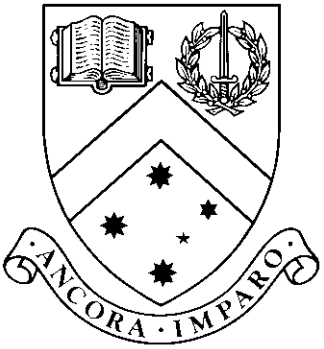
Fast Obstacle Spatial Query Processing on Navigation Mesh

by

Shizhe Zhao



Minor Thesis

Submitted by Shizhe Zhao

in partial fulfillment of the Requirements for the Degree of

Master of Information Technology (Minor Thesis) (3316)

Supervisor: Assoc. Prof. David Taniar

Caulfield School of Information Technology
Monash University

June, 2018

Contents

Fast Obstacle Spatial Query Processing on Navigation Mesh

Shizhe Zhao
szha414@student.monash.edu
Monash University, 2018

Supervisor: Assoc. Prof. David Taniar

Abstract

Obstacle k-Nearest Neighbours problem is the k-Nearest Neighbour problem in a two-dimensional Euclidean plane with obstacles (*OkNN*). Existing and state of the art algorithms for OkNN are based on incremental visibility graphs and as such suffer from a well known disadvantage: costly and online visibility checking with quadratic worst-case running times. In this research we develop a new OkNN algorithm which avoids these disadvantages by representing the traversable space as a collection of convex polygons; i.e. a Navigation Mesh. We then adapt a recent and optimal navigation mesh algorithm, *Polyanya*, from the single-source single-target setting to the the multi-target case. We also give two new and online heuristics for OkNN. In a range of empirical comparisons, we show that our approach can be orders of magnitude faster than competing methods that rely on visibility graphs. The work has been accepted for publication in *Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS'2018), colocated with IJCAI/ECAI'2018, July 2018.*

Keywords: Obstacle Nearest Neighbor, kNN, Navigation Mesh, Spatial Search, Obstacle Distance, Obstacle Navigation

Acknowledgments

Acknowledgement placeholder

Shizhe Zhao

Monash University
June 2018

Chapter 1

Introduction

1.1 Overview

Obstacle k-Nearest Neighbor (OkNN) is a common type of spatial analysis query which can be described as follows: given a set of target points and a collection of polygonal obstacles, all in two dimensions, find the k closest targets to an a priori unknown query point q. Such problems appear in a myriad of practical contexts. For example, in an industrial warehouse setting a machine operator may be interested to know the k closest storage locations where a specific inventory item can be found. OkNN also appears in AI path planning field, for example, in competitive computer games, agent AIs often rely on nearest-neighbour information to make strategic decisions such as during navigation, combat or resource gathering.

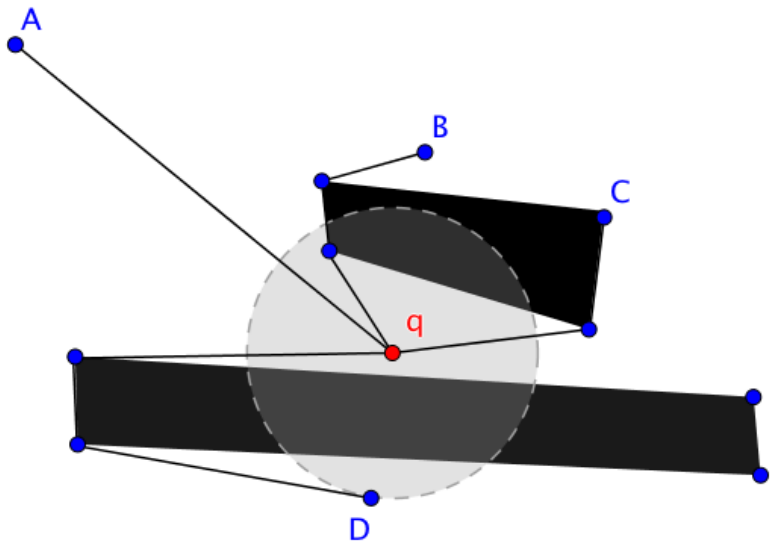


Figure 1.1: We aim to find the nearest neighbour of point q from among the set of target points A, B, C, D. Black lines indicate the Euclidean shortest paths from q. Notice D is the nearest neighbor of q under the Euclidean metric but also the furthest neighbor of q when obstacles are considered.

Traditional kNN queries in the plane (i.e. no obstacles) is a well studied problem that can be handled by algorithms based on a spatial index. A key ingredient to the success of these algorithms is the Euclidean metric which provides perfect distance information between any pair of points. When obstacles are introduced however the Euclidean metric becomes an often misleading lower-bound, and the metric becomes obstacle distance. Figure 1.1 shows such an example.

1.2 Major Challenges

Two popular algorithms for OkNN, which can deal with obstacles, are *local visibility graphs* [27] and *fast filter* [26]. Though different in details, both of these methods are similar in that they depend on the incremental and online construction of a graph of co-visible points, and use *Dijkstra* to compute shortest path. Algorithms of this type are simple to understand, provide optimality guarantees and the promise of fast performance. Such advantages make incremental visibility graphs attractive to researchers and, despite more than a decade since their introduction, they continue to appear as ingredients in a variety of kNN studies from the literature; e.g. [10–12]. However, incremental visibility graphs also suffer from a number of notable disadvantages including:

1. online visibility checks;
2. an incremental construction process that has up to quadratic space and time complexity for the worst case;
3. duplicated effort, since the graph is discarded each time the query point changes.

In section 2.4, we will introduce these two algorithms with detail, and discuss why they have such disadvantages.

1.3 Major Objectives

In this research, we develop a new method for computing *OkNN* which avoids same disadvantages in existing works. Our research extends an existing very fast point-to-point pathfinding algorithm *Polyanya* to the multi-target case.

1.4 Thesis Organisation

The rest of the thesis is organized as follows:

- In chapter 2, we review related works in different areas, includes: AI searching, spatial index and spatial query processing.
- In chapter 3, we introduce the proposed algorithms and discuss their behaviors, formal proof for correctness will be provided.
- In chapter 4, we provide results of experiments to demonstrate the performance of proposed algorithms.
- In chapter 5, we summarize our contributions and discuss future works.

Chapter 2

Literature Review

2.1 Overview

As we've mentioned in the previous chapter, OkNN problem appears in both AI path planning and Spatial query processing. Therefore, this literature review includes related works in these two fields.

In section 2.2, we introduce two classic pathfinding algorithms: *Dijkstra* and *A**, as the historical background. In section 2.3, we introduce a spatial index *R-tree*, and discuss how it solves traditional kNN problem. In section 2.4, we focus on existing works on OkNN, two algorithms based on *Local Visibility Graph* will be discussed. In section 2.5, we introduce a very fast point-to-point algorithm in AI path planning field which shows a new direction to solve OkNN problem. In section 2.6, we briefly discuss other related spatial queries which can be improved by our research.

2.2 Classic pathfinding

The most widely used pathfinding algorithm is *Dijkstra* [9]. The algorithm works on a nonnegative weighted graph, it requires a priority queue and regards the length of shortest path as key, and it visit vertices in the order of length of the shortest path until requirements are satisfied, e.g. the target has been found. When the target is the furthest vertex to the start vertex, *Dijkstra* has to explore the entire map. Based on such consideration, researchers generalized *Dijkstra* algorithm to *best-first search* which explores a graph by expanding the most promising node chosen according to a specified rule. *A** [15] is known as a famous *best-first search*, it select the path that minimizes:

$$f(n) = g\text{-}value(n) + h\text{-}value(n)$$

where n is the last node on the path, *g-value* is the length of the shortest path from start to n, *h-value* is an estimation of the shortest path from n to the goal which is problem-specific. One important property of *h-value* is admissibility, meaning that it never overestimates the actual cost to the target. For example, in an Euclidean plane with obstacles, *h-value* can be the Euclidean distance.

In following sections and the chapter 3, we will show how *Dijkstra* and *A** algorithms be applied on the OkNN problem.

2.3 Spatial Index

2.3.1 *R-tree*

R-tree has many variations [3, 14, 17, 22], they improve efficiency in different aspects, but they still provide the same functionality, so we only introduce the classic *R-tree* in this section.

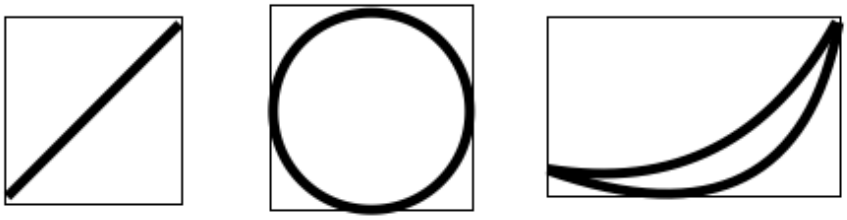
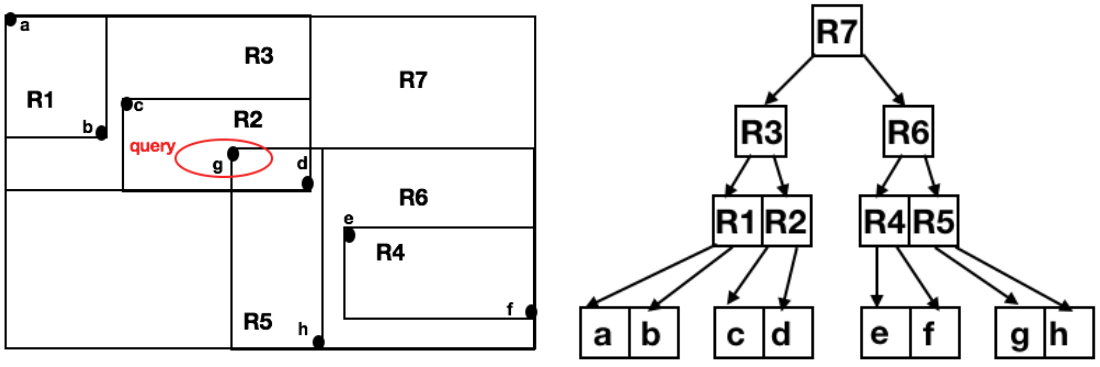


Figure 2.1: Both segments, circle and irregular shape can be represented by their MBR

R-tree is a heigh-balanced tree [14], all objects are stored in a leaf node. In leaf node, if an object is not a point, it would be represented by its *Minimal Bounding Rectangle* (MBR), figure 2.1 shows examples of MBR. Each interior node is also represented by a MBR which contains either leaf nodes or descendant interior nodes. To guarantee efficiency, each non-root node of *R-tree* can contain at least m entries and at most M entries, where m, M are specified constant when *R-tree* is built, and *R-tree*'s root always has two entries. Usually, objects retrieval start from the root, then narrow down to children nodes based on spatial information in their MBRs, and finally retrieve objects from leaf nodes. Figure 2.2 shows how to store and retrieve objects.



(a) Hierachy of MBRs

(b) Corresponding tree structure

Figure 2.2: $\{a, b, c, d, e, f, g, h\}$ is the object set, $R1, R2, R4, R5$ are leaf nodes, $R3, R6$ are interior nodes, and $R7$ is the root. The red oval is a range query, starting with $R7$, since $R6$'s MBR overlapped with query area, we narrow down to $R6$, then to $R5$, and finally retrieve g. Notice that $R3, R2$ also overlap with the query, so they will also be visited, but nothing retrieved.

From the example in figure 2.2, we can see that overlapping area will be explored multiple times in retrieval progress, which duplicated efforts. Thus, some variants use strict non-overlapping interior node (e.g. R^+*-tree* [22]), and non-overlapping *R-tree* is a wide topic in the spatial index field which beyond the scope of this thesis.

2.3.2 Nearest Neighbor Query

In the *R-tree*, all nodes are organized by their spatial information, so that the nearest neighbor of a point can be retrieved by exploring tree nodes in some order. To introduce the algorithm, we need to discuss two metrics: given query point q and the MBR of a tree node

- ***mindist*** is the minimal distance from q to the MBR, it estimates the distance from q to inside object, so this metric is the priority of the tree node;
- ***minmaxdist*** is the upper bound of the NN distance of any object inside the MBR, if the *mindist* of any MBR large than this value, then such MBR cannot contains the nearest object, so this metric is for pruning.

The algorithm starts with root node and proceeds down the tree. When it visits a leaf node, current nearest neighbor will be updated; When it visits a non-leaf node, the children of such node is sorted by *mindist*, and pruned by *minmaxdist*, then algorithm does *depth-first-search* on ordered and pruned children nodes; when algorithm finished, the updated nearest neighbor is the answer, and it can be easily generalized to finding kNN by changes below:

- Tracking k current nearest neighbors, instead of one.
- The pruning is according to the current k-th nearest neighbor.

This kind of algorithm is called *branch-and-bound* traversal, which has been well studied and widely used in other artificial intelligence areas [22], and existing NN queries are based on it with different ordering and pruning strategies, more details are in [4, 21].

2.4 Obstacle k-Nearest Neighbor

In OkNN problem, the metric is obstacle distance, so all existing OkNN algorithms are actually Obstacle Distance Computation (ODC). In following subsections, we introduce these ODC algorithms and show how to apply them on OkNN.

2.4.1 In-main-memory OkNN

Solving obstacle path problems in-main-memory has been well studied [7], these works need to compute visibility graph which any pair of co-visible vertices has an edge, figure 2.3 shows an example.

Since all edges of the shortest path belong to visibility graph [18], once precomputed it and include visible edges from query point, we can run shortest path algorithm (e.g. *Dijkstra*) to find k-nearest neighbor. However, precomputing visibility graph(VG) is costly: even the best algorithm [13] has $O(m + nlogn)$ runtime, where n is the number of vertexes and m is the number of edges, and in practice, m can reach to $O(n^2)$. In spatial database scenario, n can be more than $10,000$, so in-main-memory approach is not suitable for spatial database scenario.

2.4.2 Local Visibility Graph

Since building global VG is infeasible, researchers in spatial database field are motivated to design an algorithm that only considers query related area.

In 2004, Zhang proposed the *Local Visibility Graph* (LVG) algorithm [27] to compute obstacle distance. Assume obstacles are stored in *R-tree*, given query point q, and a target t, the algorithm runs in following steps:

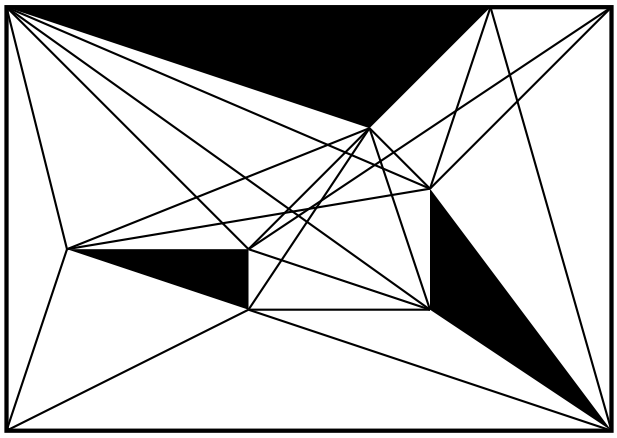
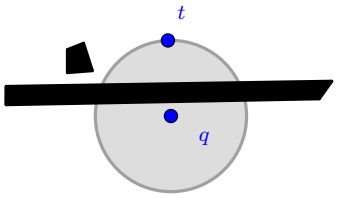
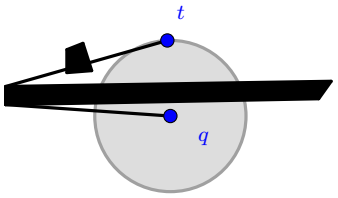


Figure 2.3: The rectangle is the boundary of the map, black polygons are obstacles and all black lines are edges in the visibility graph.

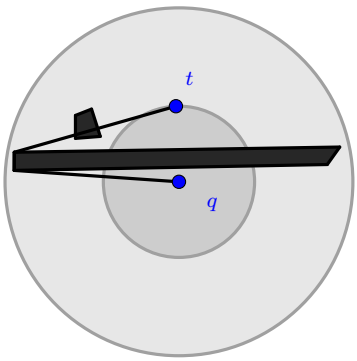
1. It starts with a small VG centered on q with radius $r = d_e(q, t)$ (figure 2.4a);
2. Then compute shortest path on the current VG (figure 2.4b);
3. Enlarge the circle to current obstacle distance $r = d_o(q, t)$, update the VG incrementally (figure 2.4b) and recompute the shortest path (figure 2.4d);
4. Repeat the previous step until $r >= d_o(q, t)$.



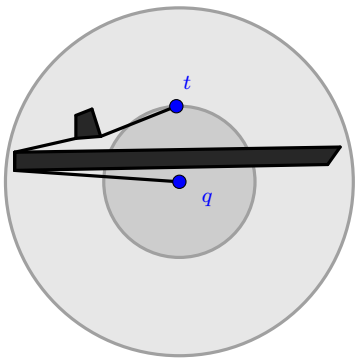
(a) the long rectangle obstacle in the circle is retrieved and included in vg.



(b) current shortest path may be blocked by some obstacles outside



(c) enlarges the circle and updates the vg



(d) recomputes the shortest path

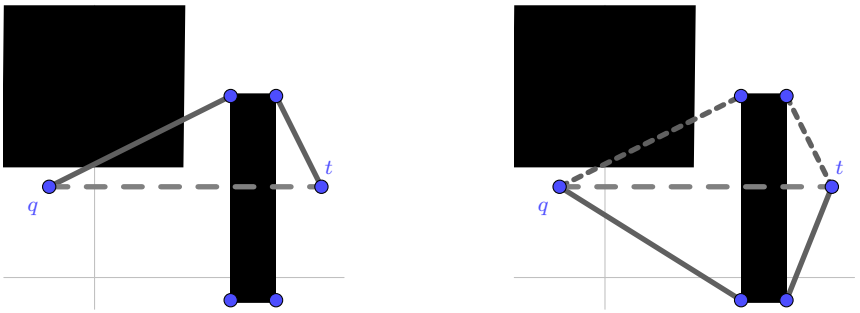
Figure 2.4: LVG algorithm

Since $r >= d_o(q, t)$, when algorithm finish, it guarantees that no obstacle on the current shortest path, and thus proving the correctness. This algorithm can be extended to multi-target scenario to solve OkNN problem:

- Initially, t is the *k-th* nearest neighbor in Euclidean space, so that it guarantees that the VG always contains at least k targets;
- Terminate when r not less than the current *k-th* nearest distance;

2.4.3 Fast filter

There's another similar work proposed by Xia [26], the difference is, instead of considering obstacles in a circle area, it only retrieves obstacles on the current shortest path, updates the LVG and recomputes the shortest path. Since fewer obstacles are involved in VG, the algorithm is called *Fast Filter*. Figure 2.5 shows an example.



(a) (b)

Figure 2.5: Black polygons are obstacles, q is query point, t is the target. Initially, the VG only contains the rectangle obstacle between the qt, and the corresponding shortest path is computed (fig 2.5a). Then retrieves the square obstacle on the current path, updates VG and recomputes the shortest path (fig 2.5b).

To solve OkNN, we need following changes:

- initially compute obstacle distance for k nearest neighbors, and store results in *max-heap* with size k;
- keep retrieving next NN and compute the obstacle distance d_o;
- when d_o not large than the top value of heap, pop top and insert the current d_o;
- terminate when the Euclidean distance to current NN is large than the Obstacle distance on top of the heap;

2.4.4 Discussion

When search space is large (e.g. target is far from the query point), *fast filter* [26] is more efficient because it only considers obstacles that might on the path, meanwhile *LVG* [27] will build visibility graph for a large area, which is slow. In a multi-target scenario, when k is large, *LVG* [27] is more efficient because *Dijkstra* is a natural single-source algorithm, meanwhile, each time *fast filter* [26] can only compute the path for a single target, so that it duplicates effort for common prefixes of k targets.

Both *LVG* [27] and *Fast filter* [26] are simple to understand, provide optimality guarantees and the promise of fast performance. Such advantages make them attractive to researchers and, despite more than a decade since their introduction, they continue to appear as ingredients in a variety of kNN studies from the literature; e.g. [10, 11]. However,

these visibility graph based algorithms also suffer from a number of notable disadvantages including:

(i) costly online visibility checks;

(ii) an incremental construction process that has up to quadratic space and time complexity for the worst case;

(iii) duplicated efforts are unavoidable, since the graph is discarded each time the query point changes.

2.5 Pathfinding on Navigation Mesh

2.5.1 Historical background

Because of the limitation of VG, *Navigation Mesh* comes to our sight. Ronald first proposed this concept in 1986 [2], then it is applied on robotics and game pathfinding.

Navigation mesh divides traversable space into convex polygons, the convexity of mesh guarantees that all insides points are co-visible so that it not needs visibility checking, Figure 2.6 shows an example. Compare to VG (fig 2.3), it can be generated by *Constrained Delaunay Triangulation* [5] in $O(nlogn)$ times, so that it is feasible to preprocess the entire map. Additionally, adding or removing obstacles only needs to modify a few numbers of mesh polygons, which is very flexible. It seems like a perfect framework for OkNN, the only problem is that how to compute obstacle distance on it.

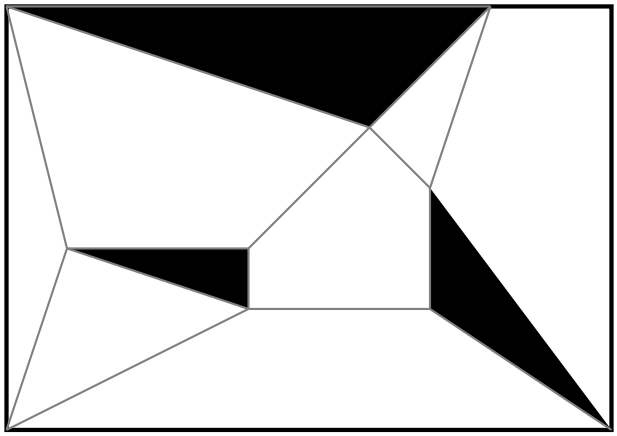


Figure 2.6: Navigation Mesh: The rectangle is the boundary of the map, black polygons are obstacles, gray lines are edges of mesh polygons.

Previous pathfinding algorithms on navigation mesh are not suitable for spatial query processing, and thus it's not attractive to researchers in this fields. Three widely used algorithms are:

- Channel Search [16]: first find an abstract path from start to target composed of polygons, then refine the abstract path to a sequence of concrete points. This algorithm only generate an approximately shortest path, the lack of optimality makes it not suitable for OkNN query.

- TA* [8]: similar to Channel Search, but it can repeat the search process until finding an optimal shortest path. The repeating is time-consuming, so it is not suitable for query processing.

- TRA* [8]: similar to TA*, but TRA* can utilize preprocessing to speed up pathfinding. The problem is that the precomputed information will be invalid when the environment change, so it is not suitable for database scenario.

However, this fact has been changed by a recent work called *Polyanya* [6]. It is a fast, optimal and flexible pathfinding algorithm which is perfect for query processing in spatial database.

2.5.2 Polyanya

Polyanya can be seen as an instance of *A**: it performs a best-first search using an admissible heuristic function to prioritize nodes. The mechanical details are however quite different, there are three key components:

- **Search Nodes**: Conventional search algorithms proceed from one traversable point to the next. *Polyanya*, by comparison, searches from one *edge* of the navigation mesh to another. In this model, search nodes are tuples (I, r) where each $I = [a, b]$ is a contiguous interval of points and r is a distinguished point called the *root*. Nodes are constructed such that each point $p \in I$ is visible from r. Meanwhile, r itself corresponds to the last turning point on the path: from q to any $p \in I$. Figure 2.7 shows an example.

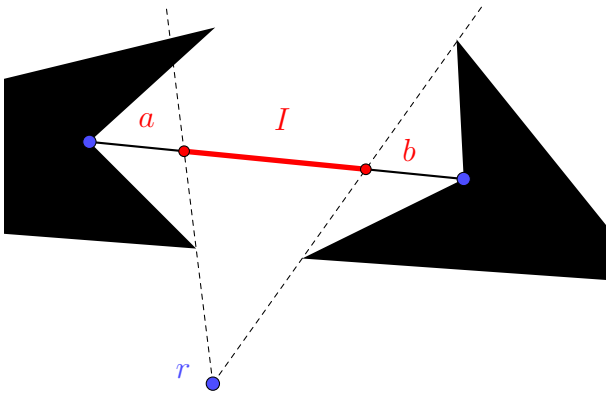


Figure 2.7: Search nodes in Polyanya. Notice that the interval $I = [a, b]$ is a contiguous subset of points drawn from an edge of the navigation mesh. The corresponding root point, r, is either the query point itself or the vertex of an obstacle. Taken together they form the search node (I, r).

- **Successors**: Successor nodes (I', r') are generated by "pushing" the current interval I away from its root r and throught the interior of an adjacent and traversable polygon. A successor is said to be *obervable* if each point $p' \in I'$ is visible from r. The successor node in this case is formed by the tuple (I', r). By contrast, a successor is said to be *non-observable* if the *taut* (i.e. locally optimal) path from r to each $p' \in I'$ must pass through one of the endpoints of current interval $I = [a, b]$. The successor node in this case is formed by the tuple (I', r') with r' as one of the points a or b. Figure 2.8 shows an example.

  Note that the target point is inserted in the open list as a special case (observable or non-observable) successor whenever the search reaches its containing polygon. The interval of this successor contains only the target.

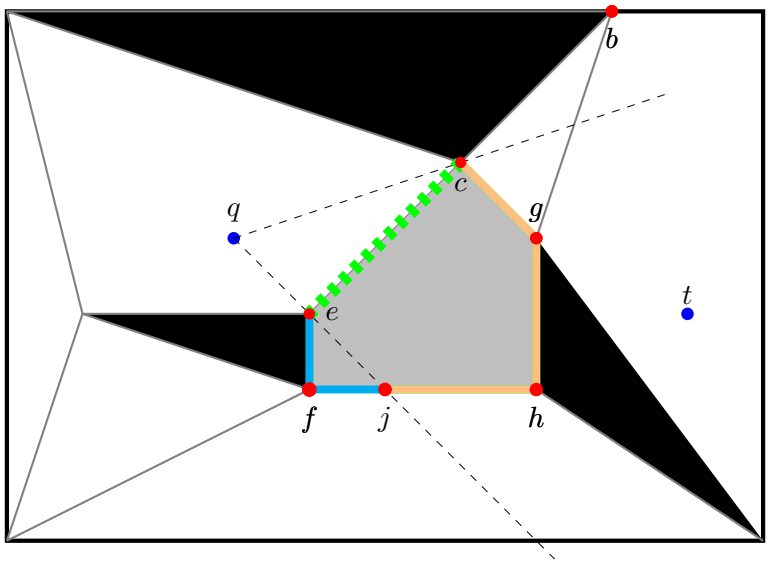


Figure 2.8: We expand the node $([e, c], q)$ which has $([c, g], q)$ and $[h, j], q$ as observable successors. In addition, the nodes $([e, f], e)$, $([f, j], e)$ are non-observable. All other potential successors can be safely pruned (more details in [6]).

- **Evaluation**: When prioritising nodes for expansion, *Polyanya* makes use of an *f-value* estimation function which bounds the length of the optimal path: from q, throught the current node (i.e. via some $p \in I$) and onto the target. There are three cases to consider which describe the relative positions of the target in relation to the current node. These are illustrated in Figure 2.9. The objective in each case is to choose the unique $p \in I$ that minimise the estimate. The three cases together are sufficient to guarantee that the estimator is admissible.

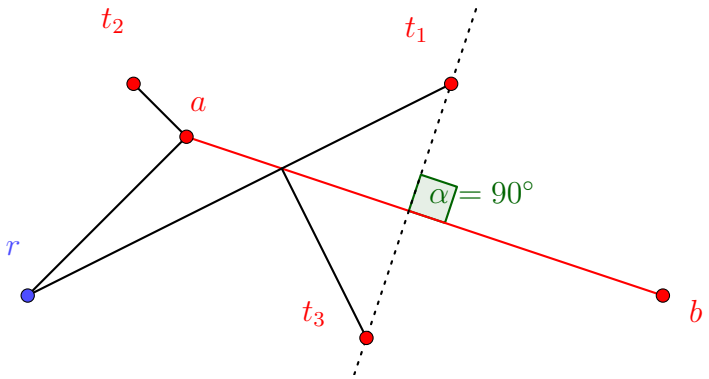


Figure 2.9: Polyanya f-value estimator. The current node is (I, r) with $I = [a, b]$ and each of t_1, t_2, t_3 are possible target locations. **Case 1**: the target is t_1. In this case the point $p \in I$ with minimum f-value is at the intersection of the interval I and the line $r \to t_1$. **Case 2**: the target is t_2. In this case the $p \in I$ with minimum f-value is one of two endpoints of I. **Case 3**: the target is t_3. In this case the $p \in I$ with minimum f-value is obtained by first mirroring t_3 through $[a, b]$ and applying Case 1 or Case 2 to the mirrored point (here, t_1). Notice that in this case, simply r to t_3 doesn't give us the *h-value*, based on definition, it must reach the interval first.

Similar to *A**, *Polyanya* terminates when the target is expanded or when the open list is empty. Extending it to multi-target OkNN is not a trivial problem, we will discuss this in chapter 3.

2.6 Other Obstacle Spatial Queries

Obstacle spatial query processing is a broad research area, and many existing works are still based on visibility graph, in this section, we review those works which can get benefit from OkNN in our research.

- **Obstacle Range Query**(OR): given the query point q and a range r, it returns all targets which obstacle distance to q are less or equal to r [27]. To solve this by OkNN, we can let k be infinite and terminate the algorithm when current obstacle distance large than r.
- **Obstacle Reverse k-Nearest Neighbor**(ORkNN): it is proposed in 2011 [11] for $k = 1$, and be generalized to $k > 1$ in 2016 [10]. ORkNN given query point q and k, return a set of targets which regards q as its OkNN: $\{t|q \in OkNN(t, k)\}$. The query processing has two stages: (i) *search stage*: explore search space to get a set of candidates; (ii) *refine stage*: calls OkNN for each candidate, remove a candidate if q is not its OkNN. The first stage needs to compute obstacle distance to prune search space, and the second stage can directly get benefit from the improvement of OkNN.
- **Continuous Obstacle k-Nearest Neighbor**(COkNN): it's proposed in 2009 [12], similar to OkNN, but the query becomes a segment. The algorithm first generates a list of "split" points, and reduce the problem to finding the OkNN for these points. It also needs to compute obstacle distance to prune search space.

2.7 Summary

In this chapter, we've reviewed works that are related to our research. We firstly introduced the background knowledge by discussingA^* and *Dijkstra*. Then we discussed existing works basically in three fields: Spatial Index, Spatial Query Processing and AI Pathfinding. The major finding is that, in OkNN problem, existing VG based algorithms are hard to improve, so we are motivated to look for a new framework, meanwhile, the new work in AI Pathfinding field shows us a new direction.

Chapter 3

Proposed Algorithms

3.1 Overview

From the previous chapter, we have introduced a very fast point-to-point algorithm *Polyanya*, in this chapter, we discuss how to effectively adapt *Polyanya* for OkNN settings where there are multiple candidate targets. In section 3.2, we introduce formal problem statement and math notations; in section 3.3, we introduce two less efficient but very straightforward solution to show the motivations of our proposed research; section 3.4 and 3.5 present our research works which discuss the design of our algorithms and the correctness in theory.

3.2 Problem Statement

OkNN is a spatial query in two dimensions that can be formalised as follows:

Definition 3.2.1. Obstacle k-Nearest Neighbour (OkNN):
Given a set of points T, a set of obstacles O, a distinguished point q and and an integer k:
return a set kNN $= \{t|t \in T\}$ such that $d_o(q,t) \leq d_o(q,t_k)$ for all $t \in$ kNN.

Where:

- O is a set of non-traversable polygonal obstacles.
- T is a set of traversable points called *targets*.
- q is a traversable point called the *query point*.
- k is an input parameter that controls the number of nearest neighbours that will be returned.
- d_e and d_o are functions that measure the shortest distance between two points, as discussed below.
- t_k is the k^{th} nearest neighbour of q.
- $h_p(n,t)$ is the *h-value* in Polyanya for a given search node n and a target t.
- $h_v(n)$ is the *h-value* in Interval Heuristic for a given search node n.
- $h_t(n)$ is the *h-value* in Target Heuristic for a given search node n.

Stated in simple words, the objective is to find the set of k targets which are closest to q from among all possible candidates in T. When discussing distances between two points q and t we distinguish between two metrics: $d_e(q,t)$ which is the well known Euclidean metric (i.e. "straight-line distance") and $d_o(q,t)$ which measures the length of a shortest path $\pi_{q,t} = \langle q, \ldots, t \rangle$ between points q and t such that no pairwise segment of the path intersects any point inside an obstacle (i.e. "obstacle avoiding distance").

3.3 Motivation

Since *Polyanya* instantiates $A*$ search and since that algorithm is itself a special case of *Dijkstra*'s well know technique, there exists a simple modification at hand: we can simply remove the influence of the cost-to-go heuristic and allow the search to continue until it has expanded the k^{th} target, let's call this ***zero-heuristic***. All other aspects of the algorithm, including termination [1], remain unchanged.

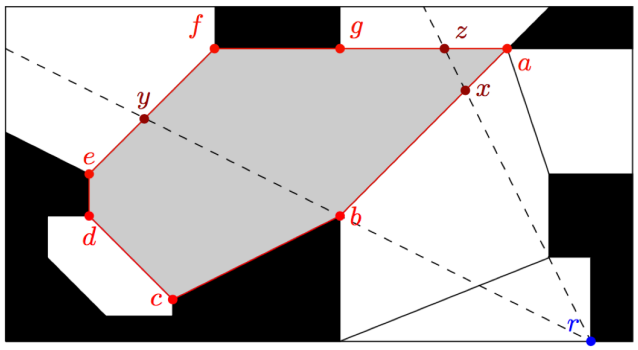


Figure 3.1: Example of successors from [6].

The version of *Polyanya* we have just described is unlikely to be efficient. Without a heuristic function for guidance, nodes can only be prioritized by the *g-value* of their root point, which is settled at the time of expansion. However, the *g-value* does not reflect the distance between the root and its corresponding interval. For example, in Figure 3.1, all observable successors would have the same expansion priority. Thus we may expand many nodes, all equally priomising but having distant intervals, and all before reaching a nearby target node with a slightly higher *g-value*.

Another naive adaption is repeatedly calling an unmodified point-to-point *Polyanya* search, from the query point and to each target, let's call this ***brute-force Polyanya***, see in algorithm 1. It is obvious that this solution is inefficient when targets are many, however, in chapter4 we will see it outperforms other proposed algorithms in certain contexts. To

Algorithm 1: Brute-force Polyanya

```
1 foreach t in targets do
2 |   polyanya.run(start, t);
3 end
```

deal with this problem we develop two online heuristics which can be fruitfully applied to OkNN:

- The Interval Heuristic, which prioritizes nodes using the closest point from its associated interval.
- The Target Heuristic, which relies on a Euclidean nearest neighbour estimator to provide a target dynamically at the time of expansion.

Each of these heuristic is applied in the usual way compute a final expansion priority: $f(n) = g\text{-}value(n) + h\text{-}value(n)$. In the remainder of this chapter we explore these ideas in turn.

[1] There are two cases to consider depending on whether the query and target points are in the same polygon or in different polygons. Both are described in [6]

3.4 Interval Heuristic

In some OkNN settings targets are myriad and one simply requires a fast algorithm to explore the local area. This approach is in contrast to more sophisticated methods which apply spatial reasoning to prune the set of candidates. The idea we introduce for such settings is simple and can be formalised as follows:

Definition 3.4.1. Given search node (I, r), the Interval Heuristic computes a value $h_i(I, r)$ which is the minimal Euclidean distance from the root r to the segment I.

Applying the Interval Heuristic h_i requires solving a simple geometric problem: finding the closest point on a line. The operation has low constant time complexity and we apply standard techniques. Algorithm 2 shows an example.

Algorithm 2: Polyanya OkNN with interval heuristic

```
   Result: k-nearest neighbor
 1 init_search();
 2 find_final_nodes_case1();
 3 while heap not empty do
 4 |   node = heap.pop();
 5 |   if node is final node then
 6 |   |   result.add(node);
 7 |   |   if result.size is k then
 8 |   |   |   return resut;
 9 |   |   else
10 |   |   |   continue;
11 |   |   end
12 |   else
13 |   |   find_final_nodes_case2();
14 |   |   successors = genSuccessors(node);
15 |   |   foreach suc in successors do
16 |   |   |   suc.gValue = node.gValue + dist(node.root, suc.root);
17 |   |   |   suc.hValue = dist_to_segment(suc.r, suc.I);
18 |   |   |   suc.fValue = suc.gValue + suc.hValue;
19 |   |   |   heap.push(suc);
20 |   |   end
21 |   end
22 end
```

Theorem 3.4.1. **(consistency)**: The *f-value* of successor node is not less than the *f-value* of the parent search node.

Proof. Using the interval heuristic, when the successor is a search node, its *f-value* can be interpreted as a lower-bound of the length of path from s to any point on the segment I through root r, and since it is generated by pushing away from the parent search node, its *f-value* is larger than *f-value* of the parent search node. If successor is a target node, the *f-value* is the the length of the corresponding path and not less than the parent search node (the two values are equal when the target is on the segment I). □

Corollary 3.4.2. Expanding a target node corresponds to finding a shortest path.

Proof. As per Theorem 3.4.1, when a final node is expanded there exists no remaining candidate on the open list which can reach the node with a smaller f-value. □

3.5 Target Heuristic

In some OkNN settings the set of target are few (i.e. sparse), or there is a filter on the query, for example, the query is like "the nearest storage location where capacity >= 100". In these cases, without a reasonable heuristic guide, it is possible to perform many redundant expansions in areas where no nearest neighbor can exist, Figure 3.2 shows an example. In such cases more sophisticated spatial reasoning can help to prune the set

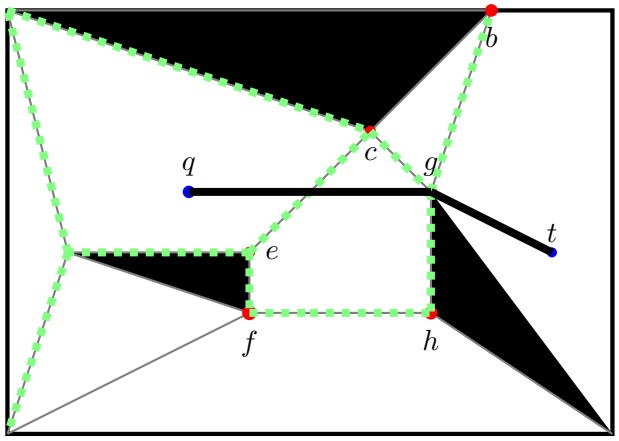


Figure 3.2: Search space of Interval Heuristic:q is query point, t is a target, green dashed segments are the interval of expanded search nodes. From the figure we can see that the algorithm does unnecessary expansions in the direction that no target.

of nearest neighbours and guide the search. The idea we introduce for such settings can be formalised as follows:

Definition 3.5.1. The closest target t of a search node n is $t \in T$ where $h_p(n,t)$ is minimum, the *h-value* of target heuristic is $h_t(n) = h_p(n,t)$.

We implement this idea as follow: When the relative location between targets and r are in case 3 of the h_p, instead of flipping targets, we flip r, and thus formed six areas as shown in Figure 3.3. For $t \in areaA$, the *h-value* is $d_e(r,a) + d_e(a,t)$; for $t \in areaA'$, the *h-value* is $d_e(r',a) + d_e(a,t)$, and because $d_e(r',a) = d_e(r,a)$, we can combine $areaA$ and $areaA'$, so we need to find the nearest neighbor of a for $t \in areaA \cup areaA'$; by the same reason, we can combine $areaB, areaB'$, so finally we formed four areas. Then following the example, we may reason as follows:

- Suppose the next nearest target t is in $areaA \cup areaA'$ (equiv. $areaB \cup areaB'$). Then the optimal path must pass through the point a (equiv. b) so the minimum h-value can be computed as: $\mathbf{min}\{d_e(r,a) + d_e(t,a)\}$ such that t in $areaA \cup areaA'$ (equiv. $areaB \cup areaB'$).

- Alternatively, suppose the next nearest target t is instead in $areaC$. Then the optimal path must pass through a point p in the open interval (a,b). So the minimum h-value can be computed by minimising across all target points in $areaC$. A similar argument applies to a next nearest neighbour in $areaC'$ and we can apply the same strategy, but only after mirroring the root point r through the interval. This operation is in contrast to the heuristic used by Polyanya in the point-to-point setting, which mirrors the target through the interval.

Identifying the candidate target with minimum h-value in each of the four cases can be improved, from a linear-time operation to NN query, by storing all of the targets in a spatial data structure such as R^*-Tree [3]. Thus we may compute a lower-bound estimate

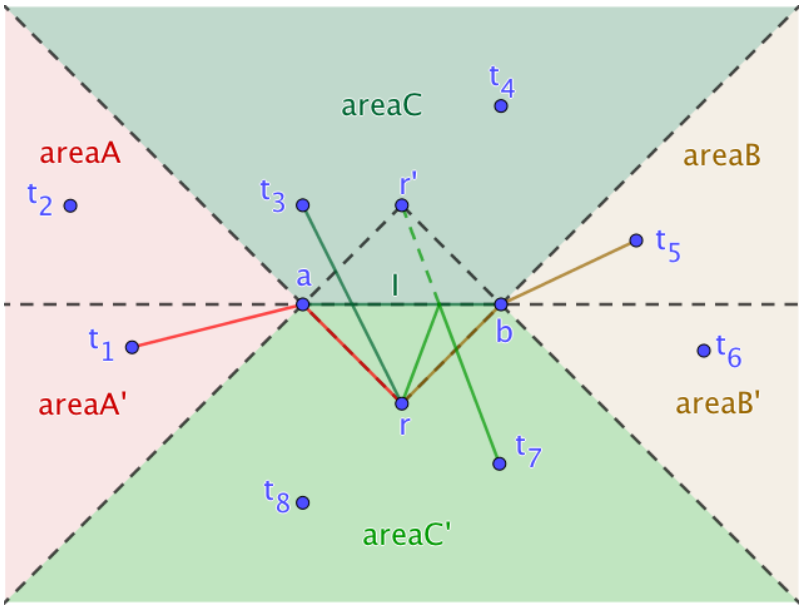


Figure 3.3: **(i)** In $areaA \cup areaA'$, t_1 is the nearest to a; **(ii)** in $areaB \cup areaB'$, t_5 is the nearest to b; **(iii)** in $areaC$, t_3 is the nearest to r; **(iv)** in $areaC'$, t_7 is the nearest to r';

to the next nearest neighbour by minimising over four candidates returned by the R^*-tree instead of evaluating all possible target points.

3.5.1 Further Refinements

We may notice that the Target Heuristic described thus far is potentially costly, when compared to one constant time operation in h_p. To mitigate this we could call h_t less often. An observation is that a parent search node and its successor may use same closest target t in their h_t. In this case, instead of running a new query, the successor can directly inherit the t from the parent. We call this strategy *lazy compute* and apply it throughout our experiments. We find it reduces total running time by approximately 20%.

Lemma 3.5.1. Given search node $n = (I, r)$, its successor $n' = (I', r')$, and a target t that minimises Euclidean distance to n from among all possible candidates. Further suppose h_p(n, t) = h_p(n', t) . Then t is also a target that minimises Euclidean distance to n' among all possible candidates.

Proof. If there is t' closer to n' than t, then $h_p(n', t') < h_p(n', t) = h_p(n, t)$, and because of Lemma 3.4.1, $h_p(n, t') <= h_p(n', t')$, so, t can't be the closest target of n, which is conflict with assumption. Thus, such t' doesn't exist. □

Now, each search node has a target, and the search behavior should be broadly similar to the point-to-point setting. But there is one significant difference: when a nearest neighbor t has been found, t should no longer influence the search process. Thus, we need to remove t from search space and re-assign (i.e. update) all search nodes in the queue which use t as their closest target. To avoid exploring the entire queue we propose instead the following simple strategy: when such a node is dequeud from the open list, we apply h_t to compute a new target and we push the node back onto open all without generating any successors. We call this *lazy reassign*.

Lemma 3.5.2. Let's call these search nodes who need reassignment **pseudo nodes**, and others **real nodes**. **Lazy reassign** never changes the relative order of real nodes in queue.

Proof. Let n_1, n_2 be any pair of real nodes, and s be any pseudo node. After the reassignment, if s become neither n_1 nor n_2, then inserting a third party search node has nothing to do with the relative order of n_1 and n_2; otherwise, without the loss of generality, assume s becomes n_1. If the relative order of them is $< n_2, s >$, then $f(n_2) <= f(s)$, and because of the definition 3.5.1, we have $f(s) <= f(n_1)$, so relative order of n_1, n_2 doesn't change. Alternatively, if the relative order is $< s, n_2 >$, then n_1 will be push to queue before n_2 pop out, so the relative order of n_1, n_2 doesn't change as well. □

In Algorithm 3, we arrive at last at the final form of Polyanya OkNN. The algorithm accepts either h_v and h_t as a heuristic function and has, in both cases, the same high level steps.

Algorithm 3: Polyanya OkNN

```
   Result: k-nearest neighbor
 1 init_search();
 2 find_final_node_in_initialization();
 3 whileheap not empty node = heap.pop();
 4 if node is final node then
 5 |   result.add(node);
 6 |   if result.size is k then
 7 |   |   return result;
 8 |   else
 9 |   |   continue;
10 |   end
11 else
12 |   if node.target has been found then
13 |   |   node.target = get_closest_target(node);
14 |   |   node.hValue = get_h_value(node, node.target);
15 |   |   node.fValue = node.hValue + node.gValue;
16 |   |   heap.push(node);
17 |   |   continue;
18 |   else
19 |   |   find_final_node_in_search();
20 |   |   successors = genSuccessors(node);
21 |   |   foreach suc in successors do
22 |   |   |   suc.gValue = node.gValue + dist(node.root, suc.root);
23 |   |   |   hValue = get_h_value(suc, node.target);
24 |   |   |   if fabs(hValue - node.hValue) <= EPS then
25 |   |   |   |   suc.target = node.target;
26 |   |   |   |   suc.hValue = hValue;
27 |   |   |   else
28 |   |   |   |   suc.target = get_closest_target(suc);
29 |   |   |   |   suc.hValue = get_h_value(suc, suc.target);
30 |   |   |   end
31 |   |   |   suc.fValue = suc.gValue + suc.hValue;
32 |   |   |   heap.push(suc);
33 |   |   end
34 |   end
35 end
```

3.6 Summary

In this chapter, we start with two straightforward adaptions of point-to-point Polyany and finally proposed two heuristics for the Polyanya OkNN algorithm. For *Interval Heuristic*, it supports multi-target by removing t from *h-value*; for *Target Heuristic*, it supports multi-target by employing spatial index and computing closest target dynamically.

Chapter 4

Empirical Analysis

4.1 Overview

To evaluate Polyanya OkNN we consider two distinct setups and one large map with containing 9000 polygonal obstacles (this benchmark is described further in the next section).

In the first setup, targets are numerous and densely distributed throughout the map. Our principal point of comparison in this case is *LVG* [27] which is a state of the art method based on incremental visibility graphs. In the second setup, targets are few and sparsely distributed. Our principal point of comparison in this case is *brute-force Polyany*. We motivate these decisions as follows:

- When the map is large and targets are many (commonly the case in spatial database settings) *LVG* considers only a small part of map and so its query processing can be very fast. *Brute-force Polyany* meanwhile is infeasible to run (there are too many searches).
- When the map is large targets are few (<= 10) *LVG* builds a visibility graph for almost entire map, and ends up with quadratic runtime complexity, which is unacceptable. Meanwhile, brute force *Polyanya*, which is a very fast point-to-point pathfinding algorithm, can be competitive even when called repeatedly. This comparison is motivated by recent prior work involving kNN queries on road networks [1], where other fast point-to-point algorithms were shown to provide state-of-the-art performance in multi-targets scenarios, even when compared against dedicated kNN algorithms.

In experiments, we examine performance based on elapsed time and expanded search node. For *LVG*, we count the number of expanded node in *Dijkstra*.

The navigation mesh of map is generated by *Constrained Delaunay Triangulation*, which is $O(nlogn)$; the implementation of such algorithm is in library *Fade2D* [1], the total time on such preprocessing is about 6s. We implemented in C++ the LVG algorithm (more details below) as well as two versions of multi-target Polyanya: one each for Target and Interval Heuristic. Our code is compiled with *clang-902.0.39.1* using *-O3* flag, under *x86_ 64-apple-darwin17.5.0* platform. All of our source code and test data set are publicly available [2] All experiments are performed on a 2.5 GHz Intel Core i7 machine with 16GB of RAM and running OSX 10.13.4.

[1] http://www.geom.at/fade2d/html

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

4.2 Implementation of *LVG*

As discussed *LVG* is our primary competitor in experiments with dense targets. However, since there is no publicly available implementation, we write one ourselves. We implement the method as per the description in the original paper [27]. The only differences is in construction of the visibility graph: *LVG* uses the rotational plane-sweeping algorithm from [23] which runs worst case $O(n^2 logn)$ time. In our work we opted to simplify development complexity and use a R^*-tree [3] query for visibility checking. Each such query runs in $log(n)$ time and we perform one check for each unique pair of vertices. Thus the total complexity to build a visibility graph is also $O(n^2 logn)$. This implementation, as with the rest of our code, is made publicly available.

The implmentation of R^*-tree we use is also publicly available [3], and appears in other recently published work [25].

4.3 Benchmark

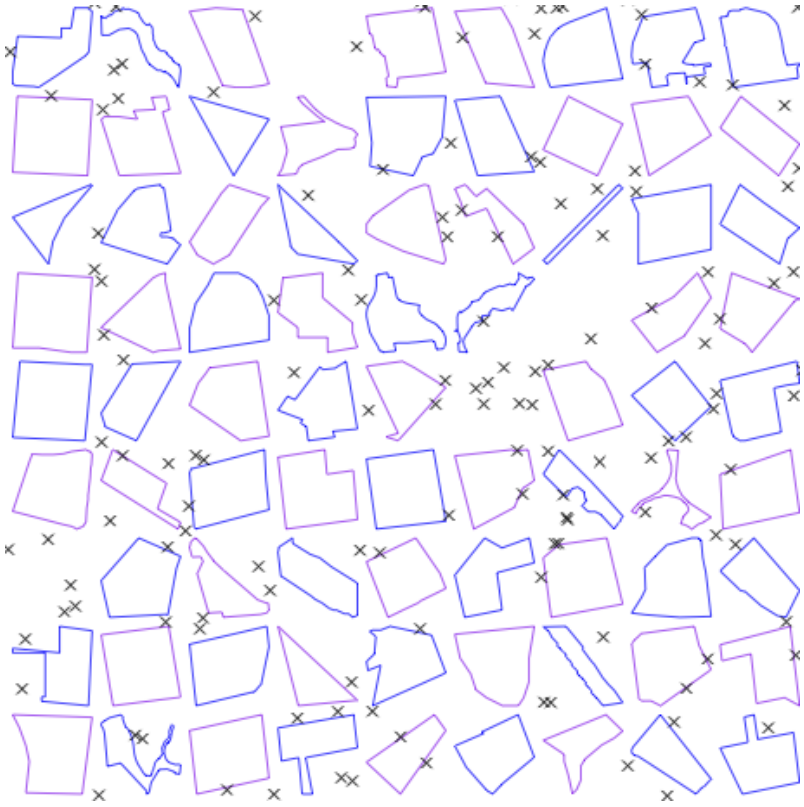


Figure 4.1: Example: a generated map with many targets. Color border polygons are obstacles, black crosses are targets.

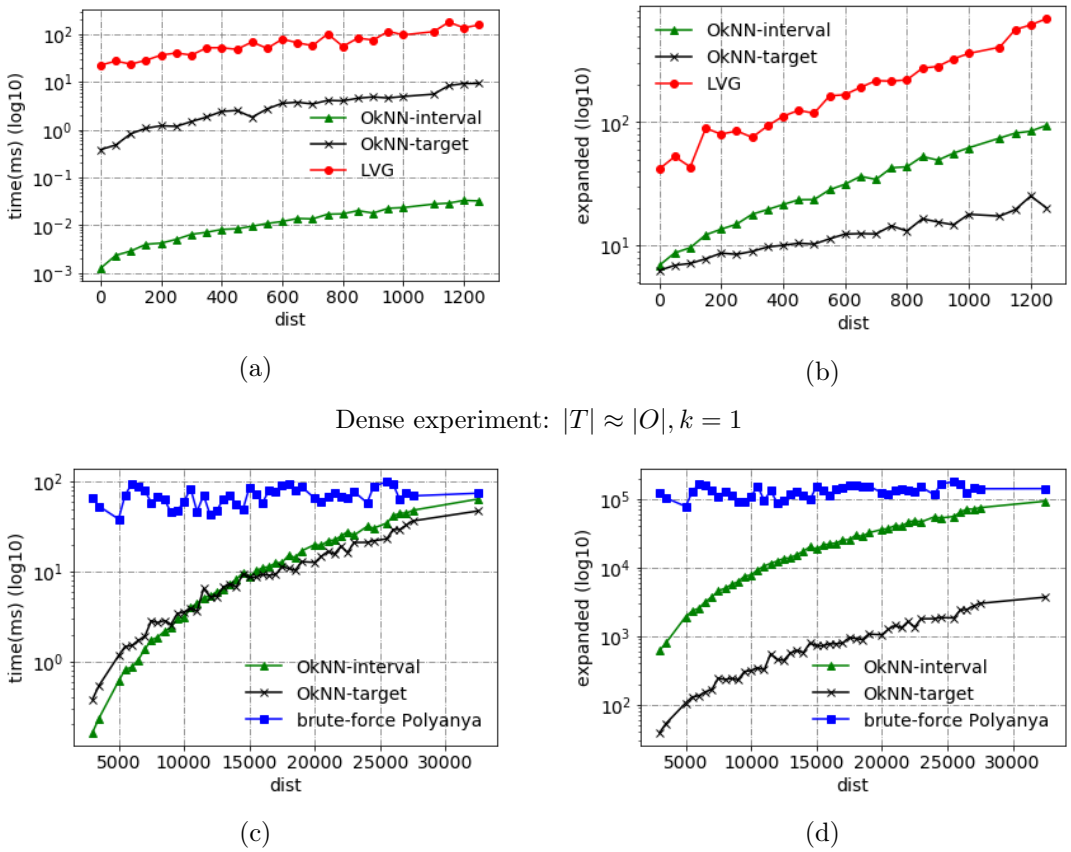
The data set from our main competitor LVG [27] is no longer available on the public Internet so we opt to generate new benchmark problems. We extract the shape of all parks in Australia from *OpenStreetMap* [20] and use these shapes as polygonal obstacles. There are about 9000 such polygons in total. Next we generate a map by tiling all obstacles in the empty square plane. For the tiling, we first divide the square plane into grid having $\lceil \sqrt{|O|} \rceil$ number of rows and columns. Then we assign each polygon to a single grid cell and normalize the shape of polygon by to fit inside the cell. Figure 4.1 gives an example

[3] https://github.com/safarisoul/ResearchProjects/tree/master/2016-ICDE-RkFN

a map generated in this way. For each experiment, we're using 1000 random query points, grouping results by *x-axis*, and computing average; the size of each bucket is at least 10.

One thing needs to be highlighted is that, unlike *LVG* [27] where obstacles are always rectangular, we consider polygons of arbitrary shape, which is more realistic and potentially more challenging as there are more vertices to consider. The total number of vertices across all polygons is more than 100,000.

4.4 Experiment 1: lower bounds on performance



(a) (b)

Dense experiment: $|T| \approx |O|, k = 1$

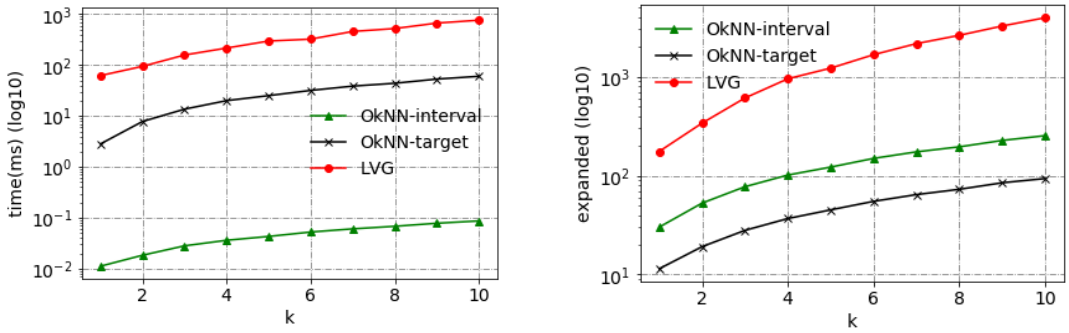
(c) (d)

Sparse experiment: $|T| = 5, k = 1$

Figure 4.2: Experiment1: examine the performance when *dist* increase, where *dist* is the obstacle distance to retrieved target

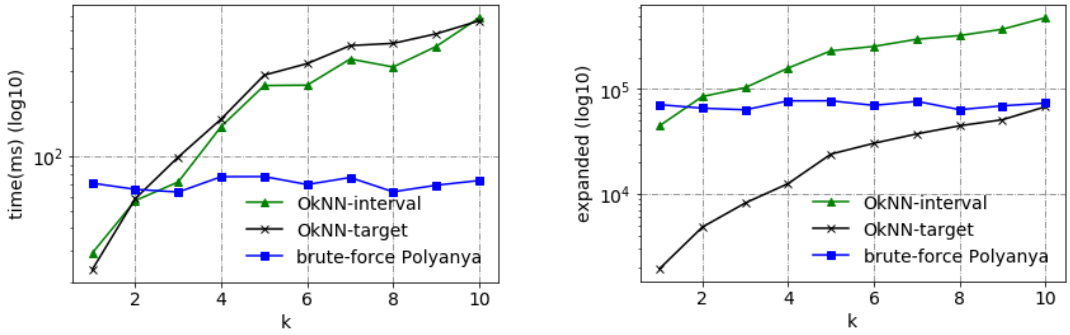
The aim of this experiment is to examine the performance of proposed algorithms in the easiest case, which is $k = 1$. Results show that in dense targets scenario, the proposed algorithms outperform *LVG* in both space and time (fig 4.2a,4.2b).

In the sparse targets scenario, *target heuristic* has the smallest number of node operations (fig 4.2d), and both *target* and *interval* heuristic are outperformed on runtime by brute-force *Polyanya* (fig 4.2c) as *dist* increases. These results suggest the *interval heuristic* has a large search space, and the *target heuristic* has a costly heuristic function. Results also show that *brute-force Polyanya* is not sensitive to *dist*, the reason is that it has to run a point-to-point search to all targets no matter where the nearest neighbor is.



(a) (b)

Dense: $k \in [1, ...10], |T| \approx |O|$



(c) (d)

Sparse: $k \in [1, ...10], |T| = 10$

Figure 4.3: Experiment2: examine the performance when k increase

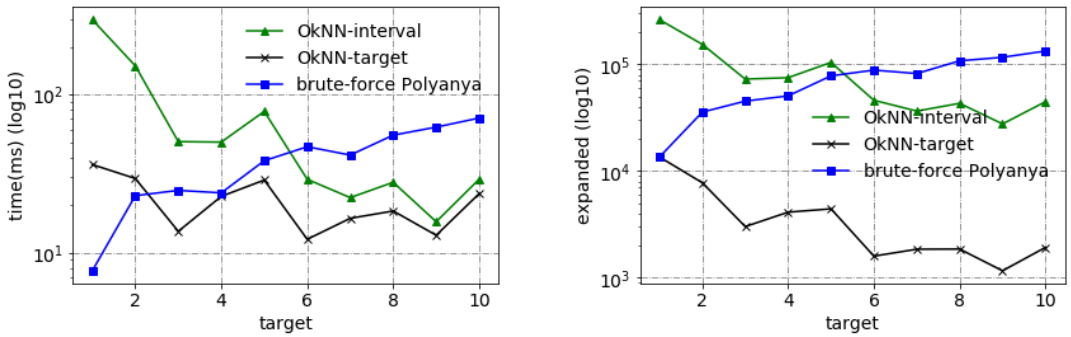
4.5 Experiment 2: computing more nearest neighbor

The aim of this experiment is to examine the performance of algorithm when query becomes harder (k increasing). Results show that, in the dense targets scenario, the proposed algorithms still outpeform *LVG*, even as k increases (fig 4.3a,4.3b). In the sparse targets scenario, results show that brute-force *Polyanya* is not sensitive to k. Meanwhile each of the two OkNN variants requires increasing amounts of node operations (and thus memory) and become quickly outperformed. A side effect of *target heuristic* when k increase is that *lazy reassign* causes extra node expansions.

4.6 Experiment 3: changing number of targets

This experiment is run only on the sparse target set. The aim of this experiment is to examine the scalability of the proposed algorithms with an increasing (but still sparse) number of targets.

Results show that proposed algorithms gradually outperform brute force *Polyanya* in both time and node operations (fig 4.4a,4.4b). This implies that the proposed OkNN variants algorithms are much better choices when the set of targets increase. Also notice that the node expansion decreases when $|T|$ goes large, for *interval heuristic*, the reason is that search nodes can reach target earlier; for *target heuristic*, in addition to previous reason, the *R-tree* query in heuristic function has good scalability. Although we have seen in other experiements that brute-force *Polyanya* has an advantage when k is large, this



(a) (b)

Figure 4.4: Experiment 3: examine the performance when $|T|$ increase, $k = 1, |T| \in [1, ...10]$

advantaged disappears as $|T|$ grows. In some practical settings $|T|$ can be in the hundreds or thousands, e.g. [1] while k is usually orders of magnitude smaller.

4.7 Experiment 4: behavior of target heuristic

From previous experiments we notice that the *target heuristic* always has smaller search space, but it's sometimes order of magnitudes slower than other competitors. Thus, we want to dig out more details about the behavior of h_t. The aim of this experiment is to examine the target heuristic from following aspects:

- the ratio of cost on heuristic function in total elapsed time;
- the ratio of reassignment in total expanded nodes;
- the ratio of heuristic function call in total expanded nodes;

.

4.7.1 Ratio of heuristic cost

Figure 4.5 shows the distribution of ratio: *heuristic cost/total cost*. The result shows that

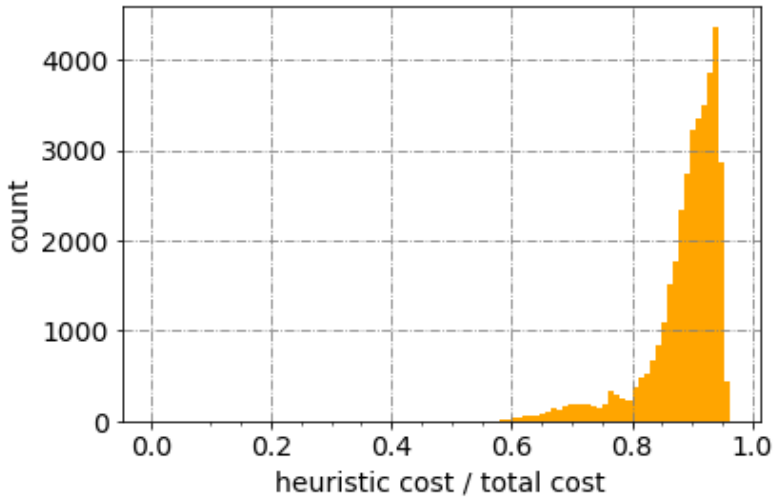


Figure 4.5: The plot is produced by records from experiments 1,2,3.

the h_t cost $\approx 90\%$ of total time, meaning that *target heuristic* needs at least an order of magnitude smaller expanded nodes to outperform competitors.

4.7.2 Side effect

Figure 4.6 shows the ratio of *lazy reassignment* when k increase. The result shows that

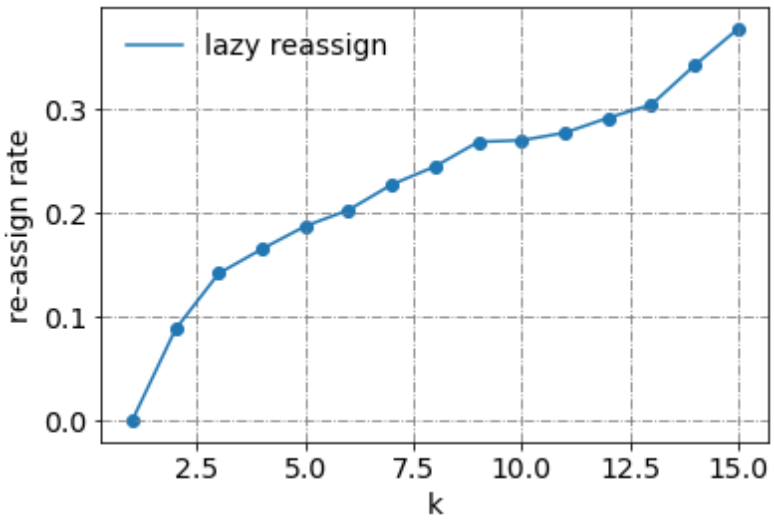


Figure 4.6: The plot is produced by records from experiments 1,2,3.

the side effect-*reassignment* happens more freqent when k increase, meaning that *target heuristic* is more suitable for small k.

4.7.3 Ratio of heuristic call

Figure 4.7 shows the distribution of ratio: *heuristic call/expanded nodes*. To ignore the

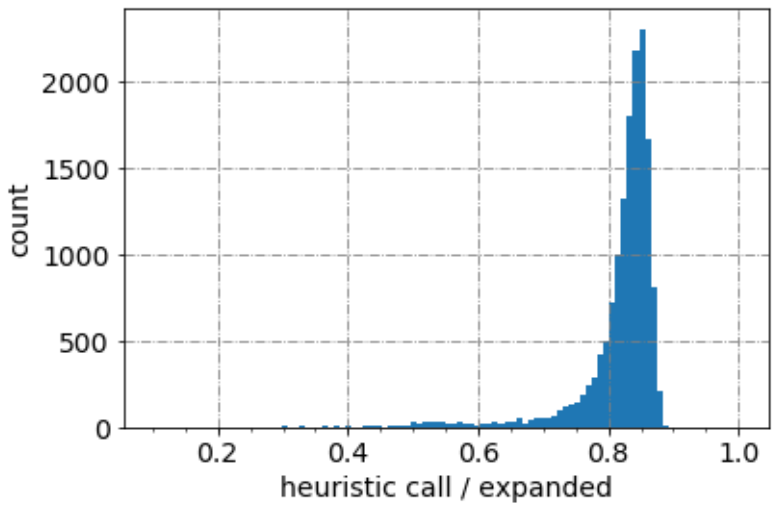


Figure 4.7: The plot is produced by records from experiments 1,2,3 where $k = 1$

side effect when $k > 1$, this plot only considers those records with $k = 1$. The plot implies the effect of *lazy compute* can $\approx 15\%$ heuristic call in the search.

4.8 Summary

This chapter is to evaluate proposed algorithms. In the chapter, we've discussed the motivation of experiment design, the benchmark problem, and results of the experiment. From these experiements, we can see that proposed algorithms outpeform the state-of-the-art orders of magnitudes, and between these proposed algorithms, they have different suitable scenarios.

Chapter 5

Conclusion and Future Work

5.1 Research Contributions

In this work, we consider efficient algorithms for OkNN: the problem of finding k nearest neighbours in a plane and in the presence of obstacles. We describe three new OkNN algorithms, all based on Polyanya [6], a recent and very fast algorithm for computing Euclidean-optimal shortest paths in the plane. The first variant involves brute force search (one query per target point). The second and third variants involve running Polyanya as a multi-target algorithm but with added heuristic guidance. We develop two new online and admissible heuristics for this purpose: the Interval Heuristic and the Target Heuristic.

We compare these variant algorithms against one another and against LVG [27], an influential and state of the art OkNN method based on incremental visibility graphs. The headline result from our experiment is that Polyanya OkNN can be up to **three orders of magnitude** faster than LVG.

Moreover, each of the three variants appears best suited to particular OkNN settings: **brute-force Polyanya** is highly effective when the number of candidates is small (independent of k); the **Interval Heuristic** works well when targets are many (again, independent of k); the **Target Heuristic** works well when targets are few and k is also small.

5.2 Future Works

We also consider future works in following directions. Firstly, due to the high performance of proposed algorithms, we believe they can be used to speed up other types of spatial queries which need to compute obstacle distance, as described in section 2.6.

Secondly, we can improve heuristic function in *target heuristic*. In experiments, we notice that the *target heuristic* cost $\approx 90\%$ of the total running time on NN query, and one possible way is combining four queries into one. Besides, in the literature review, we mainly focus on *R-tree* and its variants which have good scalability on dimensions, however, in our case, the dimension is always two (maybe three in 3D extension). Thus, other high-performance and low-dimension data structures may helpful, e.g. *KD-tree* [19] and *Locality-sensitive hashing* [24].

Thirdly, *brute-force Polyanya* can be improved by smart pruning. We notice that the *brute-force Polyanya* sometime outperforms other proposed algorithms in sparse scenarios, thus, instead of considering all targets, a pruning strategy like *fast filter* [26] may make it work in general scenario.

Bibliography

[1] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *Proceedings of the VLDB Endowment*, 9(6):492–503, 2016.

[2] Ronald C. Arkin. Path planning for a vision-based autonomous robot. volume 0727, pages 0727 – 0727 – 11, 1987.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. ACM, 1990.

[4] King Lum Cheung and Ada Wai-Chee Fu. Enhanced nearest neighbour search on the r-tree. *ACM SIGMOD Record*, 27(3):16–21, 1998.

[5] L Paul Chew. Constrained delaunay triangulations. *Algorithmica*, 4(1-4):97–108, 1989.

[6] Michael L Cui, Daniel D Harabor, Alban Grastien, and Canberra Data61. Compromise-free pathfinding on a navigation mesh. *IJCAI*, 2017.

[7] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[8] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *Aaai*, volume 6, pages 942–947, 2006.

[9] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[10] Yunjun Gao, Qing Liu, Xiaoye Miao, and Jiacheng Yang. Reverse k-nearest neighbor search in the presence of obstacles. *Information Sciences*, 330:274–292, 2016.

[11] Yunjun Gao, Jiacheng Yang, Gang Chen, Baihua Zheng, and Chun Chen. On efficient obstructed reverse nearest neighbor query processing. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in Geographic Information Systems*, pages 191–200. ACM, 2011.

[12] Yunjun Gao and Baihua Zheng. Continuous obstructed nearest neighbor queries in spatial databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 577–590. ACM, 2009.

[13] Subir Kumar Ghosh and David M Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM Journal on Computing*, 20(5):888–910, 1991.

[14] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[15] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[16] Marcelo Kallmann. Path planning in triangulations. In *Proceedings of the IJCAI workshop on reasoning, representation, and learning in computer games*, pages 49–54, 2005.

[17] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. Technical report, 1993.

[18] Tomás Lozano-Pérez and Michael A Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[19] Beng Chin Ooi, Ken J McDonell, and Ron Sacks-Davis. Spatial kd-tree: An indexing mechanism for spatial databases. In *IEEE COMPSAC*, volume 87, page 85. sn, 1987.

[20] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2017.

[21] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *ACM sigmod record*, volume 24, pages 71–79. ACM, 1995.

[22] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. Technical report, 1987.

[23] Micha Sharir and Amir Schorr. On shortest paths in polyhedral spaces. *SIAM Journal on Computing*, 15(1):193–215, 1986.

[24] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal processing magazine*, 25(2):128–131, 2008.

[25] Shenlu Wang, Muhammad Aamir Cheema, Xuemin Lin, Ying Zhang, and Dongxi Liu. Efficiently computing reverse k furthest neighbors. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1110–1121. IEEE, 2016.

[26] Chenyi Xia, David Hsu, and Anthony KH Tung. A fast filter for obstructed nearest neighbor queries. In *British National Conference on Databases*, pages 203–215. Springer, 2004.

[27] Jun Zhang, Dimitris Papadias, Kyriakos Mouratidis, and Manli Zhu. Spatial queries in the presence of obstacles. *Advances in Database Technology-EDBT 2004*, pages 567–568, 2004.

Appendix A

Dataset

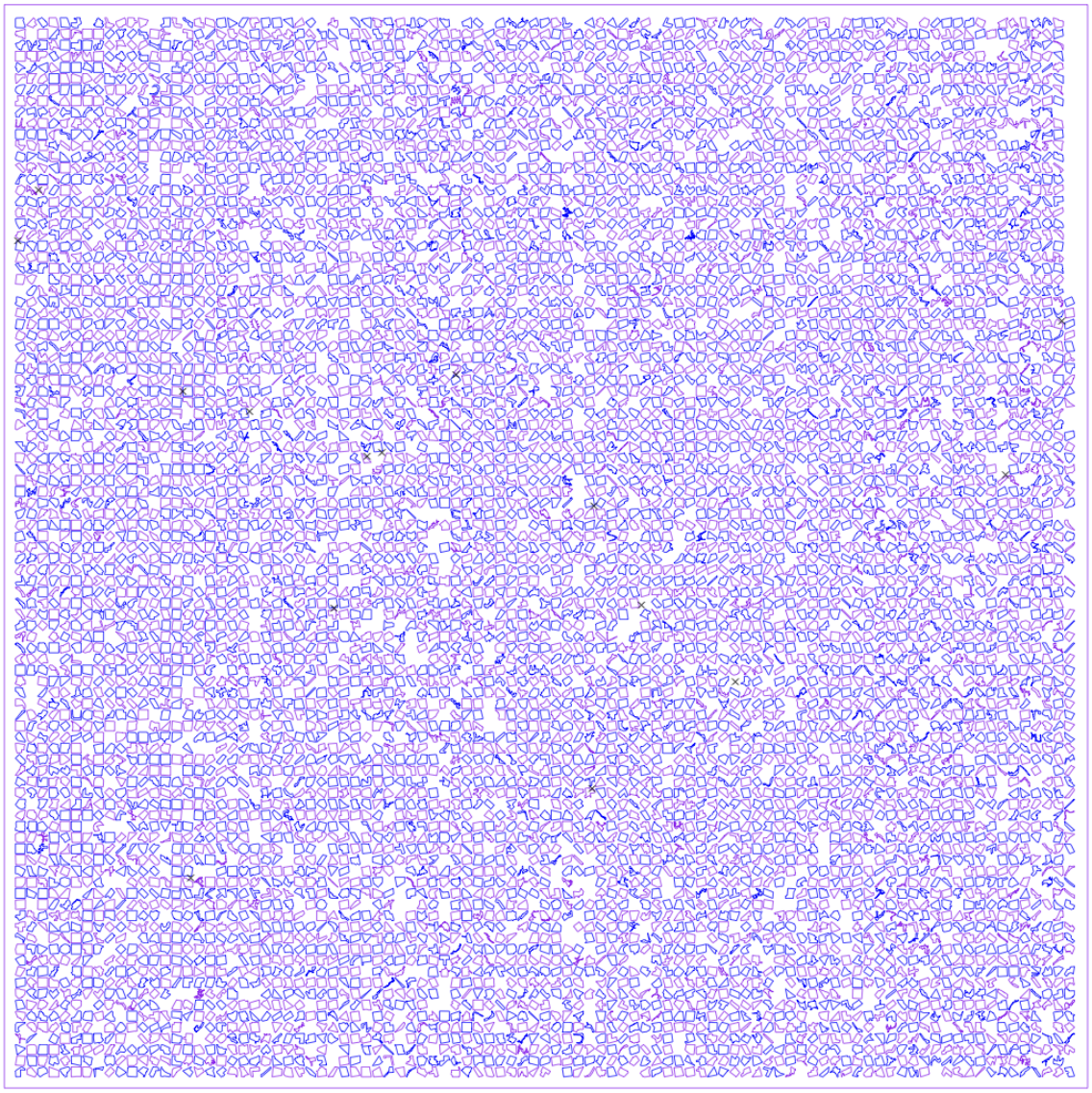


Figure A.1: The map in experiments, it has nearly 9000 polygonal obstacles, and 10,000 vertexes.

Appendix B

Source Code

Listing B.1: extract shape of parks

```
namespace pl = polyanya;
namespace rs = rstar;
namespace generator {

bool is_self_intersected(const vector<pl::Point>& poly) {
  int N = (int)poly.size();
  if (N == 3) return false;
  for (int i=0; i<N; i++) {
    pl::Point v0 = poly[i];
    pl::Point v1 = poly[(i+1) % N];
    for (int j=i+2; j<N; j++) {
      pl::Point u0 = poly[j];
      pl::Point u1 = poly[(j+1) % N];
      if ((j+1) % N == i) continue;
      if (is_intersect(v0, v1, u0, u1))
        return true;
    }
  }
  return false;
}

void simplify_polys(const vector<vector<pl::Point>>& polys, vector<vector<
    pl::Point>>& simplified) {
  for (const auto& poly: polys) {
    vector<pl::Point> simpPoly;
    for (const auto p: poly) {
      int sz = simpPoly.size();
      if (simpPoly.empty() || p.distance(simpPoly.back()) > EPSILON) {
        if (sz >= 2 && is_collinear(simpPoly[sz-2], simpPoly[sz-1], p))
          simpPoly.pop_back();
        simpPoly.push_back(p);
      }
    }
    if (simpPoly.back().distance(simpPoly.front()) <= EPSILON)
      simpPoly.pop_back();
      simplified.push_back(simpPoly);
    }
  }
}

rs::Mbr getPolyMbr(const vector<pl::Point>& poly) {
  double min_x, min_y, max_x, max_y;
  min_x = min_y = INF;
  max_x = max_y = -INF;
  for (const auto& it: poly) {
```

```
45      min_x = min(min_x, it.x);
46      min_y = min(min_y, it.y);
47      max_x = max(max_x, it.x);
48      max_y = max(max_y, it.y);
49    }
50    return rs::Mbr(min_x, max_x, min_y, max_y);
51  }
52
53  void fit_to_box(vector<pl::Point>& poly, double cx, double cy, double len)
        {
54    // top-left corner: (cx, cy)
55    // size of box: len
56    rs::Mbr mbr = getPolyMbr(poly);
57    double min_x = mbr.coord[0][0], min_y = mbr.coord[1][0];
58    double max_x = mbr.coord[0][1], max_y = mbr.coord[1][1];
59    // move to (0, 0)
60    for (auto& p: poly) {
61      p.x -= min_x;
62      p.y -= min_y;
63    }
64    // normalize
65    for (auto& p: poly) {
66      p.x = floor(p.x / (max_x - min_x) * len);
67      p.y = floor(p.y / (max_y - min_y) * len);
68    }
69    rs::Mbr mbr2 = getPolyMbr(poly);
70    // move top-lef corner to (cx, cy)
71    for (auto& p: poly) {
72      p.x += cx;
73      p.y += cy;
74      assert(p.x >= cx - EPSILON && p.x <= cx + len + EPSILON);
75      assert(p.y >= cy - EPSILON && p.y <= cy + len + EPSILON);
76    }
77  }
78
79  void sort_by_corner(vector<vector<pl::Point>>& polys) {
80    auto cmp = [&](vector<pl::Point>& a, vector<pl::Point>& b) {
81      double min_xa, min_xb, min_ya, min_yb;
82      rs::Mbr mbra = getPolyMbr(a);
83      rs::Mbr mbrb = getPolyMbr(b);
84      min_xa = mbra.coord[0][0], min_ya = mbra.coord[1][0];
85      min_xb = mbrb.coord[0][0], min_yb = mbrb.coord[1][0];
86      if (fabs(min_ya - min_yb) > EPSILON)
87        return min_ya < min_yb;
88      else
89        return min_xa < min_xb;
90    };
91    sort(polys.begin(), polys.end(), cmp);
92  }
93
94  void normalize_polys(vector<vector<pl::Point>>& polys,  double size) {
95    sort_by_corner(polys);
96    int tot = (int)polys.size();
97    int num = sqrt((double)tot) + 1;
98    double len = floor(size / ((double)num + 2.0));
99    double d = 10;
100
101   int cur = 0;
102   for (int i=1; i<=num && cur < tot; i++) {
103     for (int j=1; j<=num && cur < tot; j++) {
104       double cx = (double)i * len;
105       double cy = (double)j * len;
106       fit_to_box(polys[cur++], cx + d, cy + d, (len - d) * 0.9);
```

```
107         rs::Mbr mbr = getPolyMbr(polys[cur-1]);
108       }
109     }
110     // add border
111     polys.insert(polys.begin(), {{d, d}, {size-d, d}, {size-d, size-d}, {d,
          size-d}});
112 }
113
114 void random_choose_polys(vector<vector<pl::Point>>& polys, int num) {
115     std::random_shuffle(polys.begin(), polys.end());
116     if (num < (int)polys.size())
117       polys.erase(polys.begin() + num, polys.end());
118 }
119
120 vector<vector<pl::Point>> remove_bad_polys(vector<vector<pl::Point>>&
      raw_polys) {
121     vector<vector<pl::Point>> res;
122     for (auto& it: raw_polys) {
123       if (!is_self_intersected(it))
124         res.push_back(it);
125     }
126     return res;
127 }
128 }// namesapce geneartor
```

Listing B.2: generate targets

```
namespace generator {

void gen_points_in_traversable(EDBT::ObstacleMap* oMap, const vector<vector
    <pl::Point>>& polys,
                               int num, vector<pl::Point>& out) {
  long long min_x, max_x, min_y, max_y;
  // ignore border
  min_x = max_x = polys[1][0].x;
  min_y = max_y = polys[1][0].y;
  for (size_t i=1; i<polys.size(); i++) {
    for (const auto& p: polys[i]) {
      min_x = min(min_x, (long long)p.x);
      max_x = max(max_x, (long long)p.x);
      min_y = min(min_y, (long long)p.y);
      max_y = max(max_y, (long long)p.y);
    }
  }

  out.resize(num);
  random_device rd;
  mt19937 eng(rd());
  uniform_int_distribution<long long> distx(min_x, max_x);
  uniform_int_distribution<long long> disty(min_y, max_y);

  for (int i=0; i<num; i++) {
    long long x, y;
    do {
      x = distx(eng);
      y = disty(eng);
      pl::Point p{(double)x, (double)y};
      if (oMap->isCoveredByTraversable(p, p)) {
        out[i] = p;
        break;
      }
    } while (true);
  }
}
}// namespace generator
```

Listing B.3: target heuristic

```
point hearest_neighbor(Area area, point p) {
  // retrieval the nearest neighbor of p in area
  priority_queue pq = priority_queue();
  double dist = mindist(rtree.root.MBR, p);
  pq.push({key=dist, value=rtree.root});
  while (!pq.empty()) {
    Entry cur = pq.top(); pq.pop();
    if (cur.value is leaf) {
      // find the nearest neighbor
      return cur.value;
    }
    for (child in cur.value.children) {
      if (child.MBR disjoint area)
        continue;
      dist = mindist(child.MBR, p);
      pq.push({key=dist, value=child});
    }
  }
  // no point in area
  return null;
}

point get_closest_target(SearchNode node) {
  point p1, p2, p3, p4, closest;
  p1 = nearest_neighbor(areaA, a);
  p2 = nearest_neighbor(areaB, b);
  p3 = nearest_neighbor(areaC, r);
  p4 = nearest_neighbor(areaC', r');
  closest = best(p1, p2, p3, p4);
  return closest;
}
```

Listing B.4: final OkNN

```
#include "knnheuristic.h"
#include "expansion.h"
#include "geometry.h"
#include "searchnode.h"
#include "successor.h"
#include "vertex.h"
#include "mesh.h"
#include "point.h"
#include "consts.h"
#include <queue>
#include <vector>
#include <cassert>
#include <iostream>
#include <algorithm>
#include <ctime>

namespace polyanya {

int KnnHeuristic::search() {
  init_search();
  timer.start();
  if (mesh == nullptr) {
    timer.stop();
    return 0;
  }

  while (!open_list.empty()) {
    SearchNodePtr node = open_list.top(); open_list.pop();

    nodes_popped++;
    if (node->reached) {
      deal_final_node(node);
      if ((int)final_nodes.size() == K) break;
      continue;
    }

    //if (reached.find(node->heuristic_gid) != reached.end()) {
    if (fabs(reached[node->heuristic_gid] - INF) > EPSILON) {
      // reset heuristic goal
      const Point& root = node->root == -1? start: mesh->mesh_vertices[node
          ->root].p;
      std::pair<int, double> nexth = get_min_hueristic(root, node->left,
          node->right);
      node->heuristic_gid = nexth.first;
      node->f = nexth.second + node->g;
      nodes_reevaluate++;
      if (node->heuristic_gid == -1) {
        break;
      };
      open_list.push(node);
      nodes_pushed++;
      continue;
    }

    const int root = node->root;
    if (root != -1) {
      assert(root < (int) root_g_values.size());
      if (root_search_ids[root] == search_id) {
        // We've been here before!
        // Check whether we've done better.
        if (root_g_values[root] + EPSILON < node->g) {
          nodes_pruned_post_pop++;
```

```
61            continue;
62          }
63        }
64      }
65      int num_nodes = 1;
66      search_nodes_to_push[0] = *node;
67      for (int i = 0; i < num_nodes; i++) {
68        // update h value before we push
69        const SearchNodePtr nxt = new (node_pool->allocate()) SearchNode(
              search_nodes_to_push[i]);
70        const Point& nxt_root = (nxt->root == -1 ? start: mesh->mesh_vertices
              [nxt->root].p);
71        assert(node->heuristic_gid != -1);
72        double geth = get_h_value(nxt_root, goals[node->heuristic_gid], nxt->
              left, nxt->right);
73        if (fabs(geth - (node->f - nxt->g)) <= EPSILON) { // heuristic not
              change
74          nxt->heuristic_gid = node->heuristic_gid;
75          nxt->f = node->f;
76        }
77        else {
78          std::pair<int, double> nxth = {-1, INF};
79          if (nxth.first == -1 || fabs(reached[nxth.first] - INF) > EPSILON)
                {
80            nxth = get_min_hueristic(nxt_root, nxt->left, nxt->right, geth,
                  node->heuristic_gid);
81          }
82          nxt->heuristic_gid = nxth.first;
83          nxt->f = nxt->g + nxth.second;
84        }
85        nxt->parent = node;
86        open_list.push(nxt);
87        nodes_pushed++;
88        nodes_generated++;
89
90        // when nxt can be final_node
91        int nxt_poly = nxt->next_polygon;
92        if (!end_polygons[nxt_poly].empty()) {
93          gen_final_nodes(nxt, nxt_root);
94        }
95      }
96    }
97    timer.stop();
98    return (int)final_nodes.size();
99  }
100
101 void KnnHeuristic::deal_final_node(const SearchNodePtr node) {
102
103   const Point& goal = goals[node->goal_id];
104   const int final_root = [&]() {
105        const Point& root = root_to_point(node->root);
106        const Point root_goal = goal - root;
107        // If root-left-goal is not CW, use left.
108        if (root_goal * (node->left - root) < -EPSILON) {
109            return node->left_vertex;
110        }
111        // If root-right-goal is not CCW, use right.
112        if ((node->right - root) * root_goal < -EPSILON)
113        {
114            return node->right_vertex;
115        }
116        // Use the normal root.
117        return node->root;
```

```
118   }();
119
120   //if (reached.find(node->goal_id) == reached.end()) {
121   if (fabs(reached[node->goal_id]-INF) < EPSILON) {
122     int end_polygon = node->next_polygon;
123     const SearchNodePtr true_final =
124       new (node_pool->allocate()) SearchNode
125       {node, final_root, goal, goal, -1, -1, end_polygon, node->f, node->g
              };
126     true_final->set_reached();
127     true_final->set_goal_id(node->goal_id);
128     reached[node->goal_id] = node->f;
129     final_nodes.push_back(true_final);
130     nodes_generated++;
131   }
132 }
133
134 void KnnHeuristic::gen_final_nodes(const SearchNodePtr node, const Point&
      rootPoint) {
135     for (int gid: end_polygons[node->next_polygon]) {
136       const Point& goal = goals[gid];
137       SearchNodePtr final_node = new (node_pool->allocate()) SearchNode(*
              node);
138       final_node->set_reached();
139       final_node->set_goal_id(gid);
140       final_node->f = final_node->g + get_h_value(rootPoint, goal, node->
              left, node->right);
141       open_list.push(final_node);
142       nodes_generated++;
143       nodes_pushed++;
144     }
145 }
146 }
```