Agenda:

- Minimum Spanning Trees (Cont'd): Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm

Reading:

- CLRS: 624-642 (covers Prim's)
- CLRS: 643-664 (covers Dijkstra's)

Recall: Minimum Spanning Tree (MST) problem

- Input: edge-weighted undirected graph
- Notions:
  - subgraph: $G' = (V, E')$, where $E' \subset E$, forest $=$ acyclic graph, trees
  - spanning subgraph: subgraph including all the vertices
  - spanning tree: spanning subgraph which is a tree — acyclic connected
    must have exactly $n - 1$ edges
    e.g., BFS/DFS-tree is a spanning tree of the graph
  - minimum spanning tree: sum of weights on tree edges is minimal
- **The MST Problem: Find a minimum spanning tree (MST) for the input graph.**
  - ... there could be more than one ..., e.g., all weights are the same, both BFS/DFS produce a MST ...
- Important for:
  - Min-cost set of edges that we need so that all vertices can reach one another
  - Learning value: a canonical example for greedy algorithms.
  - Useful info derived from the MST algorithms...
- The Minimum Spanning Forest problem: If the given graph is not necessarily connected: find MST for each CC.

Recall: Greedy algorithms and MST problem

- Greedy algorithms:
  - greedy — each step makes the best choice (locally minimum)
  - and don't look back
  - Optimal substructure: an optimal solution to the original problem contains within it optimal solutions to subproblems.
    T is MST for $G = (V, E) \Rightarrow$ for any $U \subset V$ where $T[U]$ is connected, $T[U]$ is a min-spanning tree.
- The general MST algorithm outline:
  1. A is a set of "safe" edges: they are contained in some MST T
  2. $A = \emptyset$ initially
  3. `while` $(|A| < n - 1)$ `do`: find a safe edge $e = (u, v)$ and set $A = A \cup \{e\}$.

     What happens when we halt?
- Two greedy solutions
  - Prim's Algorithm (Actually: Prim + Dijkstra + Boruvka)
    Grow T vertex-wise: A is always a MST on some $S \subset V$
  - Kruskal's (Actually: Kruskal + Boruvka)
    Grow T edge-wise: A is always a minimal set of edges without a cycle (forest)

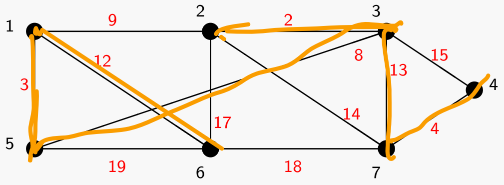
Prim's algorithm for the MST problem:

- Input: an edge-weighted (simple, undirected, connected) graph (positive weights)
- Output: an MST
- Idea:
  - Suppose we have already an MST A spanning subset S of vertices (Initially: S =a single vertex, $A = \emptyset$)
  - Grow A to span one more vertex $v \in \bar{S} = V \setminus S$ by adding a single edge (u, v) for some $u \in S$ and $v \notin S$.
  - Which edge to pick? Greedy! min-weight edge from all possible edges crossing the $(S, \bar{S})$ cut.
- First sketch:

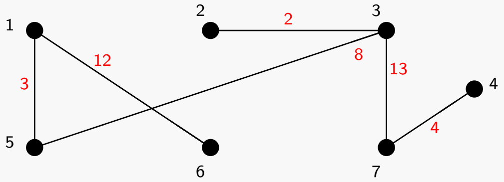
```
procedure primMST(G)        **G = (V, E)
S = {s}                     **for any start vertex s
A = ∅
while (|S| < |V|) do
    find a minimum weight edge e = (u, v):   u ∈ S and v ∉ S
    S = S ∪ {v}
    A = A ∪ {e}
return A
```

Prim's algorithm for the MST problem — an example:

- Input graph G:



- $\texttt{primMST}(G, w, 1)$ returns:



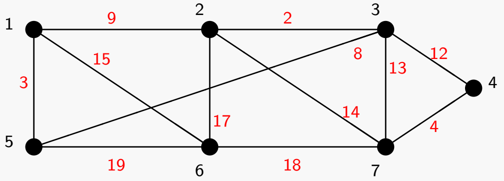
- First we prove correctness of Prim's algorithm
- Then we improve the naïve algorithm to reduce runtime
  Not surprisingly, finding the min-edge quickly is going to be useful (heaps!)

Prim's algorithm for the MST problem — correctness:

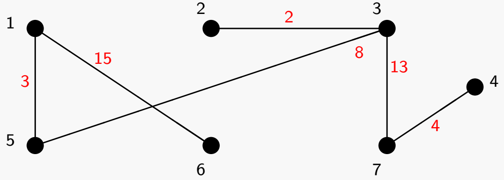
- In the proof we show the substitution property.
- At any point, suppose $A \subset T$ for some MST T, and we show that there exists a MST T' that contains $A \cup \{e\}$.
- If $e \in T$, we are done.
- Since $u \in S$ whereas $v \notin S$, so on the unique $u \to v$ path there has to be some edge $e' = (u', v')$ in T that crosses the $(S, \bar{S})$-cut. Clearly, $w(e) \leq w(e')$.
- We argue $T' = T \setminus \{e'\} \cup \{e\}$ is spanning V; and as it has $n-1$ edge it has to a spanning tree, with cost $\leq w(T)$.
- In fact, it is enough to argue u' and v' remain connected:
  Any $x \to y$ path on T either goes through e' or not. In the latter case, the $x \to y$ path remains in the T'; in the former case — use the new $u' \to v'$ path to connect x with y.
- So why do u' and v' remain connected? Well, in T there's a $u \to v$ path that e' was a part of. So u is connected to u' and v' is connected to v. So, $u' \to \underbrace{u, v}_{e} \to v'$ is a path connected u' and v' in T'. ■

Prim's algorithm for the MST problem — faster implementation:

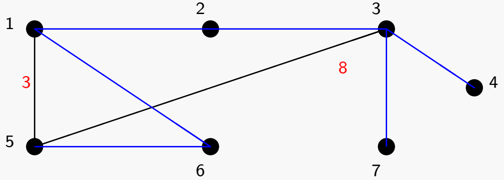
- Example: input graph G:

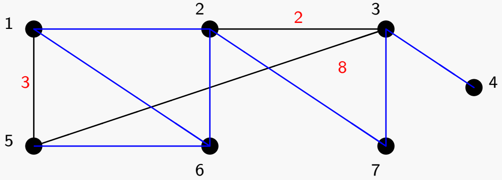


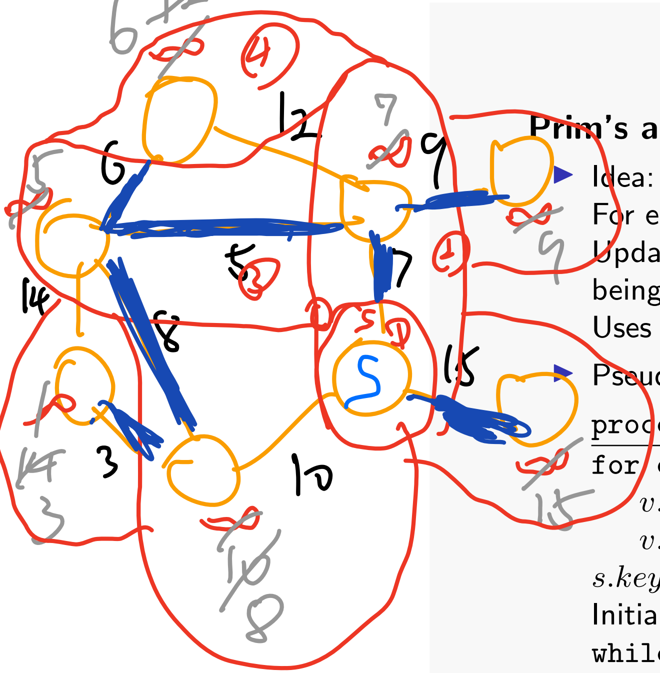
- primMST$(G, w, 1)$ returns:



- primMST$(G, w, 1)$: an intermediate tree
  Already picked black edges spanning 1,5,3; what are the candidate edges?







Prim's algorithm for the MST problem — faster implementation:

- Idea:
  For each node $\notin S$ — keep track of the min edge that connects it to S.
  Update the information only for the neighbors of the node that is currently being added to S.
  Uses a priority queue Q on $\bar{S}$ (so S is implicit — all the nodes **not** in Q)
- Pseudocode:

procedure primMST(G) **$G = (V, E)$

for each $v \in V(G)$ do
    $v.key = \infty$
    $v.predec =$ NIL
$s.key = 0$ **for some arbitrary start vertex s
Initialize a min-priority-queue Q on V using key
while $(Q \neq \emptyset)$ do
    $u =$ ExtractMin(Q) **r dequeued first
    foreach v neighbor of u do
        if $(v \in Q$ and $w(u,v) < v.key)$ then
            $v.predec = u$
            decrease-key$(Q, v, w(u,v))$ **$v.key$ is now $w(u,v)$

maintain p.Q Q on $V \setminus S$

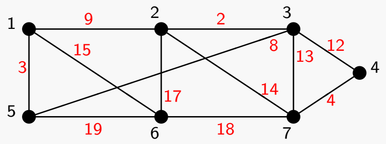
where $v.key = \min\{w(u,v) \mid u \in S\}$

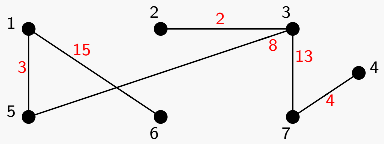
invariant

Prim's algorithm for the MST problem — faster implementation:

- Analysis of the improved algorithm:
  - correctness (almost done — need to prove that ExtractMin(Q) does extract the minimum weight edge that crosses the $(Q, \bar{Q})$-cut)
  - running time: $\Theta\Big(n + \sum_{u \in V} \Big((1 + deg(u)) \cdot \log n\Big)\Big)$
    –so: $\Theta((n+m)\log n) = \Theta(m\log(n))$ — adjacency list graph representation for a connected graph $m \geq n-1$.
    –or $\Theta(n^2 + m\log(n))$ in the adjacency matrix representation.
- Using more sophisticated data-structure (Fibonacci heap), runtime of Prim is reduced to $O(n\log(n) + m)$.

What does Prim Teach Us?

- Theorem: Let T be a spanning tree of G.
  Then T is a MST iff each edge is the min-edge crossing the cut it induces.
- I.e., take T, and some $e \in T$. Removing e from T disconnects T and creates two components $C_1, C_2 = V \setminus C_1$.
  Then the claim is that e is a minimal edge out of all the edges the cross the (C_1, C_2)-cut.
  Moreover, this is true for all edges $e \in T$.
- E.g.: 

MST: 

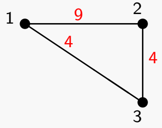
  $\min$-weight of edge crossing $(\{1,5,6\},\{2,3,4,7\})$-cut $= 8$
  $\min$-weight of edge crossing $(\{1,2,3,5,6\},\{4,7\})$-cut $= 13$
- Proof. $\Rightarrow$ If T is a MST, pick any e. For contradiction, assume e isn't a minimum edge crossing the cut it induces – replace e with an edge of strictly smaller weight. The resulting graph is a spanning tree (has $n-1$ edges and spans V) with cost strictly smaller than T. Contradiction.
- $\Leftarrow$ If all edges in T satisfy this property — how do we prove it is a MST?
- Use Prim!

What does Prim Teach Us?

- Theorem: Let T be a spanning tree of G.
  Then T is a MST iff each edge is the min-edge crossing the cut it induces.
- $=$ If all edges in T satisfy this property — how do we prove it is a MST?
- Use Prim!
  Claim: there's an instantiation of Prim that builds T, starting from s.
  In other words: we can run Prim and maintain the invariant that $T[S]$ is a single connected component ($T[S]$ is a spanning tree of S).
- Proof by induction on S. Clearly true for $|S| = 1$.
- The induction step:
  - Suppose in the transition from S to $S \cup \{v\}$ Prim picks an edge $e = (u, v)$ of weight w. Let $e_1 = (a_1, b_1), e_2 = (a_2, b_2), ..., e_k = (a_k, b_k)$ be all the edges in T with one vertex (a_i) in S and one vertex (b_i) in $\bar{S}$.
  - Since Prim picks the min-edge connecting S with $\bar{S}$, we have $w(e) \leq w(e_1), w(e) \leq w(e_2), ..., w(e) \leq w(e_k)$.
  - ASOC $w(e) < \min\{w(e_1), w(e_2), ..., w(e_k)\}$ — the weight of the edge chose by Prim is strictly smaller than all of the edges in T that leave S.
  - Look at the path $v \to u$ on T. It starts at $\bar{S}$ and edges at S, so it must use some edge e_j. So the removal of e_j separates u from v.
  - Hence e_j isn't the min-edge that separates the cut (e also crosses the same cut). Contradiction!
  - Thus $w(e) = \min\{w(e_1), ...w(e_k)\} = w(e_j)$.
  - Instantiate the priority-queue of Prim to pick b_j rather than v (both have the same key, so break ties in favor of b_j rather than v).

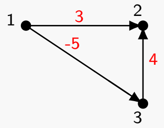
Shortest path problems:

- BFS recall: outputs every s-to-v shortest path
  - s — start vertex
  - v — reachable vertex from s (residing in a same connected component)
  - shortest — # edges
  - running time $\Theta(n+m)$
- But what if the edges of the graph have weights?
  In this case, shortest-path in terms of #edges isn't shortest weighted path.

  E.g. shortest weighted-path distance between 1 and 2 is 8.
- The weight of a path $=$ sum of weights on edges on path.
  Note: if there is no path between two nodes, the distance is set to ∞ ...
- The Shortest Path problem: find the shortest path in an edge-weighted graph connecting s and t.
- Turns out, shortest-path has a few properties that allow us to infer in the process **all** shortest-paths from a node s to any other node in the graph.
  So we study the Single-Source Shortest Path (SSSP) problem:
  Given an edge-weighted graph G and a source s, find out for each vertex $v \in V(G)$ a shortest paths from s to v.
- Variants:
  - edge weights: non-negative *vs.* arbitrary weights

Shortest Paths

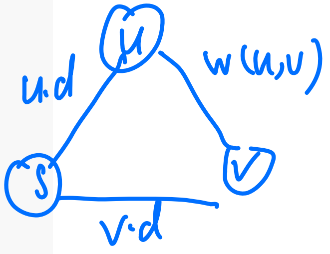
- A shortest path from u to v: out of all $u \to v$ paths, it is a path of minimal weight. (Can be more than one.)
- But a shortest path must satisfy subpath optimality:
  if $(u_0, u_1, ..., \underbrace{u_i, ..., u_j}_{\text{optimal}}, ..., u_k)$ is a shortest $u_0 \to u_k$ path, then
  $(u_i, u_{i+1}, ..., u_j)$ is a shortest $u_i \to u_j$ path.
- we denote $d(u, v)$ as the shortest path length from u to v
- Shortest paths remain well-defined if there are negative weight directed
  

  edges... (why not undirected?)
- ...as long as there isn't a negative weight path from a node to itself.
- So we assume **no negative cycles**.
- Shortest path distances — d is a metric:
  - For any u, we have $d(u, u) = 0$
  - For any u, v, we have $d(u, v) = d(v, u)$ (if G is undirected)
  - For any u, v, w, we have $d(u, w) \leq d(u, v) + d(v, w)$.

Common Outline of Single Source Shortest Path Algorithms

- Our shortest paths algorithm starts at a source s.
- It maintains a $dist$ attribute for each vertex that will serve as the estimation of the shortest-path distance.
- During the execution of the algorithm, we **always** have $u.dist \geq d(s, u)$.
- We start with `init()`: set $s.dist \leftarrow 0$ and $u.dist \leftarrow \infty$ for any $u \neq s$.
- We only update the $dist$ attribute by `Relaxing`

update ∞



`procedure relax`(u, v) `**`u, v `two adjacent nodes`

`if` $(v.dist > u.dist + w(u, v))$ `then`

    $v.dist \leftarrow u.dist + w(u, v)$ → update shortest path

- Claim: any algorithm that starts with `init()` and only updates $dist$ using `relax()` must always satisfy $v.dist \geq d(s, v)$ for any v.
- Proof: by induction on the number of times `relax()` is invoked. Base case: invoked 0 times — claim holds through `init()`. Induction step: If `relax`(u, v) doesn't change $v.dist$ we are done. Otherwise, we now have

by induction → $v.d \geq d(s,v)$

$$v.dist = u.dist + w(u, v) \geq d(s, u) + w(u, v) \geq d(s, v)$$

by triangle inequality

$d(s,v) \leq d(s,u) + w(u,v)$

$\leq v.d + w(u,v)$

- Corollary: Since `relax()` cannot increases $v.dist$, so if and when we set $v.dist = d(s, v)$ we keep $v.dist$ unchanged.

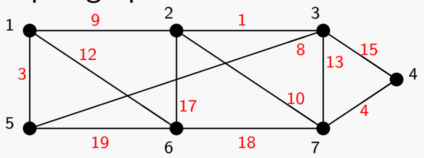
Dijkstra's SSSP algorithm:

- For graphs with non-negative weights (both directed and undirected)
- Idea in Dijkstra's algorithm:
  - Maintains a set S of vertices for which we know the shortest path. Initially, $S = \{s\}$, at the end: $S = V$.
  - Which vertex from $\bar{S} = V \setminus S$ should we pick?
  - Dijsktra: the greedy solution — the node in $\bar{S}$ with minimum $dist$.
- procedure dijkstra(G, w, s) **$G = (V, E)$, w =weights

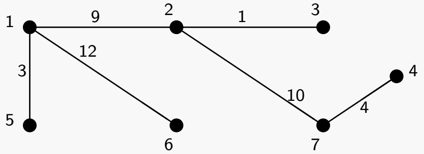
```
foreach v do                                   **initialization
    v.dist ← ∞
    v.predec ← NIL
s.dist ← 0
Build Min-Priority-Queue Q on all nodes, key = dist
            ** nodes in Q are nodes we are not yet sure about
            ** namely Q holds nodes in S̄
while (Q ≠ ∅) do
    u ← ExtractMin(Q)                          **s dequeued first
    foreach v neighbor of u do
        if (v.dist > u.dist + w(u,v)) then
            v.dist ← u.dist + w(u,v)           **a Relax() call
            v.predec ← u
            decrease-key(Q, v, v.dist)
```

Dijkstra's SSSP algorithm — an example:

- Input graph G:

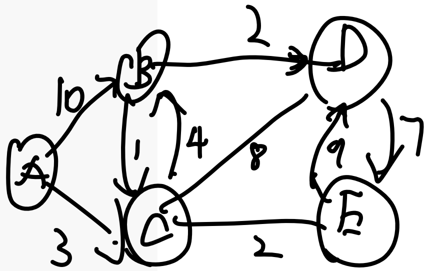


- `dijkstra`$(G, 1)$:



- `dijkstra`$(G, 1)$ trace:

v	1	2	3	4	5	6	7
$v.dist/v.predec$	0/NIL	∞/NIL	∞/NIL	∞/NIL	∞/NIL	∞/NIL	∞/NIL
1 dequeued	0/NIL	9/1	∞/NIL	∞/NIL	3/1	12/1	∞/NIL
5 dequeued	0/NIL	9/1	11/5	∞/NIL	3/1	12/1	∞/NIL
2 dequeued	0/NIL	9/1	10/2	∞/NIL	3/1	12/1	19/2
3 dequeued	0/NIL	9/1	10/2	25/3	3/1	12/1	19/2
6 dequeued	0/NIL	9/1	10/2	25/3	3/1	12/1	19/2
7 dequeued	0/NIL	9/1	10/2	23/7	3/1	12/1	19/2



S = { }   {A,B,C,D,E} = Q

{A}   0 ∞ ∞ ∞ ∞

{A,C}   0 10 3 ∞ ∞

{A,C,E}   ~~0~~ 7 ~~3~~ 11 5

Dijkstra's SSSP algorithm — Correctness:

- LI: "at the start of each while-loop iteration $u.dist = d(s,u)$ for all $u \notin Q$."
- Initialization: The statement is vacuously true when Q holds all nodes. The statement is clearly true for the first vertex taken out of Q: the source s.
- Termination: At the end of the while-loop, Q is empty, so we have found all shortest-paths distances from s.
- Maintenance:
  - Suppose LI holds at the beginning of the iteration. Denote u as the node `Extract-Min(`Q`)` takes out.
  - ASOC $d(s,u) < u.dist$.
  - Look at a shortest-path $s \to u$. $s \notin Q$ but $u \in Q$. Let (x,y) be the first edge such that $x \notin Q$ but $y \in Q$.
    (y might be u, x might be s.)
  - First, $d(s,y) \le d(s,u)$ (subpath optimality + non negative weights)
  - Second, when we took x outof Q we made sure $y.dist \le x.dist + w(x,y)$
  - Since $x \notin Q$ in the beginning of the iteration, then $x.dist = d(s,x)$.
  - Thus $y.dist \le d(s,x) + w(x,y) = d(s,y)$ (Again, subpath optimality)
  - Altogether: $y.dist = d(s,y) \le d(s,u) < u.dist$
  - If $y = u$ — immediate contradiction.
    If $y \neq u$ — then `Extract-Min(`Q`)` should return y not u.
    Contradiction in any case.

Dijkstra's SSSP algorithm — analysis:

- $|V| = n$ and $|E| = m$
- Running time:
  - `init()`— takes $O(n)$ time.
  - Initializing the priority-queue — $O(n)$
  - For each node, `ExtractMin` takes $O(\log(n))$ time.
  - ... and we search for all neighbors ($O(deg(v))$ in the adjacency list model, $O(n)$ in the adjacency matrix model)
  - For every edge, we examine the edge atmost twice (each time we take its endpoints from Q) and invoke `relax()` at most once, and decrease the key at most once.
    So $O(\log(n))$ work per edge.
- In the adjacency-list model

$$O(n)+O(n)+O(n\log(n))+O(\sum_v deg(v))+O(m\log(n)) = O((n+m)\log(n))$$

- In the adjacency-matrix model

$$O(n) + O(n) + O(n(\log(n) + n)) + O(m\log(n)) = O(n^2 + m\log(n))$$

- There exists a more refined implementation of the PQ (with Fibonacci heaps) that gives runtime $O(n\log(n) + m)$

Agenda:

- Bellman-Ford
- Floyd-Warshall

Reading:

- Textbook pages 643-663, 684-699

Bellman-Ford's SSSP algorithm for the general case:

- General case — edge weights **could be negative**
- Output:
  - Case 1. if there is a negative weight cycle, report it
  - Case 2. otherwise report all the $dist[u]$ values and the associated paths
- An key idea in the algorithm:
  If there is no negative weight cycle reachable from s,

  $\rightarrow$ every s-to-u shortest path contains at most $n-1$ edges;
  $\rightarrow$ $\exists u$ such that s-to-u shortest path contains 1 edge;
  $\rightarrow$ ...
  $\rightarrow$ at termination: for every directed edge (u, v) there must be $d[v] \leq d[u] + w(u, v)$.

  $\rightarrow\rightarrow$ $d[v]$ can be reduced $n-1$ times in order to reach value $dist[v]$, but no more.
- `procedure relax` (u, v)

  if $d[v] > d[u] + w(u, v)$ **then**
  $d[v] \leftarrow d[u] + w(u, v)$
  $p[v] \leftarrow u$

- <u>`procedure bellman-ford`(G, w, s)</u> **$G = (V, E)$

```
for each v ∈ V(G) do             **initialization
   d[v] ← ∞
   p[v] ← NIL
d[s] ← 0
for i ← 1 to n − 1               **n = |V(G)|
   for each edge (u, v) ∈ E(G) do
      relax (u, v)               **update d[v]
for each edge (u, v) ∈ E(G) do
   if d[u] + w(u, v) < d[v] then **there is a negative cycle
      return FALSE
return TRUE
```

- Define $\delta(s, v)$: length of the shrotest path from s to v
- Lemma 1: For all $v \in V$, at every iteration and over any sequence of relaxactions we have $d[v] \geq \delta(s, v)$ and once $d[v]$ achieves value $\delta(s, v)$ it never changes.

- Proof: By induction on the number of relaxation steps. Base case $i = 0$ is trivial.
  For I.S. consider the relaxation of an edge (u, v) and assume $d[v]$ changes:

$$\begin{aligned} d[v] &= d[u] + w(u, v) \\ &\geq \delta(s, u) + w(u, v) \\ &\geq \delta(s, v) \end{aligned}$$

  Also note that $d[v]$ can only increase during a relax operation and since it is $\geq \delta(s, v)$ it doesn't change once it reaches this value.
- Lemma 2: Suppose that G has no negative cycle reachable from s. Then after $n - 1$ iterations $d[v] = \delta(s, v)$ for all $v \in V$.
  Proof: Consider a vertex $v \in V$ and assumr $s = v_0, v_1, v_2, \ldots, v_k = v$ is a shortest path from s to v.
  We prove by induction on i that after iteration $i \geq 0$: $d[v_i] = \delta(s, v_i)$.
  Base: $i = 0$ is trivial
  I.S.: Suppose that $d[v_{i-1}] = \delta(s, v_{i-1})$ and consider iteration i when we relax (v_{i-1}, v_i). Then it can be seen that
  $d[v_i] = d[v_{i-1}] + w(v_{i-1}, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i) = \delta(s, v_i)$.

- To complete the proof of correctness, we show that if there is a negative cycle then the algorithm detects it.
- suppose there is a negative cycle c: $v_0, v_1, v_2, \ldots, v_k = v_0$:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

- By way of contradiction suppose the algorithm returns true, i.e. for all $1 \leq i \leq k$: $d[v_i] \leq d[v_{i-1} + w(v_{i-1}, v_i)$. Thus:

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1} + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- but $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$ since each vertex of the cycle appears exactly once in each sum.
  Thus $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$ which contradicts the assumption.

Floyd-Warshall's algorithm for All-Pairs-Shortest path:

- General case (weights can be negative but no negative cycle);
- Output: shortest path between "every" pair u, v of vertices.
- Idea: Use dynamic programming.
  Define $d[i, j, k]$ to be the length of shortest path from i to j for which all intermediate vertices are in $\{1, \ldots, k\}$, for every $1 \leq i, j \leq n$ and $0 \leq k \leq n$.
- When $k = 0 \Longrightarrow$ no intermediate vertex, so: $d[i, j, 0] = w(i, j)$.
- For general $k \geq 1$:
  - If the path does not contain k, inter. vertices only from $\{1, \ldots, k-1\}$: $d[i, j, k-1]$.
  - If the path contains k, it has two parts: one goes from i to k with intermediate only from $\{1, \ldots, k-1\}$, followed by a path from k to j using only from $\{1, \ldots, k-1\}$: $d[i, k, k-1] + d[k, j, k-1]$.
- Recurrence:

$$d[i,j,k] = \min \begin{cases} w(i,j) & k = 0 \\ d[i,k,k-1] + d[k,j,k-1] & k \geq 1 \end{cases}$$

- We compute the table bottom-up, starting from smaller values of k to larger values.

Floyd-Warshall's algorithm:

- ~~Pseudocode~~:

```
procedure Floyd-Warshall(G)
for i ⟵ 1 to n do
    for j ⟵ 1 to n do
        d[i,j,0] = w(i,j)
        b[i,j] = 0          ** b[i,j] keeps the break point vertex
for k ⟵ 1 to n do
    for i ⟵ 1 to n do
        for j ⟵ 1 to n do
            if d[i,k,k−1] + d[k,j,k−1] < d[i,j,k−1] then
                d[i,j,k] = d[i,k,k−1] + d[k,j,k−1]
                b[i,j] = k
            else
                d[i,j,k] = d[i,j,k−1]
```

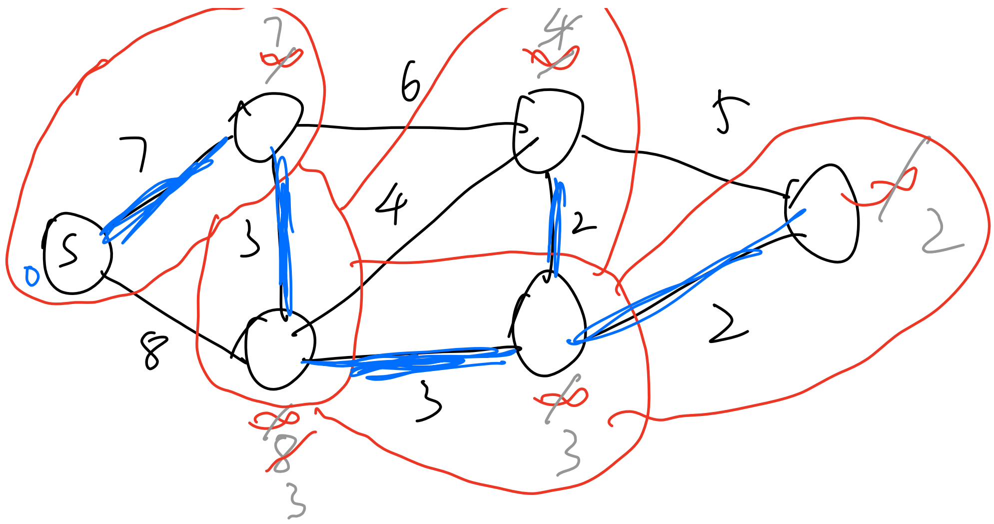
Floyd-Warshall's algorithm:

- To print the actual path between i and j:

  <u>procedure Print-FW(i,j)</u>
  if $b[i,j]=0$ then
      Print ''edge'', i, ''to'', j
  else
      Print-FW$(i,b[i,j])$
      Print-FW$(b[i,j],j)$

- Running time: $\Theta(n^3)$, better than n run of Dijkstra (one for every vertex as a source).

2



S
0
7
6
5
4
3
2
8
3
2
7
4
8
3
3
2