

Faster and More Robust Mesh-based Algorithms for Obstacle k -Nearest Neighbour

Shizhe Zhao
Monash University
Melbourne, Australia

SZHA414@STUDENT.MONASH.EDU

David Taniar
Monash University
Melbourne, Australia

DAVID.TANIAR@MONASH.EDU

Daniel D. Harabor
Monash University
Melbourne, Australia

DANIEL.HARABOR@MONASH.EDU

Abstract

We are interested in the problem of finding k nearest neighbours in the plane and in the presence of polygonal obstacles ($OkNN$). Widely used algorithms for OkNN are based on incremental visibility graphs, which means they require costly and online visibility checking and have worst-case quadratic running time. Recently **Polyanya**, a fast point-to-point pathfinding algorithm was proposed which avoids the disadvantages of visibility graphs by searching over an alternative data structure known as a navigation mesh. Previously, we adapted **Polyanya** to multi-target scenarios by developing two specialised heuristic functions: the **Interval heuristic** h_v and the **Target heuristic** h_t . Though these methods outperform visibility graph algorithms by orders of magnitude in all our experiments they are not robust: h_v expands many redundant nodes when the set of neighbours is small while h_t performs poorly when the set of neighbours is large. In this paper, we propose new algorithms and heuristics for OkNN which perform well regardless of neighbour density.

1. Introduction

The Obstacle (equiv. Obstructed) k -Nearest Neighbour Problem (OkNN) is a generalised variant of the well known Euclidean k -Nearest Nearest Problem (kNN) in two dimensions. In both cases the objective is to return the k closest neighbours (equiv. targets) to an a priori unknown query point q . In the case of OkNN however polygonal obstacles are introduced and the objective is to return shortest distances to each of the k closest neighbours without crossing any obstacles. We refer to this metric as the *obstacle distance* between traversable points. Fig 1 highlights the differences between the two problem settings.

OkNN problems appear in a number of different application areas and are of interest in several different research fields. For example, in the context of spatial databases, it is often desirable to perform clustering queries in the presence of natural or man-made obstacles (such as rivers, trees or buildings) (Tung, Hou, & Han, 2001). In the context of computer games meanwhile, e.g. as described in (Aha, Molineaux, & Ponsen, 2005), agents often

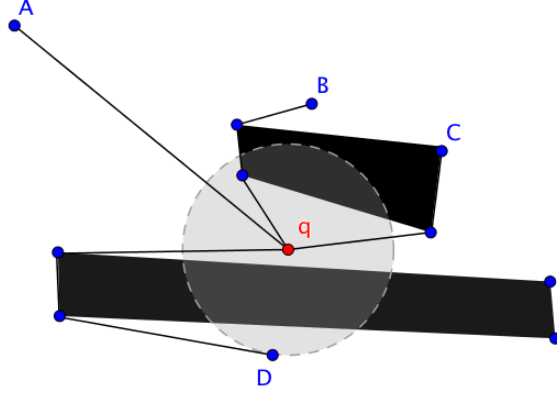


Figure 1: We aim to find the nearest neighbour of point q from among the set of target points A, B, C, D . Notice D is the nearest neighbor of q under the Euclidean metric but also the furthest neighbor of q when obstacles are considered.

rely on nearest-neighbour information, such as the location of enemies or resource points, for creating higher-level plans. Besides being a problem of direct interest, OkNN is also a fundamental sub-problem for a number of related spatial queries such as:

- **Obstacle Range Query (OR)** (Zhang, Papadias, Mouratidis, & Zhu, 2004): given a query point q and a range r , find all targets whose obstacle distance to q at most equal to r . OkNN algorithms can be applied to range queries by setting k to be infinite and terminating when the distance to the next closest target is greater than r .
- **Obstacle Reverse k-Nearest Neighbor (ORkNN)** and variants (Gao, Yang, Chen, Zheng, & Chen, 2011; Gao, Liu, Miao, & Yang, 2016)): given a query point q and a value k , return a set of targets t that have q as one of their k nearest neighbours: i.e. $\{t \mid q \in OkNN(t, k)\}$. ORkNN typically involves two stages: (i) *search stage*, which explores the space to obtain a set of candidates; (ii) *refine stage*, which removes from the set of candidates those for which q is not in the top k neighbours. Each of the two stages benefits directly from an efficient routine for OkNN.
- **Continuous Obstacle k-Nearest Neighbor (COkNN)** (Gao & Zheng, 2009): similar to OkNN, but the query point is generalised to a line segment. Solving such queries typically involves generating a list of so-called “split points”, and then invoking an OkNN query for each of these points.

Two popular algorithms for OkNN, which can deal with obstacles, are *local visibility graphs* (Zhang et al., 2004) and *fast filter* (Xia, Hsu, & 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 followed by an online Dijkstra search. 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. 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 a previous paper (Zhao, Taniar, & Harabor, 2018) we develop a new method for computing OkNN which avoids these disadvantages and which can improve runtime performance by several orders of magnitude. Our work extends *Polyanya* (Cui, Harabor, & Grastien, 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). In a range of experiments we show that each of our previously proposed techniques can be several orders of magnitude faster than *LVG* (Zhang et al., 2004), a state-of-the-art algorithm based on incremental visibility graphs.

Limitations in Our Previous Work. Our previous work described three multi-target variants of *Polyanya*, each of which offers state of the art performance but only in specific experimental scenarios:

- **Brute-force Polyanya**, a simple algorithm that invokes point-to-point Polyanya repeatedly, once for each target. Though extremely naive this algorithm was undominated in our experiments in scenarios with few targets and for values of $k > 2$.
- **Interval Heuristic** h_v , which replaces the point-to-point heuristic function of the original algorithm with a consistent and online alternative that minimises cost-to-go distance for all targets in the candidate set. In our experiments this algorithm was undominated in dense-target scenarios where the map contains many nearest neighbour candidates.
- **Target Heuristic** h_t , a costly pre-processing-based heuristic function which always chooses the Euclidean-nearest candidate for computing cost-to-go estimates. In our experiments this algorithm was undominated in sparse-target scenarios with small values of k .

Contributions. In this paper, we propose new preprocessing-based algorithms and heuristics for OkNN which improve on the performance of our previous work and which are more *robust*, in the sense that they perform well across a larger range of experimental scenarios including different target densities and for a wider range of values for k :

- We combine Polyanya with a technique known as Incremental Euclidean Restriction to derive a simple but very effective OkNN algorithm based on repeatedly solving point-to-point queries.
- We develop an efficient preprocessing framework that involves computing labels that we store with the edges of the navigation mesh. We exploit these labels during a subsequent online phase and derive two new OkNN query algorithms: (i) **Fence checking** which is very fast but only works when $k = 1$; (ii) **Fence Heuristic** h_f which works in the general case and which performs as well as and sometimes better than h_v and h_t , regardless of target density.

2. Related work

2.1 *R-tree* and traditional kNN

Traditional kNN queries in the plane (i.e. no obstacles) is a well studied problem for which there exists many well known and efficient spatial indexing schemes. Perhaps the most successful and well known of these is the *R-tree* (Guttman, 1984), a hierarchical data structure which organises a collection of spatial objects embedded in the plane (e.g. points or polygons) into a height-balanced tree. Though many variants exist (Sellis, Roussopoulos, & Faloutsos, 1987; Beckmann, Kriegel, Schneider, & Seeger, 1990; Kamel & Faloutsos, 1994), each of which improves the basic idea in some way, they all operate similarly and provide similar functionality. For this reason we opt to describe only the original work.

There are two types of nodes in an *R-tree*: leaves, which represent single objects, and interior nodes which are associated with a *Minimal Bounding Rectangle* (MBR). The MBR of an interior node contains the objects of all of its children and their associated MBRs. Meanwhile, nearest-neighbour queries involve a branch-and-bound traversal that depends on two important metrics:

- *mindist*, which bounds from below the minimum distance from the query point q to the closest object in the MBR of the current node;
- *minmaxdist*, which bounds from below the minimum distance from the query point q to the furthest object in the MBR of the current node.

The query algorithm starts at the root of the tree node and proceeds down, prioritizing nodes by *mindist* and pruning the search space by *minmaxdist*. Though improvements to this basic algorithm exist they employ a similar schema, albeit with different prioritisation and pruning strategies; e.g. (Roussopoulos, Kelley, & Vincent, 1995; Cheung & Fu, 1998).

2.2 OkNN Queries and Visibility Graphs

Solving point-to-point obstacle distance queries in main-memory is a well studied problem for which many algorithms exist (de Berg, 2000). Optimal methods usually pre-compute a visibility graph (VG), which adds edges between pairs of vertices that are co-visible. Figure 2 shows an example. Once the graph is created (including all candidate target points in the case of OkNN), queries can be resolved by way of A* or using Dijkstra’s well known algorithm. In case the query point is not one of the existing vertices of the graph, a further online insertion operation is performed before search can begin. The main drawback of visibility graphs is their worst-case time and space complexity which can be up n^2 where n is the number of vertices in the map (Ghosh & Mount, 1991). In spatial database scenarios, for example, n can be more than 10,000, so in-main-memory approaches are not suitable. Researchers in this field are thus motivated to design variant algorithms that only consider and process obstacles relevant to the current query. Two popular methods based on this idea are *Local Visibility Graphs* (LVG) (Zhang et al., 2004) and *Fast Filter* (Xia et al., 2004). We have previously discussed the strengths and weaknesses of these methods in Section 1.

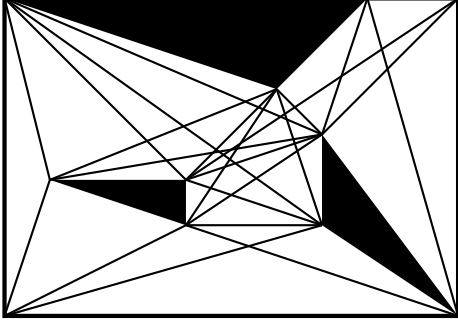


Figure 2: Example of a visibility graph. Black lines are edges in the visibility graph.

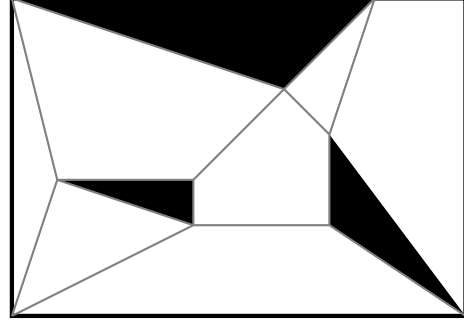


Figure 3: Example of a navigation mesh. Grey lines define individual mesh polygons.

2.3 Pathfinding on Navigation Mesh

A navigation mesh is a data structure that divides the traversable space into a set of convex polygons. Originally proposed by Arkin (Arkin, 1989), this spatial representation technique has been widely applied to pathfinding problems in areas such as robotics and computer games. Navigation meshes can be generated easily and efficiently. For example, *Constrained Delaunay Triangulations* (Chew, 1989), a popular starting point for many mesh-based algorithms, can be constructed in $O(n \log n)$ time. In other settings, such as computer games, navigation meshes are available as input “for free”, having been created, at least in part, by human designers.

Until recently pathfinding algorithms developed for navigation meshes have typically lacked both strong performance and optimality guarantees (Kallmann, 2005; Demyen & Buro, 2006). *Polyanya* (Cui et al., 2017) is a recent navigation mesh algorithm which changes the status quo by being *compromise-free*: i.e. simultaneously fast, optimal and (assuming the mesh is given as input) entirely online. We give a technical description of this algorithm in Section 4. Our previous work on the topic of OkNN (Zhao et al., 2018) extends *Polyanya*, from point-to-point pathfinding to the multi-target case.

3. Problem Statement

OkNN is a nearest-neighbour search in two dimensions that can be formalised as follows:

Definition 3.1. Obstacle k -Nearest Neighbour (OkNN): Given a set of points T , a set of obstacles O , a distinguished point q 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 .

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 $\pi_{q,t} = \langle q, \dots, t \rangle$ between points q and t such that no pairwise segment of the path intersects any point inside an obstacle (i.e. “obstacle avoiding distance”).

Solution approaches for OkNN can be broadly categorised into two schemas:

- **Linear search methods**, which involve the repeated application of a point-to-point pathfinding algorithm, from q to selected $t \in T$. Linear search methods terminate when it can be proven that the obstacle distance to all k nearest neighbours has been found. Instantiations of this schema include the OkNN algorithms known as *LVG* (Zhang et al., 2004) and *fast filter* (Xia et al., 2004).
- **Spatial partitioning methods**, which consider all candidate targets simultaneously and which attempt to solve a given instance of kNN in a single integrated search. Such methods apply prioritised search (e.g. best first or branch-and-bound) in combination with specialised heuristics that prune the set of candidates and focus attention toward the most promising target first. Instantiating this schema are a variety of Euclidean kNN methods such as *R-tree* (Guttman, 1984) and *Spatial KD-tree* (Ooi, 1987).

In our previous work we describe three algorithms for OkNN: *Brute-force Polyanya*, which instantiates linear search, and *Polyanya* in combination with the heuristics h_t and h_v , both of which instantiate the spatial partitioning search schema.

4. Polyanya and heuristic functions: h_p , h_v , h_t

In its canonical form Polyanya is a point-to-point Euclidean shortest path algorithm which can be seen as an instance of A*: it performs a best-first search using an consistent heuristic function to prioritise nodes for expansion. 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:

- **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 $p \in I$. Fig 4 shows an example.

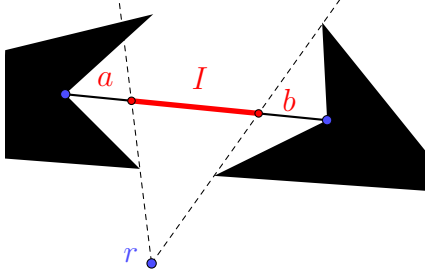


Figure 4: Search nodes in Polyanya. Notice that the interval $I = [a, b]$ is a contiguous subset of points drawn from an edge of the navigation mesh. The corresponding root point, r , is either the query point itself or the vertex of an obstacle. Taken together they form the search node (I, r) .

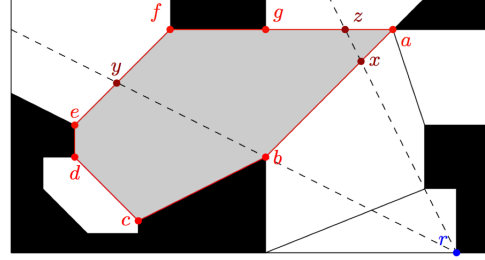


Figure 5: From (Cui et al., 2017). We expand the node $([b, x], r)$ which has $([z, g], r)$ and $([f, y], r)$ as observable successors. In addition, the nodes $([c, d], b)$, $([d, e], b)$ and $([e, y], b)$ are non-observable. All other potential successors can be safely pruned (more details in (Cui et al., 2017)).

- **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 . Fig 5 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** using h_p : 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 Fig 6. The objective in each case is to choose the unique $p \in I$ that minimises the estimate. The three cases together are sufficient to guarantee that the estimator is consistent.

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 suboptimal competitors, often by orders of magnitude.

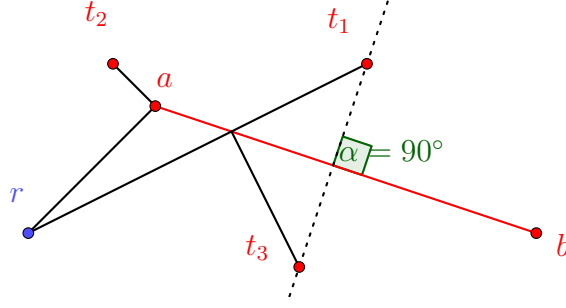


Figure 6: Polyanya f -value estimator. The current node is (I, r) with $I = [a, b]$ and each of t_1, t_2, t_3 are possible target locations. **Case 1:** the target is t_1 . In this case the point $p \in I$ with minimum f -value is at the intersection of the interval I and the line $r \rightarrow t_1$. **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.

4.1 Interval Heuristic h_v

In some OkNN settings targets are myriad and one simply requires a fast algorithm to explore the local area. The idea we introduce for such settings takes the form of a specialised heuristic function h_v which we combine with *Polyanya*. It can be formalised as follows:

Definition 4.1. Given search node $n = (I, r)$, the interval heuristic $h_v(n)$ is the minimum Euclidean distance from r to any point $p \in I$.

The Interval Heuristic h_v is consistent (Zhao et al., 2018), and applying it only requires solving a simple geometric problem: finding the closest point on a line. Since this operation is trivial h_v has a low computation overhead and can be applied entirely online. Stated differently, h_v discards all spatial information about any target and performs instead a *Dijkstra*-like search which can quickly explore surrounding polygons. When targets are many and randomly distributed this approach can be highly efficient. The main drawback is redundant search effort in non-target areas, such as can occur when targets are few and sparsely distributed.

4.2 Target Heuristic h_t

In some OkNN settings the targets are sparse, so we need an effective method to prune the search space. The idea we introduce for such settings again takes the form of a specialised and consistent heuristic, h_t , which we again combine with *Polyanya*.

Definition 4.2. Given search node $n = (I, r)$, the target heuristic $h_t(n)$ is:

$$h_t(n) = \min\{h_p(n, t) | t \in T\}$$

The idea is simple: each time *Polyanya* expands a node h_t computes a cost-to-go it does so with respect to the Euclidean-nearest candidate which has not yet been added to the k nearest neighbour set. Selecting such a target involves four nearest neighbour queries

which we solve using an R -tree. This approach computes all k nearest neighbours in a single run. Compared to h_v , this heuristic can significantly reduce the number of node expansions because it always drives the search in the direction of the most promising candidate point. This approach performs well when target are sparse. The main drawback is the high cost of computing nearest neighbours when targets are numerous, dense and found in any direction.

5. Proposed Methods

We propose two distinct ideas for improving OkNN Search. The first idea involves computing, during an offline pre-processing step, a set of “fence labels” for each edge of the navigation mesh. The labels bound the distance from the edge to the nearest point in the target candidates set. We exploit the labels to improve the performance of a subsequent online search. The second idea we propose involves running point-to-point Polyanya, from the query location and to each target in the candidate set, in the order specified by a preprocessing-based heuristic technique known as Incremental Euclidean Restriction.

5.1 Fence Labelling

Interval heuristic h_v can give us the minimum obstacle distance from the root to an edge of the mesh, an interesting observation is that if the algorithm have explored the entire map, then each edge of meshes must contains such information. Furthermore, if we ran such full-map exploration starting with each target and store obtained information, we would utilize such information for actual queries.

The described idea cannot be efficient, in spite of the time complexity, we need at least $O(ET)$ space to store obtained information, where E is the number of mesh edges, and T is the number targets.

However, similar to traditional nearest-neighbour query (Roussopoulos et al., 1995; Cheung & Fu, 1998), we can apply *minmaxdist* pruning in this case.

Definition 5.1. Given a mesh edge (A, B) , a search node $n = (I, r)$ where $I = (a, b)$ and a (resp. b) is the endpoint of I closer to A (resp. B), then:

- **mindist** = $h_v(r, I)$,
- **minmaxdist** = $\max(d_e(r, a) + d_e(a, A), d_e(r, b) + d_e(b, B))$

In other words, for a given root r , the length of shortest path to any point $p \in (A, B)$ must in range $[\text{mindist}, \text{minmaxdist}]$, see an example in Fig 7.

To distinguish the g -value in preprocessing and search, let’s define g_p as follow:

Definition 5.2. Let $n = (I, r)$ be a search node produced by target t in preprocessing:

$$g_p(r) = d_o(r, t)$$

Definition 5.3. Dominate: Given two search nodes $n_1 = (I_1, r_1)$ and $n_2 = (I_2, r_2)$, where I_1 and I_2 are on same mesh edge (A, B) , n_1 dominates n_2 iff $g_p(r_1) + \text{minmaxdist}(n_1) \leq g_p(r_2) + \text{mindist}(n_2)$.

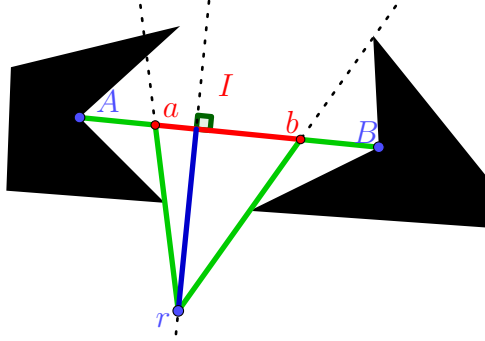


Figure 7: A, B are endpoints of the mesh edge, the length of blue path is the *mindist*, the length of one of green path is the *minmaxdist*

Lemma 5.1. Let $n_1 = (I_1, r_1)$ and $n_2 = (I_2, r_2)$ be two search nodes in preprocessing stage, where I_1 and I_2 are on same mesh edge (A, B) , and n_1 dominates n_2 ; for a given query point q , let m be any point on (A, B) . Then:

$$d_o(q, m) + d_o(m, r_1) + g_p(r_1) \leq d_o(q, m) + d_o(m, r_2) + g_p(r_2)$$

Proof. We have two equations:

$$g_p(r_1) + d_o(q, m) + d_o(m, r_1) \tag{1}$$

$$g_p(r_2) + d_o(q, m) + d_o(m, r_2) \tag{2}$$

then $(1) - (2)$ is:

$$(g_p(r_1) + d_o(m, r_1)) - (g_p(r_2) + d_o(m, r_2))$$

according to definition 5.1, we have:

$$g_p(r_1) + \text{minmaxdist}(n_1) \geq g_p(r_1) + d_o(m, r_1) \tag{3}$$

$$g_p(r_2) + d_o(m, r_2) \geq g_p(r_2) + \text{mindist}(n_2) \tag{4}$$

and because n_1 dominates n_2 , we have:

$$g_p(r_1) + d_o(m, r_1) \leq g_p(r_2) + d_o(m, r_2) \tag{5}$$

thus $(1) - (2) \leq 0$. \square

Lemma 5.1 implies that, in preprocessing, we can discard those search nodes who is dominated by others, let's call this *fence pruning*.

Definition 5.4. **Fence** lies on each mesh edge, it contains a collection of search nodes N produced by preprocessing, let's call those search nodes **labels**¹. **Fence** has a property **upper-bound** = $\min\{g_p(n.r) + \text{minmaxdist}(n) | n \in N\}$ (default value is INF), and in preprocessing stage, it blocks all passing search nodes $n' = (I', r')$ that:

$$g_p(r') + \text{mindist}(n') \geq \text{upper-bound}$$

.

1. To make discussion clear, we call this "search node" in context of preprocessing, and "label" in context of query processing

Algorithm 1: Preprocessing

Input: T (targets set), q (open list)

```

1 foreach  $t$  in  $T$  do
2   foreach  $suc$  in  $successor(t)$  do
3      $q.push(suc)$ ;
4   end
5 end
6 while  $q$  not empty do
7    $n = q.pop()$ ;
8   // fence pruning;
9    $fence = get\_fence(n.I)$ ;
10   $lowerbound = g_p(n.r) + mindist(n)$ ;
11  if  $lowerbound \geq fence.upperbound$  then
12    continue;
13  end
14   $upperbound = g_p(n.r) + minmaxdist(n)$ ;
15   $fence.upperbound = \min(fence.upperbound, upperbound)$ ;
16  foreach  $suc$  in  $successors(n)$  do
17     $q.push(suc)$ ;
18  end
19 end
    
```

Finally we propose a *floodfill*-like algorithm: in initialization, for each target $t \in T$, generate successors on the edges of mesh that contains t , and store those search nodes in a open-list; similar to OkNN search, we expand search nodes n in the order of their f -value, where $f(n) = g(n) + h_v(n)$; meanwhile, we apply *fence pruning* and update *upper-bound* of the fence before the expansion; the algorithm terminates when open-list becomes empty, see in algorithm 1.

Notice that the total number of search nodes can still up to $O(ET)$, one case is that targets are distributed in a single cluster, and all obstacles line in two rows, so that all targets produce labels on mesh edges in the middle (see in Fig 8b). However, our experiment in section 6 indicates that most of fence only contains 2 to 3 labels (search nodes), and there are two observations to convince us the effectiveness of preprocessing (see in Fig 8b):

- If two targets are very close, they may produce labels on same fence, but their successors must be pruned by one of other when root changes (known as *root pruning* (Cui et al., 2017));
- If two targets are not close, they will dominate surrounding mesh edges and block search nodes from further targets.

5.2 Apply on nearest-neighbour query

After preprocessing, for a given query point q , there are two cases:

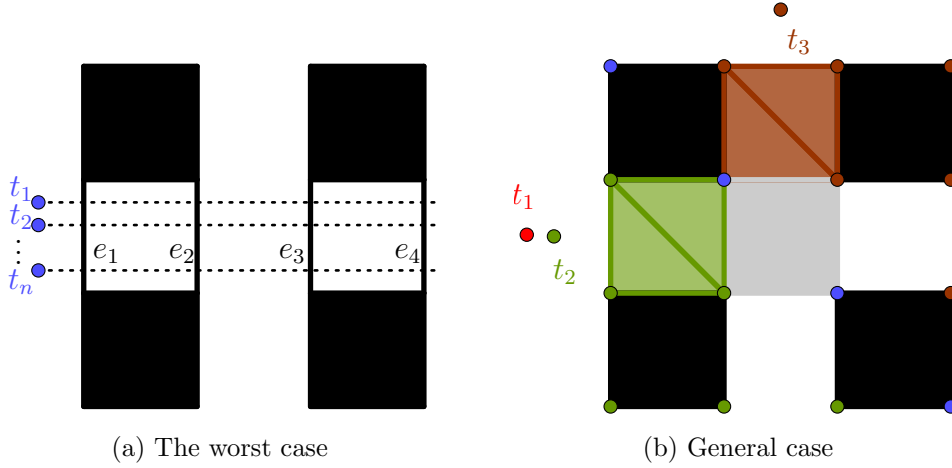


Figure 8: **(a)**: t_1, t_2, \dots, t_n have same distance to edges in the middle, so fence of these edges contain labels from all targets. **(b)**: t_1, t_2 are close and produce labels on same fence in green area, but t_1 has no labels after root change since all vertices are dominated by t_2, t_3 ; search nodes of t_2 (resp. t_3) can not reach brown (resp. green) area which is dominated by t_3 (resp. t_2).

- Case 1: there's no labels on surrounding fences;
- Case 2: all surrounding fences contain labels;

The Case 1 implies that there's no path from q to any target $t \in T$;

Lemma 5.2. *In the Case 2, at least one of label is produced by the obstacle nearest-neighbour of q .*

Proof. Assume the obstacle nearest neighbour of q is t , and surrounding fences of q not contain any label produced by t , then:

- Case 1: there's no path from t to q , which contradict to the assumption;
- Case 2: all successors of t are dominated by others, and according to lemma 5.1, t can not be the nearest-neighbour of q , which contradict to the assumption as well.

Thus, such t doesn't exists. □

Therefore, we can run point-to-point search from q to all roots of search nodes in surrounding fences, and find the nearest-neighbour, we call this algorithm *Fence Checking*, see details in algorithm 2. Notice that algorithm 2 only search to the last vertex instead of the entire path, so it is very fast.

5.3 Generalize to k-nearest neighbours

When $k > 1$, search nodes of next nearest-neighbour may be dominated by a visited nearest-neighbour and thus no labels on surrounding fences. So we still need a search process to

Algorithm 2: Fence Checking algorithm

Input: q (query point), $poly$ (mesh polygon contains q)
 1 roots = set();
 2 min_dist = INF;
 3 nearest_neighbor = null;
 4 **foreach** $edge: poly.edges$ **do**
 5 **foreach** $node: edge.fence$ **do**
 6 roots.add(node.r);
 7 **end**
 8 **end**
 9 **foreach** $r: roots$ **do**
 10 $d_o = g(r) + poly.any.run(q, r)$;
 11 **if** $d_o < min_dist$ **then**
 12 min_dist = d_o ;
 13 nearest_neighbor = start_point(r);
 14 **end**
 15 **end**

explore the map, and we need to design heuristic function to prioritise search nodes in query processing. We first present an admissible but not consistent heuristic *naive fence heuristic*: h'_f ; then we show how to adapt it to a consistent heuristic *fence heuristic*: h_f .

Definition 5.5. Naive fence heuristic h'_f : let F be the fence lies on mesh edge (A, B) , $n' = (I', r')$ be a label on F , $n = (I, r)$ be a search node produced by query point q , where I on (A, B) , then

$$h'_f(n) = \min\{h_p(n, r') + g_p(r') | n' \in F\}$$

Stated in simple words, h'_f evaluates the distance from current root r to the nearest target through interval I and r' .

Theorem 5.3. h'_f is admissible, more specific: given a search node $n = (I, r)$, for $\forall t' \in T$:

$$f(n) = g_p(r) + h'_f(n) \leq d_o(q, r) + d_e(r, m) + d_o(m, t') (m \in I)$$

Proof. Given a search node $n = (I, r)$, let $n' = (I', r')$ be the label on fence F that used by $h'_f(n)$: $h'_f(n) = h_p(n, r') + g_p(r')$ and n' is produced by a target t' ; and let p be the shortest path from q to any target t pass through r and I ; there are three cases:

- If $t = t'$, then we have

$$|p| = g(r) + d_e(r, m) + d_o(m, r') (m \in I)$$

and according to definition of h_p (Fig 6), we have $h_p(n, r') \leq d_e(r, m) + d_o(m, r') (m \in I)$, thus $|p| - f(n) = d_e(r, m) + d_o(m, r') - h_p(n, r') \geq 0$;

- If $t \neq t'$ and t has label $n_t = (I_t, r_t)$ on the F , according to definition 5.5, we have $f(n) \leq g(r) + h_p(n, r_t) + g_p(r_t) \leq |p|$;

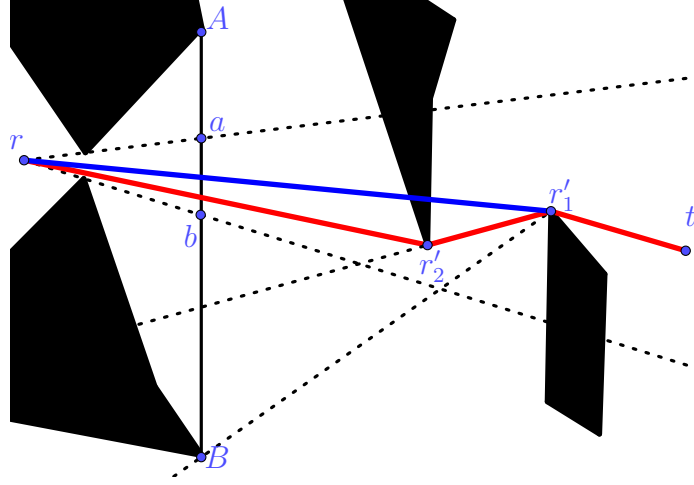


Figure 9: (A, B) is the edge of mesh, the current search node is $n = (I, r)$ where $I = (a, b)$, both r'_1 and r'_2 have labels on the fence that lies on (A, B) , red segments indicate the actual path, while the blue segment shows the path after the expansion. Before the expansion, $f(n) = g(r) + d_o(r, t)$, since $h_p(n, r'_1) + g(r'_1) < h_p(n, r'_2) + g(r'_2)$, after the expansion, the estimation becomes $g(r) + d_e(r, r'_1) + d_o(r'_1, t)$ which is less than the previous value.

- If $t \neq t'$ and none of t 's label on the F , then they must be dominated by some fences, according to lemma 5.1, we still have $f(n) = g(r) + h'_f(n) \leq |p|$.

Thus, h'_f is admissible. \square

Notice that $f(n) = |p|$ when r, r' are co-visible, and otherwise it's an underestimation. Such observation indicates that h'_f may not consistent, Fig 9 shows an example.

Therefore, when we expand a search node $n = (I, r)$, those labels $n' = (I', r')$ who lay on the fence but r, r' are not co-visible may ruin the consistency. To fix this problem, instead of checking visibility in h'_f , we can simply keeping the maximum f -value during the search.

Definition 5.6. Fence heuristic h_f : for a given search node $n = (I, r)$, h_f guarantees

$$g(r) + h_f(n) = \max(f_{parent}, g(r) + h'_f(n))$$

where f_{parent} is the f -value of the parent search node.

Theorem 5.4. *The h_f is consistent, more specific, h_f is admissible and for a given search node n and it's successor n' :*

$$f(n) \leq f(n')$$

Proof. Let $n = (I, r)$ be the current search node, and $n' = (I', r')$ be any successor. First, we show that h_f is admissible by mathematic induction:

- Initially, when n doesn't has any parent, $f(n) = g(r) + h'_f(n)$ which is admissible according to theorem 5.3;
- Assume h_f for n is admissible during the search;

Algorithm 3: OkNN search with h_f

```

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  successors = genSuccessors(node);
11  foreach suc in successors do
12    suc.g = node.g +  $d_e(\text{node.root}, \text{suc.root})$ ;
13    suc.f = max(node.f, suc.g +  $h'_f(\text{suc})$ );
14    heap.push(suc);
15  end
16 end
17 Function  $h'_f(\text{node})$ :
18   res = INF
19   foreach s in Fence(node.I) do
20     | res = min(res,  $h_p(\text{node}, \text{s.root}) + \text{s.root.g}$ )
21   end
22   return res
    
```

- For successor n' , if $f(n') = f(n)$ then h_f is admissible;
- Otherwise $f(n') = g(r') + h'_f(n')$ which is also admissible;

Thus h_f is admissible, and according to definition 5.6, $f(n') \geq f(n)$ is always true, so it is consistent. \square

Finally, we implement the fence heuristic search in algorithm 3.

5.4 IER-Polyanya: Polyanya with Incremental Euclidean Restriction

From previous experiments (Zhao et al., 2018), we notice that the **brute-force Polyanya**, a naive adaption of *Polyanya* that running point-to-point search for each target, outperforms other sophisticated competitors under certain scenarios. Such positive results motivate us to design more efficient algorithm.

Notice that for same pair of points (a, b) , we always have $d_e(q, a) \leq d_o(q, b)$, this is called *Euclidean lower-bound property* (Zhang et al., 2004). Utilize such property, we can prune the search space, similar ideas also appeared in previous literatures (Zhang et al., 2004; Xia et al., 2004; Abeywickrama, Cheema, & Taniar, 2016). We implement this idea as follow: given query point q , the algorithm use *incremental nearest neighbor* query in *R-tree* (Hjaltason & Samet, 1999) to visit target $t \in T$ in order of $d_e(q, t)$, then use Polyanya

Algorithm 4: Repeated Polyanya

Input: T (targets set), q (query point)

```

1 candidates = Heap();
2 rtree = Rtree( $T$ );
3  $t$  = rtree.iNearestNeighbor( $q$ );
4 while  $t$  is not null do
5    $d_o$  = Polyanya.run( $q$ ,  $t$ );
6   if less than  $k$  candidates then
7     candidates.add( $d_o$ );
8   else
9     if  $\max(\text{candidates}) \leq d_e(q, t)$  then
10      break;
11    end
12    if  $\max(\text{candidates}) > d_o$  then
13      candidates.remove_max();
14      candidates.add( $d_o$ );
15    end
16  end
17   $t$  = rtree.iNearestNeighbor( $q$ );
18 end

```

to compute $d_o(q, t)$ and keep track k -nearest neighbours with d_o metric; the algorithm terminates when $d_e(q, t) > d_o(q, t_k)$ where t_k is the current k -th obstacle nearest neighbour of q , see in algorithm 4.

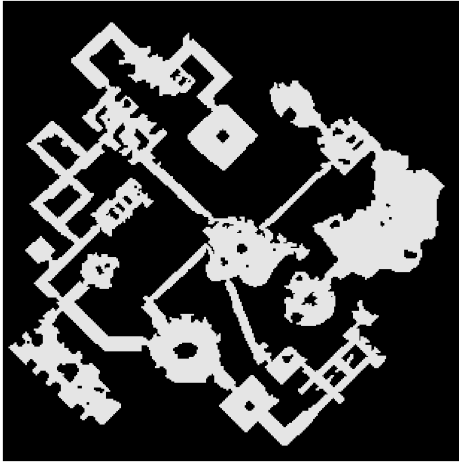
Lemma 5.5. *When algorithm 4 terminates, the candidates stores top- k smallest obstacle distances.*

Proof. Assume there is an unvisited t' that $d_o(q, t') < d_o(q, t_k)$, where t_k is the k -th nearest neighbor. Since t' is unvisited, we have $d_e(q, t') \geq d_o(q, t_k)$; according to *Euclidean lower-bound property*, we have $d_e(q, t') \leq d_o(q, t')$ and $d_e(q, t_k) \leq d_o(q, t_k)$, therefore we have $d_o(q, t') \geq d_o(q, t_k)$ which contradict to the assumption, so such t' doesn't exist. \square

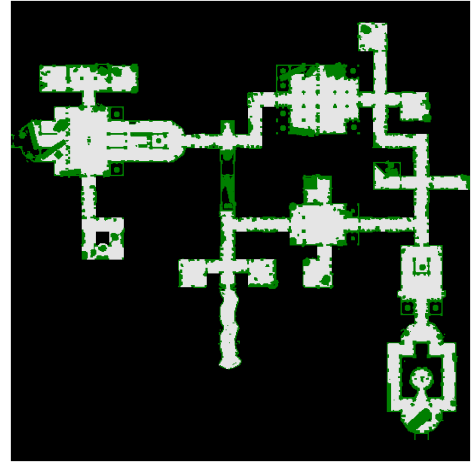
The effectiveness of *IER-Polyanya* affected by two factors:

- **False hit:** a searched target not being retrieved is called false hit. False hit is more likely happen when Euclidean metric is misleading, for example in Fig 1, D is the first target to search but wouldn't be retrieved when $k \leq 3$.
- **Overlapping search space:** since each search is independent, same search space may be explored multiple times.

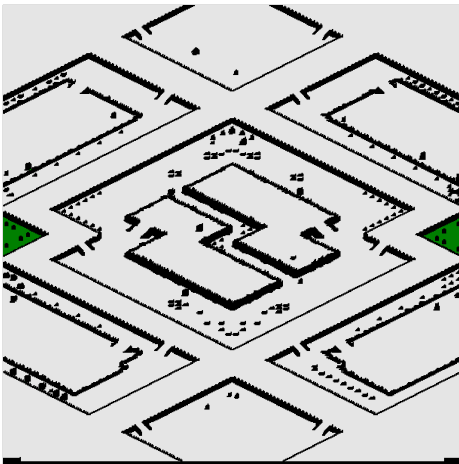
In section 6, we will examine the performance of *IER-Polyanya* based these considerations.



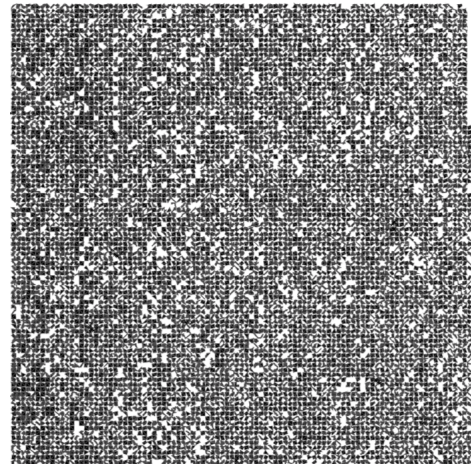
(a) AR0602SR. This map has 47 obstacles and 5504 vertices. The navigation mesh comprises 5594 polygons.



(b) brc202d. This map has 75 obstacles and 4035 vertices. The navigation mesh comprises 4164 polygons.



(c) CatwalkAlley. This map has 157 obstacles and 15301 vertices. The navigation mesh comprises 15482 polygons.



(d) 8K tiled obstacles. This map has 8385 obstacles and 109171 vertices. The navigation mesh comprises 125937 polygons.

Figure 10: Maps and summaries. NB: black and green indicates non-traversable areas.

6. Empirical Analysis

OkNN problem often appears in both AI pathfinding and spatial query preprocessing, so we examine the performance of proposed methods on different maps that corresponding two these two application scenarios. Fig 10 shows the set of maps that we consider in our experimental evaluation.

We choose three maps from a well known AI pathfinding benchmark sets (Sturtevant, 2012): *AR0602SR*, *brc202d*, *CatwalkAlley*; and a synthetic map *8K tiled obstacles* from our

Parameters	Values
k	1, 5 , 10, 25, 50
Maps	AR0602SR, brc202d, CatwalkAlley, 8K-tiled-obs
Target Density (d)	0.0001, 0.001, 0.01 , 0.1
Target distribution	random , clustered

Table 1: Parameters (default in bold)

previous work (Zhao et al., 2018)², the way we created such synthetic map is described below:

- **Tiled obstacles map.** we extract the shape of all parks in Australia from *OpenStreetMap* (OpenStreetMap contributors, 2017) and use these shapes as polygonal obstacles. There are initially 9000 such polygons³, after removing invalid polygons, there are 8385 polygons left. 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.

The navigation mesh of map is generated by *Constrained Delaunay Triangulation*, which is $O(n \log n)$; the implementation of such algorithm is in library *Fade2D*⁴, the total time on such preprocessing is about 6s.

In each query processing experiment, we choose parameters from Table 1. Notice that we define *target density* by $|T|/|V|$ where $|V|$ is the number of vertices, while other works (Zhang et al., 2004; Gao & Zheng, 2009; Gao et al., 2016) used $|T|/|O|$ as their density where $|O|$ is the number of obstacles. The reason is that game maps usually have few large continuous obstacles.

We’re using 1000 random query points for each setting, grouping results by *x-axis*, and computing average; the size of each bucket is at least 10; All algorithms appear in experiments are:

- LVG: Local Visibility Graph (Zhang et al., 2004);
- h_v : Interval heuristic (Zhao et al., 2018);
- h_t : Target heuristic (Zhao et al., 2018);
- h_f : Fence heuristic (Algorithm 3);
- fc: Fence Checking (Algorithm 2);
- IER-Poly: Polyanya with Incremental Euclidean Restriction (Algorithm 4);

2. There are few real datasets (e.g. <http://www.rtreportal.org/>), but all expired that not publicly available

3. In our previous (Zhao et al., 2018), it’s called 9000 obstacles map.

4. <http://www.geom.at/fade2d/html>

map	vertices	mean	std	min	25%	50%	75%	max
AR0602SR	5504	6.92	0.14	6.72	6.84	6.92	6.98	7.21
brc202d	4164	6.11	0.25	5.63	5.98	6.09	6.25	6.54
CatwalkAlley	15301	26.73	2.35	22.83	25.55	26.15	28.49	30.87
8K-tiled-obs	109171	537.16	24.94	499.93	522.24	537.60	552.43	580.43

Table 2: Processing time (ms)

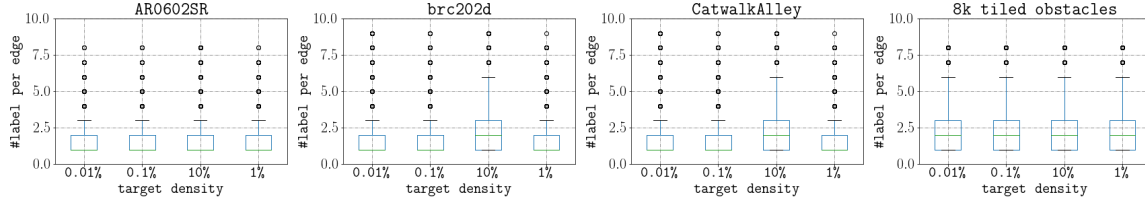


Figure 11: Number of labels per edge

These algorithms are implemented in C++ and 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 available⁵. All experiments are performed on a 2.5 GHz Intel Core i7 machine with 16GB of RAM and running OSX 10.13.4.

6.1 Experiment 1: preprocessing

This experiment is to examine the performance of preprocessing. For each map, we generated random distributed target set with four density and run floodfill with fence pruning (algorithm 1).

Time cost. We collect the execution time of each instance and group by map, each group contains data from different density. Tabel 2 shows that the *std* of each group is relative small but the mean of groups are very different, so we can conclude that the time cost of preprocessing is not sensitive to density of targets but the map itself (e.g. number of vertices).

Label size. This metric indicates the space cost and the performance during the query processing. We collect the number of labels on each mesh edge and group by map, then in each group, we further group data by density⁶. Fig 11 shows that for all maps and all densities, medians of label size per edge are similar.

6.2 Experiment 2: random distributed targets

The aim of this experiment is to examine the performance of proposed algorithms when k and d (density) change, by default, $k = 5$ and $d = 0.01$. The effectiveness of heuristic function is measured by total search time divide by number of generated node, so for same search time, larger heuristic cost meaning smaller search space. We also include the

5. <http://bitbucket.org/dharabor/pathfinding>

6. There are $\approx 1\%$ of edges have more than 10 and up to 300 labels, we filter these out for better visualisation

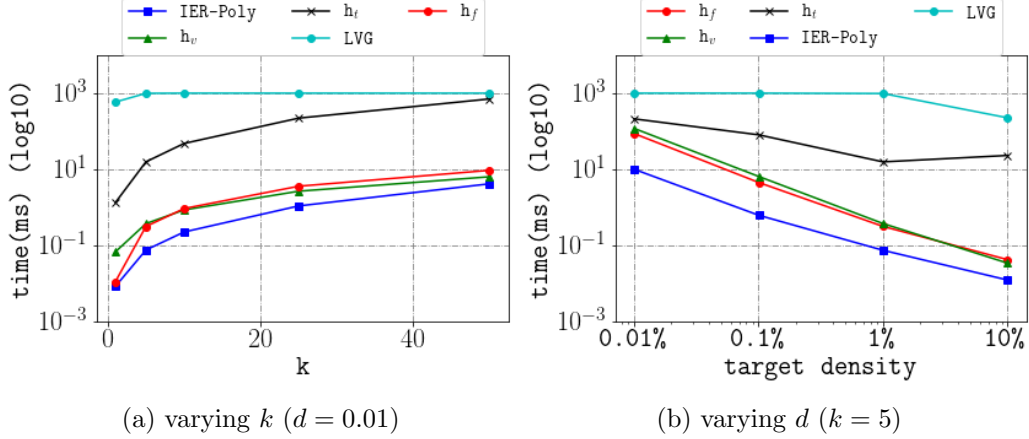


Figure 12: Performance on the default map (8K tiled obstacles)

previous state-of-the-art *LVG* in comparison, and since this method may reach quadratic time complexity under some scenarios, we clip the execution time by 10^3 ms.

From Fig 12 and Table 3, we make the following observations:

- **IER-Polyanya** always outperforms others, and when k is fixed it's nearly order of magnitude faster than h_f and h_v . The reason is that Euclidean metric in this map with random distributed targets is very good, so it has small number of false hit and thus get benefit from the fast point-to-point search.
- **Fence heuristic** has similar number of generated search nodes to h_t regarding the heuristic cost, and in general, it also has similar time cost to h_v . Besides, it significantly outperforms h_v when $k = 1$ and d is fixed. So we can conclude that h_f has both good time cost and small number of generated nodes.
- **LVG** is completely dominated by others in all cases, so we can remove this competitor in further experiments.

heuristic	mean	std	min	25%	50%	75%	max
h_f	0.82	0.19	0.05	0.72	0.80	0.91	2.83
h_p	0.26	0.07	0.02	0.23	0.25	0.29	1.77
h_t	162.19	262.96	9.23	71.29	93.23	112.16	8191.00
h_v	0.42	0.12	0.08	0.34	0.39	0.44	1.42

Table 3: The average cost (μ s) of each heuristic function

6.3 Experiment 3: clustered targets

One drawback of *IER-Polyanya* is that each repeated search is independent, so that it makes many redundant computation when those searches are overlapped. The aim of this

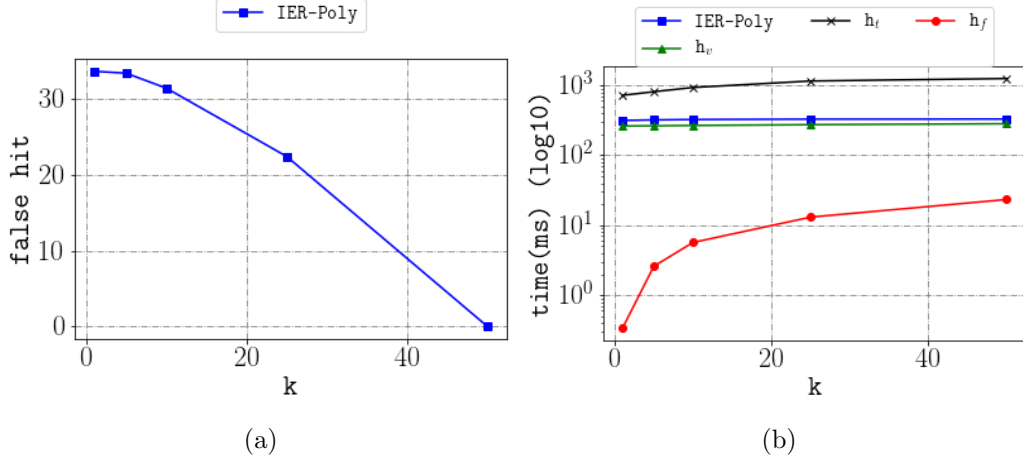


Figure 13: Performance on clustered targets

experiment is to examine the performance of *IER-Polyanya* when search space overlapped. We create a single cluster of targets, and to get rid of the influence of false hit, we let the size of cluster be 50, so that when $k = 50$ the false hit becomes 0.

From Figs 13a,13b we make following observations:

- when $k < 50$, *IER-Polyanya* is outperformed by h_f because the false hit;
- when $k = 50$, false hit is 0, but *IER-Polyanya* is still significantly outperformed by h_f , while in previous experiment, it's faster than h_f for all k . So we can conclude that *IER-Polyanya* has bad performance when search space overlapped.

6.4 Experiment 4: varying maps

Another drawback of *IER-Polyanya* is that the Euclidean metric can be misleading in some maps. The aim of this experiment is to examine the performance of *IER-Polyanya* on different types of maps.

We run 1000 random queries with default parameter $k = 5, d = 0.01$ where targets are randomly distributed. To show the influence of misleadingness, we analyze results as follow:

- We use the number of nodes generated by h_v measure the difficulty of search;
- For each search, we evaluate the speed-up factor of each method by time cost of such method divide by time cost of h_v ;
- We plot the number of false-hit and the speed-up factor with increasing difficulty for each map.

Such speed-up comparison also appears in other pathfinding literature (Harabor, Grastien, Öz, & Aksakalli, 2016).

From fig 15 we make following observations:

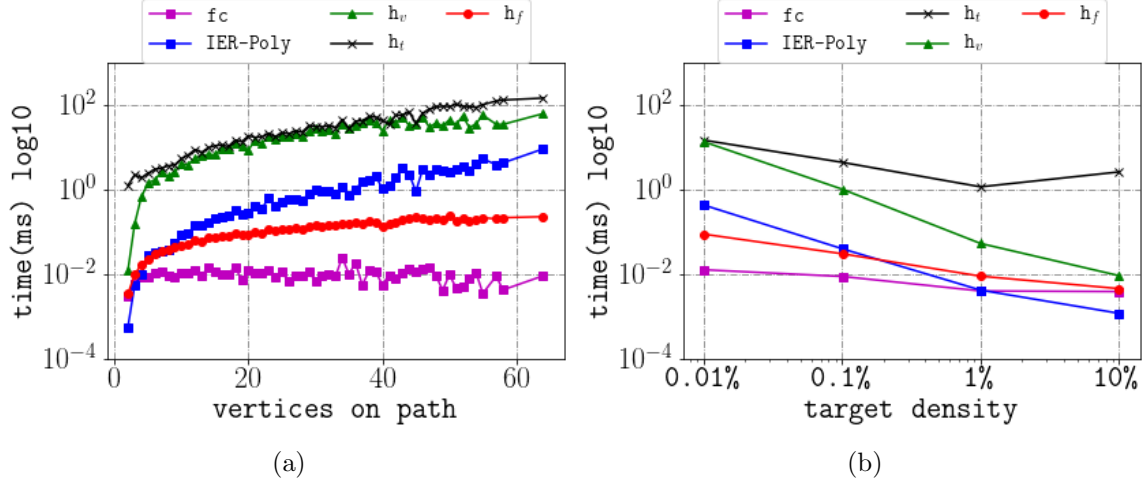


Figure 14: Performance on nn query

- In all maps, when false hit increases the speed-up factor decreases, meaning that false hit affect the performance of *IER-Polyanya*.
- *AR0602SR*, *brc202d*, *CatwalkAlley* have more false hit than *8K tiled obstacles*, meaning that Euclidean metric is misleading in those maps.
- In those misleading maps, speed-up factor of *IER-Polyanya* is relative small, and sometimes less than 1 (worse than h_v), meaning that *IER-Polyanya* doesn't work well when Euclidean metric becomes misleading.

6.5 Experiment 5: Nearest neighbor query

The advantage of *fence checking* is only searching the last vertex of the shortest path, the aim of this experiment is to examine when it performs well. In the experiment, we run 1000 random queries with different densities, then we analyze results in two aspects: (i) grouping results by number of vertices on path to examine the performance when it gets advantage; (ii) grouping results by d to examine the general performance in different density.

Fig 14a shows that *fence checking* is not sensitive to the number of vertices on path and can be orders of magnitude faster than other competitors when it gets advantage.

Fig 14b shows that *fence checking* outperforms others when $d < 0.01$, the reason is that when density increases, the number of vertices on shortest path decrease, thus *fence checking* gradually lose the advantage.

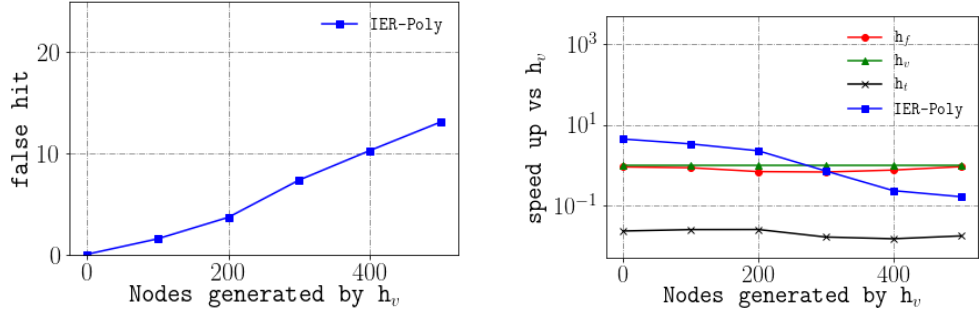
7. Conclusion and Future Work

We study the obstacle k-nearest neighbours problem which appears in both AI pathfinding and spatial database areas. In this paper, we proposed two efficient methods: **fence heuristic** and **IER-Polyanya**.

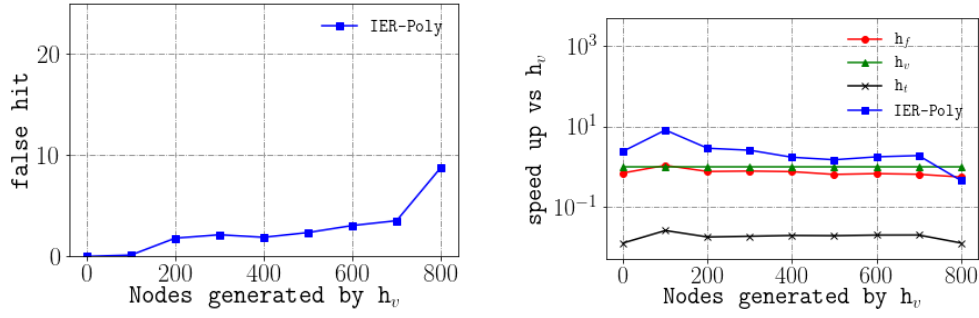
Fence heuristic utilizes precomputed information to guide the search process, compare to h_v and h_t in our previous work, it has both cheap heuristic cost and small search space,

and it performs well in whatever target density. The preprocessing for **Fence heuristic** has bad theoretical upper bound, but it is efficient in practical when a pruning strategy be applied. **IER-Polyanya** utilizes a fast point-to-point search to compute obstacle distance and Euclidean restriction to prune candidate set, it sometimes outperforms **Fence heuristic** order of magnitude, but such performance is not stable that sensitive to the distribution of targets and obstacles.

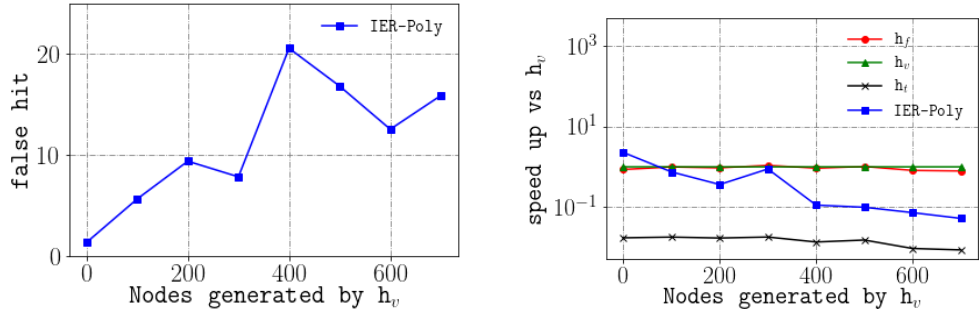
There are several possible directions for future work. Firstly, current **Fence heuristic** performs a linear search for labels on fence to compute the heuristic value, an obvious improvement is to store labels by their interval, and during the search, only retrieval labels with overlapped interval (meaning visible). Secondly, notice that adding or deleting targets only affect a small area of the map, so that we can develop these operations to efficiently maintain fence labels when targets change. Thirdly, one possible way to deal the worst case in preprocessing is reconstructing part of the navigation mesh.



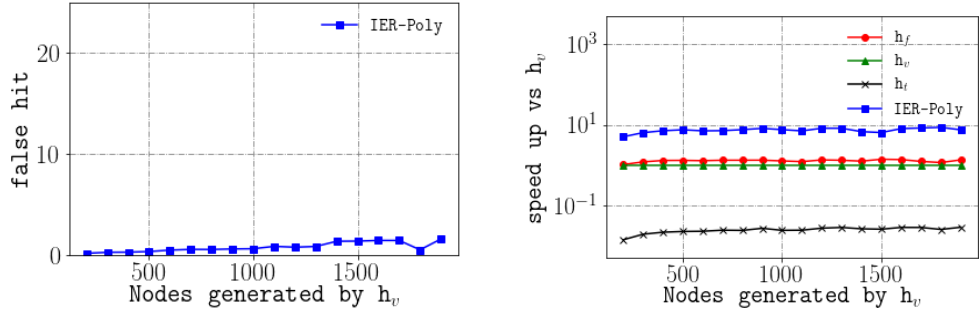
(a) AR0602SR



(b) brc202d



(c) CatwalkAlley



(d) 8K tiled obstacles

Figure 15: Performance on different maps

References

- Abeywickrama, T., Cheema, M. A., & Taniar, D. (2016). k -nearest neighbors on road networks: A journey in experimentation and in-memory implementation. *PVLDB*, 9(6), 492–503.
- Aha, D. W., Molineaux, M., & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. In Muñoz-Ávila, H., & Ricci, F. (Eds.), *Proceedings of the International Conference on Case-Based Reasoning (ICCBR)*, pp. 5–20.
- Arkin, R. C. (1989). Navigational path planning for a vision-based mobile robot. *Robotica*, 7(1), 49–63.
- Beckmann, N., Kriegel, H., Schneider, R., & Seeger, B. (1990). The r^* -tree: An efficient and robust access method for points and rectangles. In Garcia-Molina, H., & Jagadish, H. V. (Eds.), *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990.*, pp. 322–331. ACM Press.
- Cheung, K. L., & Fu, A. W. (1998). Enhanced nearest neighbour search on the r -tree. *SIGMOD Record*, 27(3), 16–21.
- Chew, L. P. (1989). Constrained delaunay triangulations. *Algorithmica*, 4(1), 97–108.
- Cui, M., Harabor, D. D., & Grastien, A. (2017). Compromise-free pathfinding on a navigation mesh. In Sierra, C. (Ed.), *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pp. 496–502. ijcai.org.
- de Berg, M. (2000). *Computational geometry: algorithms and applications, 2nd Edition*. Springer.
- Demyen, D., & Buro, M. (2006). Efficient triangulation-based pathfinding. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pp. 942–947. AAAI Press.
- Gao, Y., Liu, Q., Miao, X., & Yang, J. (2016). Reverse k -nearest neighbor search in the presence of obstacles. *Inf. Sci.*, 330, 274–292.
- Gao, Y., Yang, J., Chen, G., Zheng, B., & Chen, C. (2011). On efficient obstructed reverse nearest neighbor query processing. In Cruz, I. F., Agrawal, D., Jensen, C. S., Ofek, E., & Tanin, E. (Eds.), *19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2011, November 1-4, 2011, Chicago, IL, USA, Proceedings*, pp. 191–200. ACM.
- Gao, Y., & Zheng, B. (2009). Continuous obstructed nearest neighbor queries in spatial databases. In Çetintemel, U., Zdonik, S. B., Kossmann, D., & Tatbul, N. (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pp. 577–590. ACM.
- Ghosh, S. K., & Mount, D. M. (1991). An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.*, 20(5), 888–910.

- Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In Yormark, B. (Ed.), *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pp. 47–57. ACM Press.
- Harabor, D. D., Grastien, A., Öz, D., & Aksakalli, V. (2016). Optimal any-angle pathfinding in practice. *J. Artif. Intell. Res.*, 56, 89–118.
- Hjaltason, G. R., & Samet, H. (1999). Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2), 265–318.
- Kallmann, M. (2005). Path planning in triangulations. In *Proceedings of the IJCAI workshop on reasoning, representation, and learning in computer games*, pp. 49–54.
- Kamel, I., & Faloutsos, C. (1994). Hilbert r-tree: An improved r-tree using fractals. In Bocca, J. B., Jarke, M., & Zaniolo, C. (Eds.), *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pp. 500–509. Morgan Kaufmann.
- Ooi, B. C. (1987). Spatial kd-tree: A data structure for geographic database. In Schek, H., & Schlageter, G. (Eds.), *Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Darmstadt, 1.-3. April 1987, Proceedings*, Vol. 136 of *Informatik-Fachberichte*, pp. 247–258. Springer.
- OpenStreetMap contributors (2017). Planet dump retrieved from <https://planet.osm.org> . <https://www.openstreetmap.org>.
- Roussopoulos, N., Kelley, S., & Vincent, F. (1995). Nearest neighbor queries. In Carey, M. J., & Schneider, D. A. (Eds.), *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995.*, pp. 71–79. ACM Press.
- Sellis, T. K., Roussopoulos, N., & Faloutsos, C. (1987). The r+-tree: A dynamic index for multi-dimensional objects. In Stocker, P. M., Kent, W., & Hammersley, P. (Eds.), *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pp. 507–518. Morgan Kaufmann.
- Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(2), 144–148.
- Tung, A. K. H., Hou, J., & Han, J. (2001). Spatial clustering in the presence of obstacles. In *Proceedings 17th International Conference on Data Engineering*, pp. 359–367.
- Xia, C., Hsu, D., & Tung, A. K. H. (2004). A fast filter for obstructed nearest neighbor queries. In Williams, M. H., & MacKinnon, L. M. (Eds.), *Key Technologies for Data Management, 21st British National Conference on Databases, BNCOD 21, Edinburgh, UK, July 7-9, 2004, Proceedings*, Vol. 3112 of *Lecture Notes in Computer Science*, pp. 203–215. Springer.
- Zhang, J., Papadias, D., Mouratidis, K., & Zhu, M. (2004). Spatial queries in the presence of obstacles. In Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., & Ferrari, E. (Eds.), *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, Vol. 2992 of *Lecture Notes in Computer Science*, pp. 366–384. Springer.

- Zhao, S., Taniar, D., & Harabor, D. D. (2018). Fast k -nearest neighbor on a navigation mesh. In Bulitko, V., & Storandt, S. (Eds.), *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*, pp. 124–132. AAAI Press.