A comparative study of *k*-nearest neighbour techniques in crowd simulation

Jordi Vermeulen Arne Hillebrand Roland Geraerts

Department of Information and Computing Sciences
Utrecht University, The Netherlands

30th Conference on Computer Animation and Social Agents, May 23, 2017

Context

We want efficient crowd simulations.

Context

Large amount of computation spent on collision avoidance. Needs several nearest neighbours.

Which method for finding nearest neighbours is most efficient?

Efficient:

- Construction
- Querying
- Variance

Context

The k-nearest neighbour (kNN) problem is well-known.

- Robotics
- Machine learning
- Databases
- Computer vision
- ...

Usually: high dimensionality, separation between offline construction and online querying, disk storage.

Our case: two or three dimensions, changing data, main memory.

Data structures

Data structures selected on prevalence and availability of good implementations.

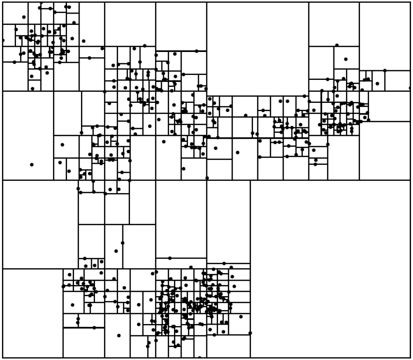
We tested:

Data structure	Construction time	kNN query time
k-d tree	$O(n \; log \; n)$	$O(k \; log \; n)$
BD-tree	$O(n \; log \; n)$	$O(k \; log \; n)$
R-tree	$O(n \; log \; n)$	$O(k \; log \; n)$
Voronoi diagram	$O(n \; log \; n)$	$O(k \; log \; n)$
$\mathtt{k}$-means	$O(n^2)$	$O(n)$
Linear search	$O(1)$	$O(n)$
Grid	$O(n)$	$O(n)$

k-d tree

Split alternatingly along axes.

Try to split remaining data in half.

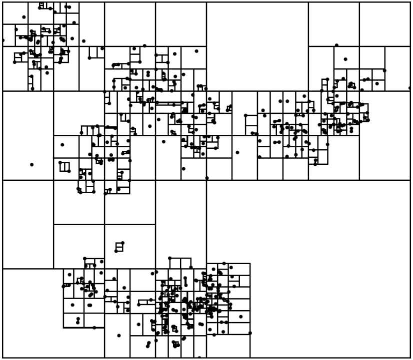


https://www.cs.umd.edu/~mount/ANN/Files/1.1.2/ANNmanual_1.1.pdf

Box-decomposition tree

k-d tree with extra split rule.

Split into inner and outer box.



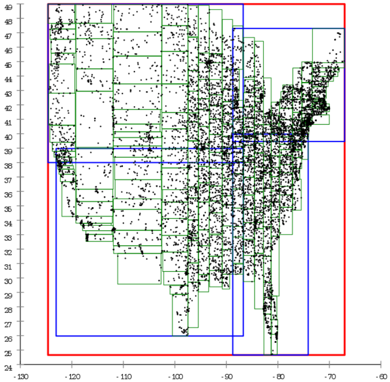
https://www.cs.umd.edu/~mount/ANN/Files/1.1.2/ANNmanual_1.1.pdf

R-tree

Point or volumetric data.

Partitions may overlap.

Insertion and deletion of data possible.



https://en.wikipedia.org/wiki/R-tree

Hierarchical k-means clustering

Assign points to centroid.

Calculate new centroid and iterate.

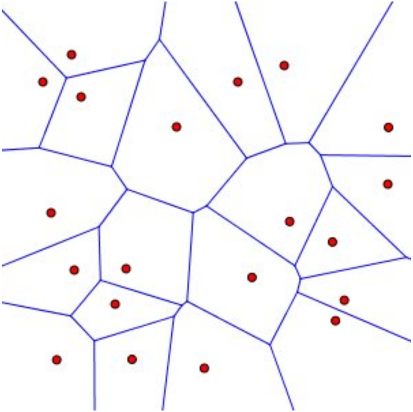
Apply hierarchically.

http://rossfarrelly.blogspot.com/2012/12/k-means-clustering.html

Voronoi diagrams

Cells of points closest to site.

Find nearest neighbours by examining neighbouring cells.



http://merganser.math.gvsu.edu/david/voronoi.08.06/

Implementations

k-d tree implementations provided by FLANN [1] and nanoflann [2].

- FLANN: general-purpose implementation
- nanoflann: highly optimised for 2D and 3D data

FLANN also provides `k`-means implementation.

BD-tree is provided by ANN [3].

---

[1] Muja and Lowe, *FLANN - Fast Library for Approximate Nearest Neighbors* (`http://www.cs.ubc.ca/research/flann/`)

[2] Blanco-Claraco, *nanoflann* (`https://github.com/jlblancoc/nanoflann`)

[3] Mount and Arya, *ANN: A Library for Approximate Nearest Neighbor Searching* (`http://www.cs.umd.edu/~mount/ANN/`)

Implementations

R-tree and Voronoi diagrams are provided by Boost [1].

R-tree has good update performance, test two versions:

1. Rebuild entire tree each time step
2. Update tree incrementally

Linear search and grid are own implementations.

[1] Gehrels et al., *Boost Geometry Library* (`http://www.boost.org/libs/geometry`)

Scenarios

Test on artificial and real-world scenarios.

Artificial: test specific properties.

- Density: uniform vs clustered
- Stationary agents: test with 25, 50 or 75% of agents not moving
- Scaling: add more agents each time step

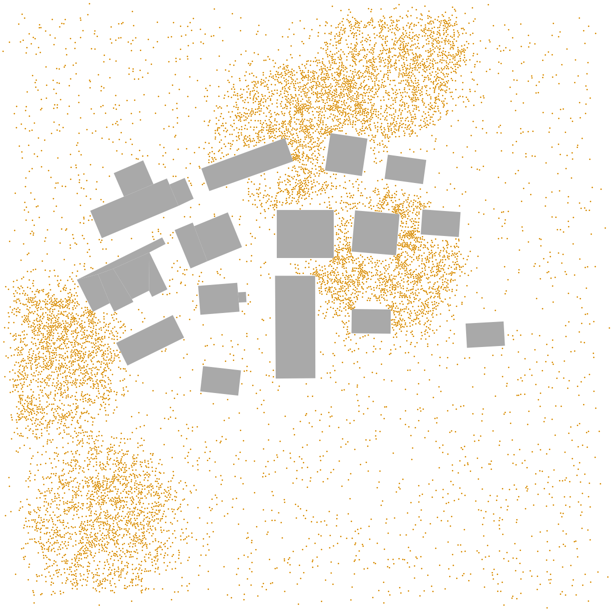
Real-world:

- Simulations of evacuation of building
- Simulations for Tour de France [1]
- Jülich trajectory data of real crowds [2]

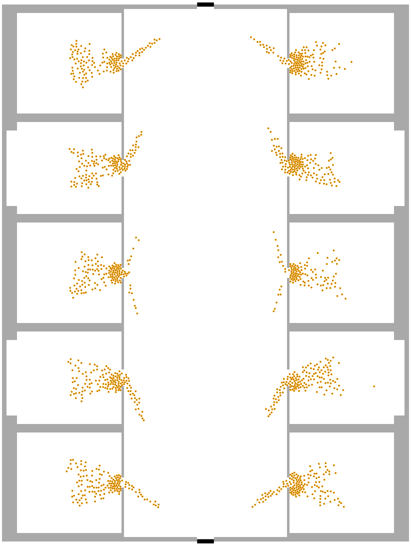
[1] van der Zwan, *The Impact of Density Measurement on the Fundamental Diagram*

[2] Keip and Ries, *Dokumentation von Versuchen zur Personenstromdynamik*

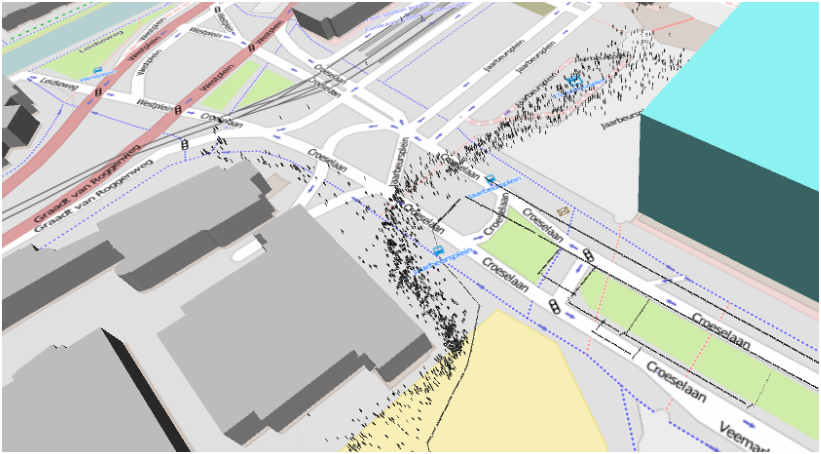
Scenarios - density



Scenarios - evacuation



Scenarios - Tour de France



Scenarios - Jülich bottleneck

Experimental setup

Jülich data only available as trajectories (tuples of id, time, x- and y-coordinate).

For fair comparison, converted all data to trajectories.

C++ testing program reads data per time step, and:

1. Builds the structure for agent positions at current time step
2. Performs *k*NN query for each agent

For realism, queries are performed in parallel.

We fix *k* at 10; collision avoidance does not need more.

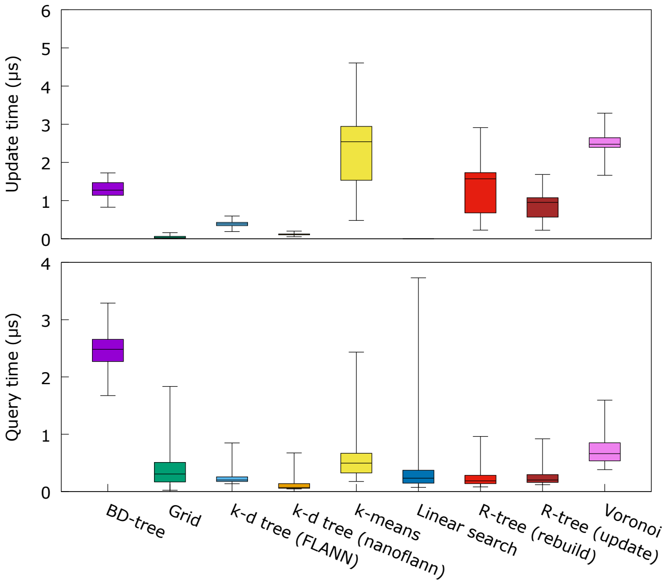
Results

Total of 62 different scenarios: multiple instances of similar settings.

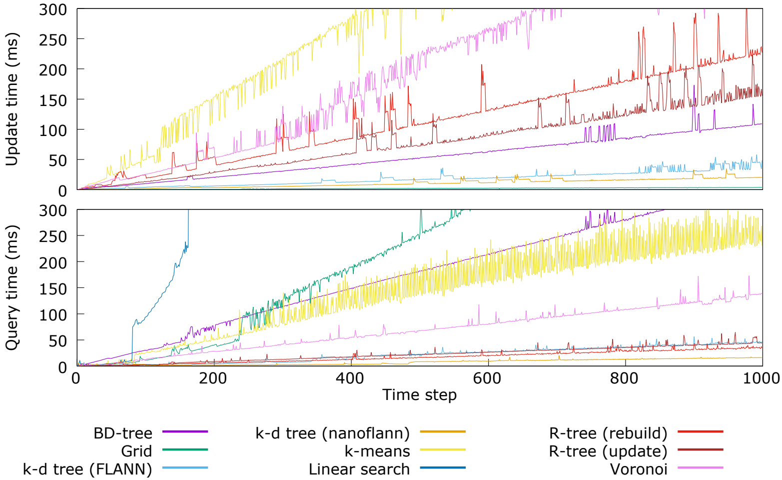
Tested on machine running Ubuntu 15.10, with two Xeon 12-core processors and 32 GB of DDR4 RAM.

Results

Overall results per agent per time step:

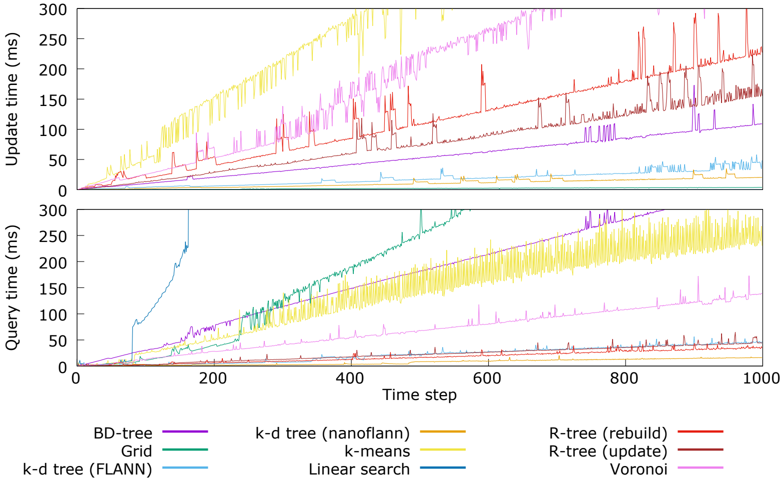


Results - scaling



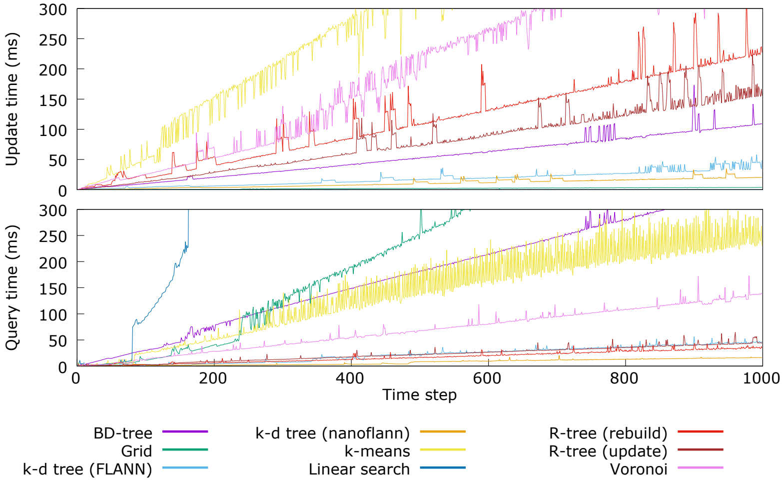
- Linear search quickly infeasible: 16 seconds per time step for 100,000 agents

Results - scaling



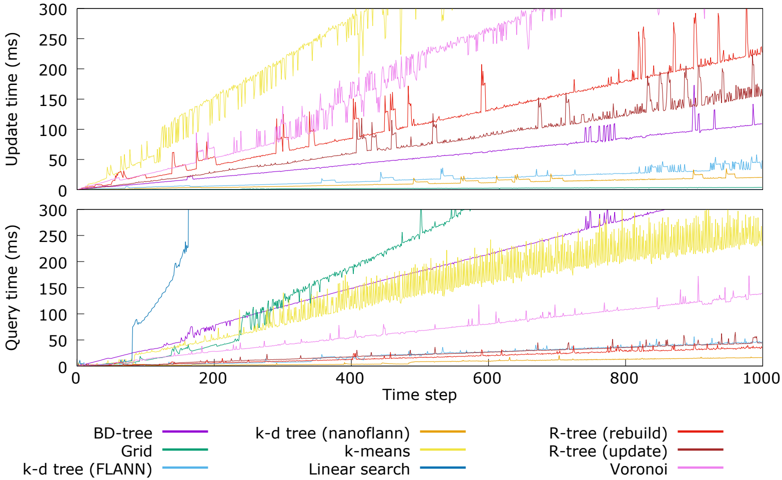
- R-tree and FLANN **k**-d tree have similar query performance, but R-tree over 3x more expensive to update

Results - scaling



- R-tree update 20% faster than rebuild

Results - scaling



- nanoflann 2x faster than FLANN: 100,000 agents in ~35 ms

Conclusion

nanoflann implementation of **k**-d tree clearly best option.

- Fastest except when number of agents very small
- Lowest variance
- 100,000 agents in 35 ms per time step

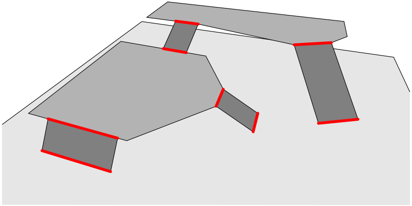
Grid competitive for small number of agents (< 1000) due to low update cost. Linear search efficient up to a few hundred agents.

Updating R-tree more efficient than rebuilding.

Future work

Currently working on extending *k*NN algorithm to *multi-layered environments*, e.g. buildings with multiple floors.

- Euclidean nearest neighbours not enough: close x- and y-coordinates may be on different floor
- Need to consider visibility



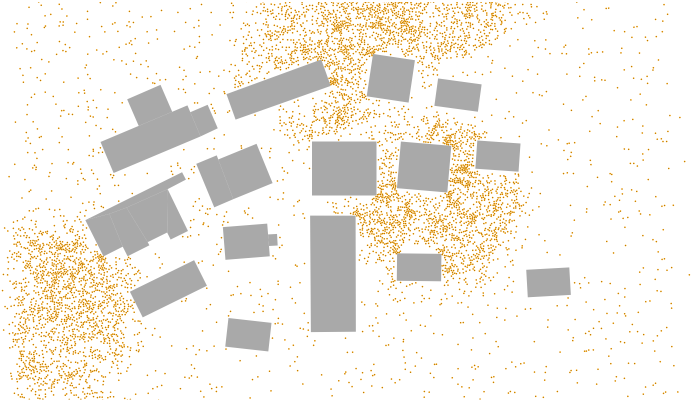
Future work

Local neighbourhood does not change much between time steps: could update only once every few steps.

- How often should we update?

Compare performance of GPU methods, looking for people with expertise.

Thanks!



J.L.Vermeulen@uu.nl

Utrechts Universiteitsfonds | Universiteit Utrecht