A FORMALIZATION OF KD-TREES (WITH $k = 2$)

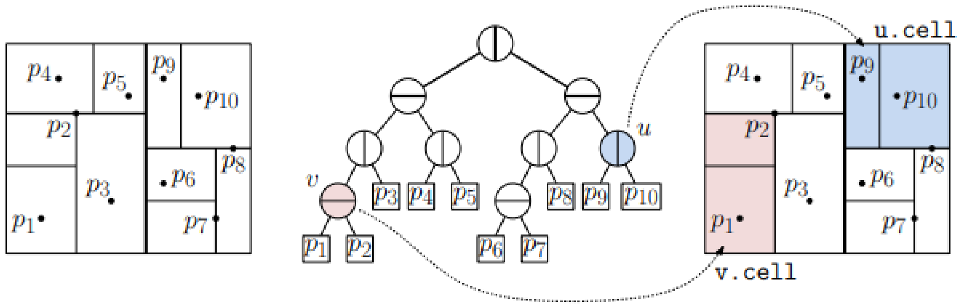
ADAM MELROD AND CHASE HUTTON

ABSTRACT. A report on our final project for CMSC631. In this project, we formalize a kd-tree, a space-partitioning geometric data structure, and prove various properties about them.

CONTENTS

1. WHAT IS A KD-TREE?

A kd-tree is an example of a partition tree. For each node of the tree we subdivide the ambient space by specifying a $(k-1)$-dimensional axis-aligned hyperplane to split the points about a dimension. This dimension is called the cutting dimension. The figure below gives an example of the two dimensional case:



The kd-tree in the figure is representative of what we wish to formalize in coq. This variation of the kd-tree concept is classified as a two dimensional point kd-tree. Observe in the figure that each split is determined by a vertical or horizontal line equation, i.e. $y = c$ or $x = c$. We denote the value of c as the cutting value. In what follows we will refer to the internal nodes of the tree as split nodes and the external nodes as point nodes.

We note that the two dimensional point kd-tree data structure is easy to implement and is quite practical. It is useful for many different types of searching problems including nearest neighbor searching and range querying.

1.1. **Formalization in Coq.**

To implement the kd-tree, we first need to describe what data a tree consists of. As discussed above, the split nodes of the tree will need to encode the a cutting dimension, a cutting value, and references to their two children. The point nodes of the tree will encode the data of point in two-dimensional space, which we will use the canonical rational library to do.

First, the cutting dimension should be either the along the X-axis or the Y-axis. We thus define the following type:

```
Inductive Dim : Type :=
| X
| Y.
```

We also define the type of a vector and a (possibly empty) pair of coordinates:

```
Definition Vector : Type := Qc*Qc.
Definition Coord : Type :=  option Vector.
```

Using the types we have defined, we define a tree:

```
Inductive tree : Type :=
| Point : Coord -> tree
| Split : Dim -> Qc -> tree -> tree -> tree.
```

The next issue we have to resolve is how we identify kd-trees. The method we decided to take was to treat being a kd-tree as a property of a tree with respect to a particular bounding box. This allows us to phrase the definition as an inductive property. To make precise the notion of a bounding box, we introduce a rectangle primitive which is described by it's bottom left point and upper right point:

```
Definition Rectangle : Type := Vector*Vector.
```

Now we make the following definition of the KDT property (we use some auxiliary properties that we define later):

```
Inductive KDT : Rectangle -> tree -> Prop :=
| KDT_Point_Empty : forall r, KDT r (Point None)
| KDT_Point_Coord : forall r v, Bounded r v -> KDT r (Point (Some v))
| KDT_Split : forall r1 r2 r3 d cv t1 t2,
    KDT r1 t1 -> KDT r2 t2 -> Aligned cv d r1 r2 ->
    Merged r1 r2 r3 -> KDT r3 (Split d cv t1 t2).
```

We now provide intuition for each of these cases. The first case makes it so that all empty trees are kd-trees with respect to any bounding box. The second condition asserts that a point is a kd-tree with respect to a bounding box only if the point is contained in that box. The third case asserts that a tree is a kd-tree with respect to a bounding box R only if the left and right subtree are kd-trees with respect to two bounding boxes that are aligned along the root node's cutting dimension at the cutting value and R is the union of the two bounding boxes. The definitions of the auxiliary properties are listed below but we omit discussion of them.

```
Inductive Bounded : Rectangle -> Vector -> Prop :=
| B : forall r v,
    fst v < fst (snd r) -> fst (fst r) < fst v ->
    snd v < snd (snd r) -> snd (fst r) < snd v ->
    Bounded r v.
```

```
Inductive Aligned : Qc -> Dim -> Rectangle -> Rectangle -> Prop :=
| CY : forall cv r1 r2,
    (snd (snd r1)) = cv -> (fst (fst r1)) = (fst (fst r2)) ->
    (snd (snd r1)) = (snd (fst r2)) ->
    (fst (snd r1)) = (fst (snd r2)) -> Aligned cv Y r1 r2
| CX : forall cv r1 r2,
    (fst (snd r1)) = cv -> (snd (snd r1)) = (snd (snd r2)) ->
    (fst (snd r1)) = (fst (fst r2)) ->
    (snd (fst r1)) = (snd (fst r2)) -> Aligned cv X r1 r2.

Inductive Merged : Rectangle -> Rectangle -> Rectangle -> Prop :=
| M : forall r1 r2 r3,
    fst (fst r3) = fst (fst r1) -> snd (fst r3) = snd (fst r1) ->
    fst (snd r3) = fst (snd r2) -> snd (snd r3) = snd (snd r2) ->
    Merged r1 r2 r3.
```

2. Decidability

Now that we have defined the KDT property, we want to show that it is decidable. This will allow us to prove that various examples are kd-trees which not be easily doable without this property.

We must define a function that decides if a tree is KDT with respect to a bounding box. We do this as follows:

```
Fixpoint kdt (r : Rectangle) (t : tree) : bool :=
    match t with
    | Point coord =>
        match coord with
        | None => true
        | Some v =>  bounded r v
        end
    | Split d cv t1 t2 =>
        match d with
        | X => (kdt ((fst r), (cv, snd (snd r))) t1) &&
            (kdt ((cv, snd (fst r)),(snd r)) t2) &&
            aligned cv X ((fst r), (cv, snd (snd r)))
            ((cv, snd (fst r)),(snd r)) &&
            merged ((fst r), (cv, snd (snd r)))
            ((cv, snd (fst r)),(snd r)) r
        | Y => (kdt ((fst r), (fst (snd r),cv)) t1) &&
            (kdt ((fst (fst r),cv), (snd r)) t2) &&
            aligned cv Y ((fst r), (fst (snd r),cv))
            ((fst (fst r),cv), (snd r)) &&
            merged ((fst r), (fst (snd r),cv))
            ((fst (fst r),cv), (snd r)) r
    end
  end.
```

To prove decidability, we want to show that the kdt function outputs true if and only if the input tree is a KDT with respect to the input bounding box. In Coq, this becomes the following statement:

```
Theorem kdt_iff_KDT :
    forall (t : tree) (r : Rectangle), KDT r t <-> kdt r t = true.
```

We relegate the formal proof to the Coq source file. The idea is to prove the decidability of the auxiliary properties used in the KDT definition, from which the decidability of the KDT property will easily follow by a relatively simple inductive argument.

3. Insertion Algorithm

With our KDT property inductively defined and its decidability proven, we wish to provide an algorithm for constructing a valid kd-tree for a given set of points in the plane. To this end, we implement the standard point kd-tree insertion algorithm as follows:

```
Fixpoint insert (v : Vector) (d : Dim) (t : tree) : tree :=
  match t with
  | Point c => match c with
              | None => Point (Some v)
              | Some v' => if bneq_d d v v' then
                  Split d (mid d v v')
                    (Point (Some (min_wrt_dim d v v')))
                    (Point (Some (max_wrt_dim d v v')))
                          else
                            t
              end
  | Split X cv t1 t2 => if q_equal (fst v) cv then t else
                        if (q_less_than (fst v) cv) then
                        [ (insert v Y t1) | cv | t2 ]
                        else [ t1 | cv | (insert v Y t2) ]
  | Split Y cv t1 t2 => if q_equal (snd v) cv then t else
                        if (q_less_than (snd v) cv) then
                        [ (insert v X t1) ~ cv ~ t2 ]
                        else [ t1 ~ cv ~ (insert v X t2) ]
  end.
```

Note that this algorithm only inserts points if they are not already present in the tree. It is recursive and works by matching on the structure of the tree. If the tree is empty we insert the point. Else, if the tree consist of a single point we construct a split node that splits the points about the mid point along the current dimension d. Otherwise, we are at a split node and we determine which child to recurse on by comparing the point along the nodes splitting dimension.

3.1. **Insertion Correctness.** We would like to say something regarding the correctness of this algorithm. To do this, we show that the algorithm preserves the KDT invariant. We state this more formally in the following theorem:

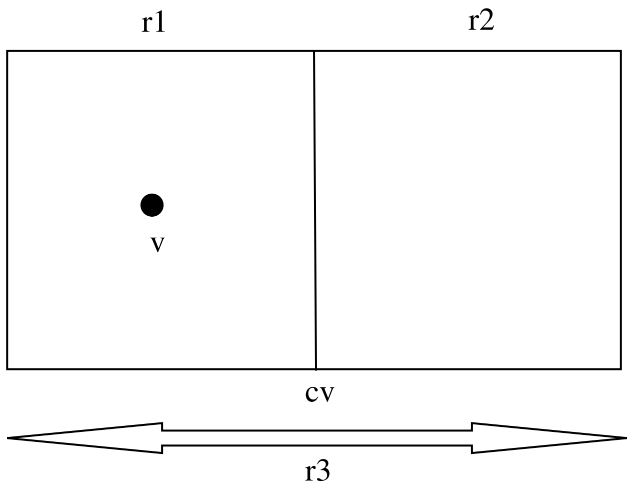
```
Theorem insertion_correctness :
    forall r t v d, KDT r t -> Bounded r v -> KDT r (insert v d t)
```

The proof of this theorem is long and technical so we shall only present the major ideas here. We refer the interested reader to the source file for the full proof. We proceed by induction on the

(KDT r t) hypothesis. The empty tree case is trivial and is solvable using auto. The single point case is also easy though requires a fair amount of technical lemmas regarding Coq's canonical rational library. The most interesting case is that of the split node. Recall that the split node stores a splitting dimension, a cutting value, and references to its two children. We consider the case in which its splitting dimension is X and the x-coordinate of the point to be inserted is less than the cutting value. The other cases follow from a similar analysis. Our goal is now to prove that given a tree and a rectangle that satisfy the KDT invariant and a point that is bound within the rectangle, inserting the point into the tree's left child will preserve the the tree's KDT property. Proving this reduces to applying the appropriate KDT constructors and the application of a very interesting lemma. We will show this lemma now:

```
Lemma generalized_bounded_lemma_x1 : forall (v1 : Vector)
(r1 r2 r3 : Rectangle) (cv : Qc), Bounded r3 v1 ->
fst v1 < cv  -> Aligned cv X r1 r2 -> Merged r1 r2 r3 ->
Bounded r1 v1.
```

We prove this lemma via a number of inversions and rewrites though the figure below provides a clearer picture of what the lemma claims.



4. Future Work

An avenue for future work would be to implement the standard range counting algorithm for axis-aligned rectangles on kd-trees and prove its correctness. The way we would go about doing this is to first write a brute force range counting algorithm and then prove that the output of the range counting algorithm is always equal to the output of the brute force algorithm.

In the same vein, we have ideas regarding proving the correctness of any general range counting algorithm on kd-trees satisfying some certain axioms. This would provide a powerful framework for verifying the correctness of a large class of algorithms on kd-trees.

One final potential extension to the project would be to implement the balanced kd-tree insertion algorithm and prove that it preserves the KDT property. ■