Efficient Object Search in Game Maps

Jinchun Du, **Bojie Shen**, **Shizhe Zhao**, **Muhammad Aamir Cheema**, **Adel Nadjaran Toosi**
Faculty of Information Technology, Monash University, Melbourne, Australia
{jinchun.du, bojie.shen, shizhe.zhao, aamir.cheema, adel.n.toosi}@monash.edu

Abstract

Video games feature a dynamic environment where locations of objects (e.g., characters, equipment, weapons, vehicles etc.) frequently change within the game world. Although searching for relevant nearby objects in such a dynamic setting is a fundamental operation, this problem has received little research attention. In this paper, we propose a simple lightweight index, called Grid Tree, to store objects and their associated textual data. Our index can be efficiently updated with the underlying updates such as object movements, and supports a variety of object search queries, including k nearest neighbors (returning the k closest objects), keyword k nearest neighbors (returning the k closest objects that satisfy query keywords), and several other variants. Our extensive experimental study, conducted on standard game maps benchmarks and real-world keywords, demonstrates that our approach has up to 2 orders of magnitude faster update times for moving objects compared to state-of-the-art approaches such as navigation mesh and IR-tree. At the same time, query performance of our approach is similar to or better than that of IR-tree and up to two orders of magnitude faster than the other competitor.

1 Introduction

Video games offer a virtual environment in which players interact with a variety of objects such as game characters, units, vehicles, equipment, weapons and other types of items. These objects can be moving, changing, appearing or disappearing, creating a dynamic and ever-evolving game world. This dynamic nature of games poses a unique challenge for efficient object search – searching for relevant nearby objects – in the game world. Object search is a crucial operation in video games, enabling players to navigate the game world, interact with objects, and complete tasks. It is also used by game engines in various contexts, including game AI, physics simulation, scripting, inventory management, quest tracking, and object tracking. For instance, the game AI employs object search to locate nearby enemies, allies, weapons, and other objects relevant to their locations and actions.

While finding shortest path/distance between two points in a game map, which is represented as a Euclidean plane containing polygonal obstacles, has been very well studied [Yap *et al.*, 2011; Shen *et al.*, 2020; Nash *et al.*, 2007], object search has received little research attention despite its practical significance. There exists some works [Zhao *et al.*, 2018b] on finding k closest objects in game maps, called k nearest neighbors (kNN), but most of the existing techniques are not designed for the dynamic game environments. Searching for relevant nearby objects in dynamic game environments is challenging as it requires efficiently handling real-time object updates while maintaining fast query performance. Additionally, in many practical applications, it is important to find nearby objects that match a specific textual description, e.g., finding the nearest "healing unit". In such scenarios, simply identifying the closest objects without considering whether they match the required textual description is insufficient. While our focus in this paper is on game maps, there are many applications of the problem we study in this paper beyond game maps such as in indoor location-based services [Cheema, 2018], home assistant technologies [Luria *et al.*, 2016; Umair *et al.*, 2021], automated warehouses [Custodio and Machado, 2020], asset tracking [Krishnan and Mendoza Santos, 2021] etc.

To the best of our knowledge, we are the first to study such textual object search in dynamic game environments. Specifically, we study keyword kNN queries that find the k closest objects that satisfy the query keywords. The state-of-the-art algorithms for traditional kNN queries are: Incremental Euclidean Restriction(IER)-Polyanya [Zhao *et al.*, 2018a]; and Interval Heuristic (IH) [Zhao *et al.*, 2018b]. Although both IER-Polyanya and IH can be extended to answer keyword kNN queries (see Section 3.2), they either suffer from poor query performance or inefficient update handling. Specifically, IER-Polyanya utilises R-tree for efficient object search but suffers from poor update handling because R-tree is not well-suited for dynamic environments. In contrast, IH employs navigation mesh which can be efficiently updated but suffers from poor query performance especially when the result objects are not close to the query.

Given the limitations of IER-Polyanya and IH, there is a need to design an effective index that can be efficiently updated in highly dynamic environments such as game maps and, at the same time, allows efficient query processing. To this end, we present a simple lightweight index, called Grid Tree, which cannot only efficiently handle object updates but also allows efficiently processing keyword kNN queries and several variants. We evaluate our approach using widely used game benchmarks [Sturtevant, 2012] and realistic keyword datasets for these games. We compare our approach with IER-Polyanya, IH, and IER-EHL (a faster version of IER-Polyanya), and show that our approach achieves the best of both worlds. Specifically, it can handle object updates by up to 2 orders of magnitude faster than IER-Polyanya and IER-EHL, and its update cost is comparable to IH (specifically, faster for object movements and slower for object insertions/deletions). At the same time, its query performance is comparable to IER-EHL, several times faster than IER-Polyanya, and up to two orders of magnitude faster than IH. We also discuss how our approach can efficiently answer several other variants of textual object search queries.

2 Preliminaries

We consider a Euclidean plane containing a set of obstacles, each represented as a polygon. Two points in the plane are **visible** to each other (i.e., **co-visible**) iff there exists a straight line connecting them that does not pass through any obstacle. A **path** $\mathcal{P}$ between two points x and y is an ordered set of points $\langle p_1, p_2, \cdots, p_n\rangle$ where $p_1 = x$, $p_n = y$ and every successive pair of points p_i and p_{i+1} $(i < n)$ is co-visible. The **length** of a path $\mathcal{P}$ is the cumulative Euclidean distance between the successive pairs of points, denoted as $|\mathcal{P}|$, i.e., $|\mathcal{P}| = \sum_{i=1}^{n-1} Edist(p_i, p_{i+1})$ where $Edist(p_i, p_{i+1})$ is the Euclidean distance between p_i and p_{i+1}. A path $\mathcal{P}$ is a **shortest path**, denoted as $sp(x, y)$, if there is no other path between x and y shorter than $\mathcal{P}$. We use $d(x, y)$ to denote the length of the shortest path, i.e., $d(x, y) = |sp(x, y)|$.

We consider a set of objects O in the traversable (i.e., non-obstacle) area of the Euclidean plane. Each object $o_i \in O$ is represented as a tuple $(o_i.\rho, o_i.\tau)$ where $o_i.\rho$ is a two-dimensional point representing location of o_i in the Euclidean plane and $o_i.\tau$ is its textual description represented as a set of keywords. Similar to many existing works in dynamic environments [Mouratidis *et al.*, 2005; Hidayat *et al.*, 2022], we consider a timestamp model where the time domain is discretised into a set of timestamps T. The set of objects O may change between two consecutive timestamps if new objects are added to O or some existing objects are deleted. We use O^t to denote the set of objects at a timestamp $t \in T$. Similarly, location and/or textual description of an object o_i may change and we use $o_i^t = (o_i^t.\rho, o_i^t.\tau)$ to represent an object $o_i^t \in O^t$ at a timestamp $t \in T$.

A query q is also a tuple $(q.\rho, q.\tau)$ representing its location and query keywords. There are many variants of textual object search but, in this work, our main focus is on boolean kNN query [Chen *et al.*, 2013] which is one of the most popular keyword queries.

Definition 1. *boolean kNN Query: Given a query $q = (q.\rho, q.\tau)$ issued at timestamp t and the set of objects O^t, find up to k objects closest from the query location $q.\rho$ among the objects that contain all query keywords $q.\tau$. Formally, the result set of the query R contains up to k objects from O^t such that $\forall o_i^t \in R$: $q.\tau \subseteq o_i^t.\tau$ and $\nexists o_j^t \in O^t \setminus R$: $d(q.\rho, o_j^t.\rho) < d(q.\rho, o_i^t.\rho) \wedge q.\tau \subseteq o_j^t.\tau$.*

Example 1. *Figure 1 shows a game map where black polygons represent obstacles (i.e., non-traversable area). The maps contains six objects o_1 to o_6 along with their associated textual description, e.g., $o_1.\tau = \{w, x\}$. Consider a boolean 1NN query q shown on the map with $q.\tau = \{x, y\}$. The objects o_2, o_3 and o_6 are the candidate objects (shown in green filled circles) as each of these contains both of the query keywords x and y. However, o_2 is the closest object among these from q considering the obstacle-avoiding distance. Thus, the result for query q is o_2.*

In Section 4.4, we discuss how our approach can be used to answer several variants of this query. Also, note that a traditional kNN query is a special case of boolean keyword kNN query when there is no query keyword, i.e., $q.\tau = \emptyset$.

3 Related Work

3.1 Pathfinding in Game Maps

Pathfinding in game maps, finding shortest path between two locations, has been extensively studied, e.g., see [Demyen and Buro, 2006; Oh and Leong, 2017; Uras and Koenig, 2015; Shen *et al.*, 2022] and references therein. Next, we briefly discuss two state-of-the-art algorithms most relevant to this work.

Polyanya [Cui *et al.*, 2017] is an efficient online pathfinding algorithm. The algorithm employs a navigation mesh [Kallmann and Kapadia, 2014] which divides the traversable area into a set of convex polygons. Polyanya instantiates a search similar to A* algorithm and treats polygon edges of the navigation mesh as search nodes. It iteratively expands the edges according to heuristic values considering their distances from source and target. When the search accesses the polygon containing target, the target is also added in the queue as a search node. The algorithm terminates when the target is expanded.

Euclidean Hub Labeling (EHL) [Du *et al.*, 2023] is the state-of-the-art pathfinding algorithm. It employs hub labeling [Abraham *et al.*, 2011] which is a highly efficient approach to compute shortest paths/distances in graphs. In the preprocessing phase, EHL computes hub labels on the visibility graph containing the convex vertices of the map. A uniform grid is superimposed on the map and, for each cell c of the grid, hub labels of the vertices visible from c are copied to the cell c. During query processing, the hub labels of the cells containing source and target are combined to find the common hub nodes and compute the shortest path/distance.

3.2 Object Search in Game Maps

Object search on geo-textual data has been very well-studied [De Felipe *et al.*, 2008; Cong *et al.*, 2009; Chen *et al.*, 2013; Chen *et al.*, 2020; Xu *et al.*, 2022] due to its applications in map-based services. Unfortunately, these techniques are not suitable for game maps which are highly dynamic and are represented differently, as a Euclidean plane containing polygonal obstacles. Next, we briefly discuss two best-known algorithms for computing traditional kNN queries in game maps and their extension for textual object search.

Interval Heuristic (IH) [Zhao *et al.*, 2018b] is based on Polyanya and replaces the heuristic of the A* search such that the search incrementally explores the space like Dijkstra search. When the search reaches a polygon that contains an object, the object is also added to the queue. The algorithm terminates when k objects are expanded. IH can be easily extended to answer keyword kNN queries by pruning every accessed object that does not satisfy query keywords. Since IH employs Polyanya which exploits a navigation mesh, handling object updates is quite efficient. Specifically, IH requires maintaining the objects located in each polygon of the navigation mesh. Thus, if an object changes its location, the object is deleted from its previous polygon and added to its new polygon. If the object remains in the same polygon, the navigation mesh does not need any update.

IER-Polyanya [Zhao *et al.*, 2018a] employs an R*-tree [Beckmann *et al.*, 1990] and incrementally retrieves nearest objects to the query location according to their Euclidean distances. For each retrieved object, it calls Polyanya to compute its actual distance from the query. The algorithm terminates when the Euclidean distance of the next retrieved object is no smaller than the actual distances of kNNs. To handle keyword queries, we use IR-tree [Li *et al.*, 2010], a popular extension of R*-tree to handle spatio-textual data. While R*-tree and IR-tree allow efficient query processing, it is computationally expensive to update them. For example, the location update is handled by first updating the structure of IR-tree in a way similar to how R-tree handles updates. Then, the textual information associated with each node is updated accordingly. Since the nodes of R*-tree and IR-tree may need to be expanded or shrunk with the updates, they are not well-suited for highly dynamic environments such as game maps.

The other two approaches in [Zhao *et al.*, 2018a], Target Heuristic (TH) and Fence Heuristic (FH), are not suitable for highly dynamic environment and were outperformed by both IER-Polyanya and IH in our initial experiments and, therefore, are not discussed/compared against in this paper.

4 Our Approach

First, we present the details of our index, called Grid Tree, in Section 4.1. Then, in Section 4.2, we discuss how the Grid Tree is updated with the changes in the underlying data. Section 4.3 presents our query processing algorithm. Finally, Section 4.4 discusses how the proposed approach can be easily extended to answer a variety of other queries. Our algorithm relies on a shortest distance computation module which is responsible for computing $d(x, y)$ between any two points x and y. Although any shortest distance computation algorithm can be used for this purpose, we employ Euclidean Hub Labeling (EHL) [Du *et al.*, 2023] because it is the most efficient shortest distance computation algorithm. We intentionally keep our index separate from the shortest distance computation module because it offers flexibility in system design.

4.1 Grid Tree

Motivation. Traditional indexes such as R-tree [Guttman, 1984], R*-tree [Beckmann *et al.*, 1990], kd-tree [Ooi, 1987], and Quad-tree [Smith and Chang, 1994] as well as their ex-

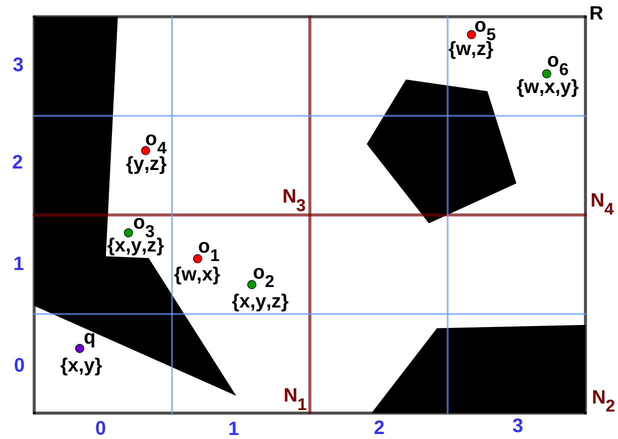


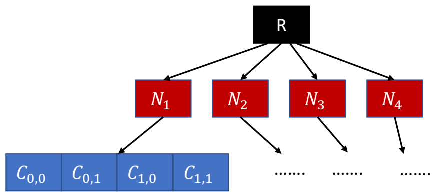
Figure 1: Boolean keyword kNN query: o_2 is the 1NN.

tensions [Chen *et al.*, 2013] to index textual information, allow efficient query processing for a variety of queries. However, a major limitation of these indexes is that they are not suitable for dynamic environments such as game maps where object updates are frequent. Therefore, we need an index that can be efficiently updated in the dynamic environment and allows efficient query processing. Next, we present the details of a simple, easy-to-implement and effective index, called Grid Tree, that can be efficiently updated and allows efficient query processing.

Strucutre of Grid Tree. Root node of the Grid Tree is a minimum bounding rectangle (MBR) of the whole map. Each node is recursively divided into four equal sized children until the size of each child node is smaller than a threshold (to be discussed in experiments). Consider a Grid Tree of height h where the root node is at level 0 and the leaf nodes are at level h. There are $2^i \times 2^i$ equal-sized nodes at level i of the Grid Tree. The leaf nodes constitute a uniform grid containing $2^h \times 2^h$ equal-sized cells. Hereafter, we use the terms leaf nodes and cells interchangeably to refer to the level h nodes.

For each leaf node n, we store an object list containing the IDs of the objects that are located inside n. Additionally, for every node n in the Grid Tree, we store a keyword list. Hereafter, when we say "objects *inside* a node n", we refer to all the objects that are in the subtree rooted at the node n. The keyword list of the node n contains all unique keywords of the objects inside n along with the frequency of each keyword, e.g., if a keyword κ appears in 5 objects inside n, its frequency is 5. We implement the keyword list as a hash map so that frequency of any keyword can be obtained/updated efficiently. Note that object lists are stored only for leaf nodes whereas keyword lists are stored for all nodes of the tree.

Example 2. *Figures 1 and 2 show a Grid Tree of height* 2*. The root node* R *of the Grid Tree is the MBR covering the whole space. The root node has four equal-sized children* N_1 *to* N_4 *shown in solid red lines. Each of these nodes is further subdivided into four children. The leaf nodes at level* 2 *represent a* 4×4 *grid (see cells shown in blue lines). In Figure 1, we refer to each leaf node as* $C_{i,j}$ *where* i *and* j *correspond to its position along x-axis and y-axis, respectively (see the blue*



Object Lists

Cell	Objects
$C_{1,1}$	o_1, o_2
$C_{0,1}$	o_3
$C_{0,2}$	o_4
$C_{3,3}$	o_5, o_6

Keyword Lists of some nodes

Node	Keywords
R	w: 3, x: 4, y: 4, z: 4
N_1	w: 1, x: 3, y: 2, z: 2
$C_{1,1}$	w: 1, x: 2, y: 1, z: 1
$C_{0,1}$	x: 1, y: 1, z: 1

Figure 2: Grid Tree: Children of N_2 to N_4 are not shown. Object Lists and Keyword Lists of **some** of the nodes are shown.

*numbers outside the map), e.g., q is located in the cell/leaf node $C_{0,0}$ and o_4 is located in the leaf node $C_{0,2}$. Figure 2 shows the structure of Grid Tree as well as the object lists and keyword lists for some of the nodes. Since the leaf node $C_{1,1}$ contains o_1 and o_2, its object list consists of o_1 and o_2. Keyword list of $C_{1,1}$ contains all keywords present in o_1 and o_2 along with their frequencies, e.g., each of w, y and z appears in only one object whereas x appears in both o_1 and o_2. The keyword list of N_1 represents the keywords and their frequencies for all objects in N_1 (i.e., o_1, o_2 and o_3). The keyword list of the root represents keywords and frequencies of all six objects. For simplicity, Figure 2 shows object lists and keyword lists only for **some** cells and nodes of the tree.*

4.2 Updating Grid Tree

Now, we explain how the Grid Tree is updated at a timestamp $t \in T$. Although our focus in this paper is on handling moving objects, for completeness, we discuss how to insert an object, delete an object, and handle the change in the location/text of an object.

Inserting a new object. To insert a new object o_i^t, we first identify the leaf node n that contains the location $o_i^t.\rho$. Then, o_i^t is added to the object list of n. Keyword list of n is also updated by incrementing the frequency of each keyword $\kappa \in o_i^t.\tau$ by one. If a keyword does not exist in the keyword list, it is added with frequency one. Then, all the ancestor nodes of n are iteratively accessed and their keyword lists are updated in the same way.

Deleting an object. To delete an object o_i^t, it is deleted from the object list of the node n containing it. The keyword list of n is also updated by decrementing the frequency of each keyword $\kappa \in o_i^t.\tau$ by one. If the frequency of any keyword is reduced to zero, it is deleted from the keyword list. The keyword lists of all ancestor nodes of n are also updated in the same way.

Handling the location change of an object. Assume that the location of an object changes between two timestamps, e.g., $o_i^{t-1}.\rho \neq o_i^t.\rho$. We update the Grid Tree at timestamp t as follows. We identify the leaf nodes n and n' that contain the locations $o_i^{t-1}.\rho$ and $o_i^t.\rho$, respectively. If n and n' are the same leaf node, we do not need to update anything. Otherwise, we delete the object from the object list of n and add it to the object list of n'. Keyword lists of n and n' are also updated accordingly, i.e., by decrementing the frequency of each keyword $\kappa \in o_i^t$ in the keyword list of n and incrementing it by one in the keyword list of n'. Then, the parent nodes of n and n' are iteratively accessed and their keyword lists are updated accordingly until a common ancestor of n and n' is reached. Note that the keyword list of the common ancestor does not need to be updated.

Example 3. *Consider the example shown in Figures 1 and 2 and assume that the object o_3 moves from its current cell $C_{0,1}$ to the cell $C_{1,1}$. We delete o_3 from the object list of $C_{0,1}$ and add it to the object list of $C_{1,1}$. The keyword list of $C_{0,1}$ is updated by decrementing the frequency of each of the keywords x, y and z by one and, consequently, the keyword list of $C_{0,1}$ becomes empty. Then, the keyword list of $C_{1,1}$ is updated by incrementing the frequencies of x, y and z by one each. As a result, the keyword list of $C_{1,1}$ is updated to $\{w : 1, x : 3, y : 2, z : 2\}$. Next, we access the parent nodes of the two cells $C_{0,1}$ and $C_{1,1}$. Since both have the same parent N_1, the keyword list of N_1 does not need to be updated.*

Handling textual change of an object. Assume that textual description of an object changes between two timestamps. For each deleted keyword κ (i.e., $\kappa \in o_i^{t-1}.\tau \wedge \kappa \notin o_i^t.\tau$), we update the keyword list of the leaf node n containing $o_i^t.\rho$ by decrementing the frequency of κ by one. For each newly added keyword κ' (i.e., $\kappa' \notin o_i^{t-1}.\tau \wedge \kappa' \in o_i^t.\tau$), we update the keyword list of n by incrementing the frequency of κ' by one. Keyword lists of all ancestors of n are also updated.

If both the location and the textual description of an object change between two timestamps, we delete the object o_i^{t-1} and insert o_i^t as discussed earlier.

Complexity Analysis. Here, we provide complexity analysis for handling the updates mentioned above. A key operation for handling the updates is to identify the leaf node of the Grid Tree that contains a particular location. This can be done in $O(1)$ because the leaf nodes correspond to a grid of $2^h \times 2^h$ equal-sized cells where h is the height of the tree. The object insertion and deletion in the object list of a cell can also be done in $O(1)$. Specifically, the object list of each cell is implemented as a linked list. Furthermore, we maintain a global object array containing all objects indexed by their IDs. For each object o_i^t, this array stores a pointer to the place of o_i^t in the object list of the cell containing it. This allows deleting an object from the object list in $O(1)$. A new object is always inserted at the end of the object list and its place in this object list is reflected in the global object array. Keyword lists are implemented as hash tables. Although the worst-case complexity is linear to the number of keywords, on average, the cost is $O(1)$. Consider an update involving $\mathcal{K}$ keywords, the average cost of updating the keyword list is $O(\mathcal{K})$. For all of the updates mentioned above, we need to update at most $O(h)$ nodes. Therefore, the total cost for each update operation is $O(\mathcal{K}h)$ on average.

We remark that while the cost of handling location change of an object is $O(\mathcal{K}h)$ in general, the cost when the object moves within the same leaf node is $O(1)$. This is because, in

Algorithm 1: Boolean kNN query processing

Input: $q.\rho, q.\tau, k$: query location, query keywords and k
Output: R: query results
1 $R = \phi$; $d^k = \infty$;
2 Initialise a min-heap H with the root node of Grid Tree ;
3 **while** $H \neq \phi$ **do**
4   deheap an entry e from H ;
5   **if** $e.key \geq d^k$ **then**
6   **return** R;
7   **if** e *is an object* **then**
8   compute $d(q.\rho, e.\rho)$;
9   **if** $d(q.\rho, e.\rho) < d^k$ **then**
10   update R and d^k by object e;
11   **else if** e *is a leaf node* **then**
12   **for** *each object* o_i^t *in the object list of* e **do**
13   **if** $q.\tau \subseteq o_i^t.\tau$ **then**
14   insert o_i^t in H with key $mindist(q.\rho, o_i^t.\rho)$;
15   **else**
16   **for** *each child node* c *of* e **do**
17   **if** c *contains all query keywords* $q.\tau$ **then**
18   insert c in H with key $mindist(q.\rho, c)$;
19 **return** R;

this case, we do not need to update the object list and keyword list of any node. This enables our proposed index to handle moving objects very efficiently. Traditional indexes such as R-tree, Quad-tree and kd-tree cannot handle moving objects in $O(1)$.

4.3 Query Processing

Algorithm 1 shows the details of our algorithm to compute boolean kNNs of a query using the Grid Tree. The algorithm initialises the result set R to be empty and d^k to infinity (line 1) where d^k is the distance of the k^{th} closest object in R. A min-heap H is initialised by inserting the root node of the Grid Tree (line 2) with key set to zero. The key of an entry e (denoted as $e.key$) inserted in the heap is a lower bound distance from $q.\rho$ to the entry e (e.g., minimum Euclidean distance from $q.\rho$ to the node e). In each iteration, the algorithm de-heaps an entry e from the heap. If $e.key$ is at least equal to d^k, the algorithm terminates by returning the result set R (line 6). This is because all remaining entries have distances from the query at least equal to d^k and, therefore, cannot contain an object closer to the query than the k^{th} closest object.

If the de-heaped entry e is an object, its distance from the query $d(q.\rho, e.\rho)$ is computed (line 8). This distance can be computed using any of the existing pathfinding algorithms. In our implementation, we use Euclidean Hub Labeling (EHL) [Du *et al.*, 2023] which is the state-of-the-art shortest path computation algorithm in game maps. If this distance is smaller than d^k, the result set R and d^k are updated accordingly (line 10). Specifically, we implement R as a max-heap with keys set to distances between the query and the objects stored in R. We insert e in R and ensure that R contains at most k objects after each iteration. If after inserting e, R contains more than k objects, the object with the largest distance (i.e., the top entry in the max-heap) is deleted from R. If R contains less than k objects, d^k is kept to be infinity. Otherwise, d^k is set to the distance of the k^{th} closest object in R (i.e., the key of the top entry in the max-heap).

If the de-heaped entry e is a leaf node of the Grid Tree, we process the objects in its object list (line 12) and insert each object o_i^t that contains all query keywords in the min-heap H (lines 13 and 14). The key of each object inserted in the min-heap is a lower bound distance between the query and the object locations. In our implementation, we use Euclidean distance between $q.\rho$ and $o_i^t.\rho$ as the lower bound distance.

Finally, if e is a non-leaf node of the Grid Tree, we process each child node c of e as follows. First, we check if c contains all query keywords or not (line 17). Specifically, a node c contains all query keywords $q.\tau$ iff, for every keyword $\kappa \in q.\tau$, κ exists in the keyword list of c. If c contains all query keywords, it is inserted in the min-heap with key set to a lower bound distance (e.g., minimum Euclidean distance) between the query location and the node c. If the heap H becomes empty, the algorithm returns R (line 19) which contains up to k closest objects found by the algorithm.

Remarks. Although our implementation uses minimum Euclidean distance as the lower bound at lines 14 and 18, other lower bounds can also be used. One feature of the Grid Tree is that its nodes do not spatially change regardless of the updates (unlike other popular spatial indexes such as R-tree, kd-tree etc.). Therefore, it is possible to precompute and store lower bound distances. E.g., one may precompute minimum distances from the convex vertices in the map to all nodes of the Grid Tree. During query processing, the closest visible vertex from the query can be used to obtain a lower bound distance for any node of the Grid Tree by using triangular inequality.

4.4 Extensions

Generalisation of boolean kNN query. Boolean kNN queries can be generalised to find k closest objects that contain at least n keywords in $q.\tau$, e.g., for each result object $o_i^t \in R$, $|q.\tau \cap o_i^t.\tau| \geq n$ where $|X|$ denotes the number of elements in a set. This generalised version can be easily handled by changing the conditions at lines 13 and 17 accordingly.

Top-k spatial keyword query. In top-k spatial keyword query [Chen *et al.*, 2013], each object is assigned a *score* computed using a scoring function that considers both its textual similarity to the query keywords and distance from query location. The query requires finding k objects with the smallest scores (assuming lower scores are better). Our algorithm can answer such queries as follows. The min-heap H is modified such that the keys are minimum scores of the entries instead of minimum distances. The minimum score of an entry e (an object or a node) is computed using the minimum Euclidean distance between e and query location and the best possible textual similarity of e to query keywords considering the keyword list of e. The algorithm employs s^k, score of the k^{th} object in R, instead of d^k. The conditions at lines 13 and 17 of the algorithm are modified such that an entry is inserted in H only if its minimum score is smaller than s^k.

Keyword range query. Given a distance range r, a keyword range query returns every object $o_i^t \in O^t$ that satisfies the query keywords and $d(q.\rho, o_i^t.\rho) < r$. Our algorithm can be

Game	#Maps	# Cells	# Trav. Cells	# Vertices
DA	67	151,420	15,911	1182.9
DAO	156	134,258	21,322	1727.6
BG	75	262,144	73,930	1294.4
SC	75	446,737	263,782	11487.5

Table 1: Total number of maps, and average number of total cells, traversable cells and vertices in each benchmark.

easily modified by replacing d^k with r. This ensures that all objects with distances less than r are included in R.

Constrained keyword kNN queries. In constrained keyword kNN queries, the goal is to find the k closest objects that satisfy the query keywords and lie in a specified region (called constrained region) of the map. E.g., one may want to find the nearest "artillery unit" in a specific zone of the game map. This query can be answered easily using our proposed algorithm by adding a filter to prune every entry e that does not overlap with the constrained region.

5 Experiments

5.1 Settings

We run our experiments on a 3.2 GHz Intel Core i7 machine with 32 GB of RAM. All the algorithms are implemented in C++ and compiled with -O3 flag. We run experiments on widely used game map benchmarks [Sturtevant, 2012] of four popular games: Dragon Age II (DA); Dragon Age Origins (DAO); Baldur's Gate II (BG) and StarCraft (SC). In total, this gives us 373 maps each represented as a grid map. Table 1 shows details of these benchmarks including the average size – represented by total number of cells and the total number of traversable (i.e., non-obstacle) cells in the maps – and average number of obstacle vertices. We generate the objects, their keywords and queries as follows.

Object generation. Initial location of each object is a randomly generated point in the traversable region of the map. We evaluate the effect of object density which is the ratio of number of objects to the number of traversable cells in the map, e.g., object density of 1% indicates that the number of objects is 1% of the total number of traversable cells in the map. We vary the density from 0.1% to 10% and the default density is 1%. Although we also study the effect of insertions/deletions, our main focus is on moving objects. We define *mobility* of an object set as the percentage of objects that move between two timestamps. We vary the mobility from 10% to 100% and the default mobility is 70%. We generate the moving objects as follow. For each moving object, we randomly choose a target location in the trarversable region of the map and compute the shortest path from the initial location of the object to the target location. The object then starts moving towards the target and travels 1 unit distance (i.e., which is equal to the width/height of one cell in the map) in each timestamp. When an object reaches the target, a new randomly generated target is chosen and the object continues to travel on the shortest path towards this new target.

Keyword generation. For each game, we use ChatGPT (Jan 9 version) to obtain 100 items in the game along with their descriptions. Specifically, we use prompts like "describe characters in [game map]" to get a list of items including characters, units, weapons, gems, potions etc. We keep prompting ChatGPT until it generates 100 items and their descriptions[1]. We use `nltk`, an NLP library, to remove stop words and normalise the remaining words (e.g., "abilities" and "ability" both are normalised to "ability"). After this pre-processing, maximum, minimum, and average number of keywords per item in each game are as follows: DA (19,7,15); DAO (17,7,12); BG (17,6,12); SC (18,7,11). For each object, we randomly assign it to an item type in the relevant game. Let m be the number of keywords in that item, we randomly choose a number r between 1 and m and randomly assign r keywords of this item to the object.

Query generation. For each experiment, we generate 100 queries per timestamp. Location of each query is randomly generated in the traversable region of the map. We evaluate the effect of k which is varied from 1 to 10 where the default value of k is 3. We also evaluate the effect of number of query keywords by varying the number of query keywords from 0 to 3 where the default number of keywords is 2. Following the existing works on geo-textual object search [Chen *et al.*, 2013], we generate a query containing x keywords by randomly choosing an object from the map and selecting x words at random from the object as the query keywords. This ensures that the combination of query keywords is meaningful and at least one object satisfies the query keywords.

Algorithms evaluated. Our approach, Grid Tree, is shown as GT in the experiments. We evaluate different sizes of Grid Tree each shown as GT(m) where GT(m) is the Grid Tree with each leaf node of size at most $m \times m$ units. E.g., in GT(4), we stop recursively dividing nodes into children when the node size becomes less than 4×4.

We compare our approach with two state-of-the-art approaches presented in [Zhao *et al.*, 2018a]: IER-Polyanya (shown as **IER-Pol**) and Interval Heuristic (**IH**). We use the source code provided by the authors. We also compare against **IER-EHL** which is the same as IER-Pol except that the shortest distances are computed using EHL [Du *et al.*, 2023] instead of Polyanya [Cui *et al.*, 2017]. This gives a like-for-like comparison with our approach as we also employ EHL for shortest distance computation.

5.2 Results

Each experiment is run for 50 timestamps and, for each timestamp t, we first update the underlying index considering all object updates at t and then process 100 queries on the updated index. We report average update time per timestamp for all 50 timestamps as well as the average query processing time for all 5000 queries.

Effect of object density: Figure 3 shows the effect of object density on query time (top row) and update cost (bottom row) on each of the four benchmarks. Overall, the fastest algorithms in terms of query processing are GT(16) and GT(64) whereas the best performing algorithms in terms of handling updates is GT(64). We discuss the details of query time and update time below.

[1] https://github.com/goldi1027/GT-EHL

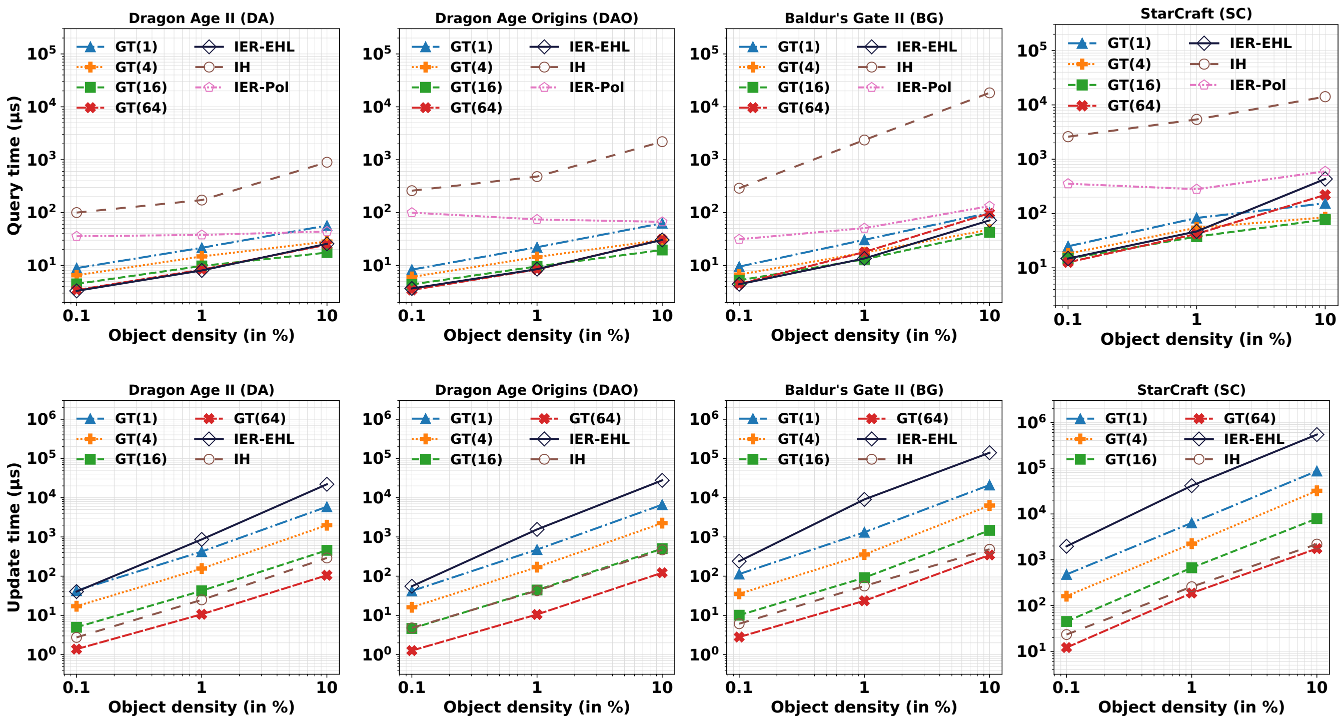


Figure 3: Effect of object density on query time (top row) and update time (bottom row) for each approach on default settings ($k = 3$, mobility = 70%, # of query keywords = 2).

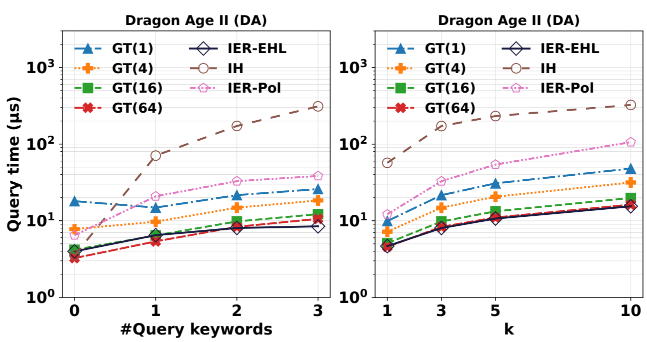


Figure 4: Effect of # of query keywords and k on query time

Query time. For lower object density, query performance of our approach improves when the leaf nodes are bigger (e.g., GT(64)) because the tree height is smaller and the search needs to traverse fewer nodes. However, as the density increases, the performance of GT(64) degrades because each leaf node contains more object requiring the algorithm to process a larger number of objects. IER-EHL outperforms the other competitors IH and IER-Poly, however, its performance is comparable to GT(16) and GT(64) for lower object density but worse for higher object density, e.g., for the SC benchmark, the query time of IER-EHL is several times higher than that of GT(16). IH is the slowest algorithm (often more than 2 orders of magnitude slower than our algorithms) because it needs to incrementally explore a large search space before it can find the answers. IER-Poly is slower than IER-EHL mainly because Polyanya is slower than EHL. However, the performance of IER-Poly does not necessarily degrade with the increase in object density. This is because the cost of shortest distance computation for Polyanya decreases when the objects are closer to the query and, for higher density, the result objects are found closer to the query.

Update time. The update handling time of Grid Tree significantly improves as the size of leaf nodes increases, e.g., see GT(64). This is because the height of the tree is smaller for GT(64) which means fewer nodes are needed to be updated. Also, the moving objects leave the leaf nodes less often because the leaf nodes are bigger as compared to the leaf nodes in GT(1). The update handling time of GT(64) is up to 2 orders of magnitude lower than the IER-EHL because IR-tree is unable to efficiently handle moving objects. Note that we do not show IER-Poly because it also employs IR-tree and, therefore, its update cost is the same as IER-EHL. IH has a significantly smaller update handling time than IER-EHL because it basically needs to maintain the object information in relevant polygons of the navigation mesh. However, its update handling time is higher than that of GT(64) but better or comparable to that of GT(16).

Effect of query keywords and k. Figure 4 shows the effect of number of query keywords and k on the query performance (the update time is not affected by them). We show the results for the DA benchmark and the results for the other benchmarks follow similar trends. The query cost of all approaches increases with the increase in number of query keywords. The cost of IH is most significantly affected which is mainly because, as the number of query keywords increases,

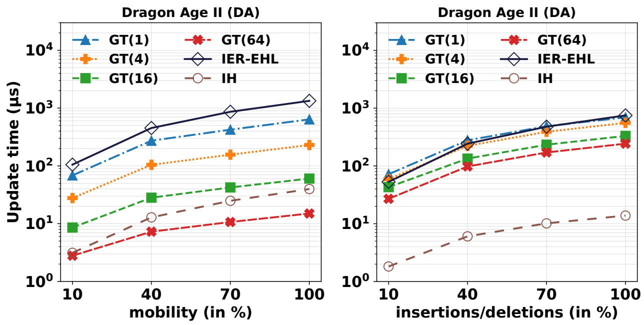
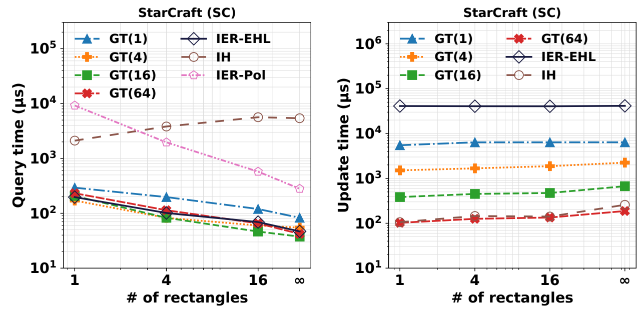


Figure 5: Update time for varying mobility and insertions/deletions.



(a) Query time (b) Update time

Figure 6: Effect of # of rectangles

there are fewer objects that contain all query keywords. As a result, IH needs to incrementally explore larger search space to find the answer. As expected, the cost of all approaches increases with the value of k because the search space increases. GT(64) and IER-EHL are the best performing algorithms in terms of query cost.

Effect of mobility and updates. Figure 5(left) shows the effect of number of moving objects at each timestamp (shown as mobility). GT(64) handles the updates most efficiently and scales better mainly because the moving objects are less likely to leave the leaf nodes as the leaf nodes get bigger and, therefore, requiring fewer updates. Although less common in game maps than moving objects, the objects may be inserted and deleted in game maps. Figure 5(right) studies the effect of insertions/deletions in the game maps . For each experiment shown as $x\%$ insertions/deletions, at each timestamp t, we first randomly insert $\frac{x}{2}\%$ of the total objects in the map as new objects and then randomly delete the same number of objects. The cost is average update cost per timestamp. The cost of our approach increases mainly because the Grid Tree needs to be traversed for each insertion and deletion (unlike object movements which may not require any update if the object is in the same leaf node). On the other hand, the cost of IH and IER-EHL is lower for insertions/deletions than for object movement. This is because to handle a single object movement, these approaches require almost double the work, i.e., deleting the object followed by reinsertion. Grid Tree can still handle updates much faster than IER-EHL but its update cost is up to 1 order of magnitude higher than IH. However, as shown earlier, IH is up to 2 orders of magnitude slower than Grid Tree thus the higher update cost pays off in terms of querying performance especially when the number of deletions/insertions are small compared to the number of queries.

Effect of object distribution. The previous experiments show the results where the object source and target locations are randomly distributed in the traversable space of game maps. In this experiment, we show the results for cases where object source and target locations are clustered in certain areas of the map. Specifically, for each experiment, we randomly generate x rectangles in the traversable space where each rectangle area is 1% of the total space. Object source and target locations are then generated only within these rectangles. We study the effect of x (i.e., the number of rectangles/clusters) by varying x to 1, 4, 16 and infinity. Here, a smaller x implies that objects are clustered in fewer regions in the map and $x = \infty$ corresponds to the random distribution. Note that the total number of objects remain the same for all experiments (set to default density of 1% as explained in Section 5.1). The query set is exactly the same as the previous experiments.

Figure 6(a) shows the effect of object distribution on query performance on the SC benchmark. Query times of all algorithms except IH increase for smaller x. This is because these algorithms use Grid Tree or IR-tree for indexing the objects and the processing cost increases when the objects are densely populated in certain areas. Since IH incrementally explores the search space, its cost depends on how far the objects are located from the query location. When the objects are clustered in certain areas, for most of the queries in this experiment, they are found closer which results in improved querying cost. Overall, query performance trend is similar to the previous experiments, i.e., GT is similar to or better than IER-EHL and 1 to 2 orders of magnitude faster than IH. Figure 6(b) shows the effect of object distribution on the update time of the underlying indexes. The update cost of Grid Tree and IH decreases slightly for smaller x. This is because when x is small, the object source and target locations are closer to each other resulting in fewer objects moving out of the leaf nodes of the Grid Tree or the polygons of navigation mesh used in IH thus resulting in lower update cost.

6 Conclusions

This paper presents the Grid Tree, a lightweight index for storing moving objects and efficiently retrieving textually relevant nearby objects in dynamic video game environments. Extensive experiments on widely used game map benchmarks using realistic keywords demonstrate that the proposed approach generally outperforms the state-of-the-art algorithms in terms of update time and query performance. Grid Tree is a simple, easy-to-implement and highly efficient index which makes it well-suited for deployment in video games, enabling efficient object search in highly dynamic environments. This work also has applications in domains such as indoor location-based services and automated warehouses.

Acknowledgements. This work is supported by Australian Research Council (ARC) DP230100081 and FT180100140.

References

[Abraham *et al.*, 2011] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, pages 230–241. Springer, 2011.

[Beckmann *et al.*, 1990] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[Cheema, 2018] Muhammad Aamir Cheema. Indoor location-based services: challenges and opportunities. *SIGSPATIAL Special*, 10(2):10–17, 2018.

[Chen *et al.*, 2013] Lisi Chen, Gao Cong, Christian S Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. *Proceedings of the VLDB Endowment*, 6(3):217–228, 2013.

[Chen *et al.*, 2020] Lisi Chen, Shuo Shang, Chengcheng Yang, and Jing Li. Spatial keyword search: a survey. *GeoInformatica*, 24(1):85–106, 2020.

[Cong *et al.*, 2009] Gao Cong, Christian S Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, 2(1):337–348, 2009.

[Cui *et al.*, 2017] Michael Cui, Daniel Damir Harabor, and Alban Grastien. Compromise-free pathfinding on a navigation mesh. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 496–502. ijcai.org, 2017.

[Custodio and Machado, 2020] Larissa Custodio and Ricardo Machado. Flexible automated warehouse: a literature review and an innovative framework. *The International Journal of Advanced Manufacturing Technology*, 106:533–558, 2020.

[De Felipe *et al.*, 2008] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. In *2008 IEEE 24th International Conference on Data Engineering*, pages 656–665. IEEE, 2008.

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

[Du *et al.*, 2023] Jinchun Du, Bojie Shen, and Muhammad Aamir Cheema. Ultrafast euclidean shortest path computation using hub labeling. In *AAAI*, 2023.

[Guttman, 1984] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[Hidayat *et al.*, 2022] Arif Hidayat, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Ying Zhang. Continuous monitoring of moving skyline and top-k queries. *The VLDB Journal*, 31(3):459–482, 2022.

[Kallmann and Kapadia, 2014] Marcelo Kallmann and Mubbasir Kapadia. Navigation meshes and real-time dynamic planning for virtual worlds. In *ACM SIGGRAPH 2014 Courses*, pages 1–81. 2014.

[Krishnan and Mendoza Santos, 2021] Sivanand Krishnan and Rochelle Xenia Mendoza Santos. Real-time asset tracking for smart manufacturing. *Implementing Industry 4.0: The Model Factory as the Key Enabler for the Future of Manufacturing*, pages 25–53, 2021.

[Li *et al.*, 2010] Zhisheng Li, Ken CK Lee, Baihua Zheng, Wang-Chien Lee, Dik Lee, and Xufa Wang. Ir-tree: An efficient index for geographic document search. *IEEE transactions on knowledge and data engineering*, 23(4):585–599, 2010.

[Luria *et al.*, 2016] Michal Luria, Guy Hoffman, Benny Megidish, Oren Zuckerman, and Sung Park. Designing vyo, a robotic smart home assistant: Bridging the gap between device and social agent. In *2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 1019–1025. IEEE, 2016.

[Mouratidis *et al.*, 2005] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 634–645, 2005.

[Nash *et al.*, 2007] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Theta^*: Any-angle path planning on grids. In *AAAI*, volume 7, pages 1177–1183, 2007.

[Oh and Leong, 2017] Shunhao Oh and Hon Wai Leong. Edge n-level sparse visibility graphs: Fast optimal any-angle pathfinding using hierarchical taut paths. In *Tenth Annual Symposium on Combinatorial Search*, 2017.

[Ooi, 1987] Beng Chin Ooi. Spatial kd-tree: an indexing mechanism for spatial database. In *COMPSAC 87, The Eleventh Annual Int. Comp. Software & App. Conf.*, pages 433–438, 1987.

[Shen *et al.*, 2020] Bojie Shen, Muhammad Aamir Cheema, Daniel Harabor, and Peter J. Stuckey. Euclidean pathfinding with compressed path databases. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4229–4235. ijcai.org, 2020.

[Shen *et al.*, 2022] Bojie Shen, Muhammad Aamir Cheema, Daniel D Harabor, and Peter J Stuckey. Fast optimal and bounded suboptimal euclidean pathfinding. *Artificial Intelligence*, 302:103624, 2022.

[Smith and Chang, 1994] John Smith and S-F Chang. Quadtree segmentation for texture-based image query. In *Proceedings of the second ACM international conference on Multimedia*, pages 279–286, 1994.

[Sturtevant, 2012] Nathan R. Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

[Umair *et al.*, 2021] Muhammad Umair, Muhammad Aamir Cheema, Omer Cheema, Huan Li, and Hua Lu. Impact of covid-19 on iot adoption in healthcare, smart homes, smart buildings, smart cities, transportation and industrial iot. *Sensors*, 21(11):3838, 2021.

[Uras and Koenig, 2015] Tansel Uras and Sven Koenig. Speeding-up any-angle path-planning on grids. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 234–238, 2015.

[Xu *et al.*, 2022] Tao Xu, Aopeng Xu, Joseph Mango, Pengfei Liu, Xiaqing Ma, and Lei Zhang. Efficient processing of top-k frequent spatial keyword queries. *Scientific Reports*, 12(1):1–17, 2022.

[Yap *et al.*, 2011] Peter Yap, Neil Burch, Robert Craig Holte, and Jonathan Schaeffer. Block a*: Database-driven search with applications in any-angle path-planning. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[Zhao *et al.*, 2018a] Shizhe Zhao, Daniel D Harabor, and David Taniar. Faster and more robust mesh-based algorithms for obstacle k-nearest neighbour. *arXiv preprint arXiv:1808.04043*, 2018.

[Zhao *et al.*, 2018b] Shizhe Zhao, David Taniar, and Daniel D Harabor. Fast k-nearest neighbor on a navigation mesh. In *Eleventh Annual Symposium on Combinatorial Search*, 2018.