WATCH 03/09 lecture again in detail

Algorithms for Data Science

CSOR W4246

03/09 Lecture

Eleni Drinea
Computer Science Department

Columbia University

Shortest paths in weighted graphs (Bellman-Ford, Floyd-Warshall)

Outline

1. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Implementations
   - Graphs with **negative** edge weights: why Dijkstra fails
2. Single-source shortest paths (negative edges): Bellman-Ford
   - A DP solution
   - An alternative formulation of Bellman-Ford
3. All-pairs shortest paths (negative edges): Floyd-Warshall

Today

1. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Implementations
   - Graphs with **negative** edge weights: why Dijkstra fails
2. Single-source shortest paths (negative edges): Bellman-Ford
   - A DP solution
   - An alternative formulation of Bellman-Ford
3. All-pairs shortest paths (negative edges): Floyd-Warshall

Graphs with **non-negative** weights

Input

- a weighted, directed graph $G = (V, E, w)$; function $w : E \to R^+$ assigns non-negative real-valued weights to edges;
- a source (origin) vertex $s \in V$.

Output: for every vertex $v \in V$

1. the length of a shortest s-v path;
2. a shortest s-v path.

Dijkstra's algorithm (Input: $G = (V, E, w), s \in V$)

Output: arrays $dist$, $prev$ with n entries such that

1. $dist(v)$ stores the length of the shortest s-v path
2. $prev(v)$ stores the node before v in the shortest s-v path

At all times, maintain a set S of nodes for which the distance from s has been determined.

- Initially, $dist(s) = 0$, $S = \{s\}$.
- Each time, add to S the node $v \in V - S$ that
  1. has an edge from some node in S;
  2. minimizes the following quantity among all nodes $v \in V - S$

$$d(v) = \min_{u \in S : (u,v) \in E} \{dist(u) + w(u, v)\}$$

- Set $prev(v) = u$.

Implementation

`Dijkstra-v1`$(G = (V, E, w), s \in V)$

`Initialize`(G, s)
$S = \{s\}$
while $S \neq V$ **do**
Select a node $v \in V - S$ with at least one edge from S so that

$$d(v) = \min_{u \in S, (u,v) \in E} \{dist[u] + w(u, v)\}$$

$S = S \cup \{v\}$
$dist[v] = d(v)$
$prev[v] = u$
end while

`Initialize`(G, s)
for $v \in V$ **do**
$dist[v] = \infty$
$prev[v] = NIL$
end for
$dist[s] = 0$

Improved implementation (I)

Idea: Keep a conservative overestimate of the true length of the shortest s-v path in $dist[v]$ as follows: when u is added to S, update $dist[v]$ for all v with $(u, v) \in E$.

`Dijkstra-v2`$(G = (V, E, w), s \in V)$

  `Initialize`(G, s)
  $S = \emptyset$
  while $S \neq V$ **do**
    Pick u so that $dist[u]$ is minimum among all nodes in $V - S$
    $S = S \cup \{u\}$
    for $(u, v) \in E$ **do**
      `Update`(u, v)
    end for
  end while

`Update`(u, v)

  if $dist[v] > dist[u] + w(u, v)$ **then**
    $dist[v] = dist[u] + w(u, v)$
    $prev[v] = u$
  end if

Improved implementation (II): binary min-heap

Idea: Use a priority queue implemented as a binary min-heap: store vertex u with key $dist[u]$. Required operations: `Insert`, `ExtractMin`; `DecreaseKey` for `Update`; each takes $O(\log n)$ time.

`Dijkstra-v3`$(G = (V, E, w), s \in V)$

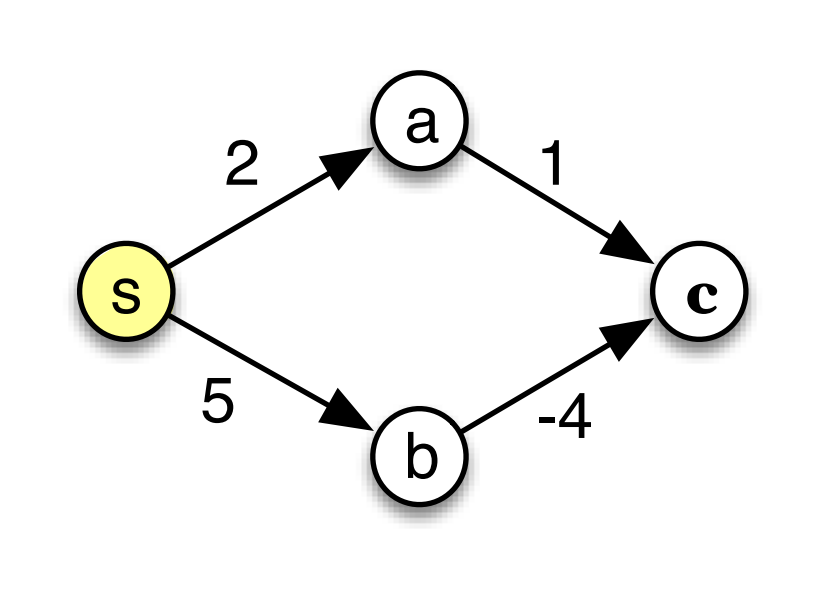
```
Initialize(G, s)
Q = {V; dist}
S = ∅
while Q ≠ ∅ do
    u = ExtractMin(Q)
    S = S ∪ {u}
    for (u, v) ∈ E do
        Update(u, v)
    end for
end while
```

Running time: $O(n \log n + m \log n) = O(m \log n)$

When is `Dijkstra-v3()` *better than* `Dijkstra-v2()`*?*

Example graph with **negative** edge weights

Looking for single-origin shortest paths from s



Initially,

$S = \{s\}$

$dist[s] = 0$

iter 2:

$S = \{s, a\}$

$dist[a] = 2$

iter 3:

$S = \{s, a, c\}$

$dist[c] = 3$

iter 4:

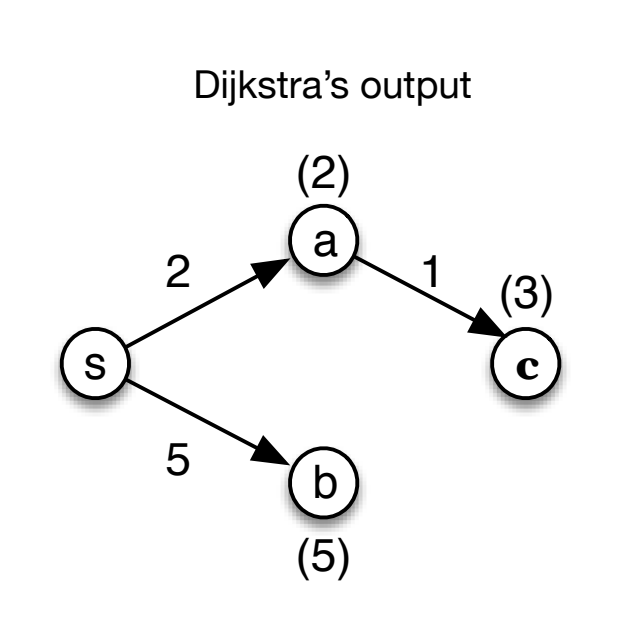
$S = \{s, a, c, b\} = V$

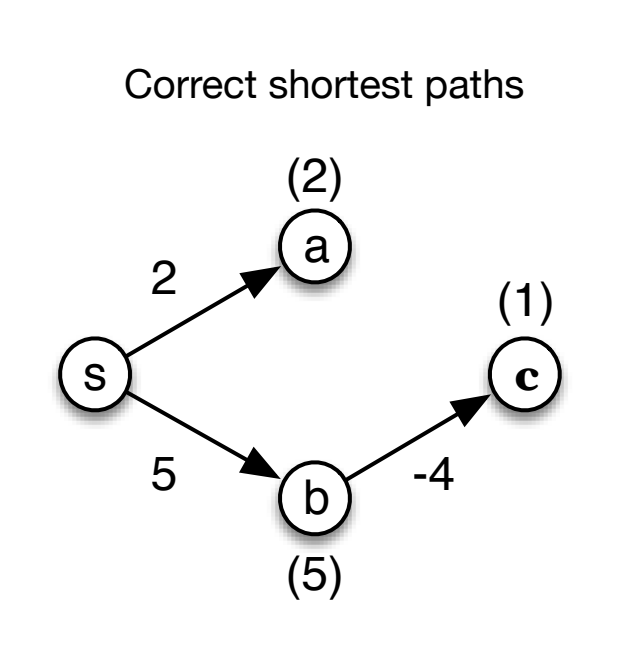
$dist[b] = 5$

Does this terminate w/ the correct output?

No, because dist[c] should be 1 using $s \to b \to c$

Dijkstra's output and correct output for example graph





Dijkstra's algorithm will first include a to S and then c, thus missing the shorter path from s to b to c.

Negative edge weights: why Dijkstra's algorithm fails

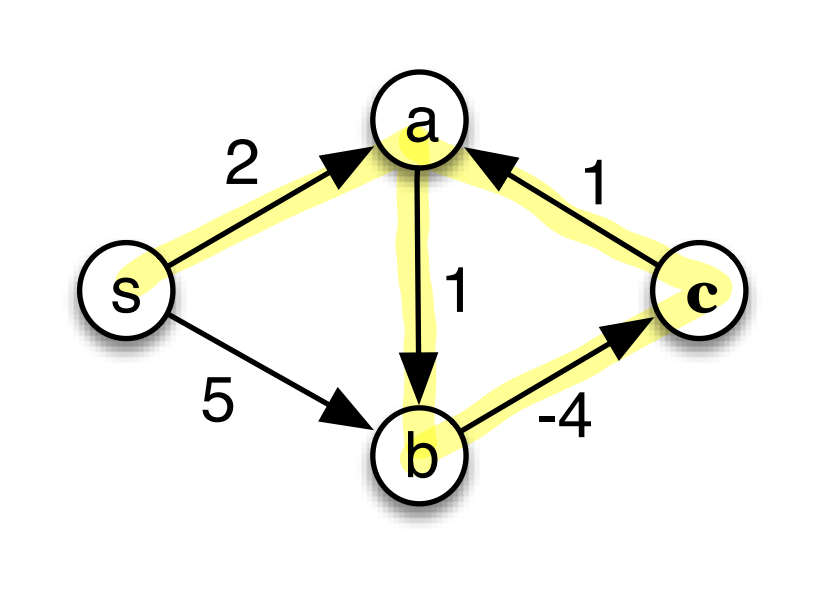
- Intuitively, a path may start on long edges but then compensate along the way with short edges.
- Formally, in the proof of correctness of the algorithm, the last statement about P does not hold anymore: even if the length of path P_v is smaller than the length of the subpath $s \to x \to y$, negative edges on the subpath $y \to v$ may now result in P being *shorter* than P_v.

Bigger problems in graphs with negative edges?

Negative Cycle:

A cycle s.t. when you add the weights of it's edges, it's a negative #.

$a \to b \to c \to a$ is a negative cycle.



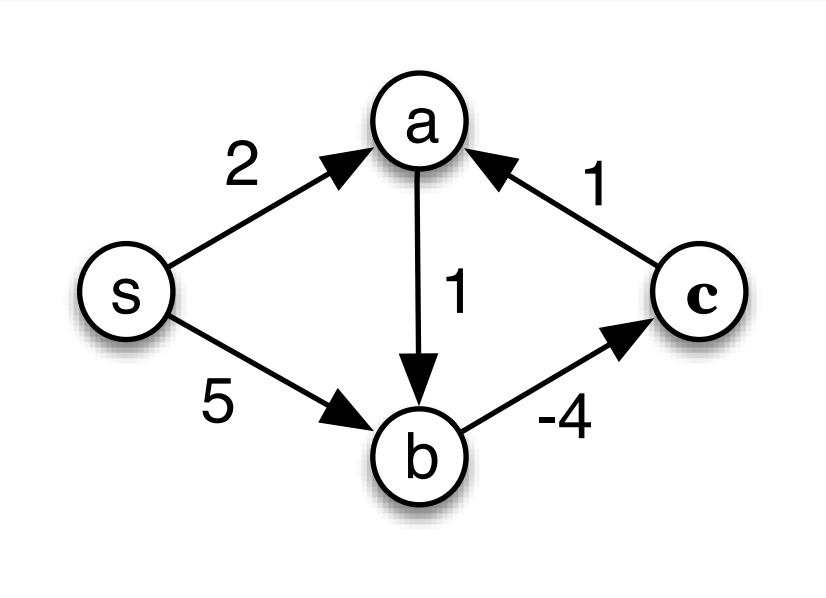
$dist(a) =?$ 2

$dist(b) = 3$

$dist(c) = -1$

dist[a] initially seems to be 2, but if you traverse the cycle $a \to b \to c \to a$, dist[a] = 0. If we do it again then dist[a] = -2, ... $\Rightarrow$ dist[a] = $-\infty$

Bigger problems in graphs with negative edges?



1. $dist(v)$ goes to $-\infty$ for every v on the cycle (a, b, c, a)
2. **no** solution to shortest paths when negative cycles

$\Rightarrow$ need to detect negative cycles

Today

1. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Implementations
   - Graphs with **negative** edge weights: why Dijkstra fails
2. Single-source shortest paths (negative edges): Bellman-Ford
   - A DP solution
   - An alternative formulation of Bellman-Ford
3. All-pairs shortest paths (negative edges): Floyd-Warshall

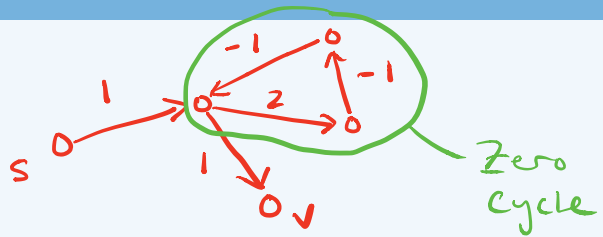
Single-source shortest paths, negative edge weights

Input: weighted directed graph $G = (V, E, w)$ with $w : E \to R$; a source (origin) vertex $s \in V$.

Output:

1. If G has a negative cycle reachable from s, answer **"negative cycle in G"**.
2. Else, compute for every $v \in V$
   - 2.1 the length of a shortest s-v path;
   - 2.2 a shortest s-v path.

Properties of shortest paths



Zero cycles have no impact on shortest paths (e.g. shortest $s \to v$ path) so we can omit zero cycles from a shortest path.

Suppose the problem **has** a solution for an input graph.

- *Can there be negative cycles in the graph?* No
- *Can there be positive cycles in the graph?* Yes
- *Can the shortest paths contain positive cycles?* No
- *Consider a shortest s-t path; are its subpaths shortest? In other words, does the problem exhibit optimal substructure?*

Increases length of path. Can achieve a shorter path by omitting the cycle

Let P be a SIMPLE shortest s-t path.

Now consider any u on the s-t path. If there were a shorter path to u, we could use that path to shorten the overall s-t path, contradicting the shortest path.

For every $v \in V$, $\exists$ simple shortest s-v path. ($\leq n-1$ edges)

FACT: $\forall v \in V$, $\exists$ SIMPLE SHORTEST s-v path (i.e., it has $\leq n-1$ edges)

Consider a shortest s-t path; are its subpaths shortest? In other words, does the problem exhibit optimal substructure?

Let P be a (simple) shortest s-t path.

If there were a shorter path to u, we could use that path instead. Suppose P' is such a a path. Then,

$$w(P'_{su}) < w(P_{su})$$

and thus,

$$w(P'_{su}) + w(P_{ut}) < w(P_{st})$$

Therefore, a shorter s-t path exists. ⚔

CONTRADICTION.

Hence, all subpaths on the shortest path are shortest.

Determine appropriate subproblems for a DP recurrence:

- We are looking for the length of the shortest s-v path for every vertex in the graph.
  - For every vertex in the graph, we have to have a parameter in the subproblem that represents the # of edges on the path we are constructing.

$OPT(i, v) \equiv$ min weight of an s-v path that uses $\leq i$ edges.

$\equiv$ length of a shortest s-v path that uses at most i edges.

We want: $OPT(n-1, v)$, for every $v \in V$

That is, we want:
a shortest s-v path that uses $\leq n-1$ edges

(since we know that a simple shortest path cannot use more than $n-1$ edges.)

Boundary Conditions:

$$OPT(0, v) = \begin{cases} 0, & \text{if } v = s \\ \infty, & \text{if } v \neq s \end{cases}$$

s ; 0^v ← No. edges

└ since no edges, v is not reachable from s.

$$OPT(i,v) = \begin{cases} 0 & , \text{ if } v = s \\ w_{sv} & , \text{ if } (s,v) \in E \\ \infty & , \text{ o.w.} \end{cases}$$

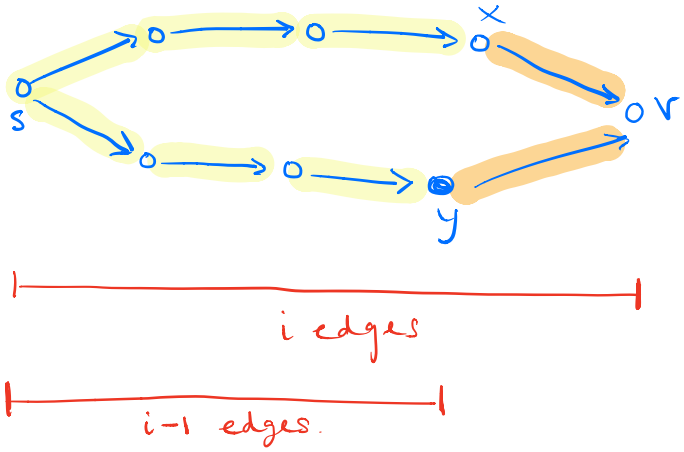
$OPT(i,v) \equiv$ length of a shortest s-v path that uses at most i edges

---

(Progressively allowing more and more edges to improve on shortest path.)

$$OPT(i,v) = OPT(i-1,v)$$, if shortest s-v path that uses $\leq i$ edges actually uses $\leq i-1$ edges

$$OPT(i,v) = \min_{(x,v)\in E} \{ OPT(i-1,x) + w_{xv} \}$$, if shortest s-v path that uses $\leq i$ edges actually uses i edges.



$$\min_{(x,v)\in E} \{ OPT(i-1,v) + w_{xv} \}$$

Assumption: While the graph may have negative edges, it is assumed that there are no negative cycles.

Then,

$$OPT(i,v) = \min \begin{cases} OPT(i-1,v) & O(1) \\ \min\limits_{(x,v)\in E} \{ OPT(i-1,x) + w_{xv} \} & O(1) \quad O(\deg(v)) \end{cases}$$

OR

$$OPT(i,v) = \min \left\{ OPT(i-1,v), \min_{(x,v)\in E} \{ OPT(i-1,x) + w_{xv} \} \right\}$$

Boundary Conditions:

$$OPT(0,v) = \begin{cases} 0 & , \text{if } v = s \\ \infty & , \text{if } v \neq s \end{cases}$$

$$OPT(1,v) = \begin{cases} 0 & , \text{if } v = s \\ w_{sv} & , \text{if } (s,v) \in E \\ \infty & , \text{o.w.} \end{cases}$$

- We do not yet have a recursive algorithm since we don't know which case we are in. Simply minimize over the two cases and we have a recursive algorithm:

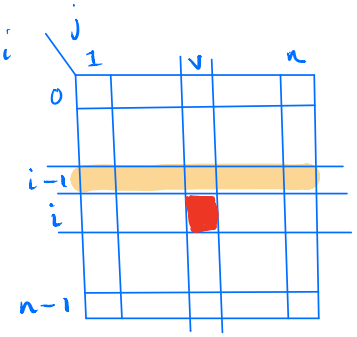
$$OPT(i,v) = \min\left\{ OPT(i-1,v),\ \min_{(x,v)\in E}\left\{ OPT(i-1,x) + w_{xv} \right\} \right\}$$

- # of subproblems: $0 \le i \le n-1$, n vertices $\Rightarrow \Theta(n^2)$ subproblems.

- time per subproblem: $O(n)$

$\Rightarrow$ Total time: $n^2 \cdot O(n) = O(n^3)$

- Order to fill in the matrix M: row-by-row

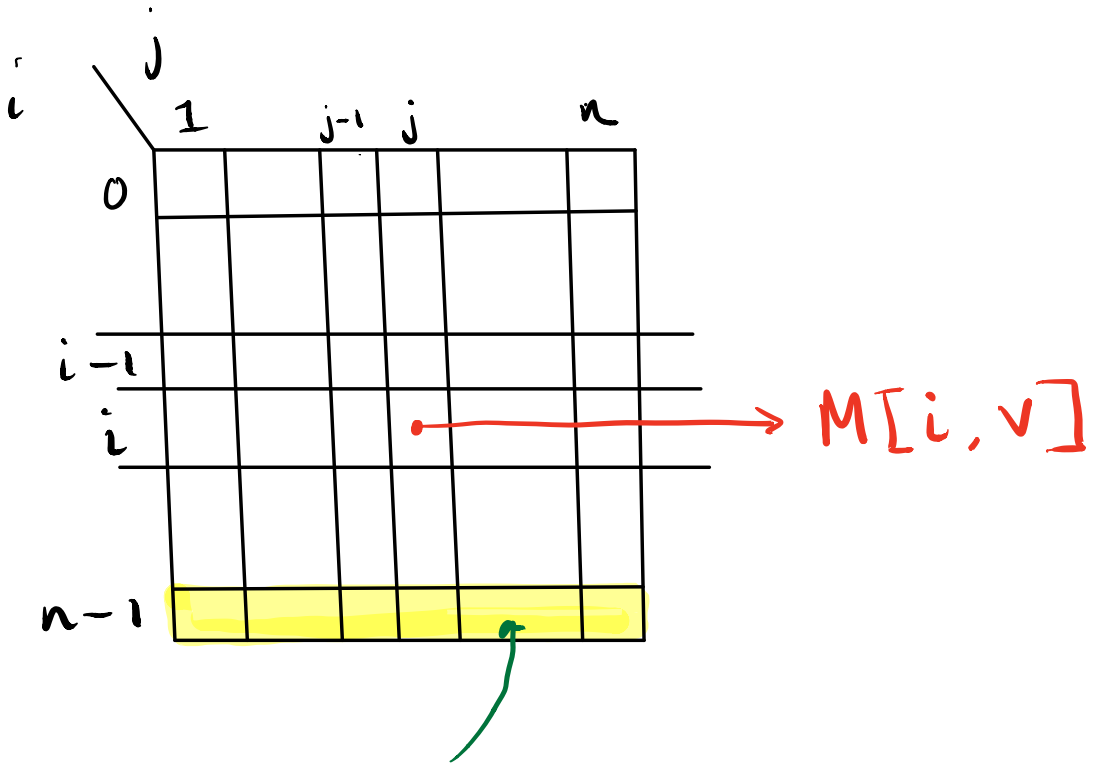


Only need entire row above i to compute $OPT(i,v) = M[i,v]$

[Output lies in the last row.]

M

$M[i,v] = OPT(i,v)$



- We want: entire last row $M[n-1, v]$
- Time per subproblem: $O(\deg(v))$
- # subproblems: $\Theta(n^2)$
- Total time: $O(nm)$

Since, one row of M requires $O(\sum \deg(v)) = O(m)$ and there are n rows.

- Order to fill in M: row-by-row
- Total space: $\Theta(n^2)$ ← Can be improved to using 2 rows.

$$OPT(i,v) = \min\left\{ OPT(i-1,v),\ OPT(i-1,x) + w_{xv} \right\}$$

$O(1)$ to compute (look-up table)

- $O(1)$ for fixed x and v, but → need to compute for every in-neighbor of v.

Whole time is $O(\deg(v))$

Time to fill in row i : $\sum_{v \in V} O(\deg(v)) =$

$= O(m)$

Total time for the algorithm : $O(mn)$

Filling in n rows, (from 0 to $n-1$) and we spend $O(m)$ time to fill in each row.

Towards a DP solution

Key observation: if there are no negative cycles, a path cannot become shorter by traversing a cycle.

Fact 1.

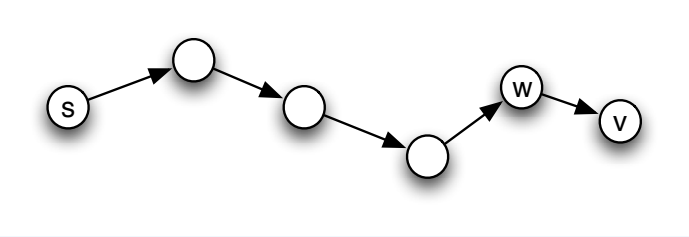
If G has no negative cycles, then there is a shortest s-v path that is simple, thus has at most $n-1$ edges.

Fact 2.

The shortest paths problem exhibits optimal substructure.

Facts 1 and 2 suggest a DP solution.

Subproblems



Let

$$OPT(i, v) = \text{cost of a shortest } s\text{-}v \text{ path with } \textit{at most } i \text{ edges}$$

Consider a shortest s-v path using at most i edges.

- If the path uses at most $i - 1$ edges, then

$$OPT(i, v) = OPT(i - 1, v).$$

- If the path uses i edges, then

$$OPT(i, v) = \min_{x:(x,v)\in E} \{OPT(i - 1, x) + w(x, v)\}.$$

Recurrence

Let

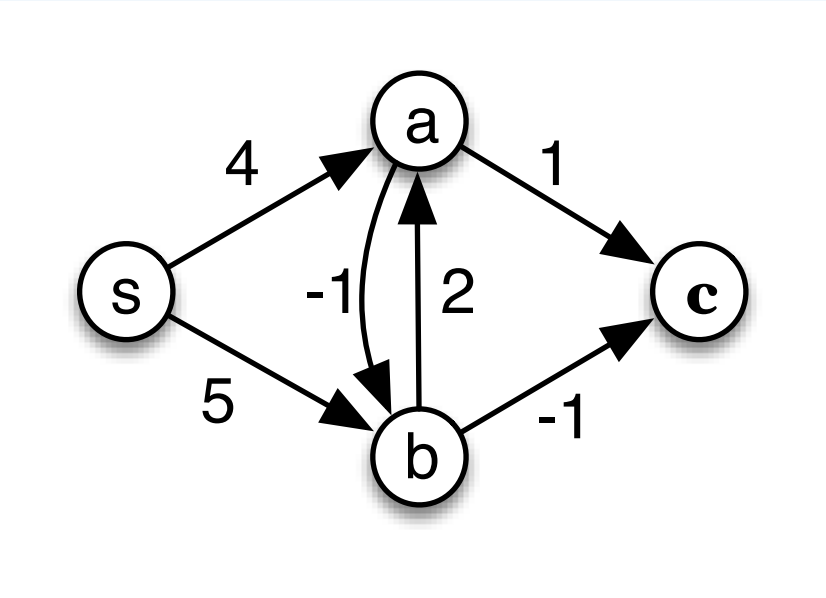
$$OPT(i, v) = \text{cost of a shortest } s\text{-}v \text{ path using } \textit{at most } i \text{ edges}$$

Then

$$OPT(i,v) = \begin{cases} 0 & , \text{if } i = 0, v = s \\ \infty & , \text{if } i = 0, v \neq s \\ \min \begin{cases} OPT(i-1, v) \\ \min\limits_{x:(x,v)\in E} \{OPT(i-1, x) + w(x, v)\} \end{cases} & , \text{if } i > 0 \end{cases}$$

Example of Bellman-Ford

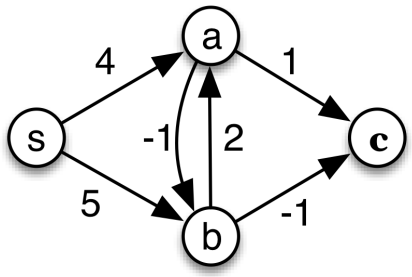
Compute shortest s-v paths in the graph below, for all $v \in V$.



EXAMPLE:

for $i = 1, \ldots, n-1$ **do**
  for $v \in V$ *(in any order)* **do**

$$M[i,v] = \min \left\{ \begin{array}{l} M[i-1,v] \\ \min\limits_{x:(x,v)\in E} \left\{ M[i-1,x] + w(x,v) \right\} \end{array} \right.$$



length of the shortest s-s path using at most 0 edges.

i \ v	s	a	b	c
0	0	∞	∞	∞
1	0	4	5	∞
2	0	4	3	4
3	0	4	3	2

$= M$

Note: Since there are 4 vertices in the graph, the greatest # of edges in a shortest path is 3

$$M[2,b] = \min\left\{ \underbrace{M[1,b]}_{=5}, \min\left\{ \underbrace{M[1,s] + w_{sb}}_{=0+5=5}, \underbrace{M[1,a] + w_{ab}}_{4-1=3} \right\} \right\}$$

$$= \min\{5, \min\{5,3\}\} = \min\{5,3\}$$

$$= 3 = M[2,b]$$

$$M[2,c] = \min\left\{ \underbrace{M[1,c]}_{=\infty}, \min\left\{ \underbrace{M[1,a] + w_{ac}}_{=4+1=5}, \underbrace{M[1,b] + w_{bc}}_{=5-1=4} \right\} \right\}$$

$$= \min\{\infty, \min\{5,4\}\} = 4 = M[2,c]$$

$$M[3,c] = \min\left\{ M[2,c], \min\left\{ M[2,a] + w_{ac}, M[2,b] + w_{bc} \right\} \right\}$$

$$= \min\{4, \min\{4+1, 3-1\}\} = 2 = M[3,c]$$

Pseudocode

$n \times n$ dynamic programming table M such that $M[i, v] = OPT(i, v)$.

`Bellman-Ford`$(G = (V, E, w), s \in V)$

for $v \in V$ **do**
  $M[0, v] = \infty$ ← Initialize first row.
end for
$M[0, s] = 0$
for $i = 1, \ldots, n-1$ **do**
  for $v \in V$ *(in any order)* **do**

$$M[i, v] = \min \left\{ \begin{array}{c} M[i-1, v] \\ \min\limits_{x:(x,v)\in E} \left\{ M[i-1, x] + w(x, v) \right\} \end{array} \right.$$

  end for
end for

Running time & Space

- We improve space by only using 2 rows to compute the final row.

- **Running time:** $O(nm)$
- **Space:** $\Theta(n^2)$ —can be improved (*coming up*)
- To reconstruct actual shortest paths, also keep array $prev$ of size n such that

  $prev[v]$ = predecessor of v in current shortest s-v path.

Improving space requirements

Only need two rows of M at all times.

△ Actually, only need one (see Remark 1)! Thus drop the index i from $M[i, v]$ and only use it as a counter for #repetitions.

for $i = 1, \ldots, n-1$ **do**
  for $v \in V$ *(in any order)* **do**

$$M[v] = \min\left\{M[v], \min_{x:(x,v)\in E}\left\{M[x] + w(x, v)\right\}\right\}$$

↑ equiv. to dist[v] from Dijkstra's Algorithm

Remark 1.

Throughout the algorithm, $M[v]$ is the length of some s-v path. After i repetitions, $M[v]$ is no larger than the length of the current shortest s-v path with at most i edges.

Early termination condition: if at some iteration i **no** value in M changed, then stop (*why?*)

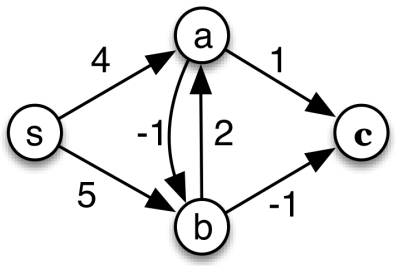
∗ This allows us to detect negative cycles!

Using modified recurrence on the same graph:

for $i = 1$ to $n-1$:
  for $v \in V$:

$$M[v] = \min\left\{M[v], \min_{x:(x,v)\in E}\{M[x] + w(x,v)\}\right\}$$

↳ dist[v] from Dijkstra's alg.



Initialize:

$(i=0)$ $M =$

s	a	b	c
0	∞	∞	∞

$i = 1$:

$M =$

s	a	b	c
0	4	3	2

$M[s] = 0$ (No incoming edges to s)

$M[a] = \min\{M[a], \min\{M[s] + w_{sa}, M[b] + w_{ba}\}\}$

$= \min\{\infty, \min\{0+4, \infty+2\}\} = 4$

$M[b] = \min\{M[b], \min\{M[s] + w_{sb}, M[a] + w_{ab}\}\}$

$= \min\{\infty, \min\{0+5, 4-1\}\} = 3$

$M[c] = \min\{M[c], \min\{M[a] + w_{ac}, M[b] + w_{bc}\}\}$

$= \min\{\infty, \min\{4+1, 3-1\}\} = 2$

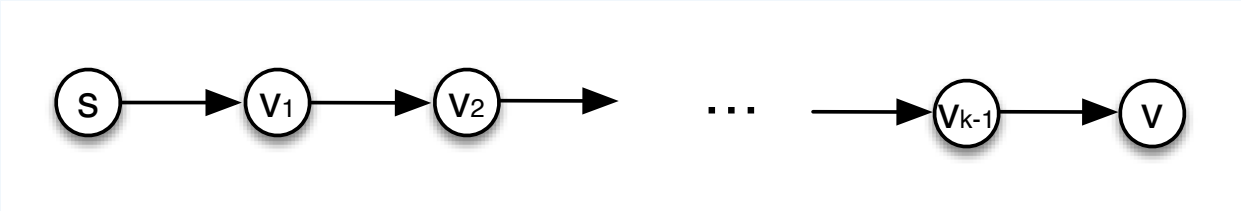
Notice that at the end of iteration 1, we have the lengths of the shortest paths using the modified recursion. However, it also depends on the order in which you update the nodes.

If we had instead updated in the order s, c, b, a, we would have needed 3 iterations. Since we updated in the order $s \to a \to b \to c$, we achieved the shortest paths in 1 iteration. Indeed, the shortest path goes through node a first, then through b and we have already updated a, then to c and we have already updated a and b.

Once the shortest path to all nodes has been reached, the values of M will not change in future iterations: M[u] will remain the same for all $u \in V$.

Early termination condition: If no entry in the array M has changed from one iteration to the next, then you know you have reached shortest paths and can stop.

An alternative way to view Bellman-Ford



- Let $P = (s = v_0, v_1, v_2, \ldots, v_k = v)$ be a shortest s-v path.
- Then P can contain at most $n - 1$ edges.
- *How can we correctly compute $dist(v)$ on this path?*

Key observations about subroutine `Update`(u, v)

Recall subroutine `Update` from Dijkstra's algorithm:

$$\texttt{Update}(u, v) : dist(v) = \min\{dist(v), dist(u) + w(u, v)\}$$

Fact 3.

Suppose u is the last node before v on the shortest s-v path, and suppose $dist(u)$ has been correctly set. The call `Update`(u, v) *returns the correct value for $dist(v)$.*

Fact 4.

No matter how many times `Update`(u, v) *is performed, it will never make $dist(v)$ too small. That is,* `Update` *is a safe operation: performing few extra updates can't hurt.*

Performing the correct sequence of updates

Suppose we update the edges on the shortest path P in the order they appear on the path (though not necessarily consecutively). Hence we update

$$(s, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v).$$

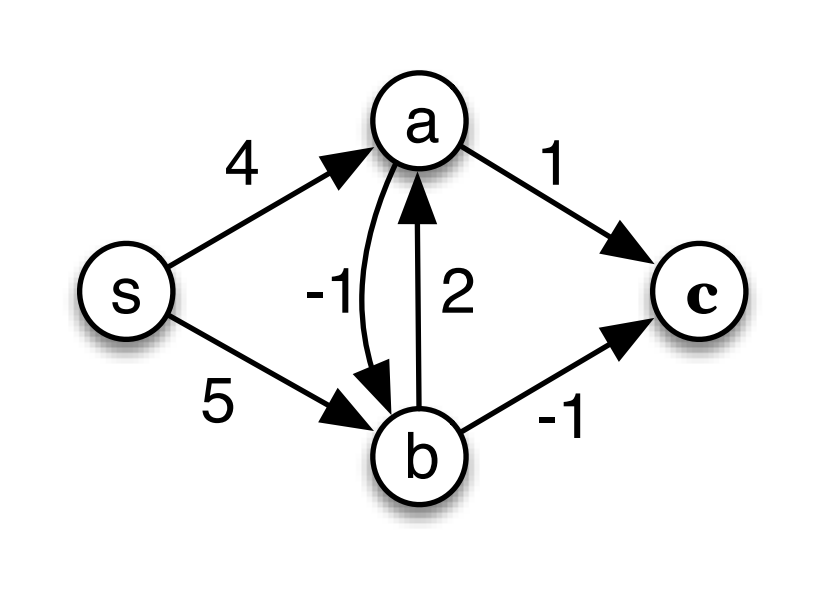
This sequence of updates correctly computes $dist(v_1)$, $dist(v_2)$, ..., $dist(v)$ (by induction and Fact 3).

How can we guarantee that this specific sequence of updates occurs?

A concrete example

Consider the shortest s-b path, which uses edges $(s, a), (a, b)$.

How can we guarantee that our algorithm will update these two edges **in this order***? (More updates in between are allowed.)*



Bellman-Ford algorithm

Update all m edges in the graph, $n-1$ times in a row!

- By Fact 4, it is ok to update an edge several times in between.
- All we need is to update the edges on the path in this particular order. This is guaranteed if we update all edges $n-1$ times in a row.

Pseudocode

We will use `Initialize` and `Update` from Dijkstra's algorithm.

`Initialize`(G, s)

for $v \in V$ **do**
    $dist[v] = \infty$
    $prev[v] = NIL$
end for
$dist[s] = 0$

`Update`(u, v)

if $dist[v] > dist[u] + w(u, v)$ **then**
    $dist[v] = dist[u] + w(u, v)$
    $prev[v] = u$
end if

Bellman-Ford

```
Bellman-Ford(G = (V, E, w), s)
    Initialize(G, s)
    for i = 1, ..., n − 1 do
        for (u, v) ∈ E do
            Update(u, v)
        end for
    end for
```

O(n) → for $i = 1, \ldots, n-1$ do

O(m) { for $(u,v) \in E$ do … end for

O(1) for each

O(nm)

You have n-1 repetitions. For each rep, update $\forall e \in E$ once.

Running time? Space?

Space:

Using 2 arrays M and prev.

$\Rightarrow$ Space = $\Theta(n^2)$

(same as before)

M is the dist array from Dijkstra's algorithm

Running time: $O(nm)$ — Same as before

Space: $\Theta(n^2)$

Both algorithms are identical!

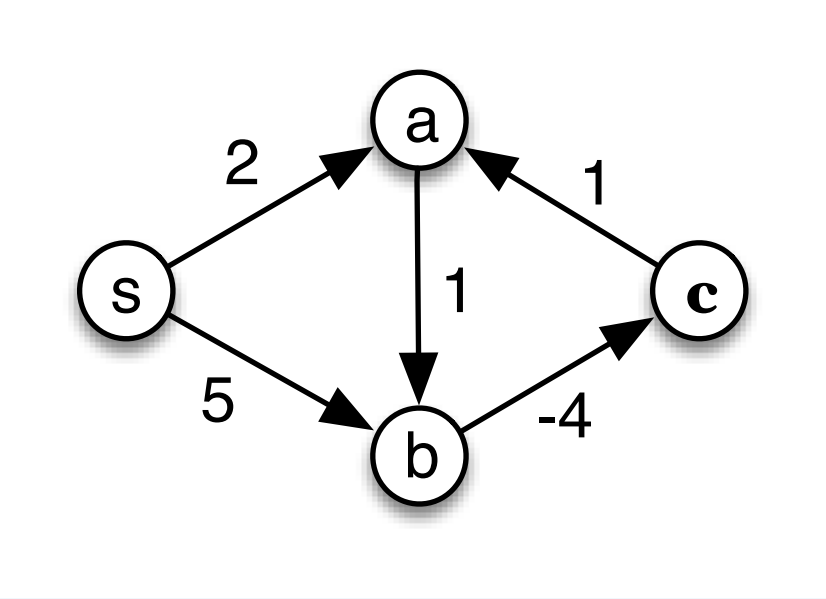
Bellman-Ford algorithm is identical to the DP approach we went through a few slides back.

Bellman-Ford does the same thing as the DP algorithm covered before. This one:

for $i = 1, \ldots, n-1$ **do**
  for $v \in V$ *(in any order)* **do**

$$M[v] = \min\left\{M[v], \min_{x:(x,v)\in E}\{M[x] + w(x,v)\}\right\}$$

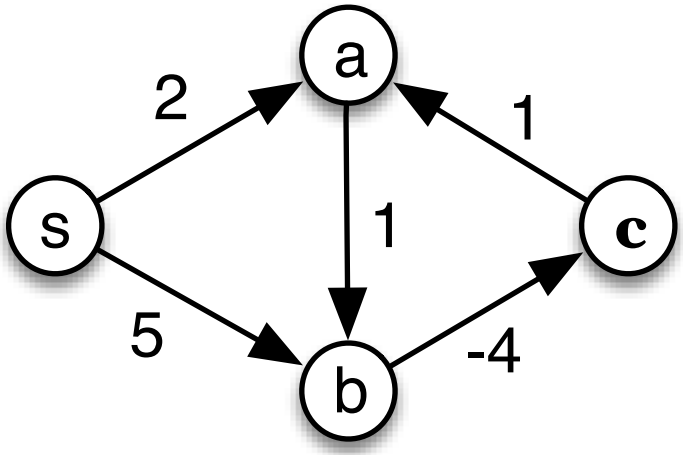
Detecting negative cycles



If G has a negative cycle, then $\exists u \in V$ s.t. $dist[u] \rightarrow -\infty$. If there are no negative cycles the array M will not change after $n-1$ iterations. On the otherhand, if M continues to change after n iterations, then there is a negative cycle.

Detecting negative cycles

If a graph has a negative cycle the distance to any node will go to $-\infty$



Run the algorithm for n iterations. If there are no changes to M from $n-1$ to n, then no neg. cycles. Otherwise, there is.

1. $dist(v)$ goes to $-\infty$ for every v on the cycle.
2. Any shortest s-v path can have at most $n-1$ edges.
3. Update all edges n times (instead of $n-1$): if $dist(v)$ changes **for any** $v \in V$, then there is a negative cycle.

Today

1. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Implementations
   - Graphs with **negative** edge weights: why Dijkstra fails
2. Single-source shortest paths (negative edges): Bellman-Ford
   - A DP solution
   - An alternative formulation of Bellman-Ford
3. All-pairs shortest paths (negative edges): Floyd-Warshall

All pairs shortest-paths

- **Input:** a directed, weighted graph $G = (V, E, w)$ with real edge weights
- **Output:** an $n \times n$ matrix D such that

$$D[i, j] = \text{length of shortest path from } i \text{ to } j$$

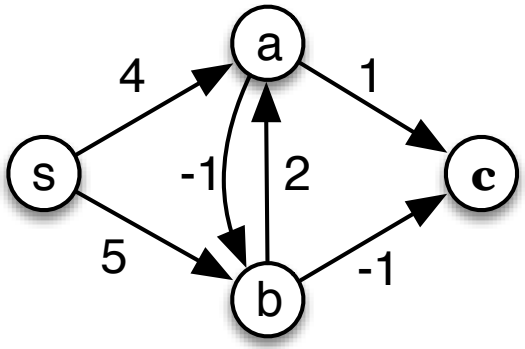
Solving all pairs shortest-paths

1. Straightforward solution: run Bellman–Ford once for every vertex ($O(n^2m)$ time).
2. Improved solution: Floyd-Warshall's dynamic programming algorithm ($O(n^3)$ time).

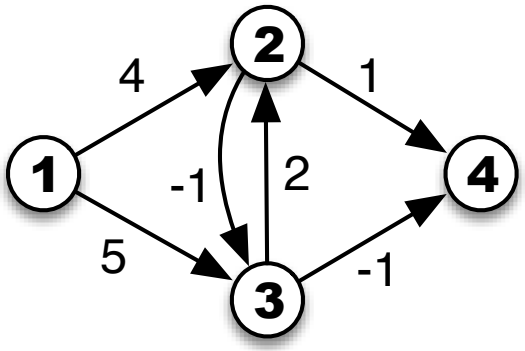
Towards a DP formulation

- Consider a shortest s-t path P.
- This path uses some intermediate vertices: that is, if $P = (s, v_1, v_2, \ldots, v_k, t)$, then $v_1, \ldots, v_k$ are intermediate vertices.
- For simplicity, relabel the vertices in V as $\{1, 2, 3, \ldots, n\}$ and consider a shortest i-j path where intermediate vertices may only be from $\{1, 2, \ldots, k\}$.
- **Goal:** compute the length of a shortest i-j path for every pair of vertices (i, j), using $\{1, 2, \ldots, n\}$ as intermediate vertices.

Example

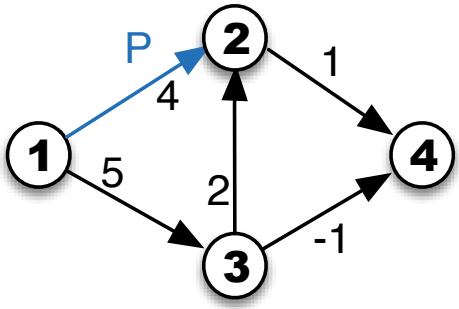


Rename {s, a, b, c} as {**1, 2, 3, 4}**

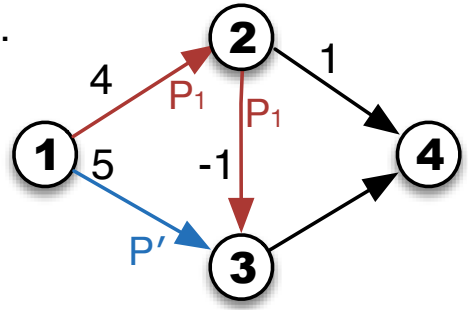


Examples of shortest paths

Shortest (**1, 2**)-path using {} or {1} is P.
Shortest (**1, 2**)-path using {1,2,3,4} is P.

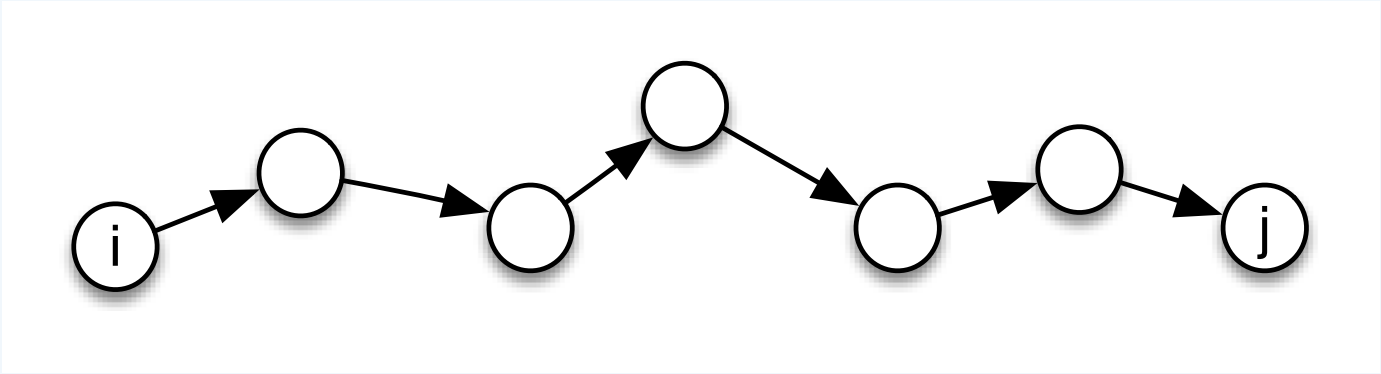


Shortest (**1, 3**)-path using {} or {1} is P′.
Shortest (**1, 3**)-path using {1,2} or {1,2,3} is P_1.
Shortest (**1, 3**)-path using {1,2,3,4} is P_1 .



A shortest i-j path using nodes from $\{1, \ldots, k\}$

Consider a shortest i-j path P where intermediate nodes may only be from the set of nodes $\{1, 2, \ldots, k\}$.



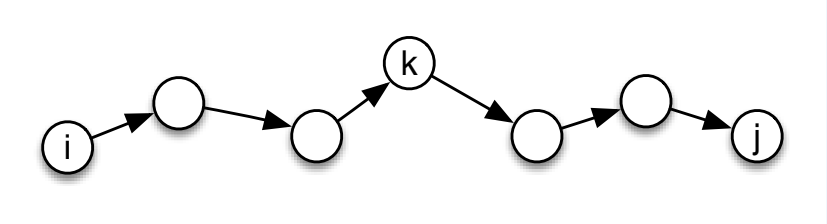
Fact: any subpath of P must be shortest itself.

A useful observation

Focus on the last node k from the set $\{1, 2, \ldots, k\}$. Either

1. P completely avoids k: then a shortest i-j path with intermediate nodes from $\{1, \ldots, k\}$ is the same as a shortest i-j path with intermediate nodes from $\{1, \ldots, k-1\}$.

2. Or, k is an intermediate node of P.



Decompose P into an i-k subpath P_1 and a k-j subpath P_2.

i. P_1, P_2 are shortest subpaths themselves.
ii. All intermediate nodes of P_1, P_2 are from $\{1, \ldots, k-1\}$.

Subproblems

Let

$$OPT_k(i,j) \quad = \quad \text{cost of shortest } i-j \text{ path } P \text{ using } \{1,\ldots,k\} \text{ as intermediate vertices}$$

1. Either k does not appear in P, hence

$$OPT_k(i,j) = OPT_{k-1}(i,j)$$

2. Or, k appears in P, hence

$$OPT_k(i,j) = OPT_{k-1}(i,k) + OPT_{k-1}(k,j)$$

Recurrence

Hence

$$OPT_k(i,j) = \begin{cases} w(i,j) & \text{, if } k = 0 \\ \min \begin{cases} OPT_{k-1}(i,j) \\ OPT_{k-1}(i,k) + OPT_{k-1}(k,j) \end{cases} & \text{, if } k \geq 1 \end{cases}$$

We want $OPT_n(i,j)$.

Time/space requirements?

Floyd-Warshall on example graph

Let $D_k[i,j] = OPT_k(i,j)$.

$D_0 =$

0	4	5	∞
∞	0	-1	1
∞	2	0	-1
∞	∞	∞	0

$D_1 =$

0	4	5	∞
∞	0	-1	1
∞	2	0	-1
∞	∞	∞	0

$D_2 =$

0	4	3	5
∞	0	-1	1
∞	2	0	-1
∞	∞	∞	0

$D_3 =$

0	4	3	2
∞	0	-1	-2
∞	2	0	-1
∞	∞	∞	0

Space requirements

- A single $n \times n$ dynamic programming table D, initialized to $w(i, j)$ (the adjacency matrix of G).
- Let $\{1, \ldots, k\}$ be the set of intermediate nodes that may be used for the shortest i-j path.
- After the k-th iteration, $D[i, j]$ contains the length of some i-j path that is no larger than the length of the shortest i-j path using $\{1, \ldots, k\}$ as intermediate nodes.

The Floyd-Warshall algorithm

```
Floyd-Warshall(G = (V, E, w))
    for k = 1 to n do
        for i = 1 to n do
            for j = 1 to n do
                D[i, j] = min(D[i, j], D[i, k] + D[k, j])
            end for
        end for
    end for
```

- Running time: $O(n^3)$
- Space: $\Theta(n^2)$