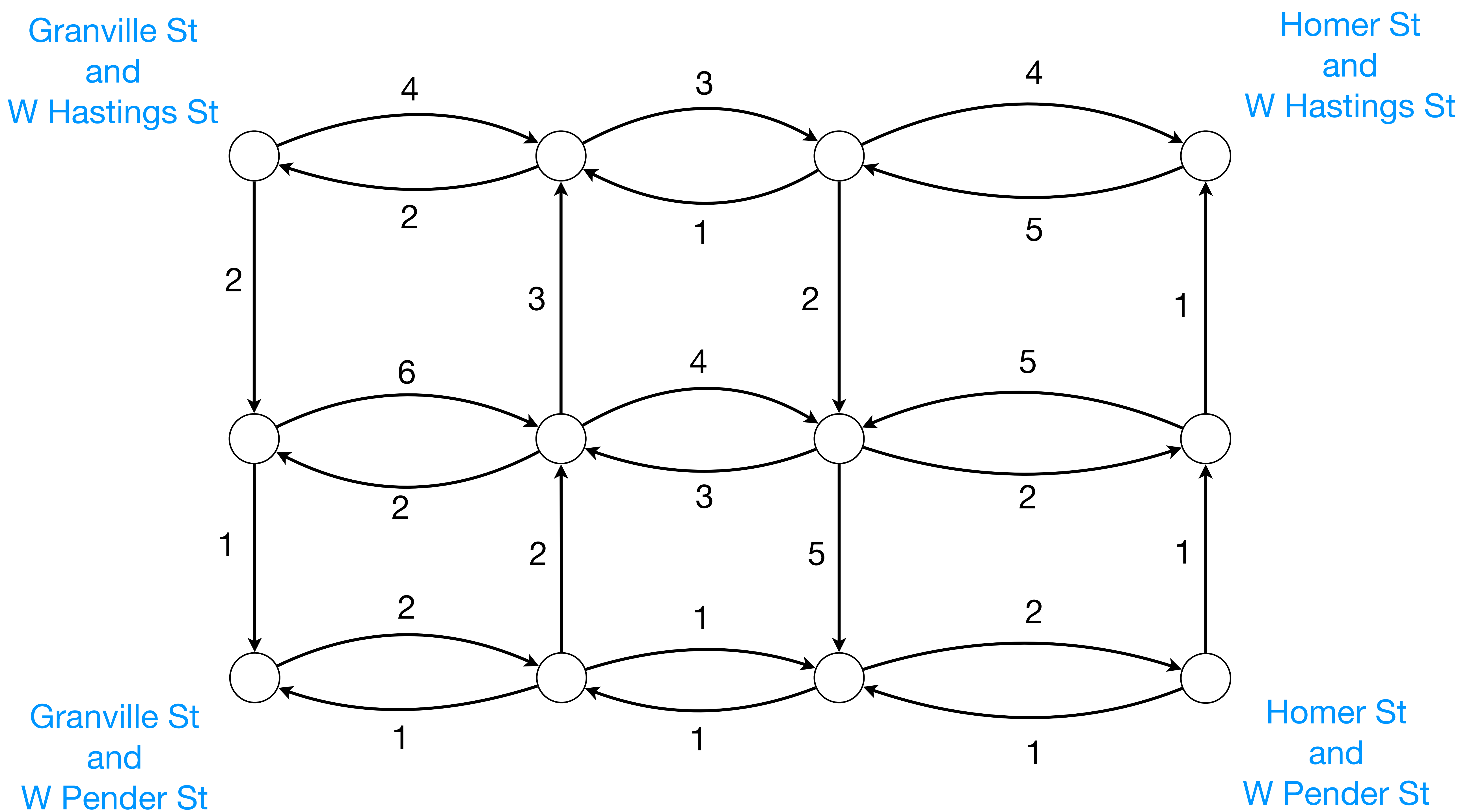
Shortest Paths

Nishant Mehta

Lectures 5-6

Finding the Fastest Way to Travel between Two Intersections in Vancouver



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.

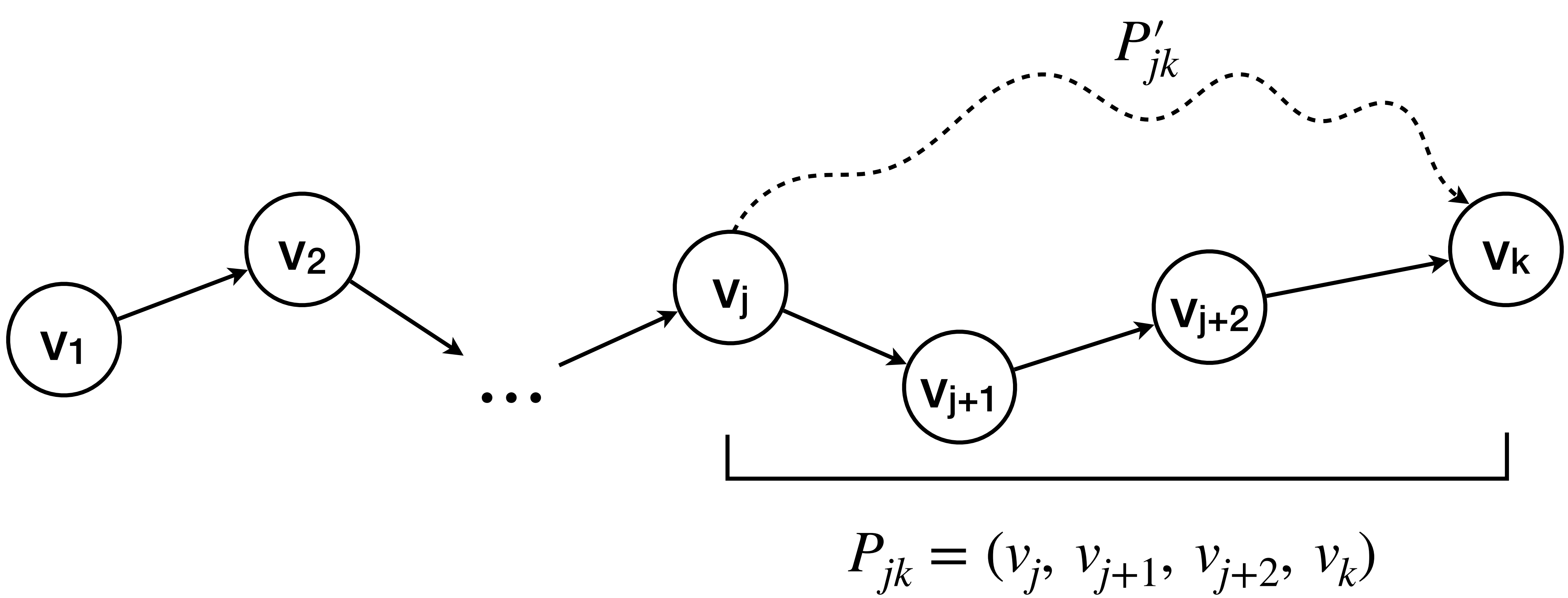
  - A path $p = (v_0, v_1, \ldots, v_k)$ from $s = v_0$ to $v = v_k$ is shortest if its length $w(p) = \sum_{j=1}^{k} w(v_{j-1}, v_j)$ is the minimum possible among all s–v paths

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



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):

$S \leftarrow \{s\}$

$d[s] \leftarrow 0$

While $S \neq V$

For all $v \notin S$ such that there is an edge (u, v) for some $u \in S$:

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

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

Add v to S

$d[v] \leftarrow c[v]$

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

$d[v] \leftarrow \infty$; $\pi[v] \leftarrow$ null;

$d[s] \leftarrow 0$

$S \leftarrow \emptyset$

Q = BuildPriorityQueue(V, d)

While Q not empty

$u \leftarrow$ DeleteMin(Q)

$S \leftarrow S \cup u$

For v in Adj[u]

Relax(u,v)

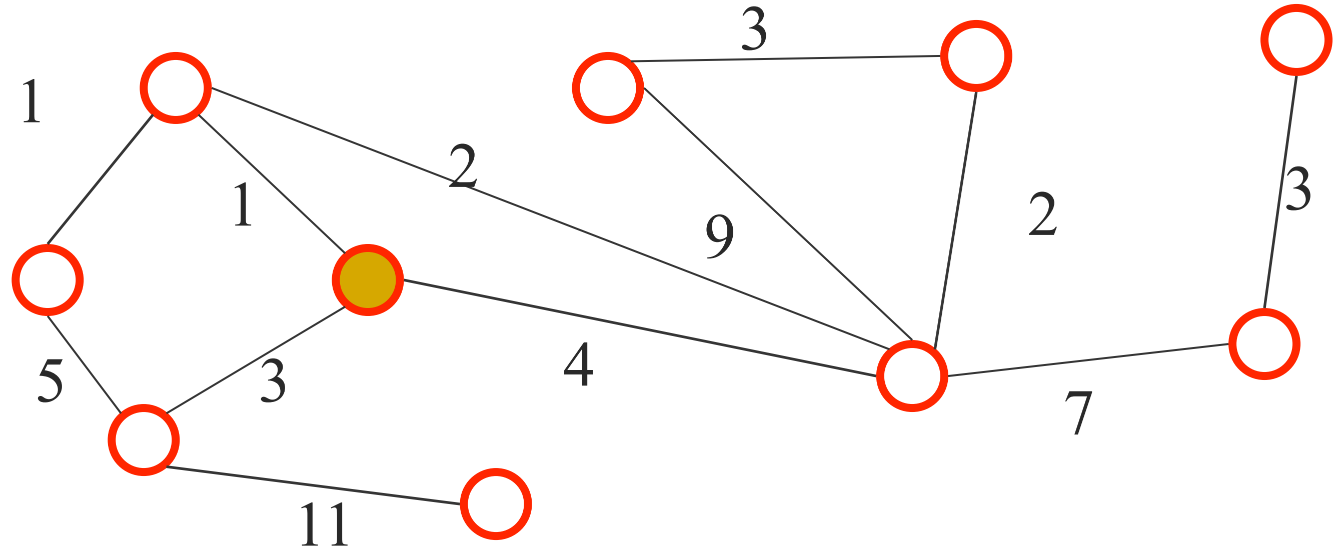
RELAX(u, v)

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

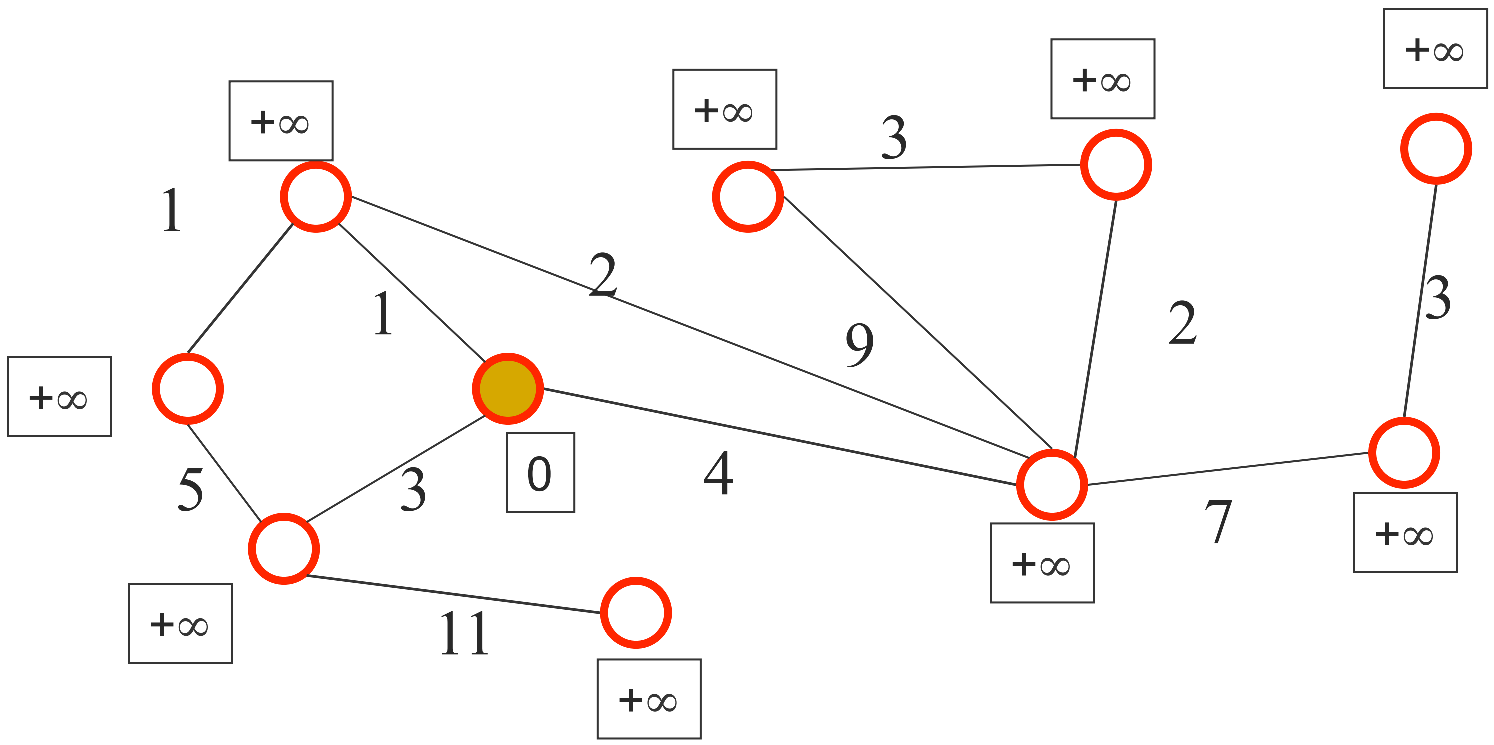
$d[v] \leftarrow d[u] + w(u, v)$

$\pi[v] \leftarrow u$

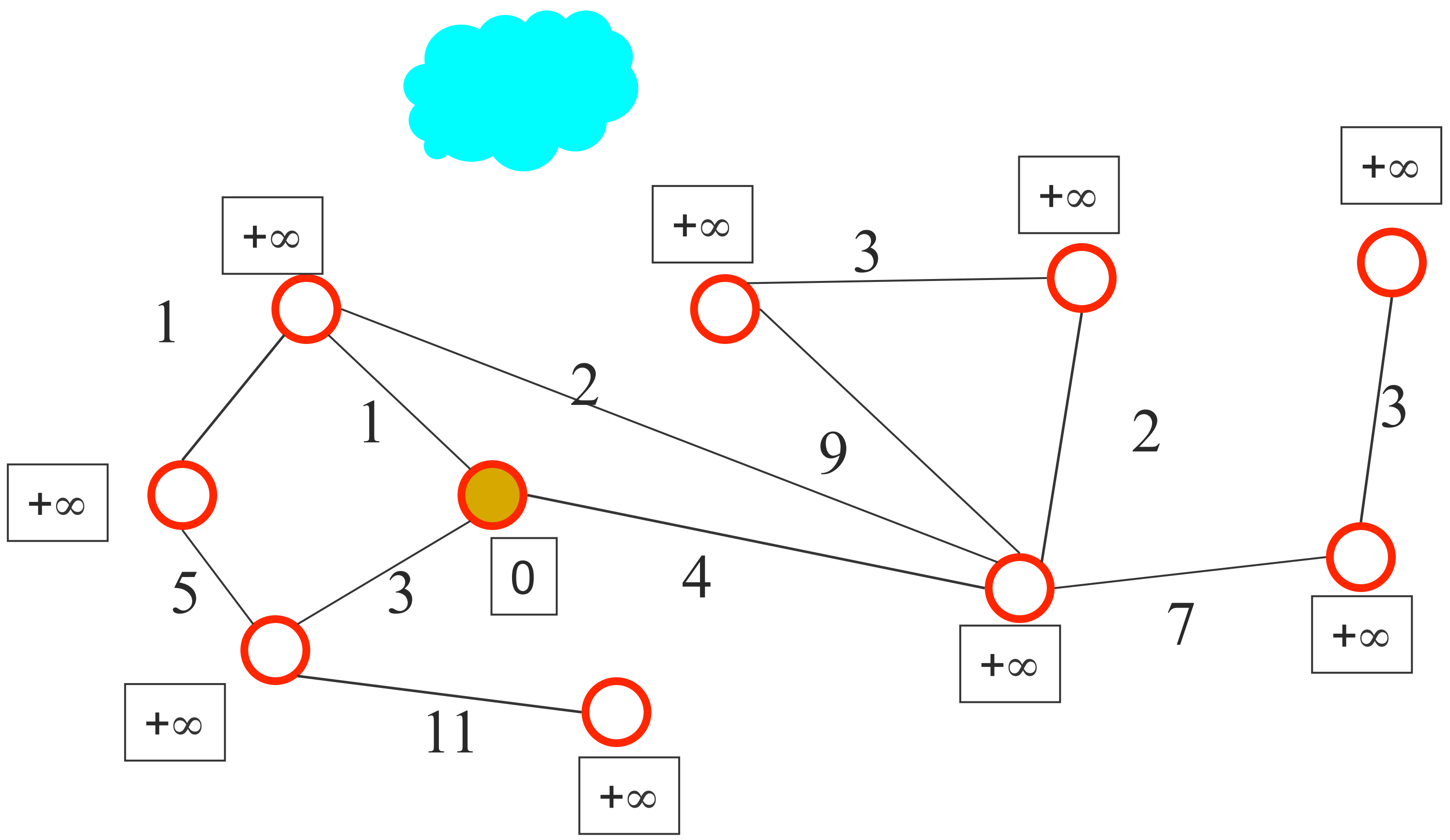
Dijkstra's algorithm: a greedy algorithm



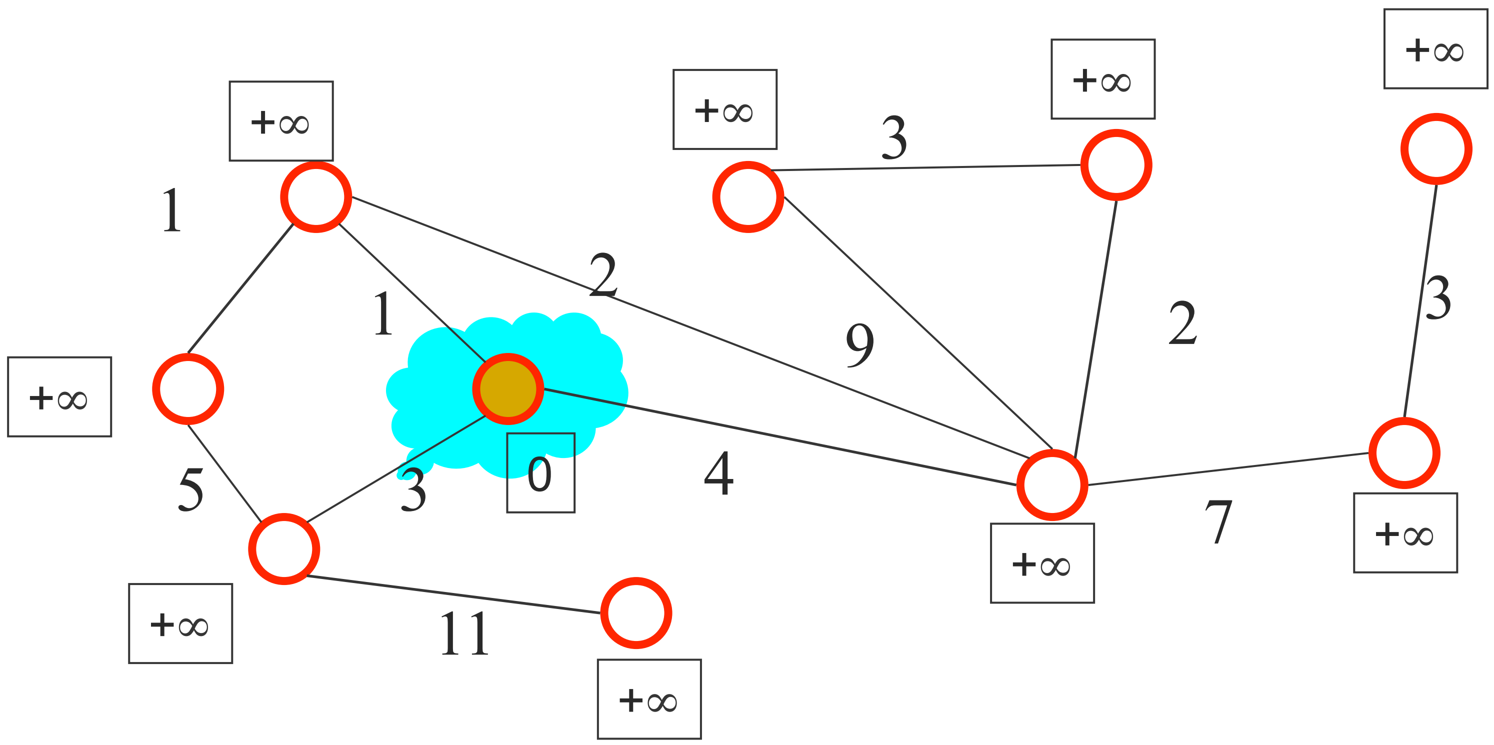
Dijkstra's algorithm: Initializing



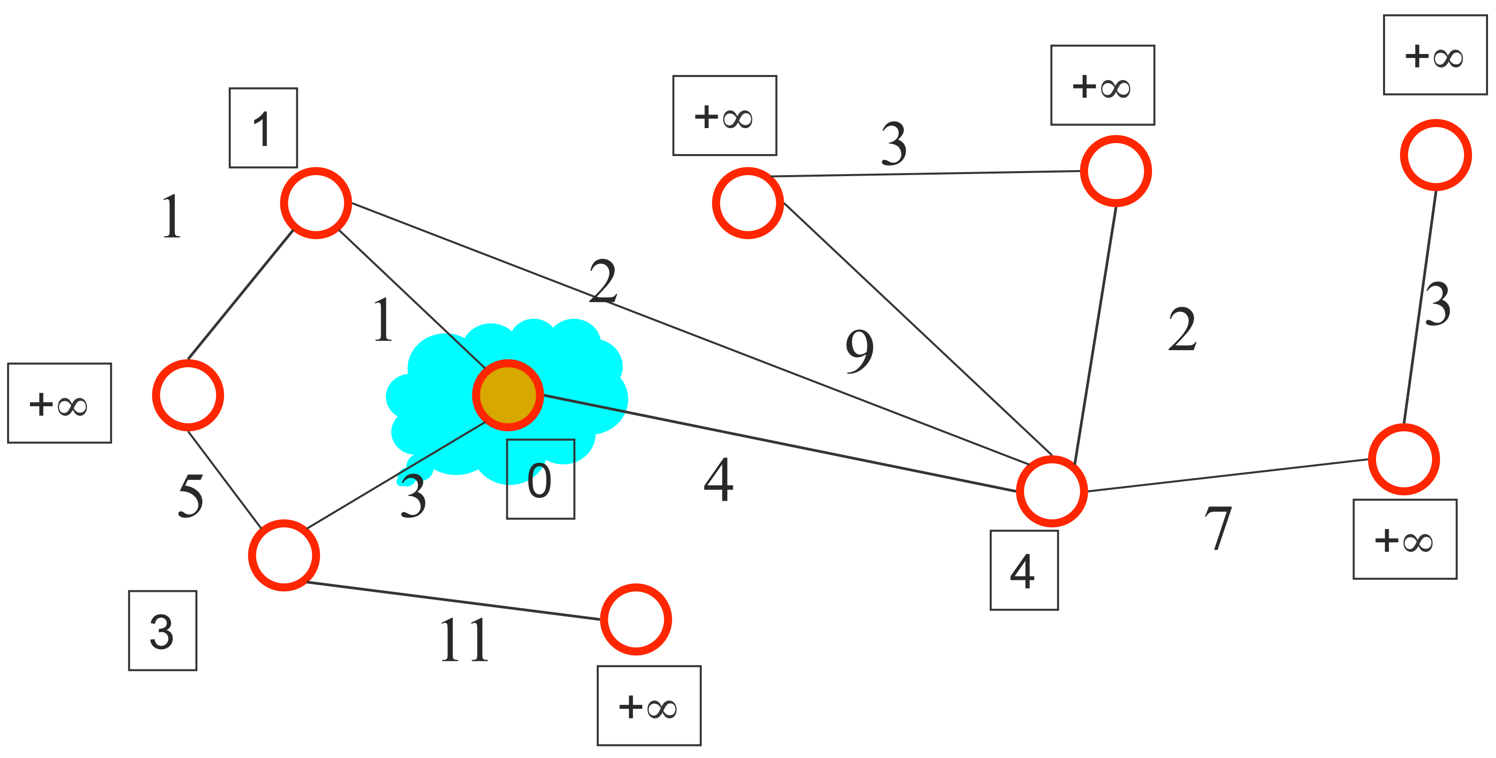
Dijkstra's algorithm: Initializing Cloud C (consisting of "solved" subgraph)



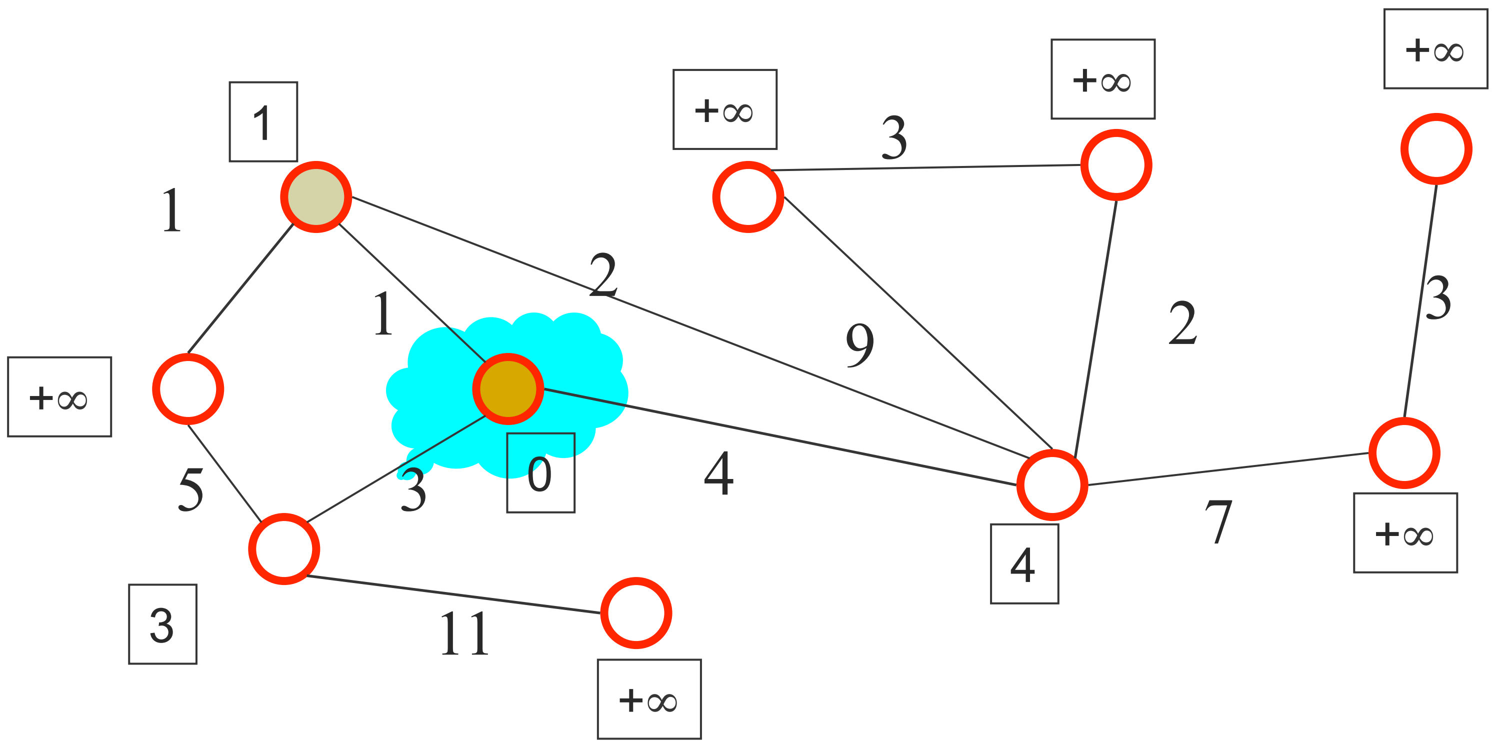
Dijkstra's algorithm: pull v into C



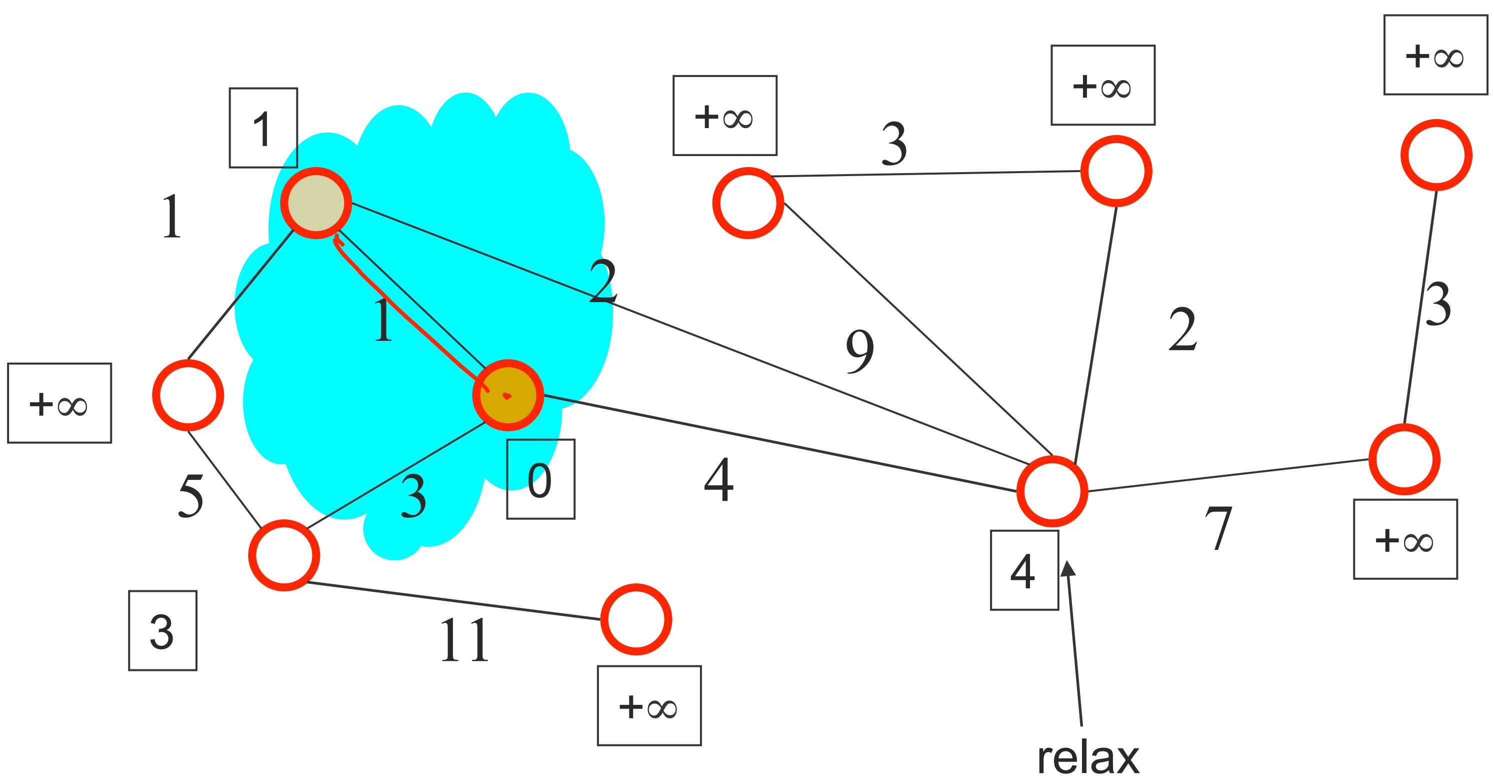
Dijkstra's algorithm: update C's neighborhood



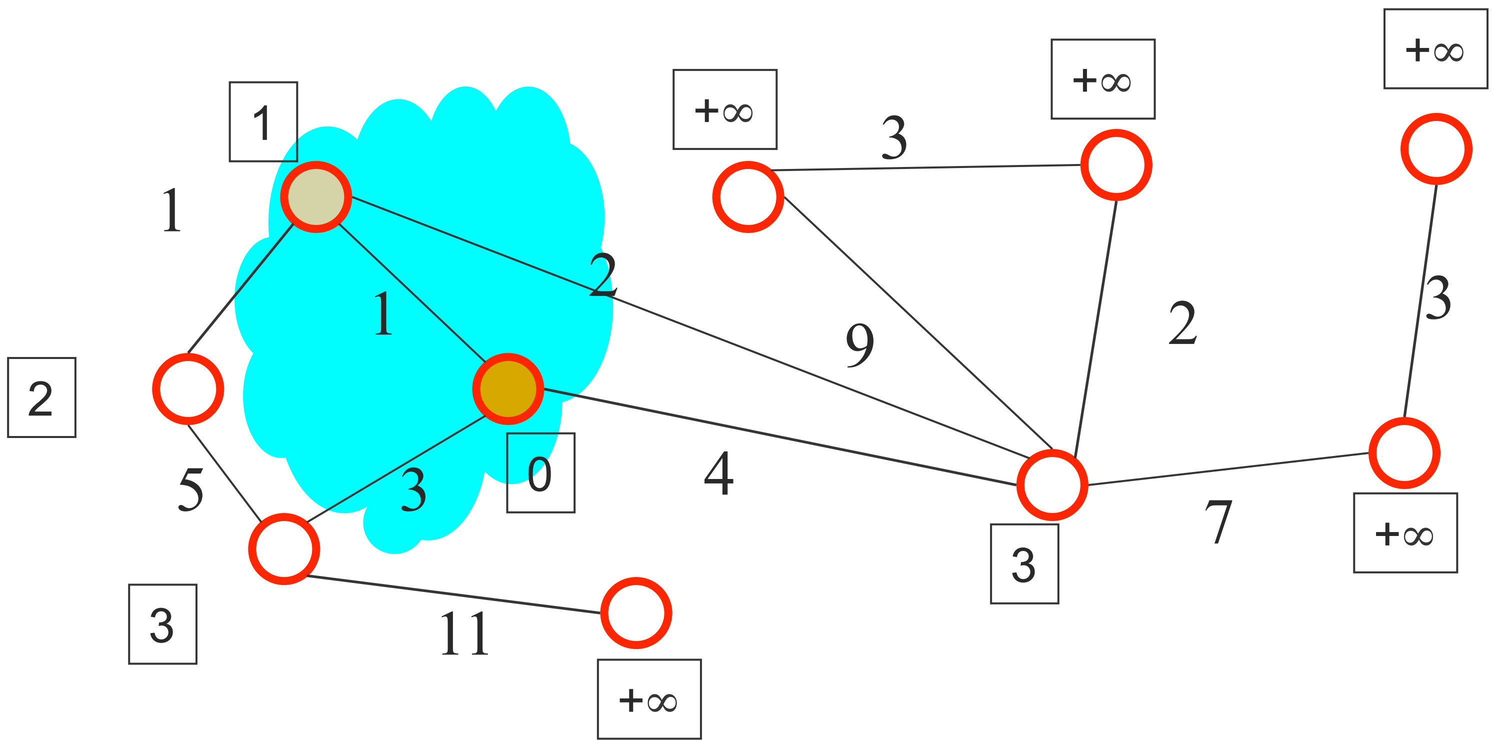
Dijkstra's algorithm: pick closest vertex u outside C



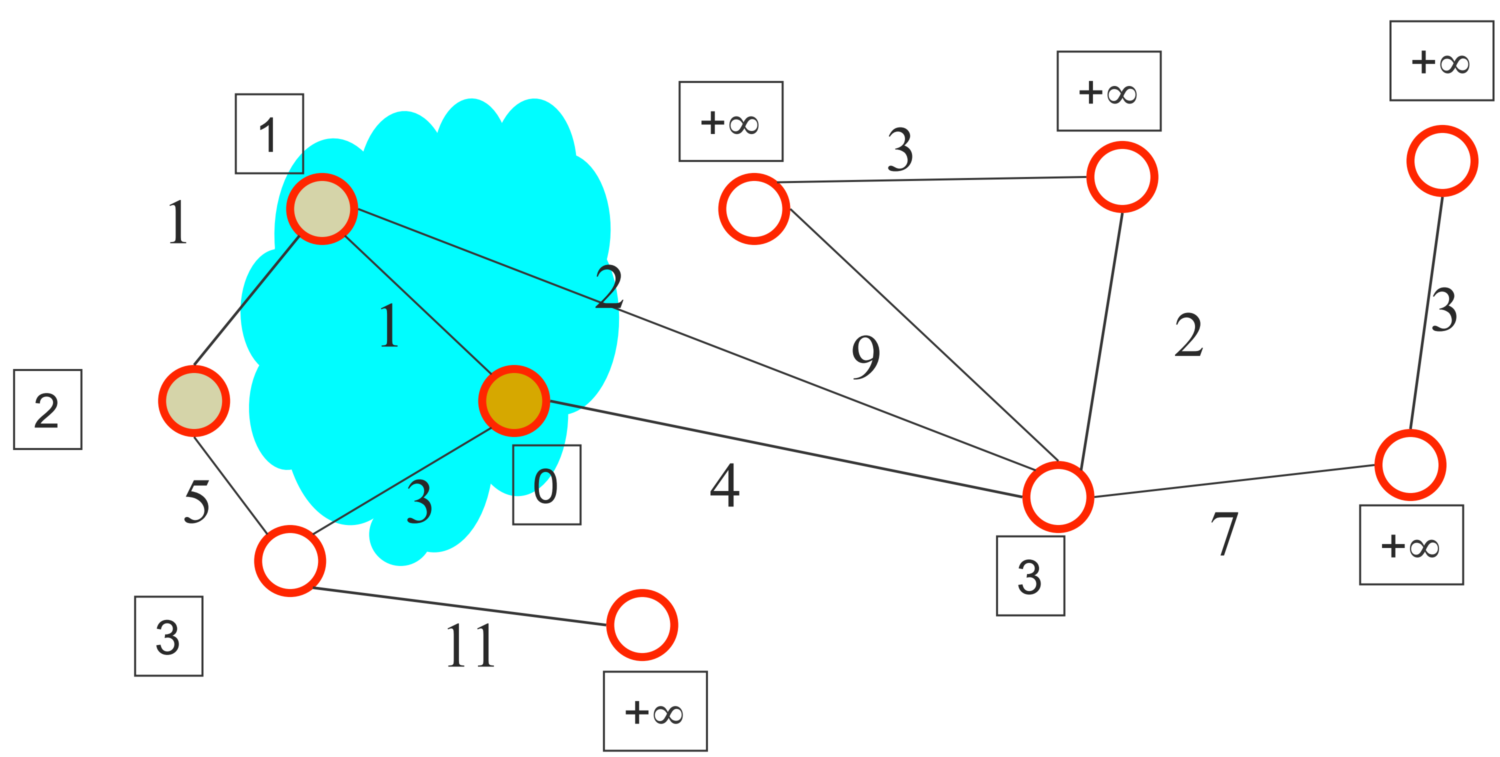
Dijkstra's algorithm: pull u into C



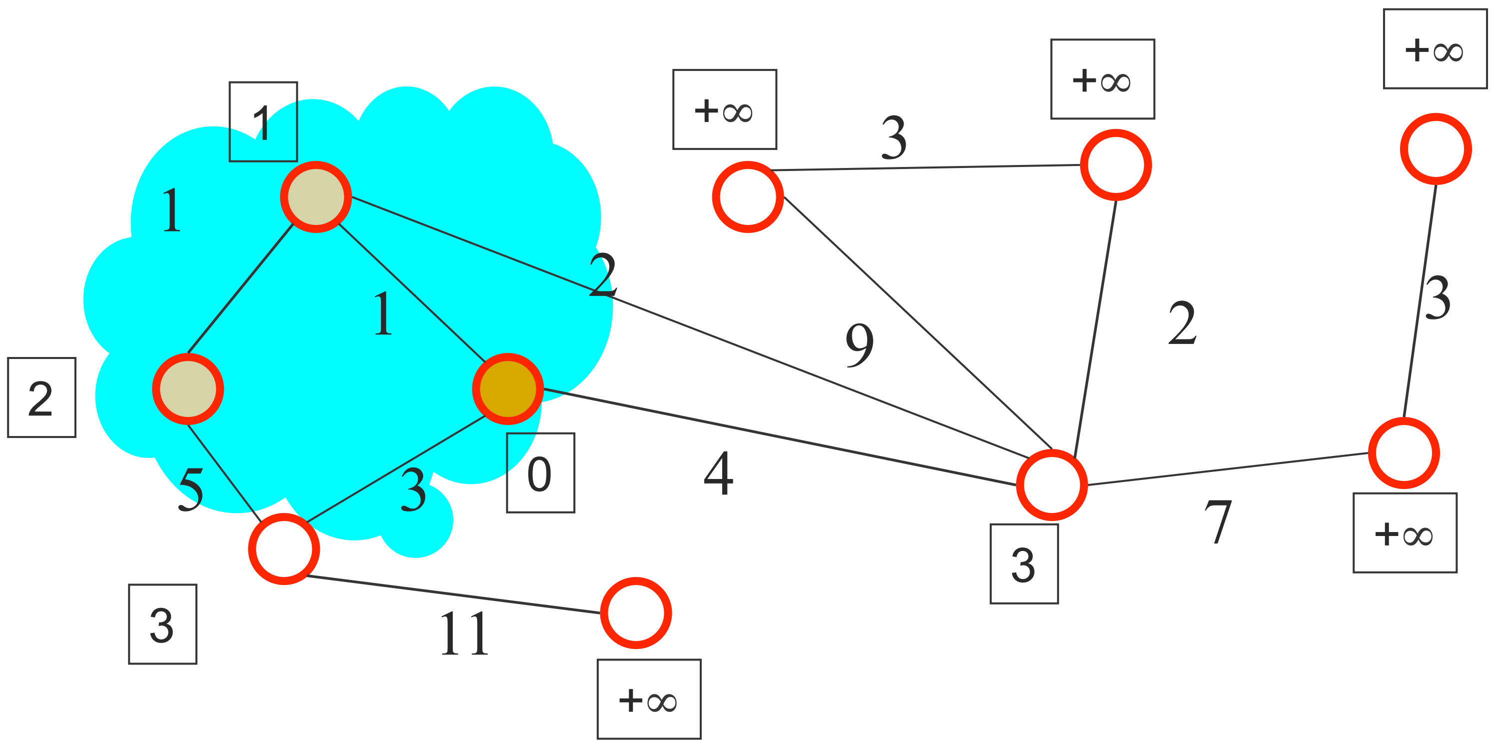
Dijkstra's algorithm: update C's neighborhood



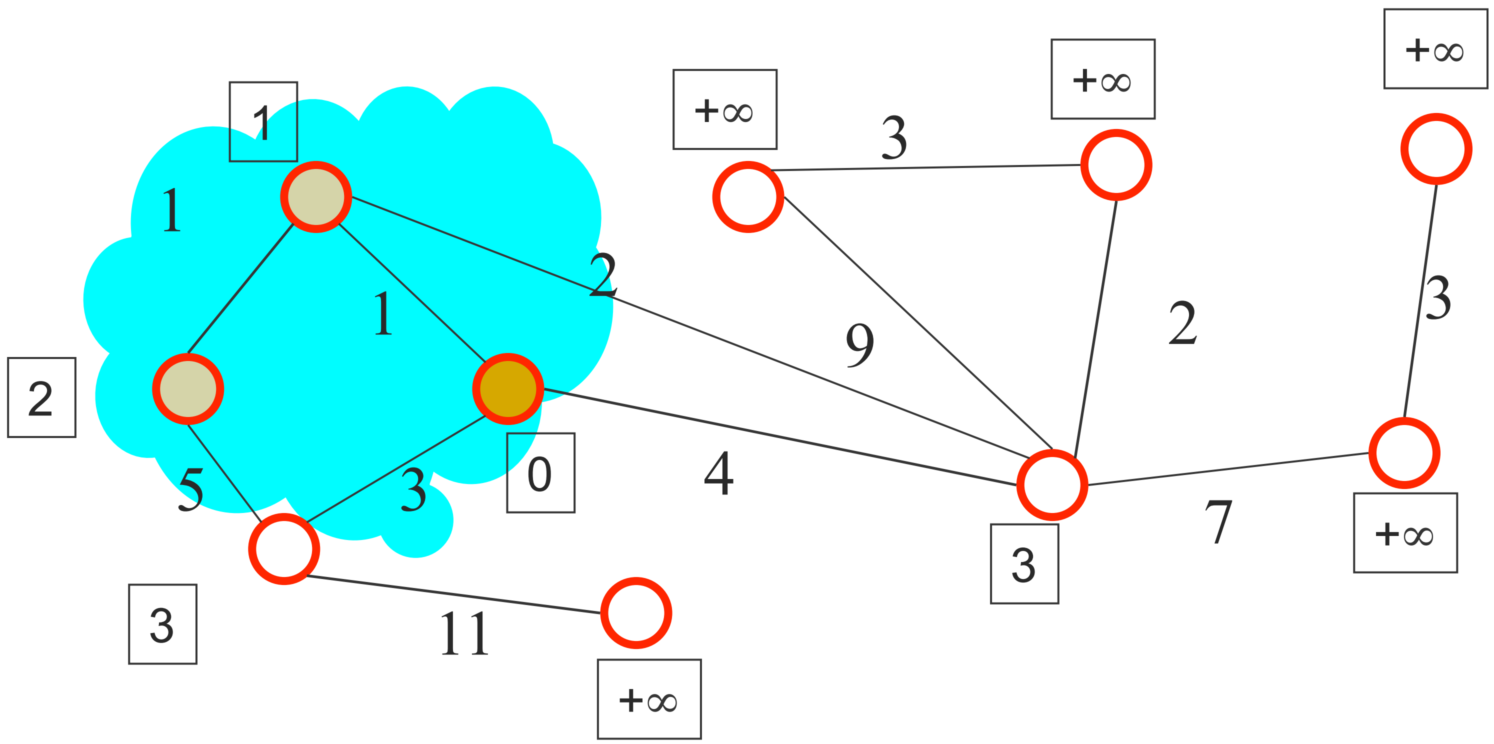
Dijkstra's algorithm: pick closest vertex u outside C



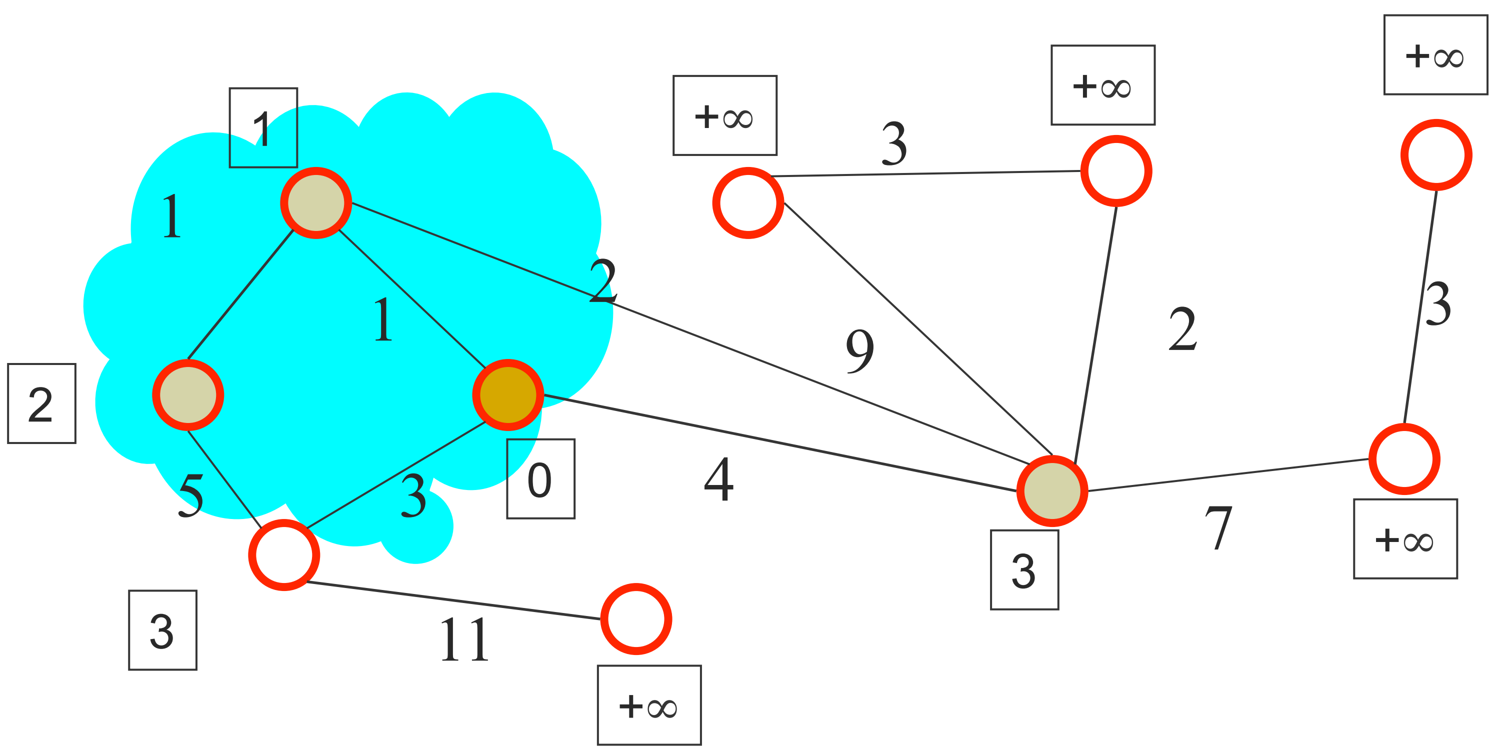
Dijkstra's algorithm: pull u into C



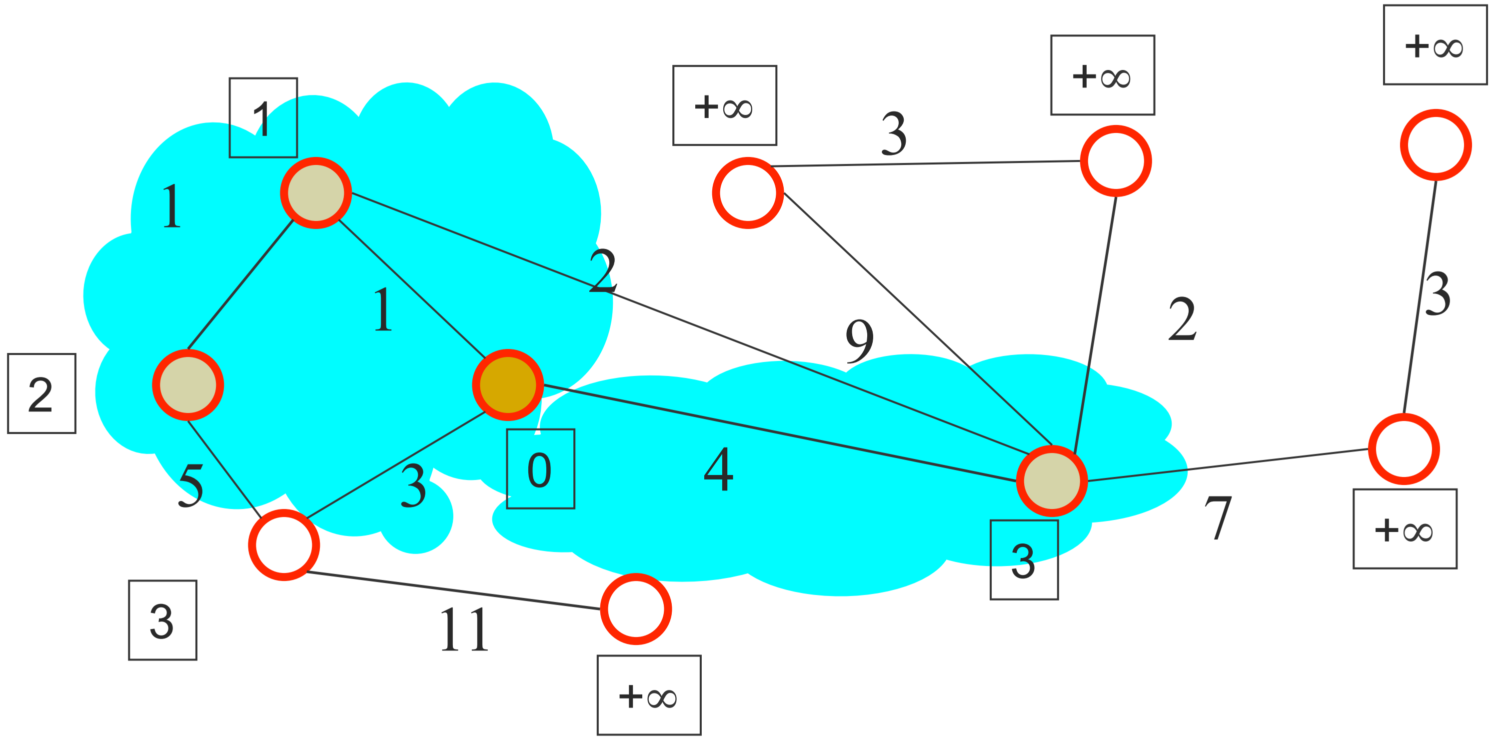
Dijkstra’s algorithm: update C’s neighborhood



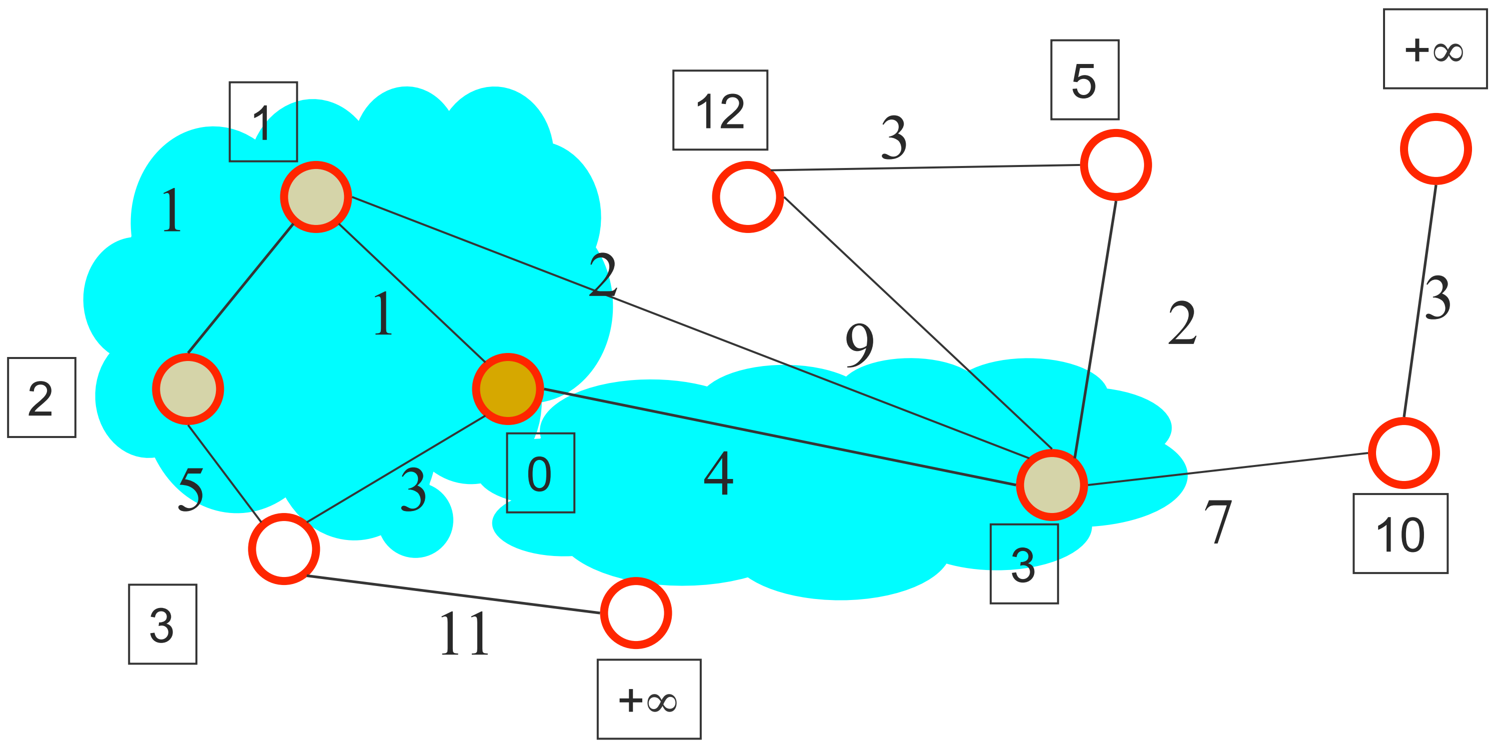
Dijkstra's algorithm: pick closest vertex u outside C



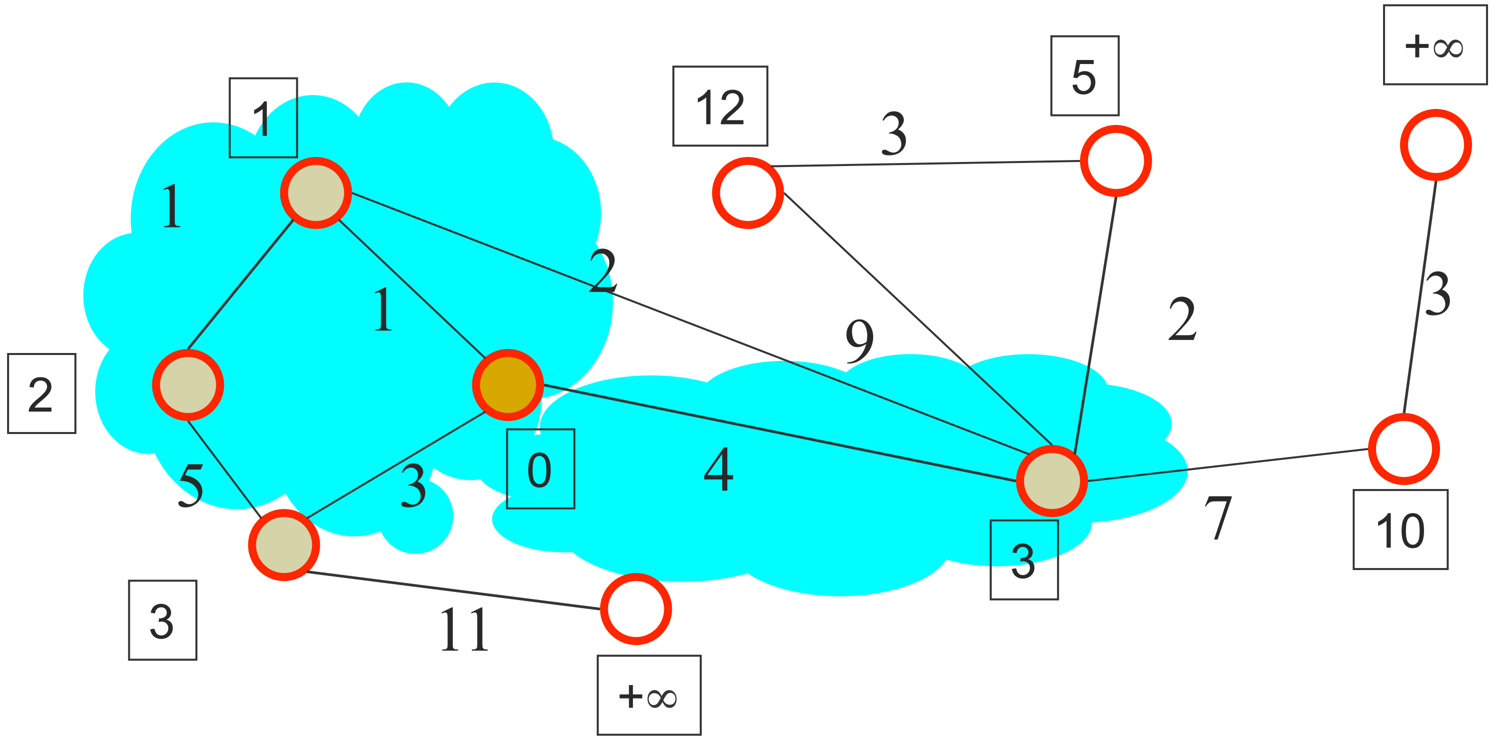
Dijkstra’s algorithm: pull u into C



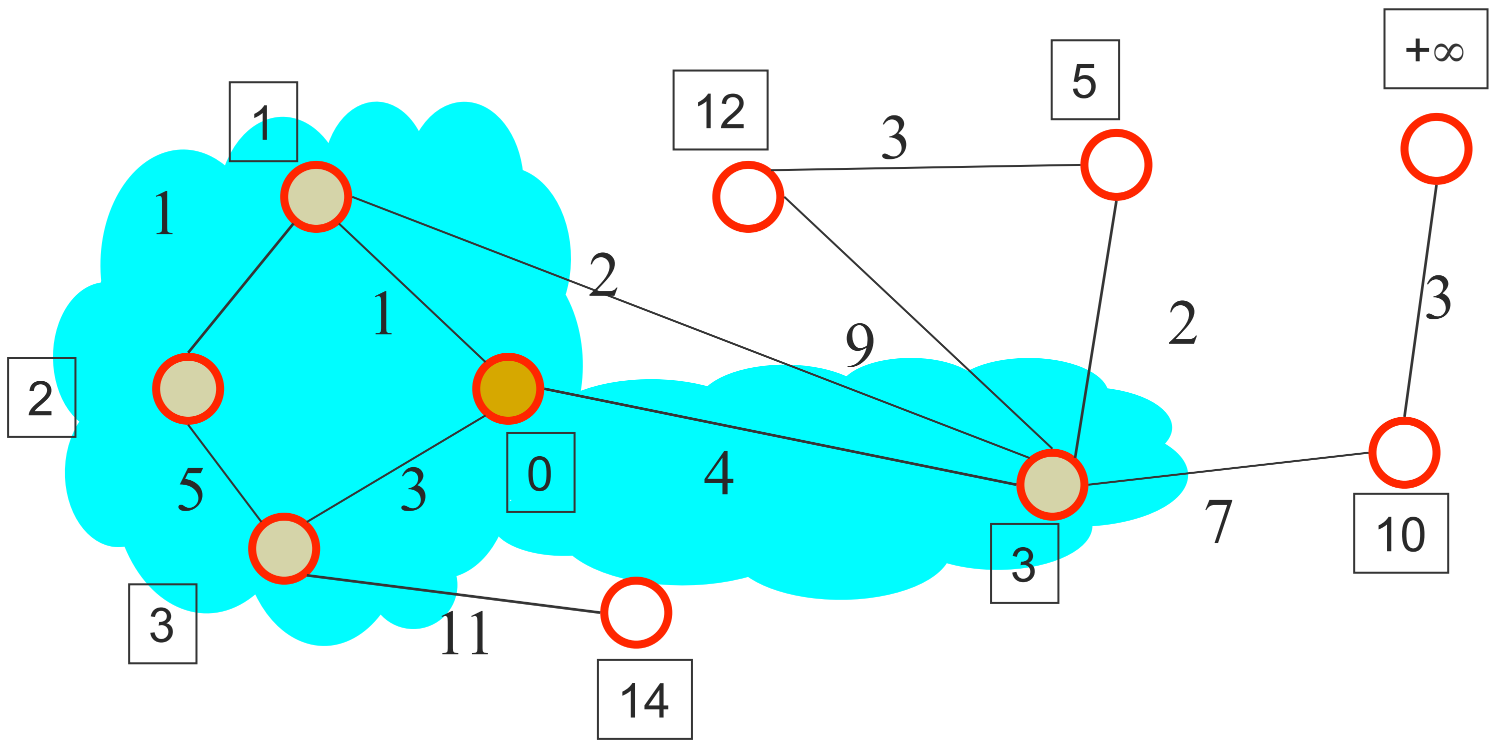
Dijkstra's algorithm: update C's neighborhood



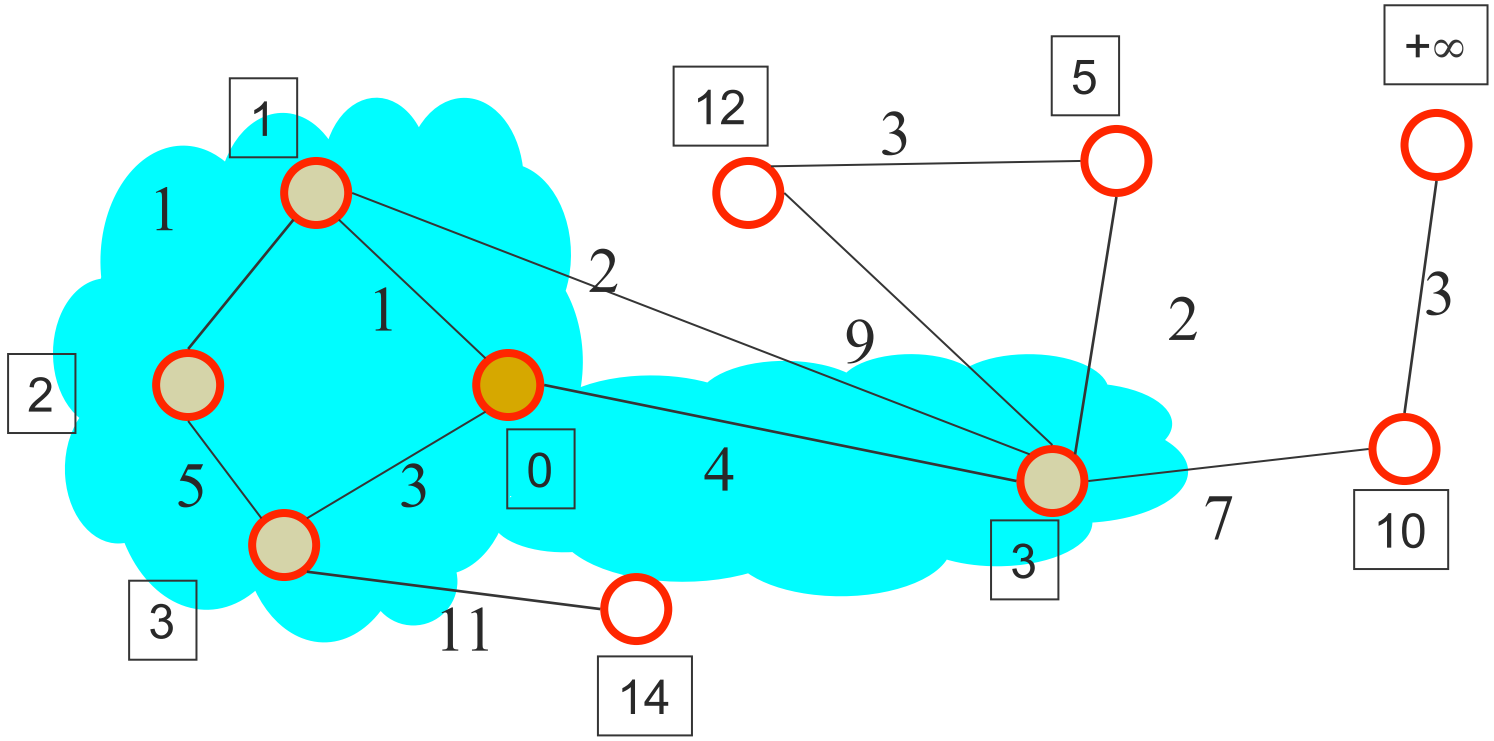
Dijkstra's algorithm: pick closest vertex u outside C



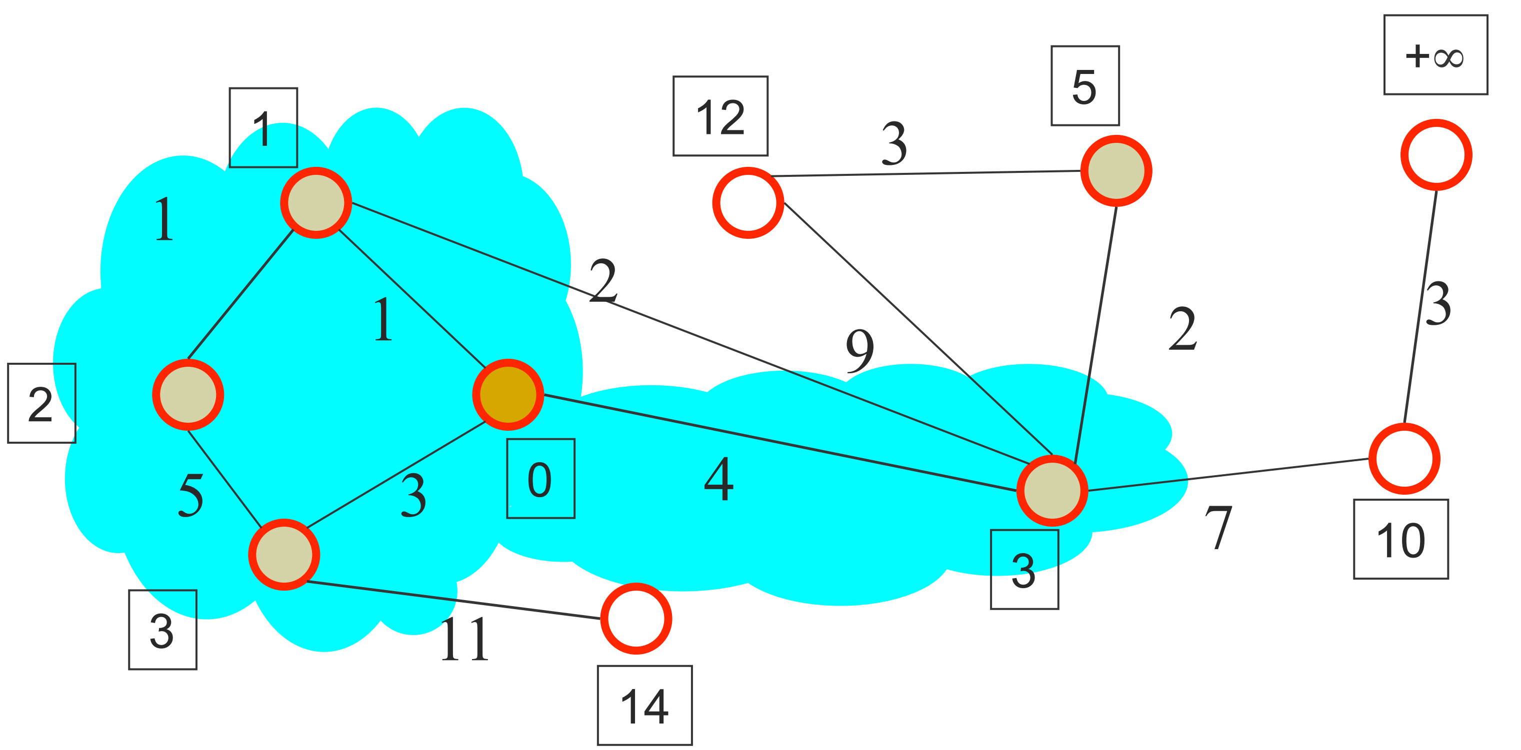
Dijkstra's algorithm: pull u into C



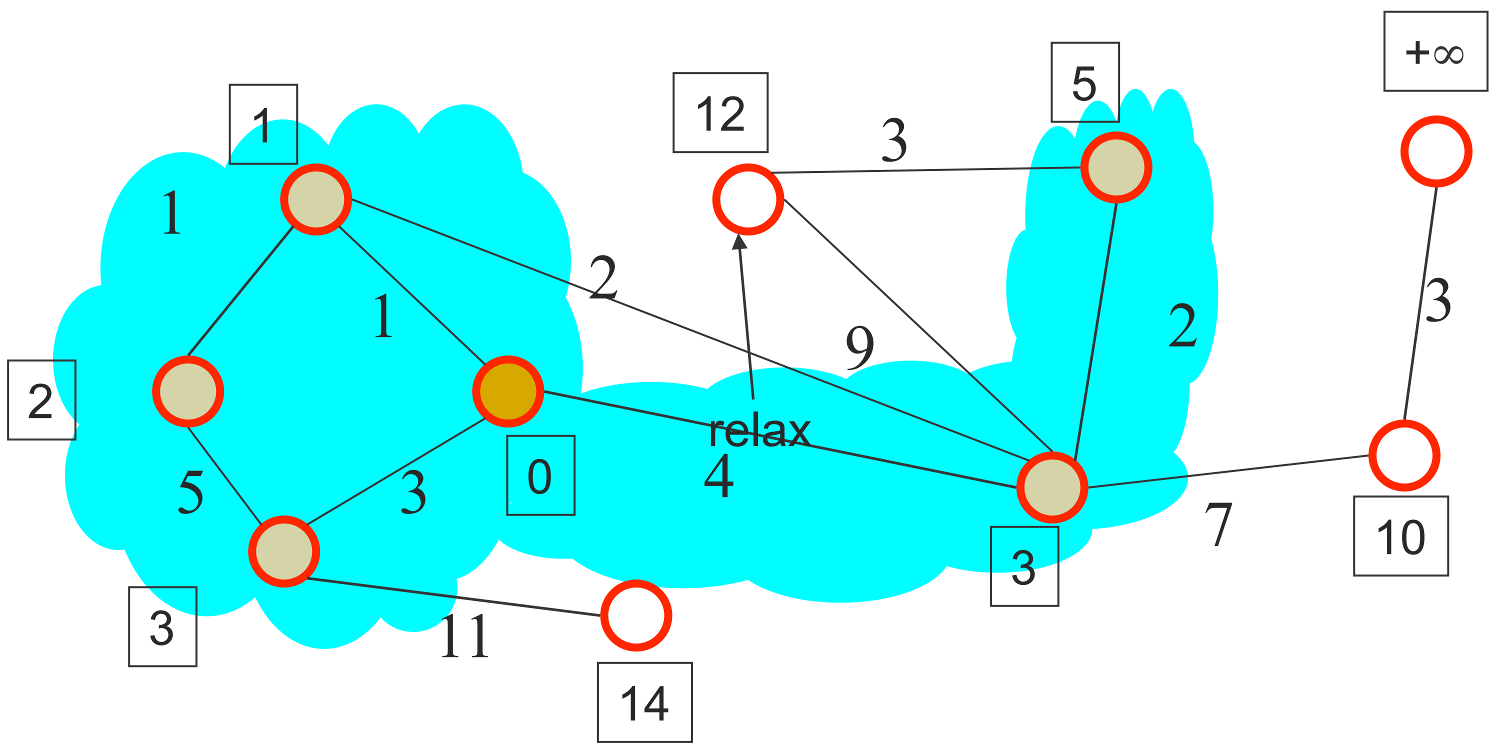
Dijkstra's algorithm: update C's neighborhood



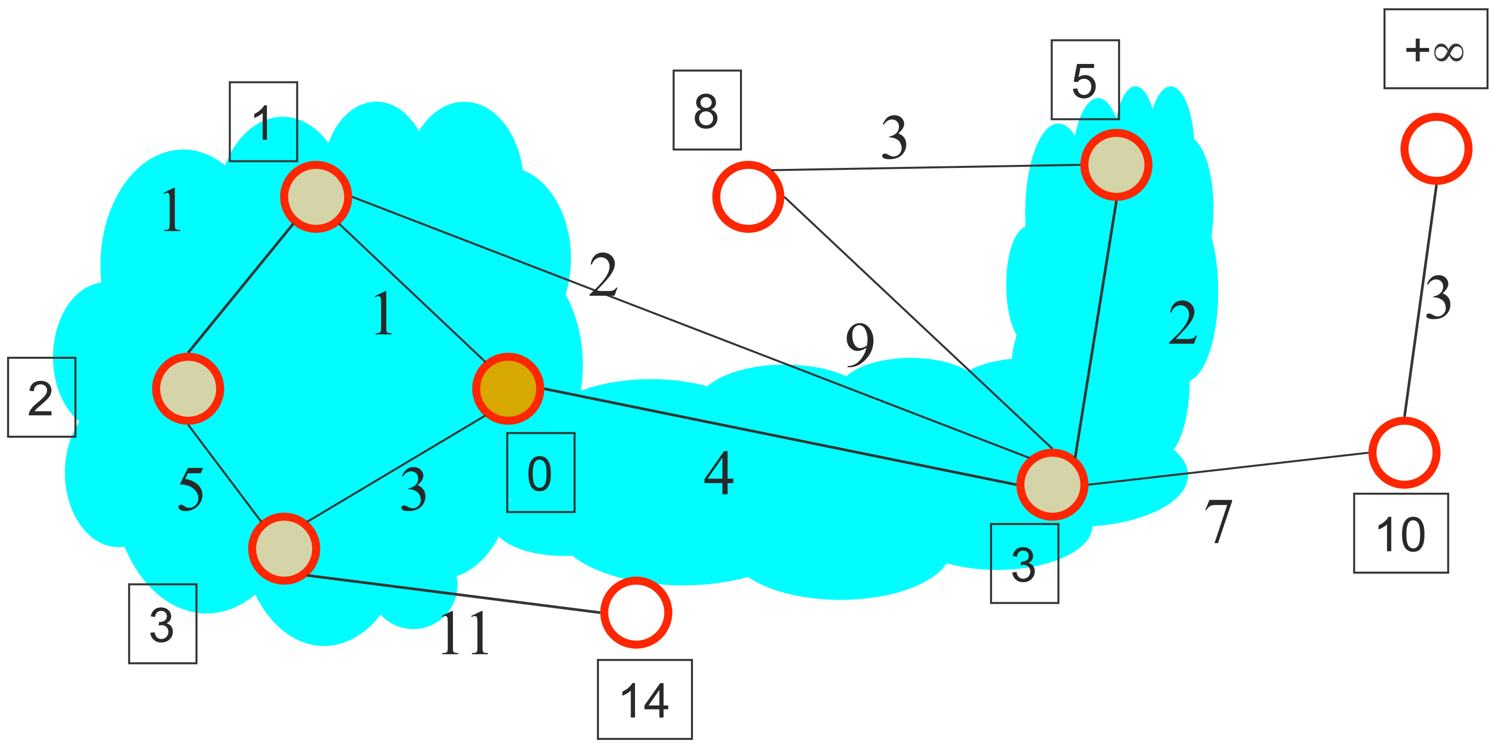
Dijkstra's algorithm: pick closest vertex u outside C



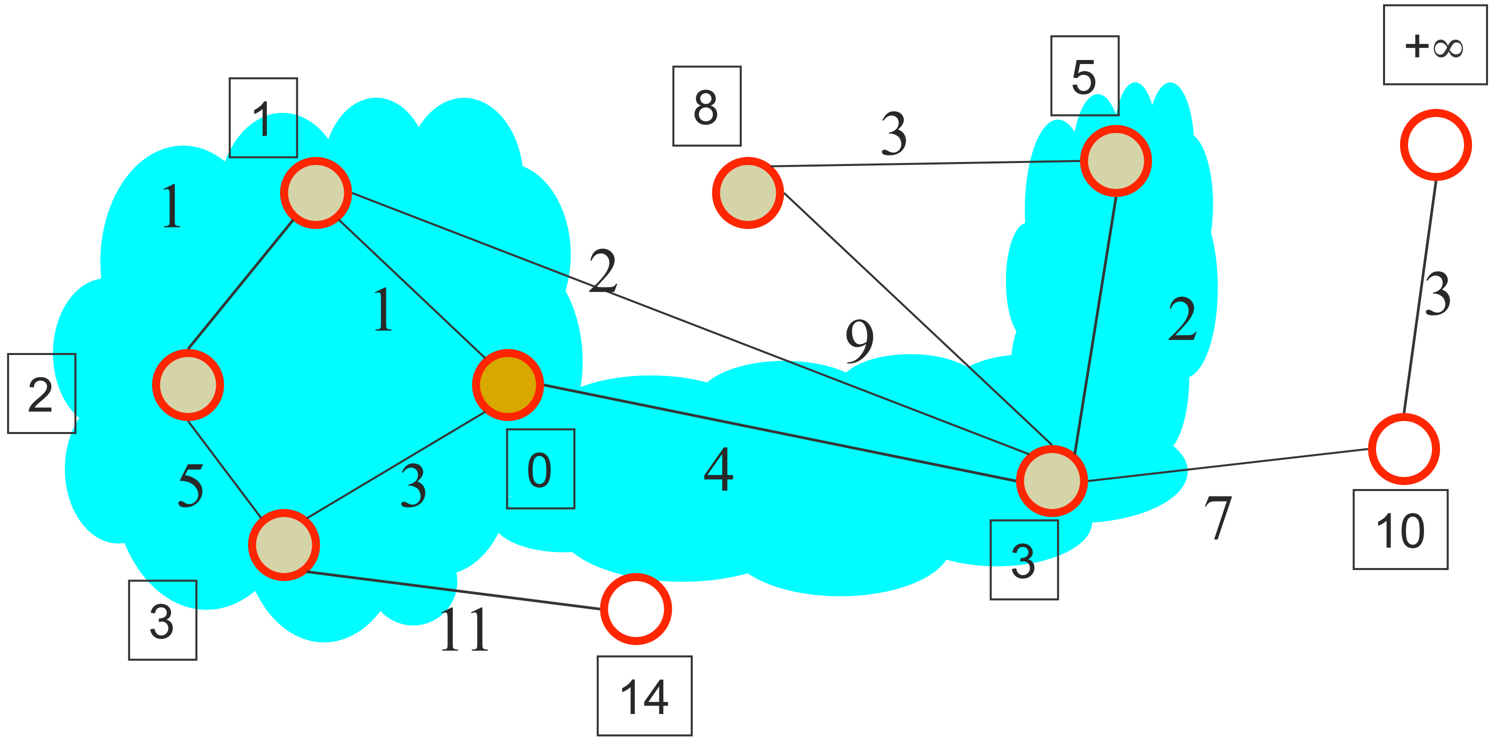
Dijkstra's algorithm: pull u into C



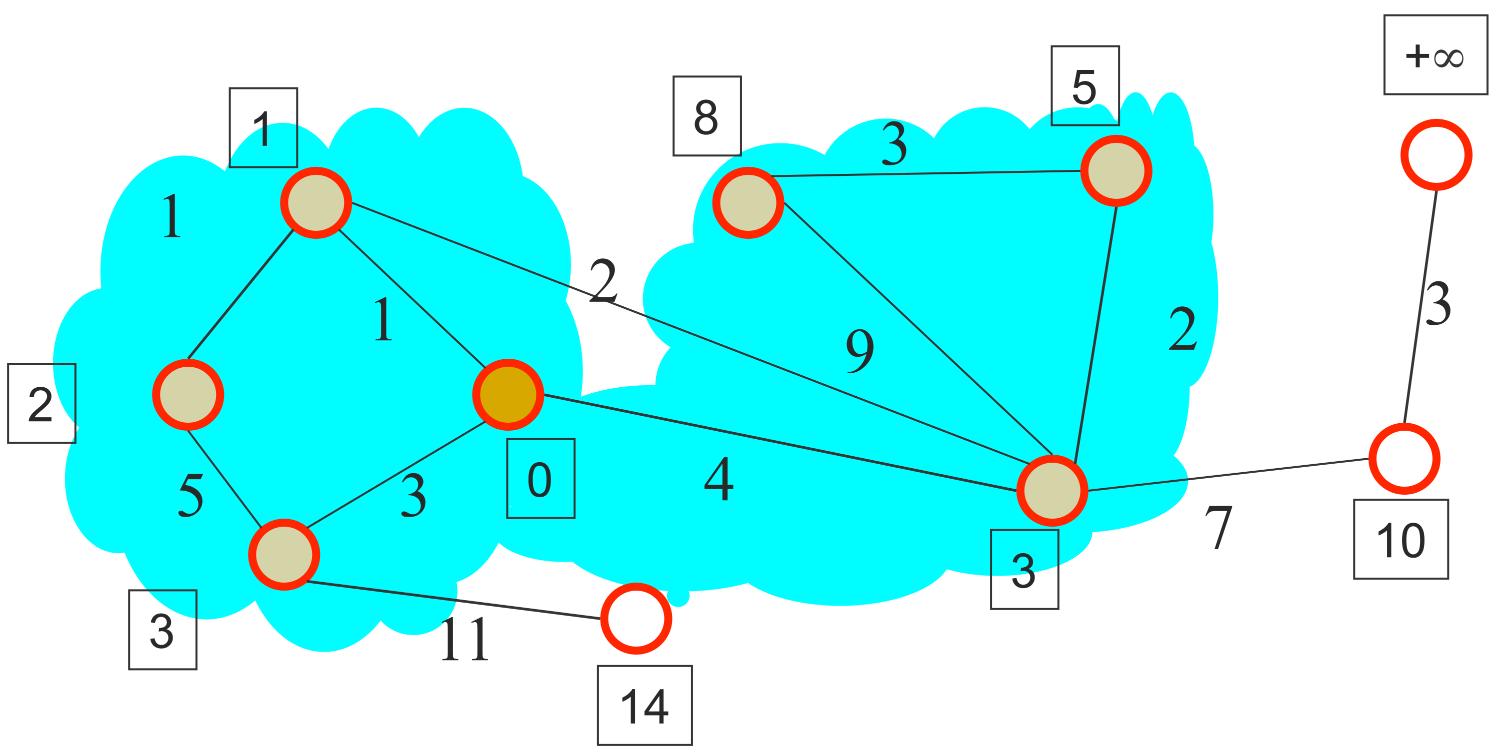
Dijkstra's algorithm: update C's neighborhood



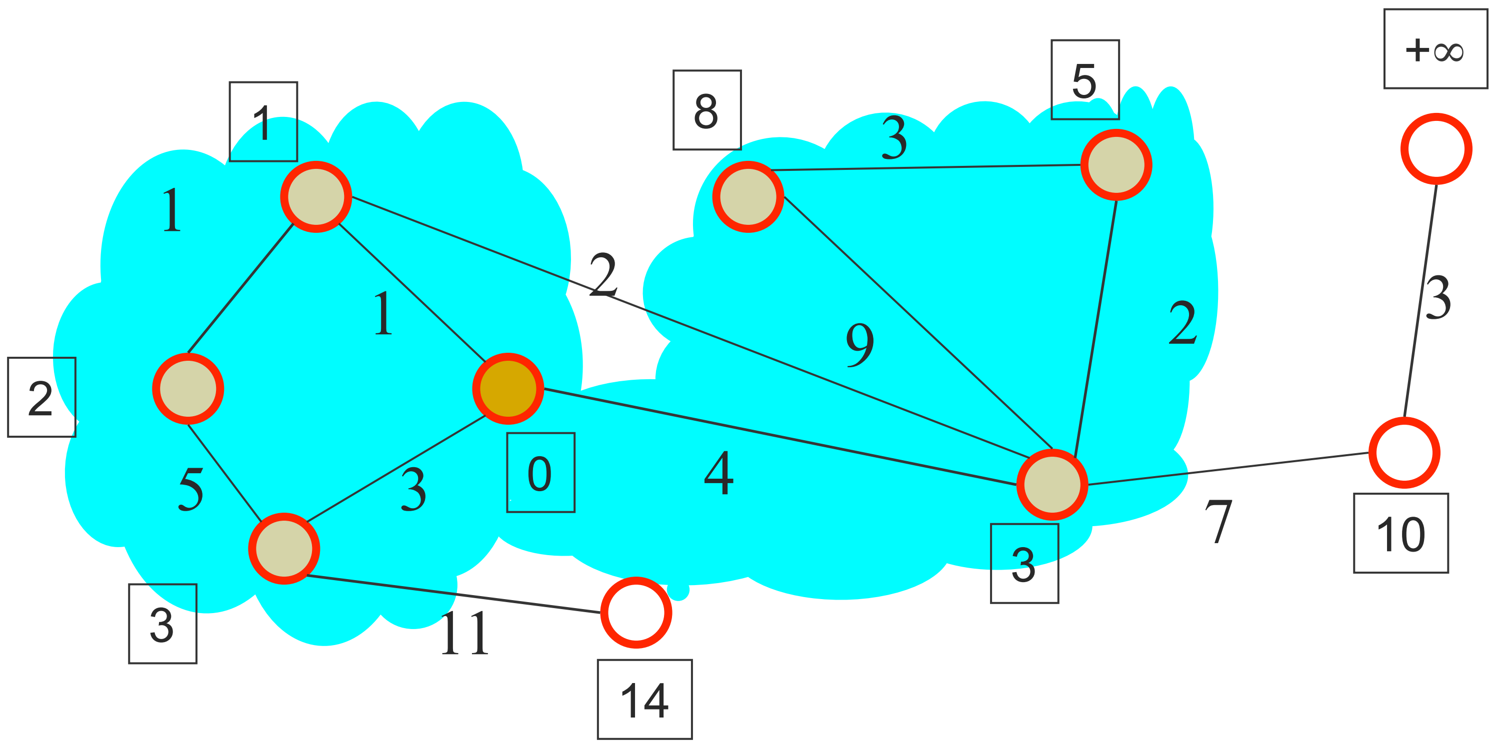
Dijkstra's algorithm: pick closest vertex u outside C



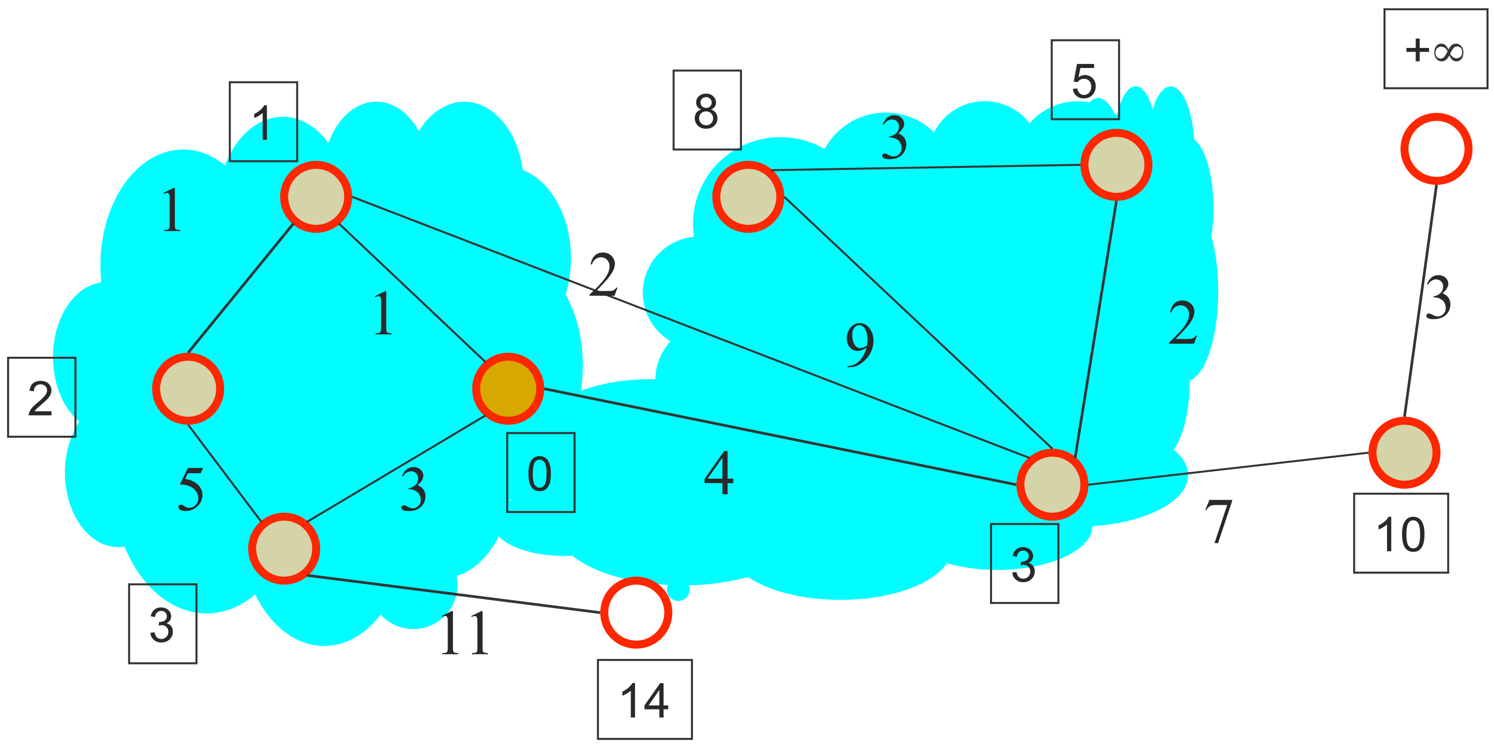
Dijkstra's algorithm: pull u into C



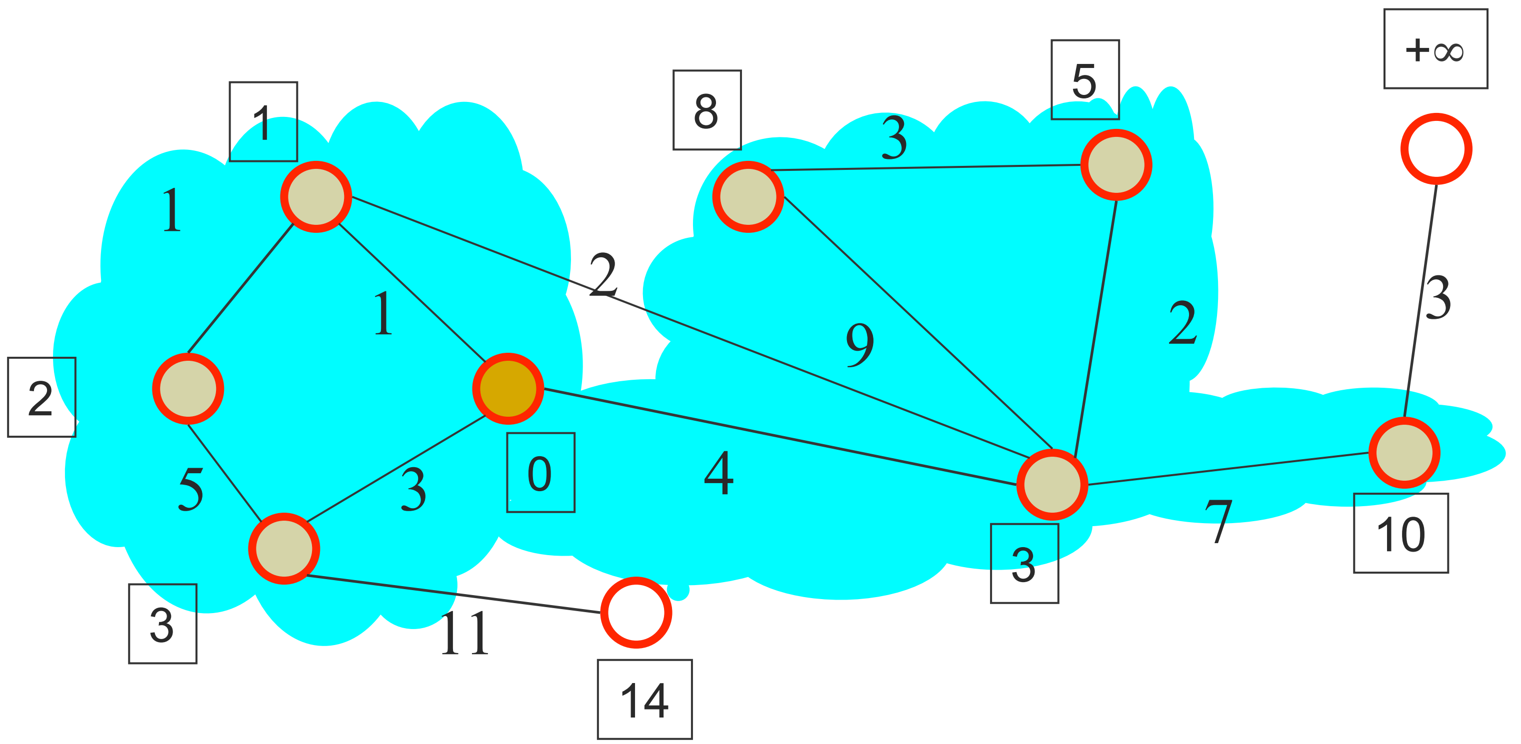
Dijkstra's algorithm: update C's neighborhood



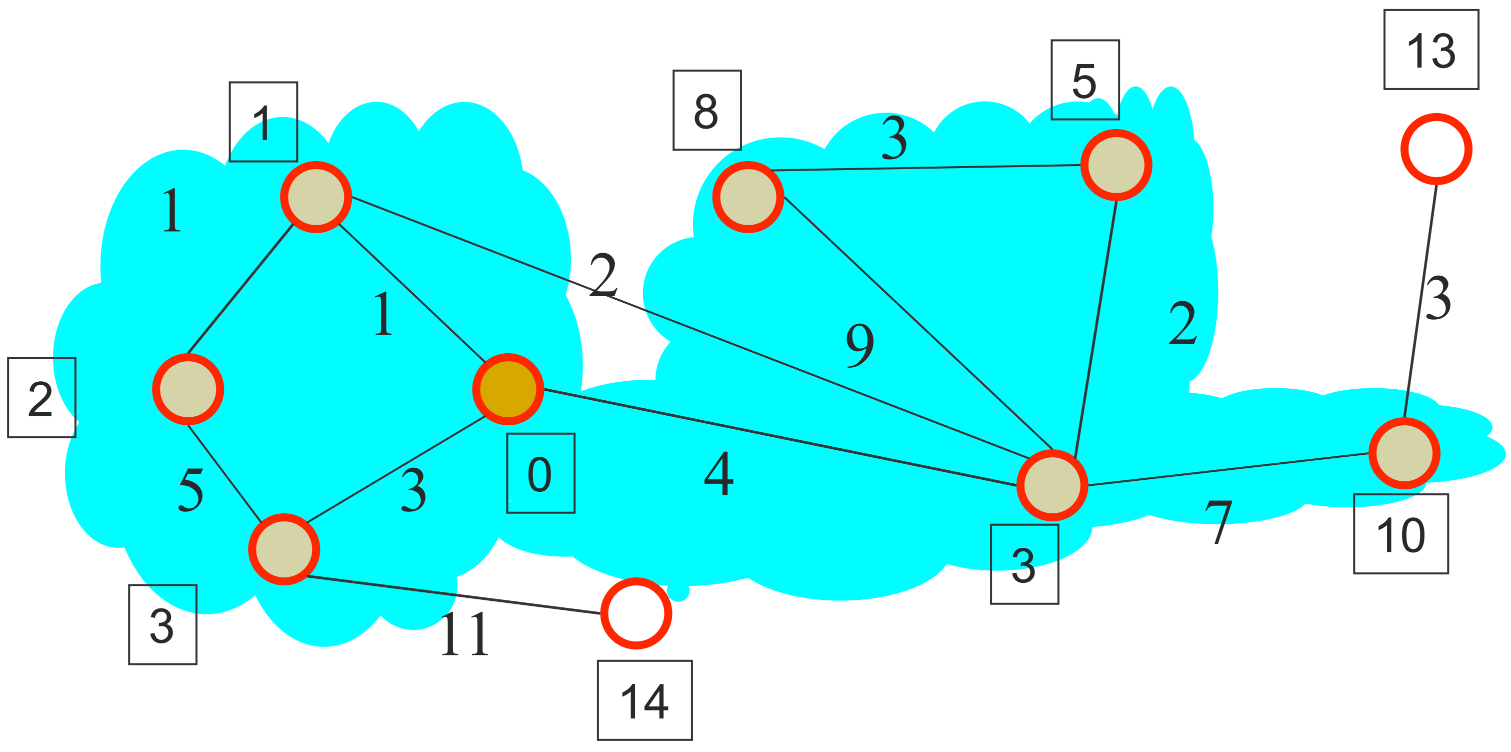
Dijkstra's algorithm: pick closest vertex u outside C



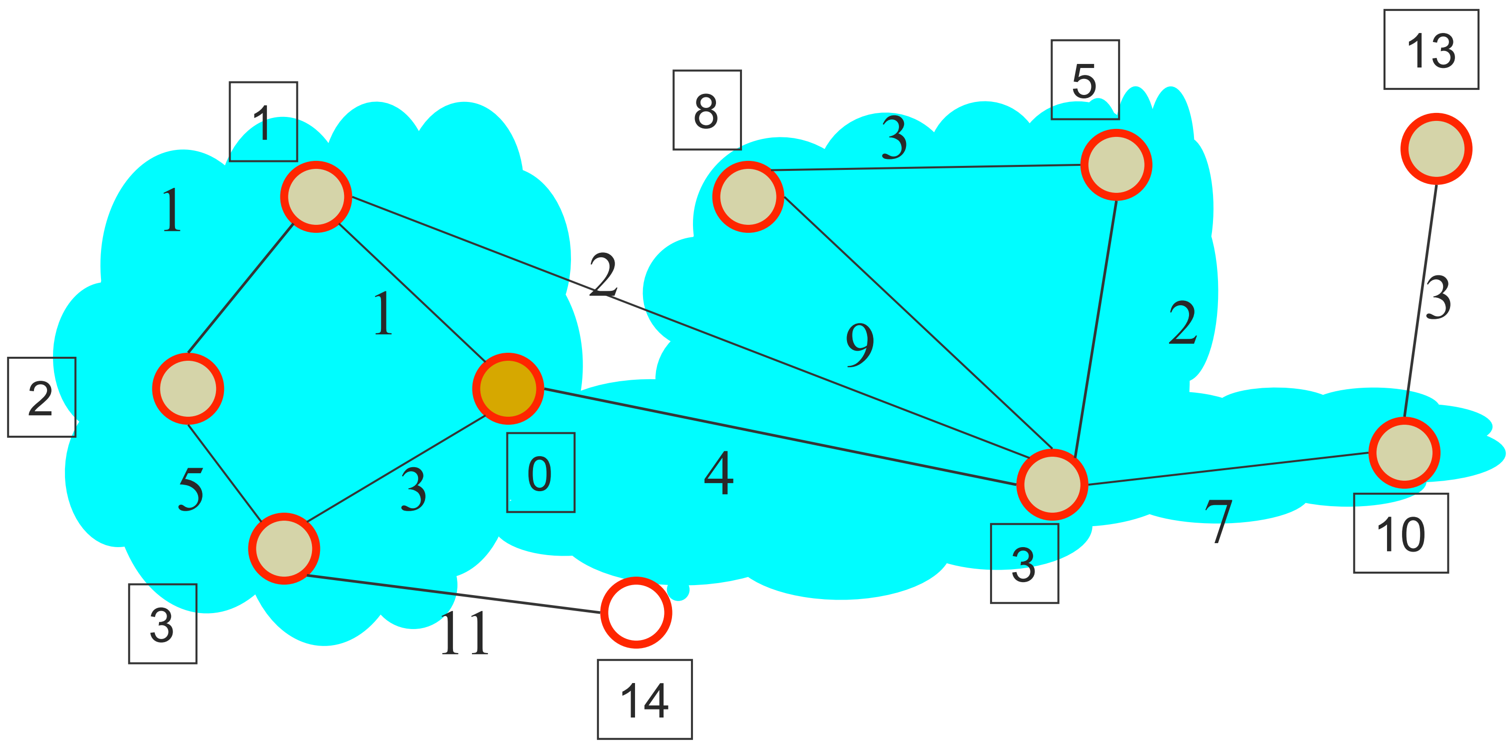
Dijkstra's algorithm: pull u into C



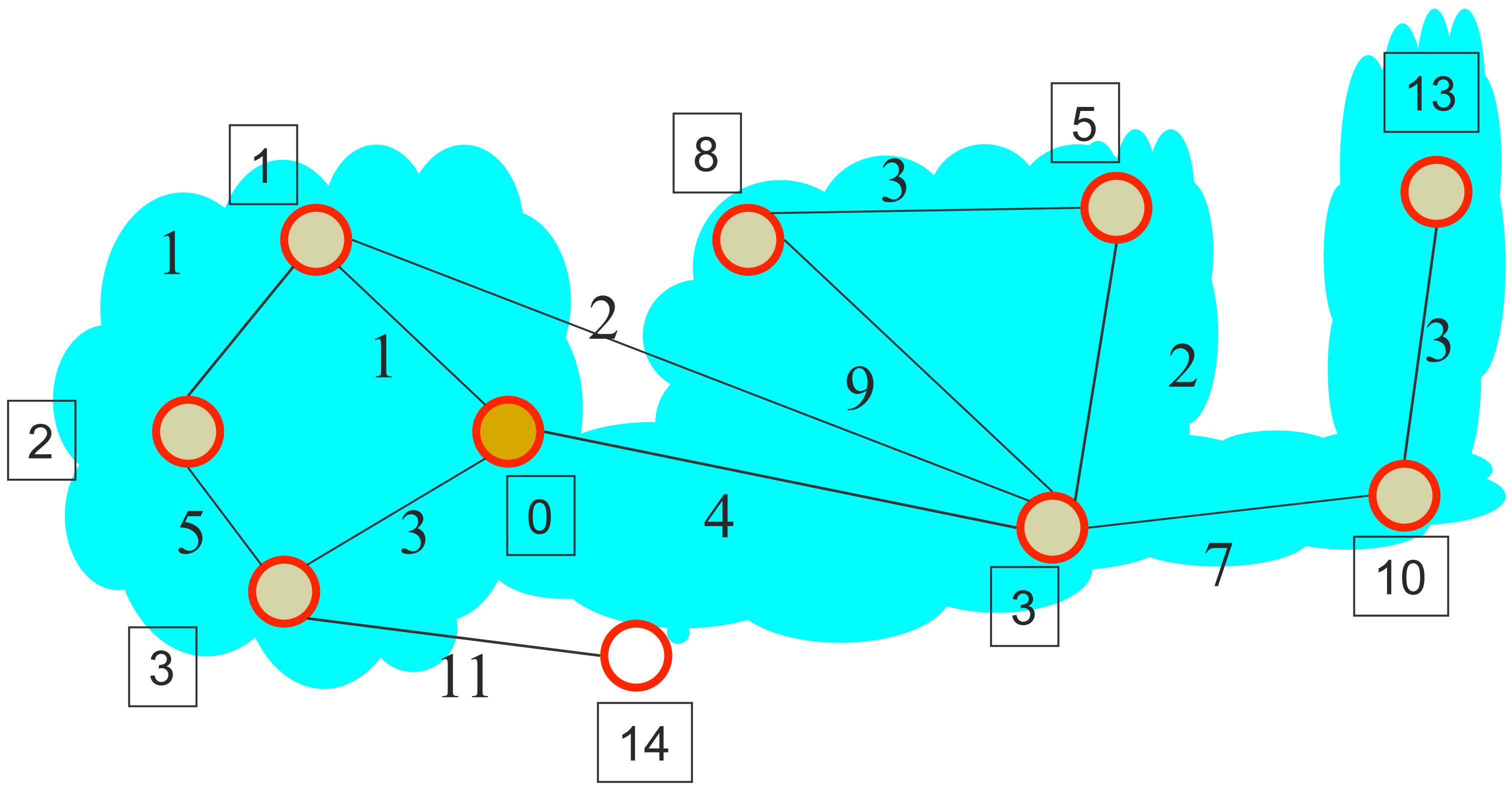
Dijkstra's algorithm: update C's neighborhood



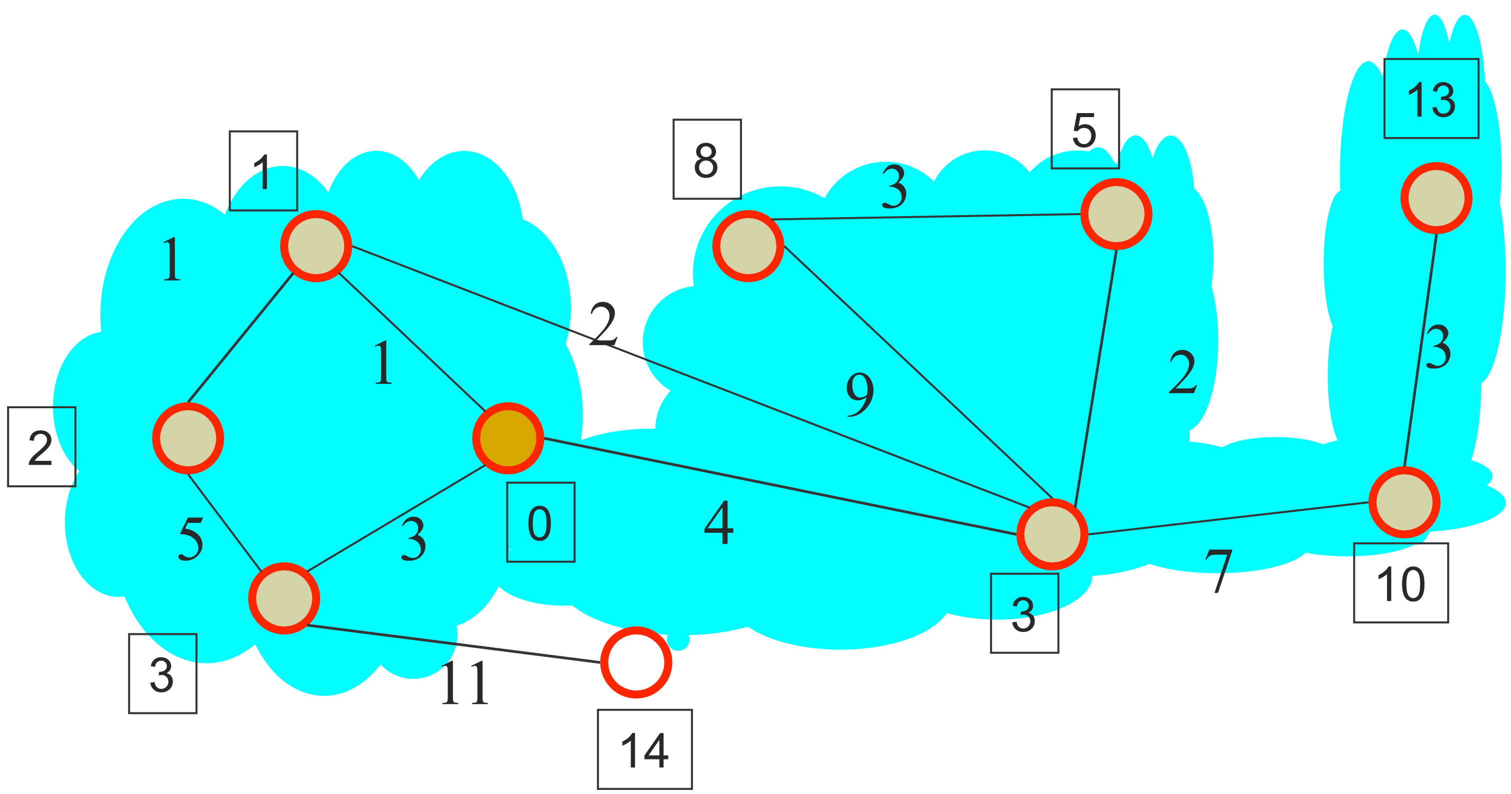
Dijkstra's algorithm: pick closest vertex u outside C



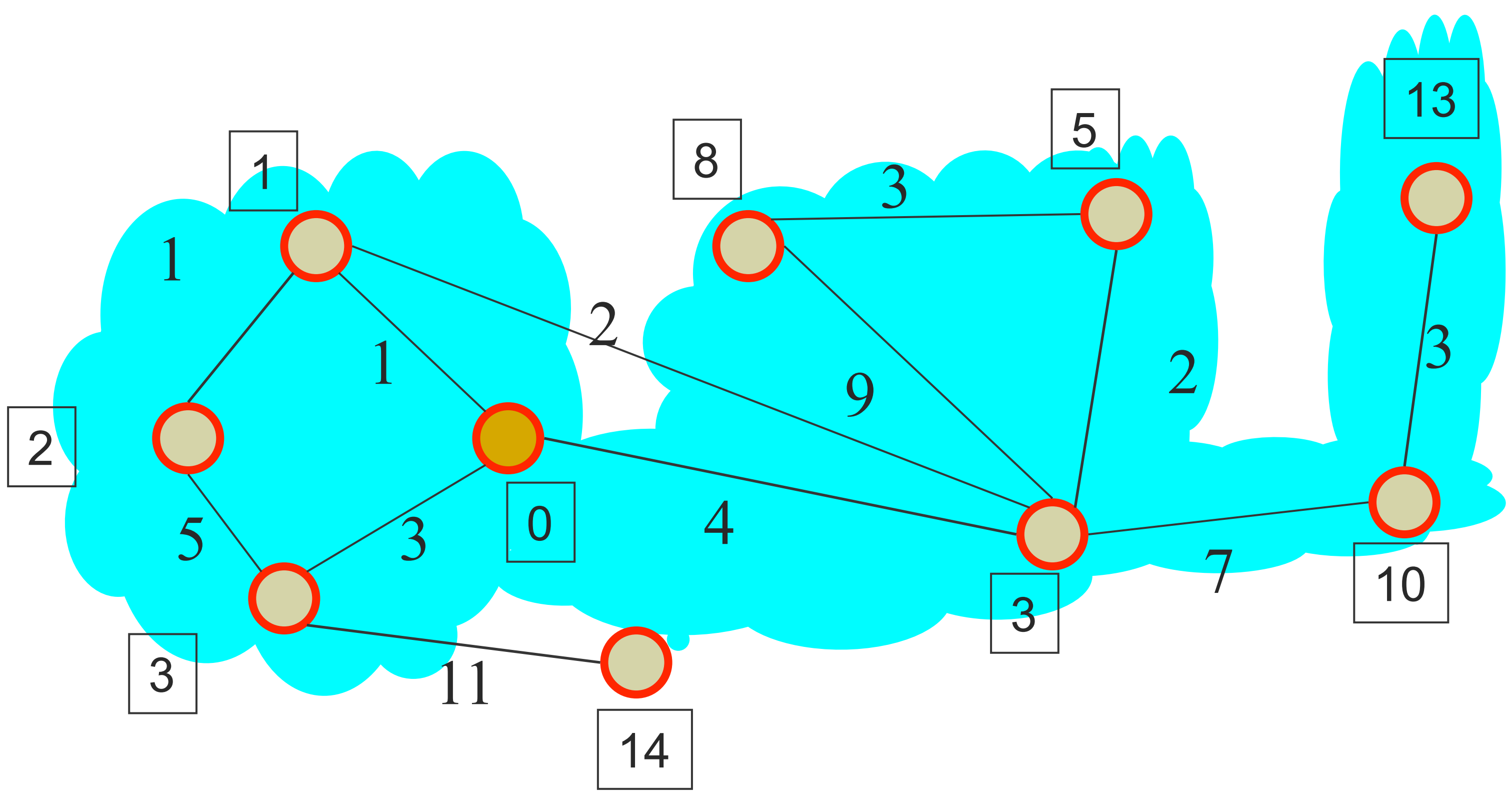
Dijkstra's algorithm: pull u into C



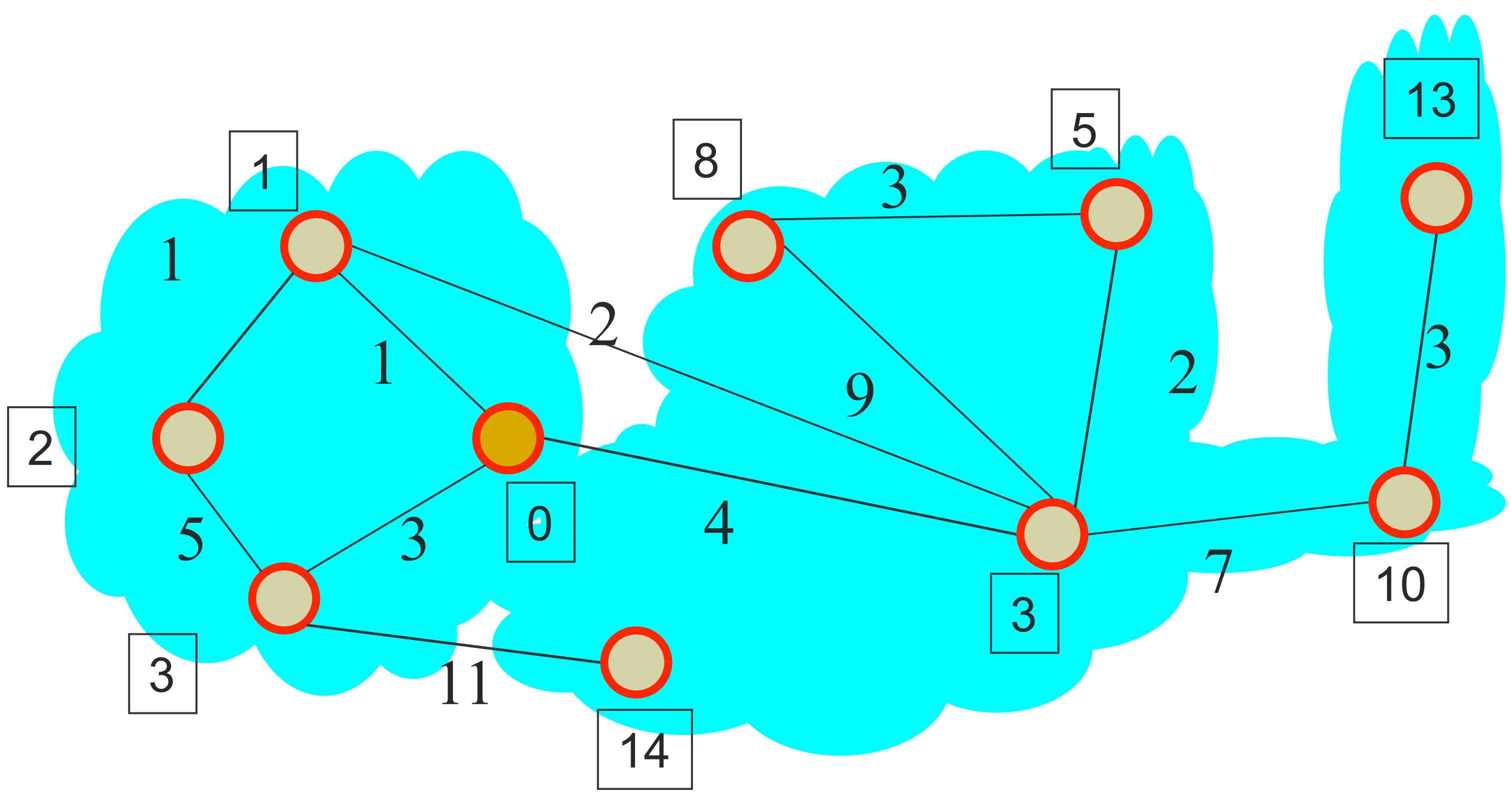
Dijkstra's algorithm: update C's neighborhood



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

$d[v] \leftarrow \infty$; $\pi[v] \leftarrow$ null;

$d[s] \leftarrow 0$

$S \leftarrow \emptyset$

Q = BuildPriorityQueue(V, d)

While Q not empty

$u \leftarrow$ DeleteMin(Q)

$S \leftarrow S \cup u$

For v in Adj[u]

Relax(u,v)

RELAX(u, v)

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

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

$\pi[v] \leftarrow u$

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] $\leftarrow \infty$; π [v] $\leftarrow$ null;

  d[s] $\leftarrow$ 0

  S $\leftarrow \emptyset$

  Q = BuildPriorityQueue(V, d) ← O(V) for binary or Fibonacci heap

  While Q not empty

V calls → u $\leftarrow$ DeleteMin(Q) ← O(log V)/call for binary or Fibonacci heaps

    S $\leftarrow$ S $\cup$ u

    For v in Adj[u]

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

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

        π[v] $\leftarrow$ u

at most E calls → UpdatePQ(v, d[v]) ← 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!

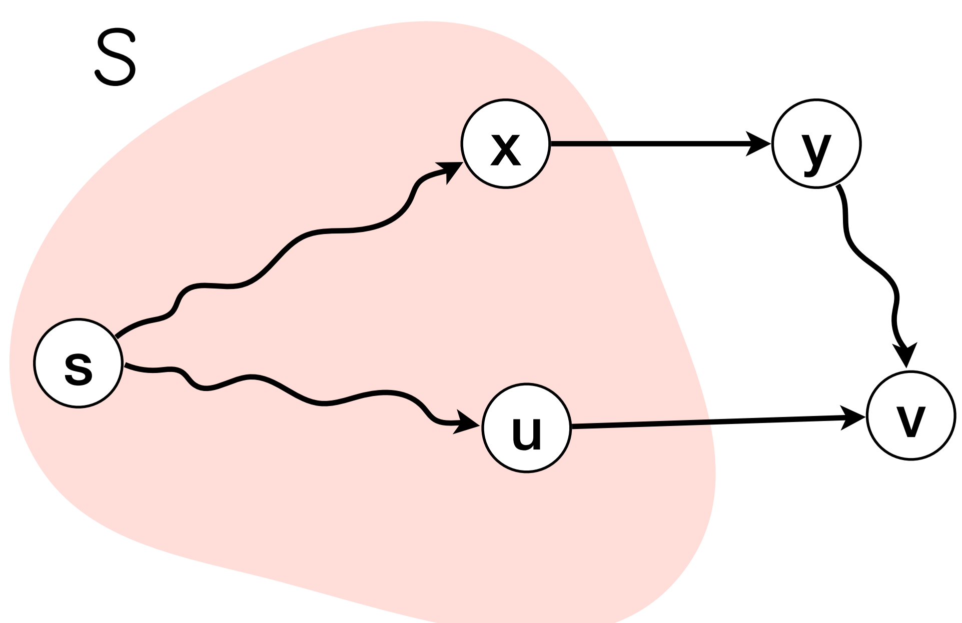
Dijkstra's Algorithm - Correctness

- 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

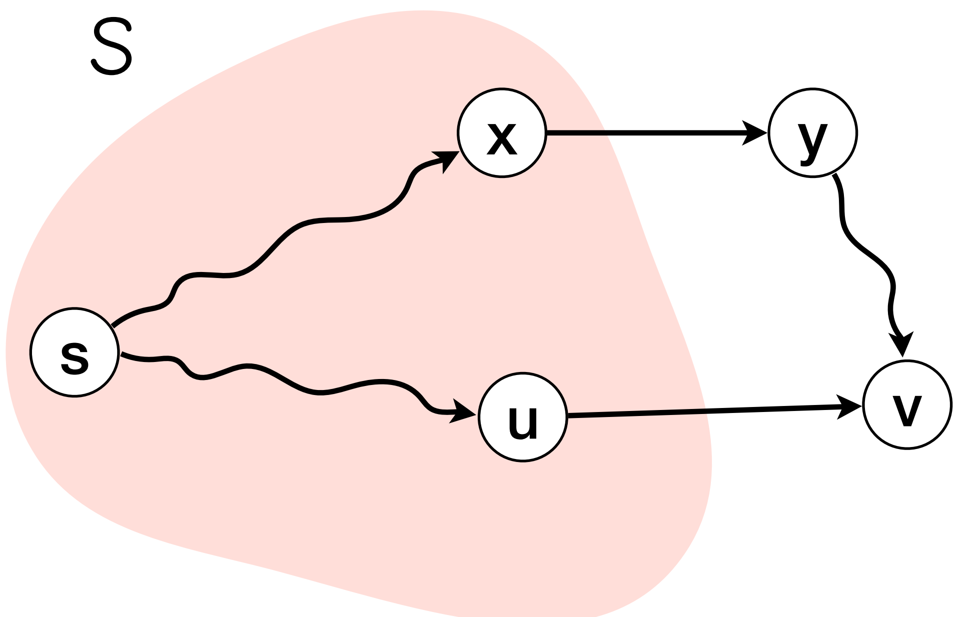
- Let |S| = k and suppose algorithm is about to add v to S, by way of u in S
- Let P_v be the algorithm's s-v path after the addition, with penultimate vertex u in S
- Consider an arbitrary alternative path P'_v
- P'_v has a first edge (x, y) that crosses the cut $(S, V \setminus S)$

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



Dijkstra's Algorithm - Correctness

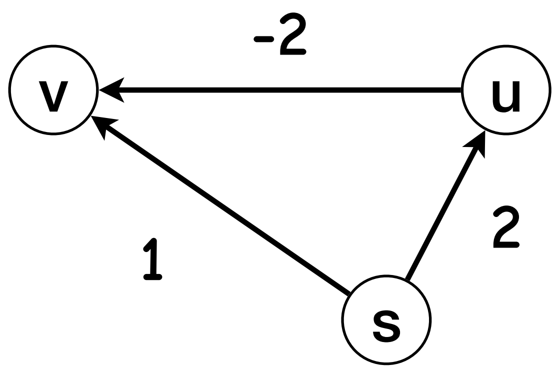
- Consider an arbitrary alternative path P'_v
- P'_v has a first edge (x, y) that crosses the cut $(S, V \setminus S)$



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