Johnson's APSP Algorithm Network Flows

Chapters 25, 26

Lecture Overview

- Johnson's All-Pairs Shortest Paths Algorithm
- Maximum Flow Problem (Introduction)

Johnson's Algorithm

Johnson's algorithm solves the All-Pairs Shortest Paths (APSP) in $O(V^2 \log V + VE)$ time. This is better than our other solutions in the case where G is sparse. Johnson's algorithm uses both Dijkstra's algorithm and the Bellman-Ford algorithm as subroutines.

A principal idea in Johnson's algorithm is reweighting. If G has negative-weight edges but no negative-weight cycles, we compute a new set of nonnegative edge weights.

Once we have all nonnegative edge weights, we can run Dijkstra's algorithm once from each vertex. Using Fibonacci heaps for the min-priority queue, we get the desired running time of $O(V^2 \log V + VE)$.

Reweighting

When we reweight G, the new edge weights w* must satisfy two properties:

1. For all pairs of vertices u, v $\in$ V, a path p is a shortest path from u to v using original weight function w if and only if p is also a shortest path from u to v using weight function w*.

2. For all edges (u, v), the new weight w*(u, v) is nonnegative.

Preserving Shortest Paths

Let δ denote shortest-path lengths derived from the weight function w, and δ^* denote shortest-path lengths derived from the weight function w^*

Lemma. Given G=(V, E) with weights w, let $h: V \to \mathbf{R}$ be any function mapping vertices to real numbers. Define

$$w^*(u, v) = w(u, v) + h(u) - h(v).$$

Let p be any path from v_0 to v_k. Then p is a shortest path with weight function w if and only if it is a shortest path with weight function w^*. I.e. $w(p) = \delta(v_0, v_k)$ iff $w^*(p) = \delta^*(v_0, v_k)$.

Furthermore, G has a negative-weight cycle using weights w iff G has a negative-weight cycle using weights w^*.

Preserving Shortest Paths

Proof. Let p = $\langle v_0, v_1, \ldots, v_k \rangle$. Then

$$\begin{aligned}
w^*(p) &= \sum_{i=1}^{k} w^*(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
&= w(p) + h(v_0) - h(v_k)
\end{aligned}$$

Therefore, any path p from v_0 to v_k has $w^*(p) = w(p) + h(v_0) - h(v_k)$. If one such path is shorter than another using weights w, it is shorter using weights w*.

Preserving Shortest Paths

Thus, $w(p) = \delta(v_0, v_k)$ if and only if $w^*(p) = \delta^*(v_0, v_k)$.

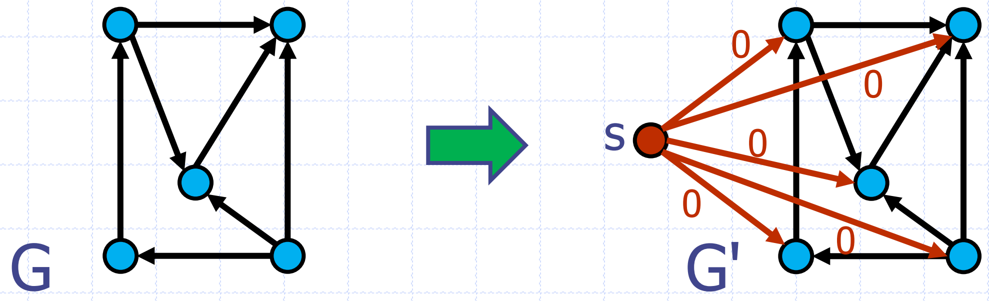
Next consider the case when p is a cycle; i.e. $v_0 = v_k$. Then

$$\begin{aligned} w^*(p) &= w(p) + h(v_0) - h(v_k) \\ &= w(p) + h(v_0) - h(v_0) \\ &= w(p). \end{aligned}$$

So each cycle is the same weight under w* as it is under w. Thus, w* has a negative-weight cycle iff w does. ■

Producing Nonnegative Weights

In order to produce nonnegative weights, Johnson's algorithm uses a clever trick of adding a vertex s to the graph with 0-weight edges from s to every other vertex in V.



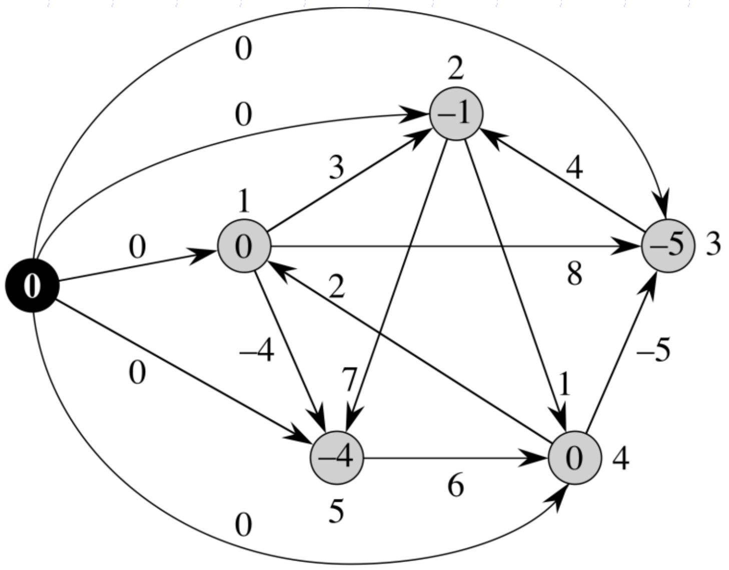
So V' = V + s. Note that no new paths between vertices of V is created by the addition of s and its edges. In particular, shortest paths are unchanged. Also note that G has a negative-weight cycle iff G' has a negative-weight cycle.

Producing Nonnegative Weights

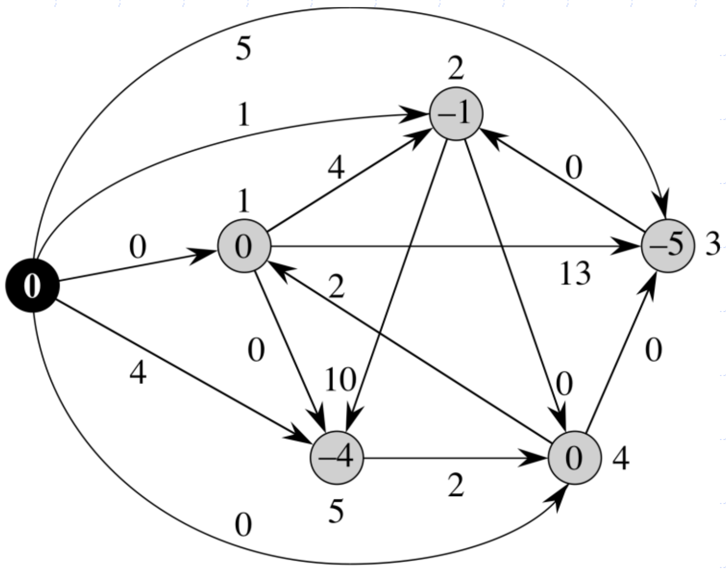
So suppose there are no negative-weight cycles in G or G', and define $h(v) = \delta(s, v)$ for all vertices v in V'. Note that this gives $h(v) = 0$ unless there is a negative-weight path from some u to v.

By the triangle inequality, we have that $\delta(s, v) \le \delta(s, u) + w(u, v)$ for all edges (u, v) in E'. We can rewrite this as $h(v) \le h(u) + w(u, v)$, or $w(u, v) + h(u) - h(v) \ge 0$. But the right-hand side of this inequality is precisely the definition of $w^*(u, v)$, so we get $w^*(u, v) \ge 0$ for every edge (u, v) in E'.

Producing Nonnegative Weights



(a)



(b)

Johnson's Algorithm

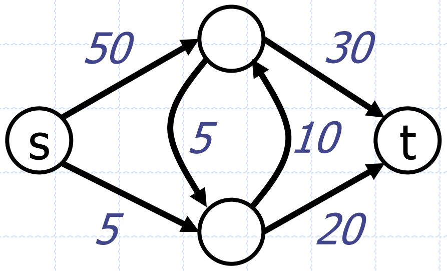
```
1.  Augment G with s and its edges, giving G'.
2.  If BELLMAN-FORD(G', s) is FALSE
3.      return NIL;                    // G has a negative-weight cycle.
4.  else
5.      for each vertex v in V'
6.          h(v) := δ(s, v) (from BELLMAN-FORD)
7.      for each edge (u, v) in E'
8.          w*(u, v) = w(u, v) + h(u) – h(v)
9.      for each vertex u in V
10.         run DIJKSTRA(G, w*, u) to compute δ*(u, v) for all v in V
11.         for each vertex v in V
12.             $d_{uv}$ = δ*(u, v) + h(v) – h(u)
13.     return D
```

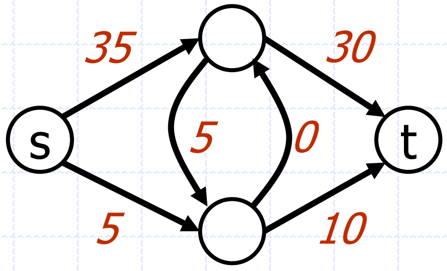
Flow Networks

- We can use weights in digraphs to model situations other than distances or costs.   In this section (Chapter 26) we will use weights to model the capacities of edges to transfer material.
- We can think of perhaps water flowing in pipes, with the edges representing pipes, and the capacities related to the diameter of the pipes and the pressures the pipes can take.   Or we can model parts moving through assembly lines, current through electrical networks, or information flowing through communication networks.
- We will designate two nodes of our digraph as the source (s) and the sink (t) of the flow. The source will have flow going out of it (only) and the sink will have flow going into it (only). At any node other than the source and the sink, the amount of material flowing into the node must equal the amount flowing out; material is not allowed to accumulate in nodes. This is called the conservation of flow.

Flow Networks

In the Maximum-Flow Problem, we wish to compute the greatest rate at which material can be shipped from the source to the sink without violating any capacity constraints.





We will see two methods of solving the maximum-flow problem. The first is known as the Ford-Fulkerson method of augmenting paths. The second is the push-relabel method.

We start with some formal definitions and properties of flow networks.

Flow networks and flows

A flow network is a weighted, directed graph G with distinguished vertices s and t. We let c (for capacity) represent the weight function.

For convenience, we will assume that for every vertex v, there is a path from s to v, and a path from v to t. Otherwise the vertex is unusable in a flow from s to t.

A flow in G is a function f: $V \times V \rightarrow \mathbf{R}$ that satisfies:

- **capacity constraint:** $f(u, v) \leq c(u, v)$ for all u, v in V.
- **flow conservation:** $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ for all u in V – {s, t}.

Flow networks and flows

The flow f(u, v) from vertex u to vertex v can be zero or positive. The value of a flow f is:

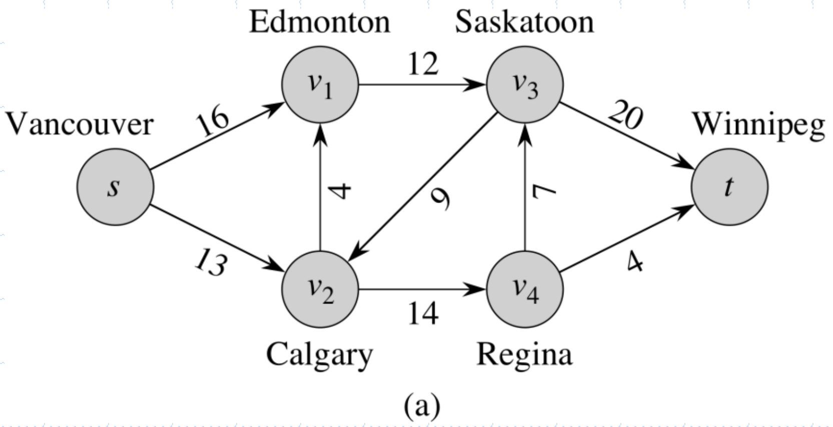
$$|\mathrm{f}| = \sum_{v \in V} f(s, v)$$

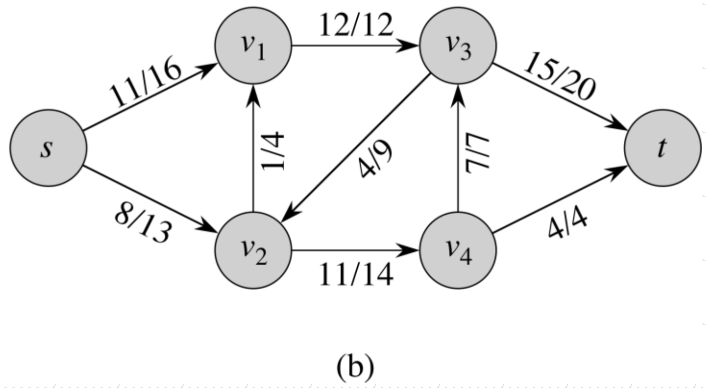
that is, the total flow coming out of the source. We wish to find a flow of maximum value.

The flow conservation property says that the total flow into a vertex other than s or t is equal to the total flow out of the vertex.

When neither (u, v) nor (v, u) is in E, there can be no flow between u and v and so f(u, v) = f(v, u) = 0.

An example





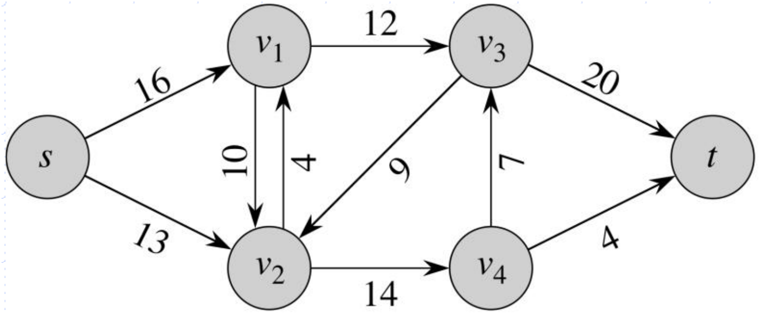
(a) The Lucky Puck Company's transport network. Capacities are in crates/day.

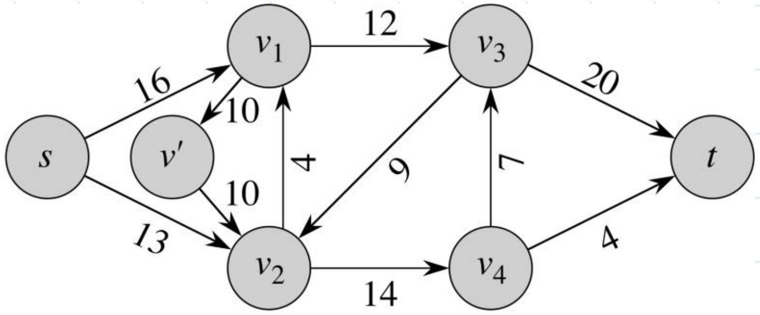
(b) A flow of value 19 in the network. This is not a maximum flow. A flow value of 23 is achievable.

Antiparallel edges

Two edges are said to be antiparallel if one edge is (u, v) and the other is (v, u). It will be convenient to assume that there are no antiparallel edges in a network.

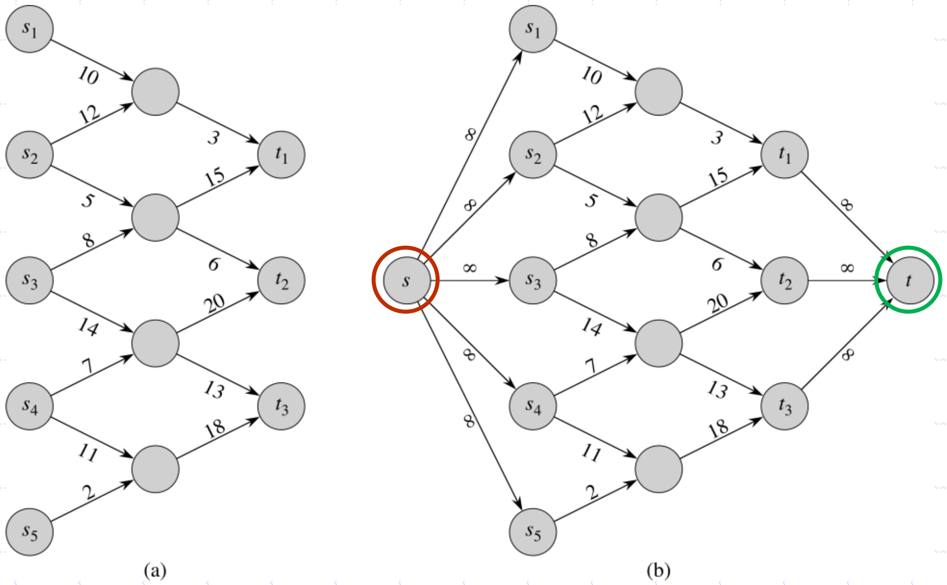
It is easy to transform a network with antiparallel edges to one with no antiparallel edges. Suppose (u, v) and (v, u) are antiparallel edges. Choose one of them, say (u, v), and split it by adding a new vertex v' and replacing edge (u, v) with the pair of edges (u, v') and (v', v). We also set the capacity of both of the new edges to the capacity of the original edge. Do this for each pair of antiparallel edges.





Multiple Sources and Sinks

A maximum-flow problem may have several sources (s_1, s_2, etc.) and sinks (t_1, t_2, etc.), rather than just one of each. We can reduce this problem to an ordinary single-source, single-sink problem. We simply add a supersource s and supersink t. The supersource has edges (s, s_i) to each source s_i with capacity ∞, and the supersink has edges (t_i, t) from each sink t_i with capacity ∞.



s_1
s_2
s_3
s_4
s_5
t_1
t_2
t_3
s
t
10
12
5
8
14
7
11
2
3
15
6
20
13
18
∞
(a)
(b)