Fast Obstacle Spatial Query 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

© Copyright

by

Shizhe Zhao

2018

Contents

Abstract . . . . . **v**

1 Introduction **1**
1.1 Overview 1
1.2 Major Challenges 2
1.3 Major Objectives 2
1.4 Thesis Organisation 2

2 Literature Review **3**
2.1 Overview 3
2.2 Classic pathfinding 3
2.3 Spatial Index 4
2.3.1 *R-tree* 4
2.3.2 Nearest Neighbor Query 5
2.4 Obstacle k-Nearest Neighbor 5
2.4.1 In-main-memory OkNN 5
2.4.2 Local Visibility Graph 5
2.4.3 Fast filter 7
2.4.4 Discussion 7
2.5 Pathfinding on Navigation Mesh 8
2.5.1 Historic background 8
2.5.2 Polyanya 9
2.6 Other Obstacle Spatial Queries 10
2.7 Summary 11

3 Proposed Algorithms **12**
3.1 Overview 12
3.2 Problem Statement 12
3.3 Motivation 13
3.4 Interval Heuristic 14
3.5 Target Heuristic 15
3.5.1 Further Refinements 16
3.6 Summary 17

4 Empirical Analysis **19**
4.1 Overview 19
4.2 Benchmark 19
4.3 Competitors 19
4.4 Experiment 1: lower bounds on performance 19
4.5 Experiment 2: computing more nearest neighbor 19
4.6 Experiment 3: changing number of targets 19

5 Conclusion and Future Work **20**
5.1 Research Contributions 20
5.2 Future Works 20

Bibliography **21**

Fast Obstacle Spatial Query 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 an 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.

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

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.

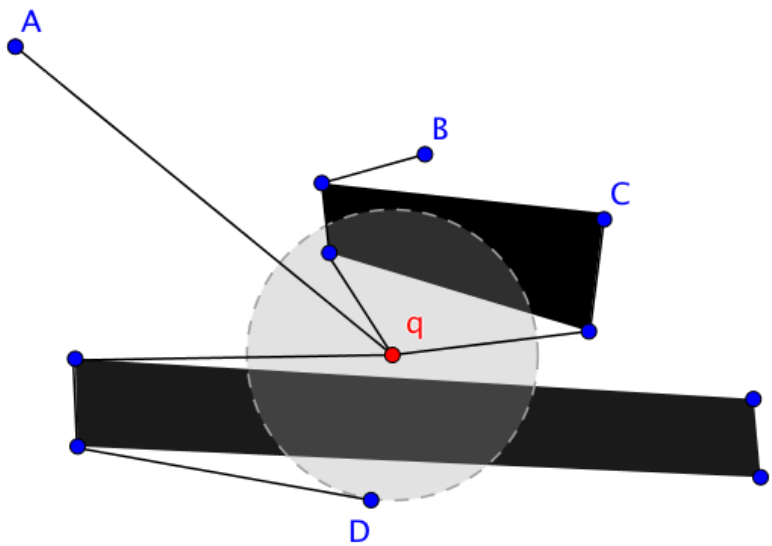


Figure 1.1: We aim to find the nearest neighbour of point q from among the set of target points A, B, C, D. Black lines indicate the Euclidean shortest paths from q. Notice D is the nearest neighbor of q under the Euclidean metric but also the furthest neighbor of q when obstacles are considered.

Tradional kNN queries in the plane (i.e. no obstacles) is a well studied problem that can be handled by algorithms based on 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* [21] and *fast filter* [20]. 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. [9–11]. 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 multi-target case.

1.4 Thesis Organisation

The rest of the thesis is organised as follows:

- In chapter 2, we review related works in different area, 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 proveded.
- In chapter 4, we provide experiment results to demonstrate the performance of propsed algorithms.
- In chapter 5, we summarize our contributions and discuss future works.

Chapter 2

Literature Review

2.1 Overview

As we've mentioned in 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 historic 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* [8]. 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 lenght of shortest path until requirements be 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 acording to a specified rule. $A*$ [14] 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 shortest path from start to n, *h-value* is a estimation of shortest path from n to the goal which is problem-specific. One important propert 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 [2, 13, 16, 19], they improve efficiency in different aspects, but they still provide same functionality, so we only introduce the classic *R-tree* in this section.

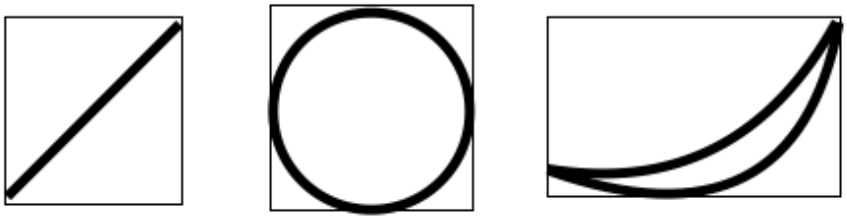
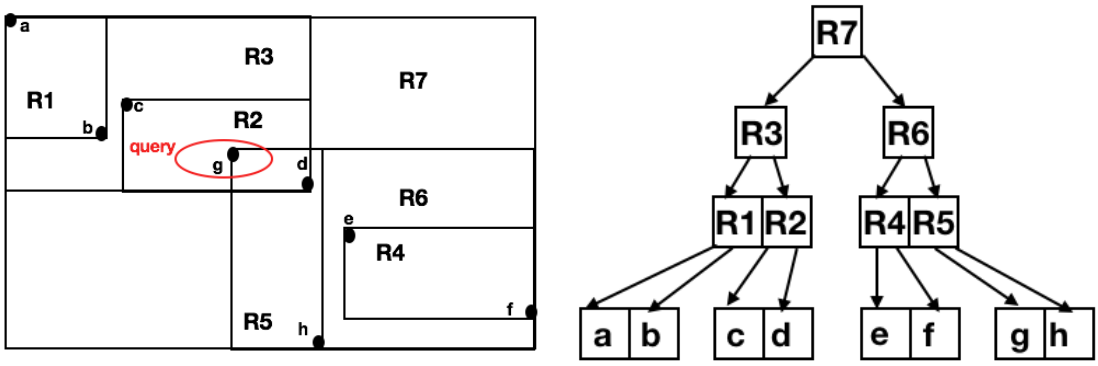


Figure 2.1: Both segments, circle and irregular shape can be represented by their MBR

R-tree is a heighb-balanced tree [13], all objects are stored in 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 example of MBRs. 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 alway 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 multiplel times in retrieval progress, which duplicated efforts. Thus, some variants use strict non-overlapping interior node (e.g. R^+*-tree* [19]), and non-overlapping *R-tree* is a wide topic in spatial index 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 uppber 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 artifical intelligence areas [19], and existing NN queries are based on it with different ordering and pruning strategies, more details are in [3, 18].

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 [6], these works need to compute visibility graph which any pair of co-visible vertices has en edge, figure 2.3 shows an example.

Since all edges of the shortest path belong to visibility graph [17], 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 [12] has $O(m + nlogn)$ runtime, where n is the number of vertex and m is the number 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 consider query related area.

In 2004, Zhang proposed the *Local Visibility Graph* (LVG) algorithm [21] to compute obstacle distance. Assume obstacles are stored in *R-tree*, given query point q, and a target t, the algorithm runs in following way:

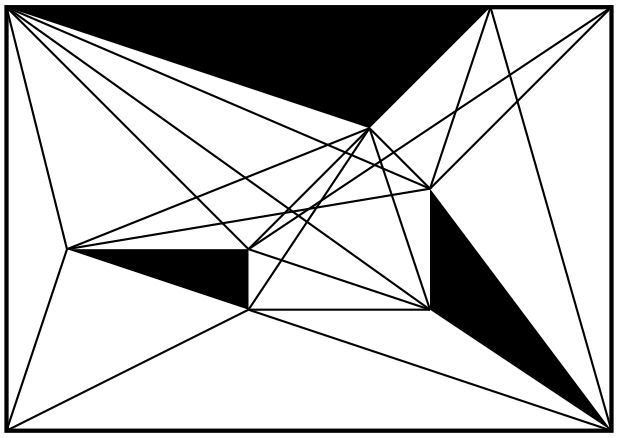
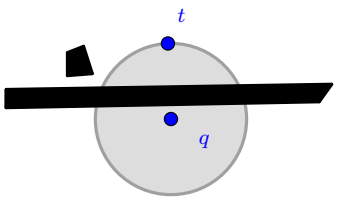
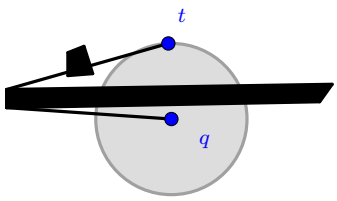


Figure 2.3: The rectangle is the boundary of the map, black polyogns are obstacles, and all black lines are edge in the visibility graph.

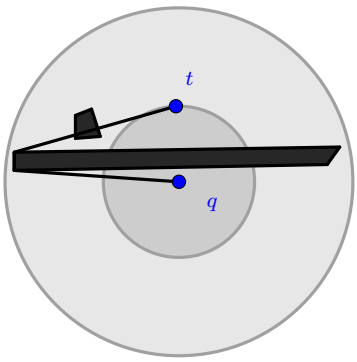
1. It starts with a small VG centered on q with radius $r = d_e(q, t)$ (figure 2.4a);
2. Then compute shortest path on current VG (figure 2.4b);
3. Enlarge the circle to current obstacle distance $r = d_o(q, t)$, update the VG incrementally (figure 2.4b) and recompute the shortest path (figure 2.4d);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



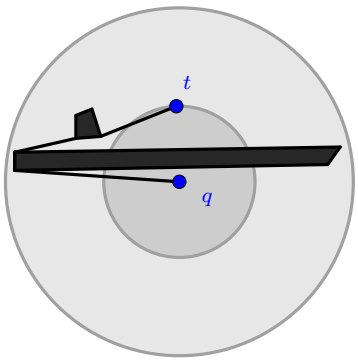
(a) the long rectangle obstacle in the circle is retrieved and included in vg.



(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 current shortest path, and thus proof 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 [20], the difference is, instead of considering obstacles in a circle area, it only retrieves obstacles on current shortest path, updates the LVG and recomputes the shortest path. Since less obstacles are involved in VG, the algorithm is called *Fast Filter*. Figure 2.5 shows an example.

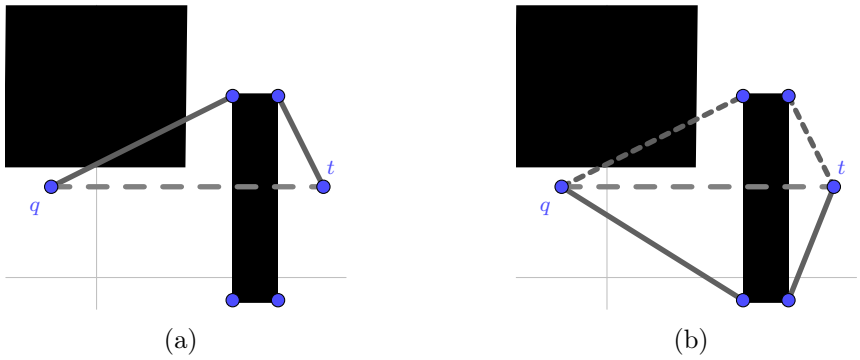


Figure 2.5: Black polygons are obstacle, q is query point, t is the target. Initially, the VG only contains the rectangle obstacle between the qt, and the corresponding shortest path is computed (fig 2.5a). Then retrieves the square obstacle on the current path, updates VG and recomputes shortest path (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* [20] is more efficient because it only consider obstacles that might on the path, meanwhile *LVG* [21] will build visibility graph for a large area, which is slow. In multi-target scenario, when k is large, *LVG* [21] is more efficient because *dijkstra* is a natural single-source algorithm, meanwhile, each time *fast filter* [20] can only compute path for a single target, so that it duplicates effort for common prefixes of k targets.

Both *LVG* [21] and *Fast filter* [20] 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. [9, 10]. 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 effort, since the graph is discarded each time the query point changes.

2.5 Pathfinding on Navigation Mesh

2.5.1 Historic background

Because of the limitation of VG, *Navigation Mesh* comes to our sight. Ronald first proposed this concept in 1986 [1], 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* [4] 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 number 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.

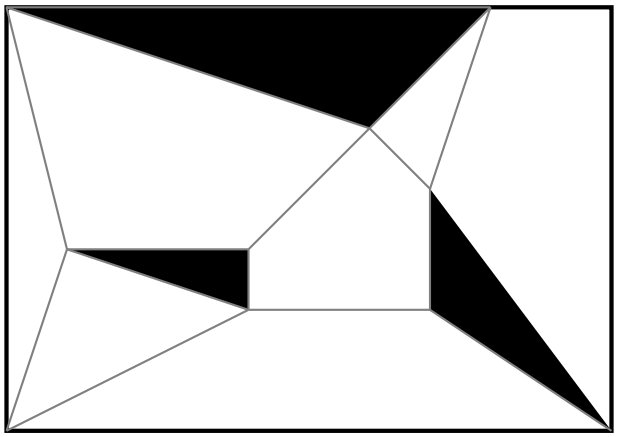


Figure 2.6: Navigation Mesh: The rectangle is the boundary of the map, black polyogns are obstacles, gray lines are edges of mesh polygons.

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 widly used algorithms are:

- Channel Search [15]: first find an abstract path from start to target composed by 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* [7]: 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* [7]: similar to TA*, but TRA* can utilize preprocessing to speed up pathfinding. The problem is that the precomputed information will be invalid when environment change, so it is not suitable for database scenario.

However, this fact has been changed by a recent work called *Polyanya* [5]. It is a fast, optimal and fexible 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 prioritise 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.

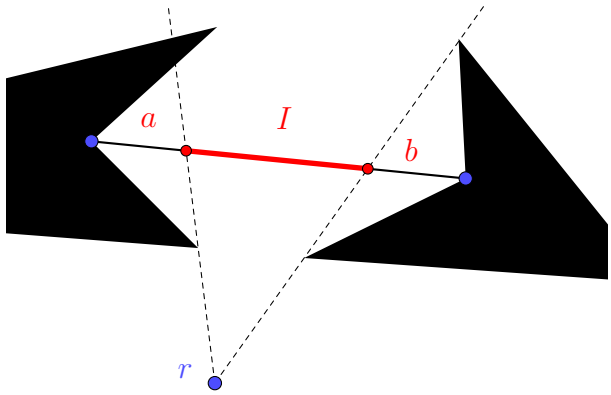


Figure 2.7: Search nodes in Polyanya. Notice that the interval $I = [a, b]$ is a contiguous subset of points drawn from an edge of the navigation mesh. The corresponding root point, r, is either the query point itself or the vertex of an obstacle. Taken together they form the search node (I, r).

- **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.

- **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.

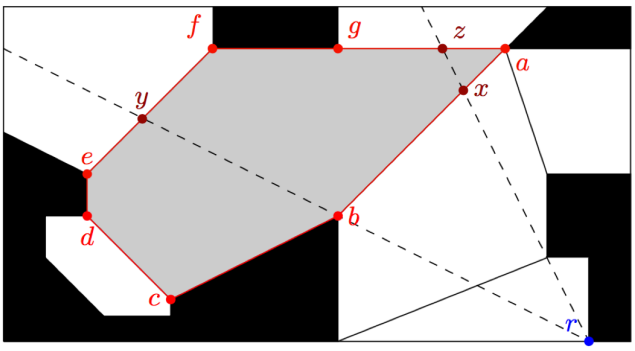


Figure 2.8: From [5]. We expand the node $([b, x], r)$ which has $([z, g], r)$ and $([f, y], r)$ as observable successors. In addition, the nodes $([c, d], b)$, $([d, e], b)$ and $([e, y], b)$ are non-observable. All other potential successors can be safely pruned (more details in [5]).

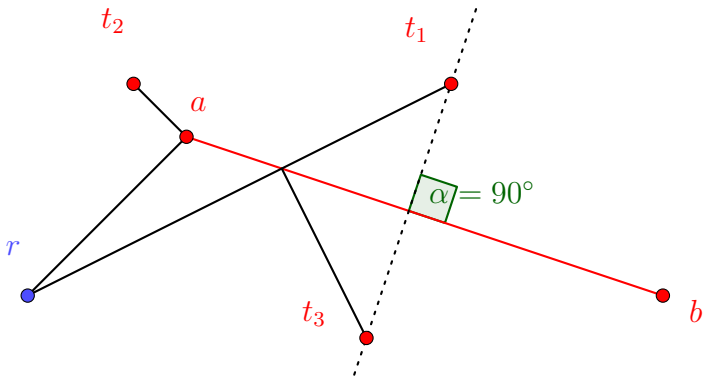


Figure 2.9: Polyanya f-value estimator. The current node is (I, r) with $I = [a, b]$ and each of t_1, t_2, t_3 are possible target locations. **Case 1**: the target is t_1. In this case the point $p \in I$ with minimum f-value is at the intersection of the interval I and the line $r \to t_1$. **Case 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 query point q and range r, it returns all targets which obstacle distance to q are less or equal to r [21]. 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 by Gao in 2011 [10] for $k = 1$, and be generalized to $k > 1$ in 2016 [9]. ORkNN given query point q and k, return a set of target 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 candidate; (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 [11], simliar 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 review works which related to our research, basically they are 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 look for a new framework, meanwhile, the new work in AI Pathfinding field shows us a new direction. Such finding inspires us to proposed our research problem: an OkNN algorithm on navigation mesh. We will discuss more detail about this in chapter 3.

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 kth 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.
- $h_t(n,R)$ is the *h-value* in Target Heuristic for a given search node n and a R^*-tree which stores all targets.

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 kth target, let's call this ***zero-heuristic***. All other aspects of the algorithm, including termination [1], remain unchanged.

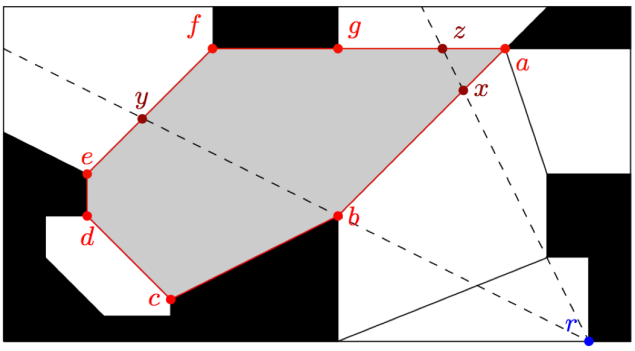


Figure 3.1: Example of successors from [5].

The version of *Polyanya* we have just described is unlikely to be efficient. Without a heuristic function for guidance, nodes can only be prioritised 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

```
foreach t in targets do
    polyanya.run(start, t);
end
```

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

- The Interval Heuristic, which prioritises nodes using the closest point from its associated interval.

[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 [5]

- 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.

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
init_search();
find_final_nodes_case1();
while heap not empty do
    node = heap.pop();
    if node is final node then
        result.add(node);
        if result.size is k then
            return resut;
        else
            continue;
        end
    else
        find_final_nodes_case2();
        successors = genSuccessors(node);
        foreach suc in successors do
            suc.gValue = node.gValue + dist(node.root, suc.root);
            suc.hValue = dist_to_segment(suc.r, suc.I);
            suc.fValue = suc.gValue + suc.hValue;
            heap.push(suc);
        end
    end
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

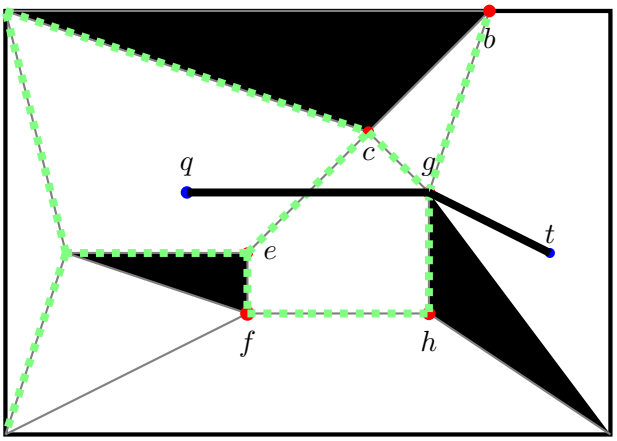


Figure 3.2: Search space of Interval Heuristic:q is query point, t is a target, green dashed segments are interval of expanded search nodes. From the figure we can see that the algorithm does unnecessary expansion in the direction that no target.

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.

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 filp 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 $t \in areaA \cup areaA'$; for 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. $areaB \cup areaB'$).

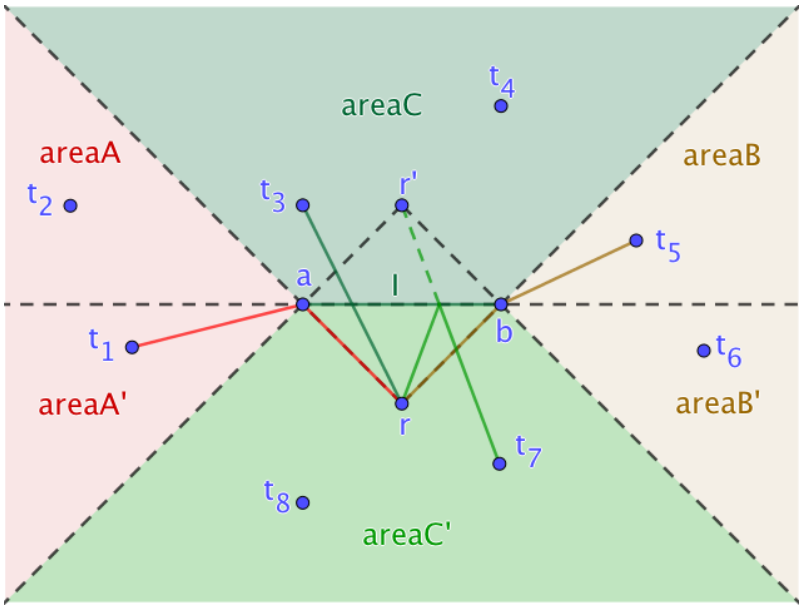


Figure 3.3: **(i)** In $areaA \cup areaA'$, t_1 is the nearest to a; **(ii)** in $areaB \cup areaB'$, t_5 is the nearest to b; **(iii)** in $areaC$, t_3 is the nearest to r; **(iv)** in $areaC'$, t_7 is the nearest to r';

- 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 a similar strategy, but only after mirroring the root point r through the interval. This operation is in contrast to the admissible f-value estimate 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 [2]. Thus we may compute a lower-bound estimate 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, requiring four NN query as compared to one constant time operation in the point-to-point case. To mitigate this we could call the function less often. For example we can assign the nearest-neighbour candidate computed at the parent node to each successor. We decide when to switch back to the Target heuristic using the point-to-point estimator: if the value is different from parent to child we need to compute a new next nearest neighbour candidate. 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_e(n, t) = h_e(n', t), where h_e is the Polyanya point-to-point estimator. 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 $h_e(n', t') < h_e(n', t) = h_e(n, t)$, and because of Lemma 3.4.1, $h_e(n, t') <= h_e(n', t')$, 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.

3.6 Summary

In this chapter, we start with two straightforward adaption and finally proposed two heuristics for the Polyanya OkNN algorithm. For *Interval Heuristic*, it support multi-target by removing t in *h-value*; for *Target Heuristic*, it support multi-target by employing spatial index and computing closest target dynamically. Each heuristic has their suitable scenario, in following chapters we will evaluate their performance under different scenarios (chapter 4) and discuss some future improvements (chapter 5).

Algorithm 3: Polyanya OkNN

```
Result: k-nearest neighbor
init_search();
find_final_node_case1();
whileheap not empty node = heap.pop();
if node is final node then
    result.add(node);
    if result.size is k then
        return result;
    else
        continue;
    end
else
    if node.target has been found then
        node.target = get_closest_target(node);
        node.hValue = get_h_value(node, node.target);
        node.fValue = node.hValue + node.gValue;
        heap.push(node);
        continue;
    else
        find_final_node_case2();
        successors = genSuccessors(node);
        foreach suc in successors do
            suc.gValue = node.gValue + dist(node.root, suc.root);
            hValue = get_h_value(suc, node.target);
            if fabs(hValue - node.hValue) <= EPS then
                suc.target = node.target;
                suc.hValue = hValue;
            else
                suc.target = get_closest_target(suc);
                suc.hValue = get_h_value(suc, suc.target);
            end
            suc.fValue = suc.gValue + suc.hValue;
            heap.push(suc);
        end
    end
end
```

Chapter 4

Empirical Analysis

4.1 Overview

4.2 Benchmark

4.3 Competitors

4.4 Experiment 1: lower bounds on performance

4.5 Experiment 2: computing more nearest neighbor

4.6 Experiment 3: changing number of targets

Chapter 5

Conclusion and Future Work

5.1 Research Contributions

5.2 Future Works

Bibliography

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

[2] 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.

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

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

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

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

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

[20] 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.

[21] 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.