Algorithms Supp 3 work

Dijkstra

1. i) Lines 13-17 are just finding the vertex with the least distance from s that we've computed so far.

ii) Not sure about line 15 since first iteration would be distance[v] < distance[None] which seems like it should cause an error?

Assuming distance[None] = ∞ then we will always set u to be the vertex with the lowest computed distance so far.

We could have comparisons to None on line 15 since initially all vertices except s have distance None and they are all in to_process.

iii) ~~We will never try to remove None on line 17 since~~

We could try to remove None on line 17 if there are vertices unreachable from s since then at some point we will have explored all reachable vertices, and there will still be some vertices in to_process. All these vertices will have distance = None since they never pass the if on line 20.

So seems like algorithm will have errors?

2. All edge weights must be ≥ 0.
If not we could have negative weight cycles and that could cause the algorithm not to terminate.

3. $O(V^2)$
line 12 while loop happens V times
line 14 and 15 loops happen V times also on each iteration of the while loop.

4. First we'll prove the assertations on lines 9 & 10.

Assert. 9: Suppose the assertation fails at some point in execution. Let v be the vertex at which this happens and T be the time.
Consider a mathematical shortest path from s to v.

$$s = u_1 \to u_2 \to \dots u_k \to \dots v$$

We have 2 cases to consider. First if one of the vertices hasn't been popped yet from by time T. Say u_k is the first vertex that hasn't been popped yet. Then,

$$\begin{aligned} \text{distance}(s \text{ to } v) &< v.\text{distance} \\ &\leq u_k.\text{distance} \quad (\text{since } v \text{ and } u_k \text{ were in} \\ &\leq u_{k-1}.\text{distance} + \text{cost}() \quad \text{priority queue}). \\ &= \text{distance}(s \text{ to } u_{k-1}) + \text{cost}(u_{k-1} \to u_k) \\ &\leq \text{distance}(s \text{ to } v) \end{aligned}$$

so we have a contradiction.

4. The other case is if ~~we have~~ all vertices have been popped by time T.

distance(s to u) < v.distance
$\leq$ distance(s to u_{k-1}) + cost(u_{k-1} to v)
= distance(s to v)

So another contradiction.
Therefore assertion a cannot fail.

Assertion 10:
On line 8 a vertex v is popped when it ~~is the shortest~~ has the shortest path from s in the queue.
Assertion a guarantees that v.distance is the true shortest distance at this point. On line 18 a vertex is only pushed if we find a shorter path to it. Since this is impossible by assertion a, v is never pushed back into to explore.

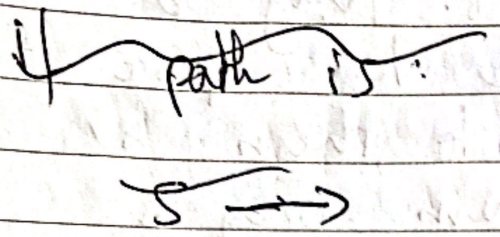
Now we can finish the proof.
Firstly it terminates since vertices are never pushed back into to explore so the while loop can only iterate V times.
When it terminates all distances are correct since when we pop a vertex it has the correct distance and we will pop all vertices as long as they are reachable from s.

4. Considering lines 17 and 18. They are there to push vertices into to explore when we find a path to them for the first time. Without them only s would ever be explored.
if we removed them and changed line 5 then the ~~assertions would no longer hold~~ the assertions would still hold. the only difference is instead of adding vertices when we first find a path we'd add them initially with distance = ∞ and just decrease whenever we find a shorter path.
It differs from algorithm 1 as it still only iterates over each vertices neighbours and not all vertices at each iteration of the while loop.

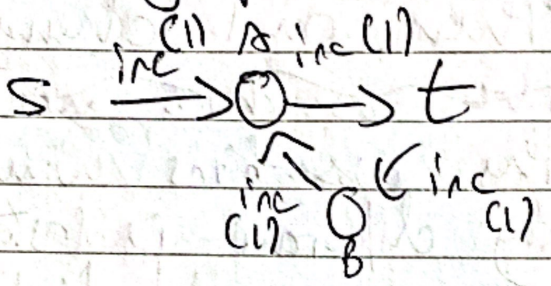
5. O(E+VlogV)
loop in line 11 — line 8 has cost logV and happens V times.

Example sheet

3.



if residual graph is:



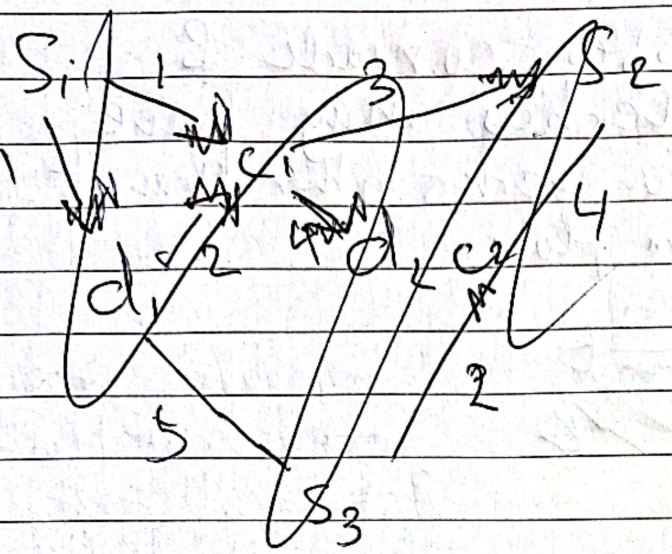
then path is $S \to A \to t \to B \to A \to t$

Edge $A \to t$ can only increase by 1 but since it's in the path twice it will be incremented 2 times on line 36.

the assumption is that the path should have no cycles.

Let p be $S \to u_1 \to u_2 \to u_3 \to \cdots \to u_{k-1} \to u_k = t$ be our augmenting path.
From our assumption each edge is only in one time, Since δ is the ~~smallest~~ value that the smallest changable edge can be changed by, when we change all other edges their flow will still be ≥ 0 and $\leq$ capacity. Since they are only incremented by δ once we have a valid flow.

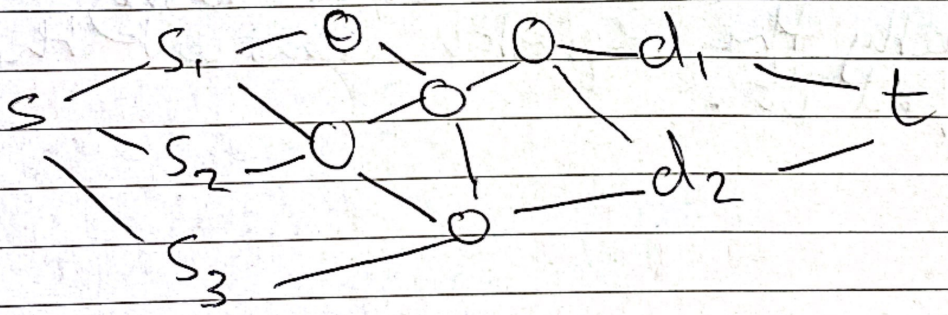
5.



Create a sink vertex and connect it to all the supply vertices and give the edges capacity s_v.

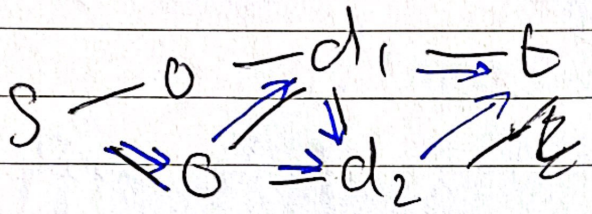
~~Do the~~

Do the same for a sink vertex t by connecting it to all demand vertices and give all edges capacity d_v.



Solve the max flow of this graph. If the flow going into t is equal to the sum of all demand vertices then we have met the demands.

7. My solution satisfies comrade B since demand vertices may have outgoing flow to a vertex other than t in the maximum flow.



If blue arrows are max flow d_1 will get more than ~~d~~ necessary.

To satisfy comrade A we just remove any outgoing edges from our helper demand vertices in graph.

then we may not be able to meet the demands even if we could before but if it's possible for every demand vertex to ~~only~~ receive exactly the requested amount then we will find it.