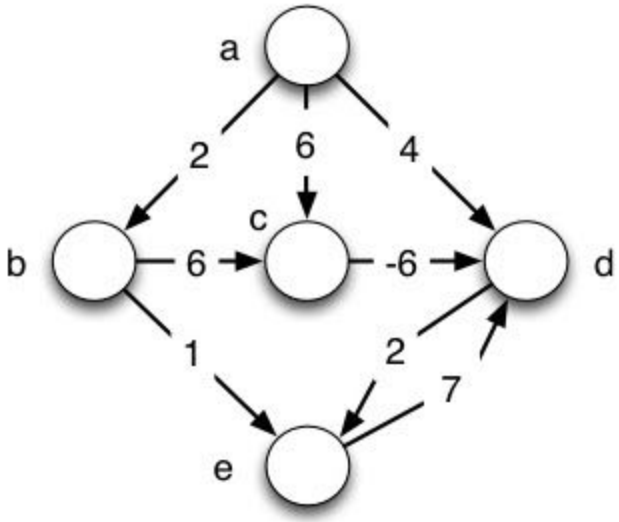
ICS 311 Fall 2020, Problem Set 10, Topics 18-20 & 22

This is a combined problem set, replacing the former problem sets 10 and 11 so that TAs can catch up on grading. The denominator of points for the course will be adjusted accordingly. In order to combine assignments we have removed some of the algorithm tracing problems. You have already traced Dikjstra's algorithm in class, and the optional problem from the same class suggests that you trace Johnson's algorithm, which includes Bellman-Ford and Dijkstra. If you do this and want to see the solution, ask. The remaining algorithms to trace are Floyd-Warshall and Edmunds-Karp.

---

1. (5 pts, 1 for each change) 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.



Blow 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 vertex 1, b is 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.

Changes are marked in yellow.

$D^{(0)}$

	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

$D^{(1)}$**: via a** (no change)

	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

$D^{(2)}$**: via b** (a is only vertex pointing to b so no other rows will change)

	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

$D^{(3)}$: via c

	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

$D^{(4)}$: via d

	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

$D^{(5)}$: via e (no change)

	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, 5 pts) 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**.

```
Floyd-Warshall(W)
1  n = W.rows
2  create n x 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
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 return D
```

We cannot parallelize the loop of line 6, because this is the iteration that implements the bottom-up dynamic programming strategy of solving smaller

subproblems before solving larger ones. We must compute the smaller k before the larger k.

We can parallelize the loops indexed by i and j without race condition because the memory location accessed inside each thread will be partitioned by the indices i and j. Specifically, every thread executing line 5 will access a unique location for each [i,j] and every thread executing line 9 will also access a unique location for each [i,j] because k is fixed.

(b, 5pts) Analyze the asymptotic runtime of your algorithm in terms of its work, span, and parallelism.

Work T_1 is the same as if there were no parallelism: **$O(n^3)$** for the three nested loops, each running from 1 to n, in lines 6-9.

For **span**, we must include the cost to spawn the **parallel for** threads. As described in CLRS, each **parallel for** can be implemented with a divide and conquer strategy that has recursion depth of Θ(lg n). Within each of these threads for lines 3 and 7 it is done again for lines 4 and 8, respectively, but we add (not multiply) as we are seeking depth, not total work, and lines 5 and 9 are constant time, so the spans of the nested parallel for loops are Θ(2lg n) = Θ(lg n). However, the second set of doubly nested parallel loops are inside a third Θ(n) loop, so the total span **$T_\infty = \Theta(n \lg n)$.**

Finally we compute **parallelism** $T_1/T_\infty = \Theta(n^3/(n \lg n))$ = **$\Theta(n^2/\lg n)$**

---

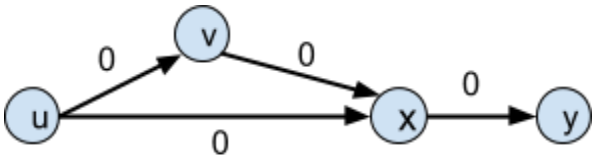
3. (5 pts) An Alternative Proof of Correctness

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.

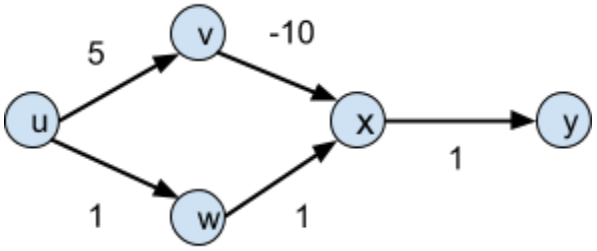
Incorrect Solution (partial credit, 3 points): Students who assume the graphs have only positively weighted edges might provide a proof by contradiction along

these lines: “The path relaxation property applies only to paths with more than one edge. Suppose that the shortest path includes edges (v_i, v_j) and (v_k, v_l). Suppose that Dijkstra’s algorithm relaxed these out of order, namely (v_k, v_l) before (v_i, v_j). This would require that v_k be earlier on the queue than v_i, which implies a lower distance estimate to v_k than v_i. But this would mean there is a lower cost path by which v_k is reached, contradicting our supposition that the shortest path includes (v_i, v_j).” A proof along these lines gets partial credit of 3 points.

Counter-example (for graph with no negative weight edges): The above proof is incorrect because it is possible that v_i and v_k have the same estimated distances and their ordering on the queue is indeterminate. For example, in the following graph where all edge weights are 0, Dijkstra’s algorithm could relax edges in the order (u,x), (u,v), (x,y), (v, x). Both paths to y are shortest paths and are correct, but the fact that Dijkstra’s algorithm finds them is *not* due to the path relaxation property, so this is a counter example to the proposed proof.



Counter-example (for graph with negative weight edges): It is easier to construct a counter example with negative weights, similar to the examples in the web notes.



Edges (u,w), (w,x) and (x,y) are relaxed first, so (x,y) is relaxed out of order even though it is on the shortest path via v.

---

4. (10 pts) Vertex Capacities

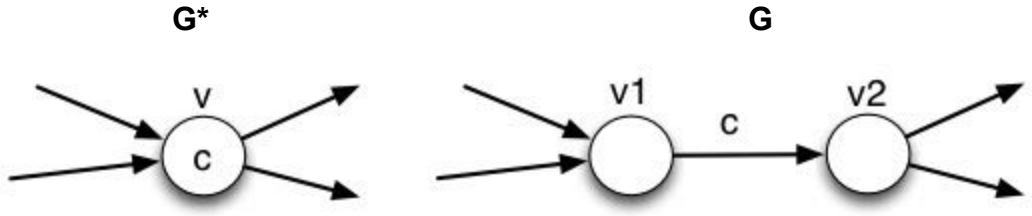
This is the challenge problem you started in class, but you will complete the proof.

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 (without 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 into v in G^* now go to v_1 in G, 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, 1pt) Given a flow f computed on G, **describe how you would construct a flow f^* on G^***.

> **Solution:** For all edges of form (u_2,v_1) in G that were constructed from (u,v) in G* as described above, assign $f^*(u,v) = f(u_2,v_1)$.

(b, 2pts) Show that when flows computed on G are converted to flows in G*, **edge capacities $c^*(u,v)$ in G^* are respected**.

> **Solution:** Every edge in G of the form (u_2,v_1) has a corresponding edge in G* with a corresponding capacity: $c(u_2,v_1) = c^*(u,v)$. We have computed a flow on G

that satisfies this capacity: $f(u_2,v_1) \leq c(u_2,v_1)$. By the conversion above, $f^*(u,v) = f(u_2,v_1)$. Thus, for each such edge, $f(u_2,v_1) \leq c(u_2,v_1)$ implies that $f^*(u, v) \leq c^*(u,v)$; that is, edge capacities are respected.

(c, 1pt) Show that when flows computed on G are converted to flows in G*, **vertex capacities $c^*(v)$ in G* are respected**.

Solution: Edge capacities in G are respected by f, so in particular $f(v_1,v_2) \leq c(v_1,v_2)$. But $f(v_1,v_2)$ is the flow $f^*(v)$ that will be assigned to vertex v (see conservation constraint below), and $c(v_1,v_2) = c^*(v)$, so $f^*(v) \leq c^*(v)$.

(d, 2.5pts) Show that when flows computed on G are converted to flows in G*, **conservation constraints are respected**.

Solution: We can assume that f respects conservation constraints at all vertices in G (i.e., incoming flow = outgoing flow), and need to show that conservation constraints are respected at all vertices in G*.

For each vertex v in G* - {s,t}, consider the corresponding vertices v_1 and v_2 in G.

By conservation of flow f in G, the sum of all flows $f(u,v_1)$ in G will be equal to the sum of all flows $f(v_1,x)$. But due to the construction of the graph, there is only one such $x = v_2$, so the sum of all flows $f(u,v_1)$ in G is equal to $f(v_1,v_2)$.

Similarly, by conservation of flow f in G, the sum of all flows $f(y,v_2)$ in G will be equal to the sum of all flows $f(v_2,w)$. But due to the construction of the graph, there is only one such $y=v_1$, so the flow $f(v_1,v_2)$ is equal to the sum of all flows $f(v_2,w)$ in G.

Therefore, transitively the sum of all flows $f(u,v_1)$ in G is equal to the sum of all flows $f(v_2,w)$ in G. But these are precisely the flows that f* assigns to edges incoming to v and outgoing from v in G*, respectively, so the sum of all flows $f^*(u,v)$ in G* is equal to the sum of all flows $f^*(v,w)$ in G*, proving conservation in G*.

(This proof can also be written algebraically. Source s and sink t have no incoming or outgoing flow, respectively, so would be handled specially.)

(e, 2.5pts) Show flow equality |f| = |f*|: a flow in G has the same value as a flow in G*.

$$|f| = \sum_{v \in V} f(s_1, v) = f(s_1, s_2) = \sum_{v \in V} (s_2, v) = \sum_{v \in V^*} f^*(s, v) = |f^*|$$

That is, the flow in G is defined as the sum of flows on edges coming out of source s_1, but this is the same as the flow on the single edge (s_1,s_2) since there is only one edge coming out of $s_1$in G. By conservation in G, this is the same as the sum of flow on edges going out of s_2 in G, which is by our mapping the sum of flow on edges going out of s in G*, which is by definition the flow in G*.

This proof relies on the common simplification that (unlike the residual network) in a flow network there are no edges going into *s* and no edges coming out of *t*. It could be rewritten without this assumption, but it is much simpler to assume that we can modify the flow network to make s and t true sources and sinks. (If a student solution includes subtracted terms, such as the sum across v∈V of f(v, s_1), then they may be assuming that the source has incoming edges.)

(f, 1pt) Finally, building on the above, show that the translated flow f* is maximal.

Solution: The above steps proved that f is a flow of G if and only if f* is a flow of G*, as they are equal value and all constraints are respected. Assume the maximum flow computed in G when translated leads to a flow f* in G* that is not maximal. Because of the equivalence in step (e), there must be a larger flow in G, which is a contradiction

---

5. (10 pts) Tracing Edmund-Karp

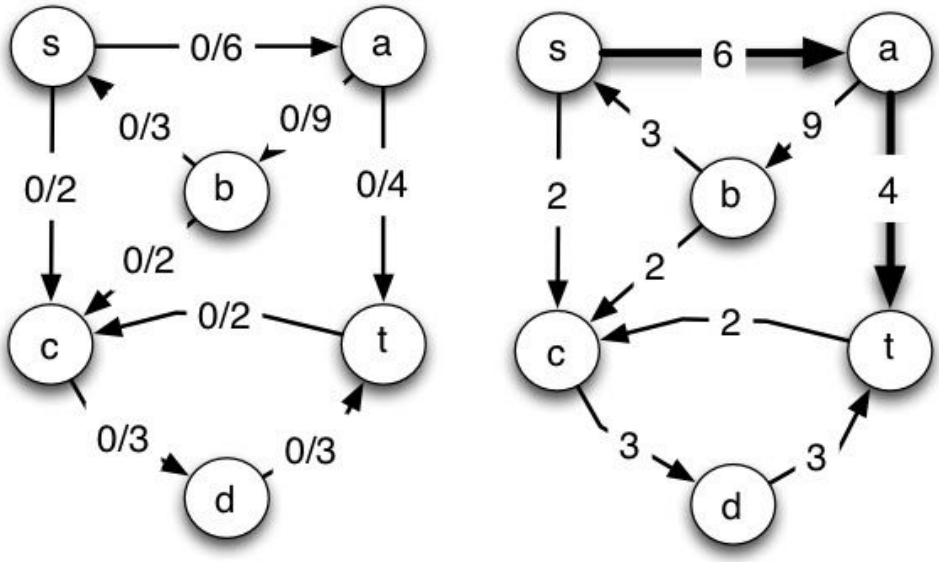
(a, 8pts) Run Edmunds-Karp on the following graph with **s** as the source and **t** as the sink.

- 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.

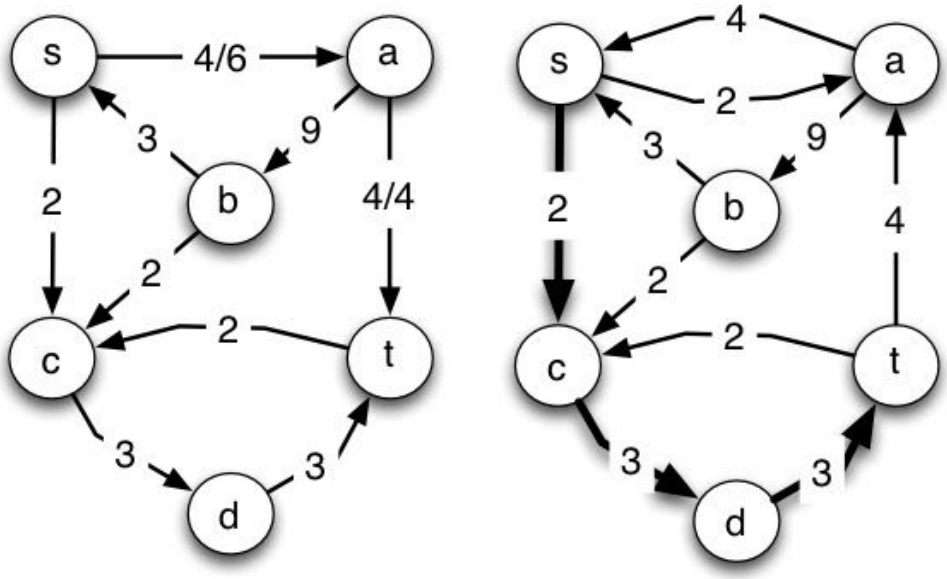
The solution begins on the next page.

Flow graphs: **Residual graphs:**

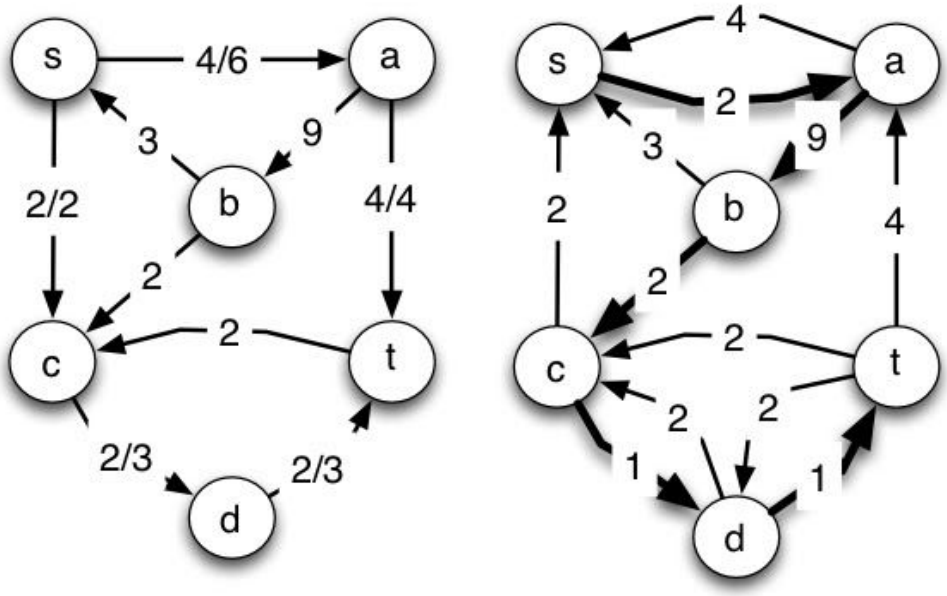
Initial flow graph on left; and first residual graph with augmenting path found:



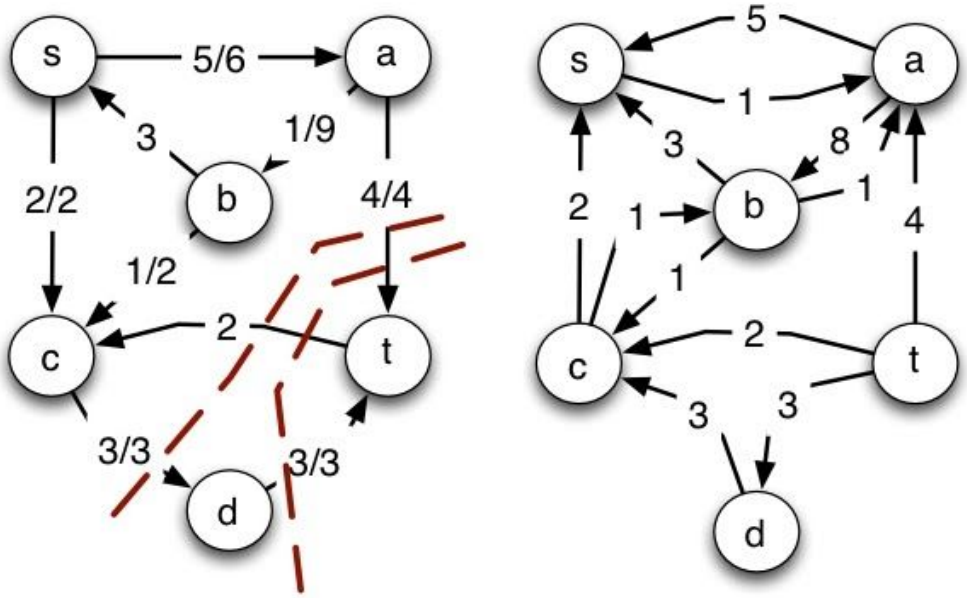
Flow of 4 found above is added on the left; new residual & augmenting path on right:



Flow of 2 found above is added on the left; new residual & augmenting path on right (next page):



Flow of 1 found above is added on the left below; new residual is shown on right. There are no more paths possible from s to t in the residual so we must be done.



(b, 2pts) When you can't update the graph any more, write the value of the flow $|f|$ that was achieved, and draw a line in the final graph showing a min cut that corresponds to this max flow.

> **Solution:** $|f| = 7$. Two possible min-cuts are shown in red (above); the student need only show one of them