Report for Fieldtree Implementation

Pouya Moetakef

Implementation of both Partition Fieldtree and Cover (Loose) Fieldtree were done in Microsoft Visual C#. The implementation includes RectangleObj to handle operations for rectangular shapes, generic node implementation which can be either Partition or Cover, and generic Fieldtree structure. The GUI is created with Windows Forms and implemented operations includes add, remove, move, point query (nearest objects to a point), range query (objects contained in a given circle), window query (objects contained in a given window), and incremental nearest neighbor. The implementation details are explained in following sections.

RectangleObj:
This object presents a rectangle object and rectangle operations. The structure saves, minimum and maximum extents in X and Y directions as well as center point, and width and height for faster access. The object can be created either by providing center point and its width and height, or by providing top-left and bottom-right corners. The minimum and maximum extents will be populated correctly even if the top-left and bottom-right corners are mixed up upon insertion.
The object further supports rectangle operations such as equality (if two rectangles have the same coordinates and size), containment checks (rectangle contained by another rectangle or circle, rectangle contains another rectangle, or rectangle contains a point), and intersection checks (rectangle intersects with another rectangle or circle). Furthermore, the objects supports calculation of shortest distance-squared from a given point.

FieldNode:
A field node is a generic type of any node that can support node operations. The generic field node contains a RectangleObj which determines its bounding box, layer number which determines how deep in the tree this node is, capacity which determines how many objects can be stored in a node, and a place to store object references. Two nodes are considered being equal if their bounding box and layer numbers are equal. A field node can support general operations such as determining intersection or containment of other objects (here RectangleObj and points) for query uses, reporting stored objects and its capacity, storing objects, removing objects, reporting its children and parent(s), making or merging children, and determining descent path through its children.

CoverFieldNode:
Cover field node used for Cover (Loose) Fieldtree extends from FieldNode. Cover node requires a p-value ($0 < p < 1$) upon creation which is used to extend the bounds by $(1 + p)$. Therefore, the

bounds setup up by FieldNode is used to store it's expanded bounds, while a second original_bounds is implemented to store the non-expanded bounds used only for partitioning and making children.
Children are stored in an array with codes as 0 = NW, 1 = NE, 2, SW, and 3 = SE. To create the four children, the original_bounds is split into four equal quadrants and the four children are coded according to their position upon creation.
Each cover field node has a single parent (except the root which has no parent), and therefore, since a child is not shared with any other parent, upon merging no extra check is required and all children can be removed instantly (which is a different behavior than Partition Field Node which will be discussed later).

PartitionFieldNode:
Partition field node used for Partition Fieldtree extends from FieldNode. The main challenge in Partition Field Node is the arrangement of children. Each partition field node parent has 9 children, with only the center being unique that is not shared between parents. Therefore, the center child has 1 parent, the N, E, W, and S children each has 2 parents, and the NE, NW, SE, and SW children each has 4 parents. Table 1 shows the the code assignment to each child.

0	1	2
3	4	5
6	7	8

Table 1. Code assignment for children in Partition Field Node.

In Partition Field Node, each layer is shifted by half the size of its field size. Therefore, to create children, each child would have (parent_size / 2) in width and height (size). The origin of the each field is hence shifted by (parent_size / 4). After computing child bounds, each new child is assigned a code based on its position as presented in table 1. It must be noted, out of 9 possible children, only the children that do not already exist in the node will be created. The new children are then assigned to a list and this list is used to look for other parents. To do this a sibling lookup method is implemented as following:

```
SiblingLookup:
Siblings = empty
If current_node is root_node then
     Report Siblings since root_node has no siblings
End if
Foreach parent in parents do
     Children_of_parent = parent.GetAllChildren()
```

```
      Foreach child in Children_of_parent do
            If child is equal to current_node
                  Comment: We have found which child of that parent
                  current_node is

                  Based on table 1 lookup possible siblings and add
                  them to Siblings While changing the code to a
                  representation if current_node was at center

                  Break from inner for loop
            End if
      End for
End for
Report Siblings
```

After finding all the siblings we connect all the new children to the existing parents.

```
ConnectSharedChildren (new_children):
Siblings = SiblingLookup()
Foreach child in new_children do
      Potential_siblings = find all the potential siblings that can
      share the given child using table 1
      Foreach potential_sibling in Potential_siblings do
            If potential_sibling exists in Siblings

                  Add child to their children list while looking up
                  child code from table 1

                  Update and add potential_sibling to child's
                  parent_list
            End if
      End for
End for
```

To merge children together, the requirement is that all children to be either empty (no stored objects and no children) or non-existent. Similar to children creation in merging all nodes are deleted one by one, and all links to all parents will be removed prior to delete. This is reverse of ConnectSharedChildren method, and since they are very similar, the merge algorithm is not presented in here.

Fieldtree Structure:

Fieldtree structure accepts a FieldNode upon creation which can be either a CoverFieldNode or a PartitionFieldNode. Therefore, to create the structure, first a root node of either type needs to be created, and this node can be set as a root node of the Fieldtree structure. The rest of the operations are generic and follow the same algorithm, and therefore they are implemented in the generic Fieldtree structure. The tree can be configured to either create all children of a layer at the same time upon overflow (known as create all) or create only children of the overflown field (known as greedy or lazy).

When an object is added to the tree, first the smallest field that can contain the object will be identified by descending the tree (called deepest_field). If deepest_field is too full to hold the object, the field will be partitioned only once, and then the object is added again and the tree will be reorganized. The algorithm of this part is presented below:

```
FindDeepestContainingField (object, starting_node)
If object does not intersect with starting_node
     Report null
End if
If object is contained in starting_node and (starting_node has no
children or starting_node.size / 2 < object.size)
     Report starting_node
End if
Descent_child = lookup child from table 1 considering object's center
point
If Descent_child is not null
     Deepest_node = FindDeepestContainingField (object,
Descent_child)
     If Deepest_node is not null
          Report Deepest_node
     End if
End if
If object is contained in starting_node
     Report starting_node
Endif
Report null


AddObject (object)
deepest_field = FindDeepestContainingField (object, starting_node =
root_node)
If deepest_field is full and has no children
     deepest_field.CreateChildren()
```

```
      AddObject (object)
      ReorganizeFields()
Else
      Store object in deepest_field
End if

ReorganizeFields()
Create a Queue
Queue.Enqueue (root_node)
While Queue is not empty do
      Node = Queue.Dequeue
      Foreach object in stored_list in Node
            deepest_field = FindDeepestContainingField (object,
            starting_node = Node)
            If deepest_field is not equal to Node
                  Remove the object from Node
                  Store object in deepest_node
            End if
      End for
      Foreach child in Node.GetChildren()
            Queue.Enqueue (child)
      End for
End while
```

To remove an object, first the object gets removed from the corresponding node, and then MergeChildren method is called on the node and its parents. It must be noted that in Fieldtree, nodes can be empty and only being used as guides to children. So it is possible to have a filled parent, empty child, and filled grandchild. Therefore, balancing tree (graph in case of partition fieldtree) is not important, and only fields with completely empty children will be checked for merging. The merge algorithm is as following:

```
MergeEmptyChildren (node)
If node is root or is not empty
      Quit
End if
If node has children and the children are empty
      node.MergeChildren()
End if
Foreach parent in parent_list do
      MergeEmptyChildren (parent)
End for
```

Implementation of point query and range query are very similar to the find operation so they are skipped here. For the move operation, when move is initiated the object and it's containing node are memorized. Upon each call, the object is deleted from the tree, the object location is updated, and then the object is inserted into the tree. So, upon each call, tree reorganization takes place, and other objects presented on the tree can descend if appropriate children get created, and upon moving the object away, they may not collapse (merge) due to the descent of other objects.

Incremental nearest neighbor search:

The next important implement is the incremental nearest neighbor search. The algorithm used here is the exact same algorithm presented in class, therefore the exact code from C# is presented as the following (with more comments added):

```
// Initialization
public void InitIncrementalNN(Point p)
{
      incrNN_queue = new SimplePriorityQueue<NodeOrObj>();
      NodeOrObj root_nodeOrObj = new NodeOrObj();
      root_nodeOrObj.SetNode(root);
      incrNN_queue.Enqueue(root_nodeOrObj, 0);
      incrNN_origin = p;
}

// Calling Find Next nearest neighbor
public List<RectangleObj> IncrementalNNFindNext()
{
      List<RectangleObj> answer = new List<RectangleObj>();
      // Starting off where the queue was last left off
      while (incrNN_queue.Count > 0)
      {
            NodeOrObj current_element = incrNN_queue.Dequeue();
            while (incrNN_queue.Count > 0 &&
            incrNN_queue.First().Equals(current_element))
            {
                  // Dequeue until there is no more of the same object left in
                  the priority queue
                incrNN_queue.Dequeue();
            }
            // If element is object return the answer
            if (current_element.IsObj())
            {
            answer.Insert(0, current_element.GetObj());
                  return answer;
            }
            // Otherwise, element type is a Node and we need to enqueue
            contents and children
```

```
            Else
            {
                  NodeType current_node = current_element.GetNode();
                  double current_dist =
                  current_node.GetBounds().GetDistanceSqToPoint(incrNN_origin);
                  // Enqueue stored objects with distance > current distance so
                  we can avoid enqueuing already enqueued objects
                  if (!current_node.IsEmpty())
                  {
                        foreach (RectangleObj obj in
                        current_node.GetAllStoredObjs())
                        {
                              double distance =
                              obj.GetDistanceSqToPoint(incrNN_origin);
                              if (distance >= current_dist)
                              {
                                    NodeOrObj obj_nodeOrObj = new NodeOrObj();
                                    obj_nodeOrObj.SetObj(obj);
                                    incrNN_queue.Enqueue(obj_nodeOrObj,
distance);
                              }
                        }
                  }
                  // Enqueue children with with distance > current distance so
                  we can avoid enqueuing already processed nodes
                  if (current_node.HasChildren())
                  {
                        foreach (KeyValuePair<int, NodeType> child_node in
                        current_node.getChildern())
                        {
                              double distance =
                              child_node.Value.GetBounds().GetDistanceSqToPoint
                              (incrNN_origin);
                              if (distance >= current_dist)
                              {
                                    NodeOrObj node_nodeOrObj = new NodeOrObj();
                                    node_nodeOrObj.SetNode(child_node.Value);
                                    incrNN_queue.Enqueue(node_nodeOrObj,distanc
                              e)
                              }
                        }
                   }
             }
        }
        return answer;
}
```

GUI operations:

GUI presents a canvas that can be set in Cover Fieldtree or Partition Fieldtree mode. The canvas size is best to set to power of 2 to avoid any complication from floating point operations. Upon clicking on “Set”, a new fieldtree structure with empty root node will be created. The bounds of each node is presented with gray or black lines. The line thickness reduces as node that are deeper in the tree and the color moves toward lighter gray. Objects (rectangles) can be added by left click and dragging the mouse on the canvas (note from action commands Add must be selected). To remove an object choose Remove from the actions and left click on the canvas. The nearest object to the cursor will be found and deleted. To move an object, select move from actions and left click to grab an object and drag to move the object. To help with visuals, when moving the containing node is highlighted by a hollow red rectangle.

Query operations include point query which finds nearest object(s) to cursor position. Range query finds all the objects contained in a circle while window query finds all the objects contained in a rectangle (window). They can be activated by selecting appropriate options from actions and left click and drag to draw the range (circle) or window (rectangle) for query operation.

The incremental nearest neighbor action requires initialization. For this action, first choose the Incremental NN from actions and then click on the canvas. An orange point is added in the place clicked indicating the center of the search query. Now the search is initialized, clicking on find next button, would find the next nearest neighbor. As for visual aid, circles representing radius of each search step are presented on the canvas. The search will be reset if another action is chosen or the query center is moved by clicking on the canvas.

Execution time comparison:

Figure 1 shows the execution time comparison for adding many rectangles (randomly generated using the same seed between 100 and 1,000,000). For cover fieldtree p-values from 0 to 1 with steps of 0.25 were chosen. As it can be seen, the difference between the curves are very small. Therefore, results suggests that for tree creation the overhead is negligible for having more complicated children generation method.

Figure 2 shows the execution time for point query operation on 100,000 randomly generated rectangles. Figure 2 also suggests that the difference between all different trees are very small.

Hence, it can be concluded that partition and cover field tree structure has similar execution time for adding objects and finding objects. Interesting future work can compare these results with other popular structures such as R-tree.

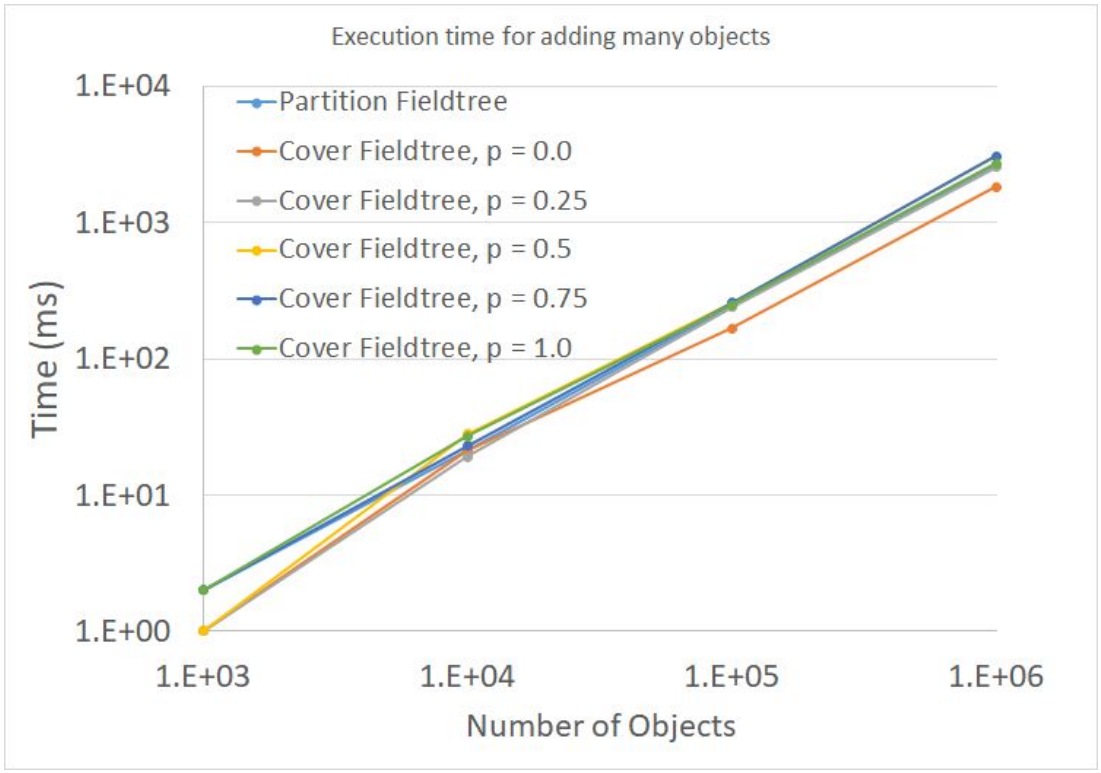

Fig. 1. Execution time for adding many rectangular objects (100 - 1,000,000)

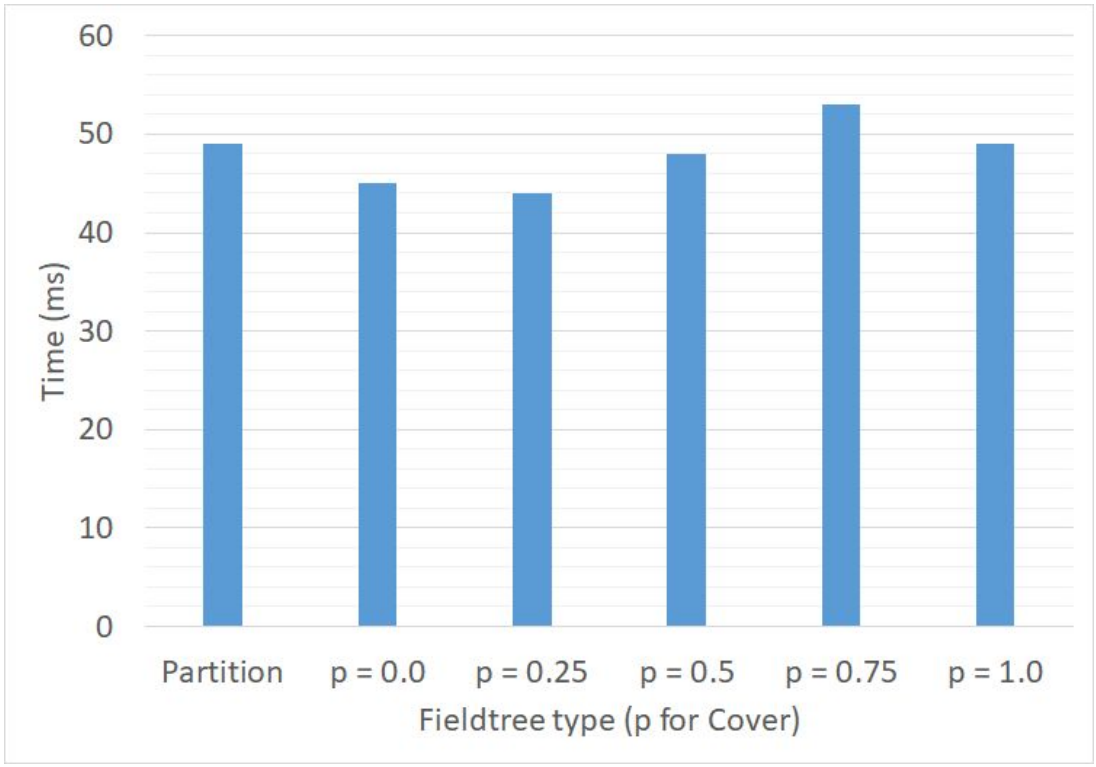

Fig. 2. Execution time for point query operation for structures with 100,000 objects.