A Comparative Study of *k*-Nearest Neighbour Techniques in Crowd Simulation

Jordi L. Vermeulen
Utrecht University
jordi@jordivermeulen.nl

Arne Hillebrand
Utrecht University
A.Hillebrand@uu.nl

Roland Geraerts
Utrecht University
R.J.Geraerts@uu.nl

Abstract

The *k*-nearest neighbour (*k*NN) problem appears in many different fields of computer science, such as computer animation and robotics. In crowd simulation, *k*NN queries are typically used by a collision-avoidance method to prevent unnecessary computations. Many different methods for finding these neighbours exist, but it is unclear which will work best in crowd simulations, an application which is characterised by low dimensionality and frequent change of the data points. We therefore compare several data structures for performing *k*NN queries. We find that the nanoflann implementation of a **k**-d tree offers the best performance by far on many different scenarios, processing 100,000 agents in about 35 milliseconds on a fast consumer PC.

Keywords: crowd simulation, nearest neighbours, comparative study

1 Introduction

Algorithms that find the *k*-nearest neighbours (*k*NN) are popular in many different fields of computer science, such as computer animation, robotics, machine learning, databases, computer vision and computational geometry. In these fields the data typically has high dimensionality, and there is a clear separation between an *offline* phase in which an index is constructed, and an *online* phase in which queries are performed. In addition, many data structures are optimised for use cases in which not all data fits into main memory.

In crowd simulation, efficient collision avoidance is important, and collision-avoidance methods require an agent (such as a virtual pedestrian or robot) to know several of its nearest neighbours (e.g. [1, 2]). Choosing a high-performance method for finding these neighbours is therefore important when simulating massive crowds. To our knowledge, a comparative study of different *k*NN methods was never performed in a comparable setting. The properties in this setting are different from those mentioned above: the data usually have only two dimensions (i.e. agents with an x- and y-coordinate), fit into main memory, and the data continuously change, making it ill-suited to the notion of a costly offline indexing phase. Due to these differences it is unclear which *k*NN data structure offers the best performance in applications such as crowd simulation.

Contribution: we compare several data structures for finding nearest neighbours in the context of crowd simulation. We test these structures on a wide variety of scenarios, giving a representative picture of the performance that can be expected. We find that using the nanoflann implementation of the **k**-d tree [3] allows us to build the tree and query it for 100,000 agents in about 35 milliseconds per time step. This method also has the lowest standard deviation of the query times.

This paper is structured as follows. In Section 2, we motivate our choice of data structures. In Section 3, we give theoretical bounds for the tested data structures and describe their implementation details. In Section 4, we describe our experimental setup, scenarios and results. In Section 5 we conclude that the nanoflann **k**-

Table 1: Worst-case construction and query times on n 2D points

Data structure	Construction time	kNN query time
k-d tree [6]	$O(n \log n)$	$O(k \log n)$
BD-tree [19]	$O(n \log n)$	$O(k \log n)$
R-tree [7]	$O(n \log n)$	$O(k \log n)$
Voronoi diagram [20]	$O(n \log n)$	$O(k \log n)$
k-means [21]	$O(n^2)$	$O(n)$
Linear search	$O(1)$	$O(n)$
Grid	$O(n)$	$O(n)$

d tree outperforms the other structures, and we provide some leads for future work.

2 Related work

For spatial data, the *k*NN problem is usually solved by building a spatial index [4]. These indices can broadly be divided into two categories: those that partition the space and those that partition the data. Well known instances of the former are quadtrees [5] and **k**-d trees [6], whereas the latter includes R-trees [7] and bounding volume hierarchies. The difference between the two is that spatial-partitioning methods recursively divide the remaining space into non-overlapping areas while trying to balance the number of objects inside each subdivision, whereas data-partitioning methods try to cluster the data in potentially overlapping areas, preferably based on spatial proximity. We test data structures from both categories, and describe each structure in more detail in the following section.

Several comparative studies of *k*NN methods exist (e.g. [8, 9, 10]), but these focus on applications of *k*NN methods in high-dimensional settings like data mining and computer vision.

When data points are constantly moving, kinetic data structures can be used to maintain geometric information [11]. However, because the trajectories of our agents change unpredictably, there does not appear to be a straightforward way to apply these techniques efficiently.

Recently, much research has been focused on using highly parallelised algorithms running on the GPU to speed up the computation of nearest neighbours; see e.g. [12, 13]. While the results are promising, implementations are generally compatible with only a single hardware vendor, and the performance may vary greatly depending on the specific hardware used, making it hard to offer a fair comparison to CPU-based implementations. As such, we do not consider GPU implementations at this time.

Several motion planning packages that use data structures to accelerate nearest neighbour queries exist. For instance, the *Open Motion Planning Library* [14] employs FLANN's hierarchical clustering [15] and the *Geometric Near-Neighbour Access Tree* [16], and the *Motion Strategy Library* [17] uses the **k**-d tree from the ANN library [18]. Such packages could be faster in low-dimensional scenarios by supporting the fastest method that we test here.

3 Data structures

We made our selection of data structures based on prevalence, theoretical performance and the availability of good implementations, taking into account the low dimensionality of our research domain and the fact that we are working in main memory. We settled on testing the data structures described below. A summary of their theoretical performance can be seen in Table 1. Note that we make a typographical distinction between the completely unrelated k's in *k*NN, **k**-d and k-means.

3.1 k*-d trees*

A **k**-d tree [6] is a spatial partitioning structure that recursively splits the data set along one of the **k** axes of the coordinate system. This is usually done in sequence: in the case where $\mathbf{k} = 2$ we split alternately on x- and y-coordinate.

Splitting continues until a maximum depth has been reached, or less than a specified number of points remains in a cell. **k**-d trees can be straightforwardly expanded to contain objects other than points, such as triangles or line segments. In this case it can be difficult to determine the optimal splitting plane; in our case of points (or disks) in two dimensions, we can simply sort the points for each dimension and split along the median. A **k**-d tree can be constructed in $O(n\ log\ n)$ time and can be used to find the *k*-nearest neighbours in $O(k\ log\ n)$ time.

3.2 BD-trees

The box-decomposition tree [19], or BD-tree (not to be confused with the bounded-deformation tree), is structurally similar to a **k**-d tree. It differs in two major ways: the rectangles that the space is decomposed into are *fat*, and each rectangle may have an associated *inner rectangle*. A fat rectangle is one with bounded ratio between the shortest and longest axis. Having fat rectangles ensures the size of the regions decreases exponentially as one traverses the tree. The *inner rectangles* allow a rectangle to be split not into two rectangles along some coordinate axis, but into an outer and an inner rectangle, where the resulting region is the set-theoretic difference between the two. A BD-tree can be constructed in $O(n\ log\ n)$ time and can be used to find the *k*-nearest neighbours in $O(k\ log\ n)$ time.

3.3 R-trees

R-trees were created as a data partitioning structure supporting multi-dimensional data in multi-dimensional spaces [7]. They are also fully dynamic: queries may be mixed with insertions and deletions without a need for periodic rebalancing. R-trees are based on B-trees, and as such are structurally similar: the number of objects in a leaf node, as well as the number of children of an internal node, is limited by both a lower and an upper bound, and all leaves appear on the same level. Many different heuristics exist for the construction of the structure; in our case we use the R*-tree [22], because it tries to prevent overlapping regions. An R*-tree can be constructed in $O(n\ log\ n)$ time and can be used to find the *k*-nearest neighbours in $O(k\ log\ n)$ time. Because R-trees theoretically handle updates very well, we test both a version in which we rebuild the entire tree every time step, and one in which we remove and insert every agent incrementally.

3.4 Voronoi diagrams

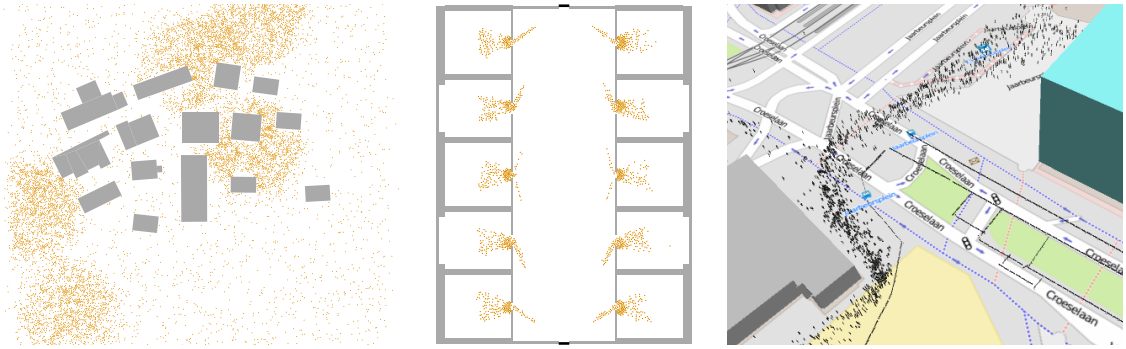
Voronoi diagrams are spatial subdivisions where each cell is exactly the set of points closest to a site [20]. We can then find the *k*-nearest neighbours by an algorithm similar to Dijkstra's algorithm for single-source shortest paths. We add a site's neighbouring cells to a priority queue, where our priority is the distance of a cell's site to the query point. We then repeatedly extract the cell with minimum distance from the queue and add its neighbours. A Voronoi diagram can be constructed in $O(n\ log\ n)$ time, and, if we use a binary heap as our priority queue, we can find the *k*-nearest neighbours in $O(k\ log\ n)$ time.

3.5 Hierarchical k-means clustering

`k`-means clustering was originally conceived to partition a population based on a sample [21]. The algorithm works by initialising `k` sites at some location and calculating for each data point which of the `k` sites it is closest to. Many seeding heuristics exist; in our application the sites are chosen randomly. Each site is then moved to the mean of all points for which it is the closest site. This process is repeated until the sites no longer move, or until a maximum number of iterations has been performed. A *hierarchical* clustering then recursively performs this algorithm on each of the clusters that has been created. The spreading of points over the clusters can be arbitrarily bad, giving us a worst-case construction time of $O(n^2)$ and a query time of $O(n)$, but if each cluster is consistently of roughly equal size, we get a construction time of $O(n\ log\ n)$ and a query time of $O(k\ log\ n)$.

3.6 Linear search and grids

Linear search is a naive, brute-force approach to the *k*NN problem: we simply iterate over all points in the set, keeping track of the *k* closest points so far. This gives a query time of $O(n)$,



(a) (b) (c)

Figure 1: Some of the different test scenarios. (a) shows the scenario with a clustered distribution of agents, (b) shows the evacuation scenario, and (c) shows one of the Tour de France scenarios.

but it has the advantage of requiring no construction whatsoever. A grid-based approach tries to improve on this in a simple way: we place a regular grid over the entire simulation area, and keep track of the agents located in each cell. During a query, we search the cell containing the query points, as well as those cells directly adjacent. The construction time is $O(n + m)$, where m is the number of cells in the grid. The worst case query time is $O(n)$, but practical performance depends heavily on both the distribution of agents and the size of the cells.

3.7 Implementations

The BD-tree was provided by the ANN library [18]. The FLANN library provided one of the k-d trees and the k-means method [15]; another k-d tree came from the nanoflann library [3]. The Boost.Geometry library was used for the R-tree and Voronoi diagrams [23]. Finally, linear search and the grid-based method were implemented by us. Note that, although the FLANN library supports the calculation of *approximate* nearest neighbours, we only use it for finding the exact nearest neighbours.

We had to make a minor modification to the ANN library to support querying from multiple threads (i.e. one variable containing state pertaining to the current query was changed from a global to a local variable). Also, while Boost supports building a Voronoi diagram, we implemented the algorithm for finding the k nearest sites ourselves, as described in Section 3.4.

Other than Boost's R-tree implementation and the grid, none of the structures straightforwardly supported adding and removing points, so for all other structures we rebuild them completely every time step. Furthermore, we use the default parameters for creating the spatial indices for each library; notably, this means that the number of clusters in the hierarchical k-means clustering is 32 per level of the hierarchy. Finally, we set the cell size for the grid to 10 metres, as this is roughly the distance in which we normally expect to find neighbours.

4 Experiments

We used data captured from real crowds and from simulations that have real-world applications. To also give some insight into specific aspects of the data structures, we introduced some artificial scenarios that highlight specific properties of the crowds or environments.

4.1 Experimental setup

As our data captured from real crowds is only available as a list of 2D coordinates per agent for each time step, we converted all our other data to this form as well, rather than performing the tests during the simulation. For each structure, we read these data per time step, updated the structure, then performed a *k*NN query for each agent. For all our tests, $k = 10$; we do not vary this parameter because collision-avoidance methods generally don't need more than 10 of the closest neighbours. Each time step, we measured the time taken to update the structure and

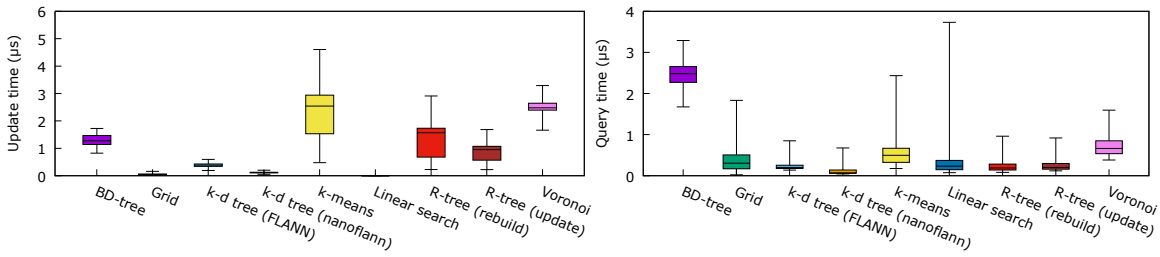


Figure 2: A box plot of the time needed for updating and querying *per agent per time step* across all experiments. The whiskers extend to the 5th and 95th percentile.

the time taken to perform all queries. To better approximate the real usage of these structures in a crowd simulation framework, we performed multiple *k*NN queries in parallel using OpenMP [24].

All our code was written in C++, as were all the libraries we used. Our experiments were performed on a desktop computer with two Intel Xeon E5-2690 v3 12-core processors and 32 GB of DDR4 ECC RAM, running Ubuntu 15.10. We used g++ 5.2.1 to compile our program.

4.2 Scenarios

Our data from real crowds was obtained from the Jülich Forschungszentrum's *Institute for Advanced Simulation*. Descriptions of the various situations can be read in reference [25]. They provide data captured from crowds in several scenarios. We used the data from the scenarios bidirectional flow, free choice of destination; bidirectional flow, ordered destination, symmetric flow rate; bottleneck; mouth hole in stadium, lower level; and mouth hole in stadium, upper level. We chose these scenarios for the presence of a relatively large number of people, as well as their relative complexity.

We also obtained data from simulations made in the ECM crowd simulation framework [26]. We used scenarios that were developed for the planning of the Grand Départ of the Tour de France [27], which were used to simulate different placement of things such as fences and footbridges. We also used scenarios simulating the evacuation of a building with different numbers of people present. A selection of our test scenarios can be seen in Figure 1.

As we also wanted to obtain data on specific aspects of the performance of the different structures, we included several artificial scenarios. We were interested in how the following aspects affect the performance:

Density: we wanted to know how the distribution of agents affected the performance of the different structures. We included one scenario with 10,000 agents distributed uniformly at random, and one in which 75% of agents is spawned in five high-density clusters, and 25% uniformly at random.

Stationary agents: if many of the agents are standing still, we would expect methods that update rather than rebuild their structures to perform better. We included scenarios in which 25%, 50% or 75% of the agents are standing still.

Scaling: we also wanted to test how the different structures scale in practice with increas-

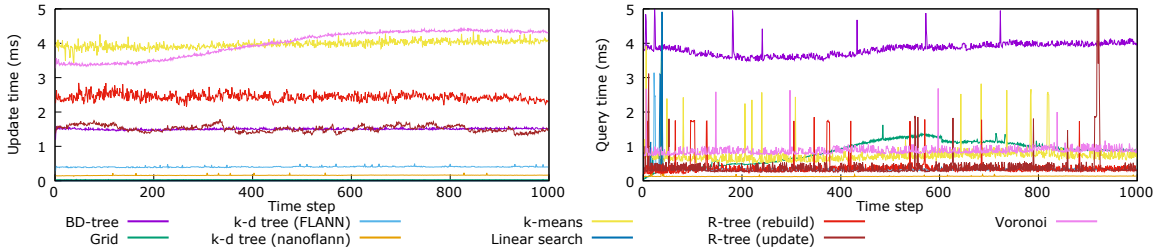


Figure 3: Total update and query times per time step for the evacuation scenario. There is a constant number of 1470 agents in the simulation.

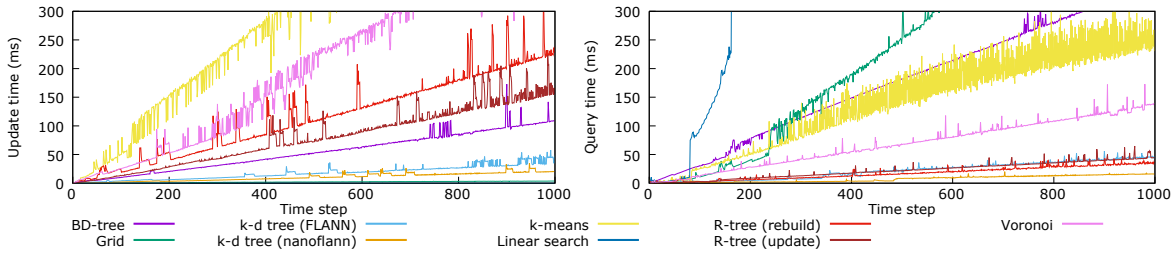


Figure 4: Total update and query times per time step for the scaling scenario. The number of agents starts at 0 and increases by 100 every time step.

ing numbers of agents. We included a scenario where 100 agents are added to the simulation every time step. We ran this simulation for 1000 time steps, scaling up to 100,000 agents.

4.3 Results

As the number of experiments is quite large (62 different scenarios) and the results for each category of experiments are quite similar, we only describe the results for one instance of each category.

4.3.1 General remarks

Looking at Figure 2, we see that the grid- and k-d tree-based methods offer the best update performance (disregarding linear search, which has no update). We also see that the query performance of the k-d trees and R-trees are better than those of the other methods. It is worth noting that this graph favours those methods that are fast for small numbers of agents (e.g. grid, linear search), as there were only a handful of tests with more than about 2,000 agents.

We note that there is a lot of noise in the measurements, especially in scenarios with a small

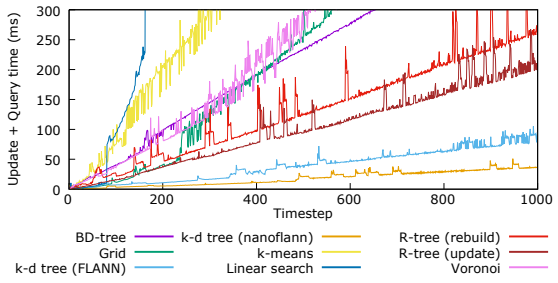


Figure 5: The sum of update and query times for the scaling scenario. The number of agents starts at 0 and increases by 100 every time step.

number of agents (such as the bottleneck scenario seen in Figure 7). For the query performance, preliminary tests indicate this is due to the usage of a large number of threads (one for each core, 24 total) on a small number of agents, meaning the scheduling overhead is large. The noise is greatly reduced when running the simulation on a single thread. The variance of the computation times for both updates and queries is most likely also influenced by the cache performance of each data structure. Our measurements suggest that k-d trees are particularly consistent in their performance, as the variance of the performance is much lower for both implementations than it is for other data structures. We also note that the peaks are not in the same locations when the experiments are repeated, meaning they are not related to the specific layout of the data points in those time steps.

The variance of the update time for the hierarchical k-means clustering is particularly high. This is likely due to the nature of the construction algorithm: depending on the quality of the initial (random) cluster locations, the algorithm needs more or fewer iterations to reach a solution. The variance for the update time for the R-tree which is rebuilt each time step is also larger than that of the R-tree which is updated incrementally, indicating that the construction algorithm is sensitive to the particular layout of the data points, unlike the method for removing and inserting one point at a time.

Overall, the nanoflann implementation of the k-d tree clearly gives the best performance: it is only outperformed in situations with very few agents, and it can process 100,000 agents in about 35 milliseconds per time step. It also has the most consistent performance, as can be seen in Figure 2: it has the smallest difference be-

tween the 5th and 95th percentile of all structures.

4.3.2 Evacuation scenario

Figure 3 shows the update and query performance on the evacuation scenario. The grid-based method shows that the update cost is very constant (and low). The query time, however, increases as the local density of agents increases, due to crowding at the exits (see Figure 1(b)). The Voronoi diagram-based method shows the opposite effect: the query time is fairly constant, but the cost of computing the Voronoi diagram increases as the density does. The nanoflann **k**-d tree has the best query performance; the FLANN **k**-d tree, linear search and both R-tree versions give similar query performance, although the R-trees have more variance in their performance. However, the FLANN **k**-d tree has about four times better update performance, and linear search requires no updates at all. This makes it quite competitive with the nanoflann **k**-d tree for this number of agents, which is still about 20% faster when considering both update and query times.

4.3.3 Scaling test

Figure 4 shows the change in performance as the number of agents steadily increases. We see that linear search quickly becomes infeasible: it performs relatively well until about 8000 agents, but then quickly starts to slow down; at 100,000 agents, it takes about 16 seconds to perform all the queries for a single time step. The grid-based method does slightly better, but at about 25,000 agents its performance also quickly deteriorates. The BD-tree and `k`-means method offer comparable query performance, but the cost of updating the `k`-means structure quickly rises as the number of agents grows. The same can be said for both R-tree methods and the FLANN **k**-d tree: the query performance is similar, but the R-trees are over three times more expensive to update.

Figure 5 shows the sum of the update and query times for this scenario, showing how much time each structure needs per time step in total. Here we see that the grid-based method is actually somewhat faster than the `k`-means

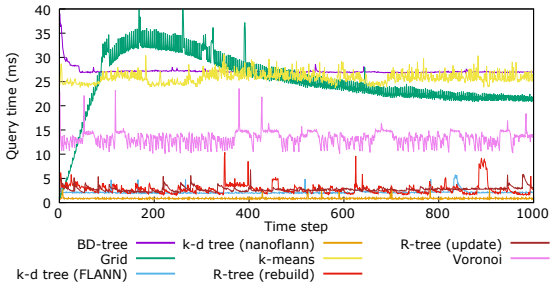


Figure 6: Total query times per time step for the clusters scenario. There are 10,000 agents in the simulation. Linear search has been omitted to show more detail in the other graphs.

method on the whole, but the curve of the graph suggests this will quickly change if even more agents are added. The update version of the R-tree is about 20% faster than the version that rebuilds the tree every time step. The **k**-d trees are the fastest by a large margin, and the nanoflann implementation is about twice as fast as the FLANN version.

4.3.4 Density test

In the clusters scenario, several locations in the environment quickly get congested, resulting in a sharp increase in the query time of the grid-based method, as seen in Figure 6. This is because some grid cells contain a disproportionally large number of agents: the cell size is too large for the local density. None of the other methods seem to be particularly affected by changes in density.

4.3.5 Bottleneck scenario

In Figure 7, the update and query performance of all structures on the bottleneck scenario are shown. We note that there is a strong correlation between the number of agents and the query and update times of all structures. The BD-tree is clearly the slowest in query performance, whereas the Voronoi diagram-based method needs the most time to update the structure. For this small number of agents, the grid-based method and linear search are very efficient, but the nanoflann implementation of **k**-d trees offers comparable performance.

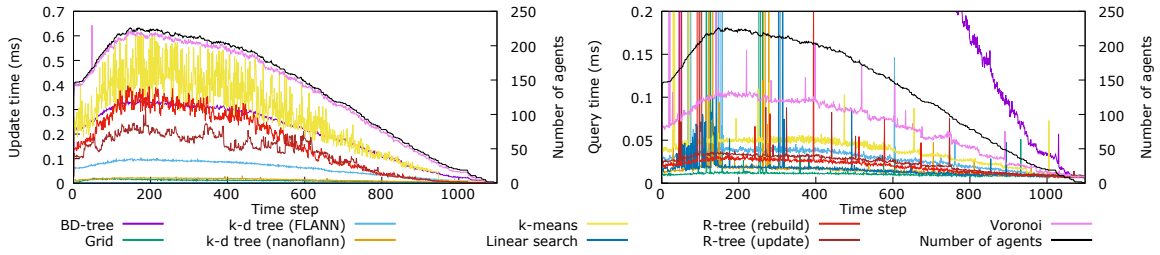


Figure 7: Total update and query times per time step for the bottleneck scenario.

4.3.6 Tour de France scenario

In Figure 8, the BD-tree once again clearly has the worst query performance. We also note, however, that this is not a suitable setting for the grid-based method, due to the high local densities and small walkable area in a large environment. Furthermore, we observe that the methods based on k-d trees clearly give more consistent performance than those using other spatial indexing structures. There are some unexpected peaks in the update time for the k-means method for which we have no definitive explanation; they may have been caused by another program temporarily running on the same core.

4.3.7 Stationary agents test

Figure 9 shows that the three tests with varying degrees of stationary agents perform as expected: the implementation that updates an R-tree rather than rebuilding it has progressively better update performance as more agents are stationary, while other structures are unaffected. With 25% of the agents standing still, the R-tree has an average update time of 10 milliseconds per time step; this drops to 3.6 milliseconds when 75% of the agents do not move. This is comparable to the FLANN k-d tree, which needs an average of 3.2 milliseconds, but still significantly slower than the nanoflann version, which requires 1.6 milliseconds on average.

5 Conclusion

Having tested nine different implementations of structures for finding the k nearest neighbours of agents in a simulated crowd on a variety of scenarios, we can conclude that the nanoflann implementation of the k-d tree is the fastest by a large margin, even for a moderate number of agents, allowing us to process 100,000 agents in about 35 milliseconds on a fast PC. Its performance also has the lowest variance of all structures. A grid-based approach can be efficient for moderate numbers of agents (up to about 1000), mostly because updating the structure is cheap, but this method is sensitive to the spatial distribution of agents.

There are several areas of interest that could be explored in future work. In a simulation, each agent only ever moves a small distance per time step. We could potentially exploit this by only updating the query structure once every few time steps. We could even guarantee to find a superset of the exact result by increasing the search radius by twice the maximum distance any agent can move between updates. This modification could potentially give a great performance boost, but research is needed to determine the number of time steps we could delay

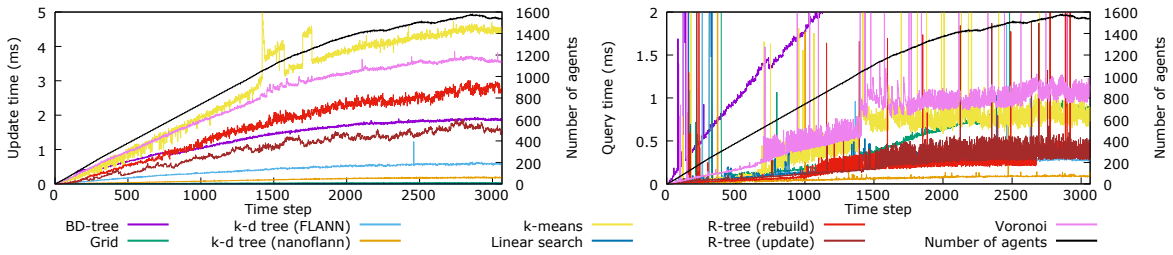


Figure 8: Total update and query times per time step for the Tour de France scenario.

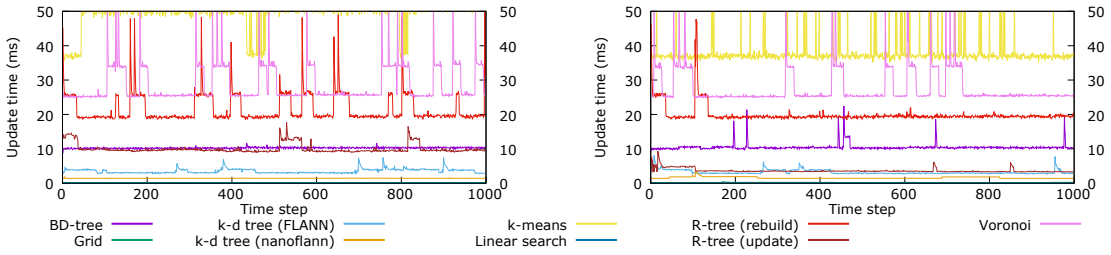


Figure 9: Total update times per time step for the stationary agents scenario. The left and right graphs are of the scenario with 25% and 75% stationary agents, respectively. Note that only the R-tree with updates (the brown line) has a significant drop in computation time.

the update by.

Additionally, the performance of the Voronoi diagram-based method could be improved by implementing a point location algorithm and by employing higher-order Voronoi diagrams [28]. However, it seems unlikely that this will make it faster than a k-d tree.

We are currently working on a solution of the *k*NN problem in *multi-layered* environments, such as a building with multiple floors. In such environments, we cannot simply give the agents with the smallest Euclidean distance in 2D, as these could be on a different floor. Instead, we must take visibility of agents into account when computing the *k* nearest neighbours.

We think that this study can benefit developers of software packages that perform *k*NN queries in a comparable setting.

References

[1] Stephen J. Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming Lin, Dinesh Manocha, and Pradeep Dubey. Clearpath: highly parallel collision avoidance for multi-agent simulation. In *Proc. ACM SIGGRAPH/Eurographics Symp. Comp. Anim.*, pages 177–187, 2009.

[2] Jur van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *IEEE Int. Conf. Robot. Autom.*, pages 1928–1935, 2008.

[3] Jose Luis Blanco-Claraco. nanoflann, 2015. `https://github.com/jlblancoc/nanoflann` (version 1.1.9, accessed 17-01-2016).

[4] Marius Muja and David G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36, 2014.

[5] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comp. Surv.*, 16(2):187–260, 1984.

[6] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[7] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pages 47–57, 1984.

[8] Alexander Ponomarenko, Nikita Avrelin, Bilegsaikhan Naidan, and Leonid Boytsov. Comparative Analysis of Data Structures for Approximate Nearest Neighbor Search. *DATA ANALYTICS*, pages 125–130, 2014.

[9] Deepika Verma, Namita Kakkar, and Neha Mehan. Comparison of Brute-Force and KD Tree Algorithm. *Int. J. of Adv. Res. in Comp. and Commun. Eng.*, 3(1):5291–5294, 2014.

[10] Nitin Bhatia and Vandana. Survey of Nearest Neighbor Techniques. *Int. J. of Comp. Sci. and Inf. Secur.*, 8(2):302–305, 2010.

[11] Leonidas J. Guibas. Kinetic data structures: A state of the art report. In *Proc. 3rd Workshop Alg. Found. Robot.*, pages 191–209, 1998.

[12] Marcelo de Gomensoro Malheiros and Marcelo Walter. Spatial sorting: an efficient strategy for approximate nearest

neighbor searching. *Computers & Graphics*, 57:112–126, 2016.

[13] Jia Pan, Christian Lauterbach, and Dinesh Manocha. Efficient nearest-neighbor computation for GPU-based motion planning. In *IEEE/RSJ Int. Conf. on Intell. Robots and Systems*, pages 2243–2248, 2010.

[14] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robot. & Autom. Mag.*, 19(4):72–82, 2012. `http://ompl.kavrakilab.org` (version 1.1.0, accessed 17-01-2016).

[15] Marius Muja and David G. Lowe. FLANN - Fast Library for Approximate Nearest Neighbors, 2012. `http://www.cs.ubc.ca/research/flann/` (version 1.8.4, accessed 17-01-2016).

[16] Sergey Brin. Near neighbor search in large metric spaces. In *Proc. 21st Int. Conf. Very Large Data Bases*, pages 574–584, 1995.

[17] Steve LaValle. Motion Strategy Library, 2003. `http://msl.cs.uiuc.edu/msl/` (version 2.0, accessed 17-01-2016).

[18] David M. Mount and Sunil Arya. ANN: A Library for Approximate Nearest Neighbor Searching, 2010. `http://www.cs.umd.edu/~mount/ANN/` (version 1.1.2, accessed 17-01-2016).

[19] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[20] Georgy Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier mémoire. sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die reine und angewandte Mathematik*, 133:97–178, 1908.

[21] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. 5th Berkeley Symp. Math. Stat. Probab.*, volume 1, pages 281–297, 1967.

[22] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pages 322–331, 1990.

[23] Barend Gehrels, Lalande Bruno, Loskot Mateusz, Wulkiewicz Adam, and Karavelas Menelaos. Boost Geometry Library, 2016. `http://www.boost.org/libs/geometry` (version 1.60, accessed 17-01-2016).

[24] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf` (accessed 17-01-2016).

[25] Christian Keip and Kevin Ries. Dokumentation von Versuchen zur Personenstromdynamik, 2009. `http://ped.fz-juelich.de/experiments/2009.05.12_Duesseldorf_Messe_Hermes/docu/VersuchsdokumentationHERMES.pdf` (accessed 17-01-2016).

[26] Wouter G. van Toll, Norman S. Jaklin, and Roland Geraerts. Towards Believable Crowds: A Generic Multi-Level Framework for Agent Navigation. *ICT.OPEN 2015*, 2015.

[27] Marijn van der Zwan. The Impact of Density Measurement on the Fundamental Diagram. Master's thesis, Utrecht University, 2015. ICA-3401928.

[28] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *16th Annu. Symp. Found. Comp. Sci.*, pages 151–162, 1975.