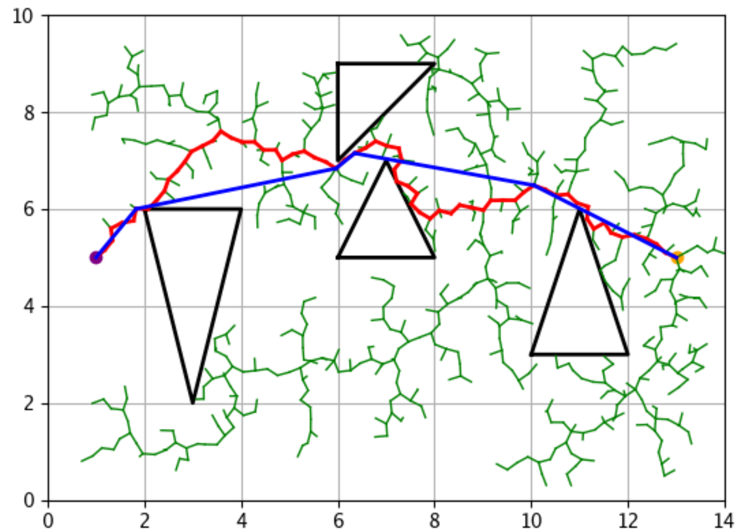
CS 133b	January 30, 2024

Problem Set 4

Problem 1 (Rapidly-exploring Random Tree Planner) - 48 points

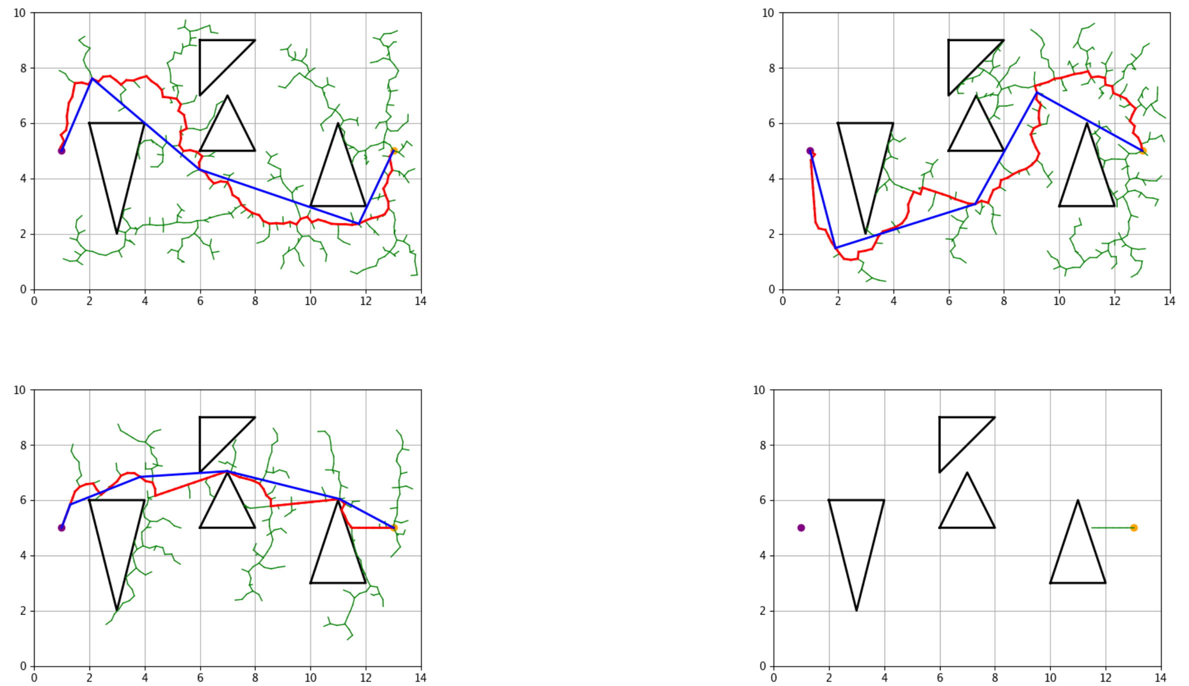


part (a)
The code is shown on the next page. An example of output is shown above.

Code for problem 1a

```python
def rrt(startnode, goalnode, visual):
    # Start the tree with the startnode (set no parent just in case).
    startnode.parent = None
    tree = [startnode]

    # Function to attach a new node to an existing node: attach the
    # parent, add to the tree, and show in the figure.
    def addtotree(oldnode, newnode):
        newnode.parent = oldnode
        tree.append(newnode)
        visual.drawEdge(oldnode, newnode, color='g', linewidth=1)
        visual.show()

    # Loop - keep growing the tree.
    steps = 0
    while True:
        # Determine the target state.
        #FIXME:
        x_coord = random.uniform(xmin, xmax)
        y_coord = random.uniform(ymin, ymax)
        targetnode = Node(x_coord, y_coord)

        # Directly determine the distances to the target node.
        distances = np.array([node.distance(targetnode) for node in tree])
        index     = np.argmin(distances)
        nearnode  = tree[index] #nearest node to target
        d         = distances[index] #distance between nearest node and target

        # Determine the next node.
        #FIXME:
        if d <= DSTEP:
            nextnode = targetnode
        else:
            # directional vector between nearnode and targetnode
            dir_vec = np.array([targetnode.x - nearnode.x, targetnode.y - nearnode.y])
            dir_vec = dir_vec/np.linalg.norm(dir_vec)
            new_x = nearnode.x + DSTEP * dir_vec[0]
            new_y = nearnode.y + DSTEP * dir_vec[1]
            nextnode = Node(new_x, new_y)

        # Check whether to attach.
        if nextnode.inFreespace() and nearnode.connectsTo(nextnode):
            addtotree(nearnode, nextnode)

            # If within DSTEP, also try connecting to the goal.  If
            # the connection is made, break the loop to stop growing.
            #FIXME:
            dist_togoal = nextnode.distance(goalnode)
            if dist_togoal <= DSTEP and nextnode.connectsTo(goalnode):
                addtotree(nextnode, goalnode)
                break

        # Check whether we should abort - too many steps or nodes.
        steps += 1
        if (steps >= SMAX) or (len(tree) >= NMAX):
            print("Aborted after %d steps and the tree having %d nodes" %
                  (steps, len(tree)))
            return None

    # Build the path.
    path = [goalnode]
    while path[0].parent is not None:
        path.insert(0, path[0].parent)
```

For the first row: 5% probability on the left, 50% probability on the right
For the second row: 99% probability on the left, 100% probability on the right

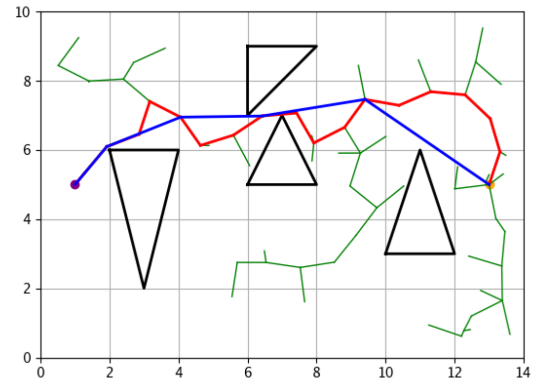


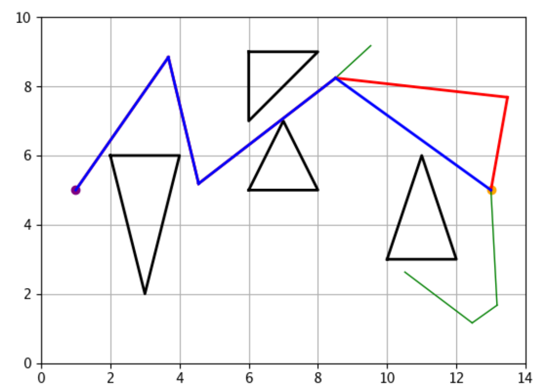
part (b)

As you increase the probability that the goal node is selected as the target node, then the tree becomes smaller (less nodes are added to the tree). In the case that the probability is 100 percent, no path is ever made between the start and goal because there is an obstacle. With probability of 99 percents, there are instances when a path cannot be found given the step size and the max number of steps (attempts), though in the output provided above a path is found. The code is on the next page, the probability of that the goal node is selected as the target node is adjusted accordingly (variable goal_p).

Code for problem 1b

```python
def rrt(startnode, goalnode, visual):
    # Start the tree with the startnode (set no parent just in case).
    startnode.parent = None
    tree = [startnode]
    def addtotree(oldnode, newnode):
        newnode.parent = oldnode
        tree.append(newnode)
        visual.drawEdge(oldnode, newnode, color='g', linewidth=1)
        visual.show()

    # Loop - keep growing the tree.
    steps = 0
    while True:
        #probability that goal is selected as target,
        #adjusted as necessary for the given problem
        goal_p = 0.05
        val = np.random.choice(a = np.array([0,1]), p = np.array([goal_p, 1-goal_p]))

        if val == 0:
            targetnode = goalnode

        else:
            x_coord = random.uniform(xmin, xmax)
            y_coord = random.uniform(ymin, ymax)
            targetnode = Node(x_coord, y_coord)

        # Directly determine the distances to the target node.
        distances = np.array([node.distance(targetnode) for node in tree])
        index     = np.argmin(distances)
        nearnode  = tree[index] #nearest node to target
        d         = distances[index] #distance between nearest node and target

        # Determine the next node.
        #FIXME:
        if d <= DSTEP:
            nextnode = targetnode
        else:
            # directional vector between nearnode and targetnode
            dir_vec = np.array([targetnode.x - nearnode.x, targetnode.y - nearnode.y])
            dir_vec = dir_vec/np.linalg.norm(dir_vec)
            new_x = nearnode.x + DSTEP * dir_vec[0]
            new_y = nearnode.y + DSTEP * dir_vec[1]
            nextnode = Node(new_x, new_y)

        # Check whether to attach.
        if nextnode.inFreespace() and nearnode.connectsTo(nextnode):
            addtotree(nearnode, nextnode)

            # If within DSTEP, also try connecting to the goal.  If
            # the connection is made, break the loop to stop growing.
            #FIXME:
            dist_togoal = nextnode.distance(goalnode)
            if dist_togoal <= DSTEP and nextnode.connectsTo(goalnode):
                addtotree(nextnode, goalnode)
                break

        # Check whether we should abort - too many steps or nodes.
        steps += 1
        if (steps >= SMAX) or (len(tree) >= NMAX):
            print("Aborted after %d steps and the tree having %d nodes" %
                  (steps, len(tree)))
            return None

    # Build the path.
    path = [goalnode]
    while path[0].parent is not None:
        path.insert(0, path[0].parent)
```

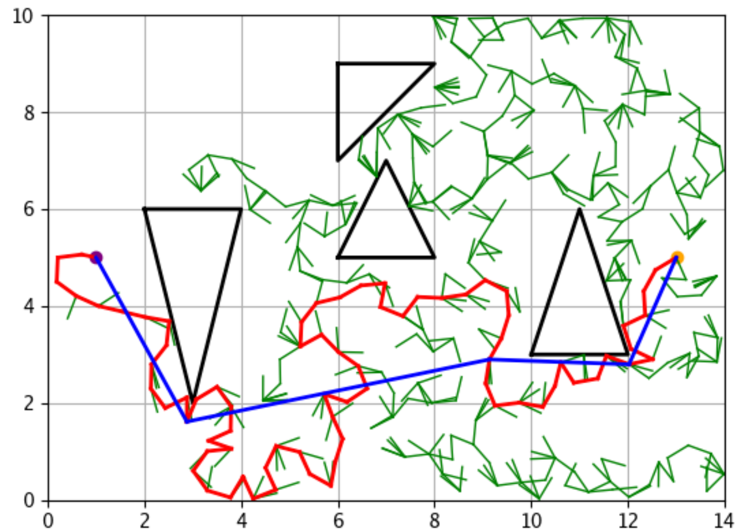




part (c)

The top graph corresponds to an output given when the step size was 1, the bottom graph corresponds to a step size of 5. As shown above we can see that the number of nodes in the tree significantly decrease from when the step size was 0.25 (part b), especially when the step size is set to 5. Additionally the number of steps is also decreased. The code is the same as in 1b (shown in the previous page), the step size (DSTEP) is adjusted accordingly.

Problem 2 (Expansive-Spaces Tree Planner) - 48 points



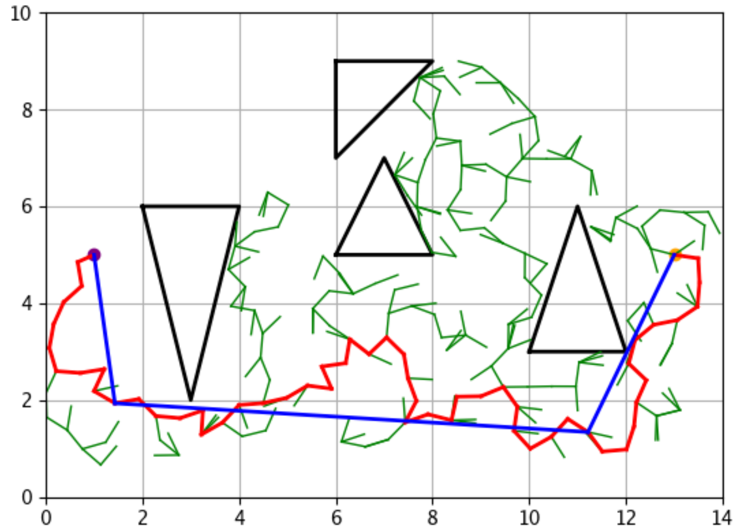
part (a)

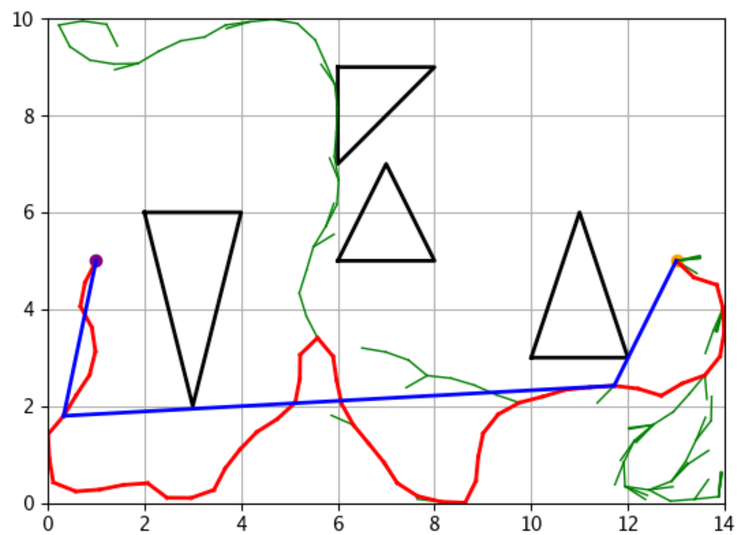
A function and attribute were added to the Node class (shown below). The rest of the code changed is shown on the next page (part of the est function). DSTEP was set to 0.50. An example of output is shown above.

```
class Node:
    def __init__(self, x, y):
        # Define a parent (cleared for now).
        self.parent = None

        # Define/remember the state/coordinates (x,y).
        self.x = x
        self.y = y
        self.neighbors = []

    # added this
    # returns the number of neighbors within distance radius
    def num_neighbors(self, tree, radius):
        count = 0
        for node in tree:
            if node.x == self.x and node.y == self.y:
                pass
            else:
                if self.distance(node) <= radius:
                    count += 1
        return count
```

Code for problem 2a (within est function)

```
while True:
    # Determine the local density by the number of nodes nearby.
    # KDTree uses the coordinates to compute the Euclidean distance.
    # It returns a NumPy array, same length as nodes in the tree.
    X = np.array([node.coordinates() for node in tree])
    kdtree  = KDTree(X)
    numnear = kdtree.query_ball_point(X, r=1.5*DSTEP, return_length=True)

    # Directly determine the distances to the goal node.
    distances = np.array([node.distance(goalnode) for node in tree])

    # Select the node from which to grow, which minimizes some metric.
    #FIXME
    # get the number of neighbors within distance DSTEP*1.50 of every node
    num_neighbors = np.array([node.num_neighbors(tree, DSTEP*1.50) for node in tree])
    min_neighbors = np.min(num_neighbors)

    # get the indices that correspond to nodes with min_neighbors amount of neighbors
    # within distance DSTEP*1.50
    indices = np.where(num_neighbors == min_neighbors)[0]
    i = np.random.choice(indices)
    grownode = tree[i]

    # Check the incoming heading, potentially to bias the next node.
    if grownode.parent is None:
        heading = 0
    else:
        heading = atan2(grownode.y - grownode.parent.y,
                        grownode.x - grownode.parent.x)

    # Find something nearby: keep looping until the tree grows.
    while True:
        # Pick the next node randomly.
        #FIXME:
        heading = random.uniform(-pi, pi)
        dx = DSTEP * cos(heading)
        dy = DSTEP * sin(heading)
        x_coord = grownode.x + dx
        y_coord = grownode.y + dy
        nextnode = Node(x_coord, y_coord)

        # Try to connect.
        #FIXME...
        if nextnode.inFreespace() and grownode.connectsTo(nextnode):
            addtotree(grownode, nextnode)
            break

    # Once grown, also check whether to connect to goal.
    #FIXME...
    dist_togoal = nextnode.distance(goalnode)
    if dist_togoal <= DSTEP and nextnode.connectsTo(goalnode):
        addtotree(nextnode, goalnode)
        break
```



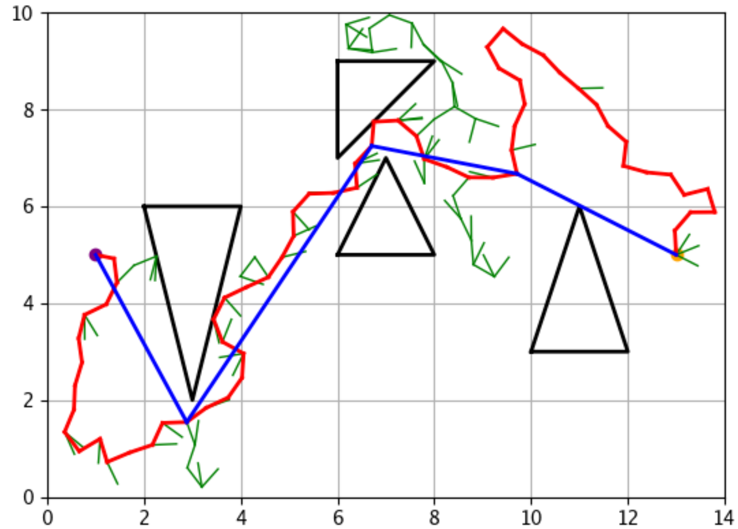


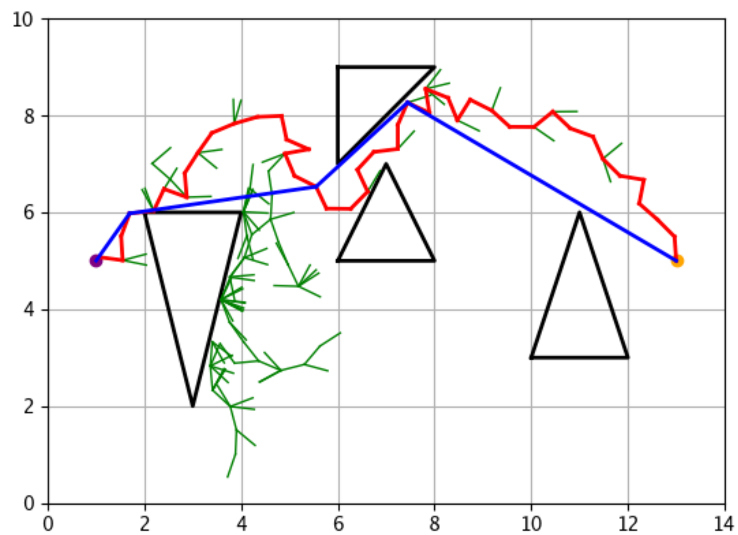
part (b)

A function and attribute were added to the Node class (same as shown in part a). The rest of the code changed is shown on the next page (part of the est function). DSTEP was set to 0.50. Example of outputs are shown above. The top graph is with a standard deviation of pi/2 while the bottom is with a standard deviation of pi/8. As you can see, in both cases, the branches are straighter than they were in part a. Though this is more noticeable with the smaller standard deviation of pi/8.

Code for problem 2b (within est function)

```python
while True:
    # Determine the local density by the number of nodes nearby.
    # KDTree uses the coordinates to compute the Euclidean distance.
    # It returns a NumPy array, same length as nodes in the tree.
    X = np.array([node.coordinates() for node in tree])
    kdtree  = KDTree(X)
    numnear = kdtree.query_ball_point(X, r=1.5*DSTEP, return_length=True)

    # Directly determine the distances to the goal node.
    distances = np.array([node.distance(goalnode) for node in tree])

    # Select the node from which to grow, which minimizes some metric.
    #FIXME
    # get the number of neighbors within distance DSTEP*1.50 of every node
    num_neighbors = np.array([node.num_neighbors(tree, DSTEP*1.50) for node in tree])
    min_neighbors = np.min(num_neighbors)

    # get the indices that correspond to nodes with min_neighbors amount of neighbors
    # within distance DSTEP*1.50
    indices = np.where(num_neighbors == min_neighbors)[0]
    i = np.random.choice(indices)
    grownode = tree[i]

    # Check the incoming heading, potentially to bias the next node.
    if grownode.parent is None:
        heading = 0
    else:
        heading = atan2(grownode.y - grownode.parent.y,
                        grownode.x - grownode.parent.x)

    # Find something nearby: keep looping until the tree grows.
    while True:
        # Pick the next node randomly.
        #FIXME:
        std = pi/2 #standard deviation
        theta = np.random.normal(loc = heading, scale = std)
        dx = DSTEP * cos(theta)
        dy = DSTEP * sin(theta)
        x_coord = grownode.x + dx
        y_coord = grownode.y + dy
        nextnode = Node(x_coord, y_coord)

        # Try to connect.
        #FIXME...
        if nextnode.inFreespace() and grownode.connectsTo(nextnode):
            addtotree(grownode, nextnode)
            break

    # Once grown, also check whether to connect to goal.
    #FIXME...
    dist_togoal = nextnode.distance(goalnode)
    if dist_togoal <= DSTEP and nextnode.connectsTo(goalnode):
        addtotree(nextnode, goalnode)
        break
```





part (c)

A function and attribute were added to the Node class (same as shown in part a). The rest of the code changed is shown on the next page (part of the est function). DSTEP was set to 0.50. Examples of outputs are shown above. The top graph is with a scale of 1, while the bottom is with a scale of 5. As you can see, in both cases, the tree now tries to go towards the goal.

Code for problem 2c (within est function)

```
while True:
    # Determine the local density by the number of nodes nearby.
    # KDTree uses the coordinates to compute the Euclidean distance.
    # It returns a NumPy array, same length as nodes in the tree.
    X = np.array([node.coordinates() for node in tree])
    kdtree  = KDTree(X)
    numnear = kdtree.query_ball_point(X, r=1.5*DSTEP, return_length=True)

    # Directly determine the distances to the goal node.
    scale = 1.0 #scale factor used in metric
    distances = np.array([node.distance(goalnode) for node in tree])

    # Select the node from which to grow, which minimizes some metric.
    #FIXME
    # get the number of neighbors within distance DSTEP*1.50 of every node
    num_neighbors = np.array([node.num_neighbors(tree, DSTEP*1.50) for node in tree])
    metric = num_neighbors + scale * distances
    min_val = np.min(metric)

    # get the indices that correspond to nodes with min value for metric
    indices = np.where(metric == min_val)[0]
    i = np.random.choice(indices)
    grownode = tree[i]

    # Check the incoming heading, potentially to bias the next node.
    if grownode.parent is None:
        heading = 0
    else:
        heading = atan2(grownode.y - grownode.parent.y,
                        grownode.x - grownode.parent.x)

    # Find something nearby: keep looping until the tree grows.
    while True:
        # Pick the next node randomly.
        #FIXME:
        std = pi/2 #standard deviation
        theta = np.random.normal(loc = heading, scale = std)
        dx = DSTEP * cos(theta)
        dy = DSTEP * sin(theta)
        x_coord = grownode.x + dx
        y_coord = grownode.y + dy
        nextnode = Node(x_coord, y_coord)

        # Try to connect.
        #FIXME...
        if nextnode.inFreespace() and grownode.connectsTo(nextnode):
            addtotree(grownode, nextnode)
            break

    # Once grown, also check whether to connect to goal.
    #FIXME...
    dist_togoal = nextnode.distance(goalnode)
    if dist_togoal <= DSTEP and nextnode.connectsTo(goalnode):
        addtotree(nextnode, goalnode)
        break
```

Problem 3 (Time Spent) - 4 points

I spent about 2.5 hours on this set. No bottlenecks for me.