Applied Algorithms Assignment 7

Fei Fan (Peter) Chen

February 19, 2021

Exercise 1 (10 pts)

A 5000 word code book requires 13-bits to express, leaving at most 20 − 13 = 7 bits for parity and error correction. To localize two errors in a 20 bit (information + parity bit) string, we can represent at most $2^7 = 128$ 2-bit error states. However the number of error states are $\binom{20}{2} + 1 = 191$, where the extra 1 is for the no error state. This does not even include the states involving a single parity bit error, which is 20 more different states. As the number of error states is larger than the number of states that can be represented by the available bits for parity use, we cannot correct for more than 2 error bits.

Exercise 2 (15 pts)

a) See Figure 1.

b) See Figure 2.

c) See Figure 3.

Exercise 3 (10 pts)

See source code and Figure 4.

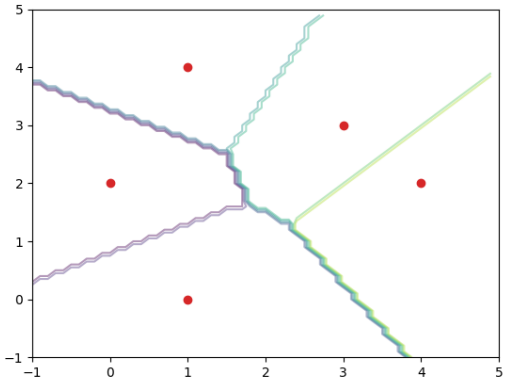


Figure 1: Quad Tree Implementation.

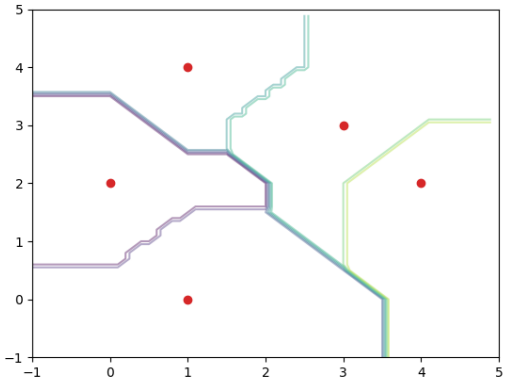


Figure 2: Quad Tree Implementation.

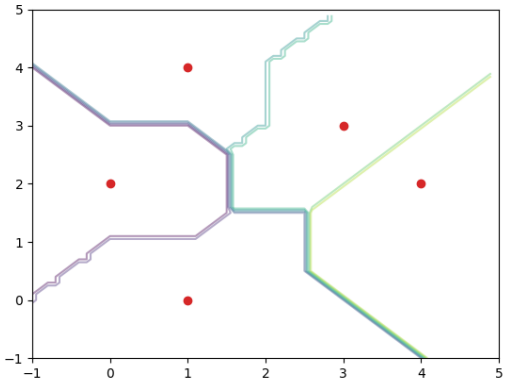


Figure 3: Quad Tree Implementation.

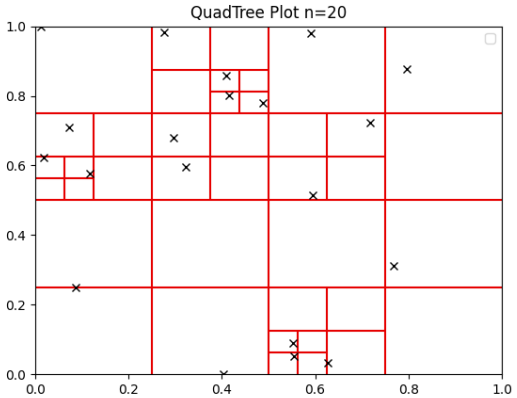
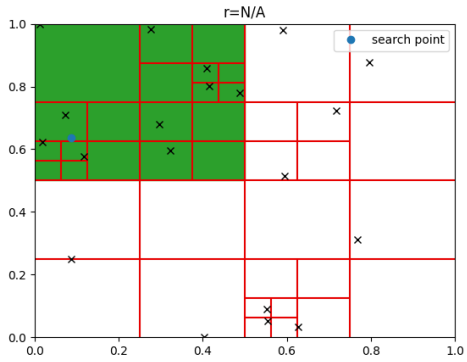


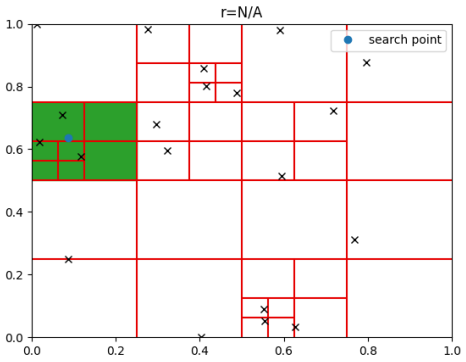
Figure 4: Quad Tree Implementation.

Exercise 4 (10 pts)

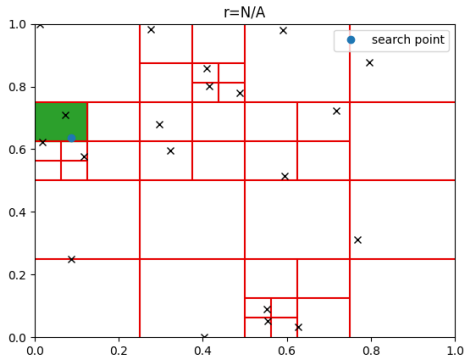
See source code and Figure 5.



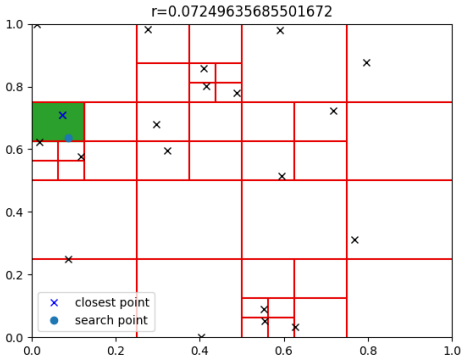
(a) Step 1: Find the subtree the node is in



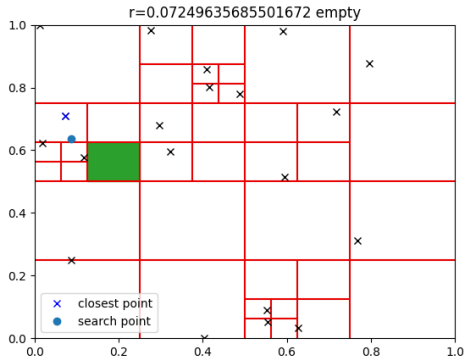
(b) Step 2: Find the subtree the node is in



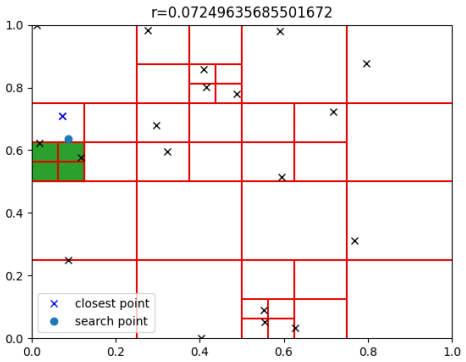
(c) Step 3: Find the subtree the node is in



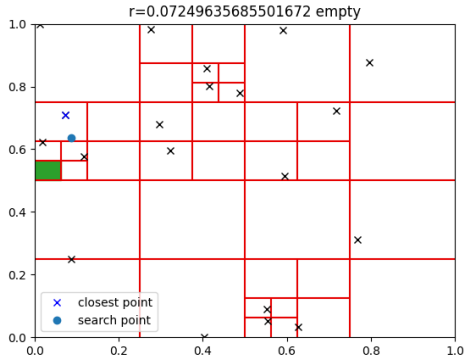
(d) Step 4: Find closest node distance r



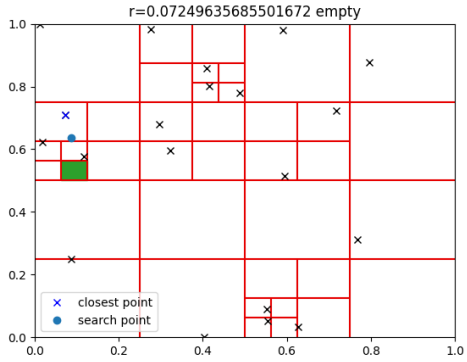
(e) Step 5: Search in close enough neighbours



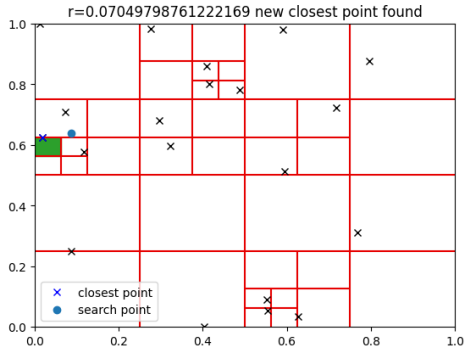
(f) Step 6: Search in close enough neighbours



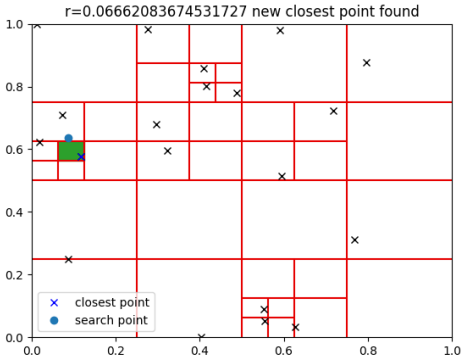
(g) Step 7: Search in close enough neighbours



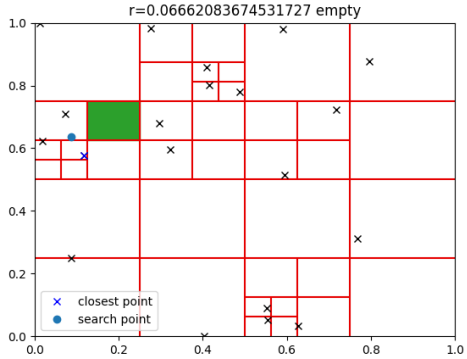
(h) Step 8: Search in close enough neighbours



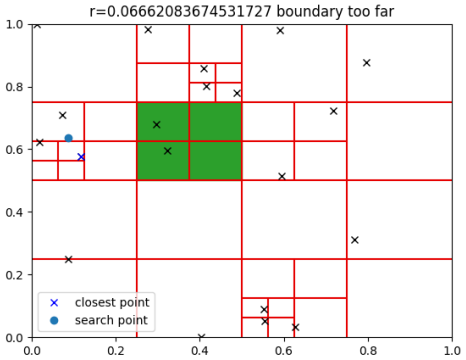
(i) Step 9: Search in close enough neighbours



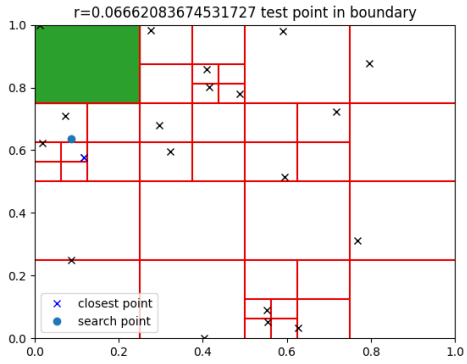
(j) Step 10: Search in close enough neighbours



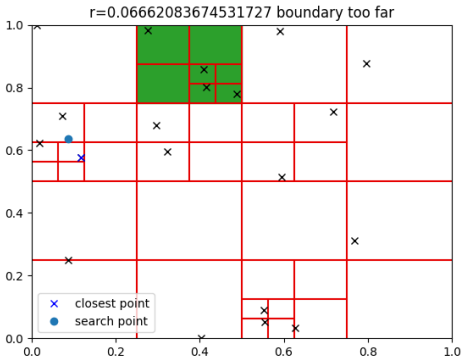
(k) Step 11: Search in close enough neighbours



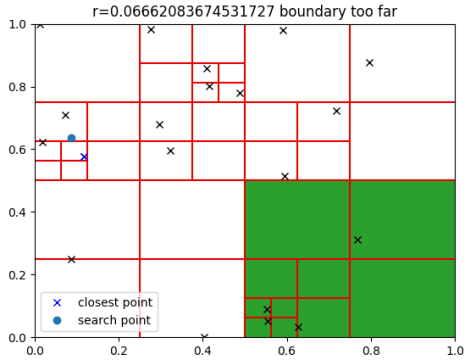
(l) Step 13: Search in close enough neighbours



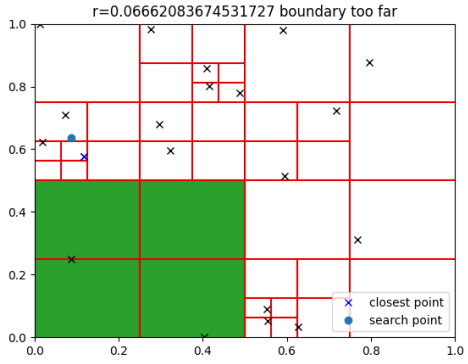
(m) Step 14: Search in close enough neighbours



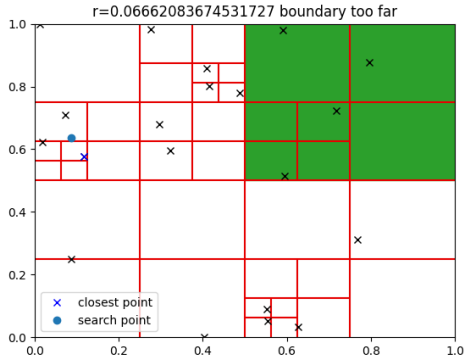
(n) Step 15: Search in close enough neighbours



(o) Step 17: Search in close enough neighbours



(p) Step 18: Search in close enough neighbours



(q) Step 19: Search in close enough neighbours

Figure 5: Quad Tree Search.

Code

```
#!/usr/bin/python

import random
import matplotlib.pyplot as plt
import math

class QDTreeNode():
  def __init__(self, bottom_left, bottom_right, top_left, top_right):
    # should have check to make sure it is a rectangle, but meh
    self._bl = bottom_left
    self._br = bottom_right
    self._tl = top_left
    self._tr = top_right
    self._mid = ((bottom_right[0] + bottom_left[0])/2,
                 (top_left[1] + bottom_left[1])/2)
    self._points = []

    self._node_bottom_left = None
    self._node_bottom_right = None
    self._node_top_left = None
    self._node_top_right = None

  def add_point(self, point):
    self._points.append(point)

  def build(self):
    if len(self._points) <= 1:
      return
    self._node_bottom_left = QDTreeNode(
      bottom_left=self._bl,
      bottom_right=(self._mid[0], self._bl[1]),
```

```
      top_left=(self._bl[0], self._mid[1]),
      top_right=self._mid
    )

    self._node_bottom_right = QDTreeNode(
      bottom_left=(self._mid[0], self._br[1]),
      bottom_right=self._br,
      top_left=self._mid,
      top_right=(self._br[0], self._mid[1])
    )

    self._node_top_left = QDTreeNode(
      bottom_left=(self._tl[0], self._mid[1]),
      bottom_right=self._mid,
      top_left = self._tl,
      top_right = (self._mid[0], self._tl[1])
    )

    self._node_top_right = QDTreeNode(
      bottom_left=self._mid,
      bottom_right=(self._tr[0], self._mid[1]),
      top_left = (self._mid[0], self._tr[1]),
      top_right = self._tr
    )

    for p in self._points:
      if p[0] > self._mid[0]:
        if p[1] > self._mid[1]:
          self._node_top_right.add_point(p)
        else:
          self._node_bottom_right.add_point(p)
      else:
        if p[1] > self._mid[1]:
          self._node_top_left.add_point(p)
        else:
          self._node_bottom_left.add_point(p)

    self._node_bottom_left.build()
    self._node_bottom_right.build()
    self._node_top_left.build()
    self._node_top_right.build()

# returns the shortest straight-line distance
# from point to the bounding box
def distance(self, point, bound1, bound2):
    if (point[1] <= bound1[1] and point[1] >= bound2[1]) or \
       (point[1] <= bound2[1] and point[1] >= bound2[1]):
      d = abs(bound1[0] - point[0])
```

```
    elif (point[0] <= bound1[0] and point[0] >= bound2[0]) or \
         (point[0] >= bound1[0] and point[0] <= bound2[0]):
      d = abs(bound1[1] - point[1])
    else:
      xDistance1 = bound1[0] - point[0]
      xDistance2 = bound2[0] - point[0]
      yDistance1 = bound1[1] - point[1]
      yDistance2 = bound2[1] - point[1]
      d1 = xDistance1*xDistance1 + yDistance1*yDistance1
      d2 = xDistance2*xDistance2 + yDistance2*yDistance2
      d = min(d1, d2)
      d = math.sqrt(d)
    return d

  def in_bound(self, point):
    if (point[0] <= self._br[0] and point[0] > self._bl[0]) and \
      (point[1] >= self._br[1] and point[1] < self._tr[1]):
      return True
    return False

  def search(self, point, closestPoint, closestR, snapshot):
    if len(self._points) == 0:
      snapshot.show(boundingBox=self._box(self),
         searchPoint=point, closestPoint=closestPoint,
         r=f"{closestR} empty")
      return closestPoint, closestR
    elif len(self._points) == 1:
      x = self._points[0][0] - point[0]
      y = self._points[0][1] - point[1]
      d_min_to_local = math.sqrt(x*x + y*y)
      if closestR < d_min_to_local:
        snapshot.show(boundingBox=self._box(self),
           searchPoint=point, closestPoint=closestPoint,
           r=f"{closestR} test point in boundary")
        return closestPoint, closestR
      else:
        snapshot.show(boundingBox=self._box(self),
           searchPoint=point, closestPoint=self._points[0],
           r=f"{d_min_to_local} new closest point found")
        return self._points[0], d_min_to_local

    d1 = self.distance(point, self._bl, self._br)
    d2 = self.distance(point, self._bl, self._tl)
    d3 = self.distance(point, self._tl, self._tr)
    d4 = self.distance(point, self._br, self._tr)
    d_min_to_boundary = min(d1, d2, d3, d4)
    if closestR < d_min_to_boundary:
      snapshot.show(boundingBox=self._box(self),
```

```
            searchPoint=point, closestPoint=closestPoint,
            r=f"{closestR} boundary too far")
        return closestPoint, closestR

    # search rest of the quad tree
    snapshot.show(boundingBox=self._box(self),
            searchPoint=point, closestPoint=closestPoint, r=closestR)
    p, r = self._node_bottom_left.search(point, closestPoint,
        closestR, snapshot)
    p, r = self._node_bottom_right.search(point, closestPoint,
        closestR, snapshot)
    p, r = self._node_top_left.search(point, closestPoint,
        closestR, snapshot)
    p, r = self._node_top_right.search(point, closestPoint,
        closestR, snapshot)
    return p, r

def _box(self, node):
    return [ node._bl[0], node._br[0], node._bl[1], node._tl[1] ]

def findNearest(self, point, snapshot=None):
    if len(self._points) == 0:
        # our box is 1 x 1, so there can be no distance longer than 2
        return None, 2.0
    if len(self._points) == 1:
        x = self._points[0][0] - point[0]
        y = self._points[0][1] - point[1]
        return self._points[0], math.sqrt(x*x + y*y)

    if self._node_bottom_right.in_bound(point):
        box = "br"
        snapshot.show(boundingBox=self._box(self._node_bottom_right),
            searchPoint=point, closestPoint=None, r="N/A")
        p, r = self._node_bottom_right.findNearest(point, snapshot)
        snapshot.show(boundingBox=self._box(self._node_bottom_right),
            searchPoint=point, closestPoint=p, r=r)
    elif self._node_bottom_left.in_bound(point):
        box = "bl"
        snapshot.show(boundingBox=self._box(self._node_bottom_left),
          searchPoint=point, closestPoint=None, r="N/A")
        p, r = self._node_bottom_left.findNearest(point, snapshot)
        snapshot.show(boundingBox=self._box(self._node_bottom_left),
            searchPoint=point, closestPoint=p, r=r)
    elif self._node_top_left.in_bound(point):
        box = "tl"
        snapshot.show(boundingBox=self._box(self._node_top_left),
          searchPoint=point, closestPoint=None, r="N/A")
        p, r = self._node_top_left.findNearest(point, snapshot)
```

```
      snapshot.show(boundingBox=self._box(self._node_top_left),
          searchPoint=point, closestPoint=p, r=r)
    else:
      box = "tr"
      snapshot.show(boundingBox=self._box(self._node_top_right),
          searchPoint=point, closestPoint=None, r="N/A")
      p, r = self._node_top_right.findNearest(point, snapshot)
      snapshot.show(boundingBox=self._box(self._node_top_right),
          searchPoint=point, closestPoint=p, r=r)

    # Now we search other boxes with distance less than r
    if box != "br":
      p, r = self._node_bottom_right.search(point, p, r, snapshot)
    if box != "bl":
      p, r = self._node_bottom_left.search(point, p, r, snapshot)
    if box != "tl":
      p, r = self._node_top_left.search(point, p, r, snapshot)
    if box != "tr":
      p, r = self._node_top_right.search(point, p, r, snapshot)

    return p, r

  def draw(self, shade=0.9):
    if len(self._points) <= 1:
      return

    plt.hlines(y=self._mid[1], xmin=self._bl[0], xmax=self._br[0],
    colors=(shade, 0.0, 0.0))
    plt.vlines(x=self._mid[0], ymin=self._bl[1], ymax=self._tl[1],
    colors=(shade, 0.0, 0.0))

    self._node_bottom_left.draw()
    self._node_bottom_right.draw()
    self._node_top_left.draw()
    self._node_top_right.draw()

class QuadTree():
  def __init__(self, boxA=(0.0,0.0), boxB=(1.0,0.0), boxC=(0.0,1.0),
    boxD=(1.0,1.0), n=20):
    self._root = QDTreeNode(boxA, boxB, boxC, boxD)
    self.points = [ (random.uniform(0,1), random.uniform(0,1))
        for _ in range(n) ]
    self._construct()

  def _construct(self):
    # create 4 partitions around self._root.mid()
    for p in self.points:
      self._root.add_point(p)
```

```
    self._root.build()

  def findNearest(self, point):
    self._root.findNearest(point, snapshot=self)

  def show(self, boundingBox=None, searchPoint=None,
    closestPoint=None, r=None):
    plt.figure()
    plt.xlim([0,1])
    plt.ylim([0,1])
    if r is None:
      plt.title(f"QuadTree Plot n={len(self.points)}")
    else:
      plt.title(f"r={r}")
    if boundingBox is not None:
      plt.axvspan(xmin=boundingBox[0],
                  xmax=boundingBox[1],
                  ymin=boundingBox[2],
                  ymax=boundingBox[3], color="tab:green")
    # draw the points
    x = []
    y = []
    for p in self.points:
      x.append(p[0])
      y.append(p[1])

    plt.plot(x, y, 'x', color='black')
    self._root.draw()

    if closestPoint is not None:
      plt.plot(closestPoint[0], closestPoint[1], 'x', color='blue',
      label='closest point')
    if searchPoint is not None:
      plt.plot(searchPoint[0], searchPoint[1], 'o', color='tab:blue',
      label='search point')
    plt.legend()


if __name__ == "__main__":
  random.seed(None)
  qd = QuadTree()
  qd.show()
  qd.findNearest(point=(random.uniform(0,1), random.uniform(0,1)))
  plt.show()
```