Shortest Paths

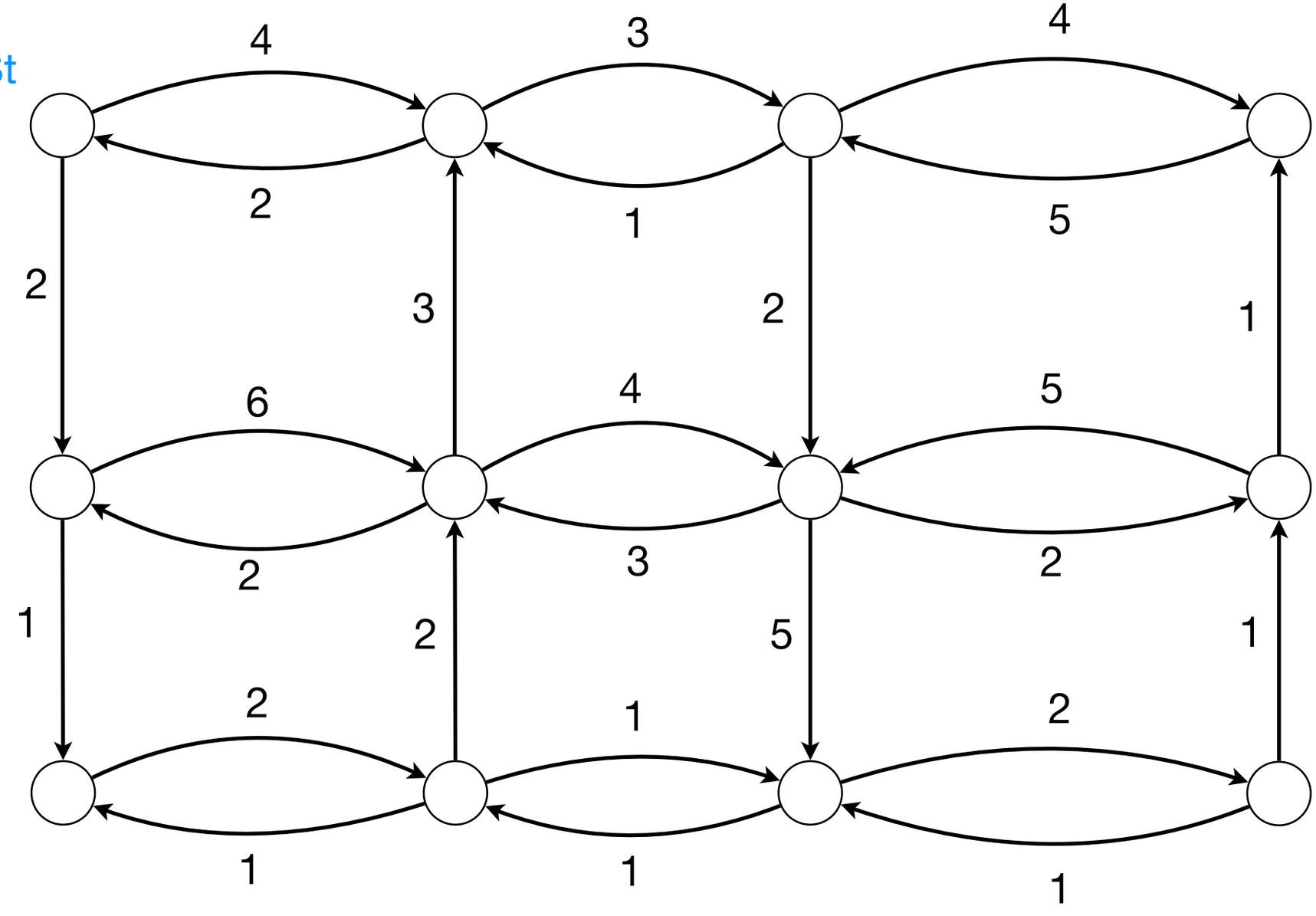
Nishant Mehta

Lectures 5–7

Finding the Fastest Way to Travel between Two Intersections in Vancouver

Granville St and W Hastings St

Homer St and W Hastings St



Granville St and W Pender St

Homer St and W Pender St

Shortest Paths in Weighted Graphs

- Find fastest way to travel across the country using directed graph representing roads, with edge weights representing:
  - distances
  - travel times between cities (might account for speed limits, traffic, etc.)
- Find a fastest way using flights
  - Flight distances between airports.
  - Might also allow for warps in space-time continuum. Negative travel time

Single-Source Shortest Paths

- **If graph is unweighted:**
  - Breadth-First Search is a solution (more on this soon)
- **If graph is weighted:**
  - Every edge is associated with a number:

    integers, rational numbers, real numbers (might be negative!)
  - An edge weight can represent:

    distance, connection cost, affinity

Single-Source Shortest Paths Problem

- **Input:** A weighted directed graph G = (V, E)
  and a source vertex s

- **Output:** All single-source shortest paths for s in G, i.e., for all other vertices v in G, a shortest path from s to v.

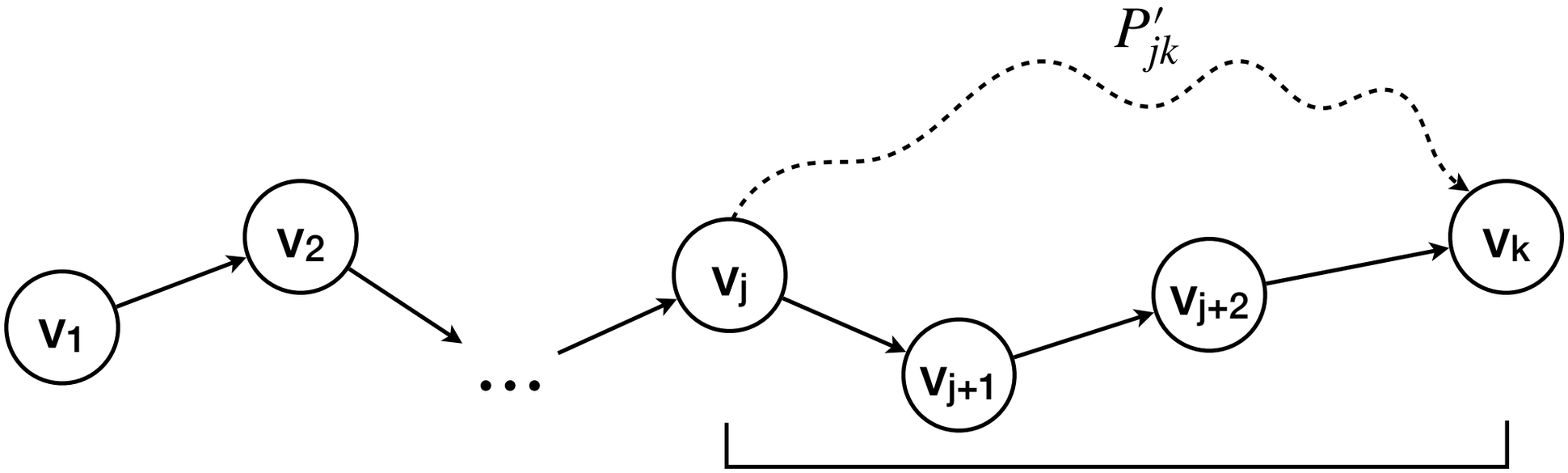
- A path $p = (v_0, v_1, \ldots, v_k)$ from $s = v_0$ to $v = v_k$ is shortest if its length $w(p) = \sum_{j=1}^{k} w(v_{j-1}, v_j)$ is the minimum possible among all s–v paths

Optimal substructure

Optimal substructure - an optimal solution to a problem contains with it optimal solutions to subproblems

Example:

- Problem: Find shortest path from vertex v_1 to vertex v_k
- Subproblem: Find shortest path from intermediate vertex v_j to v_k



$$P_{jk} = (v_j, v_{j+1}, v_{j+2}, v_k)$$

Subpaths of shortest paths are shortest paths

Lemma

Let $P_{1k} = (v_1, v_2, \ldots, v_k)$ be a shortest path from v_1 to v_k.
Take some arbitrary i, j satisfying $1 \leq i < j \leq k$, and let $P_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ be the subpath of P_{1k} from v_i to v_j.
Then P_{ij} is a shortest path from v_i to v_j.

Relax: The most important function for today's lecture

RELAX(u, v)

If d[u] + w(u, v) < d[v]

$d[v] \leftarrow d[u] + w(u, v)$

$\pi[v] \leftarrow u$

Single-source shortest paths for weighted DAGs

- Suppose we have a weighted directed acyclic graph (DAG)

- An easy way to solve single-source shortest paths problem:

  (1) Use topological sort to obtain topological ordering
  (basically, use DFS + a slight amount of extra work)

  (2) For each vertex u in topological order

  For all vertices v adjacent to u

  RELAX(u, v)

- Runtime?

Single-source shortest paths for weighted DAGs

- Suppose we have a weighted directed acyclic graph (DAG)

- An easy way to solve single-source shortest paths problem:

  (1) Use topological sort to obtain topological ordering
  (basically, use DFS + a slight amount of extra work)

  (2) For each vertex u in topological order

  For all vertices v adjacent to u

  RELAX(u, v)

- Runtime? $O(V + E)$

Single-source shortest paths for weighted DAGs

- Suppose we have a weighted directed acyclic graph (DAG)

- An easy way to solve single-source shortest paths problem:

  (1) Use topological sort to obtain topological ordering
  (basically, use DFS + a slight amount of extra work)

  (2) For each vertex u in topological order

  For all vertices v adjacent to u

  RELAX(u, v)

- Runtime? $O(V + E)$

- Claim: Above algorithm is correct. Let's prove it!

Breadth-First Search for Unweighted Graphs

For each vertex, keep track of a color:

- WHITE: Unvisited
- RED: Visited and Active - some adjacent vertices might not been added to queue yet
- BLACK: Visited and Inactive - all adjacent vertices have been added to queue

Pseudocode:

1. For all $u \in V$
2. Color u WHITE, set $d[u] = \infty$, and set $\pi[u] = \text{null}$
3. Color s RED and set $d[s] = 0$
4. Enqueue s into empty queue Q
5. While Q is not empty:
6. $u \leftarrow \text{Dequeue}(Q)$
7. For each WHITE vertex v adjacent to u
8. Color v RED
9. Set $d[v] = d[u] + 1$ and $\pi[v] = u$.
10. Enqueue v into Q
11. Color u BLACK

Dijkstra's Algorithm

Dijkstra's Algorithm

Input: A simple directed graph G with nonnegative edge-weights and a source vertex s in G

Output: A number d[u] for each vertex u in G such that d[u] is the weight of the shortest path in G from s to u

Dijkstra's Algorithm - Conceptual Version

Dijkstra(V, E, s):

S ← {s}

d[s] ← 0

While S ≠ V

For all v ∉ S such that there is an edge (u, v) for some u ∈ S:

cost c[v] ← $\min_{\{(u, v):\ u \text{ in } S\}}$ d[u] + w(u, v)

Of these vertices, let v be one for which c[v] is minimum

Add v to S

d[v] ← c[v]

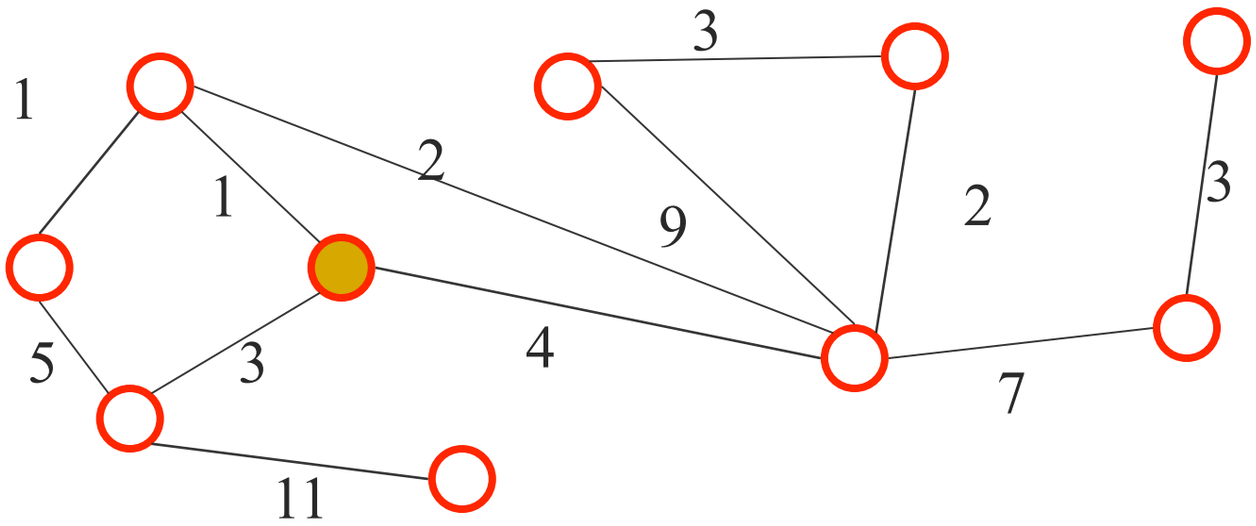
Note: this version doesn't use Relax! But for an implementation, it's good to do so. Also, this version doesn't keep track of the predecessor array!

Dijkstra's Algorithm

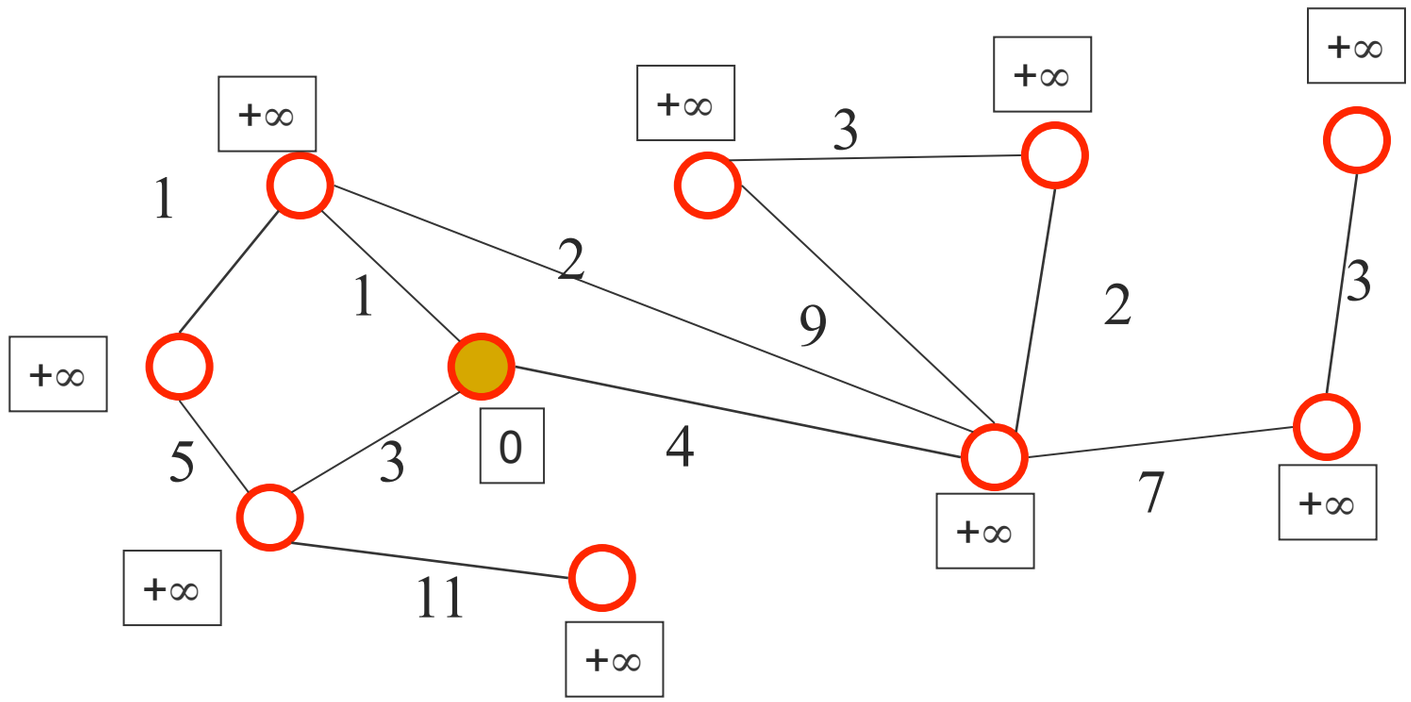
```
Dijkstra(V, E, s):
    For v in V
        d[v] ← ∞ ; π[v] ← null;
    d[s] ← 0
    S ← ∅
    Q = BuildPriorityQueue(V, d)
    While Q not empty
        u ← DeleteMin(Q)
        S ← S ∪ u
        For v in Adj[u]
            Relax(u,v)
```

```
RELAX(u, v)
    If d[u] + w(u, v) < d[v]
        d[v] ← d[u] + w(u, v)
        π[v] ← u
```

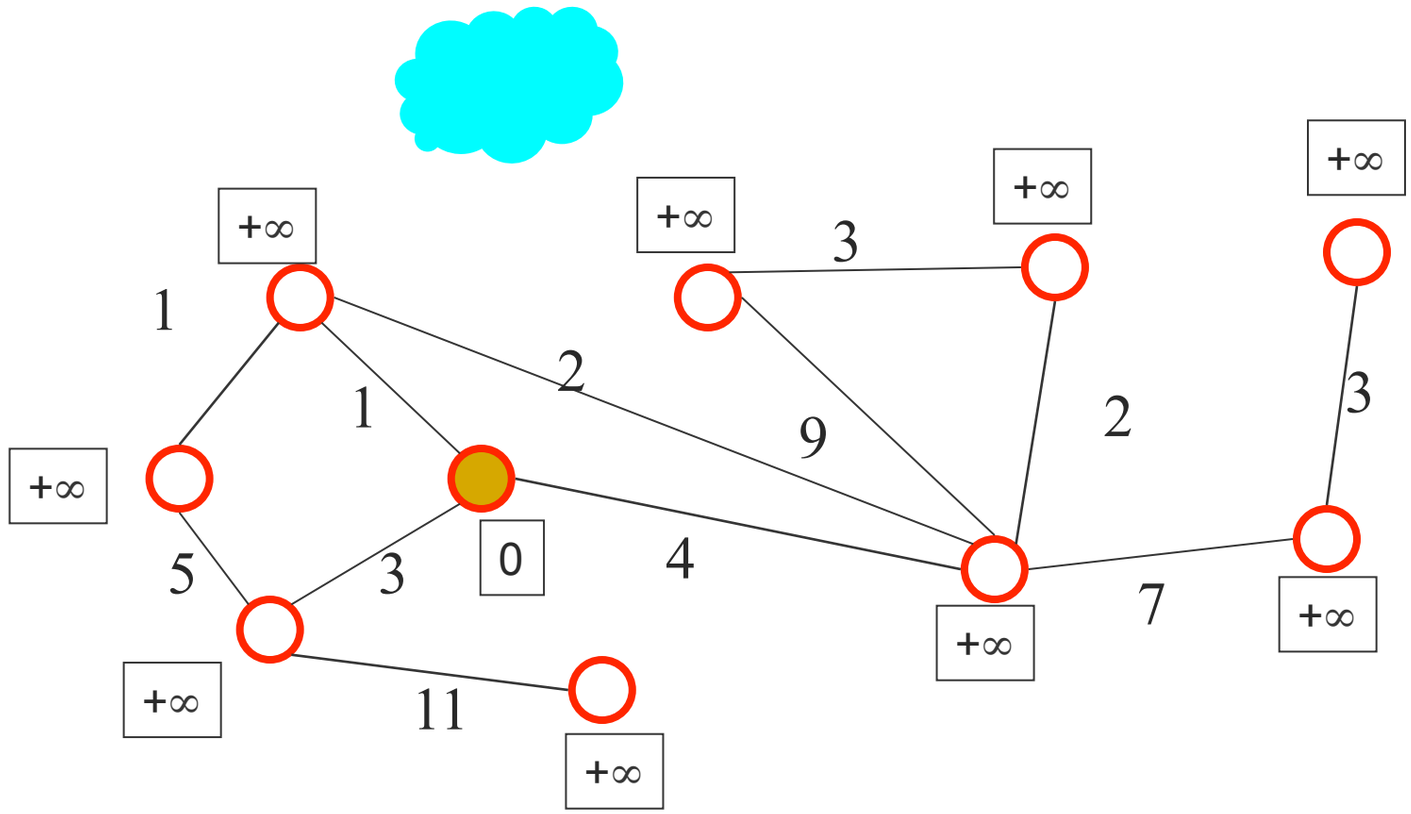
Dijkstra's algorithm: a greedy algorithm



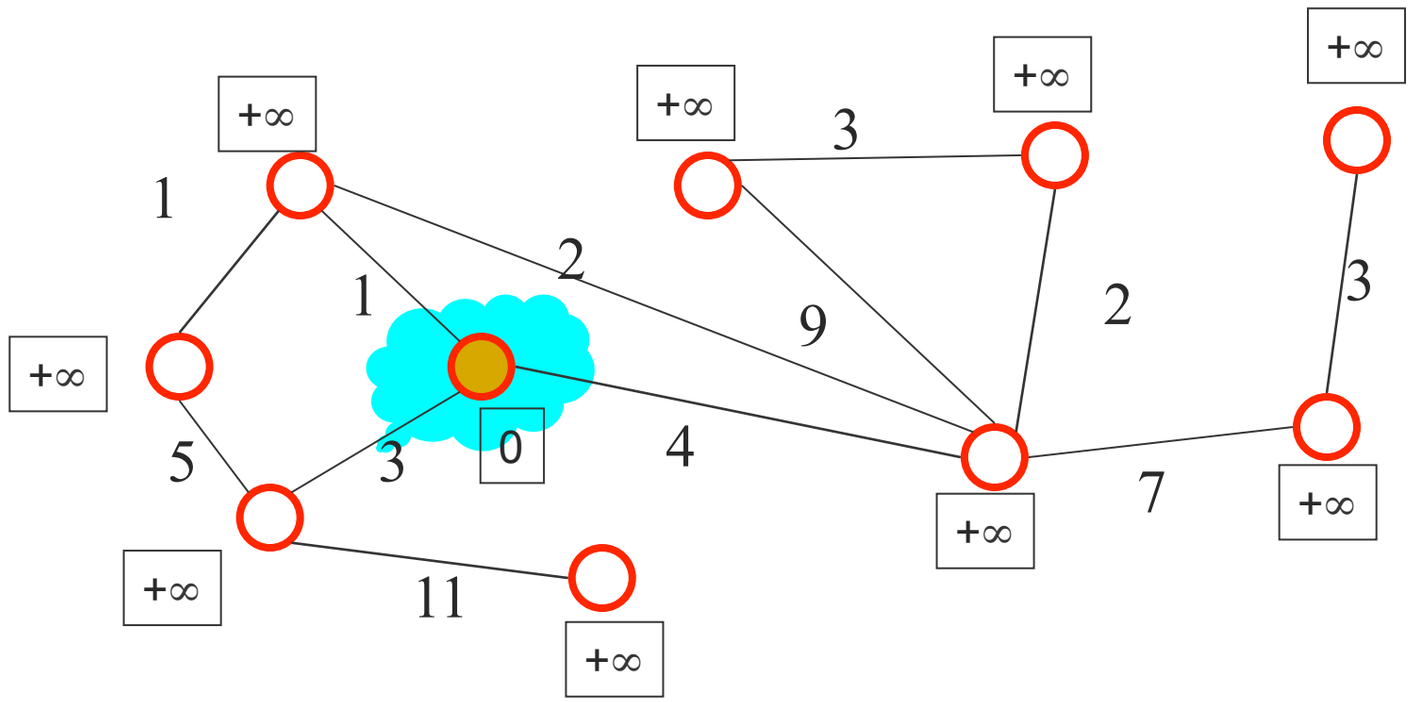
Dijkstra's algorithm: Initializing



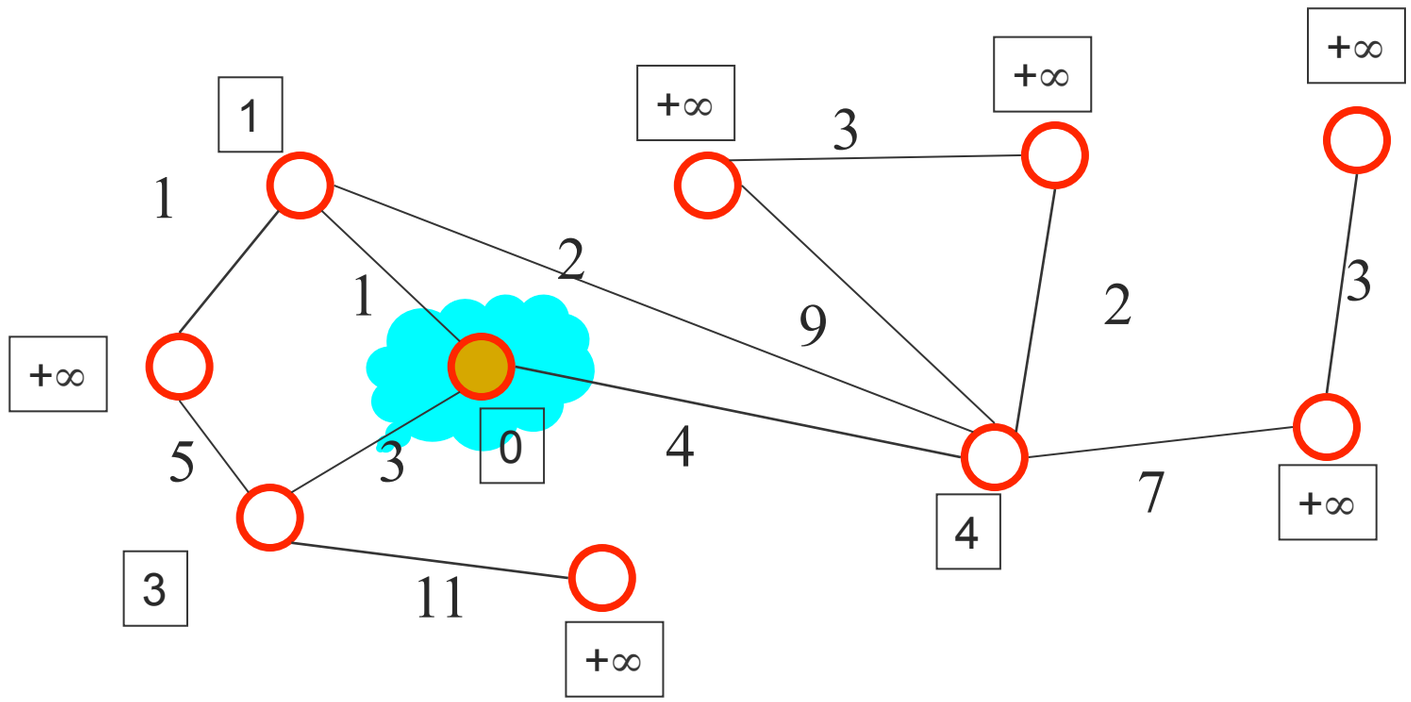
Dijkstra's algorithm: Initializing Cloud C (consisting of "solved" subgraph)



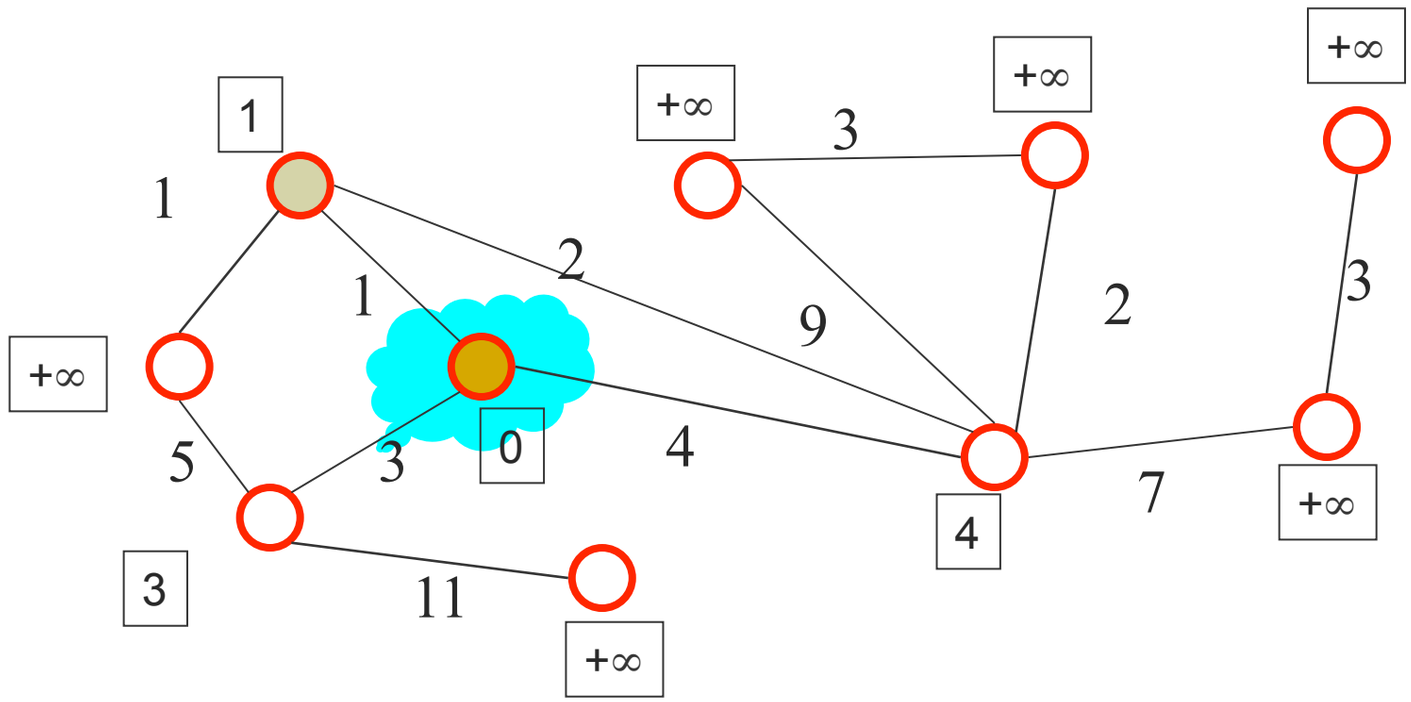
Dijkstra's algorithm: pull v into C



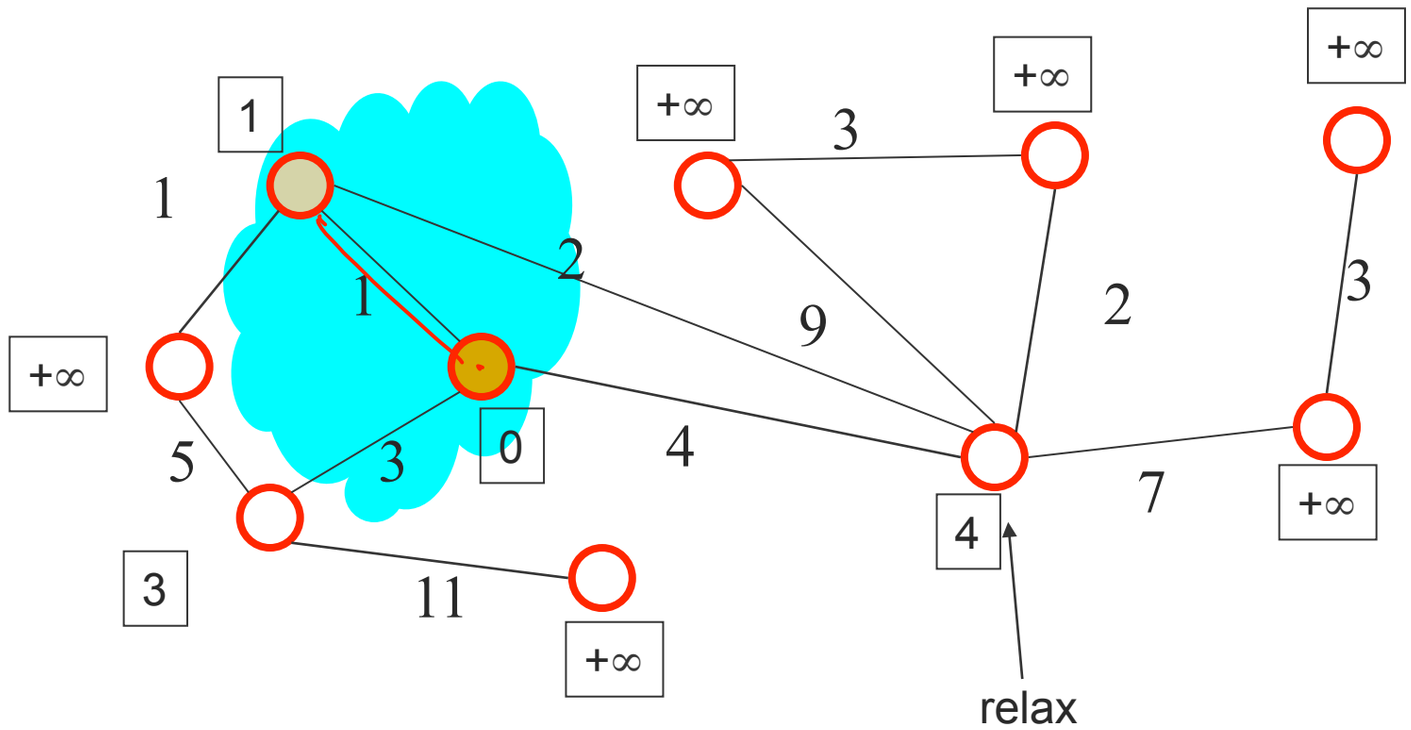
Dijkstra's algorithm: update $C's$ neighborhood



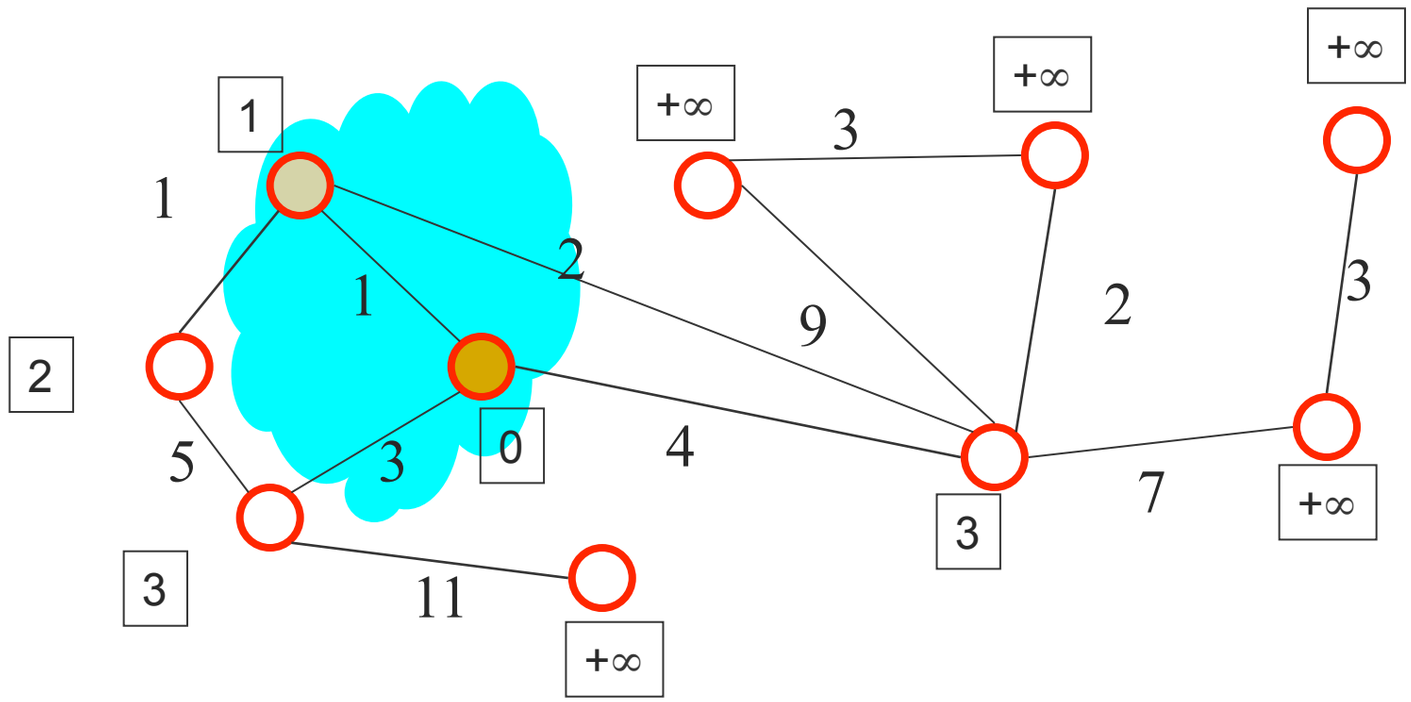
Dijkstra's algorithm: pick closest vertex u outside C



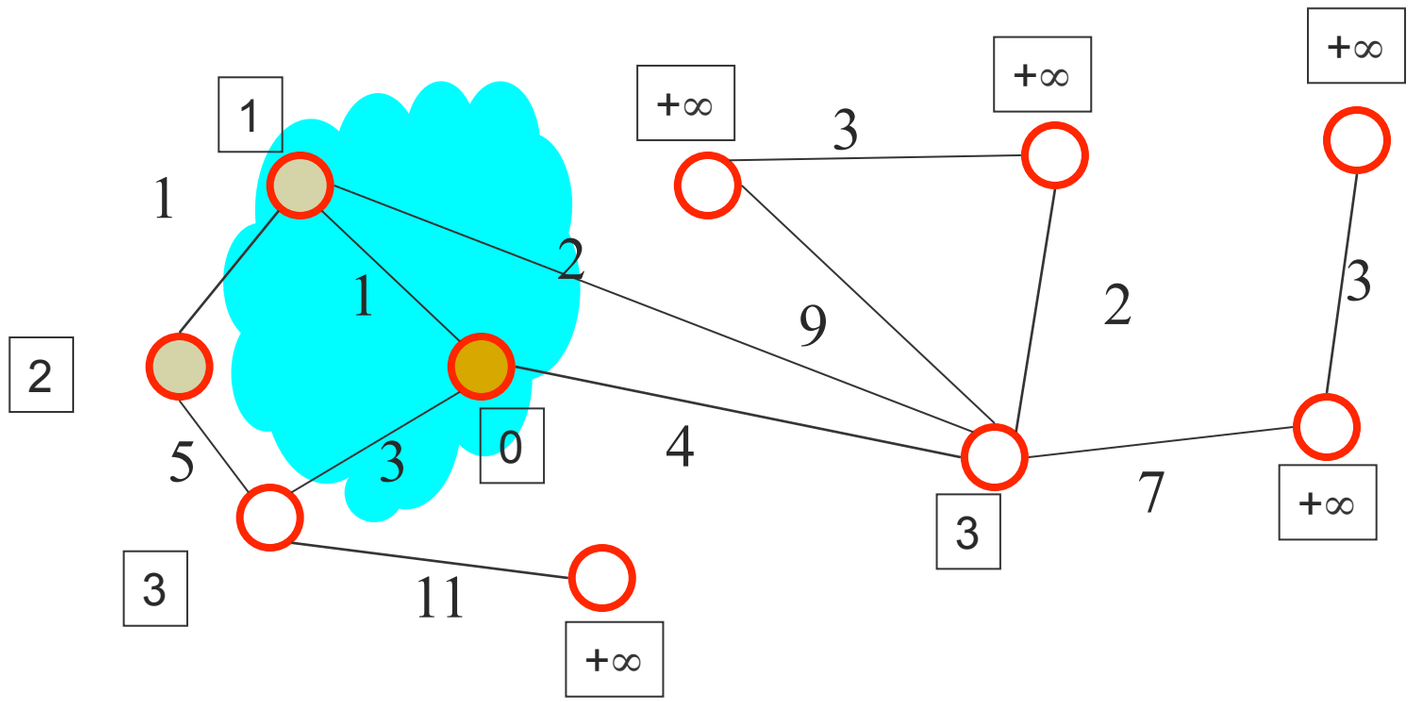
Dijkstra's algorithm: pull u into C



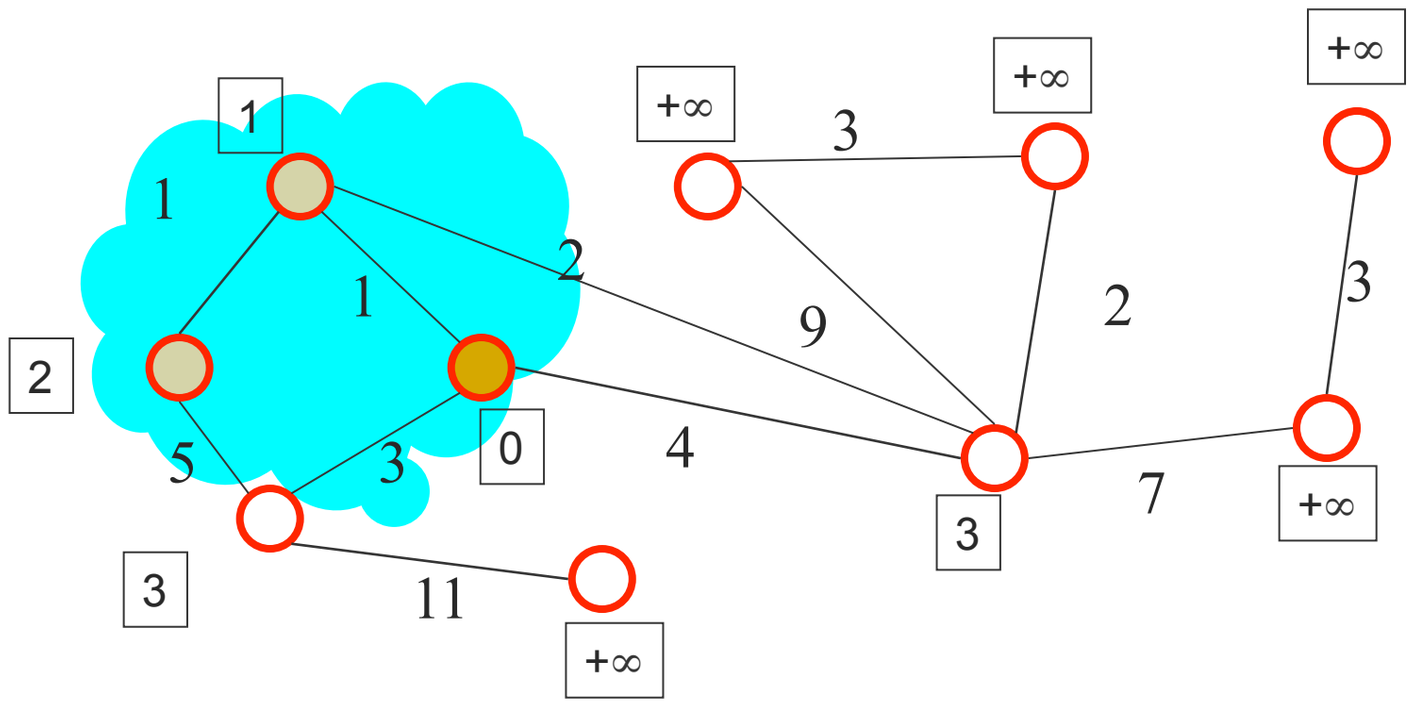
Dijkstra's algorithm: update $C's$ neighborhood



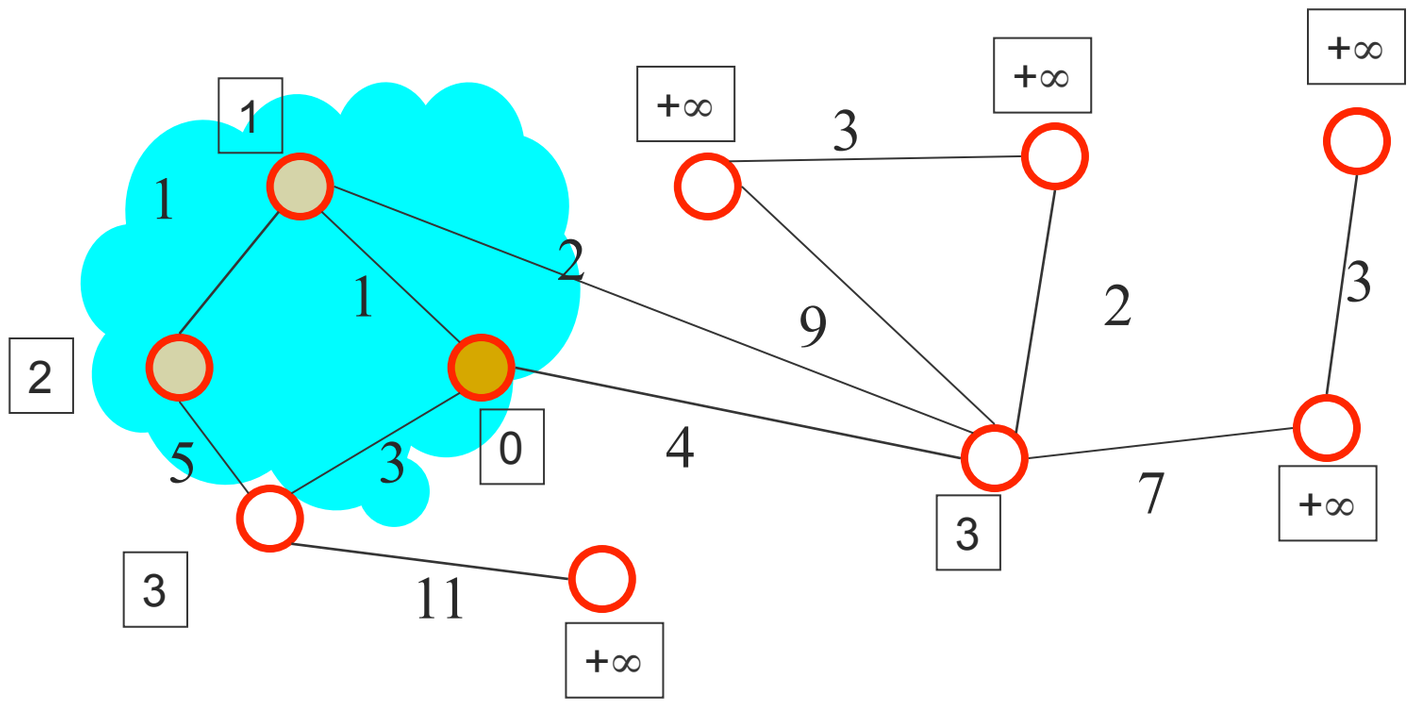
Dijkstra's algorithm: pick closest vertex u outside C



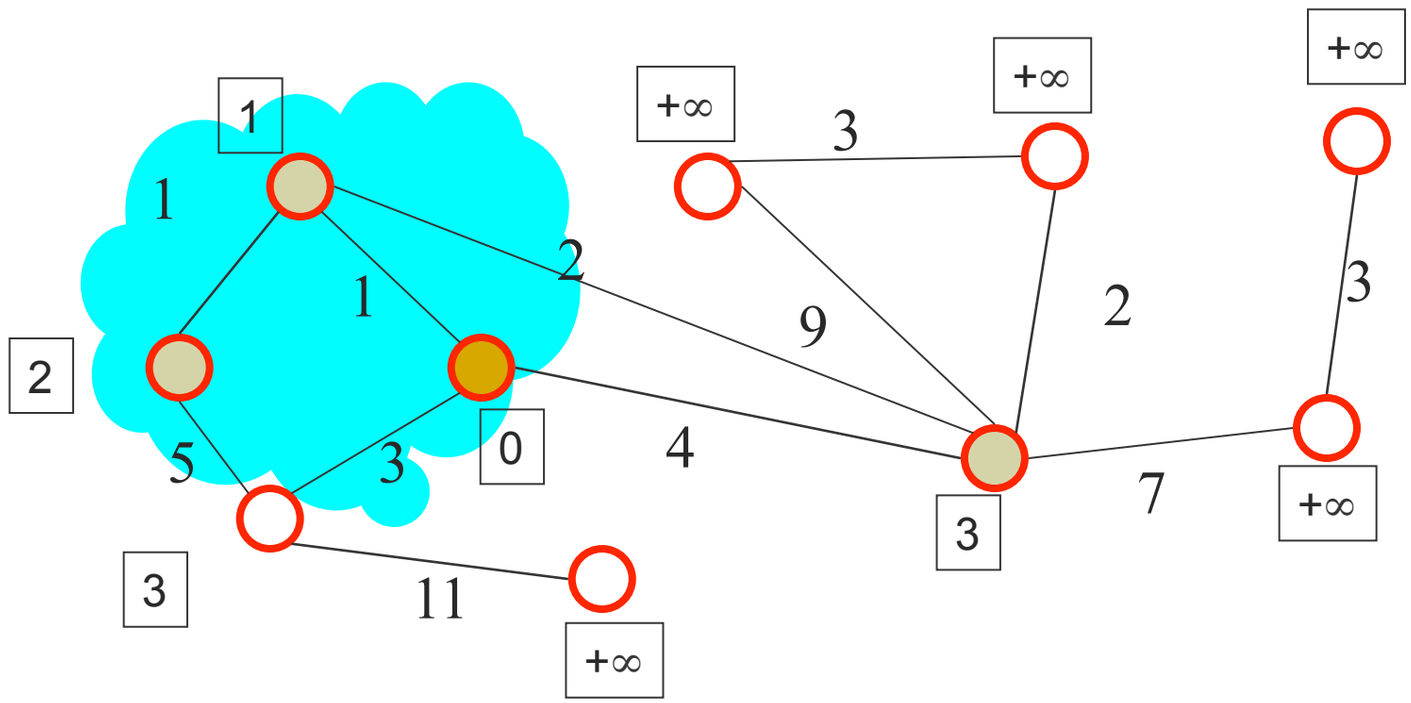
Dijkstra's algorithm: pull u into C



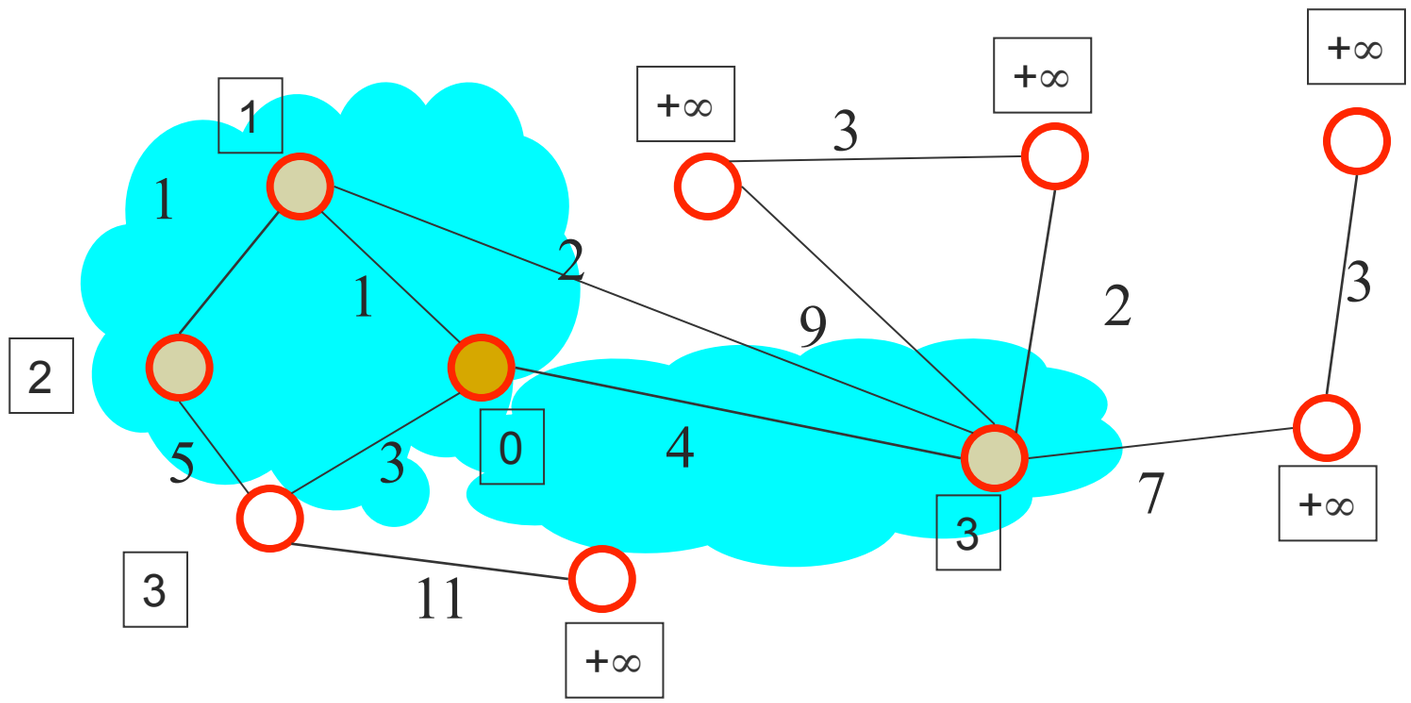
Dijkstra's algorithm: update $C's$ neighborhood



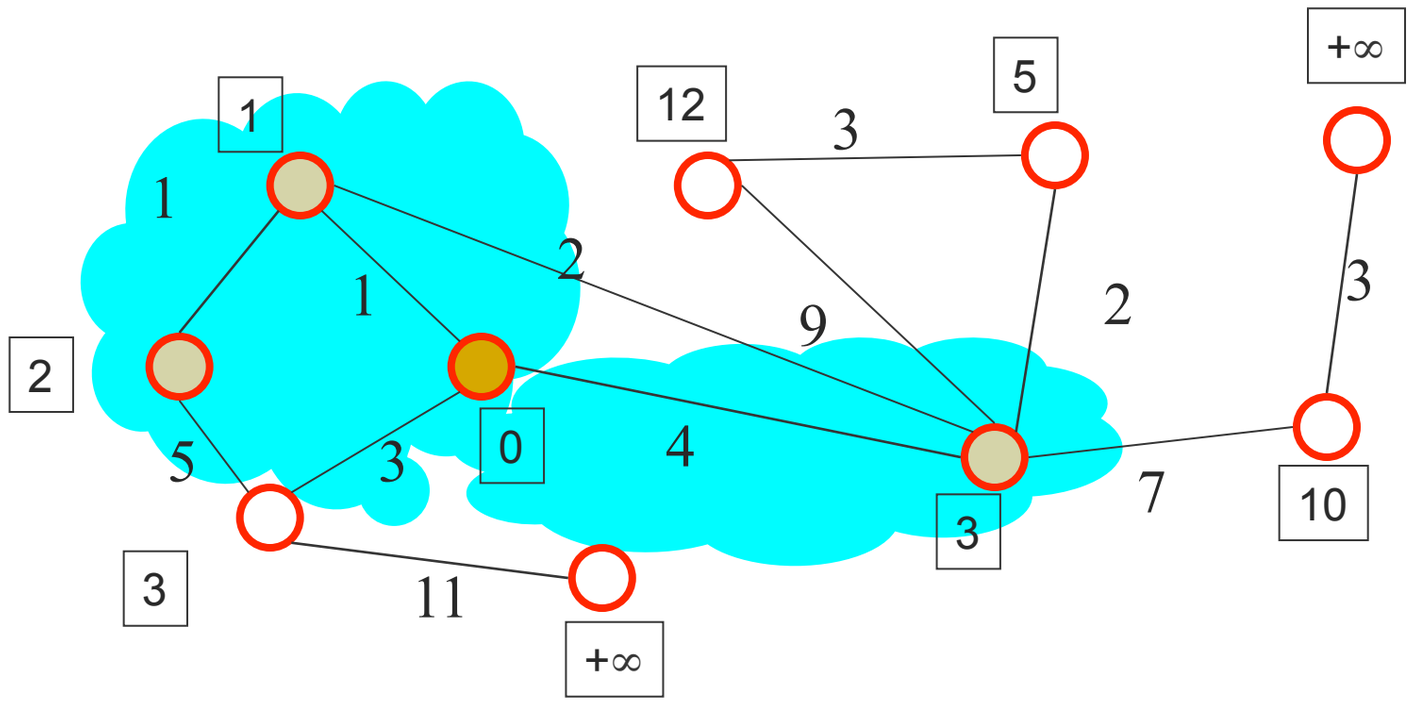
Dijkstra's algorithm: pick closest vertex u outside C



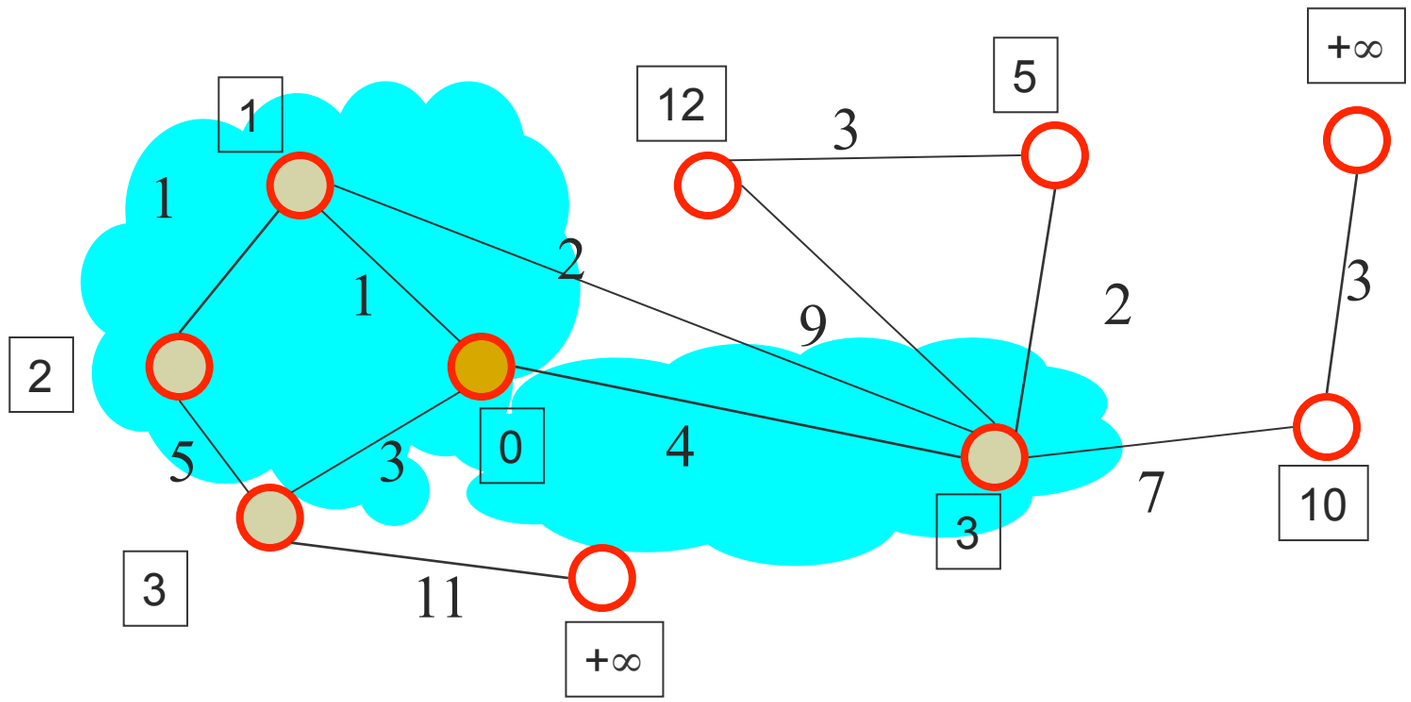
Dijkstra's algorithm: pull u into C



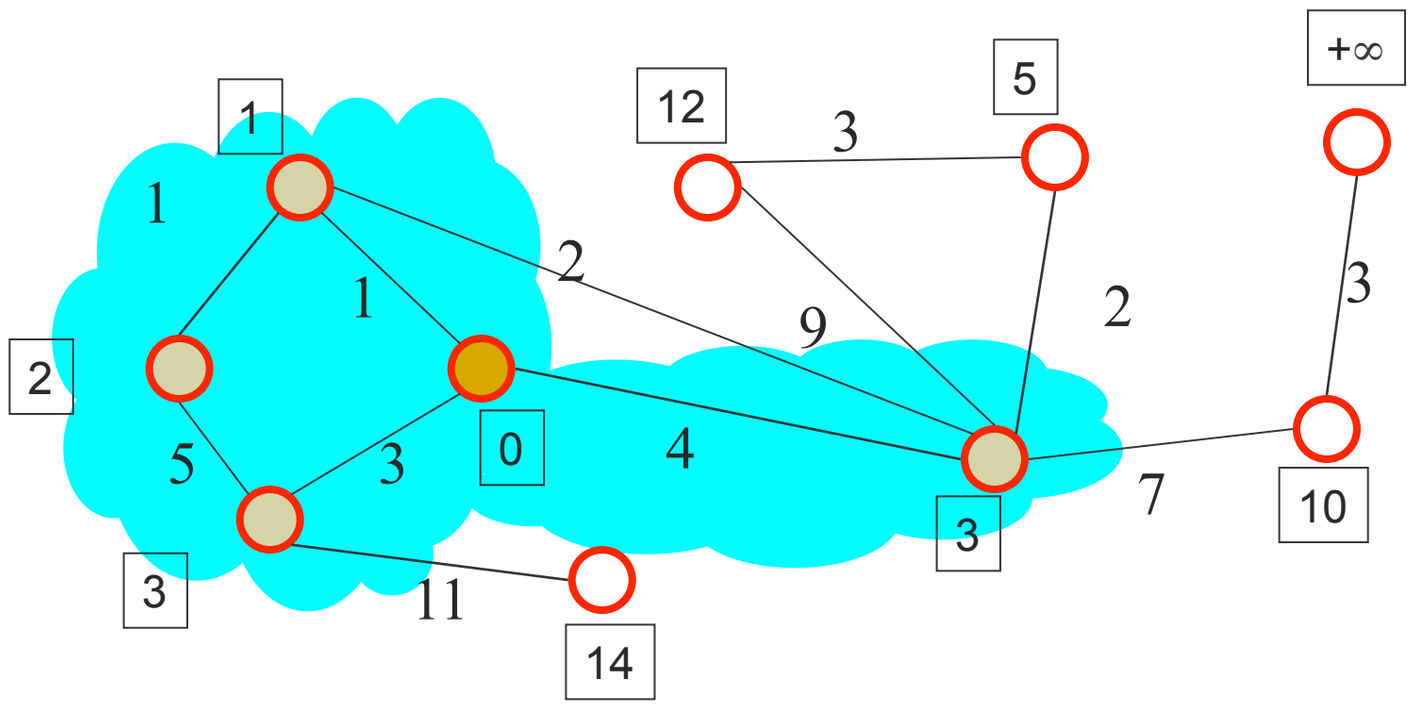
Dijkstra's algorithm: update $C's$ neighborhood



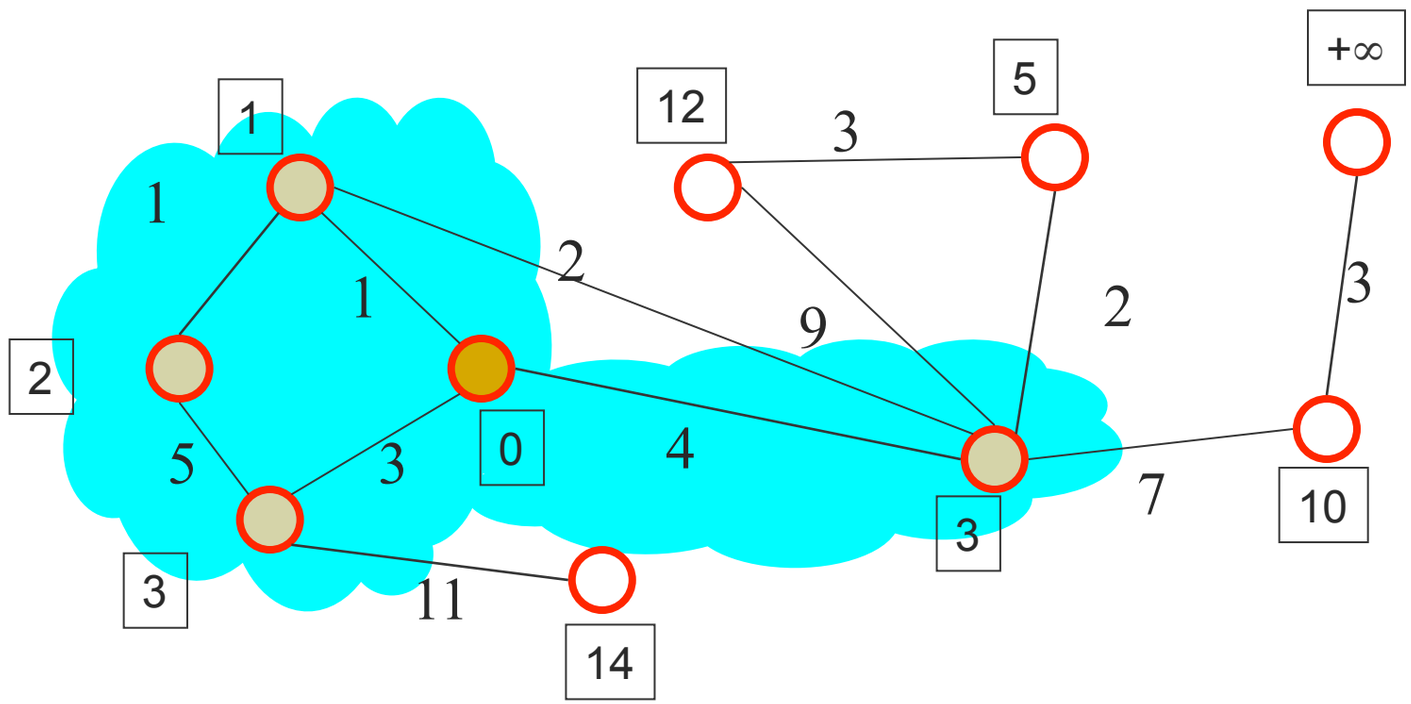
Dijkstra's algorithm: pick closest vertex u outside C



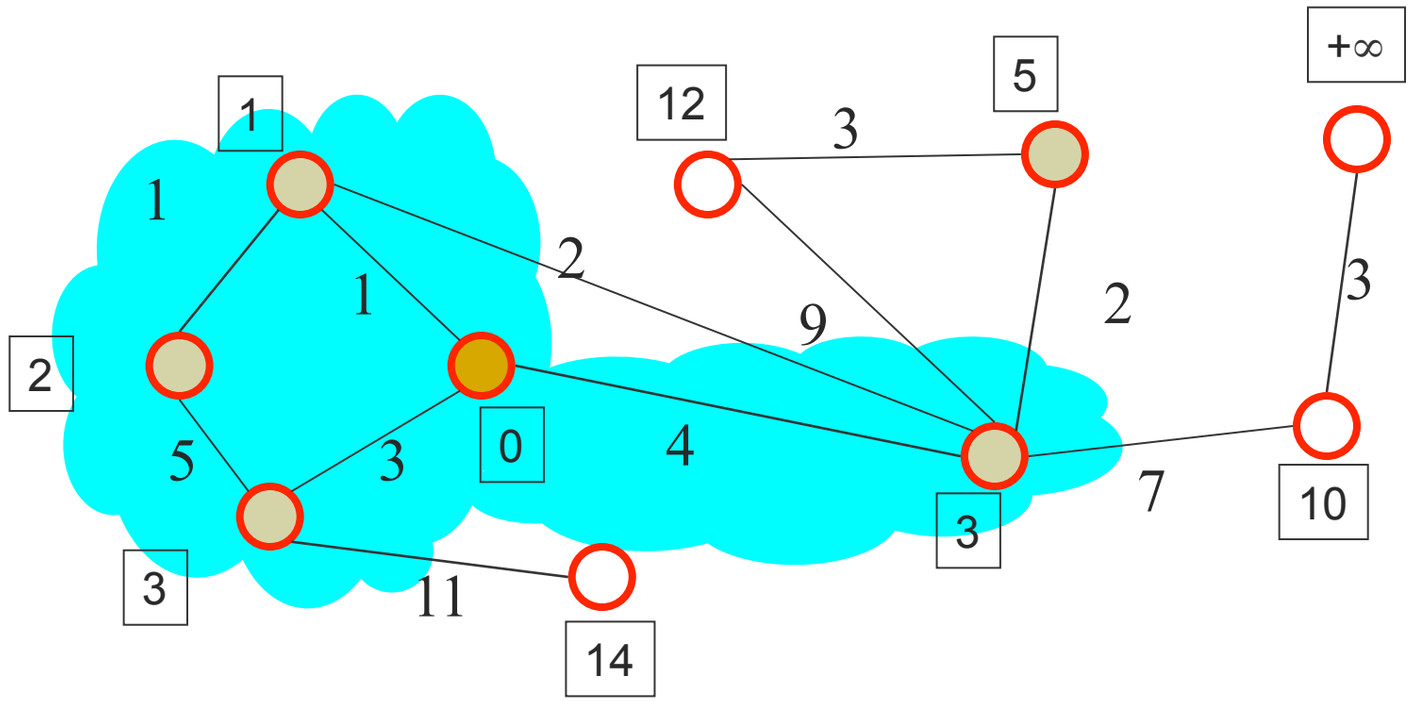
Dijkstra's algorithm: pull u into C



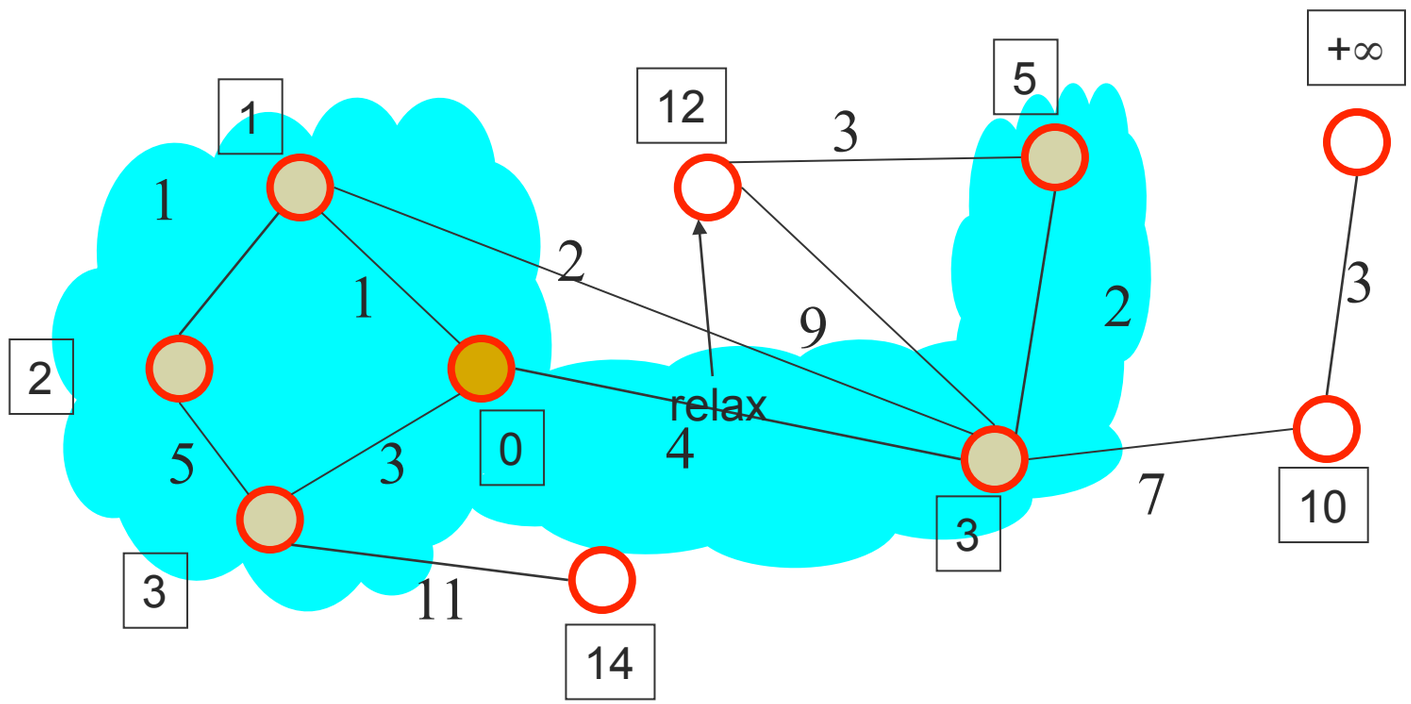
Dijkstra's algorithm: update $C's$ neighborhood



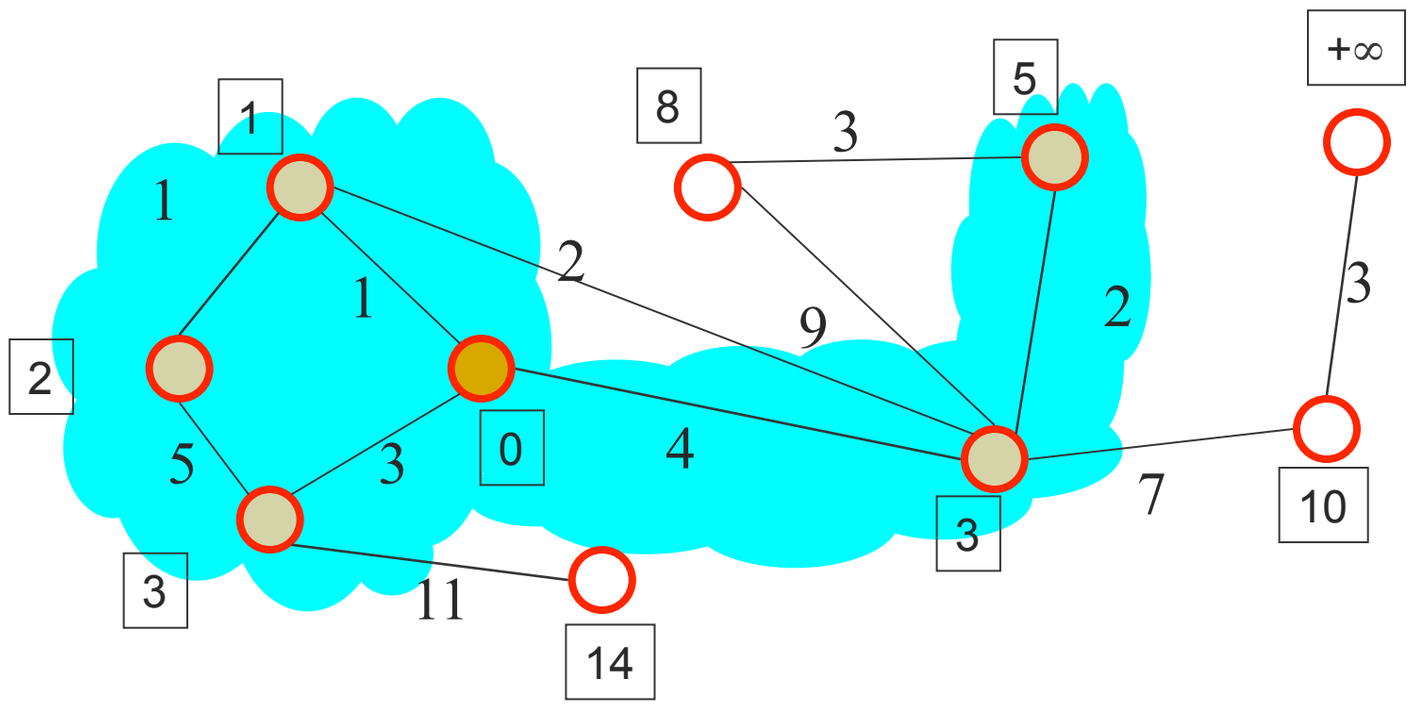
Dijkstra's algorithm: pick closest vertex u outside C



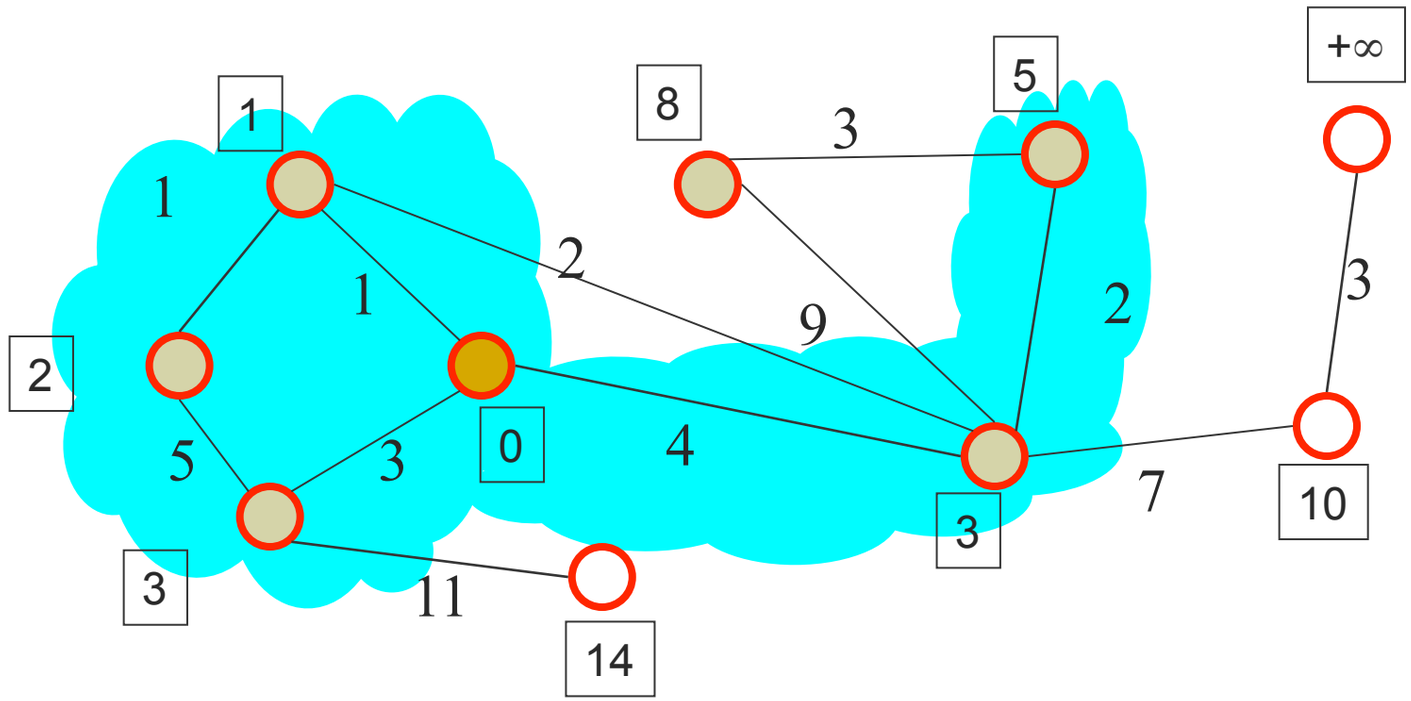
Dijkstra's algorithm: pull u into C



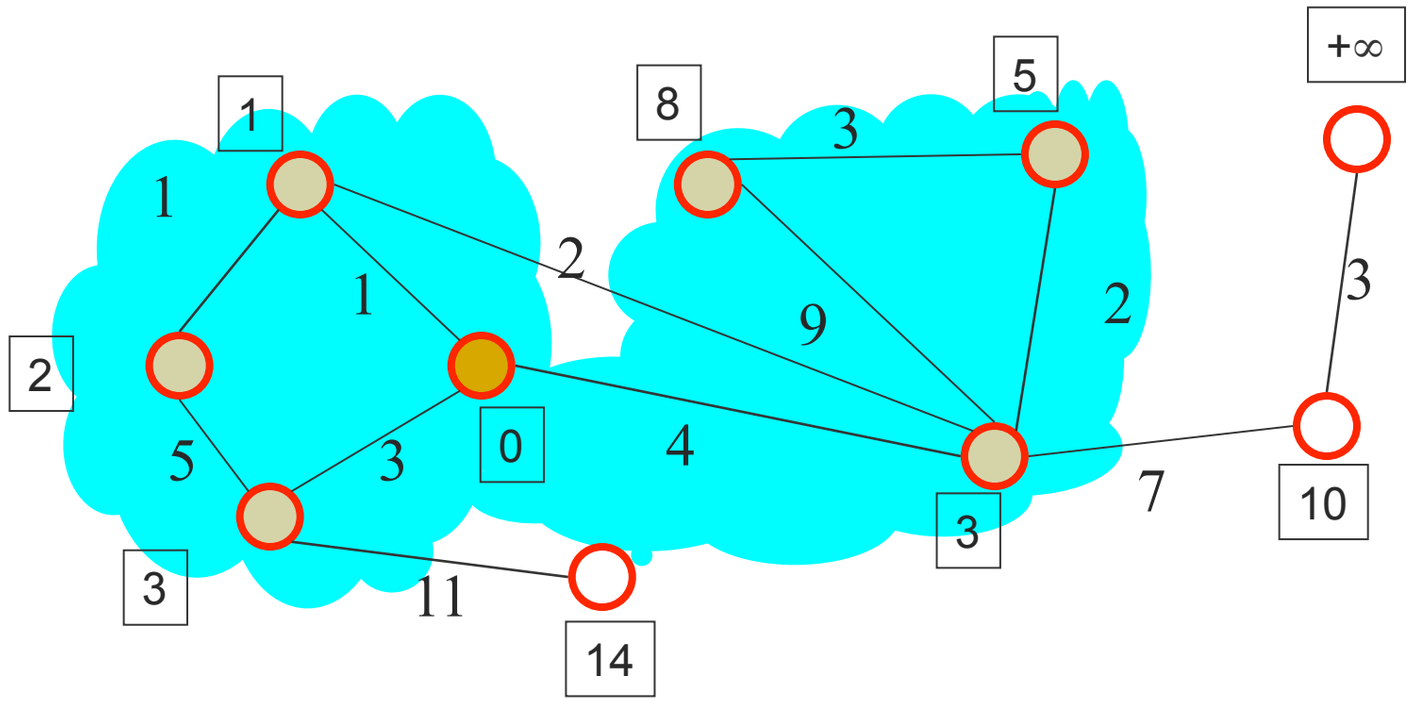
Dijkstra's algorithm: update $C's$ neighborhood



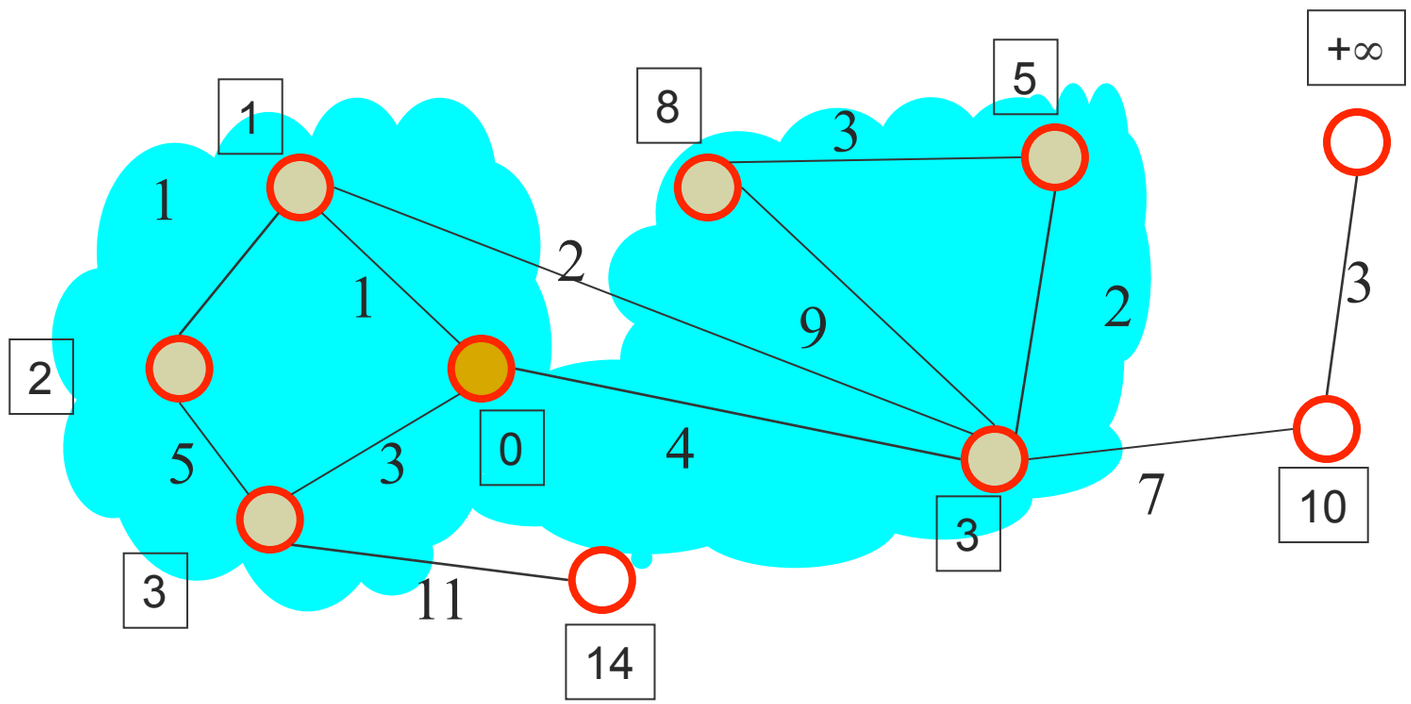
Dijkstra's algorithm: pick closest vertex u outside C



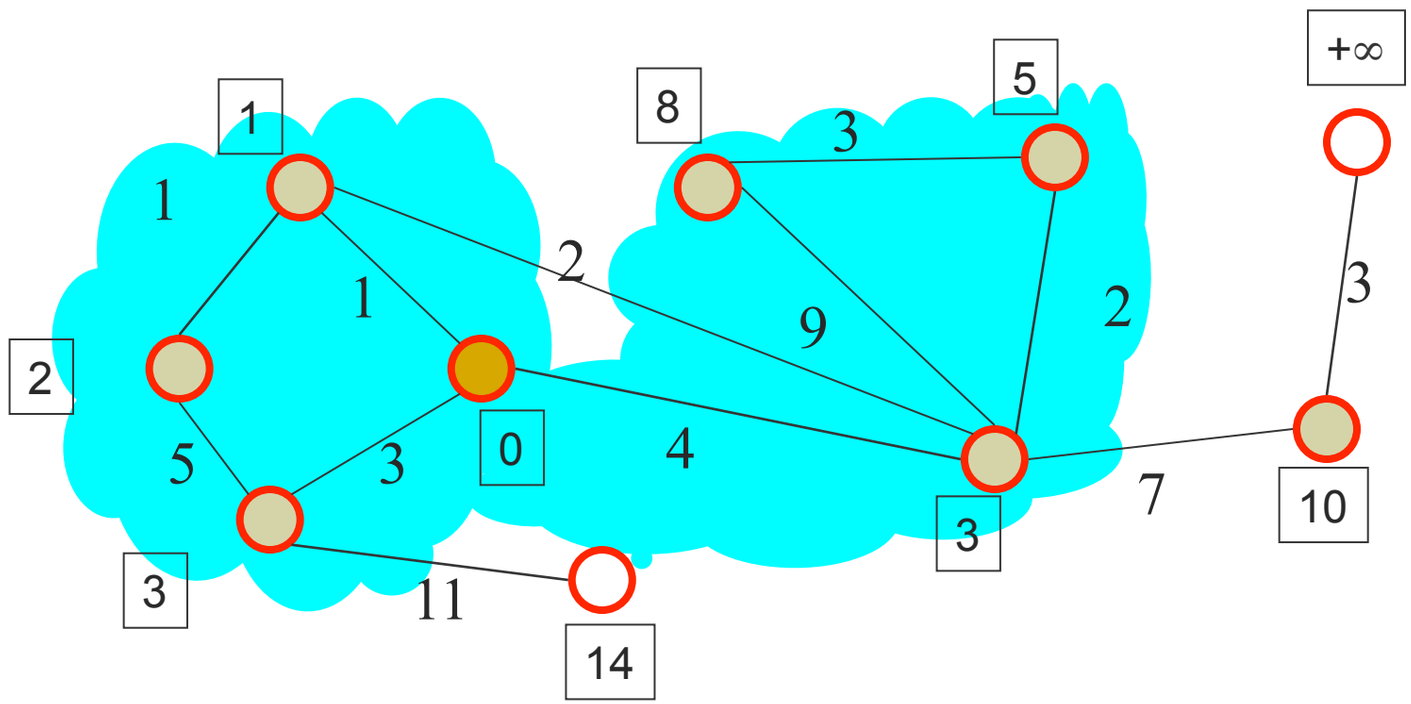
Dijkstra's algorithm: pull u into C



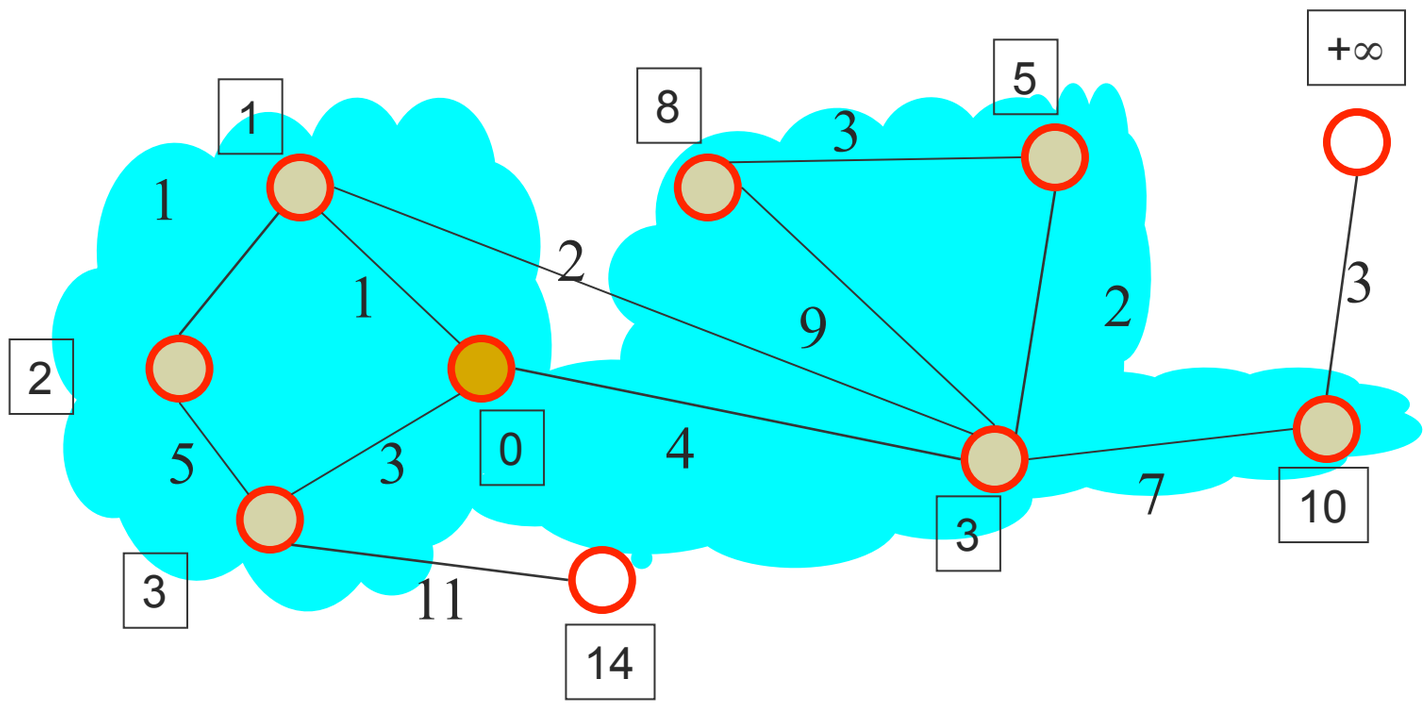
Dijkstra's algorithm: update $C's$ neighborhood



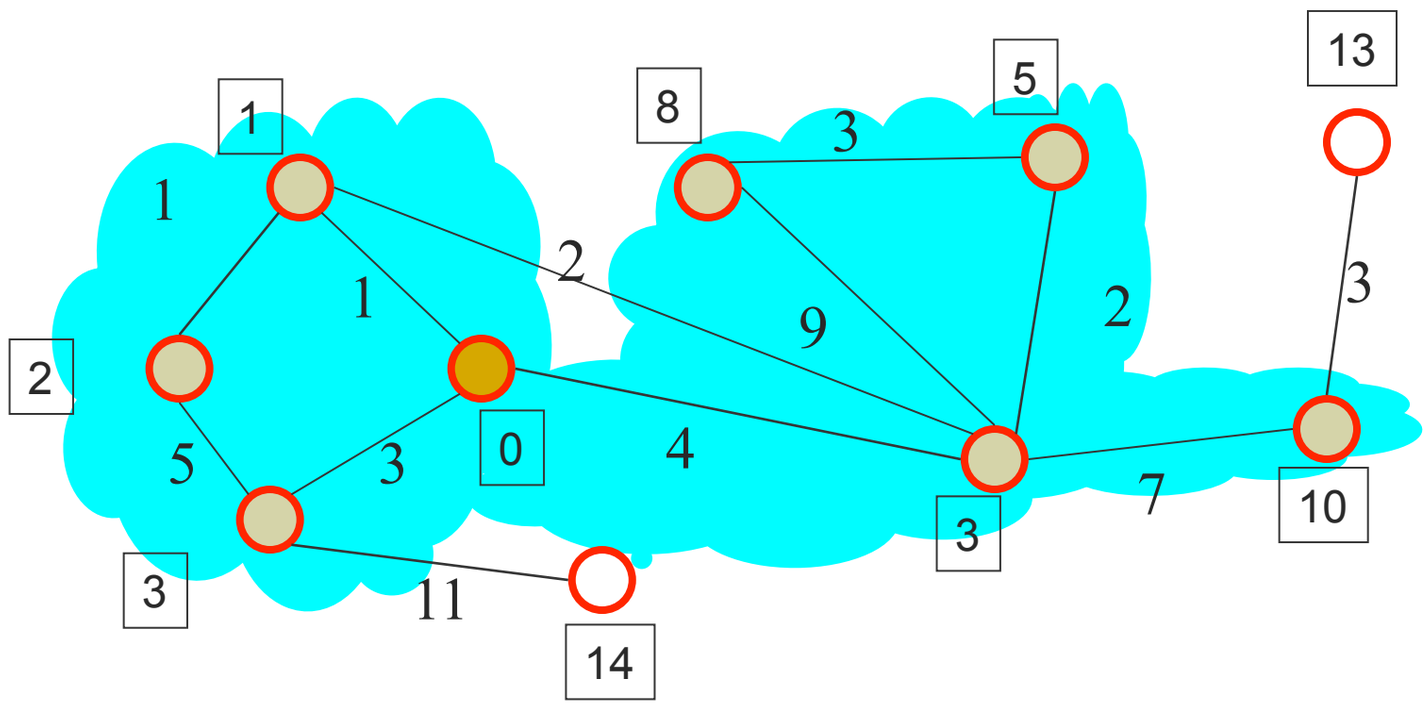
Dijkstra's algorithm: pick closest vertex u outside C



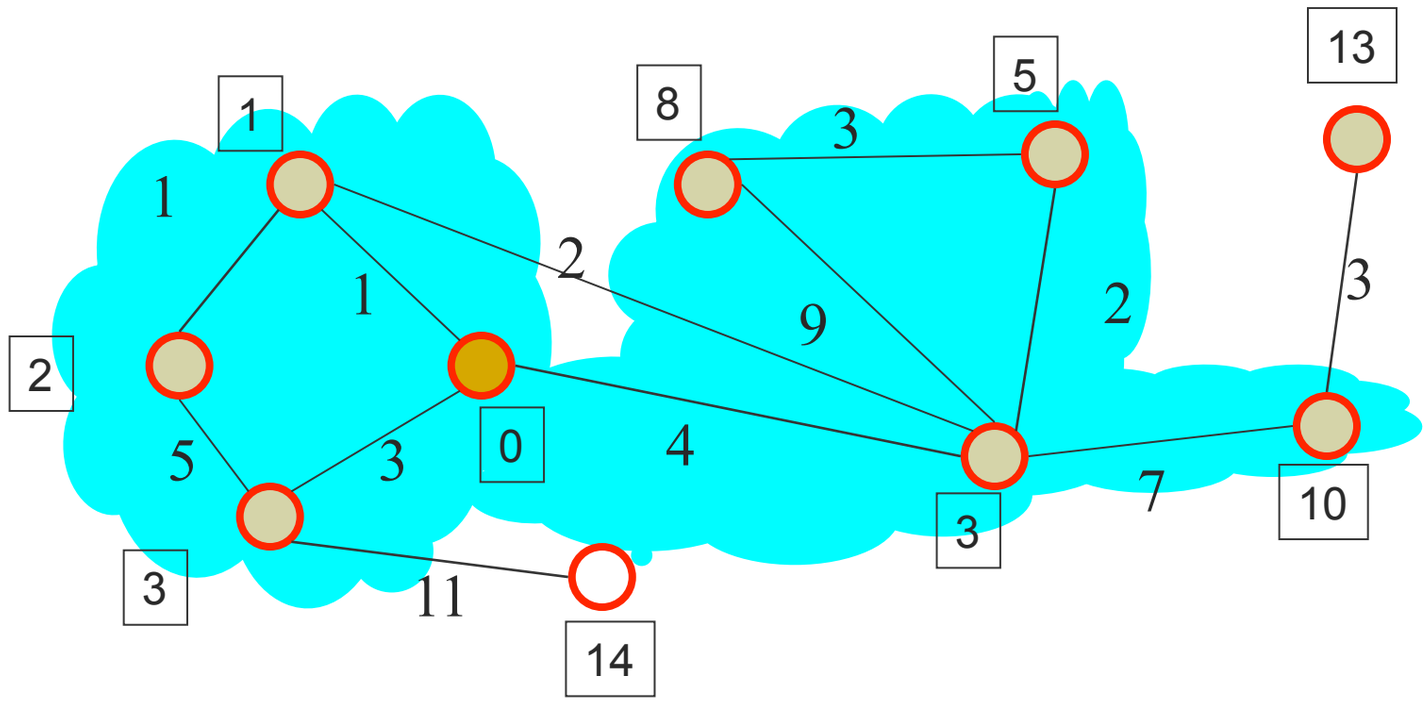
Dijkstra's algorithm: pull u into C



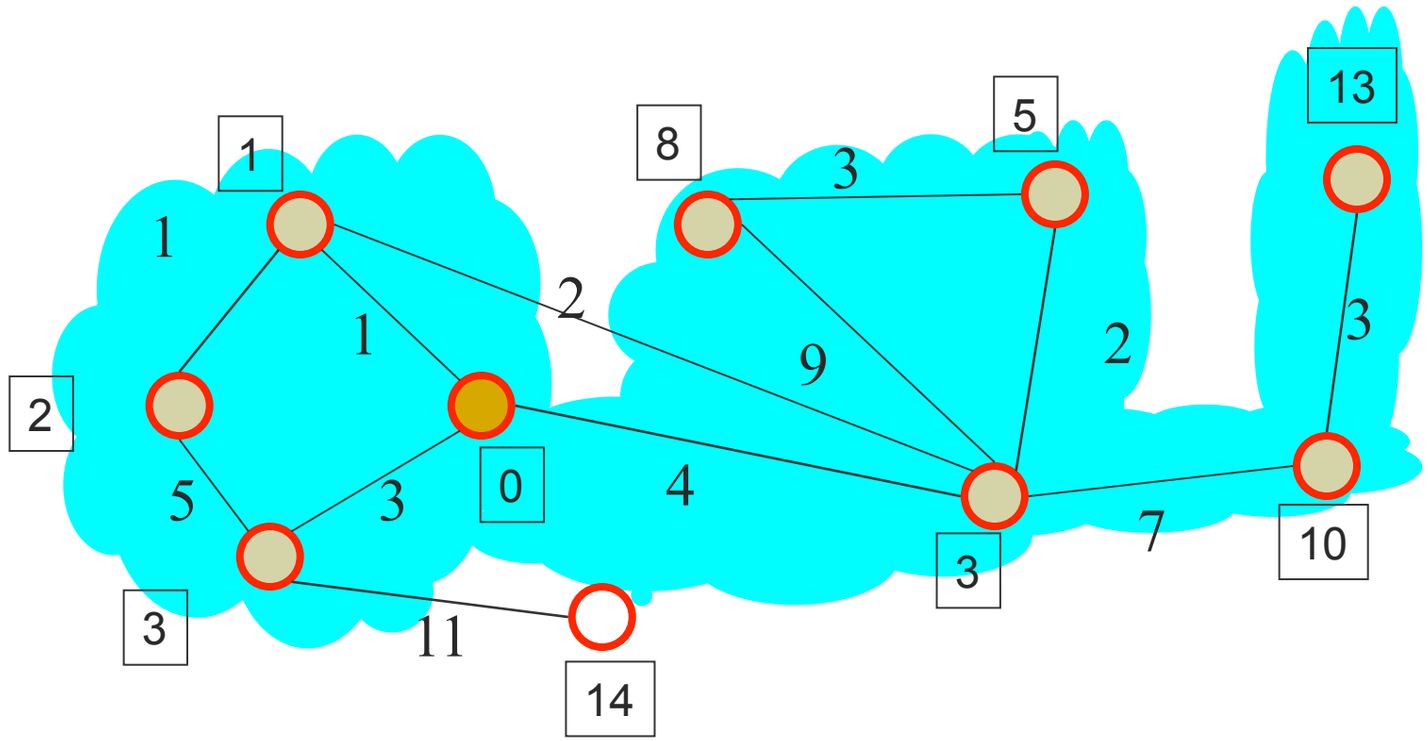
Dijkstra's algorithm: update $C's$ neighborhood



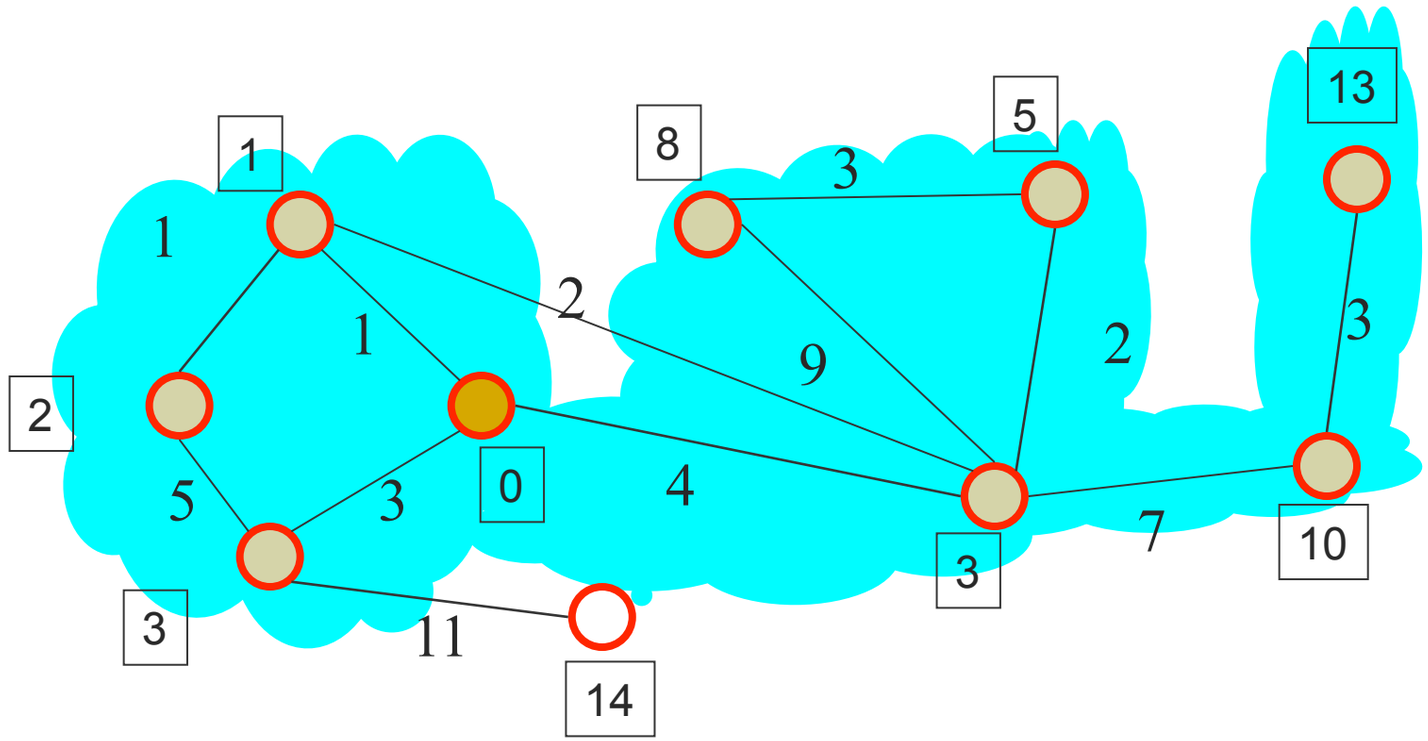
Dijkstra's algorithm: pick closest vertex u outside C



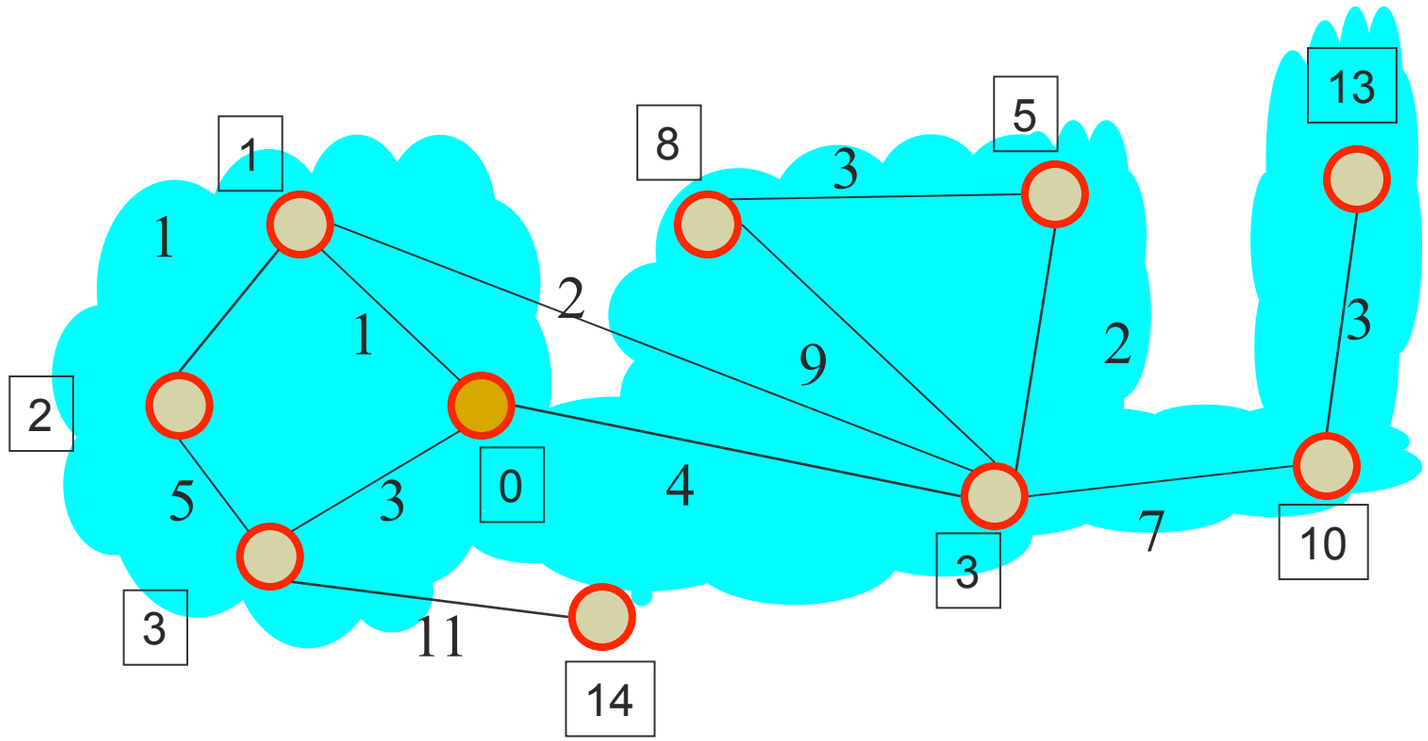
Dijkstra's algorithm: pull u into C



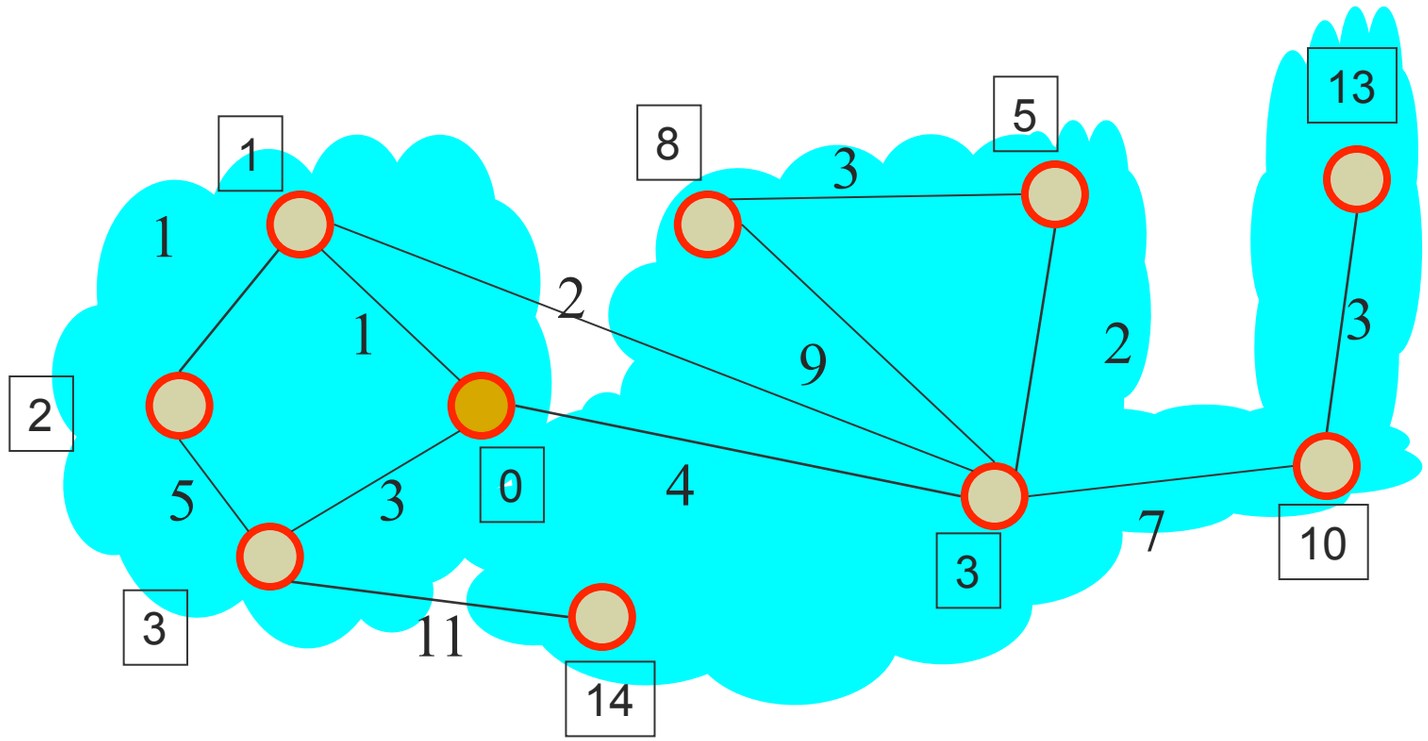
Dijkstra's algorithm: update $C's$ neighborhood



Dijkstra's algorithm: pick closest vertex u outside C



Dijkstra's algorithm: pull u into C



Dijkstra's Algorithm

```
Dijkstra(V, E, s):
    For v in V
        d[v] ← ∞ ; π[v] ← null;
    d[s] ← 0
    S ← ∅
    Q = BuildPriorityQueue(V, d)
    While Q not empty
        u ← DeleteMin(Q)
        S ← S ∪ u
        For v in Adj[u]
            Relax(u,v)
```

```
RELAX(u, v)
    If d[u] + w(u, v) < d[v]
        d[v] ← d[u] + w(u, v)
        π[v] ← u
```

Dijkstra vs Prim

```
Dijkstra(V, E, s):
    For v in V
        d[v] ← ∞ ; π [v] ← null;
    d[s] ← 0
    S ← ∅
    Q = BuildPriorityQueue(V, d)
    While Q not empty
        u ← DeleteMin(Q)
        S ← S ∪ u
        For v in Adj[u]
            If d[u] + w(u, v) < d[v]
                d[v] ← d[u] + w(u, v)
                π[v] ← u
                UpdatePQ(v, d[v])
```

```
Prim(V, E, s):
    For v in V
        d[v] ← ∞ ; π [v] ← null;
    d[s] ← 0
    S ← ∅
    Q = BuildPriorityQueue(V, d)
    While Q not empty
        u ← DeleteMin(Q)
        S ← S ∪ u
        For v in Adj[u]
            If w(u, v) < d[v]
                d[v] ← w(u, v)
                π[v] ← u
                UpdatePQ(v, d[v])
```

Dijkstra's Algorithm - Runtime

```
Dijkstra(V, E, s):
    For v in V
        d[v] ← ∞ ; π [v] ← null;
    d[s] ← 0
    S ← ∅
    Q = BuildPriorityQueue(V, d)
    While Q not empty
        u ← DeleteMin(Q)
        S ← S ∪ u
        For v in Adj[u]
            If d[u] + w(u, v) < d[v]
                d[v] ← d[u] + w(u, v)
                π[v] ← u
                UpdatePQ(v, d[v])
```

O(V) for binary or Fibonacci heap

V calls

O(log V)/call for binary or Fibonacci heaps

at most E calls

O(log V)/call for binary heap

O(1)/call for Fibonacci heap

RELAX preserves upper bound property of d[v]

- **Upper bound property:** Any sequence of calls to RELAX maintains the invariant that $d[v] \geq \delta(s, v)$ for all $v \in V$

- Proof: simple exercise

- Important consequence:

  If no path from s to v, then $d[v] = \infty$ always!

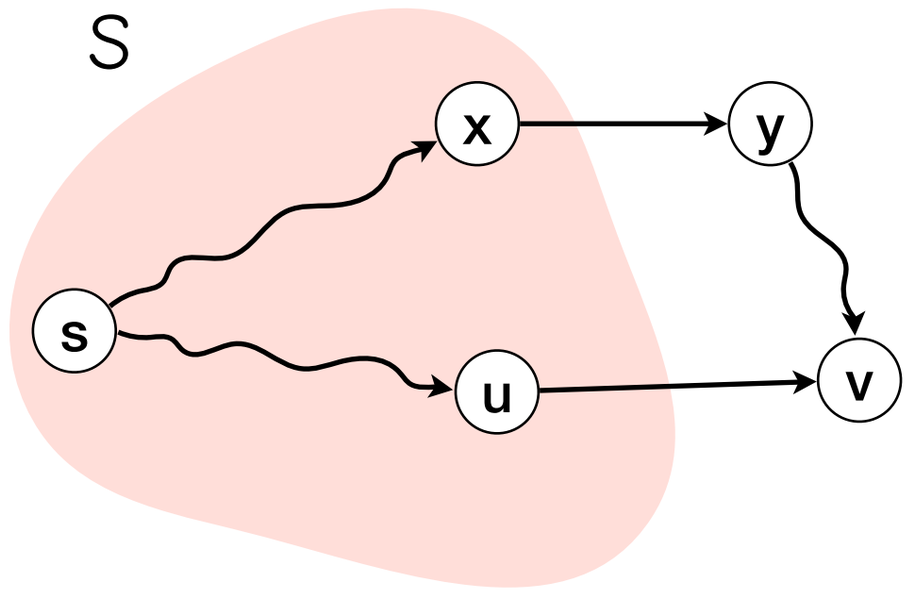
Dijkstra's Algorithm - Correctness

- Claim: for all v in S, the algorithm's path P_v from s-v is a shortest s-v path
- Proof by induction
- Base case: |S| = 1, with S = {s}
  - Clearly, P_s = (s) is a shortest s-s path (of length zero!)
  - Inductive step
    - Suppose the claim holds for |S| = k
    - Prove that it holds for |S| = k + 1

Dijkstra's Algorithm - Correctness

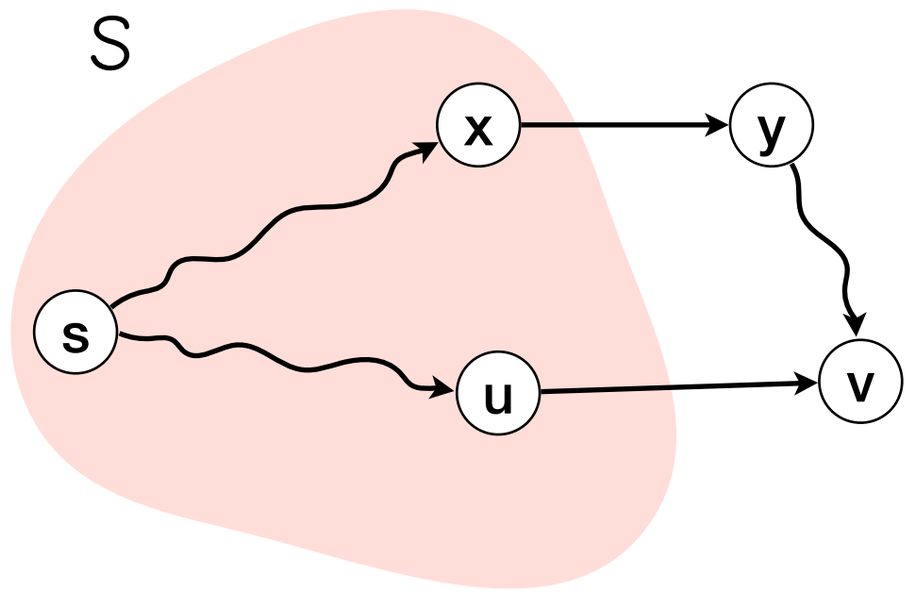
- Let |S| = k and suppose algorithm is about to add v to S, by way of u in S
- Let P_v be the algorithm's s-v path after the addition, with penultimate vertex u in S
- Consider an arbitrary alternative path P'_v
- P'_v has a first edge (x, y) that crosses the cut $(S, V \setminus S)$

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



Dijkstra's Algorithm - Correctness

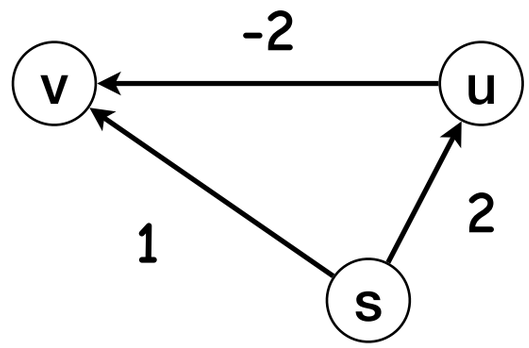
- Consider an arbitrary alternative path P'_v
- P'_v has a first edge (x, y) that crosses the cut $(S, V \setminus S)$



Path P'_v cannot be shorter than P_v

$$
\begin{aligned}
w(P'_v) &\geq \delta(s,x) + w(x,y) \\
&= d[x] + w(x,y) && \text{(inductive hypothesis)} \\
&\geq d[u] + w(u,v) && \text{(v is next vertex added to S)} \\
&= \delta(s,u) + w(u,v) && \text{(inductive hypothesis)} \\
&= w(P_v)
\end{aligned}
$$

Dijkstra's Algorithm - Negative Weights



What would Dijkstra do?

"Greed is good."

-Gordon Gekko

"Greed is good."
-Gordon Gekko

"Greed is not good
(when a graph has
negative edge weights)."
- Bernie Sanders

Bellman-Ford Algorithm

Recall: Path Relaxation Property

- Let $p = (v_0, v_1, \ldots v_k)$ be a shortest path from v_0 to v_k
- Initialize d and π with source s
- Suppose that a sequence of Relax calls occurs which includes the subsequence:
  - RELAX(v_0, v_1), RELAX(v_1, v_2), …, RELAX(v_{k-1}, v_k)
- Then after the last RELAX call in this subsequence and for all times thereafter, we have $d[v_k] = \delta(s, v_k)$
- (Proved last time)

Bellman-Ford Algorithm

- An Observation:
  - Suppose shortest path from vertex s to vertex *t* consists of 1 edge: p = (v_0, v_1) with s = v_0 and *t* = v_1
  - Then after calling RELAX(v_0, v_1):

$$d[t] = d[v_1] = \delta(v_0, v_1) = \delta(s, t)$$

    - Shortest path from s to *t* has been found!
- How to ensure RELAX(v_0, v_1) gets called?

Initialize d and π with source s

For each edge (u, v) $\in$ E

RELAX(u, v)

Bellman-Ford Algorithm

- An Observation:
  - Suppose shortest path from vertex s to vertex *t* consists of 2 edges: p = (s = v_0, v_1, v_2 = *t*)
  - Then after calling RELAX(v_0, v_1), RELAX(v_1, v_2):

$$d[t] = d[v_2] = \delta(v_0, v_2) = \delta(s, t)$$

    - Shortest path from s to *t* has been found!
- How to ensure RELAX(v_0, v_1), RELAX(v_1, v_2) gets called?

Initialize d and π with source s

For j = 1 $\rightarrow$ 2

For each edge (u, v) $\in$ E

RELAX(u, v)

Bellman-Ford Algorithm

- If no negative cycles, shortest path from vertex s to vertex *t* consists of (at most) n-1 edges: p = (v_0, v_1, …, v_k) with $k \leq$ n-1
- After calling RELAX(v_0, v_1), RELAX(v_1, v_2), …, RELAX(v_{k-1}, v_k):

$$d[t] = d[v_k] = \delta(v_0, v_k) = \delta(s, t)$$

  - Shortest path from s to *t* has been found!
- How to ensure subsequence RELAX(v_0, v_1), …, RELAX(v_{k-1}, v_k) of calls occurs?

Initialize d and π with source s

For j = 1 $\rightarrow$ n-1

  For each edge (u, v) $\in$ E

    RELAX(u, v)

Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

Initialize d and π with source s

For j = 1 $\rightarrow$ n-1

For each edge (u, v) $\in$ E

RELAX(u, v)

For each edge (u, v) $\in$ E

If d[v] > d[u] + w(u, v)

Return False

Return True

RELAX(u,v)

If d[u] + w(u,v) < d[v]

d[v] $\leftarrow$ d[u] + w(u,v)

π[v] $\leftarrow$ u

Bellman-Ford Algorithm - Correctness

Claim 1:

If there are no negative cycles:

(A) The algorithm correctly finds the shortest paths
($d[v] = \delta(s, v)$ for all v) and predecessor array is correct.

(B) The algorithm returns True.

Claim 2:

If there is a negative cycle, the algorithm detects it
and returns False

Bellman-Ford Algorithm - Correctness

Claim 1:

If there are no negative cycles:

(A) The algorithm correctly finds the shortest paths
($d[v] = \delta(s, v)$ for all v) and predecessor array is correct.

Proof:

This we already showed in the derivation of the algorithm!

The desired subsequence of calls to RELAX occurs,
which is all that is required.

Bellman-Ford Algorithm - Correctness

Claim 1:

If there are no negative cycles:

(B) The algorithm returns True

Proof:

We only need to verify that

$$d[v] \leq d[u] + w(u, v) \text{ for all edges } (u, v) \in E$$

From Claim 1 (A), this is equivalent to

$$\delta(s, v) \leq \delta(s, u) + w(u, v) \text{ for all edges } (u, v) \in E$$

This must be the case. Why? An s-v path that first visits u and then follows edge (u,v) cannot have less weight than the shortest s-v path

Bellman-Ford Algorithm - Correctness

Claim 2:

If there is a negative cycle, the algorithm detects it and returns False

Proof:

Suppose (for a contradiction) that the algorithm returns True

Then $d[v] \leq d[u] + w(u, v)$ for all edges $(u, v) \in E$ (★)

Let the negative cycle be $(v_0, v_1, \ldots, v_k)$, where $v_0 = v_k$

Summing inequality (★) over each edge in the cycle yields:

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

Bellman-Ford Algorithm - Correctness

(Proof of Claim 2)

Suppose (for a contradiction) that the algorithm returns True

Then $d[v] \leq d[u] + w(u, v)$ for all edges $(u, v) \in E$ (★)

Let the negative cycle be $(v_0, v_1, \ldots, v_k)$, where $v_0 = v_k$

Summing inequality (★) over each edge in the cycle yields:

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

But since $v_0 = v_k$, it holds that $\sum_{j=1}^{k} d[v_j] = \sum_{j=1}^{k} d[v_{j-1}]$

The above implies that $0 \leq \sum_{j=1}^{k} w(v_{j-1}, v_j)$,
a contradiction of the cycle being negative!

Bellman-Ford Algorithm - Time Complexity

- O(V E)
  - (For loop from 1 to n-1) · (Nested for loop over edges)
- Compare to Dijkstra's algorithm
  - O(E log V)
  - Dealing with negative-weight edges has a cost!

Single-Source Shortest Paths Algorithms

Type of Graph	Algorithm	Time complexity
Unweighted graph	BFS	O(V + E)
Weighted DAG	Topological sort/DFS-based	O(V + E)
Weighted directed graph (nonnegative weights)	Dijkstra's - Binary heap	O(E log V)
	Dijkstra's - Fibonacci heap	O(V log V + E)
Weighted directed graph (any weights)	Bellman-Ford	O(V E)

All-Pairs Shortest Paths Problem

- **Input**: A weighted directed graph G = (V, E)
- **Output**: All shortest paths in G, i.e., for all pairs of vertices s, t in V, a shortest path from s to t

All-Pairs Shortest Paths

- First approach - run single-source shortest paths algorithm n times, once per choice of source vertex

Type of Graph	Algorithm	Time complexity	Dense Graph Time complexity
Nonnegative weights	Dijkstra's - Binary heap	$O(V E \log V)$	$O(V^3 \log V)$
	Dijkstra's - Fibonacci heap	$O(V^2 \log V + V E)$	$O(V^3)$
Any weights	Bellman-Ford	$O(V^2 E)$	$O(V^4)$

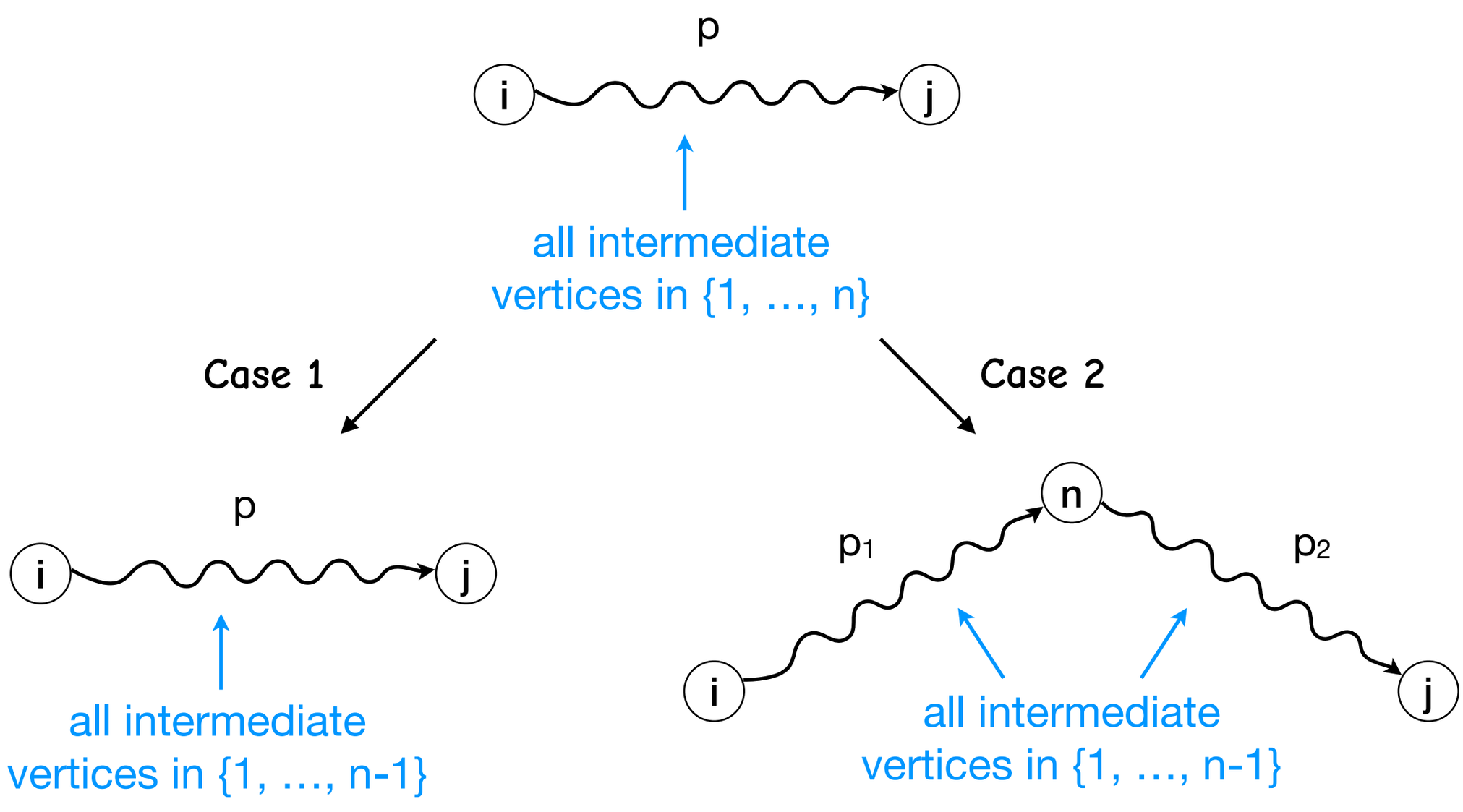
All-Pairs Shortest Paths

- Need to store upper bound on weight of shortest path for every pair of vertices
- Switch from array d to matrix D of size $n \times n$
  - D_{ij} = upper bound on weight of shortest path from i to j
- Switch from predecessor array π to predecessor matrix Π
  - Π_{ij} = predecessor of j in some shortest path from source i

Floyd-Warshall Algorithm

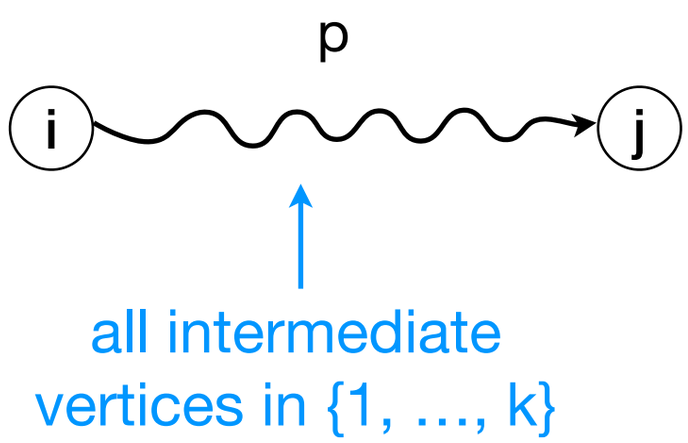
Floyd-Warshall Algorithm

- Let p be a shortest path from i to j.
- Clearly, all intermediate vertices in path p are in {1, …, n}
- Also, we can split p into at most 2 paths whose intermediate vertices are in {1, …, n-1}

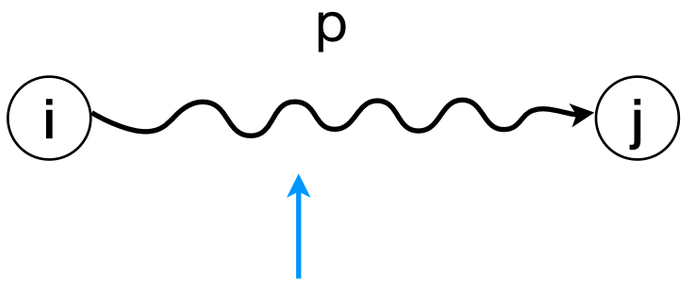


All-Pairs Shortest Paths

- For k = 0, 1, …, n: Let $D_{ij}^{(k)}$ be the weight of the shortest path from i to j for which all intermediate vertices are in {1, …, k}

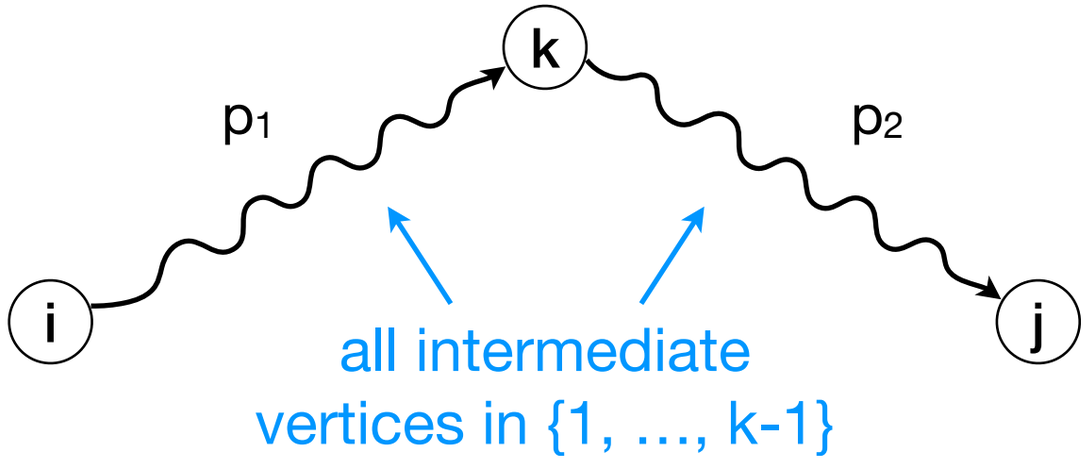


Case 1



$$D_{ij}^{(k)} = D_{ij}^{(k-1)}$$

Case 2



$$D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$$

Floyd-Warshall Algorithm

- Recurrence: $D_{ij}^{(k)} \leftarrow \min\left\{D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right\}$
- Base case: $D_{ij}^{(0)} \leftarrow w(i,j)$
  - Why? Because no intermediate vertices can be used

FLOYD-WARSHALL(W)

$D^{(0)} \leftarrow W$

For k = 1 $\rightarrow$ n

  For i = 1 $\rightarrow$ n

    For j = 1 $\rightarrow$ n

      $D_{ij}^{(k)} \leftarrow \min\left\{D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right\}$

Return $D^{(n)}$

Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

$D^{(0)} \leftarrow W$

For k = 1 → n

  For i = 1 → n

    For j = 1 → n

$$D_{ij}^{(k)} \leftarrow \min\left\{D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right\}$$

Return $D^{(n)}$

Time Complexity

$O(n^3)$

Correctness? $D_{ij}^{(n)}$ is weight of shortest path with intermediate vertices in {1, ..., n}. This is shortest path itself!

Floyd-Warshall Algorithm

What about that predecessor matrix?

How do we print a shortest path?

Case 1: $D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \geq D_{ij}^{(k-1)}$

Path will not change

Reuse predecessor from before: $\Pi_{ij}^{(k)} \leftarrow \Pi_{ij}^{(k-1)}$

Case 2: $D_{ik}^{(k-1)} + D_{kj}^{(k-1)} < D_{ij}^{(k-1)}$

Update path to [path from i to k] + [path from k to j]

$\Pi_{ij}^{(k)} \leftarrow \Pi_{kj}^{(k-1)}$

"Set predecessor of j in shortest path from source i using intermediate vertices in {1, …, k} to be predecessor of j in shortest path from source k using intermediate vertices in {1, …, k-1}"