An Indexed K-D Tree for Neighborhood Generation in Swarm Robotics Simulation

Zhongyang Zheng and Ying Tan*

[1] Key Laboratory of Machine Perception and Intelligence(Peking University), Ministry of Education
[2] Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
ytan@pku.edu.cn

Abstract. In this paper, an indexed K-D tree is proposed to solve the problem of neighborhoods generation in swarm robotic simulation. The problem of neighborhoods generation for both robots and obstacles can be converted as a set of range searches to locate the robots within the sensing areas. The indexed K-D tree provides an indexed structure for a quick search for the robots' neighbors in the tree generated by robots' positions, which is the most time consuming operation in the process of neighborhood generation. The structure takes full advantage of the fact that the matrix generated by robots neighborhoods is symmetric and avoids duplicated search operations to a large extent. Simulation results demonstrate that the indexed K-D tree is significantly quicker than normal K-D tree and other methods for neighborhood generation when the population is larger than 10.

Keywords: Neighborhood generation, k-d tree, simulation, range search, indexed k-d tree, swarm robotics.

1 Introduction

The researches of swarm robotic systems require plenty of physical robots, which makes it hard to afford for many research institutions [1]. Simulations on computers are developed to visually test the structures and algorithms on computer. Although the final aim is real robots, it is often very useful to perform simulations prior to investigations with real robots. Simulations are easier to setup, less expensive, normally faster and more convenient to use than physical swarms [2]. There exists several commonly-used simulation platforms, such as Player/Stage [3], Swarmanoid Simulator [4] and etc.

In the real-life swarm robotics applications, robots in the swarm can detect other robots within certain sensing ranges (they are inferred as neighbors in this paper) to exchange positions, current running states and other environment information. All these detections can be done with the help of on-board sensors. However, the problem becomes complicated in simulation, as we judge if

* Corresponding author.

Y. Tan, Y. Shi, and H. Mo (Eds.): ICSI 2013, Part II, LNCS 7929, pp. 53–62, 2013.

two robots are neighbors through calculating their distance. Thus, research on generating neighbors for simulations is necessary.

1.1 Neighborhood Generation in Simulation

In the simulation of swarm robotics, sensing range of robots is fixed and the area it covers shapes as a square or circle (cube or sphere in 3-D situations). We denote all the robots inside the sensing area of a robot as the neighborhood of this robot, which needs to be calculated every iteration to determine all the neighbors of that robot.

We also need to detect if robots are near any obstacles in the environment, i.e. generate the obstacles' neighborhoods. Obstacles can vary in many aspects. They can be static or dynamic, appear or disappear during the simulation. Their sizes and shapes can be different, from small points to large polygons and they can have different sensing areas and only robots within the areas can detect them.

The problem of neighborhood generation can be defined as a set of range search problems. There exists a constant D and two collections, collection of dynamic points R (stands for robots) and collection of search ranges Q (sensing areas of obstacles). For any points P in R, we need to find all the points in R that are within the distance D to P. We also need to find all the ranges in Q that contain P. Positions of points in R and searching ranges in Q may be changing over time, and the results are calculated every iteration.

For a naive implementation, we calculate the distances of every two robots to see if they are within the sensing range. The computation complexity is $O(n^2)$, where n is population size. The time is quite short when n is small, but it becomes intolerantly large when n grows which is just the case in swarm robotics, as the population size should be at least tens or hundreds. So a smarter method should be introduced.

1.2 K-D Tree

K-d tree (shorted for k-dimensional tree), proposed by Bentley at 1975 [5], is a space-partitioning data structure for organizing points in a k-dimensional space. K-d tree is a binary tree of k-dimensional points. Every non-leaf node can be thought as a splitting hyper plane that divides the space into two half-spaces at a specified dimension. For example, if x axis is chosen for a node, points with a smaller x values than the node appear in the left sub tree and points with larger x values are in the right sub tree.

K-d tree is a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches) [6], calculating multi-scale entropy [7] or triangle culling for ray-triangle intersection tests in ray tracing [8]. So far, researches using k-d trees has concentrated on traversing them quickly, as well as on building them to be efficient for general applications [9], but usages for special applications are not focused.

2 The Indexed K-D Tree

Range searches in the neighborhood generation problem can be clustered into two types: search for ranges centered at robots or obstacles. Since the search ranges of obstacles may change or obstacles may be added into or removed from the environment, it's hard to build an optimized space indexing data structure for obstacle range searches. Thus, we focus on optimizing range searches for generating robots' neighborhoods while the detection of obstacles remains the same as the normal k-d tree in the proposed indexed k-d tree.

2.1 Motivation

We denote the results of all neighborhoods of the swarm as a matrix N, where $N(i, j)$ indicates robot j is a neighbor of robot i. Since the sensing ranges of each robot are the same, the matrix is symmetric. After searching a node for its neighbors, we can also determine whether it's the neighbor of other robots in the same time. Thus, this node can be removed from tree to shorten the searching time for the remaining nodes. However, in a normal k-d tree, removal of a node will take $O(logn)$ time, where n is size of the tree. In the next iteration, the very node will be added back to the tree when re-building the entire tree, which will take another $O(logn)$ time. We have to apply these remove and insert operations for $n - 1$ points each iteration. With such strategy, we spend more time for searching although we originally purpose is to save time.

We propose an indexed k-d tree to take full advantage of this strategy and guarantee the strategy really saves time in the searching process. Each node in the tree is assigned a zero-based index, ranges from 0 to $n - 1$ and distinct to each other. It should be noted that a index is assigned to a node in the tree structure rather than the specific robot whose position is the value of the node. The tree is re-arranged every iteration since the robots are moving. The index of the tree node stays the same while it may refer to different robots in different iterations. The easiest way of implementing such method in the computer program is to store all the nodes in a array and index in the array indicates the index of the node.

To simplify the removal operation, we do not really remove the node from the tree, but we ignore all the nodes that have been removed. The nodes should be removed (or ignored) carefully in a certain order that can benefit searching for the rest nodes. The ignored nodes should be distinguished easily by its index during the search, i.e. no longer than $O(1)$ time. All the children nodes of a tree branch must be searched before we can ignore the root node of entire tree branch safely. In our indexed k-d tree, indexes of nodes are assigned following a simple principle:

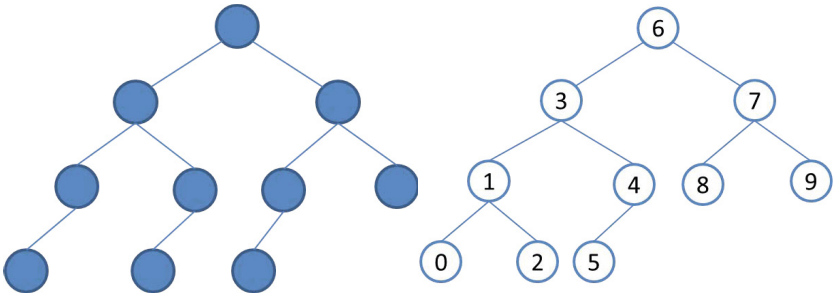
Tree Indexing Principle. $\forall i \in [0, n-1)$, nodes indexed $0, \ldots, i$ are connected and the sub tree formed by these nodes is a valid k-d tree with the height of not more than $log(i) + 1$.

With this simple principle, we can search and remove the indexed nodes in the descending order. We can ignore all the nodes with larger indexes than that

of the center of current search range, when we apply range searches on the rest of the nodes, which still form a valid k-d tree according to the principle. The restriction of the height of the sub tree is to optimize the indexing so that the sub tree is more balanced and saves searching time.

2.2 The Indexed Structure

The structure of the proposed indexed k-d tree is shown in Figure 1. We can easily see from the figure that, unlike normal k-d tree, the tree is unbalanced as nodes in the last layer are left-aligned. Since the searching time is more related with the height of the tree rather than size, we should always try to keep the remaining sub trees lower. As we remove the nodes with larger indexes first when searching, the height of sub trees of the unbalanced tree will descend more quickly than a normal tree.



(a) Normal k-d tree (b) Indexed k-d tree

Fig. 1. Comparison between normal and indexed k-d tree

The tree structure is initialized before the simulation starts according to the population size of the swarm. The nodes will fill all the layers of the tree from top to bottom and fill a layer from left to right. After that, tree nodes are indexed according to the following three rules.

Tree Indexing Rule 1. The indexes of a node and all its sub-nodes are continuous.

When building and updating the tree, we need to split the positions of the robots at certain dimension in k-d tree. We store all these values in an array. Continuous indexes can benefit the split process for easier program readability and faster execution time since reading continuous memory blocks is quicker than sparse ones.

Tree Indexing Rule 2. The indexes of nodes in the right sub tree of node i are always larger than i and nodes in the left sub tree.

A larger index indicating the node will be removed earlier in the search process. Since the tree is unbalanced and the right side is usually smaller than the left side, nodes in the right sub tree should have larger indexes.

Tree Indexing Rule 3. The indexes of nodes in the left sub tree of node i is smaller than i if node i is the first node of that layer, and vice versa.

As for indexing the left sub tree, there exist two situations. First nodes of the layer (indexed 6, 3, 1 or 0 in Figure 1b) should have larger indexes than nodes in their left sub trees, since they are currently the root of the current search tree with an empty right sub tree. In the next step of search process, the root node is "removed" and the height of the tree will reduce by one. For other nodes, their left sub trees should be "removed" earlier than themselves, so they have smaller indexes than nodes in left sub trees.

3 Neighborhood Generation Using the Indexed K-D Tree

When using the indexed k-d tree in simulations, we first initialize the tree structure before simulation starts using the rules proposed in the previous section. Every iteration in the simulation, we first update the values in the tree nodes for splitting the space and then apply range searches for neighborhood generation.

The update operation works similar with the construction of the tree. In a normal construction of k-d tree, a split operation is applied for each node. Nodes in the left and right sub trees of one node are split by the node itself at specified dimension. The complete operation has a computational complexity of $O(nlogn)$ where n is size of the tree. However, the robots have a limited maximum moving speed, which is normally small compared to the sensing range. So we assume that the places of the nodes in the tree remain almost the same in two consequent iterations. Therefore, we always try to split the updated sub trees using the same old node that split the values in previous iteration. The complete construction will take $O(n)$ time in the best condition. The average construction time is $O(nlogn)$ and is guaranteed to be not more than that of a normal tree.

After updating the tree, we generate the neighborhoods for all robots. With each tree node indexed, searching becomes simple. We traverse the node index i from $n-1$ downwards to 0. For each index i, we search for neighbors of node i in the tree formed only by the nodes indexed from 0 to $i-1$ which is a valid indexed k-d tree according to the principle. Since the result matrix N is symmetric, we assign all $N(j,i) = N(i,j)$ where $j < i$. We also assign $N(i,i)$ to be true or false according to the requirements of the application. After assigning all the values related with robot i in matrix N, we step to next index $i-1$ until i reaches 0. From Figure 1b, we can see that the root of the tree changes during the search and the tree gradually shrinks to the left corner.

If obstacles are involved in the simulation, we search the neighbors of these obstacles in the complete tree just like normal range searching in a normal k-d tree.

The pseudo code of neighborhood generation using the indexed k-d tree is shown in Algorithm 1.

Algorithm 1. Code for Neighborhood Generation at Every Iteration using Indexed K-D Tree

```
Update tree, trying to use the old split nodes
for i = n − 1 to 0 do                    //Robots' neighborhoods
   Set up search range R centered at node i
   Search the tree formed by nodes [0,i-1) using range R
end for
for all obstacle o do                    //Obstacles' neighborhoods
   Set up search range R centered at obstacle o
   Search the entire tree using range R
end for
Return neighborhoods
```

4 Results and Discussions

In this section, experimental results and discussions are presented. We first compare the efficiency of the improved k-d tree with other implementations in different population sizes, and then compare the performance under obstacle environments.

To test the efficiency of neighborhood generation for different methods, we run tests on our self-built simulation platform. The robots wander in the environment with randomly generated directions. They bounce at the borders and have a rate of 5% to change their direction with randomly generated ones. All our experiments are repeated 10 times, each has 10,000 iterations. The time in our results are the total time used for generating neighborhoods for all these iterations. The time that spends on initializing and calculating robots' movements is not included.

4.1 Time Comparison among Different Population Sizes

We compare the efficiency of our indexed k-d tree with normal k-d tree and the naive implementation under different population sizes in this section. The results are shown in Table 1. In the table, naive stands for the simple implementation that calculates all the distances of every two robots, normal stands for the normal k-d tree and indexed stands for our proposed k-d tree in this paper. The normal k-d tree use the same updating strategy as our indexed k-d tree and searches the entire tree for every robot, which is faster than removing a node and adding it back using the normal way. The running times in milliseconds stands for the time for all 100,000 iterations (10,000 iterations each for 10 times).

From the table, we can see that as population grows, time for indexed k-d tree grows slower than naive, as the computational complexity tells. Our indexed k-d tree speeds less time than naive when population becomes more than 13. As population increases, we can see that indexed k-d tree becomes 20 to 30 percent quicker than naive when population is fewer than 30. When population grows to 100 and 1000, this percentage becomes even larger to 60 and even 75. We

Table 1. Time Comparison among Different Populations (Lower is better)

Population	Running Time (ms)			Time Ratio	
	Naive	K-D Tree		Indexed /	Indexed /
		Normal	Indexed	Naive	Normal
2	**17.91**	52.38	41.56	232.05%	79.34%
3	**34.32**	98.05	68.28	198.97%	69.64%
4	**57.48**	166.79	115.92	201.66%	69.50%
5	**87.08**	239.69	153.04	175.74%	63.85%
6	**125.34**	360.77	264.63	211.14%	73.35%
7	**169.88**	384.96	270.77	159.39%	70.34%
8	**217.81**	480.22	328.61	150.87%	68.43%
9	**283.61**	633.33	375.43	132.38%	59.28%
10	**337.62**	632.06	425.74	126.10%	67.36%
11	**416.04**	838.45	502.89	120.88%	59.98%
12	**490.28**	905.74	610.45	124.51%	67.40%
13	**575.31**	1040.42	624.12	108.48%	59.99%
14	668.96	1046.03	**645.76**	96.53%	61.73%
15	768.43	1254.40	**715.47**	93.11%	57.04%
16	859.12	1456.15	**781.45**	90.96%	53.67%
17	969.94	1666.82	**928.95**	95.77%	55.73%
18	1091.89	1692.73	**977.72**	89.54%	57.76%
19	1221.11	1880.05	**985.22**	80.68%	52.40%
20	1360.52	2001.45	**1145.62**	84.20%	57.24%
21	1474.70	1971.13	**1246.12**	84.50%	63.22%
22	1632.14	2172.43	**1343.55**	82.32%	61.85%
23	1789.37	2197.13	**1347.18**	75.29%	61.32%
24	1924.72	2444.18	**1532.71**	79.63%	62.71%
25	2132.77	2860.92	**1605.33**	75.27%	56.11%
26	2295.74	2755.71	**1614.63**	70.33%	58.59%
27	2467.44	3045.49	**1925.85**	78.05%	63.24%
28	2632.62	3031.58	**1962.45**	74.54%	64.73%
29	2870.51	3334.43	**2036.15**	70.93%	61.06%
30	3036.72	3472.01	**2104.65**	69.31%	60.62%
100	32,774	19,775	**12,946**	39.50%	65.47%
1000	3,520,591	1,148,533	**862,571**	24.50%	75.10%

can say, according to the results, the indexed k-d tree over-performs the naive implementation even when population is small, not to mention the advances when population size blows.

We can also see that our indexed k-d tree is 40% quicker than normal k-d tree when population is larger than 9. We can see the advantage of indexed k-d tree drops a little when population is very large. The reason for such situation is that updating time of the tree increases quickly than the searching time. As for normal k-d tree, we can see that even when population is 30, it's still slower than the naive implementation. Actually, in our experiment, it won't be faster than naive until population reaches 35.

In summary, our indexed k-d tree is 40% quicker than normal k-d tree and at least 20-30% quicker than the naive implementation when population is more than 10. The advantage becomes more notable when population grows. We can defer that our indexed k-d tree is applicable in real-time simulations, especially for swarm robotics applications which normally have a population size of at least 10.

4.2 Time Comparison under Obstacle Situation

In obstructive environments, calculation of the neighborhoods of obstacles is also involved as we mentioned in previous sections. In a swarm robotic simulation, if obstacles are introduced, the number of obstacles can vary from a few to hundreds. In this section, we compare the performance of algorithms under environments with 10-50 obstacles since the trends are displayed completely

Table 2. Running Time Ratio in Obstacle Range Searches (Lower is better)

Population	Number of Obstacles 10	20	30	40	50
2	226.41%	217.73%	188.16%	213.31%	219.46%
3	171.87%	152.51%	162.49%	164.14%	164.37%
4	167.75%	142.51%	145.66%	143.43%	136.36%
5	138.93%	123.78%	123.29%	121.46%	122.81%
6	125.93%	117.00%	111.10%	108.74%	105.85%
7	106.89%	107.05%	103.89%	97.81%	97.81%
8	107.50%	100.14%	92.45%	93.75%	97.75%
9	97.78%	92.98%	95.99%	90.68%	86.43%
10	97.42%	86.80%	88.04%	86.08%	79.39%
11	94.91%	85.88%	81.64%	78.10%	76.23%
12	86.09%	81.33%	77.98%	78.10%	75.26%
13	81.66%	77.44%	73.96%	71.87%	72.49%
14	83.94%	71.90%	71.43%	69.54%	69.55%
15	80.64%	71.42%	69.46%	67.02%	68.03%
16	78.30%	70.90%	65.33%	65.03%	64.35%
17	85.30%	70.23%	65.09%	64.17%	63.53%
18	75.45%	66.59%	67.01%	63.02%	61.64%
19	68.46%	68.44%	63.01%	61.26%	57.20%
20	68.17%	63.48%	63.88%	59.92%	58.05%
21	66.97%	65.14%	58.36%	58.33%	58.07%
22	67.02%	61.49%	57.44%	55.48%	54.69%
23	67.16%	63.74%	59.85%	56.51%	54.20%
24	64.98%	64.17%	57.00%	56.66%	53.84%
25	63.88%	61.49%	56.44%	55.14%	52.76%
26	63.69%	57.10%	55.23%	54.14%	50.73%
27	64.77%	58.33%	53.90%	51.37%	50.49%
28	60.12%	56.90%	53.81%	52.45%	48.89%
29	62.14%	56.54%	50.08%	50.14%	52.67%
30	65.10%	58.79%	52.99%	48.59%	46.05%

with these results. From the previous section, we can see our indexed k-d tree is always quicker than normal k-d tree. Since these two k-d trees use the same strategy for searching obstacles, we only compare the result between indexed k-d tree and naive implementation. The time ratios (indexed / naive) under different populations and numbers of obstacles are shown in Table 2.

From the table, we can see that the two algorithms have the same trend as the situation without obstacles when population increases. As number of obstacles increases, time ratio is decreasing rapidly and our indexed k-d tree has a larger time advantage. The populations, at which indexed k-d tree becomes quicker than naive implementation, decrease to 7-9 due to different number of obstacles, compared with 13 when no obstacles are involved. As population increases, the time ratio decreases to 50-60 which means indexed k-d tree spends almost only half the time than the naive implementation. The results demonstrate that our indexed k-d tree has a greater advantage than in no obstacle environments and it's more applicable in simulations since numbers of obstacles can be tens or hundreds in most swarm robotic simulations.

5 Conclusion

In this paper, we consider the problem of neighborhood generation in swarm robotic simulations. The problem can be converted into a series of range search problems. Based on the characteristics of this problem, we proposed an indexed k-d tree, which provides a structure for quickly searching a set of ranges centered at the points which formed the k-d tree. The simulation results show that our indexed k-d tree is almost 40% quicker for calculating robots' neighborhoods than normal k-d tree. It is also significantly quicker than the naive implementation in most population sizes and numbers of obstacles for neighborhood generation.

Due to the tree structure and node indexes, the indexed k-d tree is 30-40% faster than traditional methods for calculating neighborhoods of all the robots in the swarm, which is the most time consuming operation of neighborhood generation problem in swarm robotics simulations. This result demonstrates that the indexed k-d tree can accelerate the process of generating neighborhoods significantly and is very applicable for swarm robotics simulations.

Acknowledgements. This work is supported by the National Natural Science Foundation of China under grants number 61170057 and 60875080. Professor Tan Ying is the corresponding author.

References

1. Shi, Z., Tu, J., Zhang, Q., Liu, L., Wei, J.: A survey of swarm robotics system. In: Tan, Y., Shi, Y., Ji, Z. (eds.) ICSI 2012, Part I. LNCS, vol. 7331, pp. 564–572. Springer, Heidelberg (2012)
2. Michel, O.: Webotstm: Professional mobile robot simulation. arXiv preprint cs/0412052 (2004)

3. Vaughan, R.: Massively multi-robot simulation in stage. Swarm Intelligence 2(2), 189–208 (2008)
4. Pinciroli, C.: The swarmanoid simulator. Technical report. Citeseer (2007)
5. Bentley, J.: Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)
6. Arroyuelo, D., Claude, F., Dorrigiv, R., Durocher, S., He, M., López-Ortiz, A., Ian Munro, J., Nicholson, P., Salinger, A., Skala, M.: Untangled monotonic chains and adaptive range search. Theoretical Computer Science 412(32), 4200–4211 (2011)
7. Pan, Y., Lin, W., Wang, Y., Lee, K.: Computing multiscale entropy with orthogonal range search. Journal of Marine Science and Technology 19(1), 107–113 (2011)
8. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time kd-tree construction on graphics hardware. ACM Transactions on Graphics (TOG) 27, 126 (2008)
9. Wald, I., Havran, V.: On building fast kd-trees for ray tracing, and on doing that in o (n log n). In: IEEE Symposium on Interactive Ray Tracing 2006, pp. 61–69. IEEE (2006)