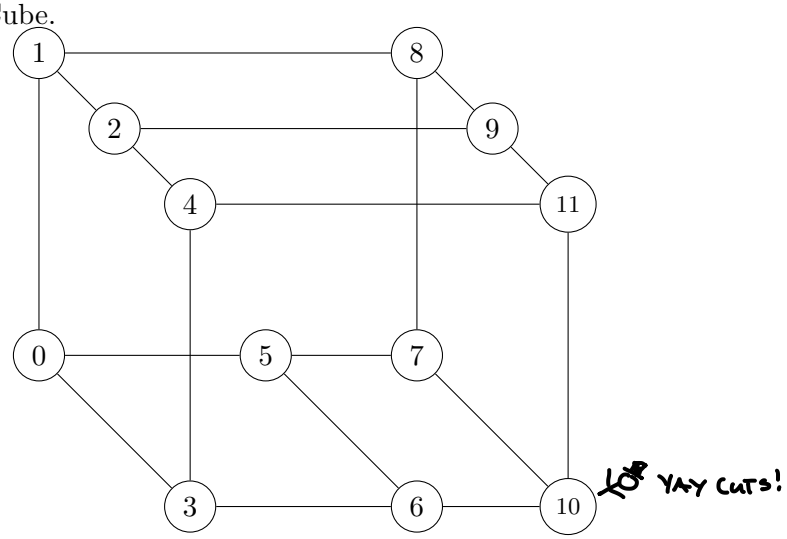
CSC 404 - Homework 5 - Name:

Problem 1 (Min Cut and Monte Carlo)**.** Consider the following graph/network - this beast is called Bidiaki's Cube.

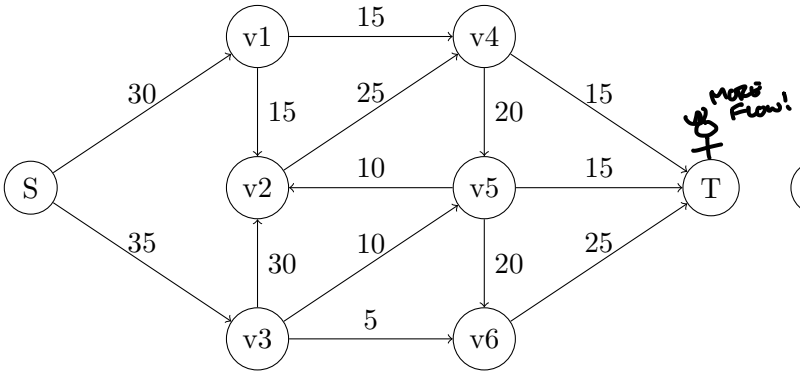


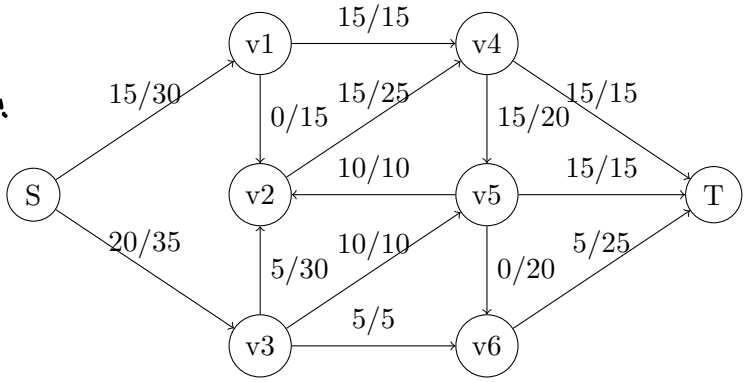
a. Run the Simple Algorithm (Karger's Algorithm) on this graph to determine an edge-cut for this graph. Show the steps/contractions :-).

b. Run the Contraction/Simple Algorithm $M = n^2 \ln(1000) = 20^2 \ln(1000) \approx 2763$ times and record the smallest cut observed (and two corresponding regions). This is (probably) the minimum cut – with probability of success of 99.999%. (Did your run(s) come up with an edge cut with this size?)

c. Run the Contraction/Simple Algorithm 10 times. How many of these runs resulting in the minimum cut? What about 100, 1000, 10000,... times?

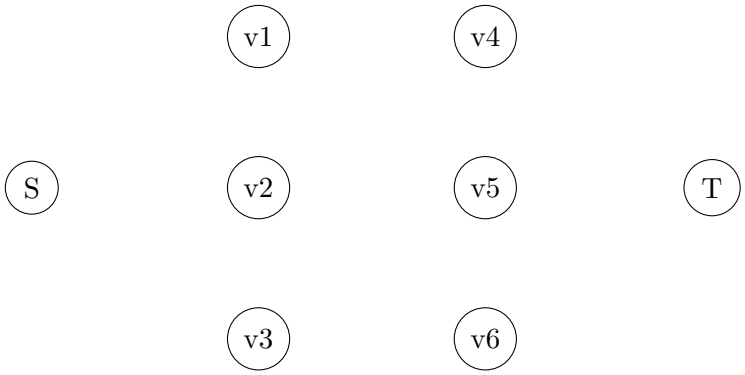
Problem 2 (Ford Fulkerson – One More Time :-)). Consider the following flow network, with current flow f:





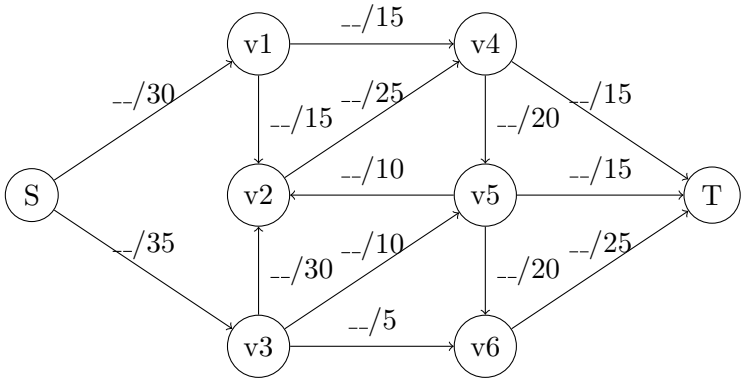
a. Show this is a valid flow. That is, confirm the in-flow = out-flow at all non source/sink locations.

b. Sketch out the resulting residual network

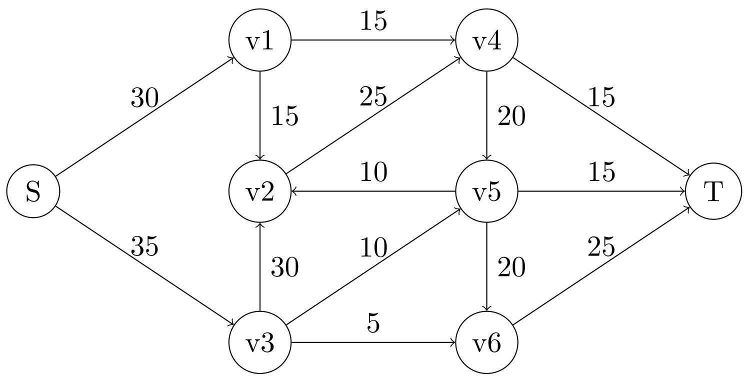


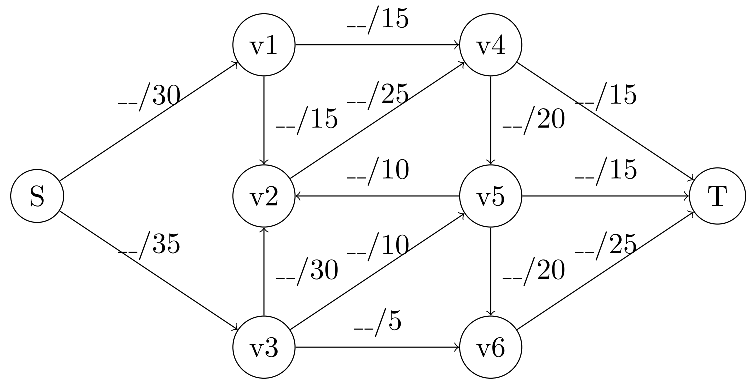
c. Within the residual network there (should) be a path $s \to v_3 \to v_2 \to v_5 \to v_6 \to t$. What is the value of additional resources (i.e., $|f'|$) we can carry along this flow, f'? (Note – this path will make use of a red/negative path to 'undo' a poor decision in an earlier step)

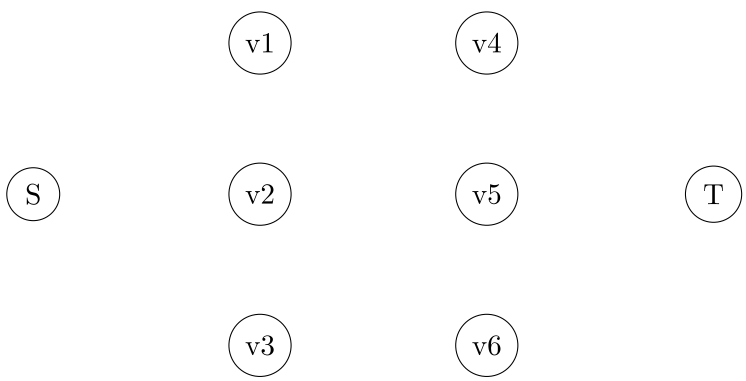
d. Sketch out the resulting augmentation of flow f by f'.

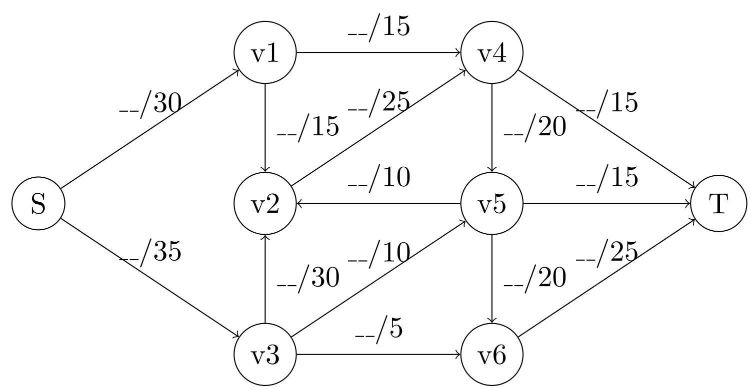


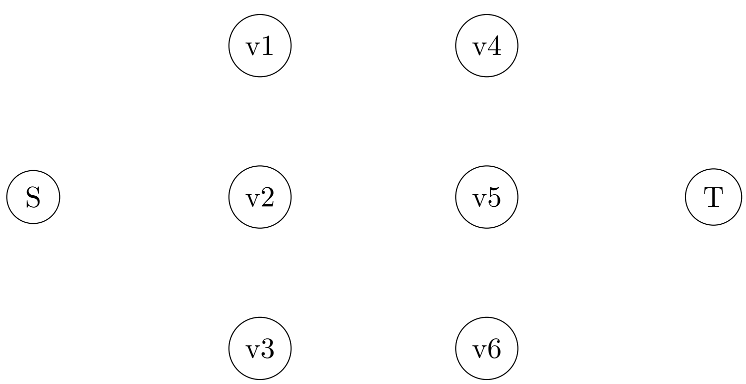
Problem 3. Our goal in this problem is to run the full Ford-Fulkerson Algorithm. We start with a flow of zero and begin with the following residual network (top left network). At each step identify a path within the residual network, construct the augmentation of flow (use blank flow graphs on the right column), and construct the resulting residual network (use blank network in the left column). Keep this process going until you do not have path within the residual network.

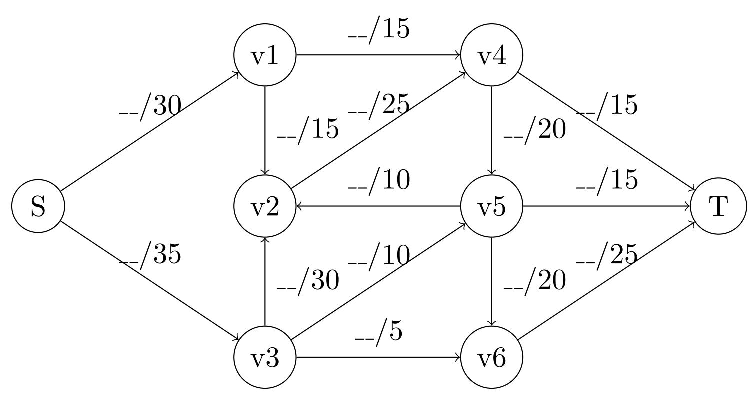


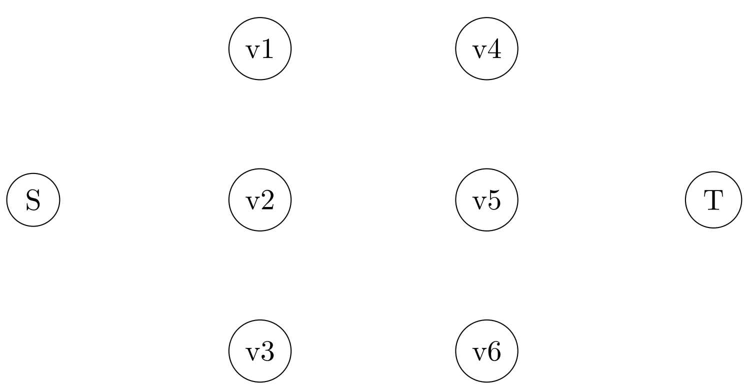


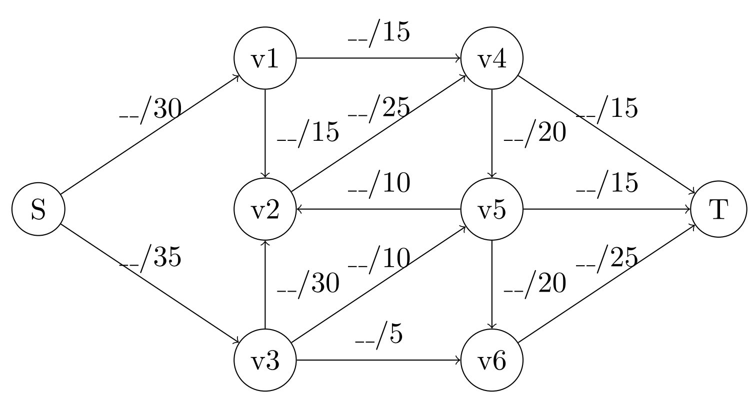




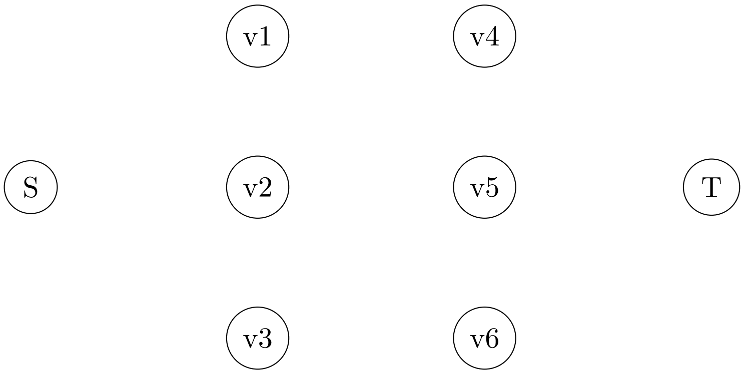


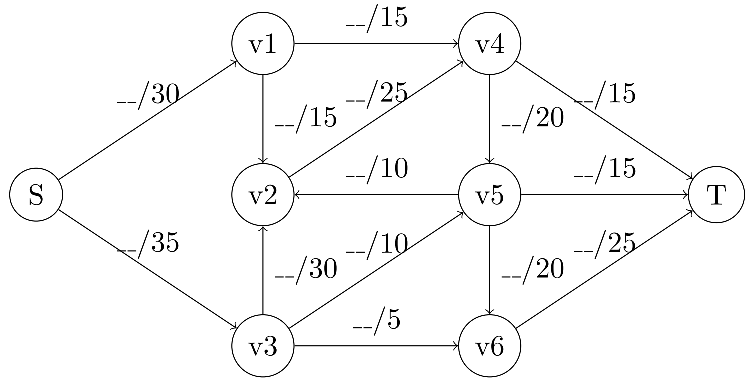


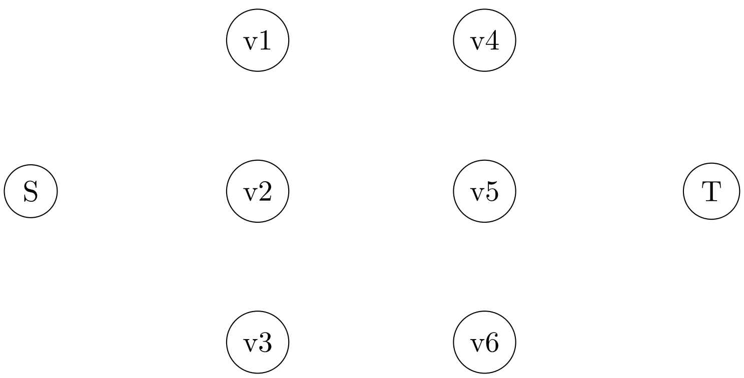


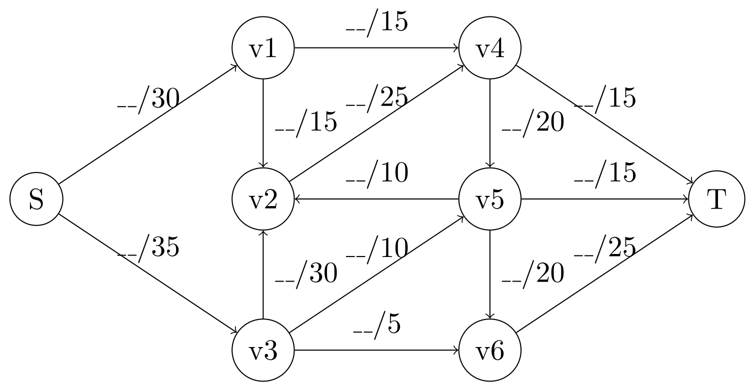


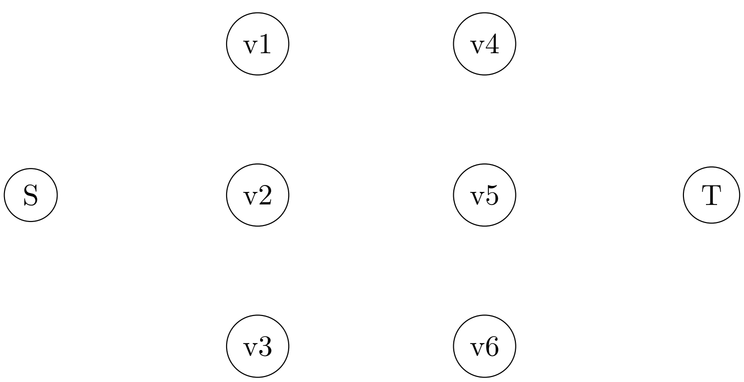
Extra space if you need a few more blank graphs. Or...You can try running it again (consider different scenarios of how you went about selecting your path. Did you use a greedy approach, optimal approach, random approach,...)

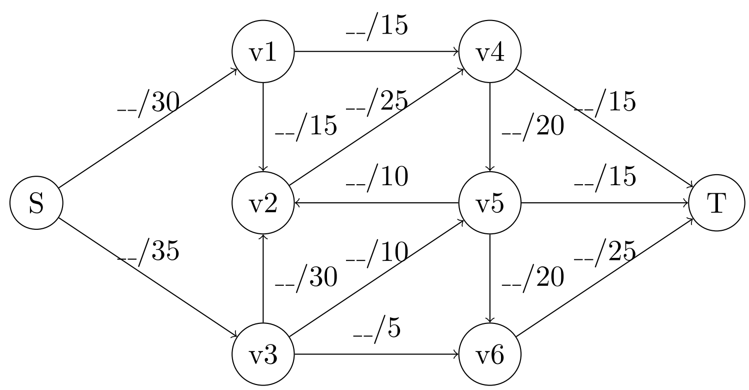


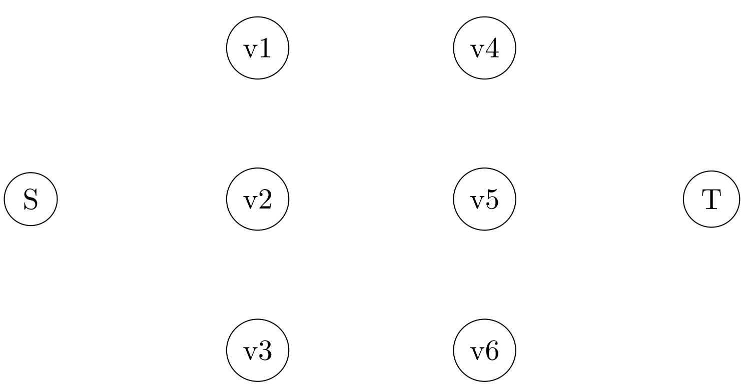


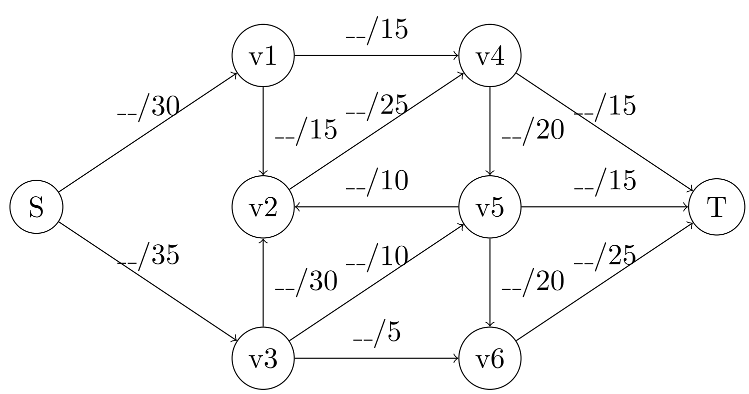






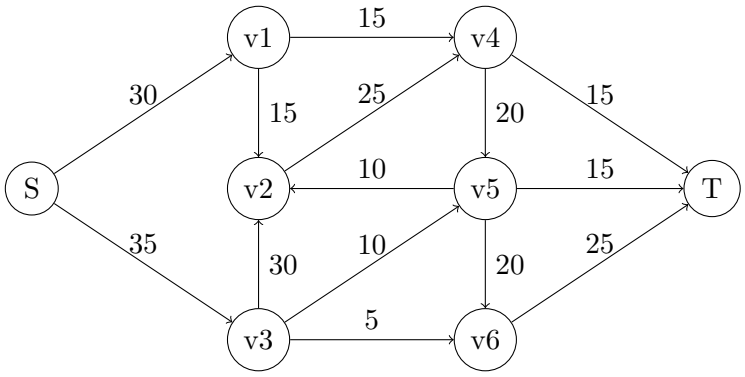




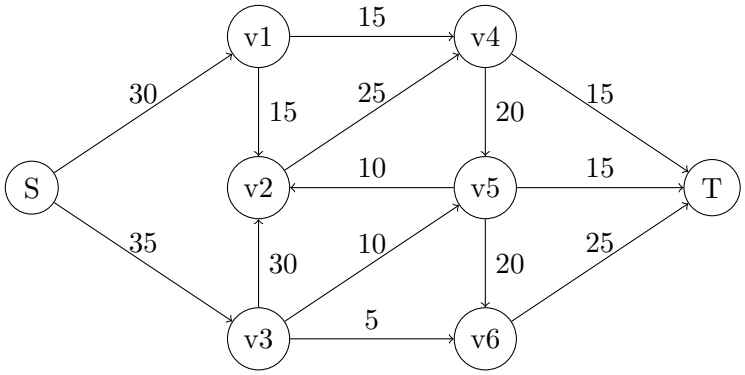


Problem 4 (Lets play with some examples!)**.** In the following, run an implementation of the Ford-Fulkerson Algorithm to determine a maximal flow/maximal matching. Try running with different tree designs to see variations in the number of augmentation necessary. Really, I just want you to play around with a few examples on the algorithms - so, have at it.

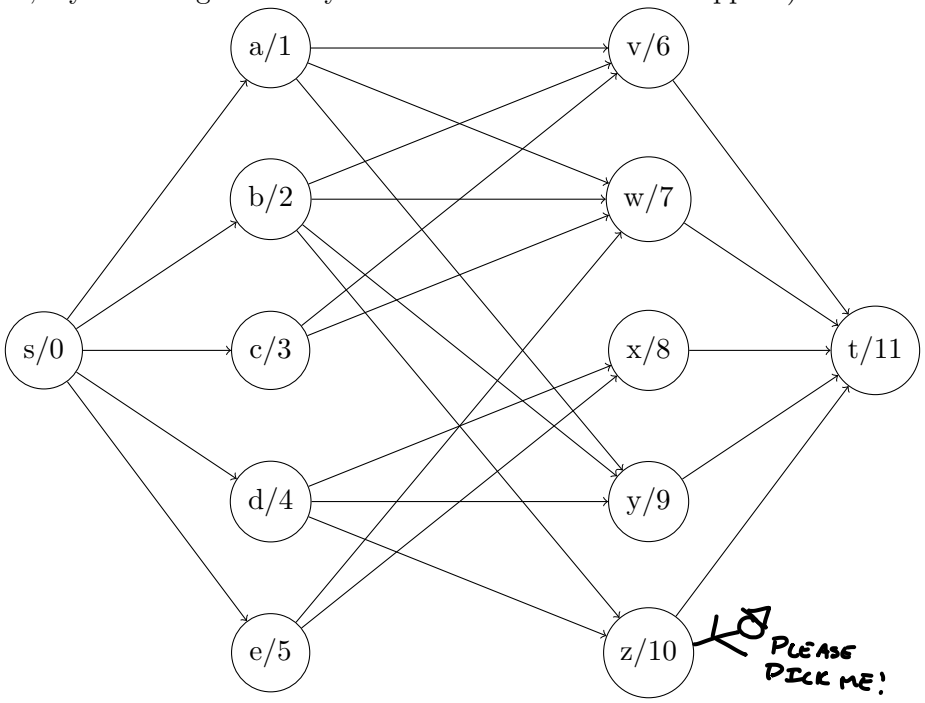
a. (The example you did by hand :-))



b. Change a few of the capacities and see how it effects the flow :-).



c. Matching! (I would recommend printing out the augmenting paths to see what is all going on behind the scenes. Also, try removing a few key connections and see what happens).

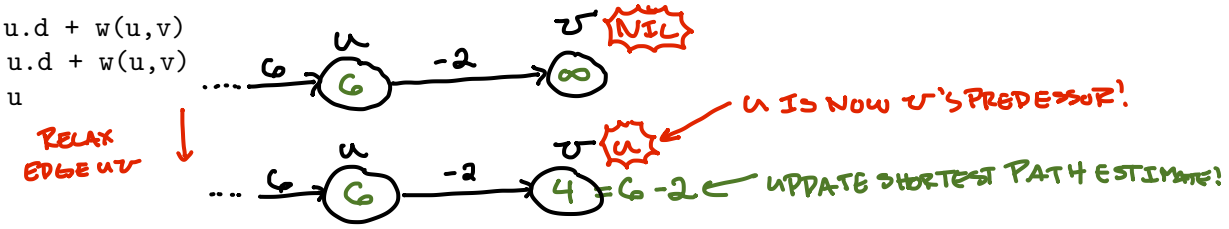


Problem 5 (Bellman-Ford Algorithm)**.** The Bellman-Ford algorithm is an algorithm that computes shortest paths from a source vertex to all of the other vertices in a weighted digraph. The Bellman-Ford algorithm is slower than Dijkstra's algorithm (which solves the same problem), but the Bellman-Ford algorithm allows for negative edge weights!

Similar to Dijkstra's Algorithm, the Bellman-Ford algorithms makes use of the technique known as *relaxation*. For each vertex, v, we maintain an attribute, call it $v.d$, which is an upper bound on the weight of the shortest path from source s to v – this is often called the *shortest-path estimate*.

The process of relaxing an edge, (u, v), consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating values, $v.d$ (shortest-path estimate) and $v.p$ (v's predecessor/parent).

```
Relax(u,v,w)
1.  if v.d > u.d + w(u,v)
2.      v.d = u.d + w(u,v)
3.      v.p = u
```

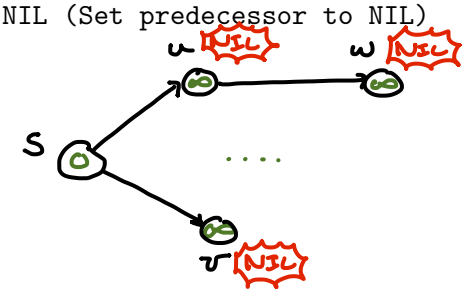


Given a weighted, directed graph $G = (V, E)$ with source s and a weight function w, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle the algorithm produces the shortest paths and their weights.

The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex v until it achieves the actual shortest-path weight.

We begin with an initialization step (set shortest paths to infinity for all non-source vertices and predecessors to NIL).

```
Initialize-Single Source(G,s)
1.  for each vertex v
2.      v.d = inf (shortest-path estimate to infinity)
3.      v.p = NIL (Set predecessor to NIL)
4.  s.d = 0
```



The Bellman-Ford algorithm returns `True` if and only if the graph contains no negative-weight cycles that are reachable from the source.

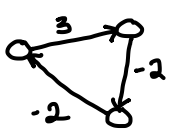
```
Bellman-Ford(G,w,s)
1.  Initialize-Single Source(G,s)
2.  for i = 1 to |V| - 1
3.      for each edge (u,v)
4.          Relax(u,v,w)
5.  for each edge (u,v)
6.      if v.d > u.d + w(u,v)
7.          return False
8.  return True
```

RUNNING TIME?

$O(|E||V|)$

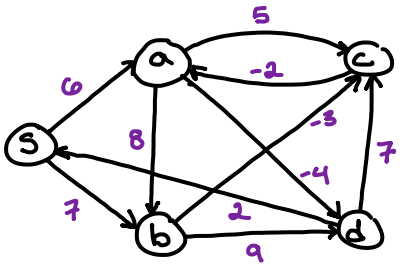
# of Edges — # of vertices

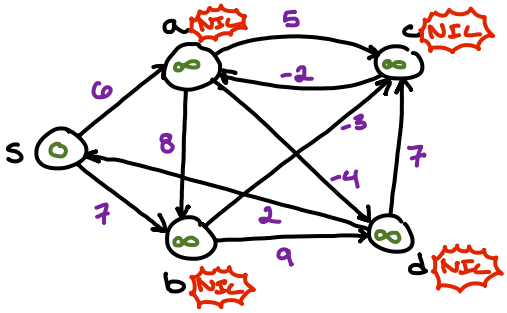
WHAT DOES THIS CHECK / AVOID?



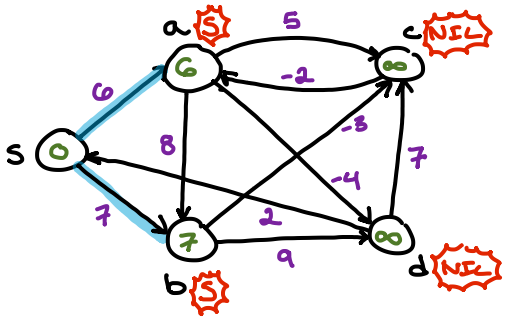
EACH 'LOOP' HAS A WEIGHT SUM OF -1.
SO, IF WE GO AROUND FOR EVER IT KICKS OUT A VALUE OF -∞ ∴.

Example: Below is a run of the Bellman-Ford Algorithm.

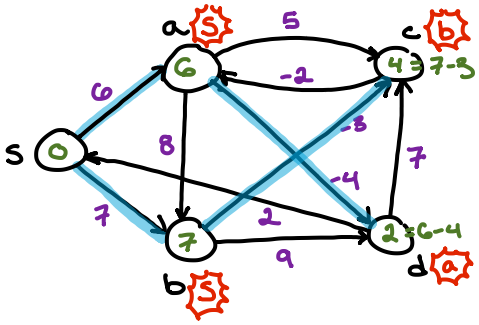




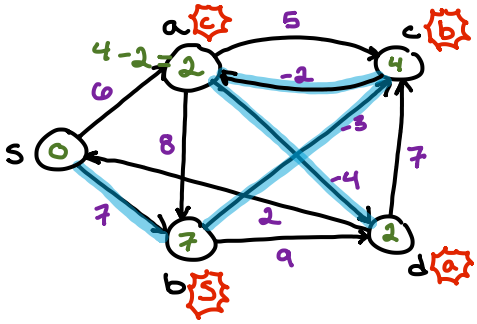
INITIALIZE SINGLE SOURCE
* THE 'd' VALUES APPEAR WITHIN THE VERTICES
* THE 'p' PREDECESSOR APPEAR WITHIN THE ☼



* RELAX EDGE sa (a.d=6 a.p=s)
* RELAX EDGE sb (b.d=7 b.p=s)

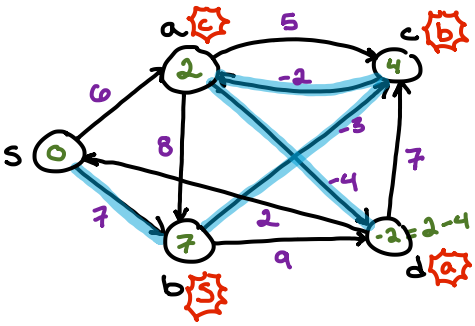


* RELAX EDGE ad (d.d=2 d.p=a)
* RELAX EDGE bc (c.d=4 c.p=b)



* RELAX EDGE ca

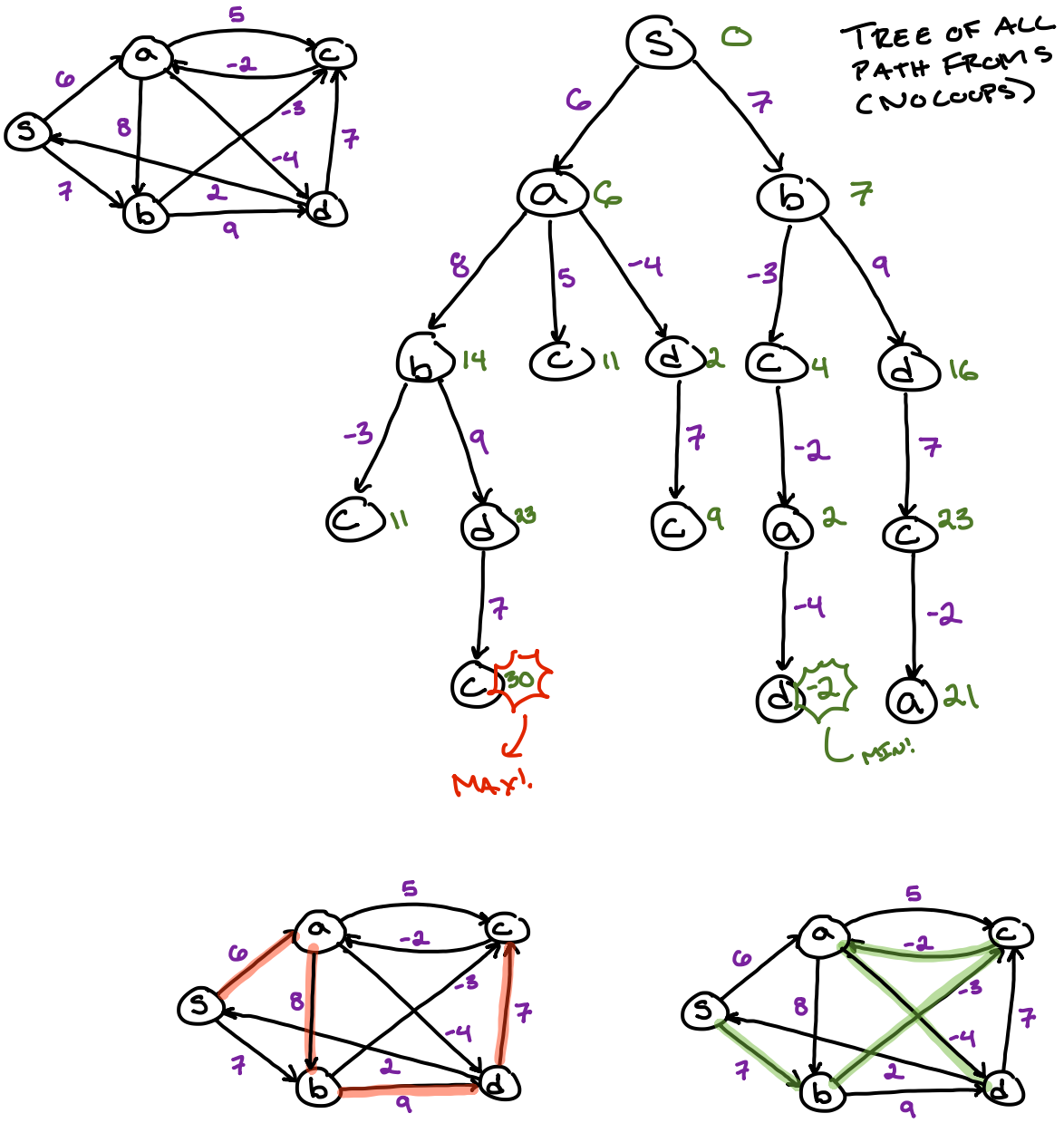
[BEFORE THE RELAX
a.d = 6 > 4 - 2 = c.d + w(c,a)]



* RELAX EDGE ad

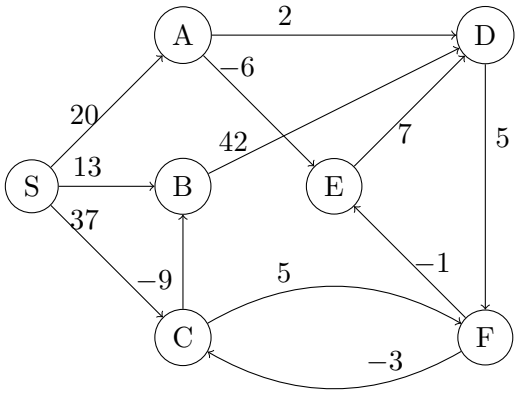
DONE!

Example: Below is a tree representing all of the paths from 0 (that do not contain any loops - i.e., never revisit a vertex). In green, we see the weight sum of each path (we can easily see what the shortest and longest paths are!).



Now it is your turn!

a. Run the Bellman-Ford algorithm on the following using s as the source.



b. (Bonus) Implement the Bellman-Ford Algorithm and run it on this directed graph.