CS 491G Combinatorial Optimization
Lecture Notes

David Owen

June 6, 8

1 Ford Procedure (continuing from previous lecture)

Ford Procedure (for solving Shortest Paths Problem):

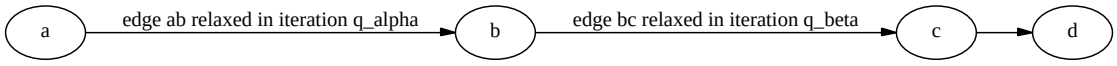
while ($\vec{y} \neq$ a feasible potential)

Find an incorrect edge ($e_{vw} : y_v + cost(e_{vw}) < y_w$)

Relax e_{vw} ($y_w = y_v + cost(e_{vw})$)

The Ford Procedure (called "Ford's Algorithm" in the textbook [1, p.21]) can be used to solve the Shortest Paths Problem for graphs that do not contain negative cycles. We call it a "procedure" rather than an "algorithm" because it is not guaranteed to terminate. In particular, the Ford Procedure will run forever on graphs containing negative cycles.

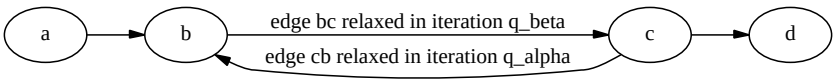
As the Ford Procedure runs on a graph, the values in $\vec{y}$ will decrease, converging to a set of shortest paths from r to all $v \in V$. The overall order of relaxations is arbitrary, and a particular edge may be relaxed more than once. Consider the following shortest path from a to d:



Suppose this shortest path from a to d is the output from an application of the Ford Procedure, that the edge e_{ab} was most recently relaxed in iteration q_α (of the code within the **while** loop above), and that the edge e_{bc} was most recently relaxed in iteration q_β.

Is it possible that $\alpha > \beta$—that is, iteration q_α occurred after iteration q_β? Before the most recent relaxation of e_{ab}, the value of y_b must have been greater than its relaxed value, otherwise the relaxation would not have been done. If $\alpha > \beta$, then this relaxation made y_b less than what it was at the time e_{bc} was relaxed. Since e_{ab} is part of the path to c, y_c may be decreased further to take into account the decrease in y_b—this means e_{bc} will need to be relaxed again. Either $\alpha < \beta$ or there are more relaxations to be done; so α must be $< \beta$ if this is truly the shortest path.

Now consider the same path but modified to include a cycle:



Is it possible this is a shortest path? If so $\alpha < \beta < \alpha < \beta \ldots$ Obviously this would lead to a contradiction. We conclude that no shortest path may contain a cycle.

2 Special Cases

For certain special graphs we can verfiy ahead of time that there will be no negative cycles. We will consider three special cases: unit-cost graphs, DAG's (directed acyclic graphs), and nonnegative-cost graphs. Before presenting and proving algorithms for these special cases, we need to define the "*scan*" operation, which will be used by these algorithms:

scan(v):

1. Let $L(v) = \{a : e_{va} \in E\}$.
2. Relax all edges e_{va}, where $a \in L(v)$ (all edges beginning at v).

2.1 Unit-Cost Graphs

We define a "unit-cost digraph" as a weighted digraph in which all weights are equal and ≥ 0 (and therefore all weights may be set $= 1$ without losing any information). Since all edge weights are ≥ 0, the graph cannot have a negative cycle. So we could use the Ford Procedure and be guaranteed termination. But the shortest path may be found more efficiently if we exploit the structure of the unit-cost digraph.

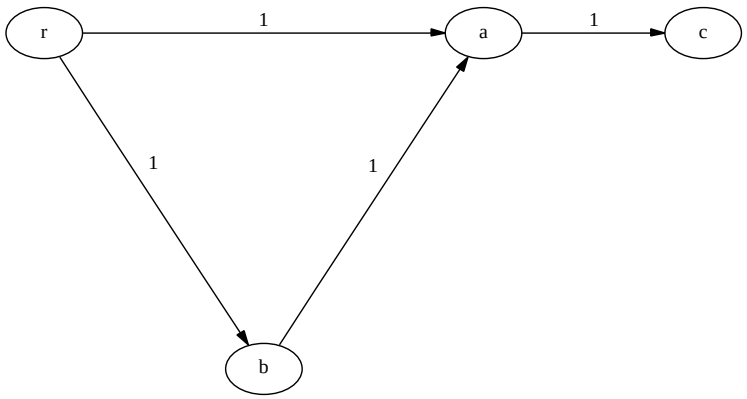
We observe that the shortest path will be the path with the least number of edges (since all edge weights are equal). This means that a BFS (breadth-first search) of the graph, beginning at r, will give us a set of shortest paths from r to all other vertices. And BFS will process each edge only once—that is, no edge will need to be relaxed more than once if we use the following algorithm, based on BFS:

BFS-based shortest paths algorithm for unit-cost digraphs:

1. Initialize $FIFO$ (first-in, first-out) queue to empty, all $\vec{y}$ values to ∞, and all $\vec{p}$ (array of vertices' parents) values to -1.
2. Put r into the queue, set $y_r = 0$, and set $p_r = r$.
3. **while** (queue $\neq$ empty)

   a) Extract vertex v from queue head.
   b) *scan*(v), updating $\vec{y}$ and $\vec{p}$.
   c) As v is scanned, check to see whether child vertices of the edges from v have ever been in the queue; if not, put them in the queue.

For example, we can run the algorithm on the following unit-cost digraph:



After initializing the queue, $\vec{y}$, and $\vec{p}$, we put r into the queue, set $y_r = 0$, and set $p_r = r$:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	∞	-1
b	∞	-1
c	∞	-1

queue $= r \rightarrow$

We take r out of the queue and $scan(r)$ (relax all edges whose tails begin at r—in this case that's e_{ra} and e_{rb}). Also, since a and b have not yet been in the queue, we put them into the queue. Updating $\vec{y}$, $\vec{p}$, and the queue, we have:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	1	r
b	1	r
c	∞	-1

queue $= a \rightarrow b \rightarrow$

We take a from the queue, *scan* it, find that c has never yet been in the queue, and update accordingly:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	1	r
b	1	r
c	2	a

queue $= b \rightarrow c \rightarrow$

We take b from the queue and *scan* it. But, although a is a child of b, there are no edges to relax, since $y_a < y_b + 1$. Finally, we take c from the queue and find that no changes result from $scan(c)$. Since the queue is empty, we are done.

To prove the correctness of this approach, we note first of all that it must be (trivially) correct for $v = r$ (y_r is initialized to 0 and never changes after that). Since it is true in that trivial case, we know it must be true for $v \in \{$the set of vertices one edge away from $r\}$, where $y_v = 1$, since y_v couldn't possibly be less than 1 in a unit-cost digraph. By induction, the algorithm must be correct for all $v \in V$ because of the optimal substructure property of shortest paths (shortest paths contain shortest subpaths): the algorithm will assign 1+ some correct $\vec{y}$ value to each new $\vec{y}$ value, which must be the shortest because no better edge ($cost(e) < 1$) can exist in a unit-cost digraph. In the case of vertices not connected to r, $y_v = \infty$ at initialization and doesn't change.

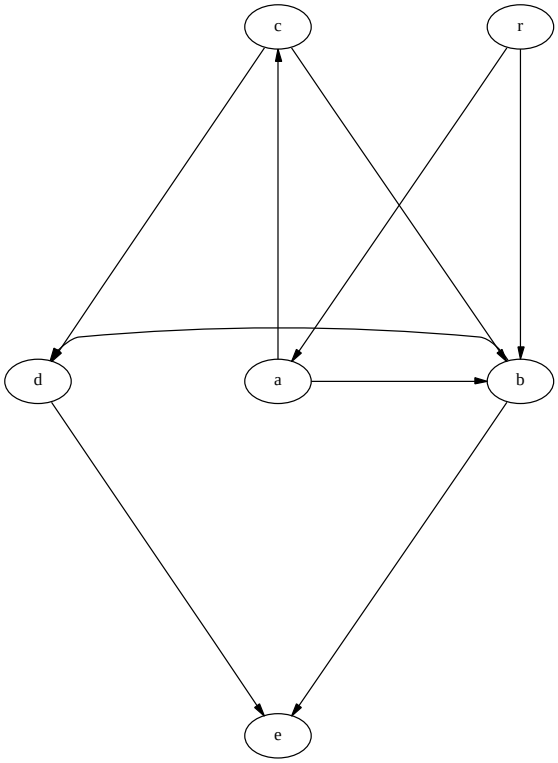
2.2 *DAG*'s (directed acyclic graphs)

A *DAG* is a directed graph with no cycles. It may have negative edge weights, but since there are no cycles, obviously there are no negative cycles. We could therefore solve the shortest paths problem for a *DAG* using the Ford Procedure and be guaranteed that the procedure would terminate. But as we found with unit-cost digraphs, it is possible to exploit the structure of a *DAG* and thereby solve the shortest paths problem more efficiently. The important feature of a *DAG* is that there will always be a "topological sort" of the vertices:

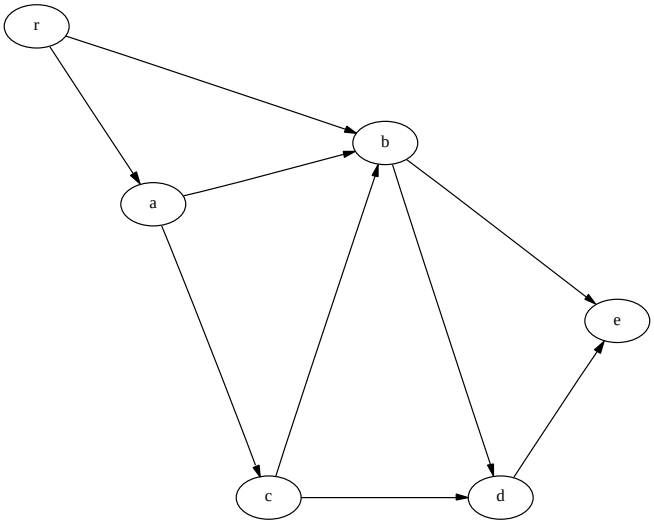
Topological Sort:

> An ordering of the vertices in a *DAG* such that if $e_{ij} \in E$, then v_i precedes v_j.

A topological sort may be thought of as an arrangement of vertices so that all of the edges point in the same direction. For example:



Although it is difficult to see, the graph above is a DAG and can be rearranged so that all of its edges point from left to right:



The topological sort of this graph's vertices would be r, a, c, b, d, e (the order from left to right in the graph above). For our DAG shortest paths algorithm, we will also need the following lemma:

Lemma:

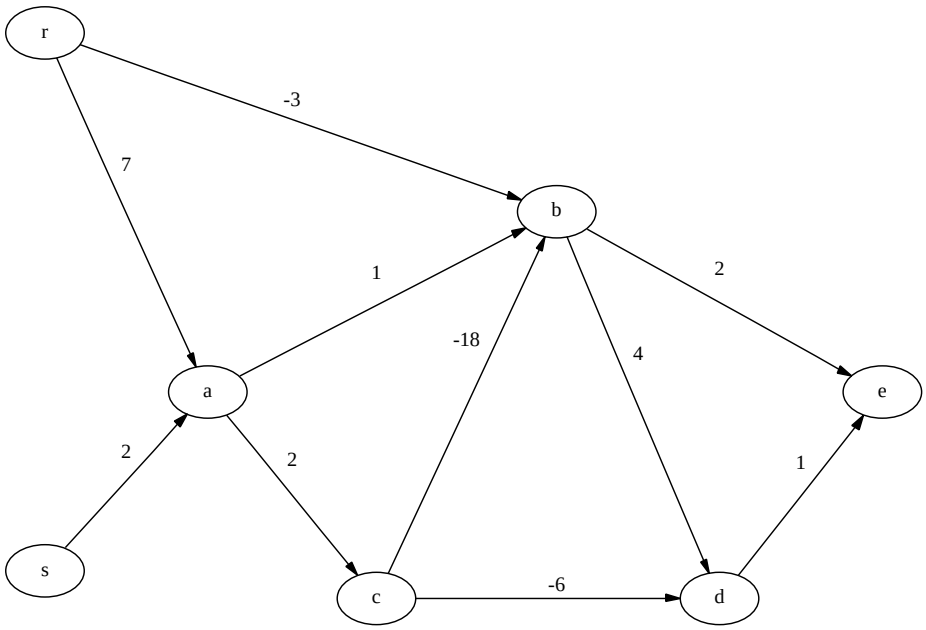
Deleting a vertex from a DAG produces a DAG structure.

For the proof (by contradiction) of the lemma, assume we have deleted a vertex and the resulting graph has a cycle (it is not a DAG). If so, there must have been a cycle in the original graph, because deleting a vertex (and any edges connected to that vertex) could not have created a new cycle. Therefore the original graph must not have been a DAG. If the original graph had been a DAG, the new graph could not contain a cycle.

DAG shortest paths algorithm:

1. Initialize $\vec{y}$ values to ∞ and set $y_r = 0$; initialize $\vec{p}$ values to -1 and set $p_r = r$.
2. Create a list L with all vertices $v \in V$ ordered in a topological sort.
3. **while** ($L \neq$ empty)
   - *a*) Remove first vertex v from L.
   - *b*) *scan*(v).

We can run the algorithm on the following DAG:



After the first two steps, we have:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
s	∞	-1
a	∞	-1
c	∞	-1
b	∞	-1
d	∞	-1
e	∞	-1

$$L = r \rightarrow s \rightarrow a \rightarrow c \rightarrow b \rightarrow d \rightarrow e \rightarrow$$

We remove r and *scan* it:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
s	∞	-1
a	7	r
c	∞	-1
b	-3	r
d	∞	-1
e	∞	-1

$$L = s \rightarrow a \rightarrow c \rightarrow b \rightarrow d \rightarrow e \rightarrow$$

We remove s and *scan* it, but because $y_s = \infty$ this has no effect on $\vec{y}$ or $\vec{p}$. So we remove a and *scan* it:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
s	∞	-1
a	7	r
c	9	a
b	-3	r
d	14	a
e	∞	-1

$$L = c \rightarrow b \rightarrow d \rightarrow e \rightarrow$$

We remove c and *scan* it:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
s	∞	-1
a	7	r
c	9	a
b	-9	c
d	3	c
e	∞	-1

$$L = b \rightarrow d \rightarrow e \rightarrow$$

We remove b and *scan* it:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
s	∞	-1
a	7	r
c	9	a
b	-9	c
d	-5	b
e	-7	b

$$L = d \rightarrow e \rightarrow$$

We remove d and *scan* it, which has no effect; nor does e, so we are done. A topological sort can be done in time $= O(n)$. It should therefore be clear that this shortest paths algorithm will terminate in time $= O(m + n)$ ($m = |E|$ and $n = |V|$).

For the proof of this algorithm's correctness, we note first that any vertices v not connected to r will have $y_v = \infty$ at initialization and will not change. For finite-length paths to vertices v (connected to r) from r, since edges are relaxed in order according to the topological sort, the vertices in the path will appear in

the same order as they appear in the topological sort (for any path from r to v found by the algorithm, only vertices preceding v in the topological sort may appear in that path).

For any shortest path P from r to v_k (found by the algorithm), we know that the shortest paths to all vertices between r and v_k (subpaths of P) in that path would have been found first, in order according to the topological sort. Clearly the first subpath found (from r to v_1 is just one edge, and is found correctly by the algorithm (if there is a lower-cost single edge alternative to that subpath, it would have been found before progressing through the topological sort to v; any shorter multiple-edge path would include preceding vertices in the sort, since there are no negative-cost edges). The next subpath found (from r to v_2) would be two edges long, and would consist of the shortest path from r to v_1 and from v_1 to v_2 (it is the shortest path from v_1 to v_2 for the same reasons the previous path (from r to v_1) must be shortest. Progressing through the algorithm in this way, we build the path from r to v_k by putting together subpaths, which are each shortest paths. Therefore our final result must be the shortest path from r to v_k.

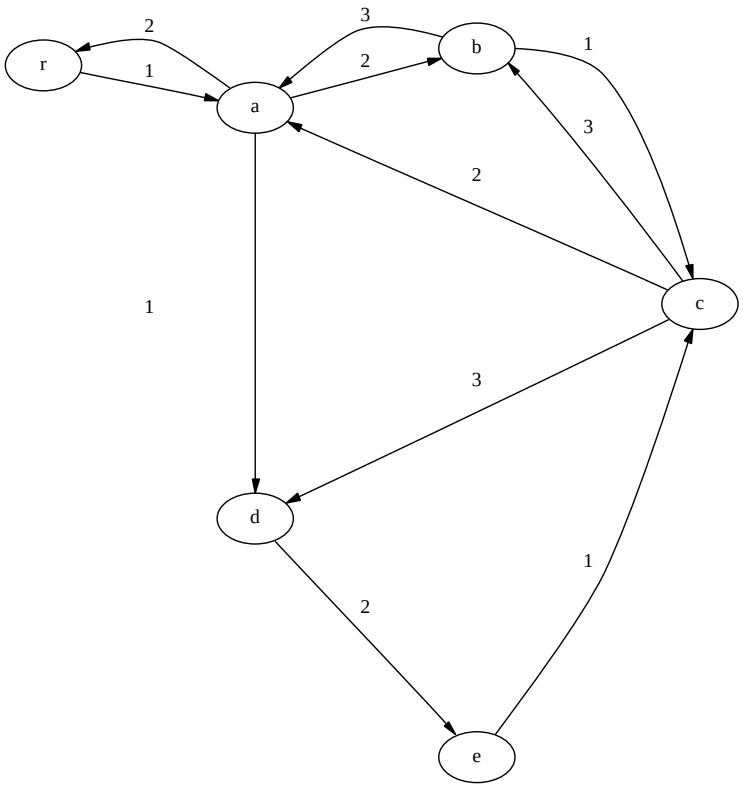
2.3 Dijkstra's Algorithm

The most important special case we consider of digraphs without negative cycles is that of digraphs with no negative weights ($c_{ij} \geq 0 \ \forall i, j$). Edge weights representing real-world cost, weight, capacity, time, distance, etc. will always be non-negative, so many (most) practical problems fit into this special case. These are the graphs for which Dijkstra's Algorithm may be used to solve the shortest paths problem (note: for Dijkstra's Algorithm we require a "priority queue" of vertices ordered by their $\vec{y}$ values—this just means we are able to remove from the queue the vertex with minimum $\vec{y}$ value).

Dijkstra's Algorithm:

1. Initialize $\vec{y}$ values to ∞ and set $y_r = 0$; initialize $\vec{p}$ values to -1 and set $p_r = r$.
2. Put all vertices $v \in V$ into a priority queue keyed by $\vec{y}$ values.
3. **while** (queue $\neq$ empty)
   - *a*) Remove vertex v with minimum y_v from the queue.
   - *b*) *scan*(v).

We can run Dijkstra's Algorithm on the following graph, which has no negative edge weights (notice the graph does contain cycles):



After the first two steps, we have:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	∞	-1
b	∞	-1
c	∞	-1
d	∞	-1
e	∞	-1

queue $= r \to c \to a \to b \to d \to e$
(a, b, c, d, e in some aribtrary order)

We remove r, *scan* it, and update accordingly:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	1	r
b	∞	-1
c	∞	-1
d	∞	-1
e	∞	-1

queue $= a \to c \to b \to d \to e$
(a is now at the head of the queue since y_a is the minimum $\vec{y}$ value of vertices in the queue)

We remove a, *scan* it, and update:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	1	r
b	3	a
c	∞	-1
d	2	a
e	∞	-1

queue $= d \to b \to c \to e$

We remove d, *scan* it, and update:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	1	r
b	3	a
c	∞	-1
d	2	a
e	4	d

queue $= b \to e \to c$

We remove b, *scan* it, and update:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	1	r
b	3	a
c	4	b
d	2	a
e	4	d

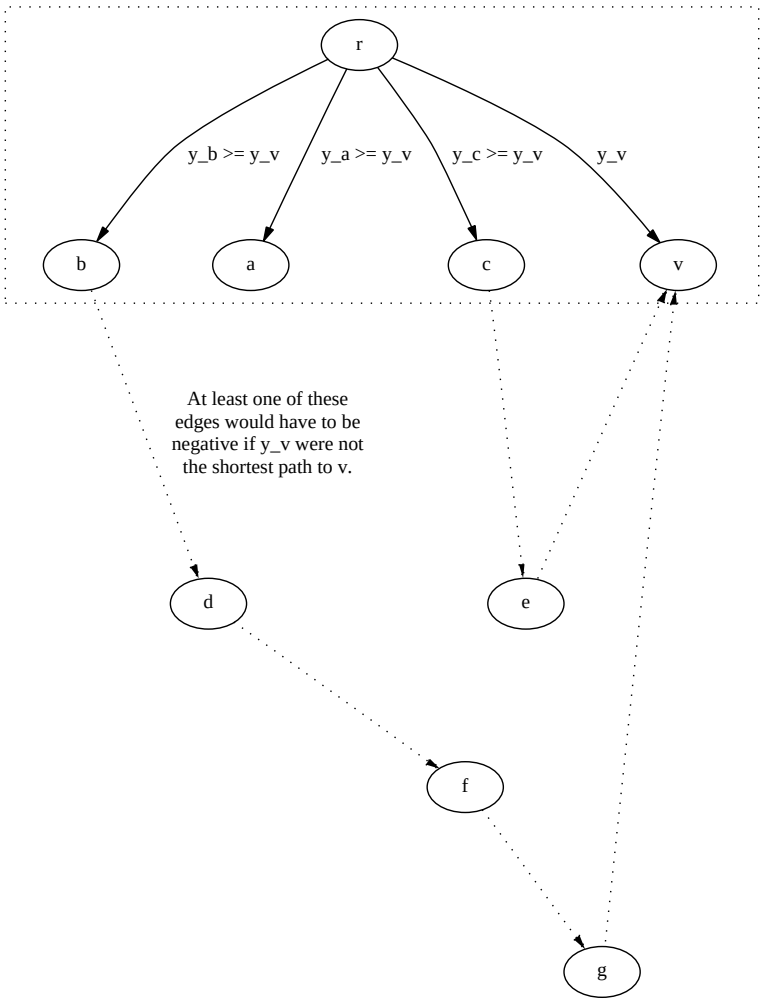
queue $= e \to c$

We remove e and *scan* it, but no changes result; nor do any changes result from a *scan* of c. So we are done. For the proof of Dijkstra's Algorithm, we begin with the following lemma:

Lemma:

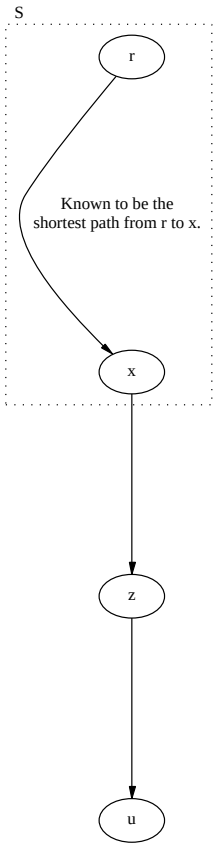
> When (during the execution of Dijkstra's Algorithm on a digraph with no negative edges) vertex v is extracted from the queue, y_v is the length of the shortest path from r to v.

The lemma is trivially true when $v = r$. When r is removed from the queue, $y_r = 0$ (from initialization), and the shortest path from r to r (obviously) has length $= 0$. For vertices $v \neq r$, we know that y_v is the minimum $\vec{y}$ value of all vertices in the queue (if we think of the queue as the set of vertices that have been reached but not yet scanned, of those vertices v is the closest to r). If there were a path to v shorter than y_v, it would have to go through one of the other "frontier" (reached but not yet scanned) vertices in the queue at the time v was removed. But we know that these vertices' $\vec{y}$ values are all $\geq y_v$. So if there is a path through one of these that eventually comes back to v, it must have a negative edge somewhere along the way or else it can not be shorter than our original y_v. Since the digraph has no negative edges, the lemma must be true for all $v \in V$.



To prove Dijkstra's Algorithm (by contradiction), let u be the first vertex for which the lemma is false, i.e. vertex u is extracted from the queue, but there is some path from r to u shorter than y_u. Let the actual shortest path from r to u have length d_u, where $d_u < y_u$. If r is not connected to u, $d_u = y_u = \infty$, we have a contradiction, and the lemma is proved. If r is connected to u, the the shortest path from r to u must exist.

Let S be the set of vertices for which y_v is known to be the shortest path for every vertex $v \in S$. Let $x \in S$ be the vertex through which the shortest path to u runs. Let z be a vertex in the path between x and u (if the lemma is false and $y_u \neq$ the length of the shortest path to u at the time u is extracted from the queue, the true shortest path must exist to be found later after some other intermediate vertex z has been extracted):



If the path shown above is truly the shortest path from r to u, then y_z must be the length of the shortest path from r to z, and the path from z to u must also be the shortest. But if all this is true, z would definitely have been extracted from the queue before u, so that at the time u is extracted z is already in S, and therefore the y_u value when u is eventually extracted is in fact the length of the shortest path from r to u.

2.4 Determining the Actual Shortest Path (not just the length—the *actual* path)

Suppose we have an oracle (or "black box" algorithm) that gives us $\vec{y}$. This tool can be used to find the actual shortest path (the intermediate vertices and edges) from r to $v \in V$ (with length $= y_v$). We would run our oracle and check y_v over and over, each time deleting an edge from the original graph. If deleting an edge causes y_v to increase, we put it back. When we get to the point where no edge can be deleted without increasing y_v, we are left with the actual path from r to v.

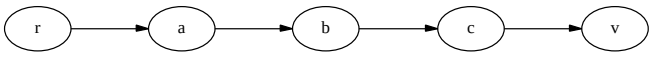
3 Shortest Paths Problem General Case (Bellman-Ford Algorithm)

In general a digraph may have negative cycles. The Ford Procedure discussed earlier will not terminate when run on a digraph with a negative cycle. The Bellman-Ford algorithm solves this problem, but before discussing it we need the concept of "embedding":

Embedding:

A path P from r to v is said to be "embedded" in an edge sequence S if edges in P appear in S in the same order as they appear in P.

The shortest paths algorithms (and procedure) discussed so far all proceed by relaxing edges in some sequence. The Ford Procedure uses an arbitrary order, the *DAG* procedure a topological sort, Dijkstra's Algorithm uses a priority queue of $\vec{y}$ values, etc. If we had some way of ordering the edges in a graph so that the relaxation sequence followed the path from r to v, this would give us the shortest path on the first try (a similar idea was used in the discussion of the Ford Procedure to show that a shortest path may not contain cycles). For example, this graph shows the shortest path from r to v:



If we could guarantee that our shortest paths algorithm relaxed edges consecutively $e_{ra}, e_{ab}, e_{bc}, e_{cv}$, we could know for sure that we had the shortest path after passing across the graph only once. In general, it is the order of relaxations that is important, not that they be done consecutively. So what we really want is for all shortest paths to be *embedded* in the sequence of edges relaxed by our algorithm.

How can we sort edges so that our relaxations proceed in the right order? We start by selecting edges beginning at r. Clearly all shortest paths must start with edges beginning at r. Next we select edges beginning one away from r. Again it should be obvious that the second edge in any shortest path will be a second edge from r. In general, the maximum length of a shortest path will be $n-1$, where n is the number of vertices in the graph. So in constructing our edge sequence for relaxations, we know we have finished when we come to the set of edges $n-1$ away from r.

The Bellman-Ford Algorithm creates the edge sequence described in the last paragraph and then goes through doing the relaxations in order. There is a finite number of relaxations to be done, so we know the algorithm will terminate.

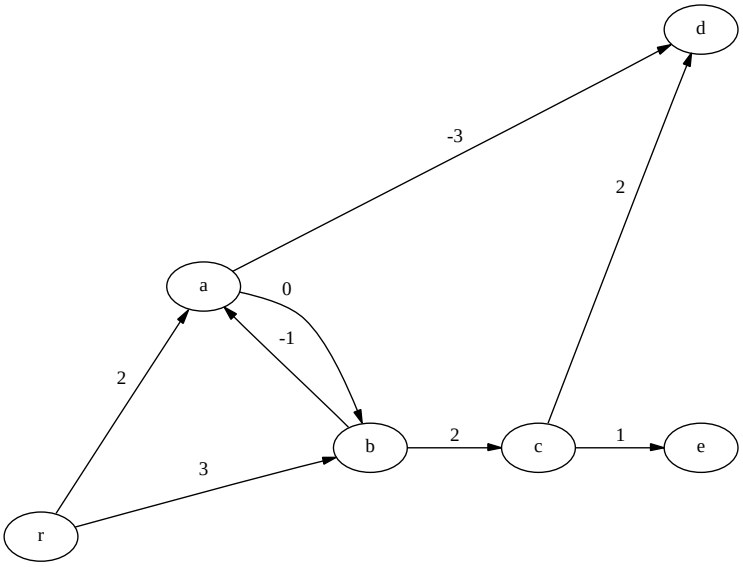
Bellman-Ford Algorithm:

1. Initialize $\vec{y}$ values to ∞ and set $y_r = 0$; initialize $\vec{p}$ values to -1 and set $p_r = r$.
2. Create a sequence of edge sets $S_1 S_2 \ldots S_{n-1}$ ordered by the number of edges away from r (described above in detail).
3. **for** ($i = 1$ to $n-1$)

   Relax all edges $e \in S_i$.
4. **if** ($\vec{y}$ not feasible)

   return "There is a negative cycle."

We can apply the algorithm to the following digraph:



We begin by selecting edges to put into the set S_1 (edges connected to r). These will be e_{ra} and e_{rb}. S_2 will get edges $e_{ab}, e_{ad}, e_{ba}, e_{bc}$, and so on, until we have:

$$
\begin{aligned}
& S_1 = \{e_{ra}, e_{rb}\} \\
& S_2 = \{e_{ab}, e_{ad}, e_{ba}, e_{bc}\} \\
& S_3 = \{e_{ab}, e_{ad}, e_{ba}, e_{bc}, e_{cd}, e_{ce}\} \\
& S_4 = \{e_{ab}, e_{ad}, e_{ba}, e_{bc}, e_{cd}, e_{ce}\} \\
S_{n-1} = \quad & S_5 = \{e_{ab}, e_{ad}, e_{ba}, e_{bc}, e_{cd}, e_{ce}\}
\end{aligned}
$$

Notice that the edges from S_2 reappear in S_3, S_4 and S_5. This is because of the cycle between vertices a and b. After initializing $\vec{y}$ and $\vec{p}$, we have:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	∞	-1
b	∞	-1
c	∞	-1
d	∞	-1
e	∞	-1

Relaxing the edges in S_1 (e_{ra}, e_{rb}) gives us:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	2	r
b	-3	r
c	∞	-1
d	∞	-1
e	∞	-1

Relaxing the edges in S_2, we have:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	-4	b
b	-4	r
c	-2	b
d	-7	a
e	∞	-1

We relax the edges in S_3:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	-5	b
b	-5	a
c	-3	b
d	-8	a
e	-2	c

We relax the edges in S_4:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	-6	b
b	-6	a
c	-4	b
d	-9	a
e	-3	c

And finally, we relax the edges in S_5 $(= S_{n-1})$:

$v \in V$	$\vec{y}$	$\vec{p}$
r	0	r
a	-7	b
b	-7	a
c	-5	b
d	-10	a
e	-4	c

Notice that with each of the iteration, all $\vec{y}$ values seem to be decreasing by 1. This is because we have a negative cycle between vertices a and b. If we were running the Ford Procedure, it would continuing relaxing and decreasing $\vec{y}$ values forever, but the Bellman-Ford Algorithm only does relaxations up through the list of edges in S_{n-1}. After finishing relaxations, Belman-Ford must verify that $\vec{y}$ is feasible (if $\vec{y}$ isn't feasible, there must be a negative cycle). So at this point we check to see whether $y_v \leq y_u + cost(e_{uv})$, where $u = p_v$. We skip the trivial case where $v = u = r$, and try $v = a$ and $u = p_v = b$:

$$y_a \leq y_b + cost(e_{ba})$$
$$-7 \leq -7 + -1$$
$$-7 \leq -8$$

Obviously the inequality does not hold, indicating that the graph has a negative cycle and therefore its shortest paths are unbounded.

References

[1] William Cook, William H. Cunningham, William Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization.* John Wiley & Sons, 1998.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.