Written Assignment 2

CS 362

February 28, 2022

Damian Franco

1 Problem 1

***Q*uestion:**

An alphabet contains letters A, B, C, D, E, F. The frequencies of the letters are 35%, 20%, 15%, 15%, 8%, and 7%. We know that the Huffman algorithm always outputs an optimal prefix-free code. However, this code is not always unique (obviously we can, e.g., switch 0's with 1's and get a different code – but, for some inputs, there are two optimal prefix-free codes that are significantly different). For the purposes of this exercise, we consider two Huffman codes to be different if there exists a letter for which one of the codes assigns a shorter code-word than the other code.

(i) Trace the Huffman algorithm and construct two different Huffman codes for the above input.

(ii) Compute the expected number of bits per character (i.e., the expected code-word length) for both codes.

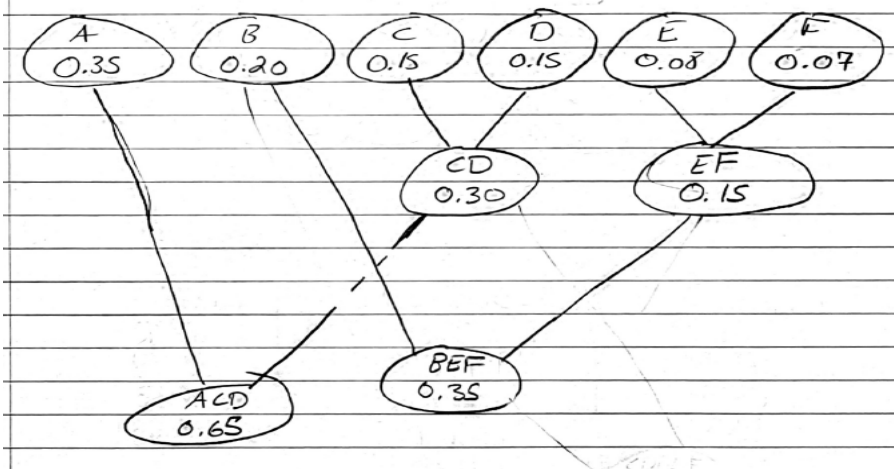
(iii) Does there exist a prefix-free code with smaller expected code-word length? Reason your answer in a sentence or two, or provide such prefix-free code.

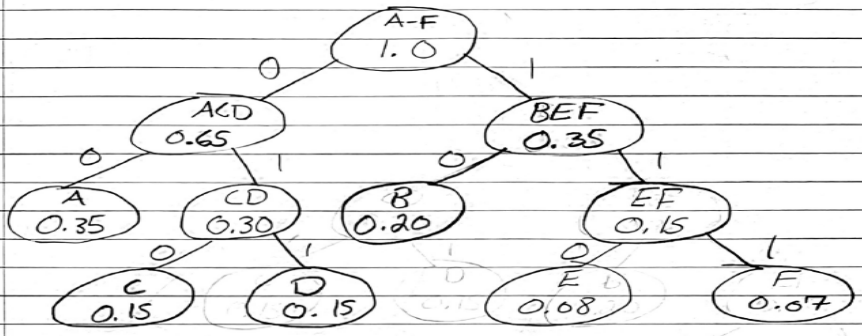
***S*olution(s):**

This problem asks us to use the Huffman algorithm to construct two optimal prefix-free codes. These two codes are significantly different due to the length of each code-word which results in a different score for each code. Score of the code determines the most optimal code for the input. The work below shows hand written work that includes the discovery of the two different Huffman generated codes, the length of each code-word for the two codes and also expanding on any other prefix-free codes that can be generated. The second code that was generated results in the shortest code-word length of 1, but also has two code-words that have the length of 4. The first code generated has the least overall length between both codes. Through my findings, no code can be found with a smaller expected code-word length then the first and second codes found below. This is due to the fact that both of these codes (1 & 2) result in the same score of 2.45. In the last image in this section, a list of all the possible codes and the lengths of each code-word are generated through the technique shown to us during lecture. Using a tree of all the possible binary value lengths allowed for more insight on all the possible codes that could be generated. All scores for each possible code is also found and helps determine which codes are the most optimal and which ones are not. The first two codes found are the most optimal and in fact have the same score. Overall, choosing both the original codes found are the most optimal and the user can not go wrong by choosing either codes.

Huffman Code #1

Character	Frequency
A	0.35
B	0.20
C	0.15
D	0.15
E	0.08
F	0.07

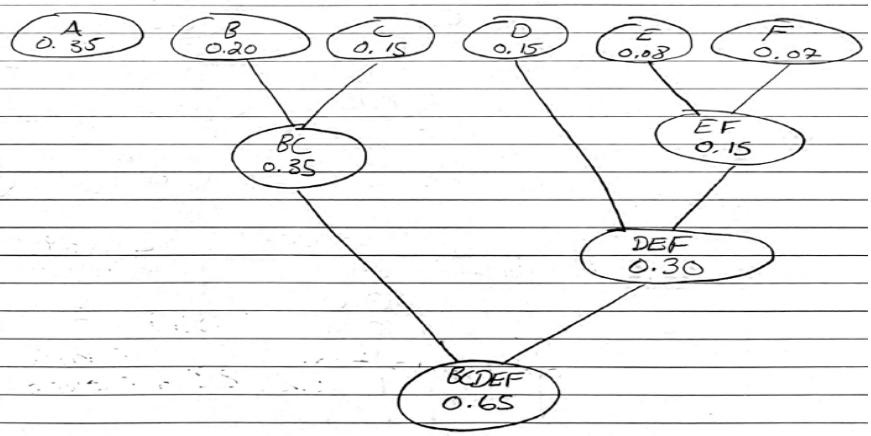


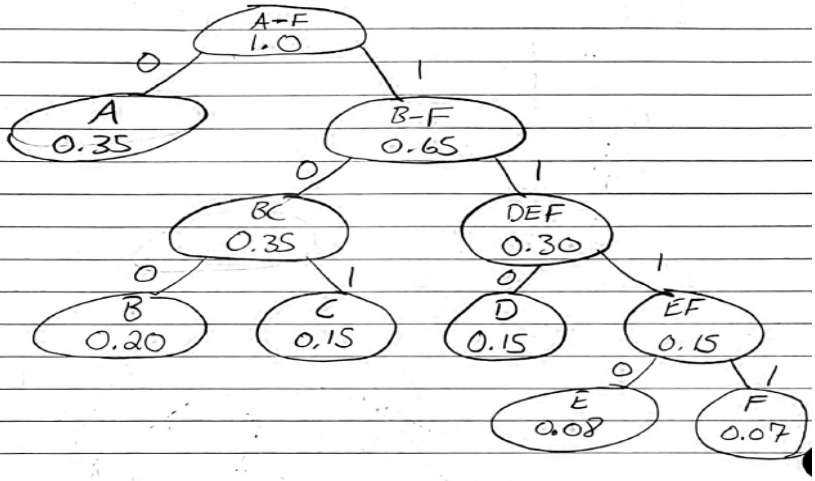


Characters	Code-Word	Length
A	0 0	2
B	1 0	2
C	0 1 0	3
D	0 1 1	3
E	1 1 0	3
F	1 1 1	3

Huffman Code #2

Character	frequency
A	0.35
B	0.20
C	0.15
D	0.15
E	0.08
F	0.07





Character	Code-Word	Length
A	0	1
B	100	3
C	101	3
D	110	3
E	1110	4
F	1111	4

Code #1

Character	freq	Code-Word	Length
A	0.35	00	2
B	0.20	10	2
C	0.15	010	3
D	0.15	011	3
E	0.08	110	3
F	0.07	111	3

Code #2

Character	freq	Code-Word	length
A	0.35	0	1
B	0.20	100	3
C	0.15	101	3
D	0.15	110	3
E	0.08	1110	4
F	0.07	1111	4

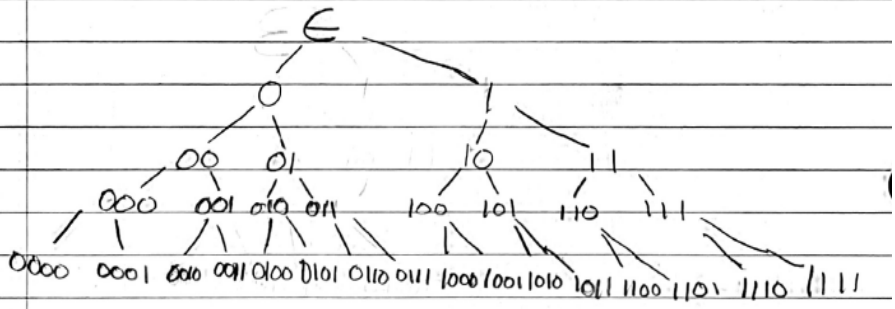
Code1 score = $0.35(2) + 0.20(2) + 0.15(3) + 0.15(3) + 0.08(3) + 0.07(3)$
$= 2.45$

Code2 score = $0.35(1) + 0.20(3) + 0.15(3) + 0.15(3) + 0.08(4) + 0.07(4)$
$= 2.45$

Code1 score = Code2 score

Possible lengths:

Character	ℓ_1	ℓ_2	ℓ_3	ℓ_4	ℓ_5
A	1	2	3	1	4
B	3	2	3	2	4
C	3	3	3	4	4
D	3	3	3	4	4
E	4	3	3	4	4
F	4	3	3	4	4



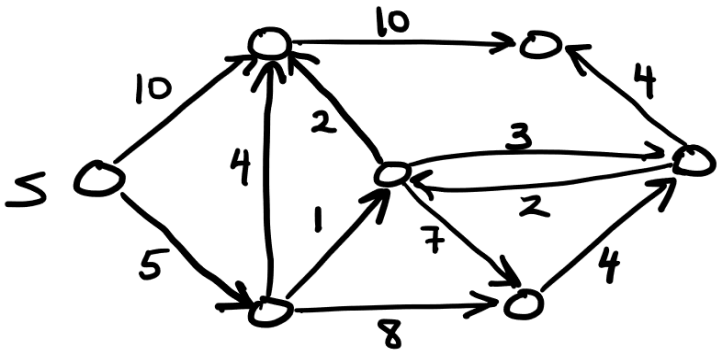
ℓ_1 score = 2.45
ℓ_2 score = 2.45
ℓ_3 score = 3.00
ℓ_4 score = 2.55
ℓ_5 score = 5.00

ℓ_1 & ℓ_2 are the best choice for codeword lengths.

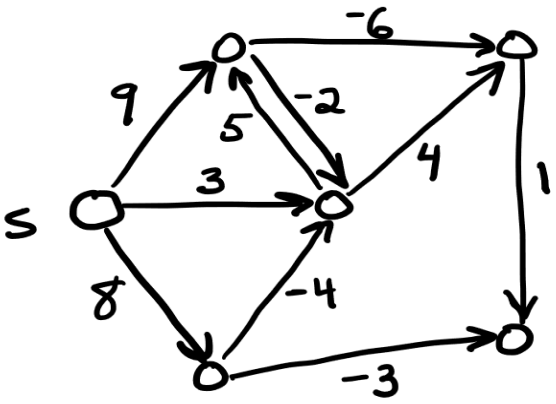
2 Problem 2

Question:
See the graphs drawn below and answer the following:

(a) By hand, trace the execution of both Dijkstra's algorithm and the Bellman-Ford algorithm on the example graph. Show your work! Did you get the same output? Which way took you less time? Which way seemed easier?



(b) Answer the same questions, but for the second graph.



***S*olution(s):**

For this question, we were given two directed graphs (a) and (b) and we were assigned to find the shortest path using Dijkstra's and Bellman-Ford algorithms. Main difference between the two graphs is the graph in part (a) consists of no negative edge weight values and the graph in part (b) does contain negative edge values. Dijkstra's algorithm is known to fail when negative edge weights are present in the graph. Bellman-Ford algorithm will be able to compute the shortest path correctly with negative edge weights present in the graph. By finding the shortest path using both algorithms, we can be able to find if both outputs are the same and if Dijkstra's algorithm will fail with the graph in part (b).

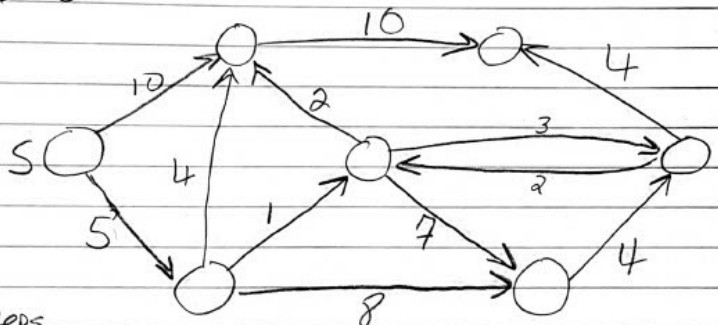
In part (a), the graph consisted of no negative edge values. Both the Dijkstra's and Bellman-Ford algorithm should produce the same shortest path output. The result in the work below shows that I was able to produce the same output for both algorithms, which was what I anticipated. Since no negative edge weights occur within this graph, both algorithm will have no issue computing the right output. The algorithm that took less time to do by hand was the Bellman-Ford algorithm. Reason is because Dijkstra's algorithm required much more copying of the graph when traversing through the algorithm. Although the Bellman-Ford algorithm is faster, I would say Dijkstra's algorithm was fairly easier to follow the steps and simple to fully understand what was happening.

In part (b), the graph did contain some negative edge values. I anticipated Dijkstra's algorithm not to work under the circumstances that negative weight exist. I was proven right by my work. First, I found the shortest path through Dijkstra's and I was able to find an output. Unsure if the output found was the correct one, I then used the Bellman-Ford output to find the shortest path and found that was a different path found by Dijkstra's algorithm. This was not surprising because Dijkstra's algorithm is known to fail when negative edge weights exist in the graph. All the edges were visited and an output was found, but since Dijkstra's algorithm stops after all nodes were visited, the process stops and the incorrect output is generated. Bellman-Ford is able to continuously check all nodes and edges to retrace if the path is truly minimized. For this graph, I believe that Bellman-Ford is faster, but Dijkstra's (even though the wrong solution was found) is simpler.

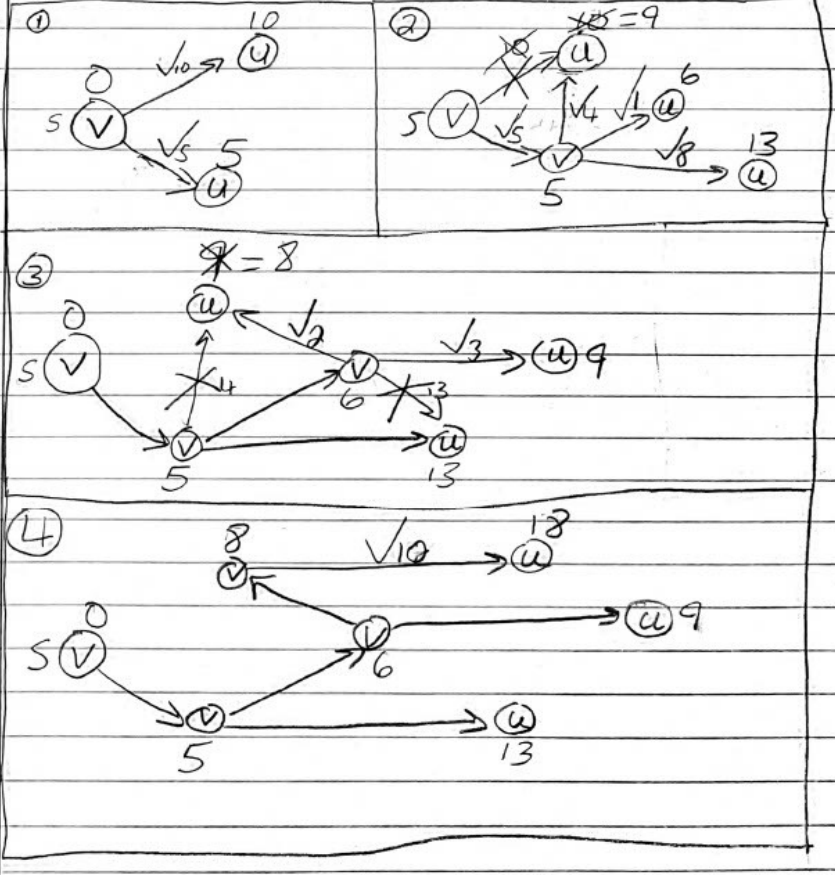
Part (a)

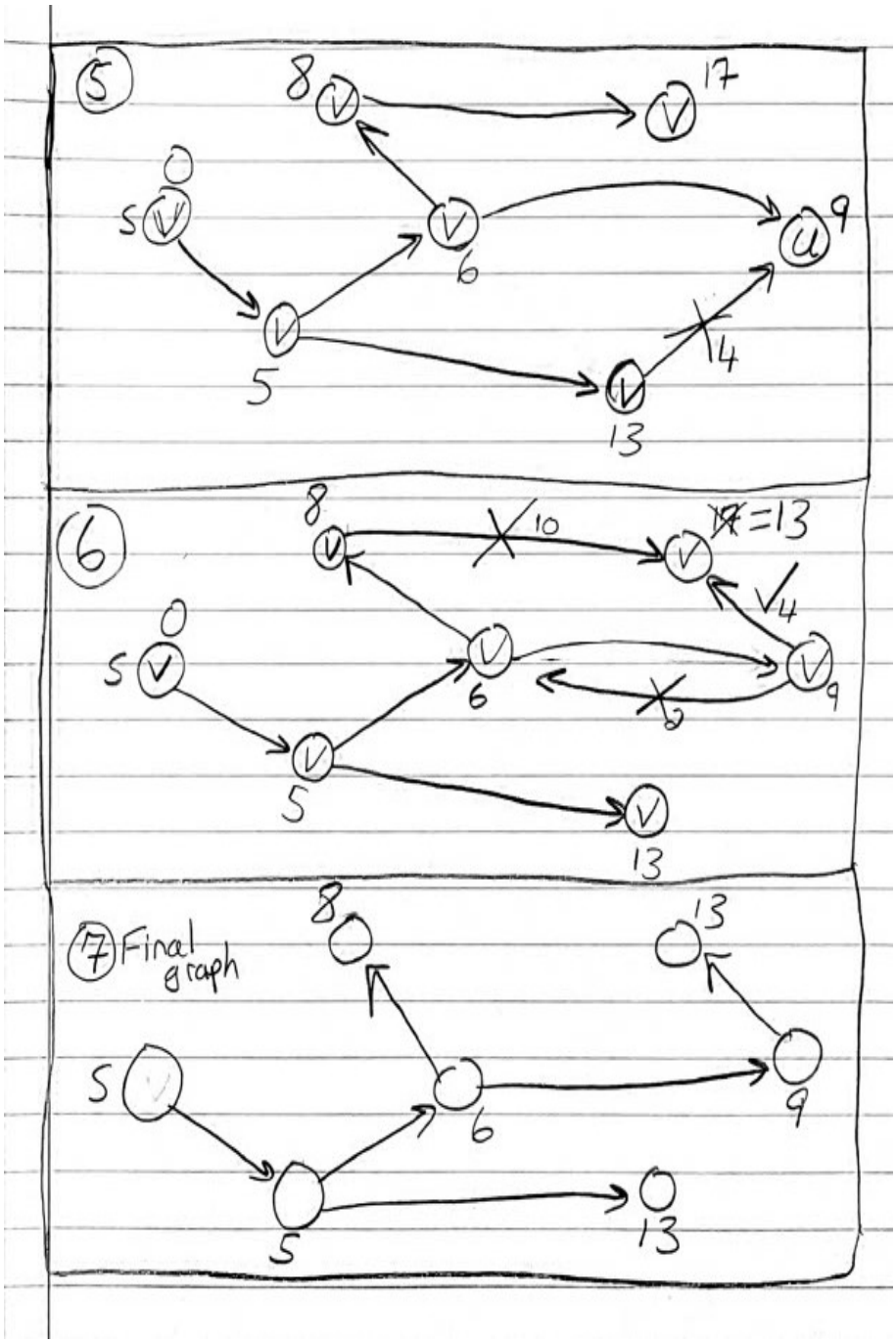
11

Dijkstra's Algorithm



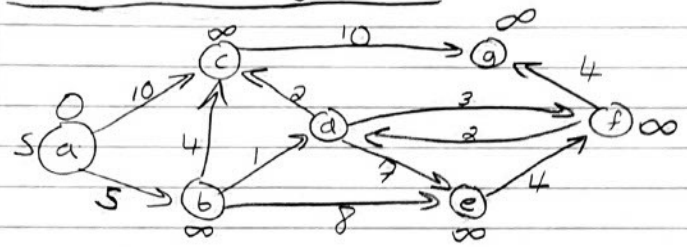
steps




5
8
17
0
s
6
9
5
4
13
6
8
10
=13
0
s
6
9
5
13
7 Final graph
8
13
s
6
9
5
13

Part (a)

Bellman-Ford Algorithm



$|V| = 7$

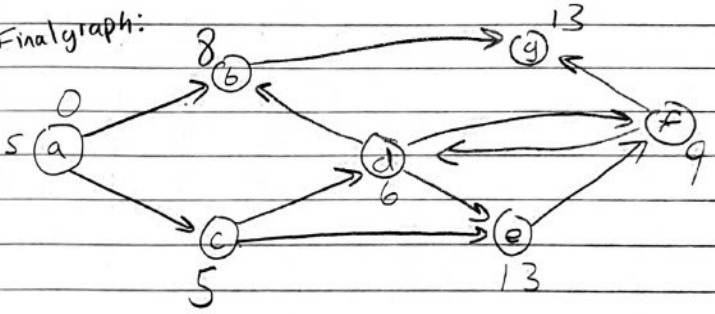
$|V| - 1 = 6$

List of edges: (f, g), (e, f), (f, d), (d, c), (d, f), (c, g), (d, e), (b, d), (b, e), (b, c), (a, b), (a, c)

a	b	c	d	e	f	g
0	∞	∞	∞	∞	∞	∞
0	5	10	∞	∞	∞	∞
0	5	9	6	13	∞	∞
0	5	8	6	13	9	∞
0	5	8	6	13	9	18
0	5	8	6	13	9	13
0	5	8	6	13	9	13

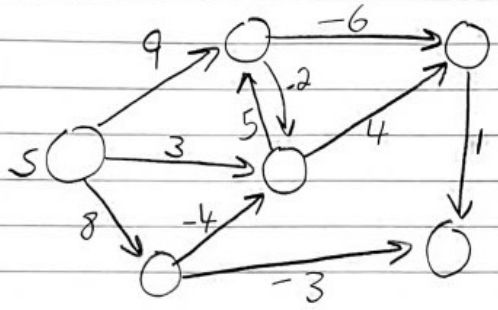
← check for cycles

Final graph:

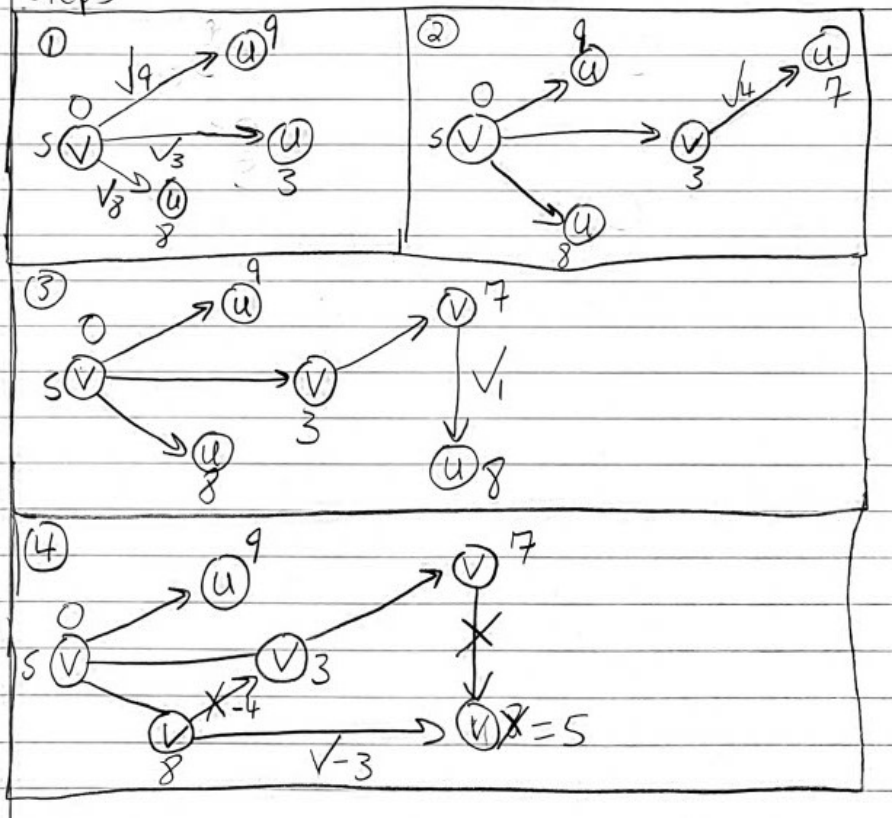


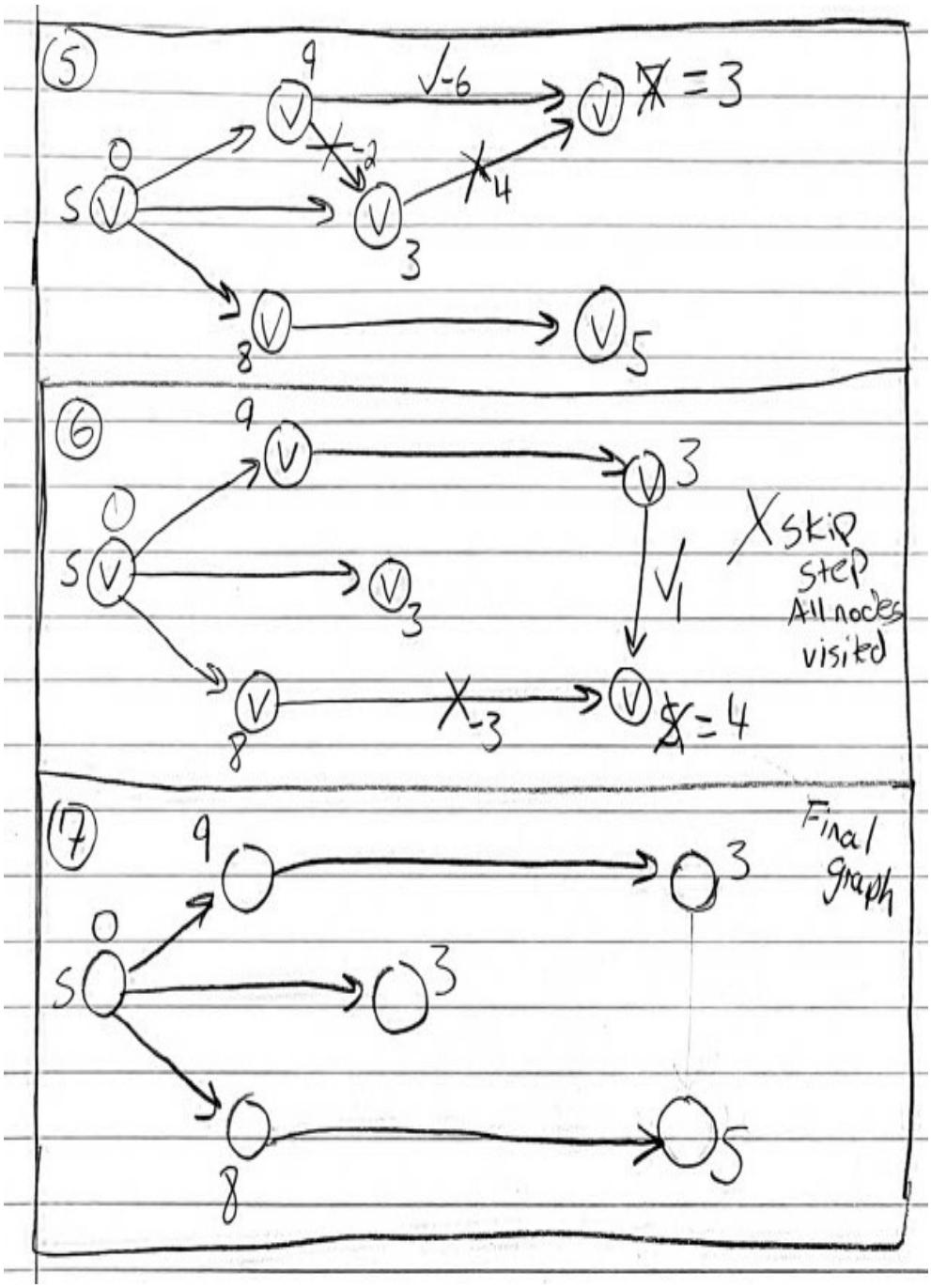
Part (b)

Dijkstra's Algorithm



Steps

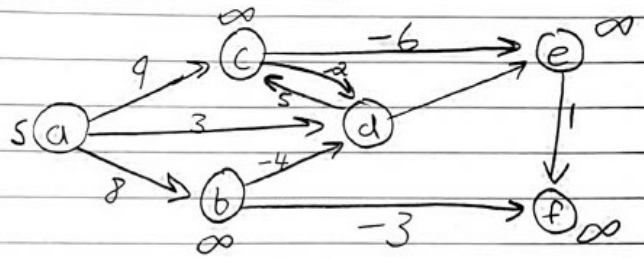



(5)
9
√-6
X=3
0
S
-2
X4
3
8
5
(6)
9
3
0
S
3
√1
X skip step All nodes visited
8
X-3
X=4
(7)
Final graph
9
3
0
S
3
8
5

Part (6)

1 1

Bellman-Ford Algorithm



$|V| = 6$

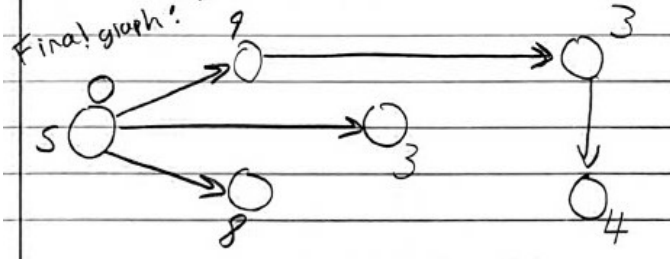
$|V| - 1 = 5$

List of edges: (b, f), (b, d), (d, e), (e, f), (c, e), (c, d), (d, c), (a, b), (a, d), (a, c)

a	b	c	d	e	f
0	∞	∞	∞	∞	∞
0	8	3	9	∞	∞
0	8	3	9	∞	5
0	8	3	9	3	5
0	8	3	9	3	4
0	8	3	9	3	4

← check for cycles

Final graph:



3 Problem 3

***Q*uestion:**

We know that when we have a graph with negative edge costs, Dijkstra's algorithm is not guaranteed to work.

(a) Does Dijkstra's algorithm ever work when we have a graph with negative edge costs? Explain why or why not.

(b) Find an algorithm that will always find a shortest path between two nodes, under the assumption that at most one edge in the input has a negative weight. Your algorithm should run in time O(m log n), where m is the number of edges and n is the number of nodes. That is, the running time should be at most a constant factor slower than Dijkstra's algorithm.

To be clear your algorithm takes as input

(i) a directed graph, G, given adjacency list form.

(ii) a weight function *f*, which, given two adjacent node, *v, w*, returns the weight of the edge between the. For non-adjacent nodes *v, w*, you may assume *f(v, w)* returns $+\infty$

(iii) a pair of nodes, *s, t*

If the input contains a negative cycle, you should find one and output it. Otherwise, if the graph contains at least one directed path from *s* to *t*, you should output a shortest path. Otherwise, output "No path."

(c) Can you generalize your idea to graphs with two negative edges? Three? What happens to the running time as the number of edges increases?

***S*olution(s):**

(a) A path will be generated by Dijkstra's algorithm when a negative weight occurs but that path will not be the shortest path, which is why it fails. Dijkstra's algorithm is unable to work when a negative weight is in the graph that the algorithm is applied to because the algorithm does not have any looping capabilities to double check for shorter paths. Each node is considered "unvisited" at the beginning of the algorithm and the algorithm ends when all nodes are considered "visited", which is why it fails. After all nodes are "visited", the algorithm will then need to loop over each node another time to find a if a shorter path exists. Dijkstra's algorithm is unable to loop back and ends at a first path found, which is not the shortest path. This is the main reason why it fails.

(b) Dijkstra's algorithm is a terrific greedy algorithm and by modifying it, there can be a possibility for the algorithm to work with at most one edge that has negative weight. This algorithm will take a directed graph *G*, a weight function *f* and a pair of start and end nodes *s, t* as the input. If there exists at least one directed path from the start node *s* to *t*, then the algorithm will

output that shortest path found. If a negative cycle exists in the input graph G, then the algorithm should find the cycle and output the cycle that appears. Lastly, if the algorithm fails to find a path from s to t, then it will return "no path."

To start, the algorithm will begin at the s node and start to traverse through the graph by the directed edges connecting each node. Nodes will be assigned a "cost" which is based on the edge weight from the incoming directed edges. The neighbors of the current node s will be chosen and traversed through and the current node will move to a neighbor node with the minimum cost. The previous node will be marked as visited and these steps will repeat until all of the nodes are visited and the last node visited was node t. To account for the negative weight, the algorithm will run through all the steps of the algorithm again. This will allow the path found to correctly account for the negative weight and produce the "real" shortest path between nodes s and t. The algorithm is somewhat similar to the Bellman-Ford Algorithm in the sense that it will run over each edge in multiple iterations to find the correct output. After a path from s to t is found after the second run of the algorithm steps, then that path will be considered the shortest path and be the output. To account for negative cycles, the algorithm can run another time and if the path changes and continues to minimize the overall cost, then a negative cycle is present. This negative cycle that is found will then output. That is the high-level overview of how the algorithm works.

Run time of this algorithm will be *O(mlog(n))* because the basis of the algorithm is build off of Dijskra's. Dijskra's algorithm run time is *O(nlog(n))* and since we are doing multiple iterations of the algorithm then the first n in the run time could be swapped with a value of m, where it equals to the number of times the algorithm loops.

(c) To generalize the idea for the multiple negative weights then the constant looping of the algorithm must be done. In fact, the algorithm must loop $n + 1$ times where n is equal to the number of negative weights that appear in the graph. This would account for all the negative edges and should be able to output the correct shortest path, but by constantly running the algorithm multiple times, the running time will become nothing short of horrendous. There is no great way to make the running time better, but by using an algorithm like Bellman-Ford, the run-time of the algorithm should remain better then the algorithm that I created.

Pseudocode for the algorithm is shown on the next page.

Algorithm 1 Pseudocode for the Dijkstra's algorithm with at most one negative edge. Also accounts for negative cycles.

```
G ← Set of all nodes within the graph
Visited ← Set of all visited nodes
ShortPath ← Set of the quickest path nodes found
CurrNode ← Start node indicated by user
Add start node to ShortPath
i = 0 ← Iteration number
for i <= 2 do
    while all nodes in G != Visited do
        NextNode = neighbor of CurrNode with at least one edge between itself
        and the the min cost between all neighbors
        Add NextNode to ShortPath
        Add CurrNode to Visited
        CurrNode = NextNode
    end while
    Increment i by 1
    if i == 2 then
        SecondSP = ShortPath
    end if
end for
if SecondSP != ShortPath then
    return ShortPath ← Negative cycle occurs
else if SecondSP == ShortPath then
    return SecondSP ← Shortest path is found
else
    return "No path"
```