CSCI 3104 Fall 2022
Instructors: Prof. Grochow and Nagesh

Problem Set 2

Due Date ..... September 12, 2022 8pm MT
Name ..... **Tyler Huynh**
Student ID **109603994**
Collaborators ..... **List Your Collaborators Here**

Contents

Instructions

- The solutions **should be typed**, using proper mathematical notation. We cannot accept hand-written solutions. Here's a short intro to LaTeX.
- You should submit your work through the **class Canvas page** only. Please submit one PDF file, compiled using this LaTeX template.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please submit this document with no fewer pages than the blank template (or Gradescope has issues with it).

- You are welcome and encouraged to collaborate with your classmates, as well as consult outside resources. You must **cite your sources in this document. Copying from any source is an Honor Code violation. Furthermore, all submissions must be in your own words and reflect your understanding of the material.** If there is any confusion about this policy, it is your responsibility to clarify before the due date.
- Posting to **any** service including, but not limited to Chegg, Reddit, StackExchange, etc., for help on an assignment is a violation of the Honor Code.
- You **must** virtually sign the Honor Code (see Section). Failure to do so will result in your assignment not being graded.

Honor Code (Make Sure to Virtually Sign)

Problem HC.
- My submission is in my own words and reflects my understanding of the material.
- Any collaborations and external sources have been clearly cited in this document.
- I have not posted to external services including, but not limited to Chegg, Reddit, StackExchange, etc.
- I have neither copied nor provided others solutions they can copy.

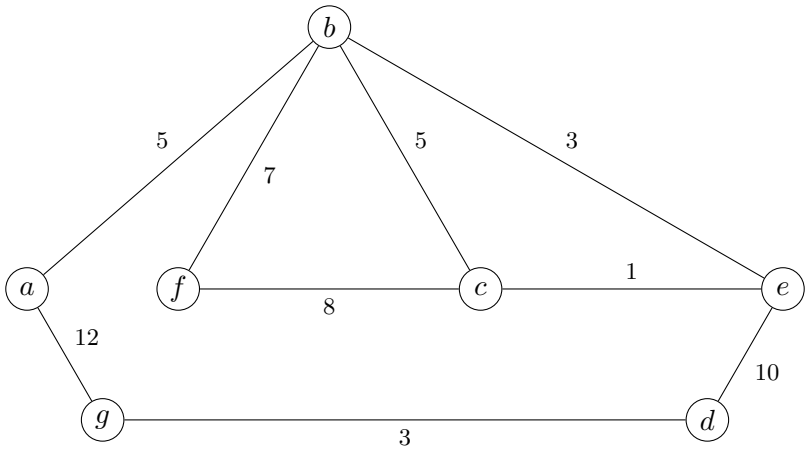
I agree to the above, Tyler Huynh. □

3 Standard 3 – Dijkstra's Algorithm

3.1 Problem 1

Problem 1. Consider the undirected weighted graph $G(V, E, w)$ pictured below. Work through Dijkstra's algorithm on the following graph, using the source vertex a. **Note:** In order to get full credits consider the following:

- Clearly include the contents of the priority queue, the distance from a and the parent of each vertex at each iteration.
- If you use a table to store the distances, clearly label the keys according to the vertex names rather than numeric indices (i.e., `dist['B']` is more descriptive than `dist['1']`).
- You do **not** need to draw the graph at each iteration, though you are welcome to do so. [This may be helpful scratch work, which you do not need to include.]
- Finally represent the shortest path graph.



Answer. Using Dijkstra's algorithm on this graph, starting from vertex "a", we will implement a priority queue as per what Dijkstra's algorithm does.

The algorithm itself:
We will first initialize a priority queue where the weights of each vertex will be set to infinity as there are no weights associated with it:

Parent:	null	null	null	null	null	null	null
Distances:	a	b	c	d	e	f	g
Weights:	∞	∞	∞	∞	∞	∞	∞

Priority Queue: []

We will now push the vertex of "a" as it is our starting vertex to our priority queue:

Parent:	null	null	null	null	null	null	null
Distances:	a	b	c	d	e	f	g
Weights:	∞	∞	∞	∞	∞	∞	∞

Priority Queue: [(a, 0)]

From the vertex of "a" we can see the vertices of both "b" and "g" with distances of 5 and 12 respectively. We will pop off the vertex of "a" and push onto the vertices of "b" and "g" and update the parents vertexes.

Parent:	null	a	null	null	null	null	a
Distances:	a	b	c	d	e	f	g
Weights:	0	∞	∞	∞	∞	∞	∞

Priority Queue: [(b, 5), (g, 12)]

From the vertex of "b" we can see the vertices of "c", "f", and "e". We will pop off the vertex of "b" and push onto our priority queue the vertices of "c", "f", and "e". The distances of the vertexes will be updated from that starting vertex of "a".

Parent:	null	a	b	null	b	b	a
Distances:	a	b	c	d	e	f	g
Weights:	0	5	∞	∞	∞	∞	∞

Priority Queue: [(e, 8), (c, 10), (f, 12), (g, 12)]

From the vertex of "e" we can see the vertex of "d" we will now pop off the vertex of "e" and push on the vertex of "d". The distances of the vertexes will be updated from that starting vertex of "a".

Parent:	null	a	b	e	b	b	a
Distances:	a	b	c	d	e	f	g
Weights:	0	5	∞	∞	8	∞	∞

Priority Queue: [(c, 9), (f, 12), (g, 12), (d, 18)]

Now we will pop off the vertex of "c", as there are now no more unvisited neighbor vertexes we will continue traversing through the graph using Dijkstra's algorithm. The distances of the vertexes will be updated from that starting vertex of "a".

Parent:	null	a	b	e	b	b	a
Distances:	a	b	c	d	e	f	g
Weights:	0	5	9	∞	8	∞	∞

Priority Queue: [(f, 12), (g, 12), (d, 18)]

Now we will pop off the vertex of "f", as there are now no more unvisited neighbor vertexes we will continue traversing through the graph using Dijkstra's algorithm. The distances of the vertexes will be updated from that starting vertex of "a".

Parent:	null	a	b	e	b	b	a
Distances:	a	b	c	d	e	f	g
Weights:	0	5	9	∞	8	12	∞

Priority Queue: [(g, 12), (d, 18)]

Now we will pop off the vertex of "g", since the distance from vertex "g" to vertex "d" is shorter than the distance from vertex "e" we will update the parent row and its respective weight. The distances of the vertexes will be updated from that starting vertex of "a".

Parent:	null	a	b	g	b	b	a
Distances:	a	b	c	d	e	f	g
Weights:	0	5	9	∞	8	12	12

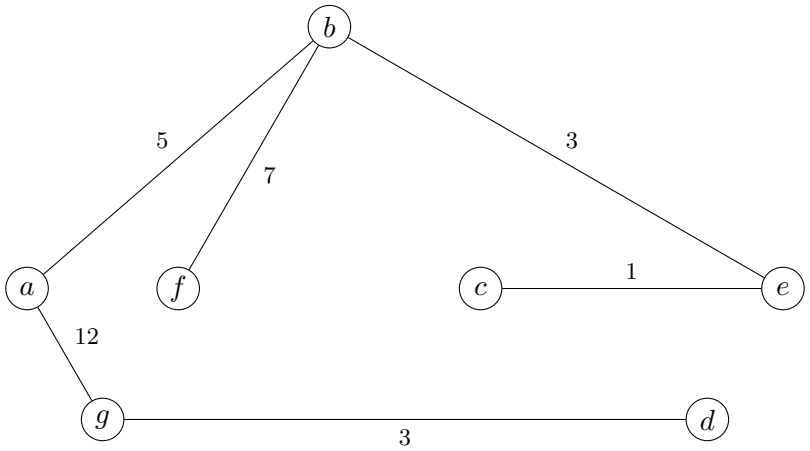
Priority Queue: [(d, 15)]

Now we will pop off the vertex of "d", since our priority queue is now empty, Dijkstra's algorithm is now done running.

Parent:	null	a	b	g	b	b	a
Distances:	a	b	c	d	e	f	g
Weights:	0	5	9	15	8	12	12

Priority Queue: []

I will now illustrate the shortest single path tree. To illustrate this I will delete the edges that will be considered in the SSPT.



□

3.2 Problem 2

Problem 2. You have three batteries, with capacities of 40, 25, and 16 Ah (Amp-hours), respectively. The 25 and 16-Ah batteries are fully charged (containing 25 Ah and 16 Ah, respectively), while the 40-Ah battery is empty, with 0 Ah. You have a battery transfer device which has a "source" battery position and a "target" battery position. When you place two batteries in the device, it instantaneously transfers as many Ah from the source battery to the target battery as possible. Thus, this device stops the transfer either when the source battery has no Ah remaining or when the destination battery is fully charged (whichever comes first).

But battery transfers aren't free! The battery device is also hooked up to your phone by bluetooth, and automatically charges you a number of dollars equal to however many Ah it just transfered.

The goal in this problem is to determine whether there exists a sequence of transfers that leaves exactly 10 Ah either in the 25-Ah battery or the 16-Ah battery, and if so, how little money you can spend to get this result. Do the following.

3.2.a Problem 2(a)

(a) Rephrase this is as a graph problem. Give a precise definition of how to model this problem as a graph, and state the specific question about this graph that must be answered. [**Note:** While you are welcome to draw the graph, it is enough to provide 1-2 sentences clearly describing what the vertices are and when two vertices are adjacent. If the graph is weighted, clearly specify what the edge weights are.]

Answer. This problem can be rephrased into a graph problem by the following:

The state of the batteries who's capacities are 40Ah, 25Ah, 16Ah represent the vertices in the graph.
The edges between the vertices within the graph represent the transfer of battery in Ah.
The edge weights of the edges represent the amount of battery in Ah that is being transferred to another battery and how much money we would spend.

Note: If two vertices are adjacent within the graph it represents that a transfer will occur within the graph.

The question for this graph would be from the Problem description, how would we be able to leave 10 Ah in either the 25 Ah or the 16 Ah battery while also using the least amount of money possible to achieve this. This would be a clear indication to use Dijkstra's algorithm as it is designed to find the shortest path possible, but in this case the shortest path would represent how little money we would use, thus the SSSPT would be a tree where we would use the least amount of money possible while also achieving our goal.

□

3.2.b Problem 2(b)

(b) Clearly describe an algorithm to solve this problem. If you use an algorithm covered in class, it is enough to state that. If you modify an algorithm from class, clearly outline any modifications. Make sure to explicitly specify any parameters that need to be passed to the initial function call. You need not write the algorithm.

Answer. For this problem we would utilize Dijkstra's algorithm where in the algorithm itself it uses a priority queue in order to store the distances from our starting vertex. From here, it will traverse through each neighbor vertex of the graph and mark the current vertex as visited and continue traversing through the graph until there are no more unvisited vertices left.

In the context of this problem Dijkstra's algorithm would be the most optimal algorithm as it would find the shortest path from the source vertex to the target vertex. This algorithm would require modifications as there is the case that multiple target batteries where only $10Ah$ remain in either the 25Ah or 16 Ah battery, could exist such that we must be able to find the shortest distance from the source vertex as possible. The algorithm from here would then take these different paths to the target vertices where only $10Ah$ remain in either the 25Ah or 16 Ah battery and compare the amount of Ah has been transferred and the amount of money that was used in order to find the shortest path, thus the cheapest option. These modifications would run after Dijkstra's original algorithm is done traversing through the graph.

Description of modifications:
1. The first step in the modifications would be to traverse through all the vertices that we have visited and checking if either the 25 Ah or 16 Ah has only 10 Ah left in the battery, this would be our target nodes.
2. The next step would be to traverse through these nodes further in order to find the ones with the shortest distance using the distance array, thus making it our final target node.
3. Finally, we would finally return the target node that has the shortest path as per Dijkstra's algorithm and my own modifications.

□

3.2.c Problem 2(c)

(c) Apply that algorithm to the question. Report and justify your answer. Here, justification includes the sequences of vertices visited and the total cost. **Note:** For full credits make sure to draw the graph with all the battery states. Then apply the algorithm that you have described until the desired final states are reached.

Answer. I will now run through my algorithm to find a target, we will start with an empty distance array and an empty priority queue as per Dijkstra's specifications. We will have a starting vertex of s where it will have a tuple of {0, 25, 16} with the 1st, 2nd, and 3rd position representing the empty 40 Ah, 25Ah, and 16 Ah battery respectively. You can see a visual representation below:

We will initialize an empty table of the distances of each vertex from the source and their distance values as well.

Vertex:	s
Distances from Source:	0

Priority Queue: $[(s, 0)]$

From here we will then visit the neighbors of s after popping it off of our priority queue with those being a and b, who's tuple looks like {16, 25, 0} and {25, 0, 16} respectively. We will now push a and b to our priority queue and update their distances

Vertex:	s
Distances from Source:	0

Priority Queue: $[(a, 16), (b, 25)]$

We will pop a off of our priority queue and visit its neighbors of c and d. It's tuples will look like this {40, 1, 0} and {16, 9, 16}. We will now update our priority queue and our table.

$$\text{dist}[c] = \text{dist}[a] + w[a \to c] = 16 + 24 = 40$$
$$\text{dist}[d] = \text{dist}[a] + w[a \to d] = 16 + 16 = 32$$

Vertex:	s	a
Distances from Source:	0	16

Priority Queue: $[(b, 25), (d, 32), (c, 40)]$

We will pop b off of our priority queue and visit its neighbors of e and f. It's tuples will look like this {40, 0, 1} and {25, 16, 0}, we will now update our priority queue and our table.

$$\text{dist}[e] = \text{dist}[b] + w[b \to e] = 15 + 25 = 40$$
$$\text{dist}[f] = \text{dist}[b] + w[b \to f] = 25 + 16 = 41$$

Vertex:	s	a	b
Distances from Source:	0	16	25

Priority Queue: $[(d, 32), (c, 40), (e, 40), (f, 41)]$

We will pop d off of our priority queue and visit its neighbors of g. This is $d's$ only neighbor because any other combination of tuples for d is a vertex we have already visited. It's tuples will look like this {32, 9, 0}, we will now update our priority queue and our table.

$$\text{dist}[g] = \text{dist}[d] + w[d \to g] = 32 + 16 = 48$$

Vertex:	s	a	b	d
Distances from Source:	0	16	25	32

Priority Queue: $[(c, 40), (e, 40), (f, 41), (g, 48)]$

We will pop c off of our priority queue and visit its neighbors of e and h. e is a neighbor of c because there exists no other combinations such that c would not have it as it's neighbor. It's tuples will look like this {40, 0, 1} and {24, 1, 16}, we will now update our priority queue and our table.

$\text{dist}[e] = \text{dist}[c] + w[c \to e] = 40 + 1 = 41 > 40$ (We will not update E because 41 would make the distance larger)

$$\text{dist}[h] = \text{dist}[c] + w[c \to h] = 40 + 16 = 46$$

Vertex:	s	a	b	d	c
Distances from Source:	0	16	25	32	40

Priority Queue: $[(e, 40), (f, 41), (h, 46, (g, 48))]$

We will pop e off of our priority queue and visit its neighbors of i. e would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {15, 25, 1}, we will now update our priority queue and our table.

$$\text{dist}[i] = \text{dist}[e] + w[e \to i] = 40 + 25 = 65$$

Vertex:	s	a	b	d	c	e
Distances from Source:	0	16	25	32	40	40

Priority Queue: $[(f, 41), (h, 46), (g, 48), (i, 65)]$

We will pop f off of our priority queue and visit its neighbors of j. f would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {9, 16, 16}, we will now update our priority queue and our table.

$$\text{dist}[j] = \text{dist}[f] + w[f \to j] = 41 + 16 = 57$$

Vertex:	s	a	b	d	c	e	f
Distances from Source:	0	16	25	32	40	40	41

Priority Queue: $[(h, 46), (g, 48), (j, 57), (i, 65)]$

We will pop h off of our priority queue and visit its neighbors of k. h would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {24, 17, 0}, we will now update our priority queue and our table.

$$\text{dist}[k] = \text{dist}[h] + w[h \to k] = 46 + 16 = 62$$

Vertex:	s	a	b	d	c	e	f	h
Distances from Source:	0	16	25	32	40	40	41	46

Priority Queue: $[(g, 48), (j, 57), (k, 62), (i, 65)]$

We will pop g off of our priority queue and visit its neighbors of l. g would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {32, 0, 9}, we will now update our priority queue and our table.

$$\text{dist}[l] = \text{dist}[g] + w[g \to l] = 48 + 9 = 57$$

Vertex:	s	a	b	d	c	e	f	h	g
Distances from Source:	0	16	25	32	40	40	41	46	48

Priority Queue: $[(j, 57), (l, 57), (k, 62), (i, 65)]$

We will pop j off of our priority queue and visit its neighbors of m. j would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {9, 25, 7}, we will now update our priority queue and our table.

$$\text{dist}[m] = \text{dist}[j] + w[j \to m] = 57 + 9 = 66$$

Vertex:	s	a	b	d	c	e	f	h	g	j
Distances from Source:	0	16	25	32	40	40	41	46	48	57

Priority Queue: $[(l, 57), (k, 62), (i, 65), (m, 66)]$

We will pop l off of our priority queue and visit its neighbors of n. l would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {7, 25, 9}, we will now update our priority queue and our table.

$$\text{dist}[n] = \text{dist}[l] + w[l \to n] = 57 + 25 = 82$$

Vertex:	s	a	b	d	c	e	f	h	g	j	l
Distances from Source:	0	16	25	32	40	40	41	46	48	57	57

Priority Queue: $[(k, 62), (i, 65), (m, 66), (n, 82)]$

We will pop k off of our priority queue and visit its neighbors of o. k would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {8, 17, 16}, we will now update our priority queue and our table.

$$\text{dist}[o] = \text{dist}[k] + w[k \to o] = 62 + 16 = 78$$

Vertex:	s	a	b	d	c	e	f	h	g	j	l	k
Distances from Source:	0	16	25	32	40	40	41	46	48	57	57	62

Priority Queue: $[(i, 65), (m, 66), (n, 82), (o, 88)]$

We will pop i off of our priority queue and visit its neighbors of p. i would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {15, 10, 16}, we will now update our priority queue and our table. As we can see this is a possible target node as the 25 Ah battery only has 10 Ah left in it.

$$\text{dist}[p] = \text{dist}[i] + w[i \to p] = 65 + 15 = 80$$

Vertex:	s	a	b	d	c	e	f	h	g	j	l	k	i
Distances from Source:	0	16	25	32	40	40	41	46	48	57	57	62	65

Priority Queue: $[(m, 66), (p, 80), (n, 82), (o, 88)]$

We will pop m off of our priority queue and m will have no neighbors because there are no combinations of tuples that we have not visited that is valid.

Vertex:	s	a	b	d	c	e	f	h	g	j	l	k	i	m
Distances from Source:	0	16	25	32	40	40	41	46	48	57	57	62	65	66

Priority Queue: $[(p, 80), (n, 82), (o, 88)]$

We will pop p off of our priority queue and r. p would only have one neighbor as there are no other combinations of tuples that we have not visited that would be valid. It's tuple will look like this {31, 10, 0}, we will now update our priority queue and our table. As we can see this is a possible target node as the 25 Ah battery only has 10 Ah left in it.

$$\text{dist}[r] = \text{dist}[p] + w[p \to r] = 80 + 16 = 96$$

Vertex:	s	a	b	d	c	e	f	h	g	j	l	k	i	m	p
Distances from Source:	0	16	25	32	40	40	41	46	48	57	57	62	65	66	80

Priority Queue: $[(n, 82), (o, 88), (r, 96)]$

As we can see this algorithm could search this tree for a very long time, such that we would run out of letters to use. We can stop the use of Dijkstra's algorithm here as anything higher than vertex p would not be the shortest path as specified by Dijkstra's algorithm. This is because everything that is still on the priority queue is greater than 80, meaning that even if we were to find another target vertex it wouldn't matter, since we have already found the shortest path. Such that our solution, with consideration of the weights would be:

The Shortest Path:

$$s : (0, 25, 16)(0)$$
$$\downarrow$$
$$b : (25, 0, 16)(25)$$
$$\downarrow$$
$$e : (40, 0, 1)(15)$$
$$\downarrow$$
$$i : (15, 25, 1)(25)$$
$$\downarrow$$
$$p : (15, 10, 16)(15)$$

$s(0) + b(25) + e(15) + i(25) + p(15) = 80$ dollars from the source vertex of s

□

4 Standard 4 – Examples Where Greedy Algorithms Fail

4.3 Problem 3

Problem 3. Recall the Interval Scheduling problem, where we take as input a set of intervals $\mathcal{I}$. The goal is to find a maximum-sized set $S \subseteq \mathcal{I}$, where no two intervals in S intersect. Consider the greedy algorithm where we place all of the intervals of $\mathcal{I}$ into a priority queue, ordered earliest start time to latest start time. We then construct a set S by adding intervals to S as we poll them from the priority queue, provided the element we polled does not intersect with any interval already in S.

Provide an example with at least 5 intervals where this algorithm fails to yield a maximum-sized set of pairwise non-overlapping intervals. Clearly specify both the set S that the algorithm constructs, as well a larger set of pairwise non-overlapping intervals.

You may explicitly specify the intervals by their start and end times (such as in the examples from class) or by drawing them. **If you draw them, please make it very clear whether two intervals overlap.** You are welcome to hand-draw and embed an image, provided it is legible and we do not have to rotate our screens to grade your work. Your justification should still be typed. If you would prefer to draw the intervals using LaTeX, we have provided sample code below.

Answer. In this Interval Scheduling problem, I will first illustrate the greedy algorithm such that we will start from the earliest start time to the latest start time. This algorithm would fail as it would not maximize the amount of intervals that would be present within the priority queue. Thus, truly creating a minimum where there can only be "x" number of intervals present.

The Greedy Solution:
Let S represent a set of intervals that will be added to from our priority queue where our priority queue is sorted from earliest start time to the latest start time, such that:

Interval:	a	b	c	d	e	f
Time:	$1:00-4:00$	$2:00-3:00$	$2:15-3:45$	$3:15-3:45$	$3:45-4:30$	$4:30-5:30$

The algorithm will first pop the interval of "a" and add it to the set of S since there are no overlaps present, such that:

$$\text{Set } S : \{[1:00-4:00]\}$$

The algorithm will not pop off the interval of "b" because there exists an overlap with the interval of "a".
The algorithm will not pop off the interval of "c" because there exists an overlap with the intervals of "a, b".
The algorithm will not pop off the interval of "d" because there exists an overlap with the intervals of "a, b, c".
The algorithm will not pop off the interval of "e" because there exists an overlap with the intervals of "a, b, c, d".

The algorithm will pop of the interval of "f" because there exists no overlap and add it to the set of S, such that:

$$\text{Set } S : \{[1:00-4:00],[4:30-5:30]\}$$

As you can see the set of S is not the optimal solution because it does not accurately represent a maximum-sized set of pairwise non-overlapping intervals. Such that it does not maximize the amount of intervals within the set of S. □

The Most Optimal Solution: Below I will show the most optimal solution where we will sort the priority queue from earliest end time to latest end time.

Let T represent a set of intervals that will be added to from our priority queue where our priority queue is sorted from earliest end time to latest end time, such that:

Interval:	a	b	c	d	e	f
Time:	$2:00-3:00$	$2:15-3:45$	$3:15-3:45$	$1:00-4:00$	$3:45-4:30$	$4:30-5:30$

The algorithm will first pop the interval of "a" and add it to the set of T since there are no overlaps present, such that:

$$\text{Set } T : \{[2:00-3:00]\}$$

The algorithm will not pop off the interval of "b" because there exists an overlap with the interval of "a".

The algorithm will pop of the interval of "c" because there exists no overlap and add it to the set of T, such that:

$$\text{Set } T : \{[2:00-3:00], [3:15-3:45]\}$$

The algorithm will not pop off the interval of "d" because there exists an overlap with the interval of "a, b, c".

The algorithm will pop of the interval of "e" because there exists no overlap and add it to the set of T, such that:

$$\text{Set } T : \{[2:00-3:00], [3:15-3:45], [3:45-4:30]\}$$

The algorithm will pop of the interval of "e" because there exists no overlap and add it to the set of T, such that:

$$\text{Set } T : \{[2:00-3:00], [3:15-3:45], [3:45-4:30], [4:30-5:30]\}$$

As you can see from the above solution we will have the most optimal solution such that we are now maximizing the amount of intervals we have present within our set of T. We can see that this is the most optimal solution as the priority queue is now sorted from the earliest end time to the latest end time, allowing for us to have a greater amount of intervals within the queue.

4.4 Problem 4

Problem 4. Consider now the Weighted Interval Scheduling problem, where each interval i is specified by

$$([\text{start}_i, \text{end}_i], \text{weight}_i).$$

Here, the weight is an assigned value that is independent of the length $\text{end}_i - \text{start}_i$. Here, you may assume $\text{weight}_i > 0$. We seek a set S of pairwise non-overlapping intervals that maximizes $\sum_{i \in S} \text{weight}_i$. That is, rather than maximizing the number of intervals, we are seeking to maximize the sum of the weights.

Consider a greedy algorithm which works identically as in Problem 3. Draw an example with at least 5 appointments where this algorithm fails. Show the order in which the algorithm selects the intervals, and also show a subset with larger weight of non-overlapping intervals than the subset output by the greedy algorithm. The same comments apply here as for Problem 3 in terms of level of explanation.

Answer. For the weighted interval scheduling problem we will implement the same greedy algorithm where the priority queue will be sorted by earliest start time to the latest start time. Then we will disprove this by using a more optimal solution. Such that we maximize the sum of the weights within our set.
The Greedy Algorithm:

Interval:	a	b	c	d	e	f
Time:	$1:00-4:00$	$2:00-3:00$	$2:15-3:45$	$3:15-3:45$	$3:45-4:30$	$4:30-5:30$
Weights:	4	5	2	3	9	3

The algorithm will first pop the interval of "a" and add it to the set of S with the weight of 4 attached since there are no overlaps present, such that:

$$\text{Sum of set } S : \{[1:00-4:00]\} = 4$$

□

The algorithm will not pop off the interval of "b" because there exists an overlap with the interval of "a".
The algorithm will not pop off the interval of "c" because there exists an overlap with the intervals of "a, b".
The algorithm will not pop off the interval of "d" because there exists an overlap with the intervals of "a, b, c".
The algorithm will not pop off the interval of "e" because there exists an overlap with the intervals of "a, b, c, d".

The algorithm will pop off the interval of "f" because there exists no overlap and add it to the set of S along with it corresponding weight, such that:

$$\text{Sum of set } S : \{[1:00-4:00], [4:30-5:30]\} = 7$$

We can see from the above algorithm that the sum of set S is equal: $3 + 4 = 7$. This does not allow for us to maximize the sum of the weights from the set of S.

The Most Optimal Solution: For the most optimal solution that would yield us with the largest sum within the set. We will order the priority queue from the earliest end time to the latest end time.

Let T represent a set of intervals that will be added to from our priority queue where our priority queue is sorted from earliest end time to latest end time, such that:

Interval:	a	b	c	d	e	f
Time:	$2:00-3:00$	$2:15-3:45$	$3:15-3:45$	$1:00-4:00$	$3:45-4:30$	$4:30-5:30$
Weights:	4	5	2	3	9	3

The algorithm will first pop the interval of "a" and add it to the set of T since there are no overlaps present along with its corresponding weight, such that:

$$\text{Set } T : \{[2:00-3:00]\} = 4$$

The algorithm will not pop off the interval of "b" because there exists an overlap with the interval of "a".

The algorithm will pop of the interval of "c" because there exists no overlap and add it to the set of T along with its corresponding weight, such that:

$$\text{Set } T : \{[2:00-3:00], [3:15-3:45]\} = 4+2$$

The algorithm will not pop off the interval of "d" because there exists an overlap with the interval of "a, b, c".

The algorithm will pop of the interval of "e" because there exists no overlap and add it to the set of T along with its corresponding weight, such that:

$$\text{Set } T : \{[2:00-3:00], [3:15-3:45], [3:45-4:30]\} = 4+2+9$$

The algorithm will pop of the interval of "e" because there exists no overlap and add it to the set of T along with its corresponding weight, such that:

$$\text{Set } T : \{[2:00-3:00], [3:15-3:45], [3:45-4:30], [4:30-5:30]\} = 4+2+9+3$$

We can see from the above algorithm that the sum of set T is equal: $4+2+9+3 = 18$. This does not allow for us to maximize the sum of the weights from the set of T.
We can see from this solution that it will allow for us to have a maximized sum of weights within the set of T. Clearly, this shows that the greedy algorithm from problem 3 is not the most optimal solution as the sum of the set $S = 7 < T = 18$.

5 Standard 5 – Exchange Arguments

5.5 Problem 5

Problem 5. Recall the Making Change problem, where we have an infinite supply of pennies (worth 1 cent), nickels (worth 5 cents), dimes (worth 10 cents), and quarters (worth 25 cents). We take as input an integer $n \geq 0$. The goal is to make change for n using the fewest number of coins possible.

Prove that in an optimal solution, we use at most 2 dimes.

Proof. Referenced Levet notes to solve this problem.
Proof By Contradiction
Non - Optimal Solution:
The least optimal solution would be one that would have more than two dimes. Consider:

$$d > 2$$

If the value of dimes were to be higher than two dimes than it would not become optimal.
Let "j" be an arbitrary natural number and let "r" be some arbitrary natural number, such that where 3 represents the amount of coins:

$$d = 3j + r$$

For this proof we will prove that we use at most two dimes.

Optimal Solution:
We will now exchange this for a more optimal solution that uses at most 2 dimes.
Such that we will exchange $3j$ dimes for an arbitrary amount of j quarters and j nickels.
We will then have a total of $2j$ combination of quarters and nickels., such that:

$$d = 2j + r$$

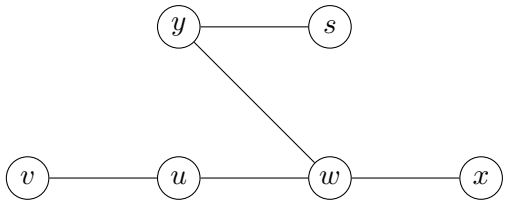
From the above proof we can see that $2j < 3j$, thus we see that we have reached the most optimal solution such that we are using at most 2 dimes.

□

5.6 Problem 6

Problem 6. Let $G = (V, E)$ be a graph. A *vertex cover* of G is a set of vertices C such that for every edge $uv \in E$, either $u \in C$ or $v \in C$. That is, every edge of G has at least one endpoint in C.

(a) Let T be the graph



Find a vertex cover C of T that includes the vertex v. Now, explain why the set $C' = (C \setminus \{v\}) \cup \{u\}$ obtained from your C by removing v and adding (if it is not already present) u is also a vertex cover of T.

(b) Now prove this property in general. That is, let T be an arbitrary tree, suppose that C is a vertex cover of T, and suppose that C contains a leaf vertex v. Let u be the unique neighbor of v in T, and let $C' = (C\setminus\{v\})\cup\{u\}$ be the set of vertices obtained from C by obtained from C by removing v and adding (if it is not already present) u. Carefully explain why C' is a vertex cover of T.

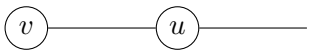
Answer. **Part A:**
A vertex cover for the graph T is one that will cover all edges on the graph, such that:

Let C be the set of vertices who's edges cover the entire graph:
$C\{\text{w, y, v}\}$

If we were to remove the vertex of v and add the vertex of u it would still be a vertex cover of T because due to the vertex of u being in the middle of the vertexes of v and w it still allows for a complete cover of the graph T. The vertex cover with v removed: $C\{\text{w, y, u}\}$

Part B:
I will now prove this property that C' is a vertex cover for an arbitrary tree T such that v is a leaf node and u is it's neighbor. If the vertex cover for the graph T has either the vertices of u or v in it then it would be a valid vertex cover. Take for example:
Assume C' is not a vertex cover such that:

We will first start off with the valid vertex cover C who's arbitrary graph S is:



Where u is connected to a random vertex such that it allows for a valid vertex cover of C for the graph of S.
We will now remove the vertex of v from the vertex cover where the vertex cover of C is no longer a valid vertex cover, since neither the vertexes of u or v are no longer covered.
We will now add in the vertex of u into our vertex cover to then connect the vertex of v again allowing for a valid vertex cover.
Now both the vertices u and v are covered by our vertex cover.

Thus, we have reached a contradiction, as since all the vertices in the example graph of S is still covered even if we remove the vertex of v, since v has only one neighbor of u it must still be a valid vertex cover if we add in the vertex of u.
Thus, C' is a valid vertex cover.

□