DSC 40B - Homework 09

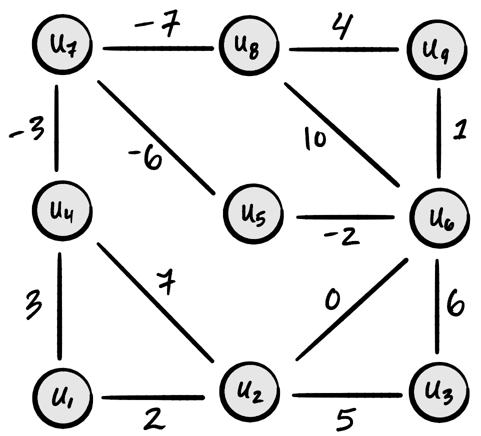
Due: Wednesday, June 14

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 p.m.

Problem 1.

In this problem you will be asked to list the edges in the minimum spanning tree of the graph below in the order that they are added by either Prim's algorithm or Kruskal's algorithm.

In order to simplify grading, please write an edge with the *smaller* node first. For example: (u_3, u_7) instead of (u_7, u_3). Also, when writing an edge, make sure to write the edge as a pair of nodes, not the weight of the edge. Thanks!



a) Suppose Prim's algorithm is run on the above graph, using node u_1 as the starting node. List the edges of the resulting minimum spanning tree computed in the order that they are added by the algorithm.

Solution:

(u_1, u_2)
(u_2, u_6)
(u_5, u_6)
(u_5, u_7)
(u_7, u_8)
(u_4, u_7)
(u_6, u_9)
(u_2, u_3)

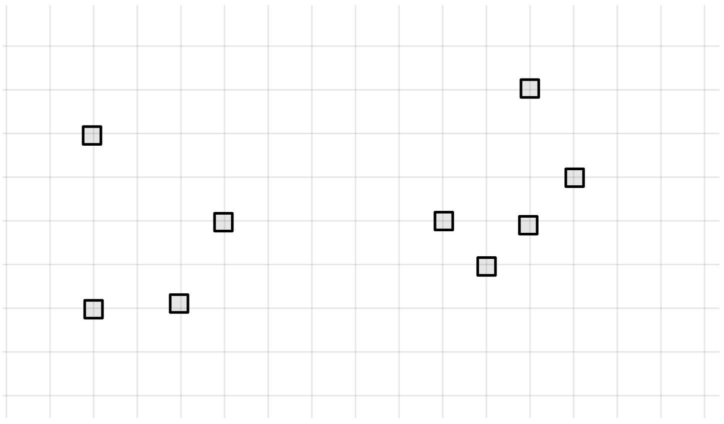
b) Suppose Kruskal's algorithm is run on the graph above. List the edges of the resulting minimum spanning tree in the order that they are added by the algorithm.

Solution:

(u_7, u_8)
(u_5, u_7)
(u_4, u_7)
(u_5, u_6)
(u_2, u_6)
(u_6, u_9)
(u_1, u_2)
(u_2, u_3)

Problem 2.

The picture below shows a set of points in 2-dimensional space. A grid is provided so that you can compute the distance between points; each grid cell is 1 unit wide and 1 unit tall. You may assume that each data point is placed on a grid intersection.



Suppose a weighted distance graph G is constructed from this data set (recall that a distance graph is a complete graph whose nodes represent points in space, and whose edges are weighted by the distance between its endpoints). Then suppose that a minimum spanning tree is computed for G. What will be the weight of the largest edge in this minimum spanning tree?

Solution: The weight of the longest edge will be 5. Kruskal's algorithm will proceed by first connecting all of the nodes (points) in the left and right clusters independently. Next, it will find the shortest edge between a node in the left cluster and a node in the right – this will be the edge between the two "middle" points. The length of this edge is 5.

Problem 3.

Suppose we are given a undirected weighted graph $G = (V, E; w)$, where $w : E \to \mathbb{R}$ is the map to assign a weight $w(e)$ to each edgee $\in E$. Let T^* be a minimum spanning tree of G.

a) Now let $G_1 = (V, E; w_1)$ be a modified version of G: In particular, G_1 share the same vertex set and edge set as G. The only difference lies in the edge weights, where for any edge $e \in E$, the new weight doubles the previous weight $w(e)$; that is, $w_1(e) = 2w(e)$.

(True or False): The tree T^* is still a minimum spanning tree for G_1 as well. You do not need to provide a justification.

Solution:

● True

○ False

b) Now let $G_2 = (V, E; w_2)$ be another modified version of G: In particular, G_1 share the same vertex set and edge set as G. The only difference lies in the edge weights, where for any edge $e \in E$, the new weight equals the previous weight $w(e) + 2$; that is, $w_2(e) = w(e) + 2$.

(True or False): The tree T^* is still a minimum spanning tree for G_2 as well. You do not need to provide a justification.

Solution:

[✓ True]

○ False

c) Let $s \in V$ be a source node. Let $\widehat{T}$ be a shortest path tree of G from the source s. Consider the modified graph G_2 defined above.

(True or False): The tree $\widehat{T}$ is a shortest path tree for G_2 from the source s as well.

Solution:

○ True

[✓ False]

Programming Problem 1.

In lecture, we saw that Kruskal's algorithm can be used to cluster a weighted graph. The name for this approach is *single linkage clustering*.

In a file named `slc.py`, write a function `slc(graph, d, k)` which accepts the following arguments:

- `graph`: An instance of `dsc40graph.UndirectedGraph`.
- `d`: A function of two arguments which takes in two nodes and returns the distance (or dissimilarity) between them.
- `k`: A positive integer describing the number of clusters which should be found.

The function should perform single linkage clustering using Kruskal's algorithm and it should return a `frozenset` of k `frozenset`s, each representing a cluster of the graph.

Example:

```
>>> g = dsc40graph.UndirectedGraph()
>>> edges = [('a', 'b'), ('a', 'c'), ('c', 'd'), ('b', 'd')]
>>> for edge in edges: g.add_edge(*edge)
>>> def d(edge):
...     u, v = sorted(edge)
...     return {
...         ('a', 'b'): 1,
...         ('a', 'c'): 4,
...         ('b', 'd'): 3,
...         ('c', 'd'): 2,
...     }[(u, v)]
>>> slc(g, d, 2)
frozenset({frozenset({'a', 'b'}), frozenset({'c', 'd'})})
```

Note: to implement Kruskal's algorithm, you'll need an implementation of a Disjoint Set Forest data structure. We've uploaded a simple one here: `https://gist.github.com/eldridgejm/983d6ce03a82bf295599e9880ef02bab`

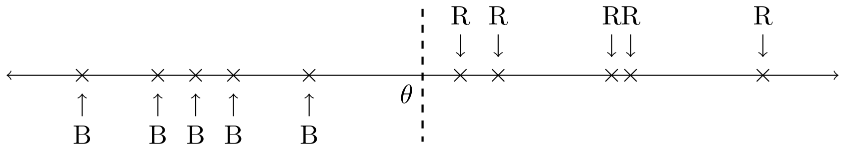
You can copy and paste this into `slc.py`, or put it in a separate file that is imported; if you do this, make sure to upload that file alongside `slc.py`.

Solution:

Programming Problem 2.

Let `data` be a list of n unique real numbers. Furthermore, suppose that each number in `data` is assigned a color – it is either `'red'` or `'blue'`. Let `colors` be a list of n strings, such that `colors[i]` gives the color of `data[i]`. In all parts of this problem, you may assume for simplicity that there is at least one data point of each color.

a) To begin, suppose that **all of the blue points are less than all of the red points**. In a file named `min_ell_theta.py`, write an efficient function called `learn_theta(data, colors)` which takes in two arguments – the lists `data` and `colors` as described above – and returns a single number θ such that all of the blue points are $\leq \theta$ and all of the red points are $> \theta$, as is depicted in the picture below. You may *not* assume that `data` is sorted. The time complexity of your algorithm should be optimal.



Solution: We can set θ to be any number between the biggest blue point and the smallest red point. In fact, we can set θ to be *exactly* the largest blue point, as then all blue points are $\leq \theta$ as desired.

We can do this in linear time by looping through all of the data points, keeping track of the largest blue point seen so far:

```python
def learn_theta(data, colors):
    biggest_blue = -float('inf')
    for i in range(len(data)):
        if colors[i] == 'blue' and data[i] > biggest_blue:
            biggest_blue = x
    return biggest_blue
```

A more "pythonic" approach is to ditch the `range` and use `zip`. `zip` takes two (or more) lists and returns pairs by "zipping" the lists together, elementwise:

```python
def learn_theta(data, colors):
    biggest_blue = -float('inf')
    for x, color in zip(data, colors):
        if color == 'blue' and x > biggest_blue:
            biggest_blue = x
    return biggest_blue
```

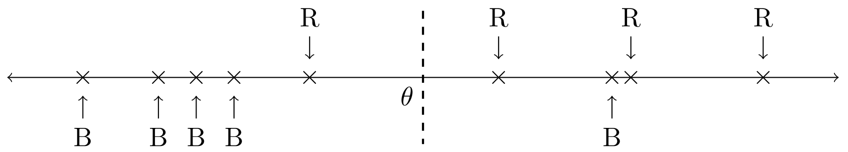
Of course, we could always use a comprehension. The below filters out the red points and returns the max of the blues:

```python
def learn_theta(data, colors):
    # return the largest blue point
    blue_points = (x for x, c in zip(data, colors) if c == 'blue')
    return max(blue_points)
```

You could make the above into a one-liner, but I like the fact that the intermediate result is stored in `blue_points`; it makes the code more readable, in my opinion. Avoid trying to show off by writing one-liners. A lot of people can cram code into one line, but it takes thought and care to write clean, readable code.

All of these solutions are essentially the same, and run in linear time.

Now suppose that a small number of the red points are less than some blue points – that is, there is some overlap, as shown below. Assume for simplicity that the largest data point is red.



We wish to find a real number θ which "best" separates the blue points and red points. Clearly the points cannot be separated perfectly. Instead, we define a loss function $L(\theta)$ which counts the number of points which are on the wrong side of θ. More precisely:

$$L(\theta) = (\text{\# of red points } \leq \theta) + (\text{\# of blue points } > \theta)$$

The loss of the θ shown above is 2, since one red point is to the left of θ and one blue point is to the right. Our goal is to design an algorithm for finding a minimizer of $L(\theta)$. This is a simple instance of the machine learning task of *classification*.

b) Also in `min_ell_theta.py`, write a function named `compute_ell(data, colors, theta)` which takes in lists `data` and `colors` as described above, as well as a floating-point number, `theta`. It should return the loss at `theta` as a floating-point number. Your algorithm should have the best possible time complexity.

Solution: We can write an algorithm that takes time $\Theta(n)$. To compute $L(\theta)$, we loop through each of the data points, keeping track of the loss incurred. If the data point x is red, but $\leq \theta$, we increment the loss by one. If x is blue, but bigger than θ, we increment the loss by one. Each check takes constant time and there are $\Theta(n)$ points, so the function takes $\Theta(n)$ time in total.

```
def compute_ell(data, colors, theta):
    loss = 0
    for x, color in zip(data, colors):
        if (color == 'red' and x <= theta) or (color == 'blue' and x > theta):
            loss += 1
    return loss
```

c) Also in `min_ell_theta.py`, write a function named `minimize_ell(data, colors)` which takes in `data` and `colors` and returns a floating-point number which minimizes the loss L for that particular data set. Your algorithm should have quadratic time complexity. You may assume for simplicity that the smallest data point is blue[1].

Solution: Note that we can reduce the search space to a finite set , since it suffices to only check the θ which coincide with a data point. That is, we loop over each of the data points x and compute $L(x)$, returning the minimizer. Since computing $L(x)$ takes linear time, and there are $\Theta(n)$ data points, this approach takes $\Theta(n^2)$ time in total.

[1]Otherwise it is possible (given a special data set) for the loss to be minimized at some θ to the left of all of the data.

```
def minimize_ell(data, colors):
    minimum_loss = float('inf')
    minimum_theta = None
    for theta in data:
        loss = compute_ell(data, colors, theta)
        if loss < minimum_loss:
            minimum_loss = loss
            minimum_theta = theta
    return minimum_theta
```

d) Now assume that `data` is sorted (and `colors[i]` is the color of `data[i]`). In the file called `min_ell_theta.py`, write a function `minimize_ell_sorted(data, colors)` which returns a minimizer θ in linear time. Your code should satisfy the loop invariant: "After the αth iteration, `blue_gt_theta` is the number of blue points which are greater than `data`$[\alpha - 1]$."

For simplicity, suppose that exactly $n/2$ of the data points are `'red'`, and $n/2$ are `'blue'`.

Solution: We can use the knowledge that exactly half the data points are red and the other half are blue to track the amount of red elements and blue elements on both sides of θ on each iteration. Using the count of red elements before θ and blue elements after θ we can calculate loss in constant time. This approach takes $\Theta(n)$ time in total.

```
def minimize_ell_sorted(data, color):
    n = len(data)
    min_theta = data[0]
    min_loss = float('inf')
    red_seen = 0
    blue_seen = 0
    for i in range(len(data)):
        if color[i] == "red":
            red_seen += 1
        elif color[i] == "blue":
            blue_seen += 1
        blue_not_seen = n/2 - blue_seen
        loss = red_seen + blue_not_seen
        if loss < min_loss:
            min_loss = loss
            min_theta = data[i]
    return min_theta
```