Analysis of Algorithms, I

CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

Thursday, March 8, 2016

Outline

1. Recap
   - Single-source shortest paths in graphs with real edge weights: a DP solution (Bellman-Ford)
2. An alternative formulation of the Bellman-Ford algorithm
3. All-pairs shortest paths (real weights): Floyd-Warshall
4. Minimum Spanning Trees (MSTs)
   - Prim's algorithm

Today

1. Recap
   - Single-source shortest paths in graphs with real edge weights: a DP solution (Bellman-Ford)
2. An alternative formulation of the Bellman-Ford algorithm
3. All-pairs shortest paths (real weights): Floyd-Warshall
4. Minimum Spanning Trees (MSTs)
   - Prim's algorithm

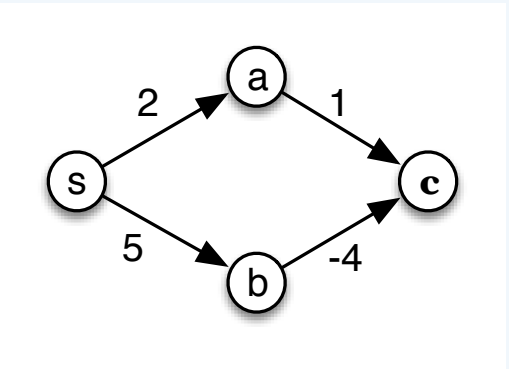
Review of the last lecture

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

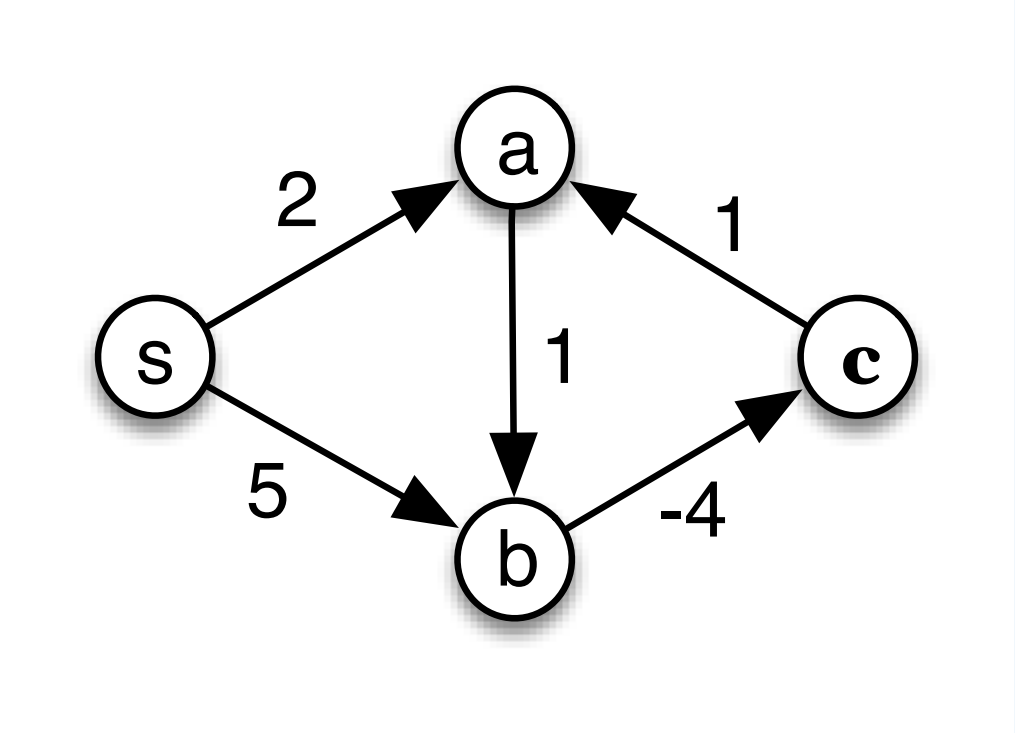
Output:

- Decide if G has a negative cycle
- If no negative cycles
  1. find the costs (weights) of shortest s-v paths for all $v \in V$
  2. reconstruct shortest s-v paths

Why Dijsktra's algorithm fails

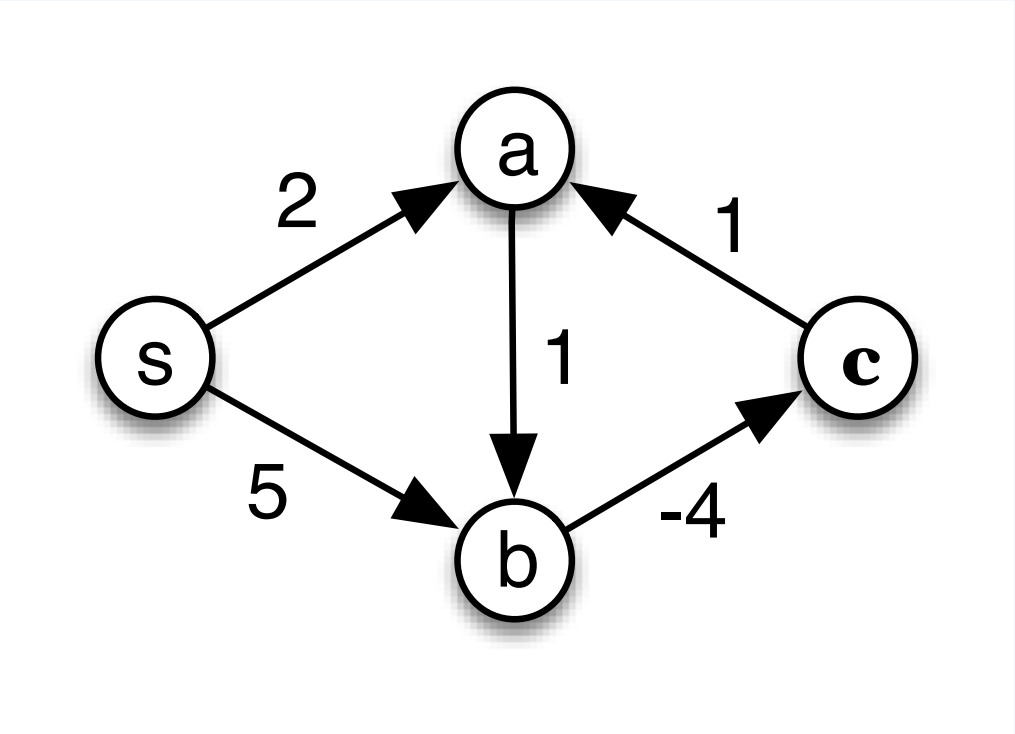


Bigger problems in graphs with negative edges?



$dist(a) = ?$

Bigger problems in graphs with negative edges?



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

$\Rightarrow$ need to detect negative cycles

Properties of shortest paths

Suppose the problem has a solution for an input graph.

- *Can there be negative cycles in the graph?*
- *Can there be positive cycles in the graph?*
- *Can the shortest paths contain positive cycles?*
- *Consider a shortest s-t path; are its subpaths shortest?*

Towards a DP solution

Key observation: if there are no negative cycles, a path cannot become cheaper 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 and recurrence

Let

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

Then

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

Pseudocode

$n \times n$ dynamic programming table M storing $OPT(i, v)$

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

for $v \in V$ **do**

$M[0, v] = \infty$

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

- **Running time:** $O(nm)$
- **Space:** $\Theta(n^2)$ —can be improved (*coming up*)
- To reconstruct actual shortest paths: 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.

△ 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.

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

Remark 1.

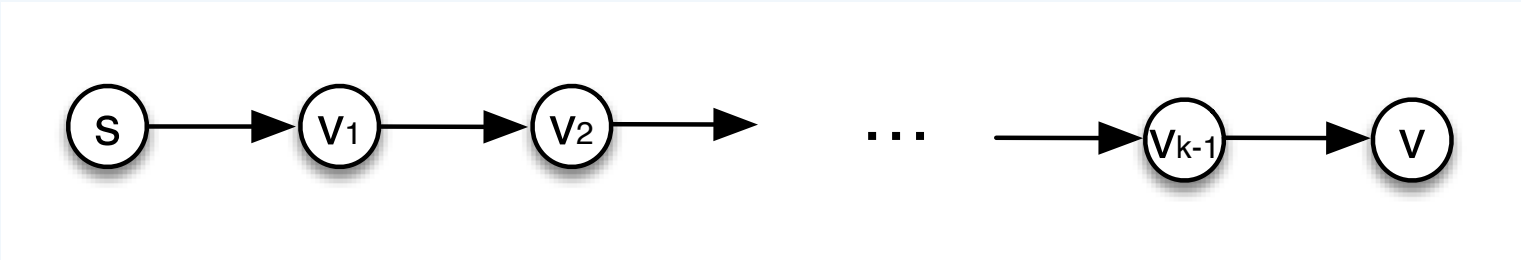
Throughout the algorithm, $M[v]$ is the cost of some s-v path. After i repetitions, $M[v]$ is no larger than the cost of the current shortest s-v path with at most i edges.

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

Today

1. Recap
   - Single-source shortest paths in graphs with real edge weights: a DP solution (Bellman-Ford)
2. An alternative formulation of the Bellman-Ford algorithm
3. All-pairs shortest paths (real weights): Floyd-Warshall
4. Minimum Spanning Trees (MSTs)
   - Prim's algorithm

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.

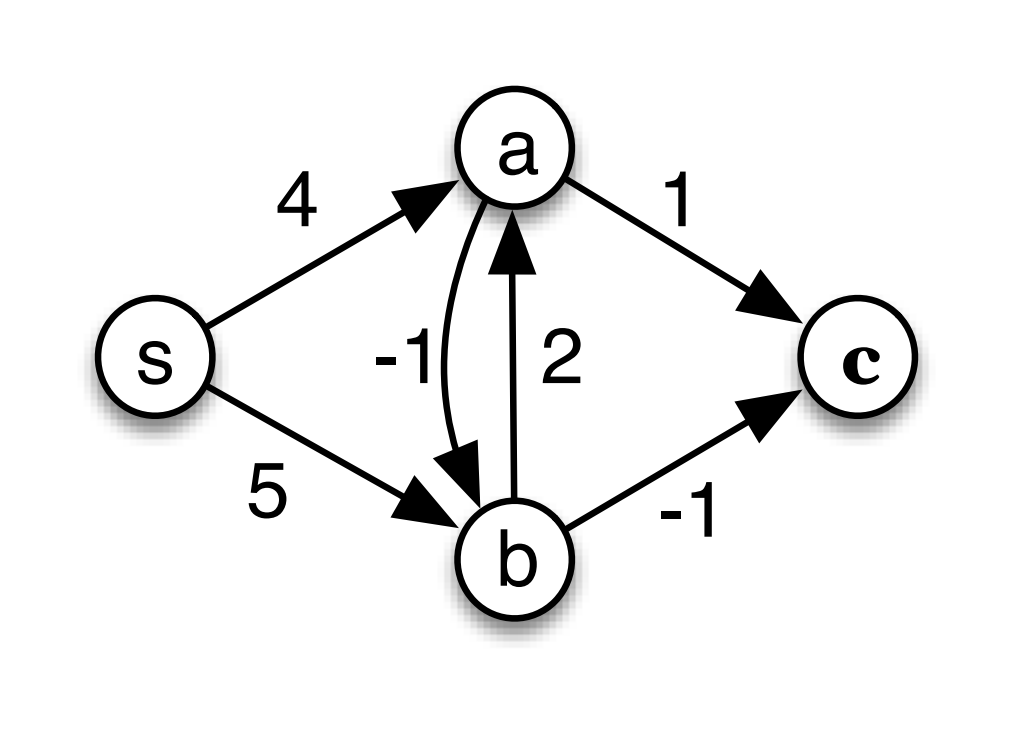
Suppose u is the last node on the shortest s-v path, and suppose $dist[u]$ has been correctly set. The call `Update`(u, v) *returns the correct value for $dist[v]$.*

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.*

Example of Bellman-Ford

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



Main idea

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 updates will happen in this order?

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 care for is that the edges on the path are updated 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)

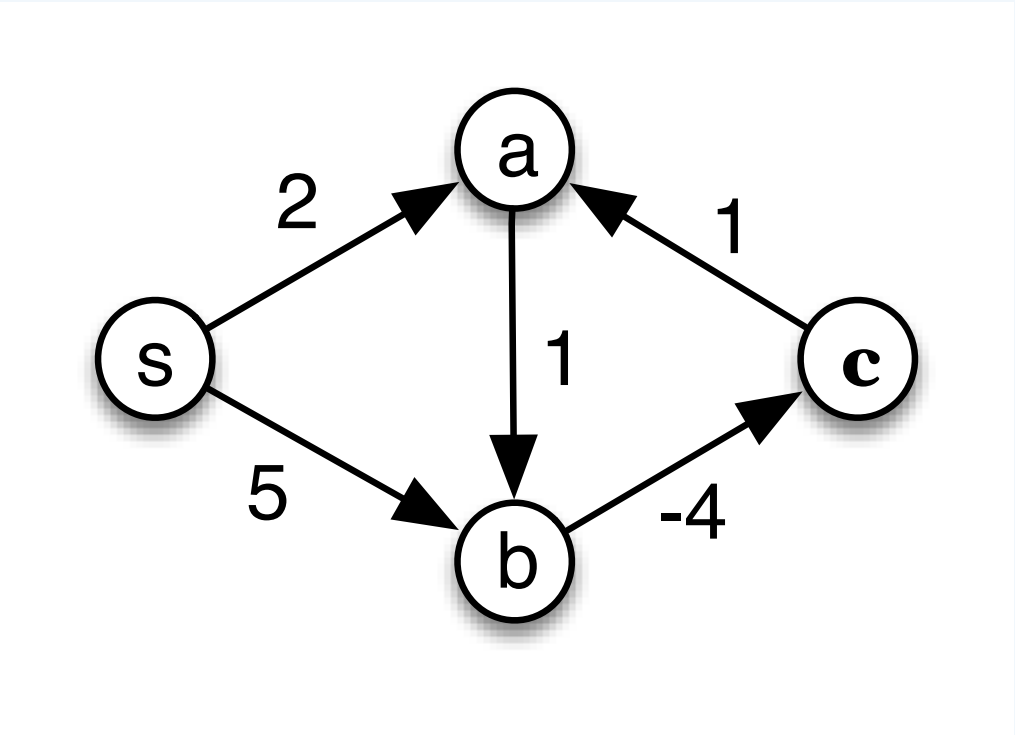
if $dist[v] > dist[u] + w(u, v)$ **then**
    $dist[v] = dist[u] + w(u, v)$
    $prev[v] = u$
end if

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
```

Running time? Space?

Detecting negative cycles



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 every edge n times (instead of $n-1$): if $dist(v)$ changes *for any* $v \in V$, there is a negative cycle.

Today

1. Recap
   - Single-source shortest paths in graphs with real edge weights: a DP solution (Bellman-Ford)
2. An alternative formulation of the Bellman-Ford algorithm
3. All-pairs shortest paths (real weights): Floyd-Warshall
4. Minimum Spanning Trees (MSTs)
   - Prim's algorithm

All pairs shortest-paths

- Input: a directed graph G with real edge weights
- Output: an $n \times n$ matrix D such that

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

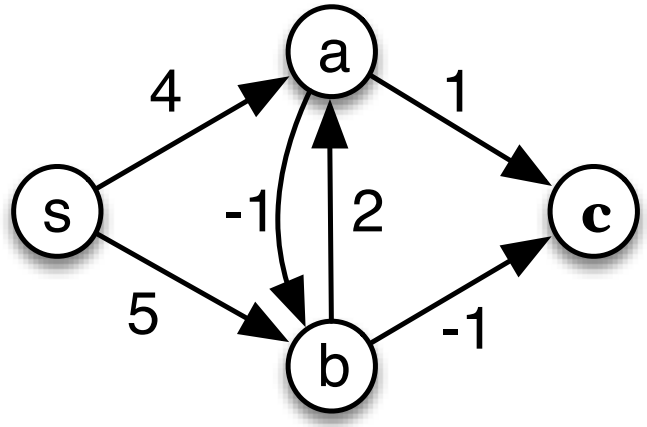
Solving all pairs shortest-paths

1. Straightforward solution: run Bellman–Ford once for every vertex ($O(n^2m)$ time).
2. Improved solution: a dynamic programming algorithm for generating shortest paths (Floyd–Warshall, $O(n^3)$ time).

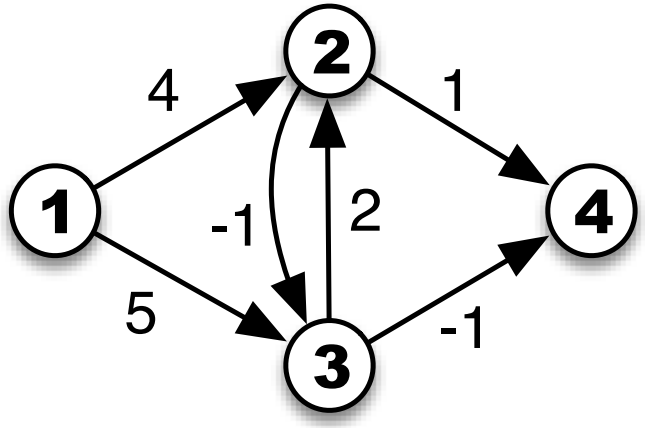
Towards a DP formulation

- Consider a shortest path P from s to t.
- 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, let $V = \{1, 2, 3, \ldots, n\}$ and consider a shortest path from i to j when intermediate vertices may only be from $\{1, 2, \ldots, k\}$.
- **Goal:** find the shortest i-j path using $\{1, 2, \ldots, n\}$ as intermediate vertices.

Example

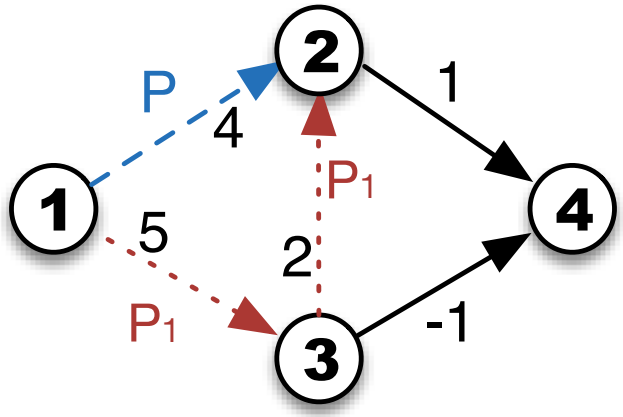


Rename {s, a, b, c} as {**1, 2, 3, 4**}

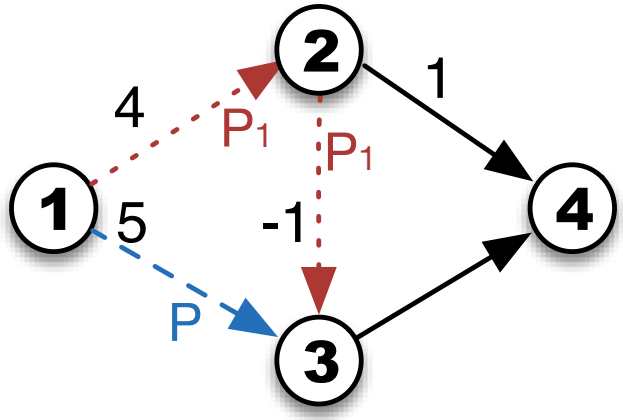


Examples of shortest paths

Shortest (**1, 2**)-path may use:
(i) no intermediate nodes (P); or
(ii) node **3** (P_1)

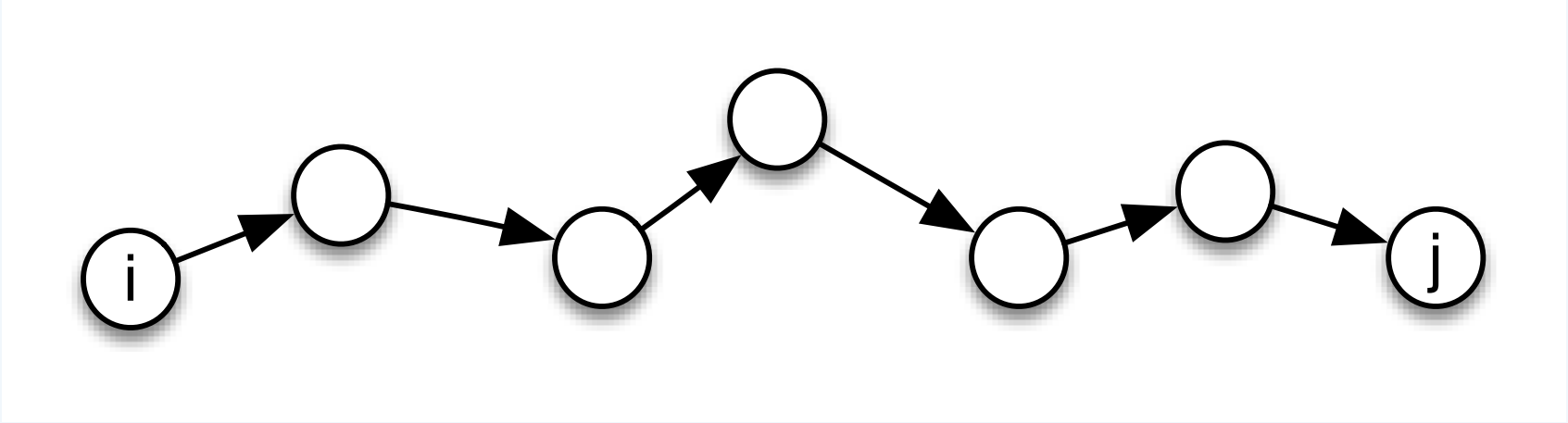


Shortest (**1, 3**)-path may use:
(i) no intermediate nodes (P); or
(ii) node **2** (P_1)



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

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



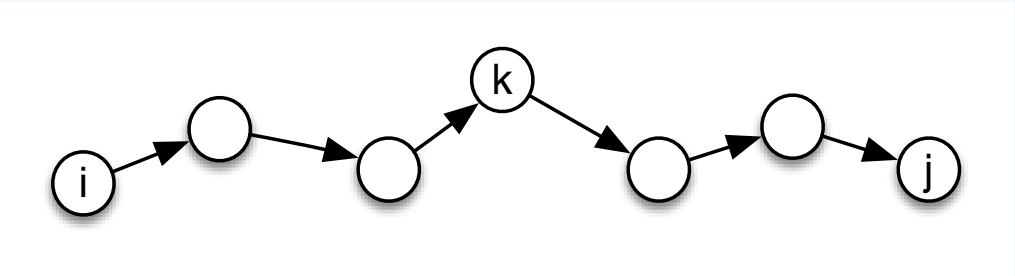
Fact: any subpath of P must be shortest itself.

A useful observation

Focus on the last node k from this set. 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

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

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

$D_0 =$

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 cost of an i-j path that is no larger than the cost 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: $\Theta(n^2)$

Today

1. Recap
   - Single-source shortest paths in graphs with real edge weights: a DP solution (Bellman-Ford)
2. An alternative formulation of the Bellman-Ford algorithm
3. All-pairs shortest paths (real weights): Floyd-Warshall
4. Minimum Spanning Trees (MSTs)
   - Prim's algorithm

The problem

Motivation: build the cheapest communication network over a set of locations.

Input: a weighted, undirected graph $G = (V, E, w)$

Output: a subset of edges $E_T \subseteq E$ such that

1. the graph $T = (V, E_T)$ is connected;
2. $\sum_{e \in E_T} w(e)$ is minimal.

Minimum weight Spanning Trees (MST)

Remark 2.

The graph $T = (V, E_T)$ is a tree: if there is a cycle, remove any edge from the cycle and obtain a connected graph with less cost.

Definition 5 (Spanning tree of a graph $G = (V, E)$).

A tree that spans all the nodes in V.

Output (restated): a minimum weight spanning tree of G.

Remarks

- Brute-force won't work: even simple graphs have many spanning trees—*how many in a simple cycle?*
- #spanning trees in the complete graph on n vertices: n^{n-2}

The cut property

Definition 6 (Cut).

A cut $(S, V - S)$ is a bipartition of the vertices.

Claim 1 (Cut property).

Assume all edge weights are distinct. Let $S \subset V$ $(S \neq \emptyset)$. Let e be the minimum-weight edge with one endpoint in S and the other in $V - S$. Then every MST contains e.

Remark 3.

The assumption of distinct edge weights is just for the purposes of the analysis; we will show how to remove it later.

Proof of the cut property

Notation: $w(T) = \sum_{e \in E_T} w(e)$

We will derive a contradiction by using an exchange argument.

- Let T' be a minimum-weight spanning tree that does not contain $e = (u, v)$.
- Then there must be some other path P in T' from u to v.
- Starting at u, follow the vertices of P: since (u, v) crosses from S to $V - S$, there must be some first vertex $v' \in V - S$ on P. Let u' be the last vertex before it in S.
- Then $e' = (u', v') \in E_{T'}$ **and** e' crosses between S, $V - S$.

Proof of the cut property (cont'd)

Exchange e with e' to obtain the set of edges

$$E_T = E_{T'} + \{e\} - \{e'\}.$$

T is a spanning tree:

- T is connected: any path in T' that used $e' = (u', v')$ is rerouted to follow P from u' to u, (u, v) and P from v to v'.
- T is acyclic (*why?*).

Since both e' and e cross between S and $V - S$ but e is the lightest edge with this property, $w(e) < w(e')$. Thus

$$w(T) < w(T').$$

Using the cut property to design MST algorithms

The cut property says: construct MST greedily by taking the lightest edge across two regions not yet connected.

`Generic-MST`$(G = (V, E, w))$

$E_T = \emptyset$ // the set of edges that will form our MST

while $|E_T| \leq n - 1$ **do**

Pick $S \subseteq V$ s.t. no edge in E_T crosses between S, $V - S$

Let $e \in E$ be a lightest edge that crosses between S, $V - S$

$E_T = E_T \cup \{e\}$

end while

Prim's algorithm

In Prim's algorithm, the edges in E_T always form a subtree and S is chosen to be the set of this subtree's vertices.

In other words:

1. Start with a root node s.
2. **Greedily** grow a tree outward from s by adding the node that can be attached as cheaply as possible at every step.

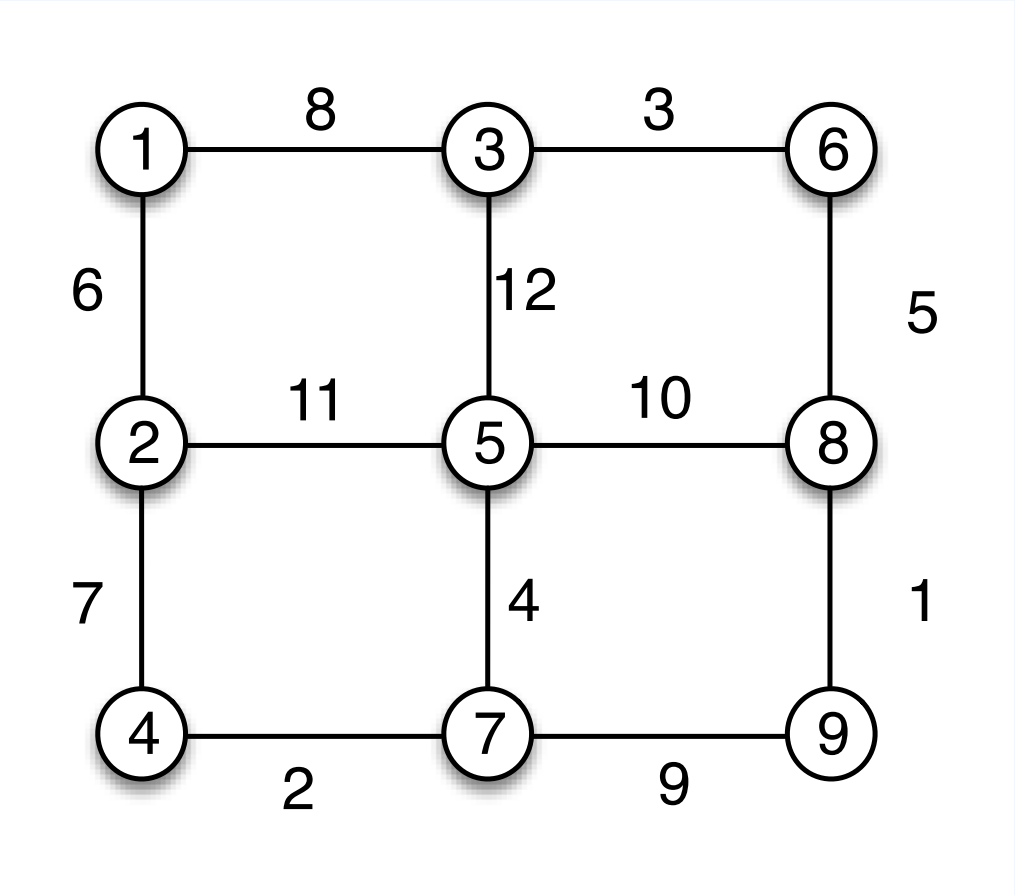
Detailed description of Prim's algorithm

1. $E_t = \emptyset$
2. Maintain a set $S \subseteq V$ on which a spanning tree has been constructed so far. Initially, $S = \{s\}$.
3. In each iteration, update
   - 3.1 $S = S \cup \{v\}$, where v is the vertex in $V - S$ that minimizes the attachment cost:

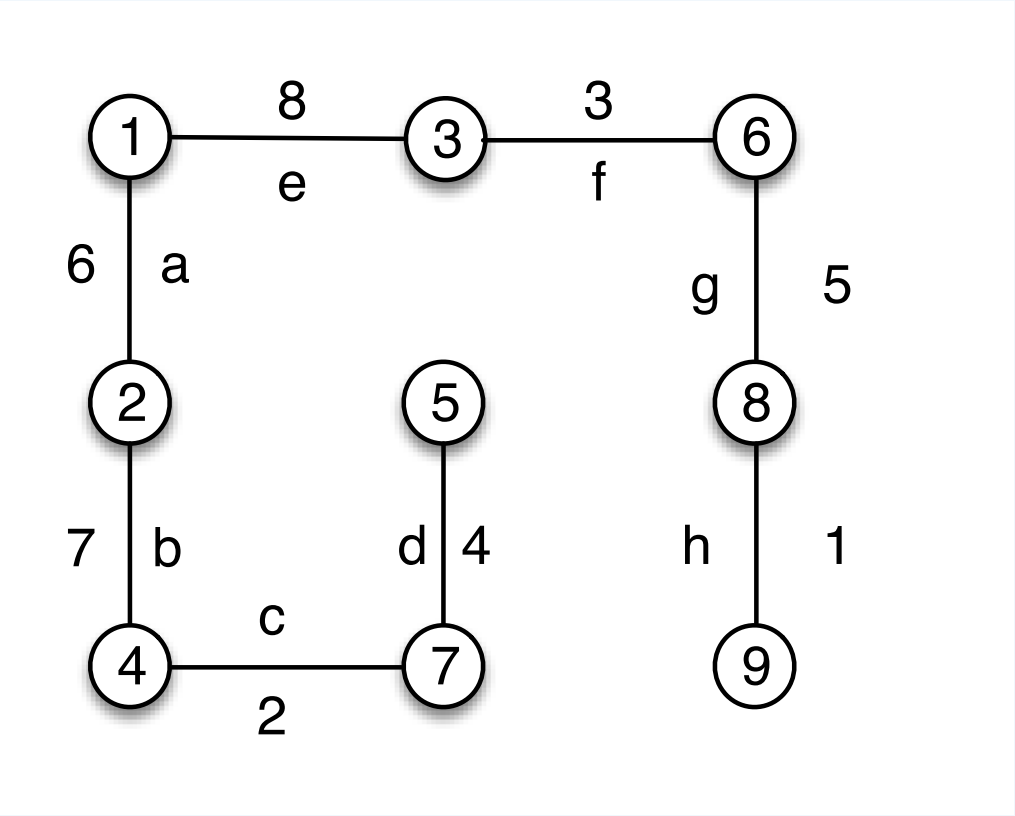
$$\min_{e=(u,v):u \in S} w(e).$$

   - 3.2 $E_T = E_T \cup \{e\}$

Example graph



Prim's MST for example graph (letters indicate the order in which edges were added)



Prim's: correctness

Follows directly from the Cut property: at every iteration an edge (u, v) is added such that

- $u \in S$, $v \in V - S$ (recall that Prim's algorithm sets S to be the set of vertices on which a partial MST has been constructed);
- (u, v) is the lightest edge that crosses between S and $V - S$.

Implementing Prim's algorithm

Similarly to Dijkstra's algorithm,

- store every node $v \in V - S$ in a priority queue Q, e.g., implemented as a binary min-heap (key= weight of the lightest edge between some node in S and v). Initially, $S = \{s\}$.
- maintain two arrays
  - $dist[v]$: stores the weight of the lightest edge between v and any vertex in S (in Dijkstra, it stored a conservative overestimate of the distance of v from the source s)
  - $prev[v]$: stores the node responsible for adding v to S

Pseudocode: how does this compute $T = (V, E_T)$?

```
Prim(G = (V, E, w), s)
    for u ∈ V do
        dist[v] = ∞; prev[v] = NIL
    end for
    dist[s] = 0
    Q = {V; dist}
    S = ∅
    while Q ≠ ∅ do
        u = ExtractMin(Q)
        S = S ∪ {u}
        for (u, v) ∈ E and v ∈ V − S do
            if dist[v] > w(u, v) then
                dist[v] = w(u, v)
                prev[v] = u
                DecreaseKey(Q, v)
            end if
        end for
    end while
```

Further implementations of Prim's algorithm

Notation: $|V| = n$, $|E| = m$

Implementation	`ExtractMin`	`Insert/` `DecreaseKey`	Time
Array	$O(n)$	$O(1)$	$O(n^2)$
Binary heap	$O(\log n)$	$O(\log n)$	$O((n+m)\log n)$
d-ary heap	$O(\log n)$	$O(\log n)$	$O((nd+m)\frac{\log n}{\log d})$
Fibonacci heap	$O(\log n)$	$O(1)$ amortized	$O(n \log n + m)$

- Optimal choice for $d \approx m/n$ (the *average* degree of the graph)
- d-ary heap works well for both sparse and dense graphs
  - *If $m = n^{1+x}$, what is the running time of Prim's algorithm using a d-ary heap?*
- **Amortized** analysis: *coming up in the next lecture*