The Safest Path via Safe Zones

Saad Aljubayrin[#1] Jianzhong Qi[#2] Christian S. Jensen[*3] Rui Zhang[#2] Zhen He[$4] Zeyi Wen[#2]

[#]*Department of Computing and Information Systems, University of Melbourne*
Melbourne, Australia
[1]`aljubayrin@su.edu.sa`
[2]`{jianzhong.qi,rui.zhang,zeyi.wen}@unimelb.edu.au`

[*]*Department of Computer Science, Aalborg University*
Aalborg, Denmark
[3]`csj@cs.aau.dk`

[$]*Department of Computer Science and Computer Engineering, Latrobe University*
Melbourne, Australia
[4]`z.he@latrobe.edu.au`

Abstract—We define and study Euclidean and spatial network variants of a new path finding problem: given a set of safe zones, find paths that minimize the distance traveled outside the safe zones. In this problem, the entire space with the exception of the safe zones is unsafe, but passable, and it differs from problems that involve unsafe regions to be strictly avoided. As a result, existing algorithms are not effective solutions to the new problem.

To solve the Euclidean variant, we devise a transformation of the continuous data space with safe zones into a discrete graph upon which shortest path algorithms apply. A naive transformation yields a very large graph that is expensive to search. In contrast, our transformation exploits properties of hyperbolas in the Euclidean space to safely eliminate graph edges, thus improving performance without affecting the shortest path results. To solve the spatial network variant, we propose a different graph-to-graph transformation that identifies critical points that serve the same purpose as do the hyperbolas, thus avoiding the creation of extraneous edges. This transformation can be extended to support a weighted version of the problem, where travel in safe zones has non-zero cost.

We conduct extensive experiments using both real and synthetic data. The results show that our approaches outperform baseline approaches by more than an order of magnitude in graph construction time, storage space and query response time.

I. INTRODUCTION

Shortest path computation has been studied extensively. However, in some scenarios, the desired path may not be the shortest one. In hazardous environments, it can be life critical to minimize the distance traveled in unsafe regions. For example, a person who drives a long distance through the desert may try to travel via villages ("safe zones") in the desert because a breakdown in an unpopulated region can be life threatening. The traditional shortest path from an origin to a destination is likely to differ substantially from a "shortest" path that is based on a preference to travel as little as possible outside populated regions. In a more familiar scenario, a tourist who plans to walk to a given destination may prefer a path that visits interesting street blocks, e.g., with interesting houses, galleries, or other sights, as much as possible. Here, traveling in interesting regions ("safe zones") is merely *preferred*, and the tourist is unlikely to choose the "safest path" if it comes at the cost of a very long walk. In the first of the above two

scenarios, we may assign zero cost to travel in safe zones, while in the second, we may assign a reduced weight $\alpha \in [0, 1)$ to travel in "safe" zones in order to capture a user's degree of preference for "safety" versus distance. To distinguish these preferred regions from safe zones, we call them *preferred zones*. In a large city, the number of buildings and blocks is very large (e.g., there are over 1 million buildings in New York City[1]). Many of these buildings/blocks are regions of interests (safe zones) and the number of safe zones is hundreds of thousands. This problem can also help cyclists find routes with the least distance outside of bicycle lanes (safe zones). According to OpenStreetMap[2], Amsterdam has over 645000 road segments and many of them have bicycle lanes. As we will see, a naive algorithm for this problem has a squared time complexity with respect to the number of safe zones. It will take a long time to solve this problem in the above applications, so the need for an efficient algorithm is compelling.

Motivated by scenarios such as these, we formulate a new problem called the *safest path via safe zones (SPSZ)*, which is to find a path that minimizes the distance traveled outside a set of discrete safe zones. We study the problem in both Euclidean and spatial network settings. Existing studies on finding a safe path aim to strictly avoid a set of unsafe regions [2], [13]–[15]. Our problem setting is different in that the entire space is unsafe except for a set of *safe zones*, and unsafe regions are still passable. Therefore, algorithms that solve these problems do not apply.

To solve the SPSZ problem in the Euclidean setting, we first transform the data space with safe zones into a graph where the safe zones, the origin, and the destination are represented by vertices, and paths between safe zones, the origin, and the destination are represented by edges. Then any shortest path algorithm may be applied to find the safest path. As the locations and sizes of the safe zones are relatively static, the graph can be precomputed. When an SPSZ query is issued, its origin and the destination are added to the graph as vertices.

A naive transformation of the data space into a graph is to add an edge between every pair of vertices with a weight that equals the unsafe distance between the two vertices. This yields

---

[1]https://www.mapbox.com/blog/nyc-buildings-openstreetmap/
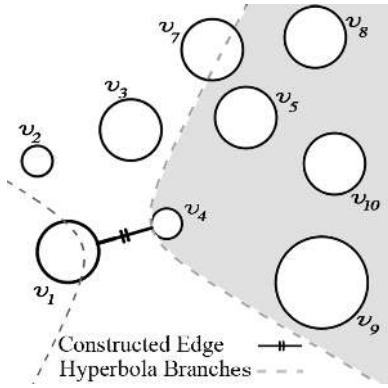[2]http://www.openstreetmap.org/

Figure 1: Pruning with hyperbola

$N(N-1)/2$ edges, where N is the number of safe zones. As we show in our experimental study, most such edges are not competitive and will never be used in any safest path. One way to filter out such superfluous edges is to apply the Floyd-Warshall algorithm [5] to find the shortest paths between all pairs of vertices. However, the cost of doing so is very high ($O(N^3)$).

We observe that the regions containing the vertices with superfluous edges can be elegantly described by hyperbolas and we propose an algorithm that utilizes the properties of hyperbolas to avoid such edges, thus obtaining a much more sparsely connected graph. Figure 1 illustrates the main idea. Assume that we are constructing edges between v_1 and the other vertices in the graph. The distances between the circles denote the unsafe distances between the vertices. First, we construct an edge between v_1 and its nearest vertex v_4. Then we compute a hyperbola based on the centers of the two vertices and the unsafe distance between them. We use the hyperbola branch closer to v_4 to divide the data space into two parts. As will be shown in Section IV, any vertex located on the v_4 side of the hyperbola branch (in the shaded region) has a shortest path to v_1 that goes through v_4. Therefore, we discard vertices representing safe zones in the shaded part of the space when creating edges between v_1 and other vertices.

When solving the SPSZ problem in the spatial network setting, straightforwardly introducing Euclidean space hyperbola into the spatial network space is not possible because the hyperbola definition does not apply to network distance. Instead, we identify a set of critical points that bound the parts of the spatial network where traveling directly to a safe zone v has shorter unsafe distance than when traveling to v through an intermediate safe zone u. We call these critical points *hyperbola points* in analogy with the solution for the Euclidean setting. To identify the hyperbola points of v, we traverse the spatial network from v in a breath-first fashion and label every network vertex d by the safe zones that the path reaching d has passed. The traversal terminates when v is surrounded by network vertices that have been labeled by other safe zones, and the surrounding network vertices are the hyperbola points.

We generalize the problem in the spatial network setting such that the cost of traveling in a safe zone is added to the total path length using a weighted distance. In the Euclidean setting, this generalized problem renders the use of either

hyperbolas or hyperbola points inapplicable. This is due to the lack of a unified distance metric on a path formed by both safe and unsafe segments with different weights. As a result, the graph will be unmanageable. Therefore, we leave the weighted version of the SPSZ problem in the Euclidean setting for future research.

We make the following contributions:

1) We propose a new path finding problem and solve the problem in Euclidean and spatial network settings.
2) To solve the problem of routing through safe zones in the Euclidean setting, we model the data space as a total graph from which edges are excluded during graph construction. We propose a novel edge pruning algorithm that utilizes hyperbolas.
3) In the spatial network setting, we consider both routing through *safe* zones and routing through *preferred* zones. Travel in safe zones has zero cost, while a weighted distance is used for travel in preferred zones to capture a user's preference for "safety" versus distance.
4) We perform extensive experiments to evaluate the efficiency of the proposed algorithms, and the results are summarized as follows:
   a) In the Euclidean setting, our graph construction algorithm is more than an order of magnitude faster than an improved naive algorithm and two orders of magnitude faster than a naive algorithm. Our path finding algorithm is up to an order of magnitude faster than the naive and improved naive algorithms.
   b) In the spatial network setting, for both the case of routing through safe zones and the case of routing through preferred zones, our algorithms outperform the baselines significantly in both graph construction and path finding.

The reminder of the paper is organized as follows. Related work is discussed in Section II. Section III presents the preliminaries and a solution framework. Sections IV and V detail the proposed algorithms in the Euclidean and spatial network settings, respectively. Experimental results are presented in Section VI. Finally, we conclude the paper in Section VII.

II. RELATED WORK

Shortest path (e.g., [2], [5], [9]) finding has been a popular research area in the past few decades. However, we are unaware of any attempt to investigate the problem of finding the safest path via safe zones in an unsafe space. Only a few existing studies consider the safety aspect. Hallam et al. [8] propose to find "*multicriterion shortest paths*," which aim to help submarines find shortest paths based on two factors: the travel time and the risk of being detected by enemy sensors. The setting is different from ours in that it assumes safe points, where the submarine stops, rather than safe zones, in that the size of a safe point (i.e., the area it spans) is not considered, and in that navigating through safe points does not reduce the unsafe distance. In our problem setting, traveling through safe zones does not contribute to the unsafe traveling, and the aim

Table I: Notation

Notation	Explanation
$\mathcal{S}$	The set of closed safe zones
o	The origin of an SPSZ query
d	The destination of an SPSZ query
P	A path from o to d
$\overline{p_i, p_{i+1}}$	A line segment in a path
$S(\overline{p_i, p_{i+1}})$	Sub-segments within the safe zones
$U(\overline{p_i, p_{i+1}})$	Sub-segments outside the safe zones
$|U(P)|$	The unsafe distance of P
s	A safe zone
α	Safe zone traveling cost weight
$dist_R()$	Spatial network distance

is to travel as little as possible outside safe zones. Therefore, the solution of Hallam et al. is not applicable to our problem.

Other studies on safest path problems assume the majority of the data space is safe and has a number of unsafe zones in it, which is opposite to our assumption. In robotic path planning [2], [12], [13], [18], it is an objective to find an optimal path for a robot in the sense that it avoids collisions with obstacles. Some studies assume that the obstacles are moving (e.g., [2]). The obstacles in these studies form the unsafe regions, while the rest of the space is safe. Similarly, in military unit path finding [14], the problem is to find an optimal path for a military unit to move from its current location to another location. These studies also try to avoid certain regions (e.g., obstacles and regions controlled by an enemy). Studies of path finding for *unmanned aerial vehicles (UAVs)* [3], [15] aim to find a path for a UAV to move to a destination safely. Here, safety implies not flying in enemy radar detection zones that form the unsafe regions while the rest of the space is considered safe. The *safest paths for cruise missiles* problem studied in [9] is addressed using a grid based approach. They assign a certain safety probability to each grid edge based on its distance from a threat region. Other works (e.g., [3]) use Voronoi diagrams to represent unsafe polygons and find the safest path when traveling along diagram edges. Most of the studies above assume the unsafe zones are strictly not passable, while unsafe regions are passable in our setting. Therefore, the existing solutions are not applicable even when the space definition of safe and unsafe areas is flipped around.

There have been lots of work on the nearest neighbor or range queries [4], [10], [11], [16], which find those objects in a database that have the smallest distances to a given query point or range. They use spatial distance as the measure whereas the SPSZ problem, after transformed to a graph, uses the graph distance as the measure when finding the path with the smallest distance. Therefore, the nearest neighbor or range query algorithms are not applicable to the SPSZ problem.

III. PRELIMINARIES

We first formalize the SPSZ problem, and then present our solution framework as well as two baseline algorithms. Table I summarizes our notation.

A. Problem Definition

We assume that we are given a set of safe zones denoted by $\mathcal{S}$, an origin point o, and a destination point d. In particular, we consider round and polygon safe zones. Let
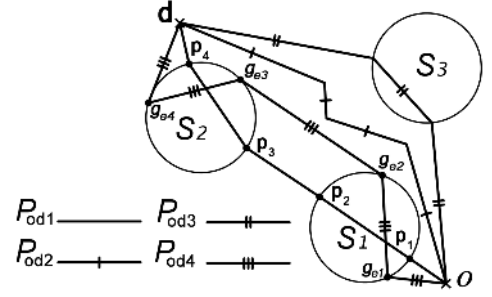

Figure 2: Paths and safe zones

$P = \langle o, p_1, p_2, ..., p_n, d \rangle$ be a *path* between o and d, where p_i denotes a point on the boundary of a safe zone. Then $\overline{o, p_1}$, $\overline{p_1, p_2}$, ..., $\overline{p_n, d}$ each denotes a segment that is either entirely within a safe zone or not within any safe zone at all. For example, in path P_{od1} of Figure 2, $\overline{p_1, p_2}$ and $\overline{p_3, p_4}$ are within safe zones s_1 and s_2, respectively, while remaining segments are not within any safe zone. We use $|P|$ to denote the length of P, where $|P|$ is computed as the sum of the lengths of the segments, i.e., $|P| = |\overline{o, p_1}| + |\overline{p_1, p_2}| + ... + |\overline{p_n, d}|$. Let $S(P)$ and $U(P)$ be the two sets of segments of P that are inside and outside of the safe zones, respectively. We call $|U(P)|$ the *unsafe distance* of P and $|S(P)|$ the *safe distance* of P. In Figure 2, $|U(P_{od1})| = |\overline{o, p_1}| + |\overline{p_2, p_3}| + |\overline{p_4, d}|$, and $|S(P_{od1})| = |\overline{p_1, p_2}| + |\overline{p_3, p_4}|$

The problem of SPSZ is defined as follows:

Definition 1: **Safest Path via Safe Zones (SPSZ) Query:** Given an unsafe Euclidean data space, a set of safe zones $\mathcal{S}$ in it, an origin point o, and a destination point d, the SPSZ query finds a path P from o to d, such that for any other path P' from o to d, the unsafe distance of P is less than or equal to P', i.e., $\forall P'(|U(P)| \leq |U(P)'|)$.

Based on the above definitions, a straightforward solution is to compute and compare the unsafe distance of all the possible paths and then return the path with the shortest unsafe distance. In Figure 2, the unsafe distance of the path P_{od1} is smaller than those of the other paths. However, in the Euclidean setting, there is potentially an infinite number of paths between any two points. Therefore, the straightforward solution may not be feasible. To overcome this problem, we redefine the problem to confine the number of candidate safest paths to a limited number, based on which we propose our problem solution.

Definition 2: **SPSZ Query (redefinition):** Given an unsafe data space, a set of safe zones $\mathcal{S}$ in it, an origin point o, and a destination point d, the SPSZ query finds a sequence of safe zones $s_1, s_2, ..., s_m$, such that $|o, s_1|^{\perp} + |s_1, s_2|^{\perp} + ... + |s_m, d|^{\perp}$ is minimized, where $|\cdot, \cdot|^{\perp}$ denotes the shortest unsafe distance between two objects (either safe zones or query points o and d). If o (or d) is in s_1 (s_m) then we let $|o, s_1|^{\perp}$ ($|s_m, d|^{\perp}$) be 0.

We explain the intuition of the problem redefinition below after introducing some notation. Given a path P, let $\langle o, s_1, ..., s_m, d \rangle$ be the sequence of the origin point (o), safe zones (s_i), and the destination point (d) passed through by P. For example, in Figure 2, both P_{od1} and P_{od4} pass through $\langle o, s_1, s_2, d \rangle$, while P_{od3} passes through $\langle o, s_3, d \rangle$. A segment $g \in U(P)$ connects a pair of adjacent objects $\langle ob_1, ob_2 \rangle$

in the sequence of P, where ob_i is an origin (destination) point or a safe zone. For example, $\overline{g_{e2}, g_{e3}} \in U(P_{od4})$ connects s_1 and s_2. While there may be infinite segments that connect $\langle ob_1, ob_2 \rangle$, we can identify the shortest *line* segment among them, e.g., $\overline{p_2, p_3}$ for $\langle s_1, s_2 \rangle$. Given any sequence of origin/destination points and safe zones, we just need to consider the path formed by the shortest line segments that connect them. As a result, the problem of finding the safest path between o and d becomes one of finding a sequence of safe zones such that the line segments connecting them have the shortest total length. For example, in Figure 2, sequence $\langle o, s_1, s_2, d \rangle$ has a path P_{od1} formed by line segments $\overline{o, p_1}, \overline{p_2, p_3}$, and $\overline{p_4, d}$, which have the shortest total unsafe distance. This is the safest path between o and d.

In the spatial network setting, the problem is defined analogously to the Euclidean definition. Here the number of possible paths between safe zones is not infinite, but it may be very large. Therefore, a similar analysis as for the Euclidean setting applies where we just need to replace "the shortest line segment" with "the shortest path in the spatial network."

B. Solution Framework

In the Euclidean setting, we transform the SPSZ query to a shortest path problem as follows. We let the set of safe zones $\mathcal{S}$ plus the origin and destination points be the set of vertices $\mathcal{V}$, i.e., $\mathcal{V} = \mathcal{S} \cup \{o, d\}$. For every pair of vertices in $\mathcal{V}$, we add an edge to the set of edges $\mathcal{E}$, and we associate it with a weight denoting the shortest distance between the two vertices. After graph construction, finding the safest path between o and d is equivalent to finding the shortest path between o and d on a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, which can be done by a standard graph shortest path algorithm such as the Dijkstra's algorithm. Different queries may have different origin and destination points, but they share the same subgraph $G^\circ = \langle \mathcal{V}^\circ, \mathcal{E}^\circ \rangle$, where $\mathcal{V}^\circ = \mathcal{S}$ and $\mathcal{E}^\circ$ contains the edges connecting the vertices in $\mathcal{S}$. Therefore, the sub-graph G° can be precomputed. When a query is issued, we add the origin point, the destination point, and the relevant edges to connect them to G°.

Therefore, we achieve a two-stage solution framework for the SPSZ query as follows:

- **Stage 1:** Precompute the graph G° on $\mathcal{S}$.
- **Stage 2:** When an SPSZ query with an origin point o and a destination point d is issued:
  (a) Add o and d to G° to form a graph G.
  (b) Perform a shortest path search on G with o as the origin and d as the destination.

In this study, we aim to obtain a graph that contains as few edges as possible while not missing any edge that may appear in a shortest path between two vertices. This is because the number of edges plays a vital role in the efficiency of graph construction and shortest path finding. The shortest path algorithm used is orthogonal to the work in this paper. We have used Dijkstra's algorithm for simplicity, although any other graph shortest path algorithms such as Contraction Hierarchies [6] or Hub Labeling [1] may be used.

In the spatial network setting, we apply the same two-stage framework and replace the Euclidean distance by the network
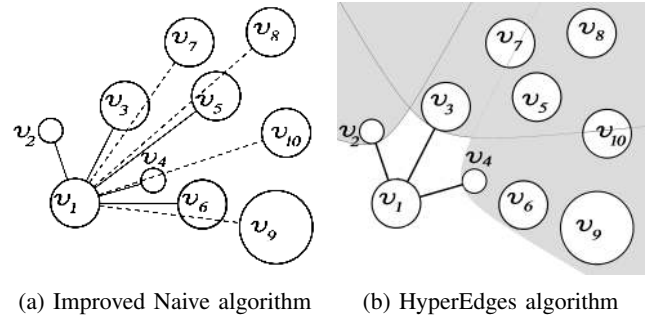

(a) Improved Naive algorithm   (b) HyperEdges algorithm

Figure 3: Constructing Edges for v_1

distance according to the weights of edges. In the following sections, we present our graph construction algorithms for both the Euclidean and the spatial network settings.

IV. SPSZ QUERY IN THE EUCLIDEAN SETTING

First we describe two straightforward solutions. Then we present our proposed algorithm named the HyperEdges algorithm. Next, we give details on how the hyperbolas work for round and polygon safe zones. Finally we discuss how to deal with overlapping safe zones.

A. Baseline Algorithms

Naive Algorithm: A naive algorithm for graph construction works as follows. First, we add an edge to every possible pair of vertices (safe zones). Second, we filter the edges by using the Floyd-Warshall algorithm [5] to compute the shortest path between every pair of vertices. Only the edges that appear in at least one of these shortest paths is kept. This algorithm produces the minimum graph in the precomputation stage. However, it is computationally expensive with cost $O(N^3)$.

Improved Naive Algorithm: The second baseline algorithm improves on the naive algorithm. Instead of adding an edge between every pair of safe zones, we only add an edge if a line segment between two safe zones does not intersect with any other safe zone. Omission of such edges works because the route between the two safe zones that goes via an intersecting safe zone is safer than the direct route. As indicated in Figure 3a, for vertex v_1, only the edges to v_2, v_3, v_4, and v_6 are added to the graph since edges to any other vertex intersects with a safe zone. We also filter the superfluous edges using Dijkstra's algorithm. In the query processing stage, origin and destination edges are added as already described.

Although the running time and the number of edges created by this algorithm is significantly smaller than for the naive algorithm, it is still expensive to check the overlap between line segments and safe zones as needed to avoid unnecessary edges.

B. The HyperEdges Algorithm

Here, we present an efficient algorithm to filter out superfluous edges, which are not used in any safest path. Our solution is based on the observation that the regions containing the vertices with superfluous edges can be elegantly described by hyperbolas, and we propose an algorithm that utilizes the properties of hyperbolas to avoid such edges, thus obtaining a

Algorithm 1: HyperEdges Algorithm - Euclidean

> **Input**: A set of safe zones $\mathcal{S}$ indexed in a quad-tree $\mathcal{Q}$
> **Output**: A precomputed graph $\mathcal{G}^\circ$
> $\mathcal{V}^\circ \leftarrow \mathcal{S}$;
> // create edges
> **for** $v \in \mathcal{V}^\circ$ **do**
> > $u \leftarrow$ next_nearest_neighbor(v, $\mathcal{Q}$);
> > **while** $u \neq null$ **do**
> > > $v.add\_edge(\langle v, u \rangle)$;
> > > $h \leftarrow$ compute_hyperbola(v,u);
> > > // use the created hyperbola h to prune nodes in quad-tree $\mathcal{Q}$
> > > prune_nearest_neighbor(h, $\mathcal{Q}$);
> > > // perform best-first search that only considers zones outside the computed hyperbolas
> > > $u \leftarrow$ next_nearest_neighbor(v, $\mathcal{Q}$);
>
> // remove superfluous edges
> **for** $v \in \mathcal{V}^\circ$ **do**
> > **for** $e \in v.edges$ **do**
> > > $P \leftarrow$ Dijkstra($e.v_1, e.v_2$);
> > > **if** $e \notin P$ **then**
> > > > $v.remove\_edge(e)$;

Figure 4: A Hyperbola

much more sparsely connected graph. We call our algorithm the **HyperEdges** algorithm.

The main idea of the HyperEdges algorithm is that, for each vertex v, we add edges connecting v with other vertices progressively, during which we use the connected vertices to prune part of the data space from being considered for edge creation based on the properties of hyperbolas. Next, we first present our graph construction and query processing algorithms and then explain in detail how we compute the parameters of the hyperbolas used and prove its correctness.

Graph Precomputation: We start with a graph containing just vertices (the safe zones) and no edges, and then we progressively add edges. To add the edges for a vertex v_1, as Figure 1 shows, we first create an edge between v_1 and its nearest vertex, which is v_4. Then we compute a *hyperbola* using the two vertices as the foci. We use the branch of the hyperbola closer to v_4 and call it the *hyperbola branch* of v_4 (the shaded curve in Figure 1). This hyperbola branch divides the data space into two parts. The property of the hyperbola guarantees that any safe zone located on the v_4 side of the hyperbola branch has a shortest path to v_1 that goes through v_4. A path that goes directly to v_1 is longer Therefore, no edges between v_1 and any vertex representing a safe zone on the v_4 side of the hyperbola branch is needed. This way, we have pruned a large number of possible edges.

Next, we find the nearest vertex of v_1 located on the unpruned side of the hyperbola branch, create an edge between it and v_1, and compute another hyperbola branch to prune edges. The above process is repeated until no vertex is left to be connected to v_1 as shown in Figure 3b. Having done the above for every vertex, we obtain a graph that contains all the necessary edges for safest path computation. In Section IV-C,
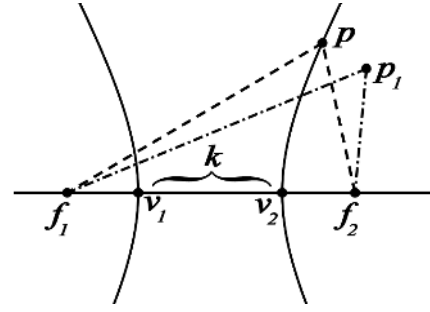
we prove that the hyperbola-based pruning algorithm is safe in that no edge that can belong to a shortest path is pruned.

As will be discussed in the following, although this graph construction algorithm is very effective, it may still contain a small number of edges that will not be in any shortest path. We call these edges *superfluous edges*. To filter superfluous edges, we run Dijkstra's algorithm for every pair of vertices that have an edge between them, and we only keep an edge if it is the shortest path between the two vertices.

Algorithm 1 summarizes the process described above, where a special index $\mathcal{Q}$ is used to index the safe zones for fast nearest vertex computation. We use the quad-tree [17] in our implementation, although any hierarchical index can be used.

Query Processing: The query origin and destination points come in three possible states: (i) both are located in safe zones; (ii) one of them is in a safe zone; (iii) both are outside safe zones. If the origin point or the destination point is in a safe zone, we simply use the safe zone as the origin (or the destination). Otherwise, we add edges to connect the points to the graph G°, which is done by applying the edge creation strategy described above. After edge creation, we run Dijkstra's algorithm to find the safest path. We do not need to filter superfluous edges because we run Dijkstra's algorithm anyway.

Since the hyperbolas are used differently for edge pruning with round and polygonal safe zones, we detail the differences next.

C. Hyperbolas for Round Safe Zones

As shown in Figure 4, a hyperbola is a smooth curve with two branches, where every point p on the curve satisfies that the distance difference from p to two points f_1 and f_2 is a positive constant k [7], i.e.

$$|\overline{f_1, p}| - |\overline{f_2, p}| = k.$$

Here, f_1 and f_2 are called the foci of the hyperbola. Straightforwardly, we can derive that a point p_1 to the *right* of the right hyperbola branch satisfies

$$|\overline{f_1, p_1}| - |\overline{f_2, p_1}| > k.$$

We exploit this property to prune edges.

In particular, Let v_c and u_c be the centers of two round safe zones v and u, respectively. We use these two centers as the foci to construct a hyperbola as shown in Figure 5. Then the right hyperbola branch divides the space into two
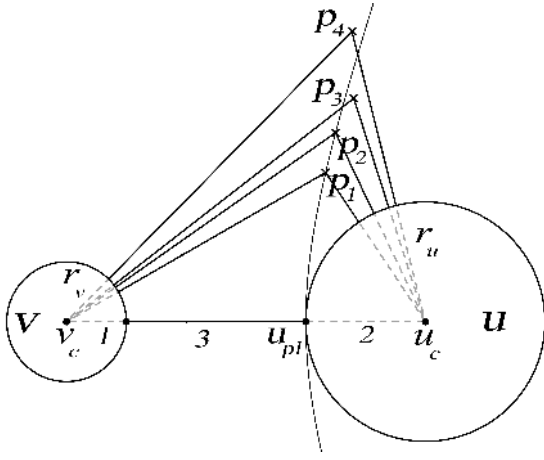
Figure 5: Hyperbola for Safe Zones

sub-spaces, and by definition, a point (e.g., p_3) on the u_c side of the sub-space satisfies

$$|\overline{v_c, p}| - |\overline{u_c, p}| > k \Rightarrow |\overline{v_c, p}| > |\overline{u_c, p}| + k.$$

By letting k be $|\overline{v_c, u_{p1}}| - r_u$, where u_{p1} denotes the closest point to v on u and r_u denotes the radius of u, we obtain

$$|\overline{v_c, p}| > |\overline{u_c, p}| + |\overline{v_c, u_{p1}}| - r_u.$$

Since $|\overline{v_c, u_{p1}}| = |v, u|^{\perp} + r_v$, where $|v, u|^{\perp}$ denotes the shortest distance between v and u and r_v denotes the radius of v, we have

$$|\overline{v_c, p}| > |\overline{u_c, p}| + |v, u|^{\perp} + r_v - r_u$$
$$\Rightarrow (|\overline{v_c, p}| - r_v) > (|\overline{u_c, p}| - r_u) + |v, u|^{\perp}.$$

Here, $|\overline{v_c, p}| - r_v$ and $|\overline{u_c, p}| - r_u$ are the minimum unsafe distances from p to v and u, respectively, while $|v, u|^{\perp}$ is the minimum unsafe distance between v and u. As a result, we know that, the unsafe distance from p to v is larger than that from p to u and then to v. It is safer to travel to u first, and there is no need to create an edge between v and any vertex in the right sub-space of the hyperbola branch of u. Thus, we add no edges to safe zones in this sub-space. We formalize the pruning strategy as the following theorem.

Theorem 1: Let two round safe zones v and u be given along with a hyperbola defined by

$$(|\overline{v_c, p}| - r_v) - (|\overline{u_c, p}| - r_u) = |v, u|^{\perp}.$$

For any point p located beyond the hyperbola branch closer to u, it is safer to travel through u rather than to travel directly from v, i.e.,

$$(|\overline{v_c, p}| - r_v) > (|\overline{u_c, p}| - r_u) + |v, u|^{\perp}.$$

Proof: The correctness of the theorem is guaranteed by the definition of hyperbola as shown by the derivation above. ∎

As an example, in Figure 5, $|\overline{p_1, v_c}| - r_v = 4$, $|\overline{(p_1, u_c)}| - r_u = 1$ and $|v, u|^{\perp} = 3$. At p_1, it is the same in terms of safety to travel to u then to v as to travel directly to v. The same applies for point p_2 because $|\overline{(p_2, v_c)}| - r_v = 1.5$ and $|\overline{(p_2, u_c)}| - r_u = 4.5$. However, point p_3 is located beyond the

hyperbola branch closer to u, and hence it is safer to travel through u to v rather than to v directly. In contrast, p_4 is not located beyond the hyperbola branch. Traveling from p_4 to v directly is safer than traveling through u:

$$|\overline{p_4, v_c}| - r_v = 6.37 < |\overline{p_4, u_c}| - r_u + |v, u|^{\perp} = 3.53 + 3 = 6.53.$$

Theorem 1 guarantees no false negatives in edge creation, i.e., only superfluous edges are pruned. However, it cannot guarantee no false positives, meaning that superfluous edges can be created. While such edges do not affect correctness, they may affect performance. Since the number of superfluous edges is usually small due to the pruning capability of the HyperEdges algorithm (within 1% of the total number of edges created in experiments), filtering them incurs small overhead.

There are two cases where superfluous edges are generated:

- If the safe zone of a vertex spans the two sub-spaces created by a hyperbola branch, we cannot prune the vertex since it contains points that should not be pruned. However, it is still possible that it is safer to travel from the vertex through u to v. An example can be seen in Figure 1, where v_7 spans the two sub-spaces created by the hyperbola branch of v_4, and hence it cannot be pruned by v_4. This type of vertex usually does not yield many superfluous edges, as they may still be fully enclosed in a sub-space pruned by some other vertices. For example, v_7 in Figure 3b is pruned by v_3.

- When adding an edge between v and u, our pruning strategy essentially disregards the vertices whose respective shortest paths to v contain only one intermediate vertex u. However, we cannot prune the vertices whose respective shortest paths to v contain additional intermediate vertices, even though this case is infrequent especially in the Euclidean setting.

Correctness: The correctness of using hyperbolas to prune the edges is guaranteed as follows. Using the proposed hyperbola based pruning algorithm, only when there is a path $v \rightarrow u \rightarrow ... \rightarrow w$ shorter than edge $v \rightarrow w$, the edge $v \rightarrow w$ will be pruned. Such a path will not exist if edge $v \rightarrow w$ appears in some shortest path. This means that the graph constructed in the pre-computing stage keeps the edges that appear in at least one of the shortest paths between any two safe zones. Therefore, in the path finding stage the correct shortest path can be found.

Complexity: The key advantage of the HyperEdges algorithm is that, as Figure 5 shows, a hyperbola branch can prune edges to vertices representing safe zones in a significant portion of the space. If the vertices surrounding a vertex v are distributed evenly, only a few hyperbola branches are needed to cover the whole space. Thus, only a few edges will be created for each vertex. Roughly speaking, unless the vertices are highly skewed, the number of edges created per vertex is on the order of $O(1)$, and the number of edges created for N vertices is on the order of $O(N)$. In the worst case scenario, none of the safe zones can serve as an intermediate node in any other safe zone's safest paths. Then no edge can be pruned and the time complexity to generate these edges is the same as the naive algorithm. However, this is usually not the case
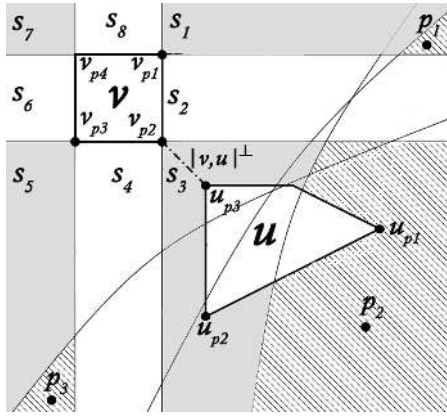
Figure 6: Hyperbolas for 3 Partitions

(a) Same overlapping set   (b) Different overlapping sets

Figure 7: Hyperbolas for Overlapping Round Safe Zones

for real data and our pruning algorithm is very effective as shown by experiments. In contrast, the naive algorithm always creates an edge for every pair of vertices, i.e., the number of edges created for N vertices is always on the order of $O(N^2)$.

D. Hyperbolas for Polygonal Safe Zones

For polygons, we need more than one hyperbola per pair of vertices to prune edges. This is because a polygon does not have a center that is equidistant to every point on the boundary, and therefore we cannot define $(|\overline{v_c, p}| - r_v) - (|\overline{u_c, p}| - r_u) > |v, u|^\perp$ on the centers v_c and u_c of two polygons v and u. To overcome this difficulty we divide the space for each vertex into multiple partitions and use multiple hyperbolas for pruning the different partitions.

We observe that the above inequality for round safe zones essentially defines a region where the unsafe distance from a point p to v ($|\overline{v_c, p}| - r_v$) exceeds the sum of the unsafe distance from p to u ($|\overline{u_c, p}| - r_u$) and the minimum unsafe distance between v and u ($|v, u|^\perp$). We rewrite this inequality as follows:

$$|v, p|^\perp - |u, p|^\perp > |v, u|^\perp.$$

Here, $|v, p|^\perp$ and $|u, p|^\perp$ denote the minimum unsafe distance from p to v and u, respectively.

We relax this inequality as follows to obtain the hyperbola for two polygon safe zones v and u:

$$|\overline{v_{p1}, p}| - |\overline{u_{p1}, p}| > |v, u|^\perp.$$

Here, v_{p1} and u_{p1} are two points from v and u that serve as the foci (cf. Figure 6, where the dashed line connecting v and u is $|v, u|^\perp$). We need to find the two points that satisfy the following for every point p:

$$|\overline{v_{p1}, p}| \leq |v, p|^\perp \text{ and } |\overline{u_{p1}, p}| \geq |u, p|^\perp,$$

so that the original inequality is also satisfied and we guarantee the correctness of pruning using the hyperbola. By definition, $|\overline{u_{p1}, p}| \geq |u, p|^\perp$ is satisfied for any point u_{p1} on u. Meanwhile, $|\overline{v_{p1}, p}| \geq |v, p|^\perp$ holds for any point v_{p1} on v. Thus, to satisfy $|\overline{v_{p1}, p}| \leq |v, p|^\perp$, we need a point v_{p1} such that $|\overline{v_{p1}, p}| = |v, p|^\perp$. This means that v_{p1} must be the closest point on v to any point p. Since different points have different closest points on v, there is no single point v_{p1} that satisfies $|\overline{v_{p1}, p}| = |v, p|^\perp$ for every different p. To overcome this
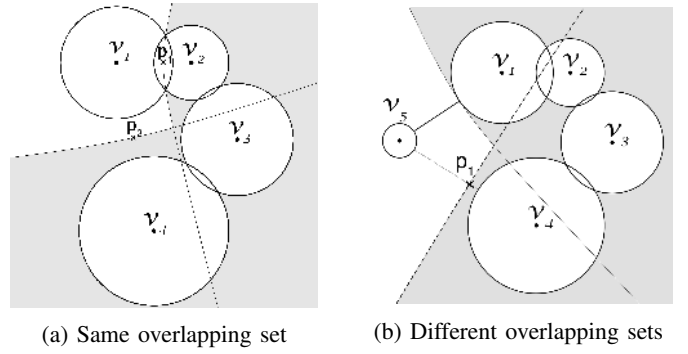
limitation, we divide the space into multiple partitions, where the points in each partition share the same closet point on v.

We extend the edges of the *minimum bounding rectangle (MBR)* of v across the space to divide the space into 8 partitions $s_1, s_2, ..., s_8$ as shown in Figure 6. We use the MBR of v rather than v directly to simplify the processing of finding the closest point on v for the points in each partition. This does not introduce false negatives because any point outside the MBR of v must be at least as close to the MBR as it is to v. It may result in superfluous edges, but the number of these is expected to be small as the MBR is usually a good approximation of a spatial object.

After the space division, every shaded partition has only one point in the MBR of v as its closest point, which is the corresponding corner point. For example, in Figure 6, v_{p1} is the closest point among the points in partitions s_1 to the MBR of v. Similarly, v_{p2}, v_{p3}, and v_{p4} are the closest points for the points in partitions s_3, s_5, and s_7, respectively. We use the corner point and a point from u to compute a hyperbola for pruning in a shaded partition. The non-shaded partitions still do not have a unique point that is the closest to all the points in the partition. This will cause a problem which we call the *blind area* problem. A blind region is a small region with uncertain safety that therefore cannot be used straightforwardly for edge pruning. Blind regions are covered shortly.

Since there are multiple partitions to be used for pruning, we compute multiple hyperbolas, each with a different pair of foci. An example is shown in Figure 6, where three hyperbola branches are computed for pruning in partitions s_1, s_3, and s_5. When u is in different partitions, we use different sets of hyperbolas, which is straightforward and hence omitted.

Blind Region: As mentioned above, for an unshaded partition, there is no single point that can serve as one of the hyperbola foci. This is because different points in an unshaded partition have different closest points on the MBR of v. For example, in Figure 6, points in s_2 may view v_{p1}, v_{p2} or some other point in-between as their closest point on the MBR of v, depending on the positions of the points. As a result, there is no single hyperbola for pruning in s_2. However, we can still achieve some pruning in this type of partitions.

Assume that we can find all the points on $\overline{v_{p1}, v_{p2}}$ and compute hyperbolas for all of them with another foci on u, such as u_{p1}. Then the intersection of all the pruning regions defined by these hyperbolas can be used safely for
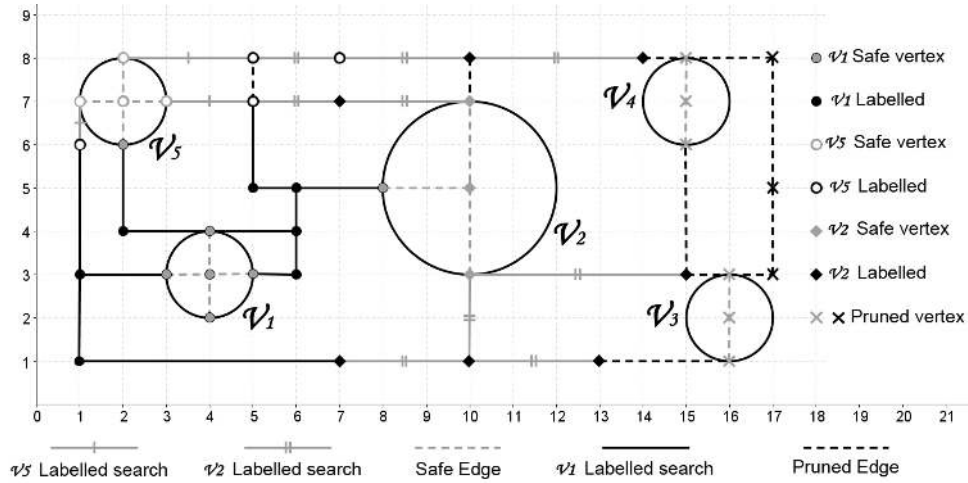
Figure 8: HyperEdges in Spatial Networks

pruning. Since this is impossible, we relax the pruning by only computing the hyperbola for v_{p1} and u_{p1}. Because v_{p1} is farthest from u_{p1} among all the points on $\overline{v_{p1}, v_{p2}}$, we know that $Hyp(v_{p1}, u_{p1})$ must have the right-most intersection point on the extended edge of $\overline{v_{p4}, v_{p1}}$. Any point in s_2 to the right of the intersection is enclosed by the pruning region defined by the hyperbola of any other point on $\overline{v_{p1}, v_{p2}}$. In contrast, we cannot infer that it is safe to use the region to the left of the intersection for pruning; thus, we call it a *blind region* and do not use it for the pruning of edges.

E. Overlapping Safe Zones

Until now we have implicitly assumed that safe zones do not overlap. However, this assumption may not hold in the Euclidean setting. Thus, the minimum unsafe distance between two safe zones may not be the direct distance between them, but a distance through some other safe zones they overlap with. We thus adjust the HyperEdges algorithm to handle this case.

Overlapping round safe zones: We first group the vertices (safe zones) to form subsets of vertices $\mathcal{V}_1$, $\mathcal{V}_2$, ..., $\mathcal{V}_m$ as follows. For each vertex v, if it has not been assigned to any subset, we find the subset that contains at least one vertex that overlaps v, and we assign v to the subset. If no such subset can be found, we create a new subset and assign v to it. This way, for any subset $\mathcal{V}_i$, traveling between any two points in $\mathcal{V}_i$ can always occur within safe zones, and the minimum unsafe distance between any two vertices in $\mathcal{V}_i$ is 0. For example, in Figure 7a, all the vertices belong to the same subset, while in Figure 7b, there are two subsets, $\{v_1, v_2, v_3, v_4\}$ and $\{v_5\}$.

In edge creation, if two vertices v and u are in the same subset, the minimum unsafe distance between them is 0, which is then used as the constant k in hyperbola computation, i.e., we compute the hyperbola as

$$(|\overline{v_c, p}| - r_v) - (|\overline{u_c, p}| - r_u) = 0.$$

If v and u are in two different subsets $\mathcal{V}_i$ and $\mathcal{V}_j$, we use the minimum unsafe distance between the two subsets as the constant k rather than using the minimum unsafe distance between the two vertices directly. This is because the former

may be shorter as there is no cost of traveling in the same subset. Formally,

$$(|\overline{v_c, p}| - r_v) - (|\overline{u_c, p}| - r_u) = \min\{|v_i, v_j|^{\perp}\}, v_i \in \mathcal{V}_i, v_j \in \mathcal{V}_j.$$

For example, in Figure 7b, when computing the hyperbola for vertices v_5 and v_4, we use the distance between v_5 and v_1 as k rather than the distance between v_5 and v_4.

Overlapping polygonal safe zones: Handling overlapping polygonal safe zones is simpler. When two polygonal safe zones v and u overlap, we simply merge them and generate a new polygon $m_{v,u}$. This newly generated polygon $m_{v,u}$ then replaces v and u in all subsequent computations.

V. SPSZ QUERY IN THE SPATIAL NETWORK SETTING

We consider a spatial network R represented by a graph $G_R = \langle V_R, E_R \rangle$, where vertices V_R represents the set of network vertices (e.g., intersections) and edges E_R represents the set of segments connecting the vertices. A safe zone s in the spatial network is represented by the set of network vertices s_V and the set of edges s_E covered by the safe zone, i.e., $s = \langle s_V, s_E \rangle$. If an edge is partly covered by a safe zone, a new vertex is introduced at the safe zone intersection point, and the edge is replaced by a safe and an unsafe edge. We call these network vertices and edges *safe vertices* and *safe edges*, respectively. For example in Figure 8, every circle represents a safe zone that covers the safe vertices and edges denoted by the gray points and dashed line segments.

To solve the SPSZ problem in a spatial network R, as discussed in Section III, we construct a graph G° where the safe zones serve as the vertices. To connect the vertices we add the paths in G_R to the graph G°. A *naive approach* to connect the vertices is to compute the shortest path between every pair of safe zones in G_R and add the (safe and unsafe) edges and vertices used by each such path to G_R. However, this would result in searching the whole spatial network for each safe zone, which is expensive.

A. The HyperEdges Algorithm

Similar to solving the SPSZ problem in the Euclidean setting, we again consider using hyperbolas to reduce the edge

creation cost. However, straightforwardly applying the Euclidean hyperbola described in Section IV-C does not guarantee correct edge pruning. This is because the Euclidean distance between two points in a spatial network is usually different from the network distance. To overcome this problem, we use *network hyperbola*. A network hyperbola is a set H of points in a spatial network, where every point p in the set satisfies

$$|dist_R(f_1, p) - dist_R(f_2, p)| = k.$$

Here, $dist_R()$ returns the network distance between two points, i.e., the length of the shortest path in the spatial network. The two points f_1 and f_2 are the foci of the network hyperbola, and k is a given positive constant. A network hyperbola contains a set of points that partition the network into two. In one of the partitions, the shortest path to f_1 of *every* point passes through f_2. In the other partition, the shortest path to f_1 of *no* point passes through f_2. Given two safe zones s_v and s_u, their network hyperbola is computed by testing whether the points in the network vertices satisfy the hyperbola inequality[3]. When a point p is to be tested, we use two safe vertices of s_v and s_u that are the closest to p as f_1 and f_2, respectively. The network distance between the two safe zones is used as the constant k. For example in Figure 8, for the two safe zones v_1 and v_5, the minimum distance between them (4 distance units) is k. The network vertex at $(5, 7)$, denoted by $n_{5,7}$ is a network hyperbola point such that

$$dist_R(n_{5,7}, n_{4,4}) - dist_R(n_{5,7}, n_{3,7}) = 6 - 2 = 4 = k.$$

We use Figure 8 to illustrate the HyperEdges algorithm using network hyperbolas:

1) For every safe zone vertex v (e.g., v_1), perform a single source shortest path search on G_R starting from all of v's border safe vertices (vertices inside v with an edge ending outside v, e.g., $n_{3,3}, n_{4,4}$). This is to find a path to connect v with every neighboring safe zone u.
2) For every safe zone u (e.g., v_5), use the network distance $dist_R(v, u)$ as k (e.g., 4) along with two safe vertices f_1 and f_2 from v and u to form a hyperbola equation. We do this for all the neighboring safe zones at the same time using a graph vertex labeling technique, which is detailed below.
3) The search of paths to neighboring safe zones continues until v is fully surrounded by network hyperbola points. The remainder of the spatial network is pruned (e.g., the x vertices).

Finding the network vertices satisfying the network hyperbola condition: In the graph pre-computation stage, we need to identify the network vertices in v's partition of the spatial network as defined by a network hyperbola. A query starting at any of these network vertices towards v should go directly to v rather than going through any intermediate safe zones. We use a labeling technique to identify these network nodes. We attach a variable l to each network vertex n to store the id of the safe zone that is passed through by the search before reaching n. The labeling technique works as follows:

[3]We do a best-first traversal on the spatial network and hence only a limited number of points are tested.
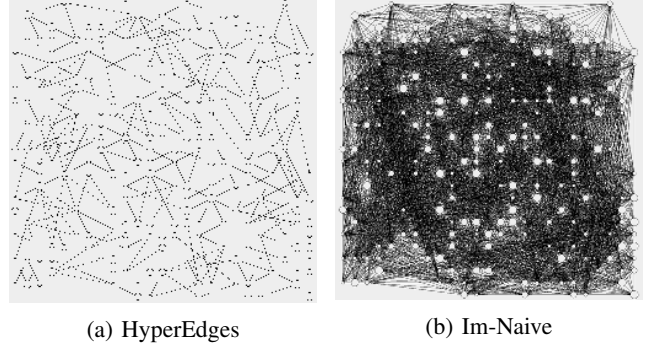

(a) HyperEdges   (b) Im-Naive

Figure 9: Edges Created in Graph Construction

1) For every safe zone v (e.g., v_1), use Dijkstra's algorithm to search the neighboring network vertices in all directions. Every found network vertex (e.g., the black dots in Figure 8) is labeled with the id of v and added to a priority queue Q. When a new neighboring safe zone u is reached during the search (e.g., v_5), we continue the search beyond u, but omit the distance to travel through the safe zone.
2) Label every network vertex reached through u with the id of u (e.g., the hollow dots for v_5) and add it to Q.
3) Continue the search as long as there are vertices labeled with v's id in Q. The search stops when Q becomes empty or Q does not contain any vertex labeled with the id of v.

Here, the termination condition means that either all network vertices have been accessed or any path from the safe zone v has reached a network vertex that is labeled by some other safe zone. In the latter scenario, a network vertex n labeled by another safe zone u has a safer path through u to v than going directly to v. Since any network vertex reached from n should also go through n first, it should also go through u first, and hence a direct path to v is unnecessary.

At query time, we treat the origin and destination points as safe zones and label the network vertices for them similarly to build edges from them to the neighboring safe zones. Then the origin and destination points are connected to the precomputed graph G°, and we can apply Dijkstra's algorithm to find the safest path.

B. Adding the Cost of Traveling in Safe Zones

In the above discussion we assumed that travel in safe zones has zero cost. We proceed to introduce a weighting parameter $\alpha \in [0, 1)$ and add the cost of traveling in safe zones to the formalization of network hyperbola as follows:

$$|dist_R(v, p) - dist_R(u, p) + (\alpha * dist_R(u))| = k.$$

Here, $dist_R(u)$ denotes the length of the shortest path to travel through a safe zone u. To accommodate this cost in the HyperEdges algorithm, we just need to change the computation of the network hyperbolas in the algorithm to use the equation above.

VI. EXPERIMENTAL STUDY

In this section, we empirically study the performance of the HyperEdges algorithm. In the experiments for the Euclidean setting, we compare HyperEdges to both the naive and the improved naive (here denoted as "*Im-Naive*") algorithms described in Section IV-A. We observe that the naive algorithm is significantly less efficient in terms of the number of created edges as well as running time than both Im-Naive and HyperEdges: this naive algorithm takes more than a day to construct a graph for a dataset of 10,000 objects. Therefore, we omit the results of the naive algorithm. In the experiments for the spatial network setting, we compare HyperEdges with the naive algorithm as described at the beginning of Section V.

We conducted the experiments on a desktop PC with 8GB RAM and a 3.4GHz Intel$^{(R)}$ Core$^{(TM)}$ i7 CPU. The disk page size is 4K bytes. For the Euclidean setting experiments, we use a real dataset containing the locations of 1042 villages in Saudi Arabia[4]. Specifically, the coordinates of the 1042 villages are used as the **base safe zone dataset**, and we generate additional safe zones following a Gaussian distribution ($\sigma = 0.166$) centered at the real villages. By default, we use 10,000 safe zones covering 8% of the total area of the data space. For the spatial network setting experiments, we use the London road network extracted from Open Street Map[5], which contains 515,120 vertices and 838,702 edges. We also extracted the location of 192 police stations in London and use them as safe zone centers. We call this the "Police Stations" dataset.

We vary parameters such as dataset size, safe zone density, and data distribution to gain insight into the algorithm performance in different settings. The detailed settings are given in the individual experiments.

A. Euclidean Setting Experiments

First, we evaluate the effectiveness of the HyperEdges algorithm in term of constraining the number of edges created. This includes the edges created in both the graph construction and query processing stages. Then we evaluate the algorithm efficiency, also in both stages. We use a **default dataset** of 10,000 safe zones covering 8% of the total area of the data space generated based on real data as described at the beginning of the section.

Edge Pruning Effectiveness: We measure the number of edges created by the different algorithms and observe that HyperEdges (i) creates a much smaller number of edges before filtering and (ii) creates the same set of edges as the improved naive algorithm does after filtering in both stages. The graph constructed by HyperEdges has almost no superfluous edges (less than 1%), while more than 90% of the edges created by Im-Naive are superfluous edges and need to be filtered. We also compare the graphs after filtering and confirm that the two algorithms achieve the same graph.

To given a more intuitive view of the algorithm performance on edge pruning, we generate 200 round safe zones with random distribution as shown in Figure 9. We ran HyperEdges and Im-Naive to create the edges, and the figure shows the edges produced before filtering. We can see that
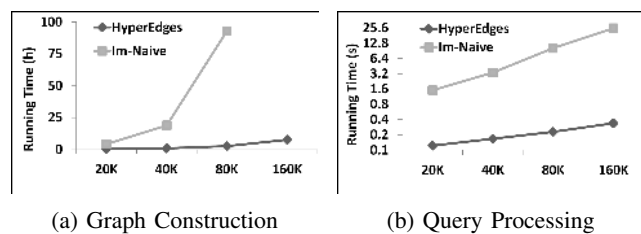
(a) Graph Construction (b) Query Processing

Figure 10: Effect of Safe Zones Cardinality

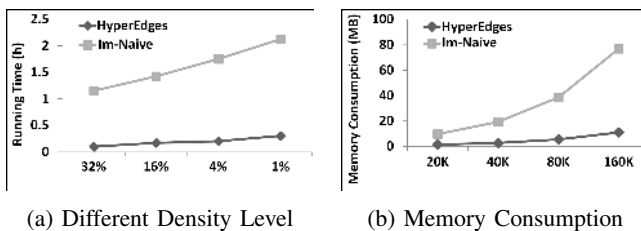(a) Different Density Level (b) Memory Consumption

Figure 11: Effect of Density and Memory Consumption

HyperEdges creates a much smaller number of edges, which demonstrates the effectiveness of our hyperbola based edge pruning technique.

Algorithm Efficiency: We measure the running time of both the graph construction and the query processing stages in different dataset cardinality, object distribution and object density level. In addition, we measure the memory consumption of the two algorithms for storing the graph created.

1) Effect of Dataset Cardinality: We used the base safe zone dataset to create datasets of different sizes. Figure 10a shows the running time on graph construction for the generated datasets of different sizes. HyperEdges is more than an order of magnitude faster than Im-Naive when the dataset size is within 80,000. At the 160,000 dataset, the Im-Naive algorithm cannot finish within 7 days and no result is obtained. This again confirms the advantage of our hyperbola based edge pruning techniques in reducing the graph construction time. Also the running time of HyperEdges increases much slower with the increase in dataset cardinality in the graph construction stage, while that of Im-Naive increases dramatically. This demonstrates the scalability of the HyperEdges algorithm.

In the query processing stage, we compare the average running time of HyperEdges with that of Im-Naive for processing 100 queries where each query has randomly generated origin and destination points. Note that in these experiments the precomputed graphs for both algorithms are the same as both algorithms produce the same graph after filtering.

Figure 10b illustrates the query processing time, which is the time taken to add the origin and destination points to the graph and to find the safest path, where the dataset cardinality is varied. We observe that, HyperEdges can be more than 50 times faster than Im-Naive when processing safest path queries. This is because HyperEdges inserts the query origin and destination points into the graph using the hyperbola based technique, which is more efficient than Im-Naive. The total query processing time of HyperEdges is at least 90% smaller than that of Im-Naive for the various datasets tested. Experiments where the other settings are varied for the query processing stage show similar results. They are omitted due to the space limit.

2) Effect of Safe Zone Density: Figure 11a shows the graph construction time where we vary the percentage of the data space covered by the safe zones from 32% to 1% by varying the radius of the safe zones. Again, HyperEdges outperforms Im-Naive constantly and it is at least seven times faster for all density levels tested. Note that the size and density of the safe zones do affect the efficiency of both algorithms. This effect is not too observable for HyperEdges in the figure due to the large range of the Y-axis. However, when the density is 1%, the running time of HyperEdges is three times that when the density is 32%.

3) Memory Consumption: Figure 11b shows the maximum memory consumption (in MB) of the two algorithms in graph construction on the datasets of different sizes. As the figure shows, HyperEdges constantly consumes less memory comparing with Im-Naive. This is because HyperEdges creates much fewer edges. Meanwhile, the advantage of HyperEdges grows as the dataset cardinality increases. This is in accordance to our discussion at Section IV-C that HyperEdges creates edges whose number increases approximately linearly to the number of safe zones, while the number of Im-Naive increases approximately quadratically.

4) Effect of Safe Zone Distribution: Figure 12a shows the graph construction time when the safe zone distribution is varied. We use $\mu = 0.5$ and $\sigma = 0.166$ for the Gaussian distribution and $\alpha = 0.1$ for the Zipfian distribution to generate 10,000 safe zones covering 8% of the total area of the space, where the safe zone centers follow the given distributions around the safe zones in the base dataset. As the figure shows, HyperEdges runs more than an order of magnitude faster than Im-Naive for all different distributions tested.

5) Effect of Safe Zone Shape: Both HyperEdges and Im-Naive can process round and polygon shaped safe zones. We test their performance with the default dataset (round sate zones) and a dataset of the same parameters but with polygon safe zones. Figure 12b shows that the running time of HyperEdges in the different safe zone shapes is significantly less than that of Im-Naive. We notice that HyperEges is about three time faster in the round safe zone experiment than in the polygon safe zone experiment. This is expected as for polygon safe zones we need more hyperbolas for each pair of safe zones to constrain the edges created.

B. Spatial Network Setting Experiments

First we validate the effectiveness of HyperEdges in reducing the number of network vertices accessed in both graph construction and query processing stages. Then, we evaluate the algorithm efficiency. The default setting is 192 safe zones (all police stations) where each safe zone has a radius of 2 kilometers and $\alpha = 0$, and the spatial network used is the London road network.

Network Pruning Effectiveness We measure the number of network vertices accessed by both HyperEdges and the naive algorithm and show that HyperEdges accesses a much smaller number of network vertices. This leads to the better performance of HyperEdges in graph construction and query processing.

Figure 13a shows the average number of vertices accessed for each safe zone (query) on the London road network.
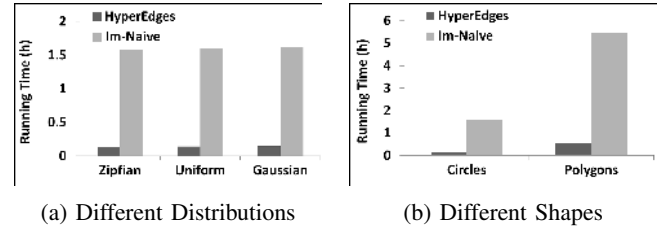

(a) Different Distributions   (b) Different Shapes

Figure 12: Effect of Safe Zone Distribution and Shape

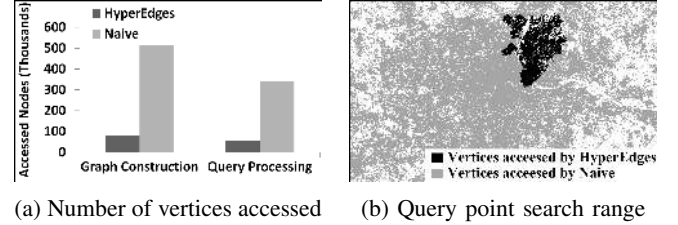(a) Number of vertices accessed   (b) Query point search range

Figure 13: Network Vertices Accessed in London Network

We can see that the average number of vertices accessed by HyperEdges in both graph construction and query processing stages is about 85% less than that of the naive algorithm. Figure 13b illustrates the search range needed by the two algorithms to add a random query point to the constructed graph in the London network dataset. As can be seen from the figure, the naive algorithm accesses about nine times more vertices than HyperEdges does. **Algorithm Efficiency:** We measure the running time in both graph construction and query processing stages using different number of safe zones, safe zone sizes and α values.

1) Effect of Safe Zone Density: We randomly sample the "Police Station" dataset to obtain safe zone datasets of different sizes. Figure 14a shows that the running time of HyperEdges is up to 85% less than that of the naive algorithm in the graph constructing stage. This is due to the advantage of using the hyperbola labeling technique in HyperEdges over the straightforward Dijkstra's algorithm based solution. Figure 14b shows that, the average running time to answer an SPSZ query of HyperEdges is again up to seven times faster than that of the naive solution. An important notice from this figure is that, the performance of HyperEdges improves as the number of safe zones increases, while the performance of the naive solution is almost the same regardless of the number of safe zones. This is because HyperEdges terminates the search earlier when it reaches enough neighboring safe zones, while the naive solution searches the entire network.

2) Effect of Safe Zone Size: We vary the radius of each safe zone from 0.5 to 4 kilometers. As Figure 15a shows, HyperEdges again outperforms the naive algorithm constantly.
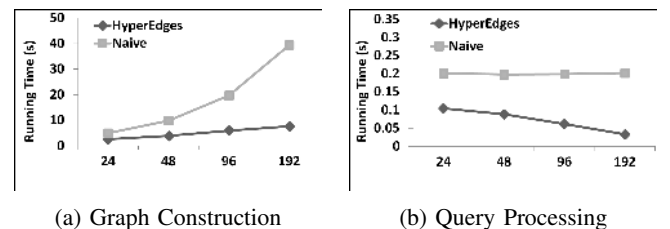

(a) Graph Construction   (b) Query Processing

Figure 14: Effect of the Number of Safe Zones

We notice that when the safe zone size is 4 both algorithms achieve better performance. This is due to overlapping of the safe zones, which leads to fewer safe zones effectively as discussed in Section IV-E. This in turn, reduces the graph construction time.

3) Effect of the Value of α: We further evaluate the effect of the value of α, which controls the weight of the cost of traveling inside safe zones. We vary the value of α in the graph construction stage from 0 to 0.75. As Figures 16a and 16b show, the running time of HyperEdges increases as the value of α increases, which is expected because when the cost of traveling within safe zones has a higher weight, the labeling technique needs to access more vertices to ensure finding the safest path. However, HyperEdges still outperforms the naive algorithm in all cases.

VII. CONCLUSIONS AND FUTURE WORK

We proposed a new path finding problem, safest path via safe zones, which finds the path between two points with the shortest unsafe distance. We modeled the problem in both Euclidean and spatial network settings as a graph shortest path problem and proposed a solution framework, which contains a precomputed graph construction stage and a path finding stage at query time. This framework uses the properties of hyperbolas to prune the search space and reduce the number of edges created. As shown by the experimental study, our algorithm outperforms both the naive and the improved naive baseline algorithms in the Euclidean setting in three aspects. First, the number of edges created by our algorithm before filtering is an order of magnitude smaller than that of the improved naive algorithm. This successfully reduces the memory consumption. Second, the time taken for edge creation by our algorithm is an order of magnitude smaller than that of the improved naive algorithm and two orders of magnitude smaller than that of the naive algorithm. Third, our algorithm in query processing is up to an order of magnitude faster than the naive and Im-Naive algorithms. Similarly, in the spatial network setting, our algorithm consistently outperforms the baseline algorithm in term of the running time and the number of vertices accessed in both graph construction and query processing stages.

For future work we will investigate whether there is a solution to the generalized version of the problem in the Euclidean setting where the cost of traveling in safe zones is added to the total path length as a weighted distance.

VIII. ACKNOWLEDGMENT

(a) Graph Construction   (b) Query Processing

Figure 15: Effect of Safe Zone Size

(a) Graph Construction   (b) Query Processing

Figure 16: Effect of Safe Zone Traveling Cost

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241. 2011.

[2] Jur Berg and Mark Overmars. Planning the shortest safe path amidst unpredictably moving obstacles. In *Algorithmic Foundation of Robotics VII*, pages 103–118. 2008.
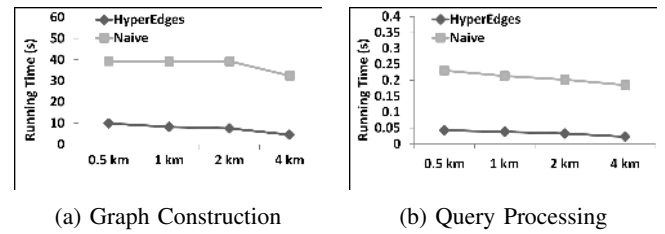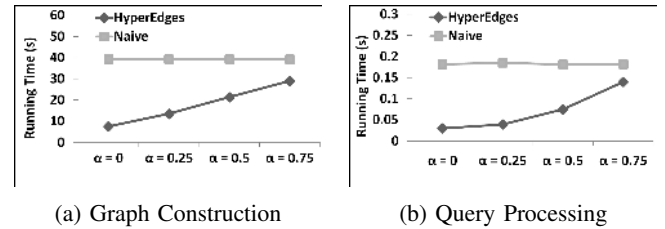
[3] Scott A Bortoff. Path planning for UAVs. In *American Control Conference*, pages 364–368, 2000.

[4] M Eunus Ali, Rui Zhang, Egemen Tanin, and Lars Kulik. A motion-aware approach to continuous retrieval of 3d objects. In *ICDE*, pages 843–852, 2008.

[5] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[6] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *SEA*, pages 319–333. 2008.

[7] A. Gray, E. Abbena, and S. Salamon. *Modern Differential Geometry of Curves and Surfaces with Mathematica*. Chapman and Hall/CRC, 2006.

[8] Christina Hallam, KJ Harrison, and JA Ward. A multiobjective optimal path algorithm. *Digital Signal Processing*, 11(2):133–143, 2001.

[9] RV Helgason, JL Kennington, and KH Lewis. Shortest path algorithms on grid graphs with applications to strike planning. Technical report, DTIC Document, 1997.

[10] HV Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *TODS*, 30(2):364–397, 2005.

[11] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, and Rui Zhang. Approximate nn queries on streams with guaranteed error/performance bounds. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 804–815. VLDB Endowment, 2004.

[12] Alain Lambert, S Bouaziz, and R Reynaud. Shortest safe path planning for vehicles. In *Intelligent Vehicles Symposium*, pages 282–286, 2003.

[13] Alain Lambert and Dominique Gruyer. Safe path planning in an uncertain-configuration space. *Robotics and Automation*, 3:4185–4190, 2003.

[14] L Leenen, A Terlunen, and H Le Roux. A constraint programming solution for the military unit path finding problem. *Mobile Intelligent Autonomous Systems*, 9(1):225–240, 2012.

[15] Shashi Mittal and Kalyanmoy Deb. Three-dimensional offline path planning for uavs using multiobjective evolutionary algorithms. *Congress on Evolutionary Computation*, 7(1):3195–3202, 2007.

[16] Sarana Nutanong, Rui Zhang, Egemen Tanin, and Lars Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB*, 1(1):1095–1106, 2008.

[17] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[18] Suk-Hwan Suh and K.G. Shin. Robot path planning with distance-safety criterion. In *IEEE Conf. on Decision and Control*, pages 634–641, 1987.