Problem

My research area is about the urban modeling and visualization, and in the current research project, I am proposing a style-based road designing tool. This tool uses a rule-based road generation engine, by which a road network starts growing from one or multiple initial seeds based on the rules. While the roads grow, an undesired road crossing can often occur, so the road generation engine has to check the crossing every time when a new road segment is added to the road network. The naïve solution to check the crossing is to check the intersection of the newly added line segment against all the existing line segments in the road network, which takes $O(n)$ time, where n is the number of the existing line segments. Thus, the total computation time for the checks until the road generation completes is $O(n^2)$, which is undesirable. The plane sweep algorithm is a well-known algorithm which can find all the intersections of n given line segments in $O\big((n+k)\log n\big)$ time, where k is the number of the intersection points. This algorithm is useful if we deal with only the static set of line segments, but for the aforementioned case, this approach takes $O(n(n+k)\log n)$, which is worse than the naïve approach.

In this final project, I propose kd-tree data structure for line segments to efficiently detect the line segments intersection for a given line segment against the other n line segments in $O(\log n)$ time in the practical cases. My cost function finds the local optimal separating plane for kd-tree construction, which improves the query performance.

Building Kd-tree for Line Segments

Kd-tree is a space partitioning data structure for points, but it can also be used for lines with some modification [Fussell et al. 1988]. Given n line segments over the two dimensional space, a kd-tree is built by the recursive scheme as shown in Fig. 1.

```
function TreeNode::insert(LineSegment L)
if L is completely on the left side of this node's plane p then
   insertLeft(L)
if L is completely on the right side of this node's plane p then
   insertRight(L)
if L crosses this node's plane p then
   (L_L, L_R) = split(L, p)
   insertLeft(L_L)
   insertRight(L_R)

function TreeNode::insertLeft(LineSegment L)
if T_L exists then
   T_L->insert(L)
else
   addLeafNode(L)
```

```
function TreeNode::insertRight(LineSegment L)
if T_R exists then
   T_R->insert(L)
else
   addLeafNode(L)
```

Figure 1. The pseudo code to build a kd-tree for line segments

The major difference of the kd-tree for line segments compared to the one for points is that it has to deal with a case when a line segment crosses a separating plane. To deal with this case, we have to split a line segment, and traverse the both child nodes. In the worst case, the line segment crosses most of the separating planes, and we have to traverse almost all the tree nodes. The naïve solution would be to use a spatial median splitting, in which the dimension is chosen in round robin fashion, and the plane is positioned at the spatial median of the space. This approach is simple, but it does not avoid the line splitting at all. To address this issue, my approach uses a cost function which estimates the traversing cost and the intersection computation cost inspired by the SAH based approach [WALD et al. 2006]. The cost of adding a separating plane p is defined as follows:

$$C(p) = K_T + K_T(P_L \log|T_L| + P_R \log|T_R|),$$

where K_T is a cost for a node traversal, P_L is the probability to traverse to the left child node, and P_R is the probability to traverse to the right child node. For each subdivision of a space, the local optimal plane which minimizes the above cost function is chosen.

Tree Traversal for Intersection Detection

Given a line segment l, my kd-tree data structure can efficiently find the intersections. It starts with the root node of the kd-tree by checking whether l intersects the line segment of the root node. If it intersects, then we are done. Otherwise, we check if l lies entirely on one side of the separating plane, or crosses the plane. For the former case, we go to either left or right child node which completely contains l, and recursively check the same things. For the latter case, we split l by the separating plane, and go to both left and right child node for the recursive checks. The pseud code of the traversal algorithm is as shown in Fig. 2.

```
function TreeNode::intersects(LineSegment L)
if this node's line segment intersects L then
   return true
if L is completely on the left side of this node's plane p then
   if T_L exists then
      return T_L->intersects(L)
   else
      return false
if L is completely on the right side of this node's plane p then
   if T_R exists then
      return T_R->intersects(L)
```

```
    else
        return false
if L crosses this node's plane p then
    (L_L, L_R) = split(L, p)
    if T_L exists then
        if T_L->intersects(L_L) then
            return true
    if T_R exists then
        if T_R->intersects(L_R) then
            return true
    return false
```

Figure 2. The pseudo code to detect the intersection for a given line segment

Analysis

During the tree construction, the main task is to find the best separating plane. If we order the line segments every time, the recursion equation will be as follows:

$$T(n) = n \log n + 2T\left(\frac{n}{2}\right).$$

Thus, $T(n)$=O$(n \log^2 n)$. However, this can be improved by ordering all the line segments by both axes beforehand. Then, the recursion equation becomes as follows:

$$T(n) = n + 2T\left(\frac{n}{2}\right).$$

Thus, $T(n)$=O$(n \log n)$. Here, I ignored an important thing. While building a kd-tree, some line segments may get split by a separating plane. This causes the increase in the number of the tree nodes and it leads to the increase in the height of the tree. Nevertheless, this is still the case that the upper bound of the computation is O$(n \log n)$ in the practical cases for the following reason.

Let p be the probability that a line segment intersects a separating plane. For each node, we traverse two children with probability p, and one child node with probability $1 - p$. Thus, the excepted number of children we traverse is $2p + 1 - p = p + 1$. Then, the total number of traversed nodes k is

$$k = \sum_{i=0}^{h-1} (p+1)^i,$$

where h is the height of the tree. Thus,

$$k = \frac{(p+1)^h - 1}{p} \text{ (for } p > 0), \qquad k = h \text{ (for } p = 0).$$

Suppose we have a1000 units × 1000 units of two-dimensional space and each line segment has at most 10 units long. Then, the probability p is at most $\frac{20}{1000} \times \frac{20}{1000} = 4 \times 10^{-4}$. If we have 10,000 line segments in this space, which is relatively dense, then the number of tree nodes will be $10000 + 4 \times 10^{-4} \times 10000 \times 10000 = 50000$. Thus, the tree height h will be around 15.6, and the expected number of the traversed nodes will be around 15.6. On the other hand, in the ideal case, which is $\log 10000 \approx 13.3$, the expected number of the traversed nodes is around 13.3. This difference will even be reduced if we have more sparse line segments. This implies that the computation time of tree construction is $O(n \log n)$ in the practical cases.

The intersection detection algorithm traverses the tree from the root to one of the leaf nodes or multiple leaf nodes if the line segment gets split. In the ideal case in which there is no split, the computation time is $O(h)$. Since the tree is balanced, the height of the tree is $O(\log n)$. Thus, the computation time is $O(\log n)$ in the ideal case. For the average computation time, we can use the same analysis that we discussed above. Thus, we can achieve $O(\log n)$ for the intersection detection in the practical cases as well.

Results

First, I compared the computation time of the kd-tree based and the $O(n^2)$ brute force approach of the intersection detection. I used a1000 units × 1000 units of two-dimensional space. Each line segment has at most 10 units long, and is distributed randomly over the space. I ran the intersection detection 1000 times to get the average computation time. The result is shown in Fig. 3.

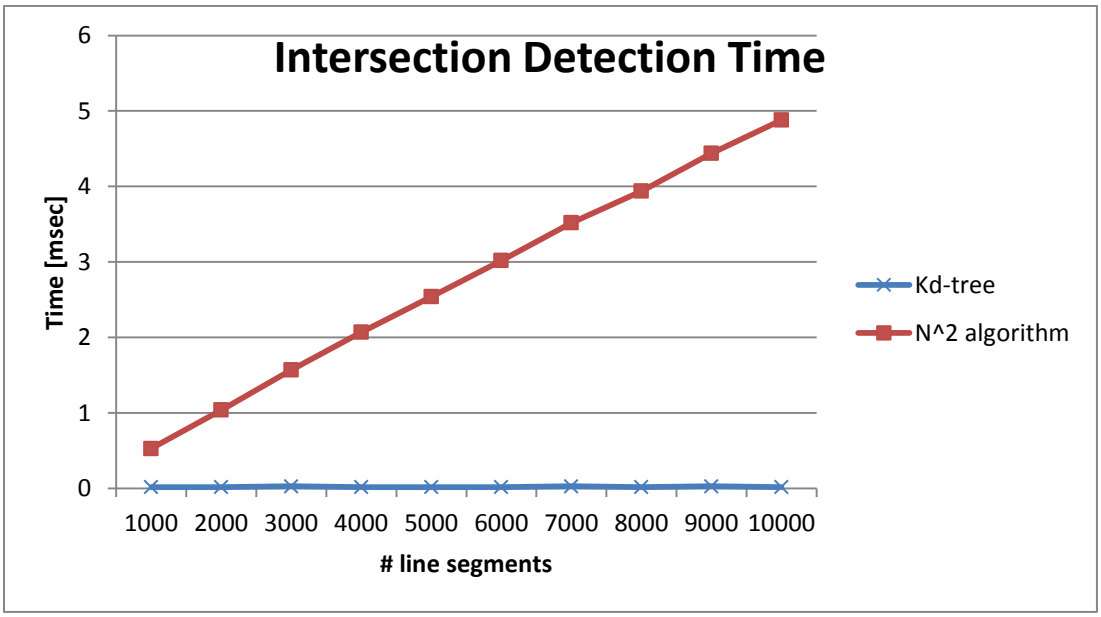

Figure 3. The comparison of the query time between the kd-tree and N^2 algorithm.

Second, I compared the computation time of the different algorithm to position a separating plane in the kd-tree construction, the cost function based and the spatial median splitting. The result is shown in Fig. 4.

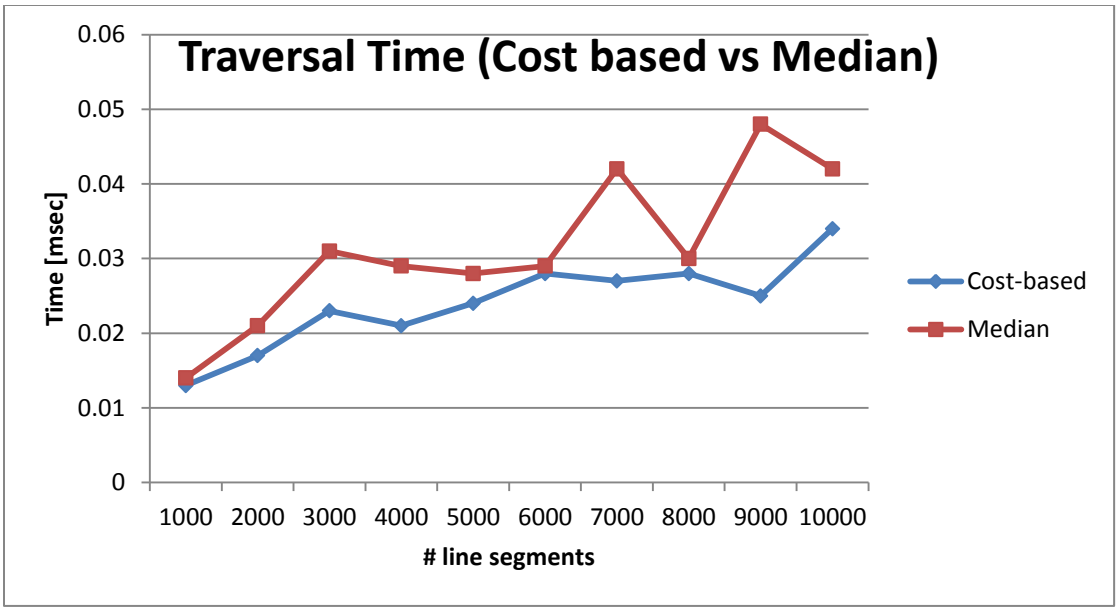

Figure 4. The comparison of the traversal time of the different algorithm to position a separating plane in the kd-tree construction, the cost function based and the spatial median splitting.

My proposed algorithm achieved average 20% increase in performance compared to the spatial median splitting.

Conclusions

I proposed a kd-tree structure for line segments, and explained how to build it and how to use it to query an intersection for a give line segment. The analysis showed that my algorithm achieved $O(n \log n)$ for the tree construction and $O(\log n)$ for the tree traversal in the practical cases. The results showed that my algorithm achieved better performance than the naïve spatial median splitting by average 20%.

References:

FUSSELL, D. AND SUBRAMANIAN, K. R. 1988. Fast Ray Tracing Using K-D Trees. Technical report.

WALD, I. AND HARVAN, V. 2006. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE symposium on interactive ray tracing.* p. 61-69.