The Eleventh International Symposium on Combinatorial Search
(SoCS 2018)

Fast *k*-Nearest Neighbour on a Navigation Mesh

Shizhe Zhao
Monash University
Melbourne, Australia
szha414 at student.monash.edu

David Taniar
Monash University
Melbourne, Australia
david.taniar at monash.edu

Daniel D. Harabor
Monash University
Melbourne, Australia
daniel.harabor at monash.edu

Abstract

We consider 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 work 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 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.

Introduction

Obstacle k-Nearest Neighbour (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. In competitive computer games meanwhile, agent AIs often rely on nearest-neighbour information to make strategic decisions such as during navigation, combat or resource gathering.

Traditional kNN queries in the plane (i.e. no obstacles) is a well studied problem (Roussopoulos, Kelley, and Vincent 1995; Cheung and Fu 1998) that can be handled by popular and well known algorithms including *KD-Tree* (Ooi, McDonell, and Sacks-Davis 1987) and *R-tree* (Guttman 1984). These methods organise the collection of target points into a hierarchical structure that serves to: (i) quickly identify a set of nearest neighbour candidates and; (ii) helps prune those candidates to return the k closest. 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. Figure 1 shows such an example.

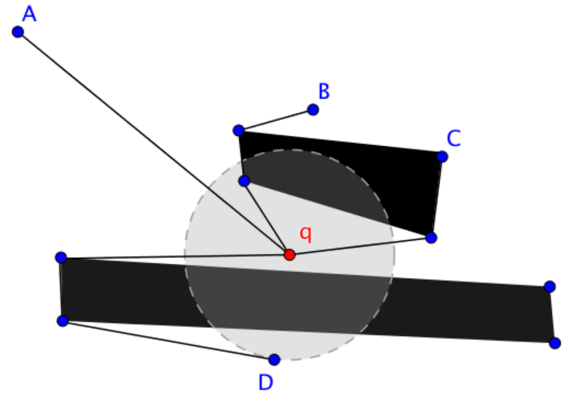


Figure 1: We aim to find the nearest neighbour of point q from among the set of target points A, B, C, D. Black lines indicate the Euclidean shortest paths from q. Notice D is the nearest neighbour of q under the Euclidean metric but also the furthest neighbour of q when obstacles are considered.

Two popular algorithms for OkNN, which can deal with obstacles, are *local visibility graphs* (Zhang et al. 2004) and *fast filter* (Xia, Hsu, and Tung 2004). 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. (Gao et al. 2011; 2016; Gao and Zheng 2009). However, incremental visibility graphs also suffer from a number of notable disadvantages including: (i) 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.

In this paper we develop a new method for computing

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

kNN in the presence of obstacles which avoids these same disadvantages. Our work extends *Polyanya* (Cui et al. 2017): a recent and very fast algorithm for computing Euclidean shortest paths on a navigation mesh: a data structure comprised of convex polygons which taken together represent the entire traversable space. Compared to visibility graphs, navigation meshes are much cheaper to construct and sometimes available as input "for free" (e.g. in computer game settings, navigation meshes are often created, at least in part, by human designers). We describe how *Polyanya* can be generalised, from point-to-point problems to the multi-target case. We also develop along the way two new, efficient heuristics which can be used for OkNN. Finally, we compare our work against incremental visibility graphs in a wide range of experimental settings where we show that *Polyanya* is in some cases orders of magnitude faster.

The rest of the paper is organised as follows: (i) we give a description of the Obstacle kNN problem and associated technical terms; (ii) we review key components of *Polyanya*; (iii) we give a formal description of our proposed algorithms including proofs and pseudocode; (iv) experimental results and discussion and; (v) concluding thoughts.

Problem Statement

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

Definition 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.

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").

Polyanya and the heuristic h_p

In this work we extend and generalise *Polyanya* (Cui et al. 2017), a recent and related algorithm that computes shortest paths between pairs of traversable points in the plane and in the presence of polygonal obstacles. At a high level, *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. Since we will employ a similar search methodology to *Polyanya*, we give herein a brief description of that algorithm. There are three key components:

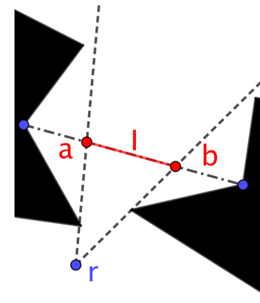


Figure 2: Search nodes in *Polyanya*. Notice that the interval $I = [a, b]$ is a contiguous subset of points drawn from an edge of the navigation mesh. The corresponding root point, r, is either the query point itself or the vertex of an obstacle. Taken together they form the search node (I, r).

- **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 shows an example.
- **Successors**: Successor nodes (I', r') are generated by "pushing" the current interval I away from its root r and through the interior of an adjacent and traversable polygon. A successor is said to be *observable* 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 3 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 for a given search node $n = (I, r)$, and target t:

  $$f(n) = g(n) + h_p(n, t)$$

  where $g(n)$ represents the length of a concrete shortest path from q to r, and $h_p(n, t)$ represents the lower-bound from r to t via some $p \in I$. There are three cases to consider which describe the relative positions of the t in relation to the r. These are illustrated in Figure 4. The objective in each case is to choose the unique $p \in I$ that

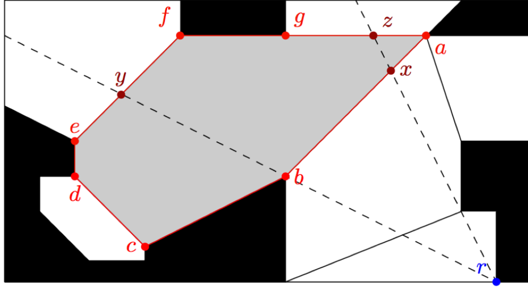


Figure 3: From (Cui et al. 2017). We expand the node $([b, x], r)$ which has $([z, g], r)$ and $([f, y], r)$ as observable successors. In addition, the nodes $([c, d], b)$, $([d, e], b)$ and $([e, y], b)$ are non-observable. All other potential successors can be safely pruned (more details in (Cui et al. 2017)).

minimises the estimate. The tree cases together are sufficient to guarantee that the estimator is admissible.

Similar to A*, *Polyanya* terminates when the target is expanded or when the open list is empty. In (Cui et al. 2017) this algorithm is shown to outperform a range of optimal and sub-optimal competitors, often by orders of magnitude. In the sections that follow we will extend this algorithm to multi-target OkNN. Our implementation is based on a publicly available version of *Polyanya*[1] from the original authors.

Multi-target Search

In this section, we discuss how to effectively adapt *Polyanya* for OkNN settings where there are multiple candidate targets (cf. just one). Since *Polyanya* instantiates A* search, and since that algorithm is itself a special case of Dijkstra's well known 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. All other aspects of the algorithm, including termination [2], remain unchanged.

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, all observable successors would have the same expansion priority. Thus we may expand many nodes, all equally promising but having distant intervals, and all before reaching a nearby candidate with a slightly higher g-value. To deal with this problem we develop two heuristics which can be fruitfully applied to OkNN:

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

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

[2]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 (Cui et al. 2017)

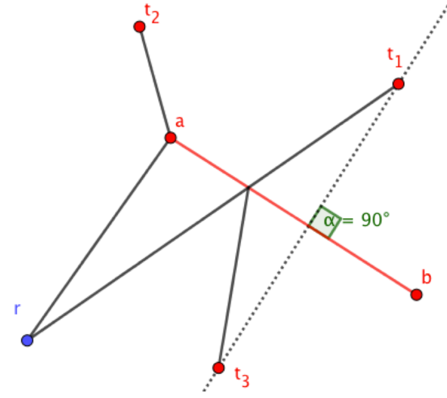


Figure 4: *Polyanya* f-value estimator. The current node is (I, r) with $I = [a, b]$ and each of t_1, t_2, t_3 are possible target locations. **Case 1**: the target is t_1. In this case the point $p \in I$ with minimum f-value is at the intersection of the interval I and the line $r \to t_1$. **Case 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.

- The Target Heuristic (h_t), which relies on a Euclidean nearest-neighbour estimator to provide a target dynamically at the time of expansion.

Each of these heuristics is applied in the usual way compute a final expansion priority: $f(n) = g(n) + h(n)$ where $g(n)$ is the (known) distance from the query point to the root, and $h(n)$ is a lower-bound on the distance to the next target. In the remainder of this section we explore these ideas in turn. In the experimental section thereafter we will also consider a third, even simpler strategy for multi-target search: repeatedly call an unmodified point-to-point *Polyanya* search, from the query point and to each target (algorithm 3). In a perhaps surprising result, we will show that each of these three alternatives yields state of the art performance in certain contexts.

The Interval Heuristic h_v

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 2 *Given search node $n = (I, r)$, the interval heuristic $h_v(n)$ is the minimum Euclidean distance from r to any point $p \in I$.*

Applying the Interval Heuristic h_v 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 1 shows an example.

Algorithm 1: OkNN: Polyanya with h_v

1 **while** *heap not empty* **do**
2   node = heap.pop();
3   **if** *node is a target that not yet reached* **then**
4   result.add(node);
5   **if** *result.size is k* **then**
6   return;
7   continue;
8   **if** *node is a target that has been found* **then**
9   continue;
10   successors = genSuccessors(node);
11   **foreach** *suc in successors* **do**
12   suc.g = node.g+ d_e(node.root, suc.root);
13   suc.f = suc. g + h_v(suc);
14   heap.push(suc);
15   **end**
16 **end**

Theorem 1 *The interval heuristic h_v is consistent.*

Proof: Let $n = (I, r)$ be the current search node, and $n' = (I, r)$ be any successor. First we show that $f(n') \geq f(n)$. We have $f(n) = g(r) + h_v(r, I)$, and $f(n') = g(r) + d_e(r, r') + h_v(r', I')$. There are two cases:

- If $r = r'$, since I' is generated by pushing away from I, we have $f(n') - f(n) = h_v(r, I') - h_v(r, I) \geq 0$,
- If $r \neq r'$, then $r' \in I$, and since the Definition 2, we have $h_v(r, I) \leq d_e(r, r')$, so that $f(n') - f(n) \geq h_v(r', I') \geq 0$.

Next we show that h_v is a lower-bound:

- If there is a target t on I, then $h_v(r, I) = d_o(r, t)$;
- Otherwise, $h_v(r, I) < d_o(r, t)$ for all $t \in T$.

□

Notice that when we expand a search node containing a candidate point t, we have also found the shortest path from q to t.

The Target Heuristic h_t

In some OkNN settings the set of targets are few (i.e. sparse) and without a reasonable heuristic guide it is possible to perform many redundant expansions in areas where no nearest neighbour can exist. In such cases more sophisticated spatial reasoning can help to prune the set of nearest neighbours and guide the search. The idea we introduce for such settings can be formalised as follows:

Definition 3 *Given a search node $n = (I, r)$, the target heuristic $h_t(n)$ is the distance from n to the closest target $t \in T$ using the heuristic $h_p(n, t)$.*

A straightforward way to implement this idea is computing the h_p for all $t \in T$ and choosing the minimum. Obviously, this approach has bad scalability. Instead, we can find the closest target efficiently by using a traditional Euclidean nearest-neighbour query (i.e. no obstacles). Such queries are supported by many spatial indexes including R-tree (more details can be found in (Roussopoulos, Kelley, and Vincent 1995; Cheung and Fu 1998)).

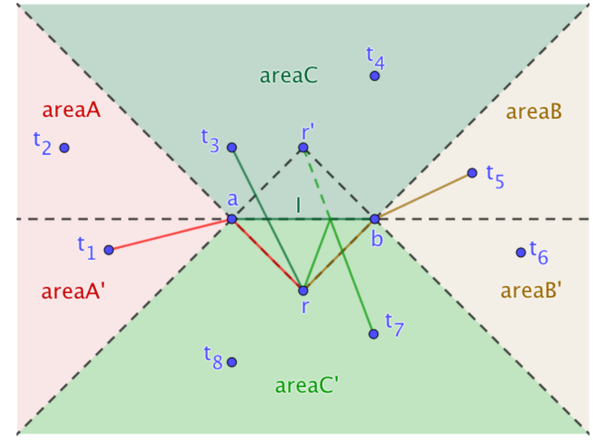


Figure 5: The heuristic h_p divides the space around the current interval into three distinct areas $(areaA, areaB, areaC)$ and three counterparts $(areaA', areaB', areaC')$ which we may reason about by mirroring points through the current interval I. We will find a candidate target from each of these areas using just four Euclidean nearest neighbour queries. To see this notice that all shortest paths to targets in $areaA$ (resp. $areaB$) and $areaA'$ (resp. $areaB'$) must intersect the interval endpoint a (resp. b). Thus we can retrieve a single nearest neighbour for both these areas. In the case of $areaC$, all shortest paths to any candidate target point must intersect the interval I. We may reason similarly about $areaC'$ but only after mirroring the root point r through the interval I to derive an equivalent root point r'. The closest candidates then are: t_1, t_5, t_3 and t_7.

Our approach divides the space around the current interval into a number of areas, as shown in Figure 4. We then find from each area a target that has minimum distance according to h_p. We will show that by choosing from among these few candidates the one with smallest distance will also satisfy Definition 3.

Using the three cases of h_p in Figure 4, given a search node $n = (I, r)$, we have following observations:

- for those $t \in T$ such that $h_p(n, t) = d_e(r, t)$, we only need to consider the nearest neighbour of r;
- for those $t \in T$ such that $h_p(n, t) = d_e(r, a) + d_e(a, t)$ (equiv. $h_p(n, t) = d_e(r, b) + d_e(b, t)$), we only need to consider the nearest neighbour of a (equiv. b);
- for those $t \in T$ such that $h_p(n, t) = h_p(n, t')$ where t' is the mirror point of t through I, we have $h_p(n, t') = h_p(n', t)$ where $n' = (I, r')$ and r' is the mirror point of r through I. In other words, instead of flipping those $t \in T$, we only flip r.

Finally, as show in Figure 5, we can form four areas.

Theorem 2 *The target heuristic h_t is consistent.*

Proof: Let $n = (I, r)$ be the current search node and $n' = (I', r')$ be any successor, such that $t \in T$ is the closest target to n and $t' \in T$ is the closest target to n'. First we show that $f(n') \geq f(n)$. We have $f(n) = g(r) + h_p(n, t)$, and $f(n') = g(r) + d_e(r, r') + h_p(n', t')$. There are four cases:

- **Case 1**: if $t = t'$ and $r = r'$, then $f(n) = f(n')$,
- **Case 2**: if $t = t'$ and $r \neq r'$, then we have $d_e(r, r') + h_p(n', t) \geq h_p(n, t)$.
- **Case 3**: if $t \neq t'$ and $r = r'$, notice that in order to compute $h_p(n', t')$ we have to intersect some point $p \in I$ before reaching t'. Thus we have $h_p(n, t) \leq h_p(n', t')$ for any $t' \in T$ and $f(n') - f(n) = h_p(n', t') + d_e(r, r') - h_p(n, t) \geq 0$.
- **Case 4**: if $t \neq t'$ and $r \neq r'$, by the Definition 3, we always choose the closest target to each search node, so $h_p(n, t) \leq h_p(n, t')$. Further, since $d_e(r, r') + h_p(n', t') \geq h_p(n, t')$, we have $f(n') - f(n) \geq d_e(r, r') + h_p(n', t') - h_p(n, t') \geq 0$.

Thus, $f(n') \geq f(n)$. Next we show that h_t is a lower-bound. To see this, notice that h_p is a lower-bound from any node to any target, and Definition 3 guarantees that we always choose the closest target that minimises h_p for all $t \in T$. □

Further Refinements for h_t

We may notice that the h_t described thus far is potentially costly, compare to the constant time operation in h_p and h_v. To mitigate this we could call the function 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 number of generated nodes by approximately 15%.

Lemma 1 *Given search node $n = (I, r)$, its successor $n' = (I', r')$, and a target t which is the closest target of n. Further suppose $f(n) = f(n')$. Then t is also the closest target of n'.*

Proof: If there is a t' such that $h_p(n', t') < h_p(n', t) = h_p(n, t)$, then $g(r) + d_e(r, r') + h_p(n', t') < g(r) + d_e(r, r') + h_p(n', t) = f(n)$ so that $f(n') < f(n)$, which conflict with Theorem 2. Thus, such t' doesn't exist. □

Now, each search node has a target, and the search behaviour should be broadly similar to the point-to-point setting. But there is one significant difference: when a nearest neighbour 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 dequeued 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 *target reassign*.

Target reassign doesn't affect the relative order of expansion. This is easy to see, since h_t is consistent and the associated target is always valid at the time of the expansion.

In Algorithm 2, we arrive at last at the final form of *Polyanya* for OkNN using the heuristic h_t.

Algorithm 2: OkNN: Polyanya with h_t

```
1  while heap not empty do
2      node = heap.pop();
3      if node is a target that not yet reached then
4          result.add(node);
5          if result.size is k then
6              return;
7          continue;
8      if node is a target that has been reached then
9          continue;
10     if node.target has been found then
11         // target reassign;
12         node.f = node.g + h_t(node);
13         heap.push(node);
14         continue;
15     successors = genSuccessors(node);
16     foreach suc in successors do
17         suc.g = node.g+ d_e(node.root, suc.root);
18         suc.f = suc. g + h_t(suc);
19         heap.push(suc);
20     end
21 end
22 Function h_t (node) :
23     hValue = h_p(node, node.parent.target)
24     if (hValue + node.g) equal f(node.parent) then
25         // lazy compute
26         node.target = node.parent.target
27         return hValue
28     else
29         node.target = get_closest_target(suc)
30         return h_p(node, node.target)
31     end
```

Empirical Analysis

To evaluate our proposed algorithms 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* (Zhang et al. 2004) 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 point-to-point search with *Polyanya*. As per Algorithm 3, we

Algorithm 3: Brute-force Polyanya

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

run one complete search for each candidate in the target set. 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 Polyanya* 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 (Abeywickrama, Cheema, and Taniar 2016), 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 generated search node (in memory). For *LVG*, we count the number of generated search node in *Dijkstra* search (in memory as well).

The navigation mesh of map is generated by *Constrained Delaunay Triangulation*, which is $O(nlogn)$; the implementation of such algorithm is in library *Fade2D* [3], 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 [4]. All experiments are performed on a 2.5 GHz Intel Core i7 machine with 16GB of RAM and running OSX 10.13.4.

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 (Zhang et al. 2004). The only differences is in construction of the visibility graph: *LVG* uses the rotational plane-sweeping algorithm from (Sharir and Schorr 1986) which runs worst case $O(n^2logn)$ time. In our work we opted to simplify development complexity and use a *R*-tree* (Beckmann et al. 1990) 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 $O(n^2logn)$. This implementation, as with the rest of our code, is made publicly available.

The implementation of R^*-tree we use is also publicly available [5], and appears in other recently published work (Wang et al. 2016).

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

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

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

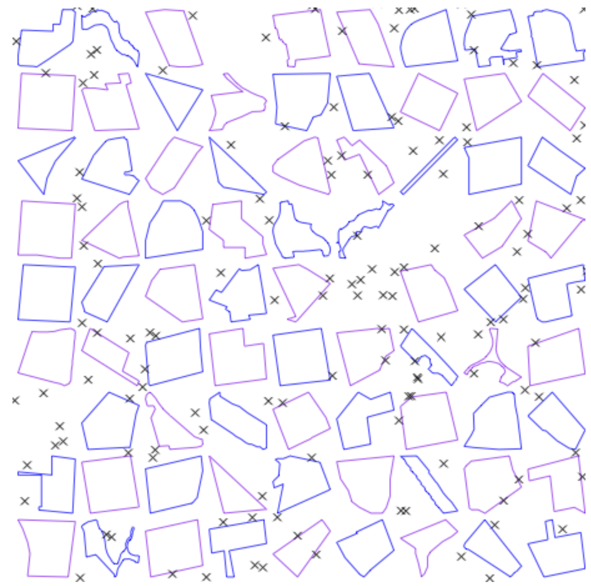


Figure 6: A generated map with many targets. Polygons are obstacles, black crosses are targets

Benchmark

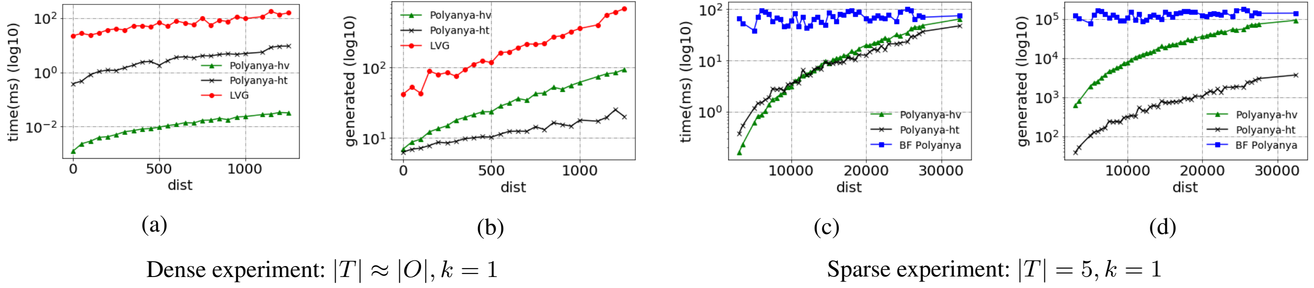
The data set from our main competitor *LVG*(Zhang et al. 2004) is no longer available on the public Internet so we opt to generate new benchmark problems. We extract the shape of all parks in Australia from *OpenStreetMap*(OpenStreetMap contributors 2017) and use these shapes as polygonal obstacles. 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 normalise the shape of polygon by to fit inside the cell.Figure 6 gives an example 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* (Zhang et al. 2004) 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.

Experiment 1: lower bounds on performance

The aim of this experiment is to examine the performance of proposed algorithms in the easiest case, which is $k = 1$.

Results for the dense targets scenario are given in Figure 7a (time) and Figure 7b (nodes). We find that both *Polyanya* variants outperform *LVG* in terms of space and time. Results for the sparse targets scenario are given in Figure 7c (time) and Figure 7d (nodes). We find that while the *target heuristic* has the smallest number of nodes generated, both it and the *interval heuristic* are outperformed in runtime

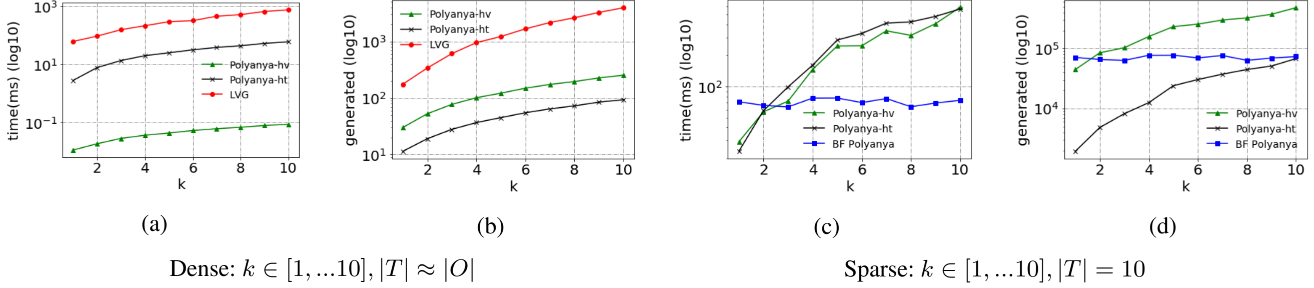


(a) (b) (c) (d)

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

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

Figure 7: Experiment 1: we fix $|T|$ and k and examine performance as obstacle distance to the kth nearest neighbour ($dist$) increases.



(a) (b) (c) (d)

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

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

Figure 8: Experiment 2: we fix the number of targets and examine performance as k increases (from 1 to 10)

by *Brute-force Polyanya*. These results suggest the *interval heuristic* has a large search space, and the *target heuristic* has a costly heuristic function. Result also show that *Brute-force Polyanya* is not sensitive to $dist$, the reason being that it has to run a point-to-point search to all targets no matter where the nearest neighbour is.

Experiment 2: computing more nearest neighbours

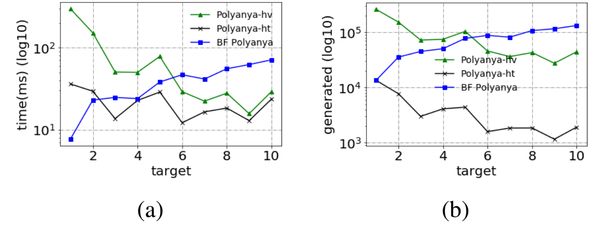
The aim of this experiment is to examine the performance of algorithm as queries become harder (k increasing).

Results for the dense scenario are given in Figure 8a (time) and Figure 8b (nodes).j We find that the proposed algorithms continue to outperform *LVG* and by similarly large margins. In the sparse targets scenario (Figure 8c (time) and Figure 8d (nodes)), results show that *Brute-force Polyanya* usually has a convincing runtime advantage and does not appear to be sensitive to k. Meanwhile each of the two OkNN variants generate increasing numbers of nodes and become quickly outperformed. A side effect of *target heuristic* in this experiment is that as k increases *target reassign* causes more nodes to be generated.

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 are given in Figure 9a (time) and Figure 9b (nodes). We find that the target and interval heuristics gradually outperform *Brute-force Polyanya* in terms of both time and number of generated nodes. This implies that these algorithms are much better choices when the set of targets increase. Also notice that nodes generated decreases as $|T|$ goes large. The reason for this is that with more targets the search quickly finds the desired k candidates and often in the vicinity of the query point. In the case of *target heuristic* we may further infer that the *R-tree* queries on which it depends have good scalability; e.e. adding more targets doesn't make these queries significantly slower. Although we have seen in other experiments that *Brute-force Polyanya* has an advantage when k is large, this advantage disappears as $|T|$ grows. In some practical settings $|T|$ can be in the hundreds or thousands e.g. (Abeywickrama, Cheema, and Taniar 2016) while k is usually orders of magnitude smaller.



(a) (b)

Figure 9: Experiment 3: we fix $k = 1$ and examine performance as $|T|$ increases (from 1 to 10)

Conclusion and Future work

In this work we consider efficient algorithms for OkNN: the problem of finding k nearest neighbours in the Eu-

clidean plane and in the presence of obstacles. We describe three new OkNN algorithms, all based on *Polyanya* (Cui et al. 2017), 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 involves running *Polyanya* as a multi-target algorithm but with added heuristic guidance. We develop two new and consistent heuristics for this purpose: the Interval Heuristic $h_v$and the Target Heuristic h_t.

We compare these variant algorithms against one another and against *LVG* (Zhang et al. 2004), an influential and state of the art OkNN method based on incremental visibility graphs. The headline result from our experiments is that OkNN with *Polyanya* can be up to orders of magnitude faster than *LVG*. Moreover, each of the three variants appears best suited to particular OkNN settings: brute force search 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.

Due to their fast performance, we believe these algorithms can be used to speed up other types of spatial queries which need to compute obstacle distance; e.g. as described in (Gao et al. 2016; Gao and Zheng 2009). Another interesting direction for future work is finding ways to further improve the Target Heuristic. In our experiments calls to this heuristic can account for approximately 80% of total search time. One possible approach to this end involves combining the four *R-tree* queries required at present into one. Finally, we notice that *Brute-force Polyanya* sometimes outperforms other proposed algorithms in sparse scenarios. This suggests another possible direction: instead of considering every target we might try to apply a pruning strategy such that the search can terminate earlier.

References

Abeywickrama, T.; Cheema, M. A.; and Taniar, D. 2016. K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *Proceedings of the VLDB Endowment* 9(6):492–503.

Beckmann, N.; Kriegel, H.-P.; Schneider, R.; and Seeger, B. 1990. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, 322–331. ACM.

Cheung, K. L., and Fu, A. W.-C. 1998. Enhanced nearest neighbour search on the r-tree. *ACM SIGMOD Record* 27(3):16–21.

Cui, M. L.; Harabor, D. D.; Grastien, A.; and Data61, C. 2017. Compromise-free pathfinding on a navigation mesh. *IJCAI*.

Gao, Y., and Zheng, B. 2009. Continuous obstructed nearest neighbor queries in spatial databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 577–590. ACM.

Gao, Y.; Yang, J.; Chen, G.; Zheng, B.; and Chen, C. 2011. On efficient obstructed reverse nearest neighbor query processing. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in Geographic Information Systems*, 191–200. ACM.

Gao, Y.; Liu, Q.; Miao, X.; and Yang, J. 2016. Reverse k-nearest neighbor search in the presence of obstacles. *Information Sciences* 330:274–292.

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

Ooi, B. C.; McDonell, K. J.; and Sacks-Davis, R. 1987. Spatial kd-tree: An indexing mechanism for spatial databases. In *IEEE COMPSAC*, volume 87, 85. sn.

OpenStreetMap contributors. 2017. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org.

Roussopoulos, N.; Kelley, S.; and Vincent, F. 1995. Nearest neighbor queries. In *ACM sigmod record*, volume 24, 71–79. ACM.

Sharir, M., and Schorr, A. 1986. On shortest paths in polyhedral spaces. *SIAM Journal on Computing* 15(1):193–215.

Wang, S.; Cheema, M. A.; Lin, X.; Zhang, Y.; and Liu, D. 2016. Efficiently computing reverse k furthest neighbors. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, 1110–1121. IEEE.

Xia, C.; Hsu, D.; and Tung, A. K. 2004. A fast filter for obstructed nearest neighbor queries. In *British National Conference on Databases*, 203–215. Springer.

Zhang, J.; Papadias, D.; Mouratidis, K.; and Zhu, M. 2004. Spatial queries in the presence of obstacles. *Advances in Database Technology-EDBT 2004* 567–568.