CMPT 280

Tutorial: kd-trees

Mark G. Eramian

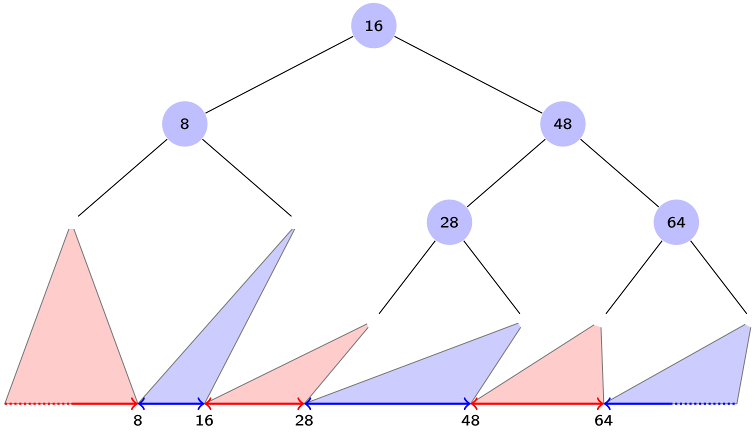
University of Saskatchewan

Multi-dimensional trees.

- kd-trees are essentially a generalization of ordered binary trees to higher dimensions.
- An ordered binary tree is a 1d-tree.
- If the elements are real numbers, it partitions a number line.
- Note that the elements stored in the tree, themselves, induce the partitioning.
- This makes it good for 1D range serches.

Multi-dimensional trees.

1d-tree partitions the real number line.

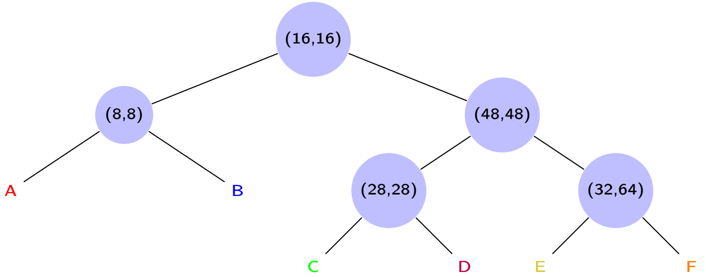


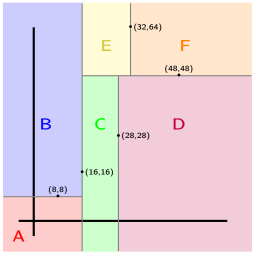
Multi-dimensional trees.

- A 2d-tree holds coordinate pairs. It partitions the cartesian plane, rather than a number line.
- Even numbered levels (including root) branch on x-coordinate.
- Odd numbered levels branch on y-coordinate.

Multi-dimensional trees.

2D Partitioning





Building a kd-tree From a Set of Points

- It is hard to insert points into a kd-tree individually.
- However, given a set of points all at once, it is comparably easy to build a tree containing that set of points.
- Algorithm is given in Question 1 of Assignment 7.
- Build algorithm relies on median-finding.
- Initial call: `kdtree(points, left=0, right=7, d=0)`

A Closer Look at kdtree

```
Algorithm kdtree (pointArray, left, right, int depth)
pointArray - array of k-dimensional points
left - offset of start of subarray from which to build a kd-tree
right - offset of end of subarray from which to build a kd-tree
depth - the current depth in the partially built tree - note that the root
        of a tree has depth 0 and the $k$ dimensions of the points
        are numbered 0 through k-1.

if pointArray is empty
    return nil;
else
    // Select axis based on depth so that axis cycles through all
    // valid values. (k is the dimensionality of the tree)
    d = depth mod k;
    medianOffset = (left+right)/2

    // Put the median element in the correct position
    jSmallest(pointArray, left, right, d, medianOffset)

    // Create node and construct subtrees
    node = a new id-tree node
    node.item = pointArray[medianPoint]
    node.leftChild = kdtree(pointArray, left, medianOffet-1, depth+1);
    node.rightChild = kdtree(pointArray medianOffset+1, right, depth+1);
    return node;
```

A Closer Look at jSmallest

```
Algorithm jSmallest(list, left, right, j)
    list - array of comparable elements
    left - offset of start of subarray for which we want the median element
    right - offset of end of subarray for which we want the median element
    j - we want to find the element that belongs at array index j
    To find the median of the subarray between array indices 'left' and 'right',
    pass in j = (right+left)/2.

    Precondition: left <= j <= right
    Precondition: all elements in 'list' are unique (things get messy otherwise!)
    Postcondition: the element x that belongs at index j if the subarray were
                   sorted is in position j.  Elements in the subarray
                   smaller than x are to the left of offset j and the
                   elements in the subarray larger than x are to the right
                   of offset j.

        [...]
```

A Closer Look at jSmallest

```

     [...]

  if( right > left )
     // Partition the subarray using the last element, list[right], as a pivot.
     // The index of the pivot after partitioning is returned.
     // This is exactly the same partition algorithm used by quicksort.
     pivotIndex := partition(list, left, right)

     // If the pivotIndex is equal to j, then we found the j-th smallest
     // element and it is in the right place!  Yay!

     // If the position j is smaller than the pivot index, we know that
     // the j-th smallest element must be between left, and pivotIndex-1, so
     // recursively look for the j-th smallest element in that subarray:
     if j < pivotIndex
             jSmallest(list, left, pivotIndex-1, j)

     // Otherwise, the position j must be larger than the pivotIndex,
     // so the j-th smallest element must be between pivotIndex+1 and right.
     else if j > pivotIndex
             jSmallest(list, pivotIndex+1, right, j)

     // Otherwise, the pivot ended up at list[j], and the pivot *is* the
     // j-th smallest element and we're done.
```

A Closer Look at partition

```
// Partition a subarray using its last element as a pivot.
Algorithm  partition(list, left, right)
list - array of comparable elements
left - lower limit on subarray to be partitioned
right - upper limit on subarray to be partitioned
Precondition: all elements in 'list' are unique (things get messy otherwise!)
Postcondition: all elements smaller than the pivot appear in the leftmost
               part of the subarray, then the pivot element, followed by
               the elements larger than the pivot.  There is no guarantee
               about the ordering of the elements before and after the
               pivot.
returns the offset at which the pivot element ended up


pivot = points[right]

swapOffset = left
for i = left to right-1
    if( points[i] <= pivot )
        swap points[i] and points[swapOffset]
        swapOffset = swapOffset + 1

swap points[right] and points[swapOffset]
return swapOffset;   // return the offset where the pivot ended up
```

Finding the median Illustrated

k-th smallest element algorithm

`Initial call: kdtree(points, left=0, right=7, d=0)`

0	1	2	3	4	5	6	7
(1,12,1)	(18,1,2)	(2,12,16)	(7,3,3)	(3,7,5)	(16,4,4)	(4,6,1)	(5,5,17)

Before `jSmallest(points, left=0, right=7, d=0, j=3)`

0	1	2	3	4	5	6	7
(1,12,1)	(2,12,16)	(3,7,5)	(4,6,1)	(5,5,17)	(16,4,4)	(7,3,3)	(18,1,2)

After partitioning.
The index `j` is smaller than the pivot index, so recurse left:

`jSmallest(points, left=0, right=pivotIndex-1=3, d=0, j=3)`

Finding the median Illustrated

k-th smallest element algorithm

0	1	2	3	4	5	6	7
(1,12,1)	(2,12,16)	(3,7,5)	(4,6,1)	(5,5,17)	(16,4,4)	(7,3,3)	(18,1,2)

Before `jSmallest(points, left=0, right=3, d=0, j=3)`

0	1	2	3	4	5	6	7
(1,12,1)	(2,12,16)	(3,7,5)	(4,6,1)	(5,5,17)	(16,4,4)	(7,3,3)	(18,1,2)

After partitioning (subarray already in order so no swaps occur).
The index `j` equals the current pivot index, which means the item at index `j` is the j-th smallest element, so we're done - the algorithm returns without recursing.

Building the kd-Tree from an array of points.

- Having found the median (wrt to dimension 0) this median becomes the root of our kd-tree.

- We then recursively find the left child of the root by calling `jSmallest()` on the subarray to the left of the median... and then recurse to find its children ... until we reach subarrays of length 1.
- Once that is complete, we recursively find the right child of the root by calling `jSmallest()` on the subarray to the right of the median, and then recurse to find it's children ...

Finding the left child of the root.

Recursively call `kdtree(points, left=0, right=j-1=2, d=(d+1)%3=1)`:

0	1	2	3	4	5	6	7
(1,12,1)	(2,12,16)	(3,7,5)	(4,6,1)	(5,5,17)	(16,4,4)	(7,3,3)	(18,1,2)

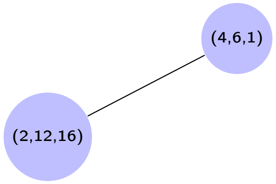
Before `jSmallest(points, left=0, right=2, d=1, j=(right+left)/2=1)`

0	1	2	3	4	5	6	7
(3,7,5)	(2,12,16)	(1,12,1)	(4,6,1)	(5,5,17)	(16,4,4)	(7,3,3)	(18,1,2)

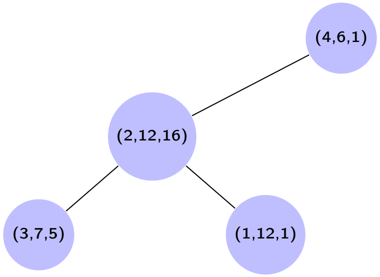
After `jSmallest(points, left=0, right=2, d=1, j=1)`
Median must be between 1 and 2, so recurse right:
`jSmallest(points, left=pivotIndex+1=1, right=2, d=1, j=1)`
The median is already in position so this will just return.

Median wrt to dimension 1 is now in position 1, so left child is (2,12,16).

Building the kd-Tree from an array of points.



Now need to recursively find the left and right children of (2,12,16). Since there is only one element on either side of (2,12,16) within the subarray points[0..2] these become the children (because they are the median of a sequence of length 1)

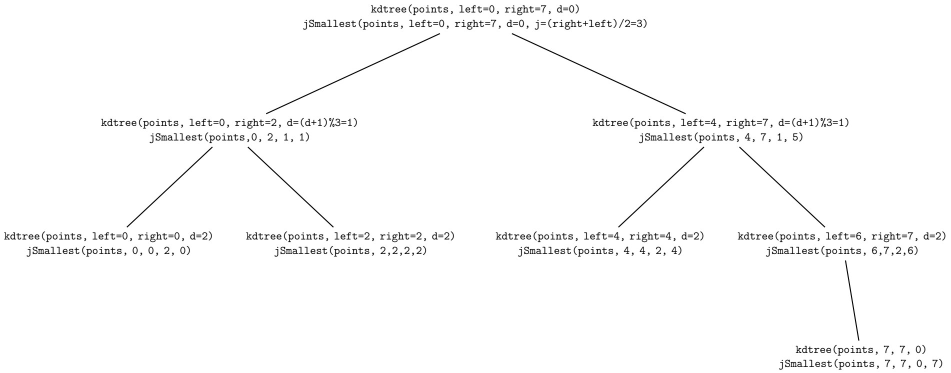


Building the kd-Tree from an array of points.

0	1	2	3	4	5	6	7
(3,7,5)	(2,12,16)	(1,12,1)	(4,6,1)	(5,5,17)	(16,4,4)	(18,1,2)	(7,3,3)

Excercise: Find the right subtree of the root. Draw the array before and after each call to jSmallest(), and determine the values of the parameters to each such call.

Recursion Tree for `kdtree`



```
kdtree(points, left=0, right=7, d=0)
jSmallest(points, left=0, right=7, d=0, j=(right+left)/2=3)

kdtree(points, left=0, right=2, d=(d+1)%3=1)
jSmallest(points,0, 2, 1, 1)

kdtree(points, left=4, right=7, d=(d+1)%3=1)
jSmallest(points, 4, 7, 1, 5)

kdtree(points, left=0, right=0, d=2)
jSmallest(points, 0, 0, 2, 0)

kdtree(points, left=2, right=2, d=2)
jSmallest(points, 2,2,2,2)

kdtree(points, left=4, right=4, d=2)
jSmallest(points, 4, 4, 2, 4)

kdtree(points, left=6, right=7, d=2)
jSmallest(points, 6,7,2,6)

kdtree(points, 7, 7, 0)
jSmallest(points, 7, 7, 0, 7)
```

Excercise: determine the contents of `points` before and after the completion of each node in the recursion tree.