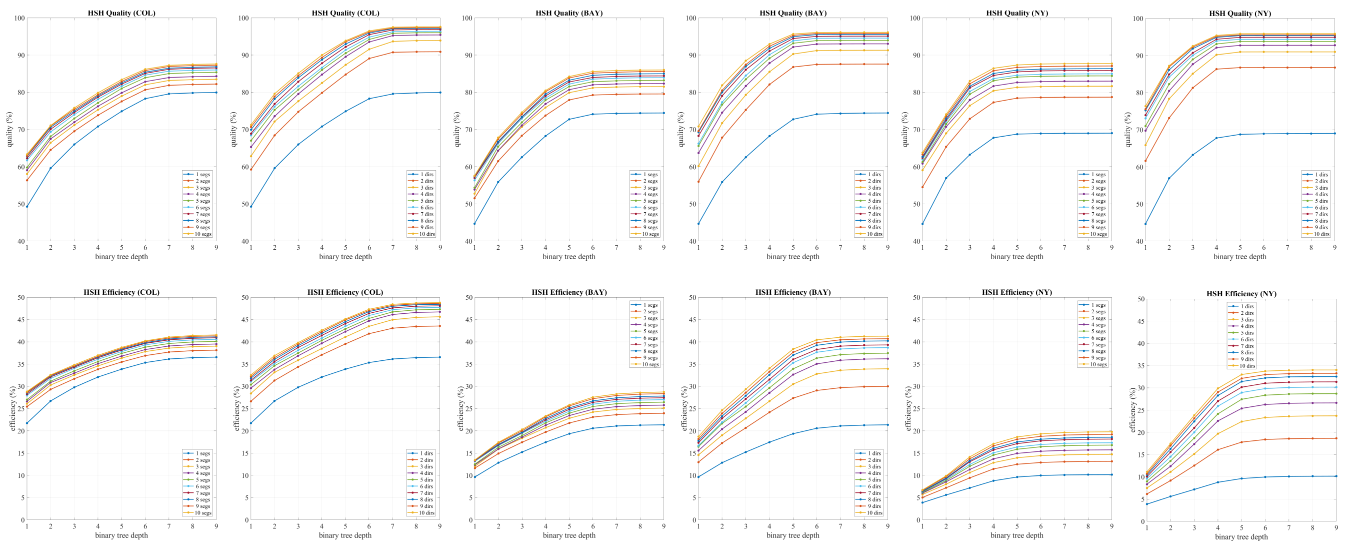

Figure 5: Heuristic qualities (top row) and efficiencies (bottom row) of HSH on the roadmaps of COL, BAY and NY for (s, t) pairs of range 5-10km as a function of depth when increasing either the number of segments per separator in a single (vertical) separator tree or the number of orientations/directions of multiple trees. It is quite obvious that increasing the number of directions is more cost-effective.

which is quite slow, and we speculate that it could still be optimized down to $O(n \log n)$. This is especially important for dynamic traffic maps, in which the edge costs (i.e. travel times) change frequently, forcing the preprocessing to be repeated. On the same note, it would be interesting to devise an efficient *update* procedure for the vertex cost vectors in the event of a few isolated changes to the edge weights.

Finally, we should mention that we have run most of our experiments on rather modest road maps representing single states of the USA, where a depth of 8 suffices to achieve very good results. Since the tree depth scales logarithmically with the number of vertices in the map, for larger maps such as Europe – a binary tree of depth 12 suffices. Similarly, depth 13 would suffice to deal well with a map of the entire continental USA.

ACKNOWLEDGMENTS

This work is partly supported by the Anhui Provincial Natural Science Foundation (2008085MF195) and the National Natural Science Foundation of China (62072422).

REFERENCES

1. H. Bast, D. Delling, A. Goldberg, M. Mueller-Hannemann, T. Pajor, P. Sanders, D. Wagner and R.F. Werneck. Route planning in transportation networks. In Algorithm Engineering: Selected Results and Surveys (L. Kliemann and P. Sanders, Eds.), p. 19-80, Springer, 2016.
2. R. Chen and C. Gotsman. Efficient fastest-path computations in road maps. arXiv Preprint arXiv: 1810.01776, 2018.
3. E. Chow. A graph search heuristic for shortest distance paths. Proc. AAAI, 2005.
4. D. Delling, A.V. Goldberg, T. Pajor and R.F. Werneck. Customizable route planning in road networks. Transportation Science, 51(2):566-591, 2017.
5. J. Dibbelt, B. Strasser and D. Wagner. Customizable contraction hierarchies. ACM J. Exp. Alg., 21(1), 2016.
6. E.W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 1959.
7. A. Efentakis and D. Pfoser. Optimizing landmark-based routing and preprocessing. Proc. ACM SIGSPATIAL Intl. Work. Comp. Transp. Sci., p. 25:25–25:30, 2013.
8. M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. Proc. IEEE FOCS, 1984.
9. R. Geisberger, P. Sanders, D. Schultes and C. Vetter. Exact routing in large road networks using contraction hierarchies. Transportation Science, 46(3):388–404, 2012.
10. A.V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. Proc. SODA, 2005.
11. P.E. Hart, N.J. Nilsson and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. IEEE TSSC, 4(2):100-107, 1968.
12. https://www.openstreetmap.org/
13. https://github.com/RoutingKit/RoutingKit