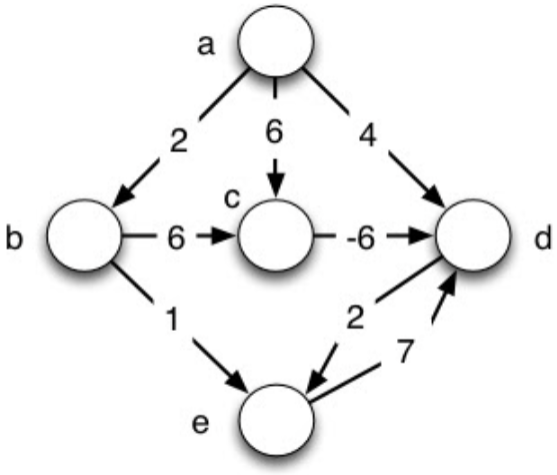
1. (5 pts) Floyd-Warshall

Recall that CLRS initially presents Floyd-Warshall as constructing a series of matrices $D^{(k)}$, and we subsequently showed it can just modify one matrix D. The $D^{(k)}$ are the states of D after processing each k in the main loop.

Below we show the matrix $D^{(0)}$ for the above graph (which is different from the one used in class). Since Floyd-Warshall assumes that vertices are indexed by integers, we map them as follows: a is a vertex 1, b is a vertex 2, etc. Run Floyd-Warshall, showing the matrix $D^{(k)}$ for each value of k. The final matrix should have values for all start vertices.



$D^{(0)}$

	1 (a)	2 (b)	3 (c)	4 (d)	5 (e)
1 (a)	0	2	6	4	∞
2 (b)	∞	0	6	∞	1
3 (c)	∞	∞	0	-6	∞
4 (d)	∞	∞	∞	0	2
5 (e)	∞	∞	∞	7	0

$D^{(1)}$

	1 (a)	2 (b)	3 (c)	4 (d)	5 (e)
1 (a)	0	2	6	4	∞
2 (b)	∞	0	6	∞	1
3 (c)	∞	∞	0	-6	∞
4 (d)	∞	∞	∞	0	2
5 (e)	∞	∞	∞	7	0

$D^{(2)}$

	1 (a)	2 (b)	3 (c)	4 (d)	5 (e)
1 (a)	0	2	6	4	3
2 (b)	∞	0	6	∞	1
3 (c)	∞	∞	0	-6	∞
4 (d)	∞	∞	∞	0	2
5 (e)	∞	∞	∞	7	0

$D^{(3)}$

	1 (a)	2 (b)	3 (c)	4 (d)	5 (e)
1 (a)	0	2	6	0	3
2 (b)	∞	0	6	0	1
3 (c)	∞	∞	0	-6	∞
4 (d)	∞	∞	∞	0	2
5 (e)	∞	∞	∞	7	0

$D^{(4)}$

	1 (a)	2 (b)	3 (c)	4 (d)	5 (e)
1 (a)	0	2	6	0	2
2 (b)	∞	0	6	0	1
3 (c)	∞	∞	0	-6	-4
4 (d)	∞	∞	∞	0	2
5 (e)	∞	∞	∞	7	0

$D^{(5)}$

	1 (a)	2 (b)	3 (c)	4 (d)	5 (e)
1 (a)	0	2	6	0	2
2 (b)	∞	0	6	0	1
3 (c)	∞	∞	0	-6	-4
4 (d)	∞	∞	∞	0	2
5 (e)	∞	∞	∞	7	0

2. (10 pts) Parallel Floyd-Warshall

In this problem, we will parallelize the Floyd-Warshall algorithm. Use the following pseudocode as your starting point. (The CLRS version had D = W for line 2; we replace this with lines 2-5 to make the loops needed for array assignment explicit.)

```
Floyd-Warshall(W)
1  n = W.rows
2  create n x n array D
3  for i = 1 to n
4      for j = 1 to n
5          D[i,j] = W[i,j]
6  for k = 1 to n
7      for i = 1 to n
8          for j = 1 to n
9              D[i,j] = min(D[i,j], D[i,k] + D[k,j])
10 return D
```

(a) Design a parallel version of this algorithm using spawn, sync, and/or parallel for as appropriate. (Copy and modify the pseudocode.) Think carefully about what can be parallelized and what can't, and explain your choices.

Algorithm 1: Parallel-Floyd-Warshall(W)

```
1  n = W.rows;
2  create n * n array D;
3  parallel for i = 1 to n;
4  parallel for j = 1 to n;
5  D[i,j] = W[i,j];
6  for k = 1 to n do
7  |  parallel for i = 1 to n;
8  |  parallel for j = 1 to n;
9  |  D[i,j] = min(D[i,j] + D[i,k] + D[k,j]);
10 end
11 return D ;
```

(b) Analyze the asymptotic runtime of your algorithm in terms of its work, span, and parallelism (all three).

Through the modification of the for loops to the parallel for, the memory that is accessed is otherwise partitioned. Each line before lines 6-8 run at most $/Theta(n^2)$ time if accounting for the double parallel for loop. Lines 6-8 have the highest run time, with a total run time of $/Theta(n^3)$. This analyzes the total run time of work.

In terms of span, the use of the divide and conquer implementation is analyzed, as the run time for parallel is known to be $/Theta(logn)$, as depth of recursion can be understood as being $\Theta(logn)$.

By using the equation in the web notes, we can find the parallelism by:

$$\frac{T_1}{T_\infty} = \frac{n^3}{nlogn} = \frac{n^2}{logn}$$

The total run time of parallelism then, in terms of what is given as work and span is $\Theta\frac{n^2}{logn}$

3. (5 pts) An Alternative Proof of Correctness

If we can show that Dijkstra's algorithm relaxes the edges of every shortest path in a directed graph in the order in which they appear on the path, then the path relaxation property applies to every vertex reachable from the source, and we have an alternative proof that Dijkstra's algorithm is correct. Either show that Dijkstra's algorithm must relax the edges of every shortest path in a directed graph in the order in which they appear on the path, or provide a counter-example directed graph in which the edges of a shortest path could be relaxed out of order and explain how that happens.

It is given in Dijkstra's algorithm that the edges of every shortest path in the graph is

relaxed. The edges can be relaxed out of order, given they have the same weight capacity. An example of this is as follows:
Given an arbitrary graph from A to F, where each vertex is connected by the nodes A $\rightarrow$ B, B $\rightarrow$ C, C $\rightarrow$ D, D $\rightarrow$ F. A is denoted as the start vertex, while F is denoted as the finishing vertex. If, for example, each of these nodes were connected by edges that weighed the same, there would be multiple paths to get from A $\rightarrow$ F, but certain edges would not be relaxed, because Dijkstra's algorithm will complete before checking the other edges from A $\rightarrow$ F. If it checks B $\rightarrow$ F, C $\rightarrow$ F, D $\rightarrow$ F, one of these given edges will not be relaxed. Dijkstra's algorithm will only relax an edge to a vertex that has an unknown shortest path, which means the second "shortest path" found will not be relaxed.

4. (10 pts) Vertex Capacities

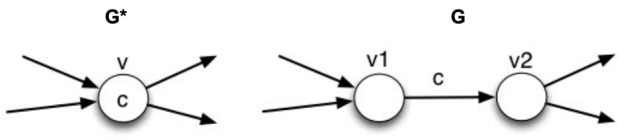
This is the challenge problem you completed in class, but you will complete the proof.

Suppose that, in addition to edge capacities, a flow network G^* has vertex capacities. We will extend the capacity function c to work on vertices as well as edges: $c^*(u, v)$ gives the usual edge capacity and $c^*(v)$ gives the amount of flow that can pass through v. But we don't want to write a new algorithm.

We want to transform a flow network $G* = (V*, E*)$ with c* that defines both edge and vertex capacities into an equivalent ordinary flow network $G = (V, E)$ and c that is defined only on edges (withou vertex capacities) such that a maximum flow in G has the same value as a maximum flow in G*. THen we can run the algorithms we already have.

In class we identified this transformation: given $G* = (V*, E*)$ and $c*$, compute $G = (V, E)$ and c as follows:

For each vertex $v \in G*$ with capacity c, make vertices v_1 and $v_2 \in G$ and a new edge (v_1, v_2) with capacity c (the same as v). Then rewire the graph so that all edges in v in $G*$ now go into v_1 in G_1 and all edges out of v in $G*$ now come out of v_2 in G (that is, replace (u,v) with (u, v_1) and (v,u) with (v_2, u)). Finally, make s_1 and t_2 be the new source and target vertices of G.



Prove that this solution is correct: that is, a solution computed on G will be a correct solution for $G*$ as follows (your proofs should use formal notation and algebra to be precise,

not just English):

(a) Given a flow f computed on G, describe how you would construct a flow $f*$ on $G*$?

Given a flow f that is computed on G, we can describe the flow $f*$ by $\forall(u_2, v_1) \in G$ that is used to construct the given flow $(u, v) \in G*$, we have:

$$f * (u, v) = f(u_2, v_1)$$

(b) Show that when flows computed on G are converted to flows in $G*$, edge capacities $c * (u, v) in G*$ are respected.

We can show that when flows computed on G are converted to flows in $G*$, the edge capacities $c * (u, v) \in G*$ are respected. The given edge capacities are defined and found by their given edge that corresponds within $G*$. This means that the capacity can be defined as:

$$c * (u, v) = c(v_2, v_1)$$

It is previously defined that $f * (u, v) = f(v_2, v_1)$. This can otherwise be interpreted that:

$$f * (u, v) \leqslant c * (u, v)$$

(c) Show that when flows computed on G are converted to flows in $G*$, vertex capacities $c * (v) \in G$ are respected.

It is deefined that the given edge capacities can be defined as $f(v_1, v_2) \leqslant c(v_1, v_2)$. Therefore, we can show that the flows computed on G are converted to flows in G*, the given vertex capacities $c * (v)$ in G are respected. We can also recognize that $f(v_1, v_2)$ is the given flow assigned to vertex v as $f * (v)$. This can thus be represented as:

$$c(v_1, v_2) = c * (v)$$

Therefore, the flow of $f * (v)$ can be represented as:

$$f * (v) \leqslant c * (v)$$

(d) Show that when flows computed on G are converted to flows in $G*$, conservation constraints are respected.

(e) Show flow equality $f = f*$: a flow in G has the same value as a flow in $G*$.

Through the definition of the flow that is in G, it si understood as the sum of the flows our of a given source in G.

$$|f| = \sum_{v \in V} f(s_1, v)$$

We can also understand that this flow is removed from this started source, so this can further be defined as:

$$\sum_{v \in V} f(s_1, v) = f(s_1, s_2)$$

This otherwise is the sum of the flow of edges that is removed and leaving the start source in G*.

$$f(s_1, s_2) = \sum v \in V f * (s, v) = |f*|$$

(f) Finally, building on the above show that the translated flow $f*$ is maximal. Hint: Proof by contradiction.

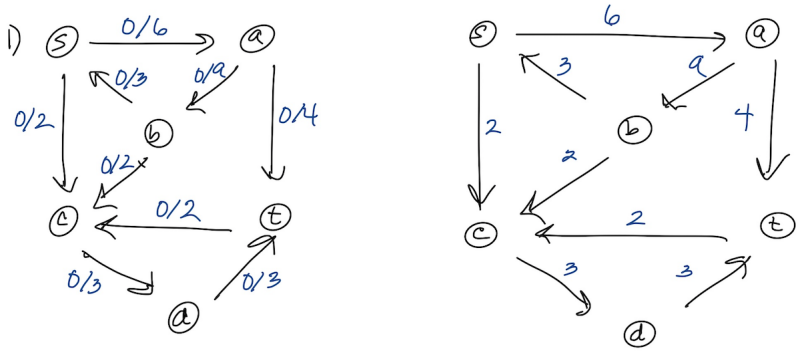
5. (10 pts) Tracing Edmund - Karp

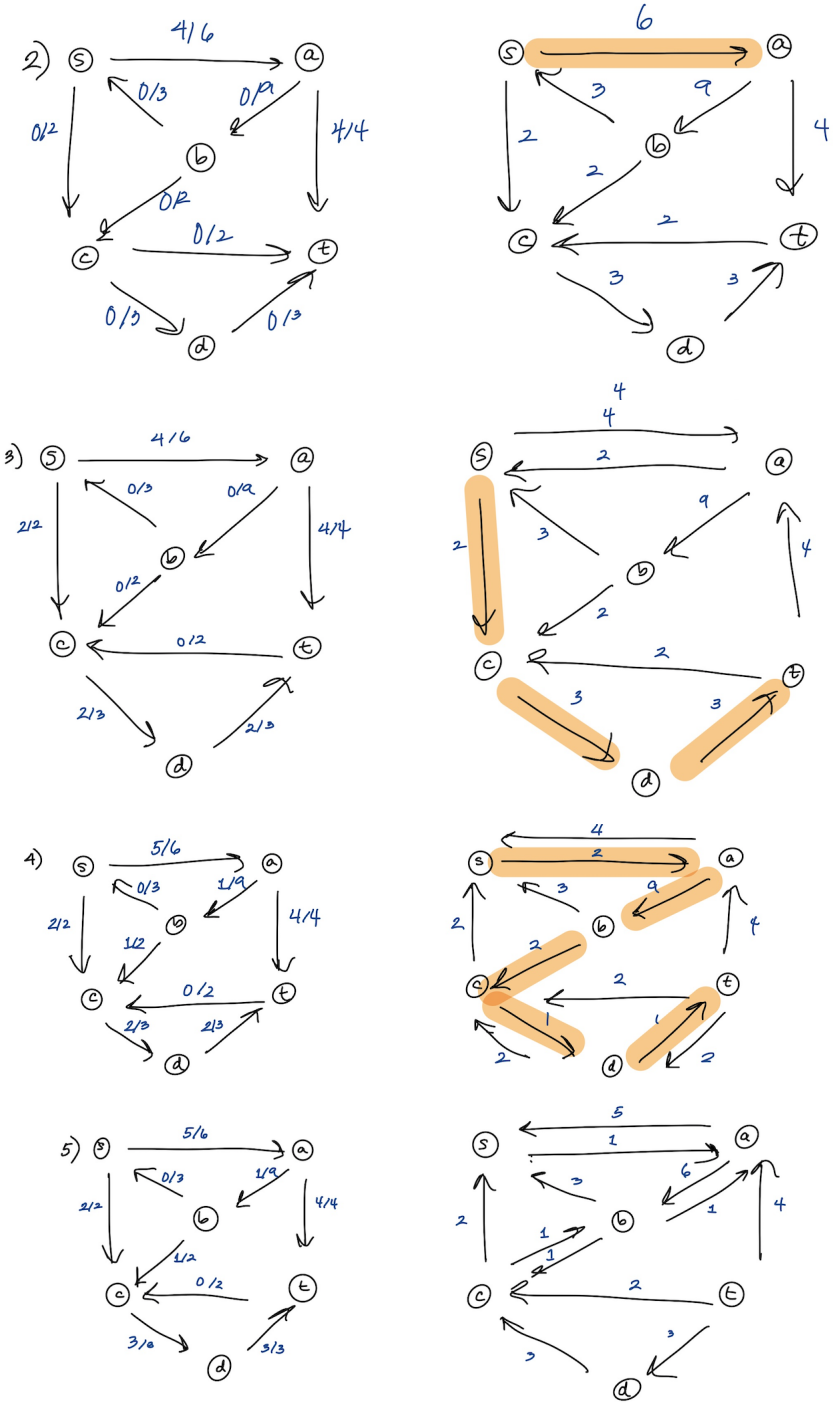
(a) Run Edmunds-Karp on the following graph with s as the source and t as the sink.

• Draw the flow graphs on the left hand side of the page with current flow indicated, and the residual graphs on the right hand side of the page.

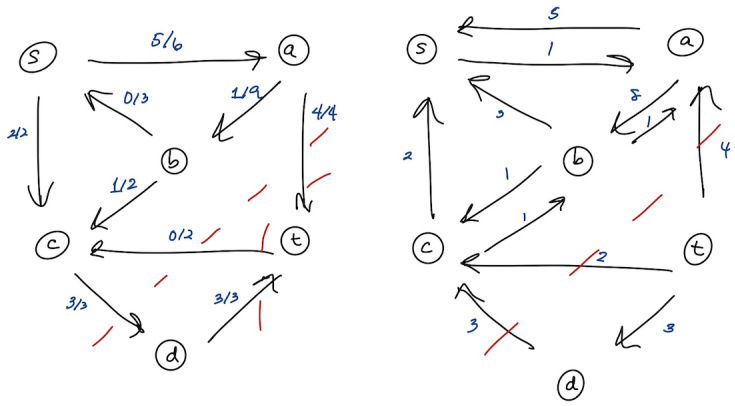
• Mark the shortest augmenting path in the residual graph with thicker lines, and use it to update flow in the flow graph.

• Repeat, redrawing both updated graphs on a new line, until you can't find an augmenting path.





(b) When you can't update the graph any more, write the value of the flow that was achieved, and draw a line in the final graph showing a min cut that corresponds to this max flow.



The graph and a printable template are on the next page for those doing this on paper. Google Drawing, Omnigraffle, and Visio templates are provided.: follow this layout.