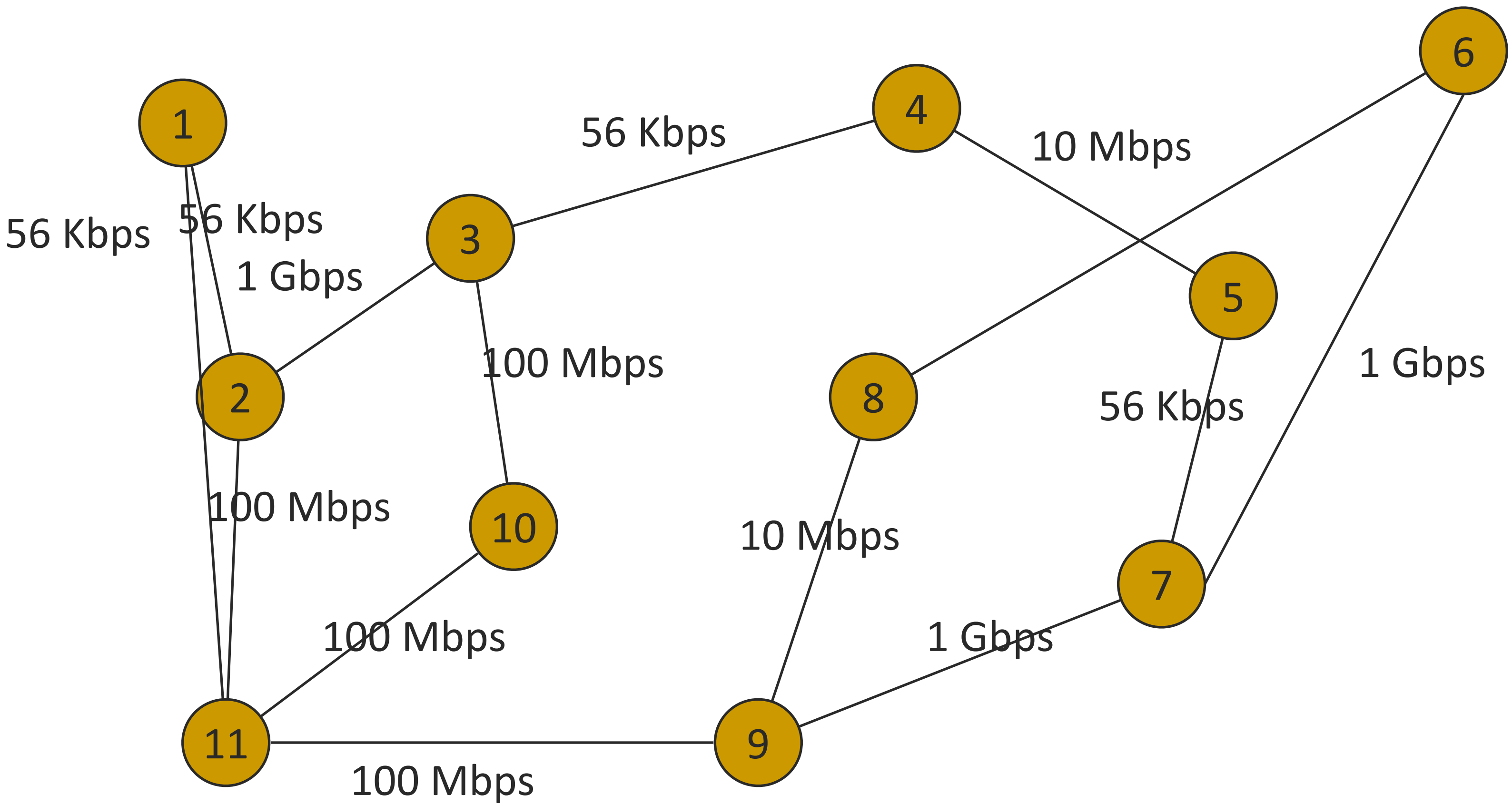
Shortest Paths

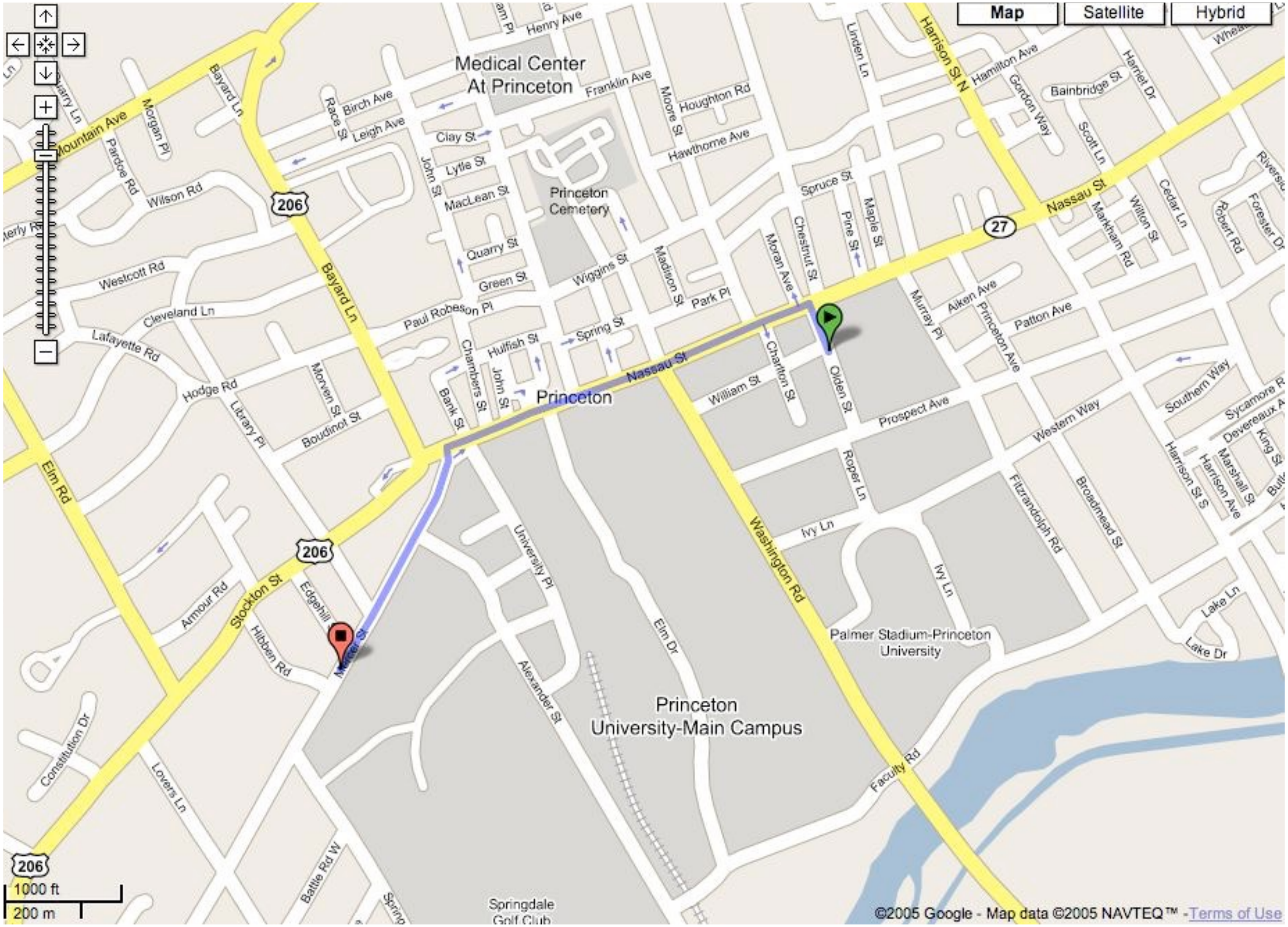
in edge-weighted graphs

Communication Speeds in a Computer Network

Find fastest way to route a data packet between two computers



Google maps


Map
Satellite
Hybrid
Medical Center
At Princeton
Princeton
Cemetery
Princeton
Nassau St
Washington Rd
Mountain Ave
Bayard Ln
Stockton St
Elm Rd
Harrison St N
Palmer Stadium-Princeton
University
Princeton
University-Main Campus
Springdale
Golf Club
206
27
1000 ft
200 m
©2005 Google - Map data ©2005 NAVTEQ™ - Terms of Use

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Texture mapping.
- Robot navigation.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

http://en.wikipedia.org/wiki/Seam_carving

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest Path problems

- Find a shortest path between two given vertices
- Single source shortest paths
- Single sink shortest paths
- All pair shortest paths

Single Source Shortest Path problems

- Undirected graphs with non-negative edge weights
- Directed graphs with non-negative edge weights
- Directed graphs with arbitrary weights

Single Source Shortest Paths

- If graph is not weighted (all edge-weights are unit-weight): BFS works
- Now assume: graph is edge-weighted
  - Every edge is associated with a positive number
    - Possible weights: integers, real numbers, rational numbers
    - Edge-weights can represent: distance, connection cost, affinity

Single Source Shortest Paths and shortest distances

- Input: An edge-weighted undirected graph and a source node v with: for every edge e edge-weight w(e) > 0
- Output: All single-source shortest paths (and their weight) for v in G: for every node $w \neq v$ in G a shortest path from v to w.
  - Here, a *path* p from v to w consisting of edges $e_0, e_1, \dots, e_{k-1}$ is shortest in G, if its length

$$w(p) = \sum_{i=0}^{k-1} w(e_i)$$

is minimum (i.e., there is no path from v to w in G that is shorter).

Let d(u) = length of shortest path from v to u

Algorithm
`DijkstraShortestPaths(G,v)`

Input: A simple undirected graph G with nonnegative edge-weights, a distinguished vertex v in G

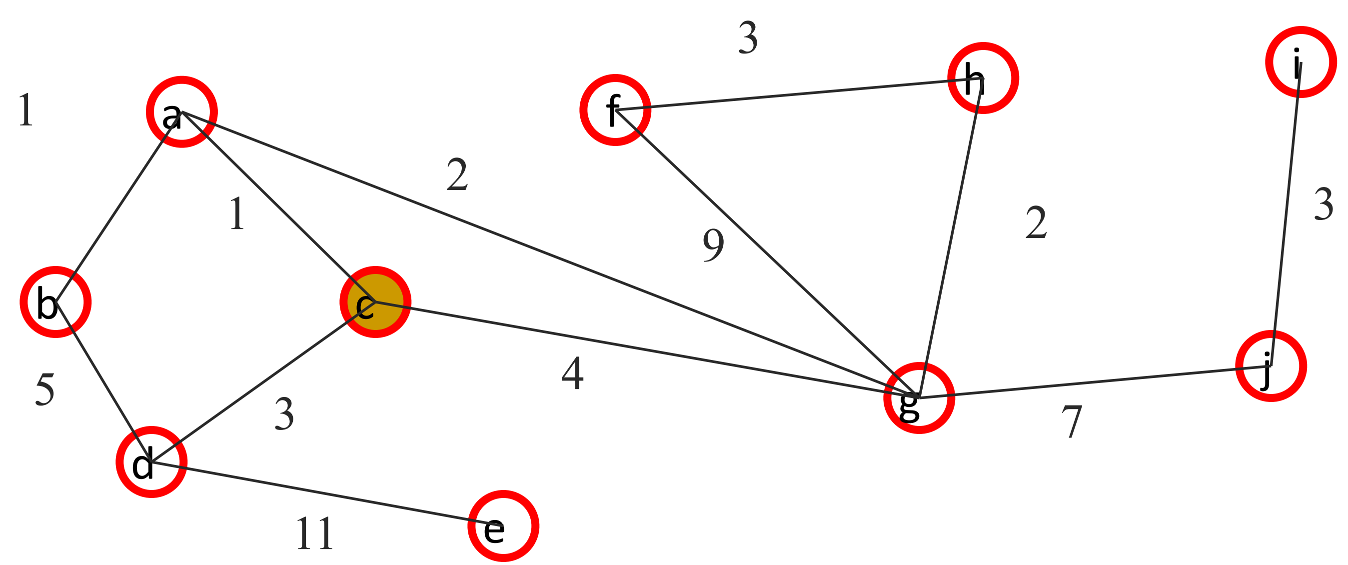
Output: A label D[u] for each vertex u in G such that D[u]=d(u)

Algorithm DijkstraShortestPaths(G,v)

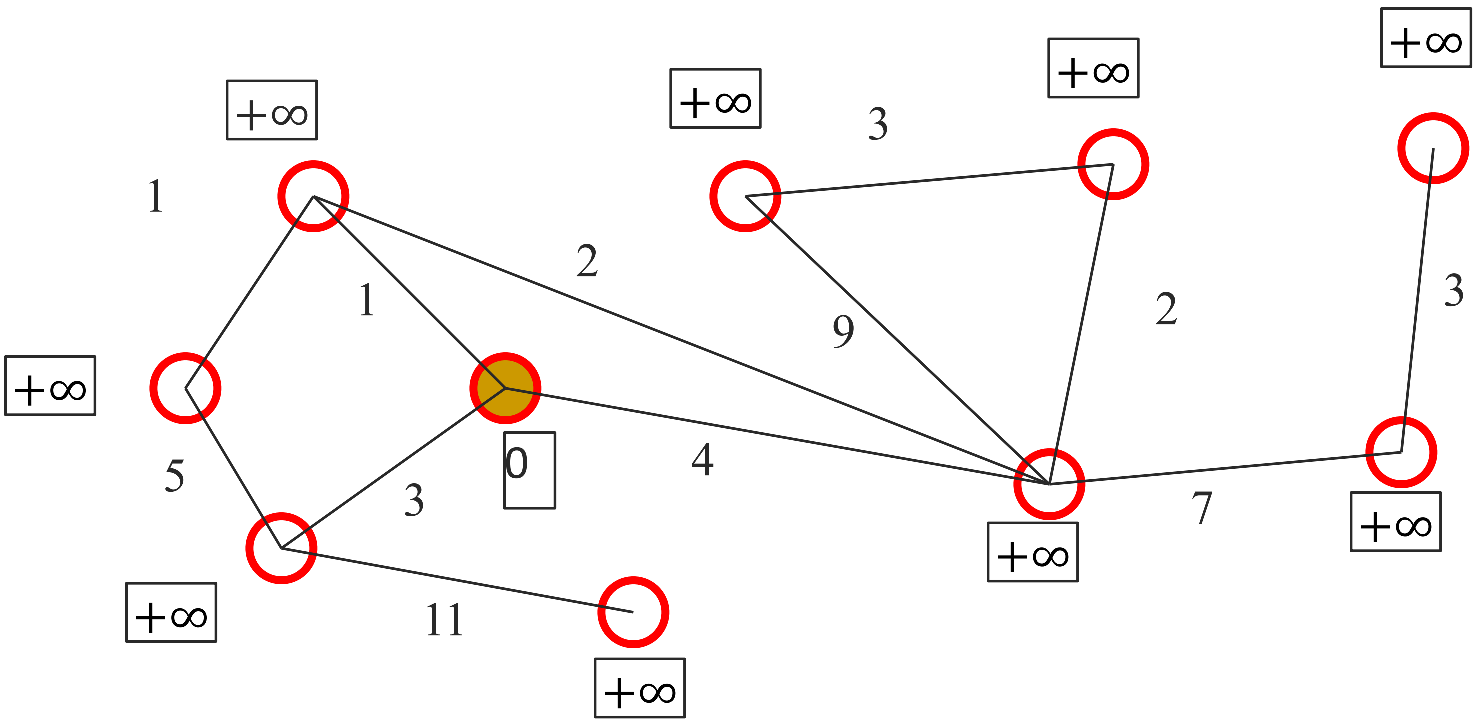
```
D[v]←0
for each vertex u≠v of G do
   D[u]← +∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to T
   for each vertex z∈N(u) with z∈Q do
      if D[u]+w((u,z)) < D[z] then
         D[z]←D[u]+w((u,z))
         update z's key in Q to D[z]
return D
```

Relaxation

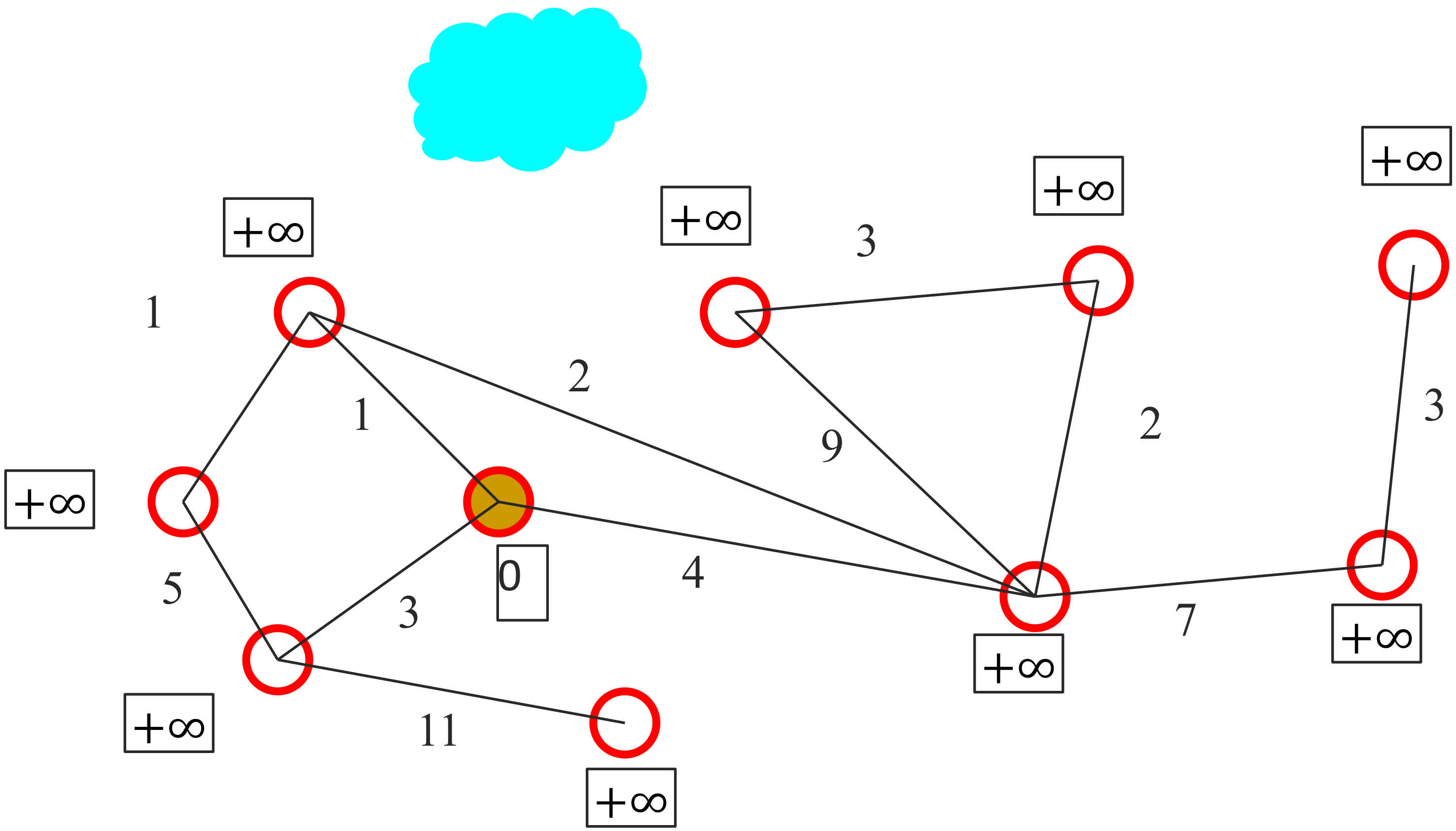
Dijkstra’s algorithm: a greedy algorithm



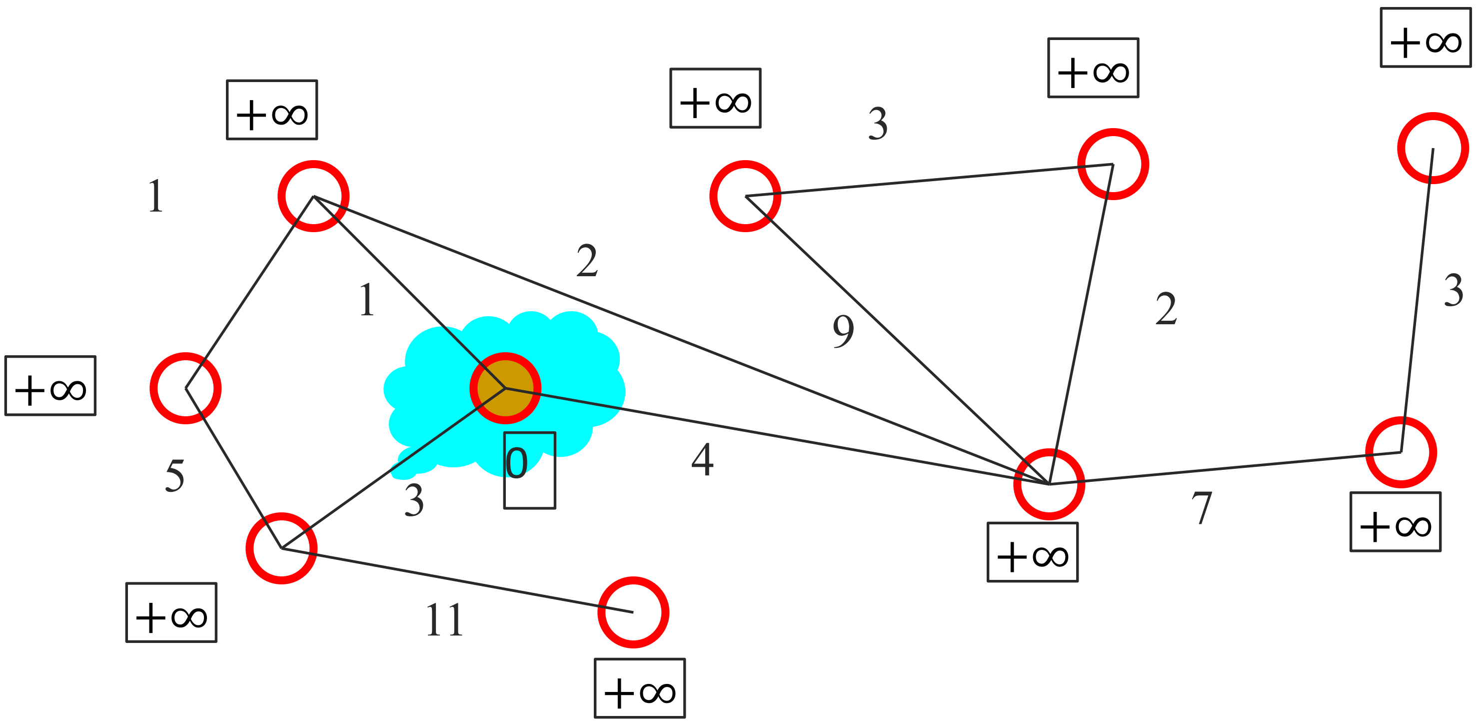
Dijkstra’s algorithm: Initializing



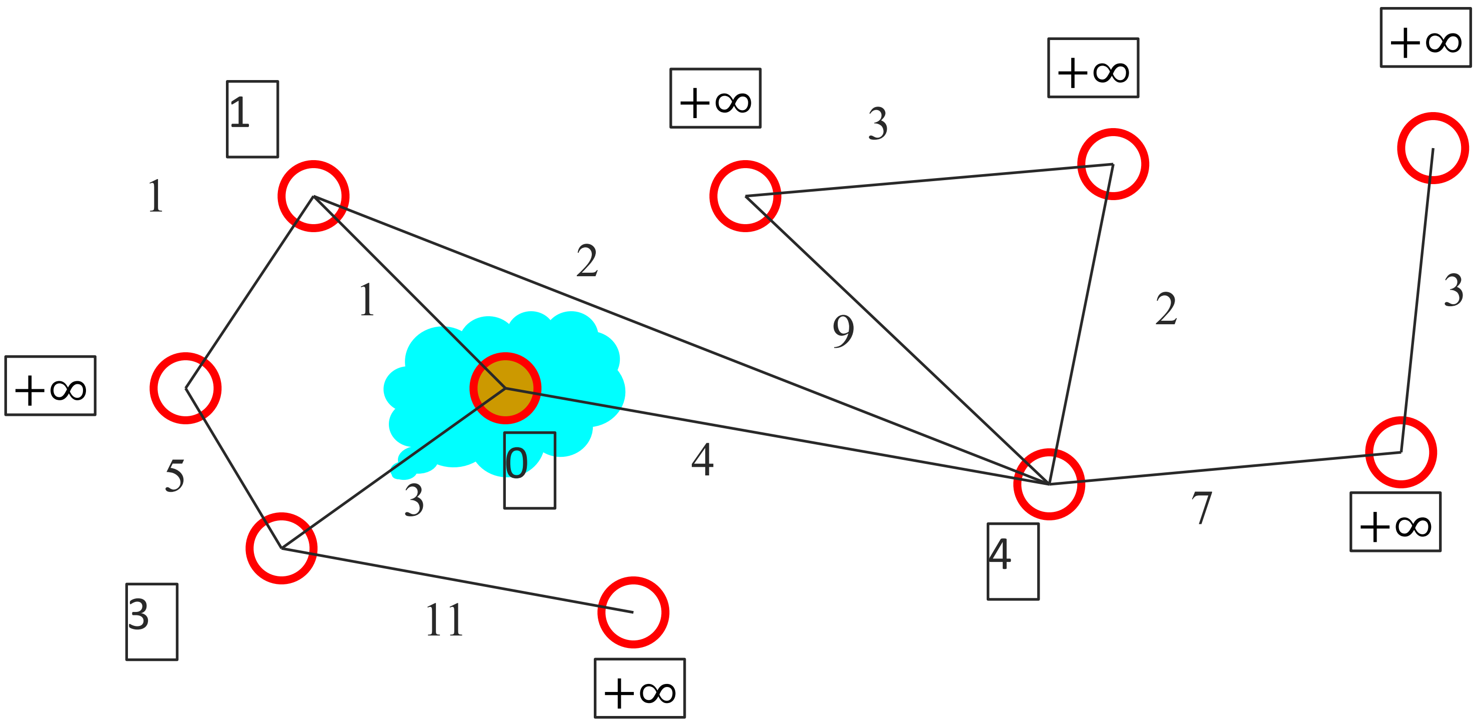
Dijkstra's algorithm: Initializing Cloud C (consisting of "solved" subgraph)



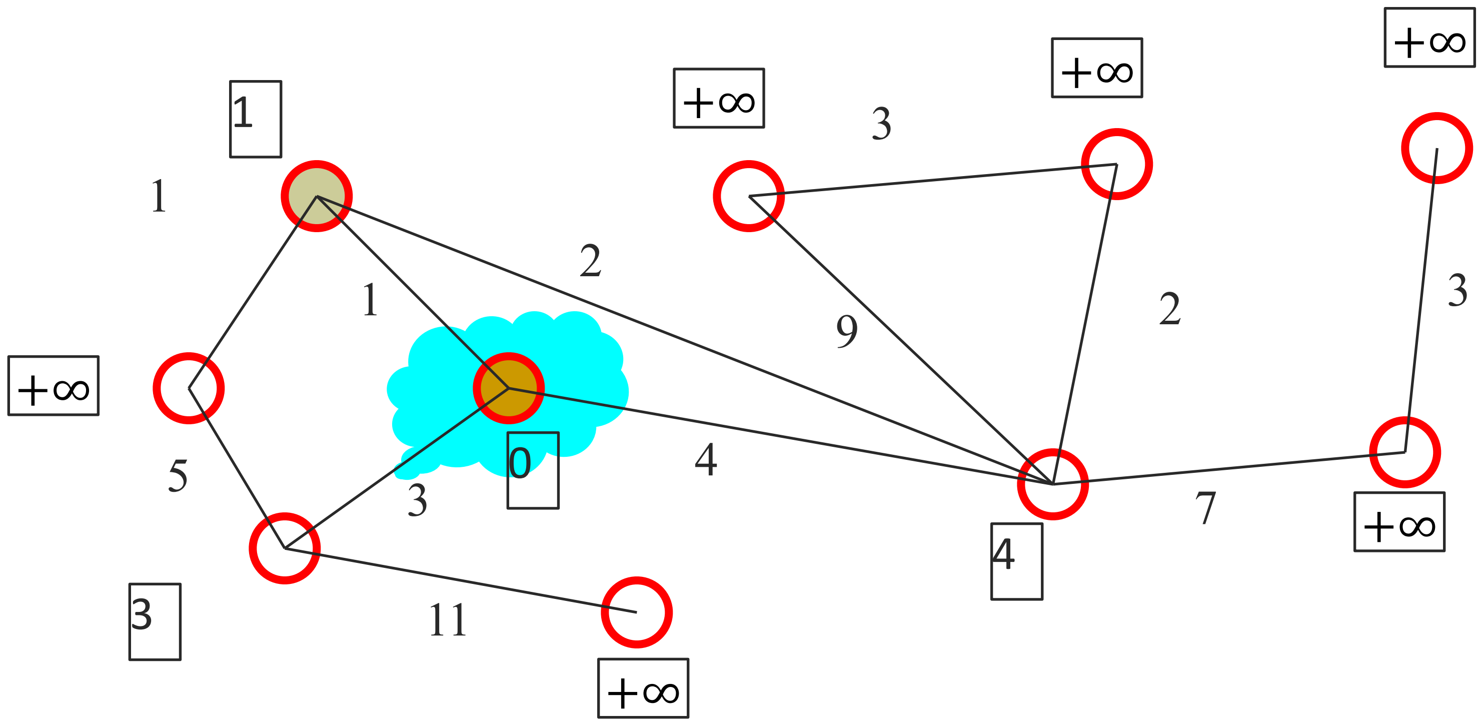
Dijkstra's algorithm: pull v into C



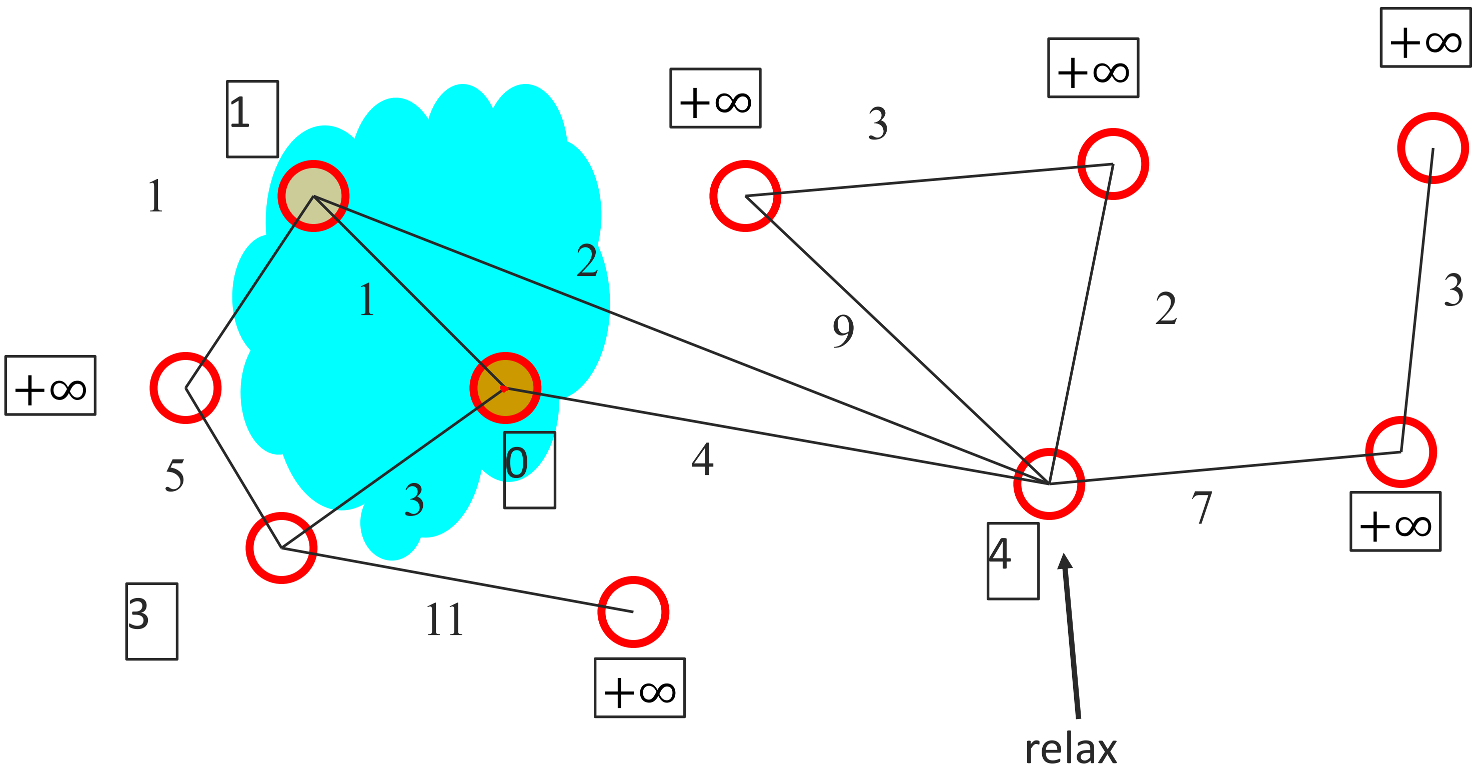
Dijkstra's algorithm: update *C's* neighborhood



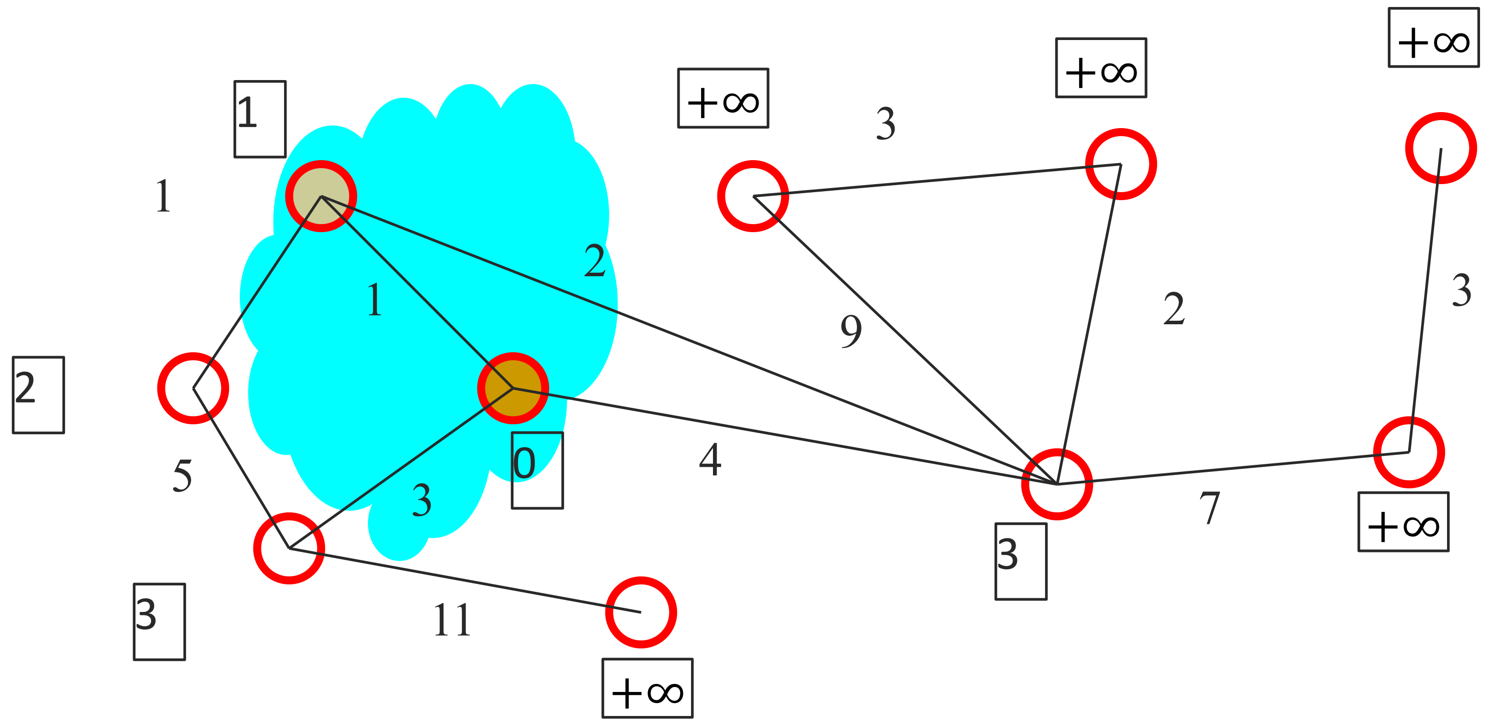
Dijkstra’s algorithm: pick closest vertex u outside C



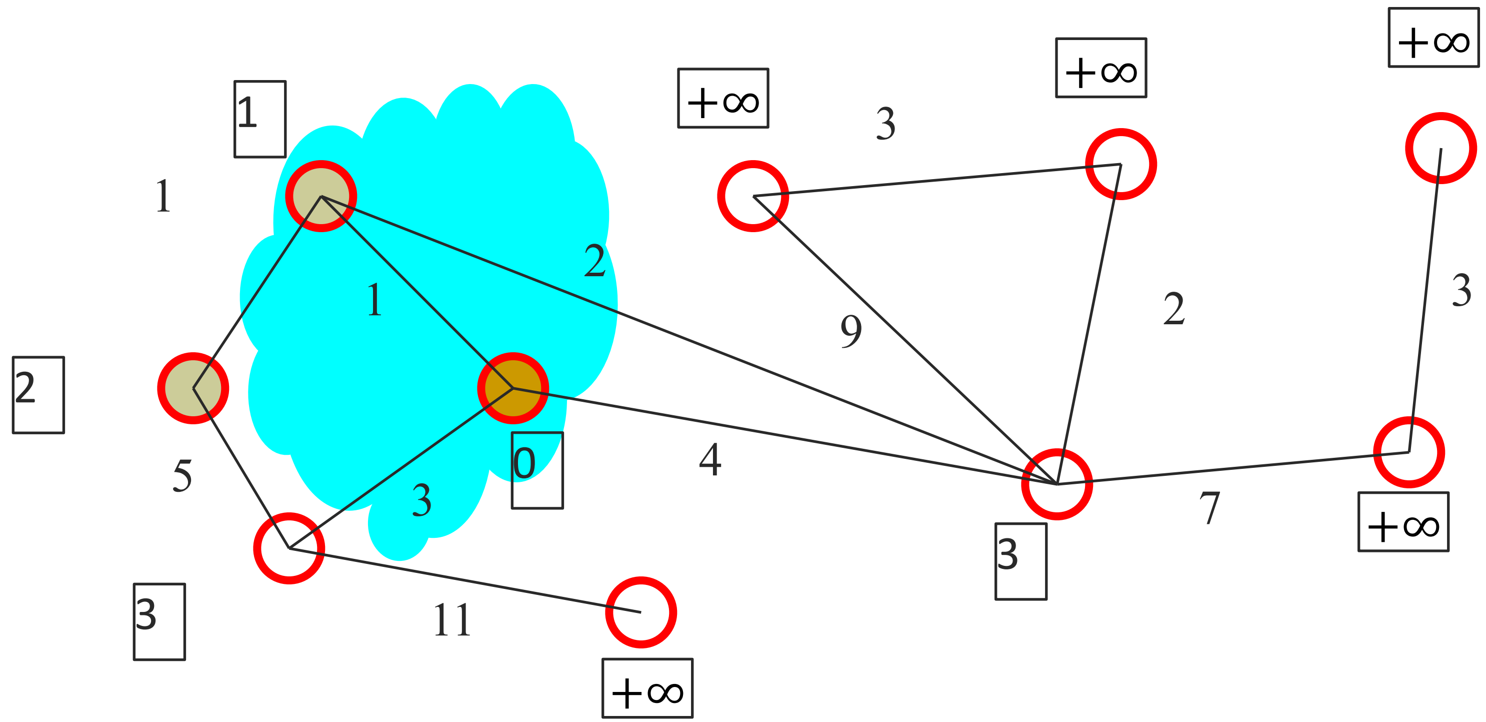
Dijkstra's algorithm: pull u into C



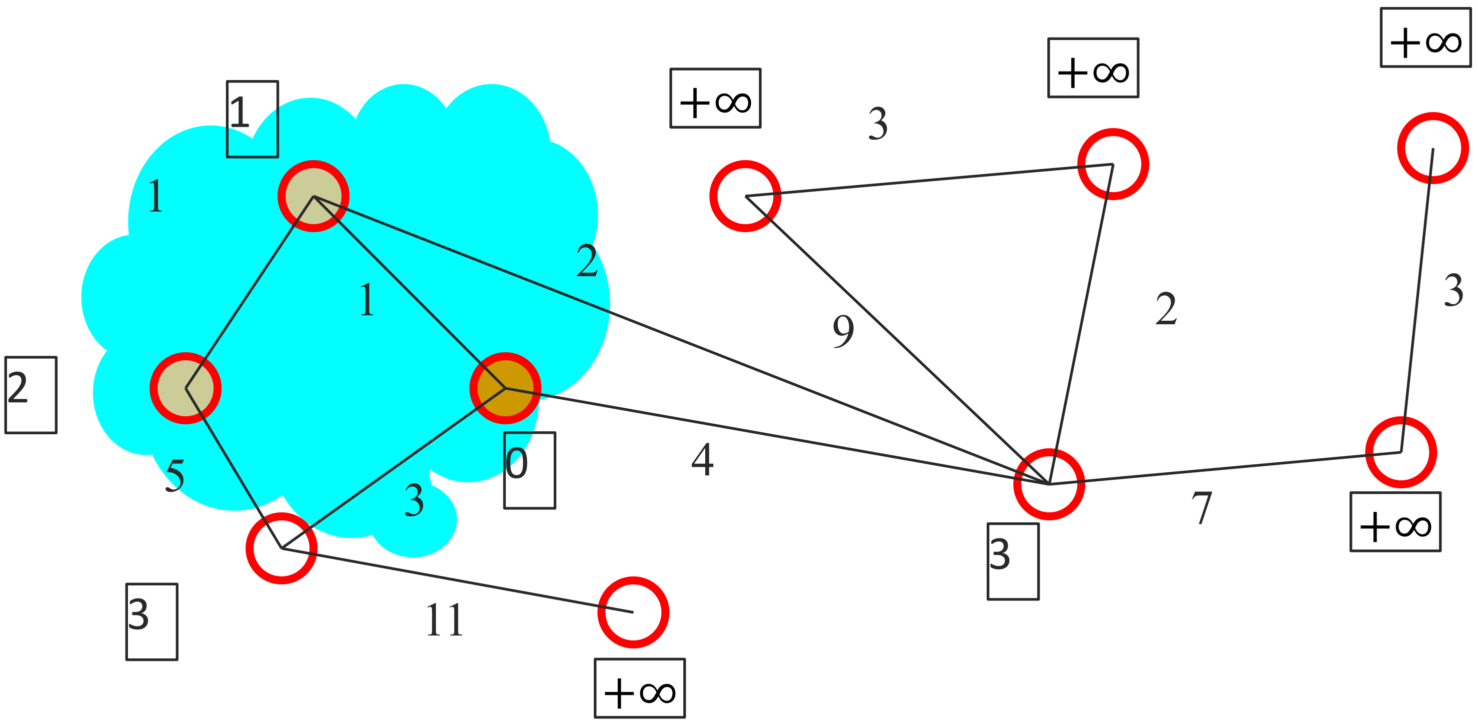
Dijkstra's algorithm: update *C's* neighborhood



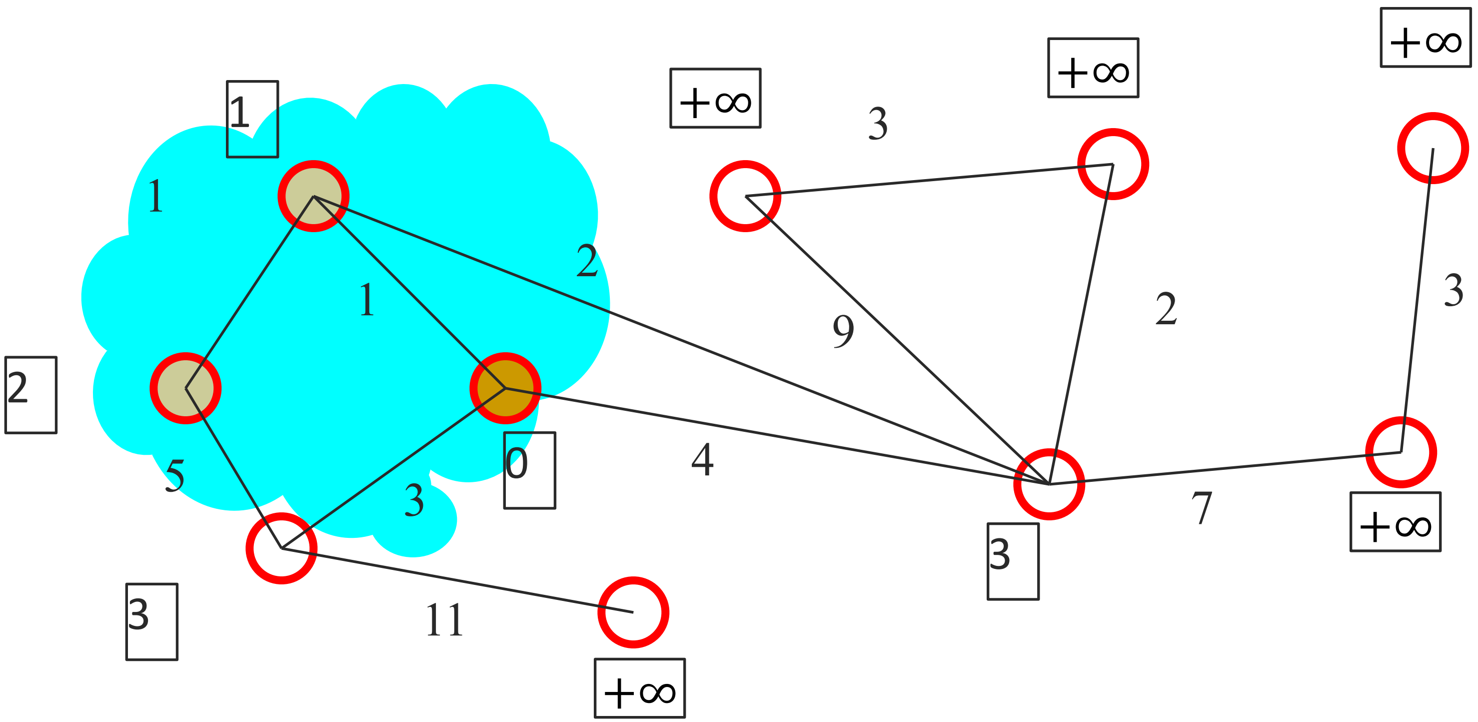
Dijkstra's algorithm: pick closest vertex u outside C



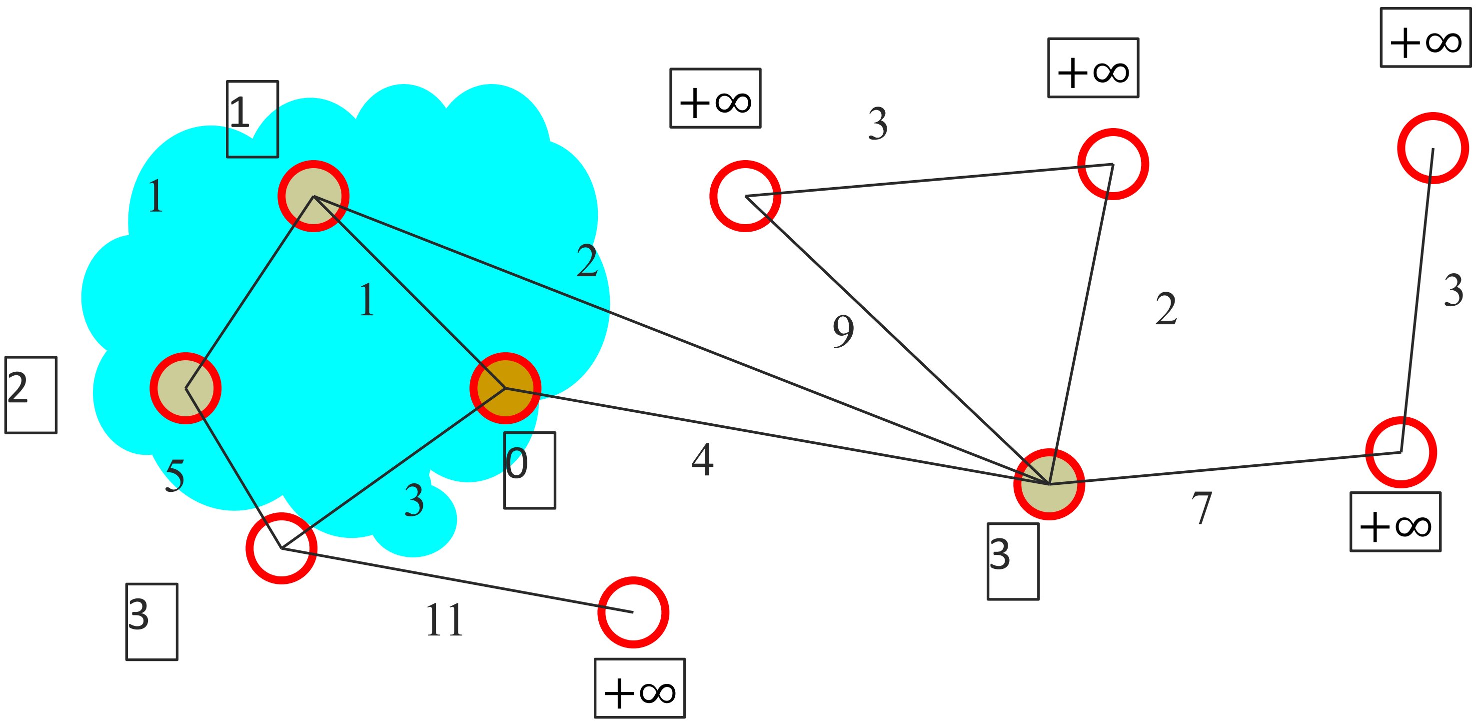
Dijkstra's algorithm: pull u into C



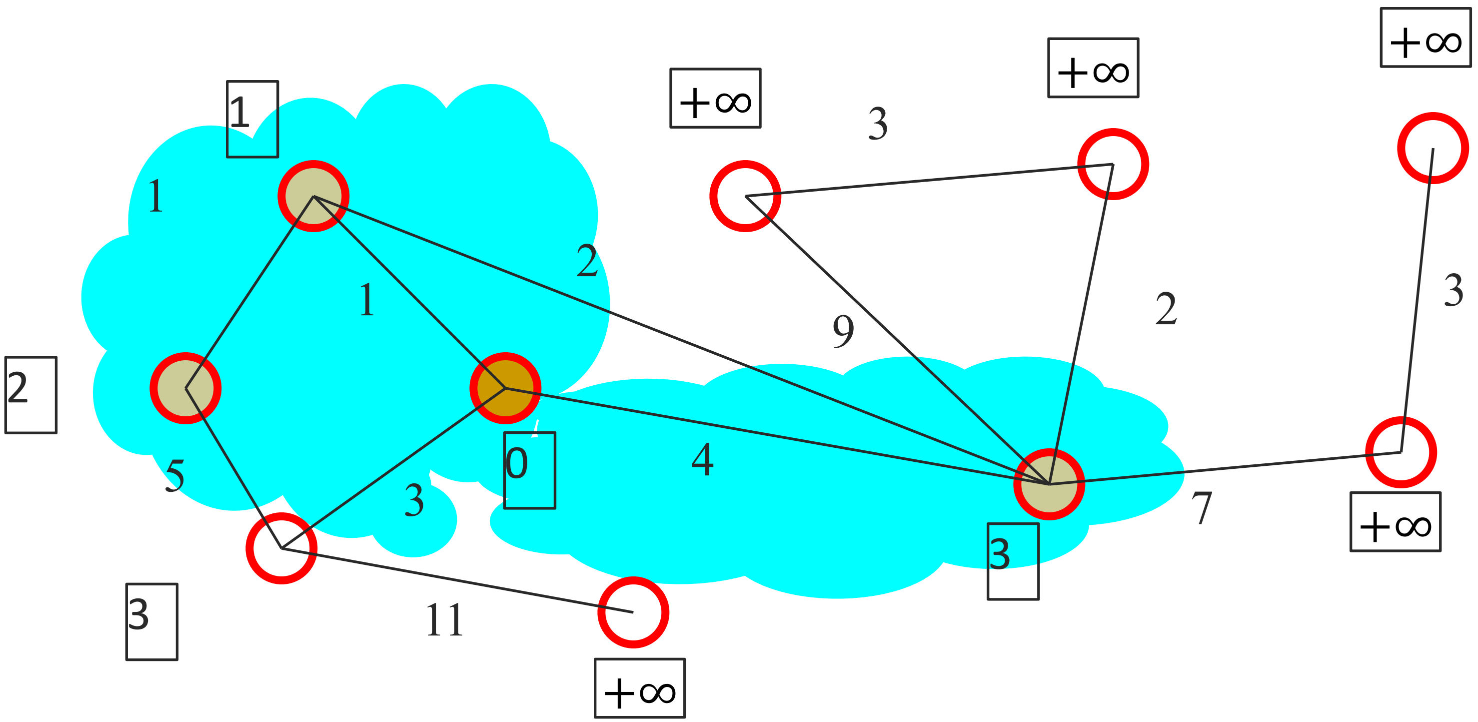
Dijkstra's algorithm: update *C's* neighborhood



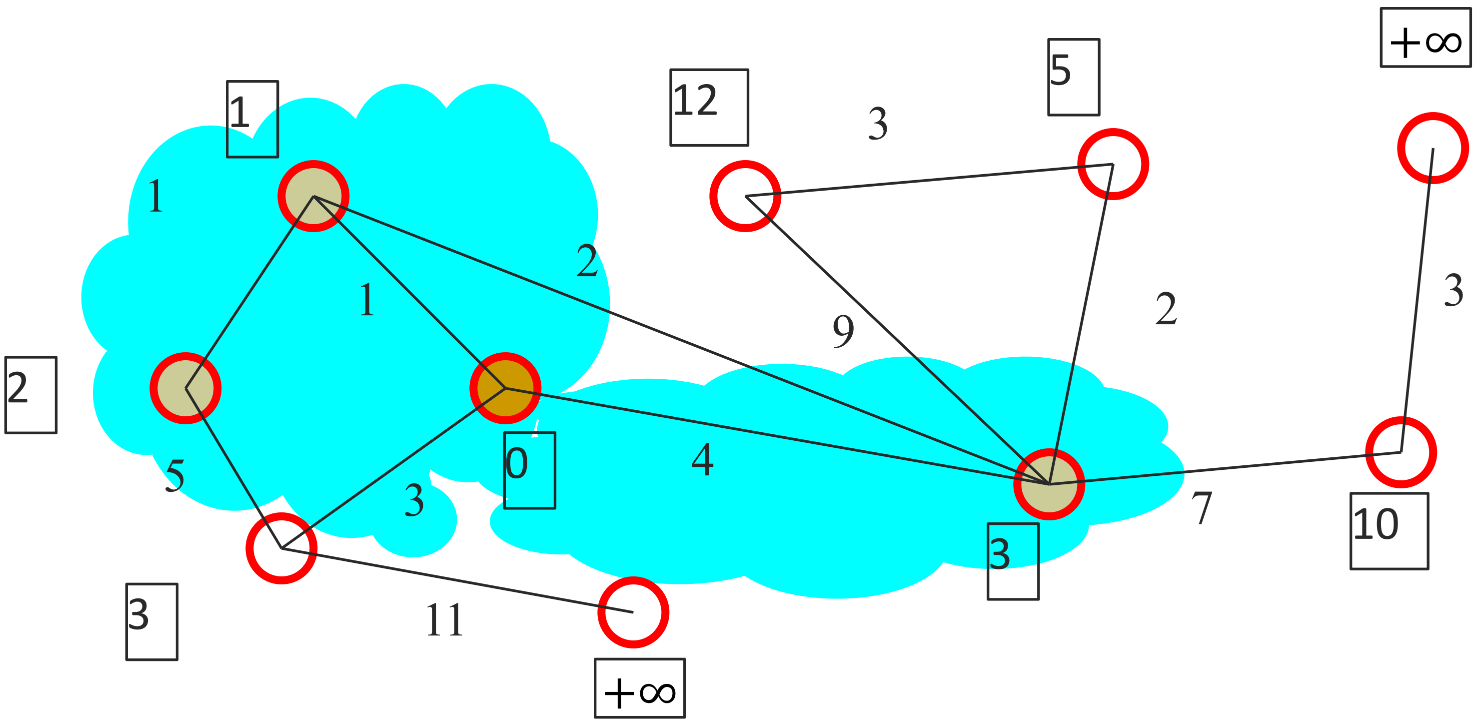
Dijkstra's algorithm: pick closest vertex u outside C



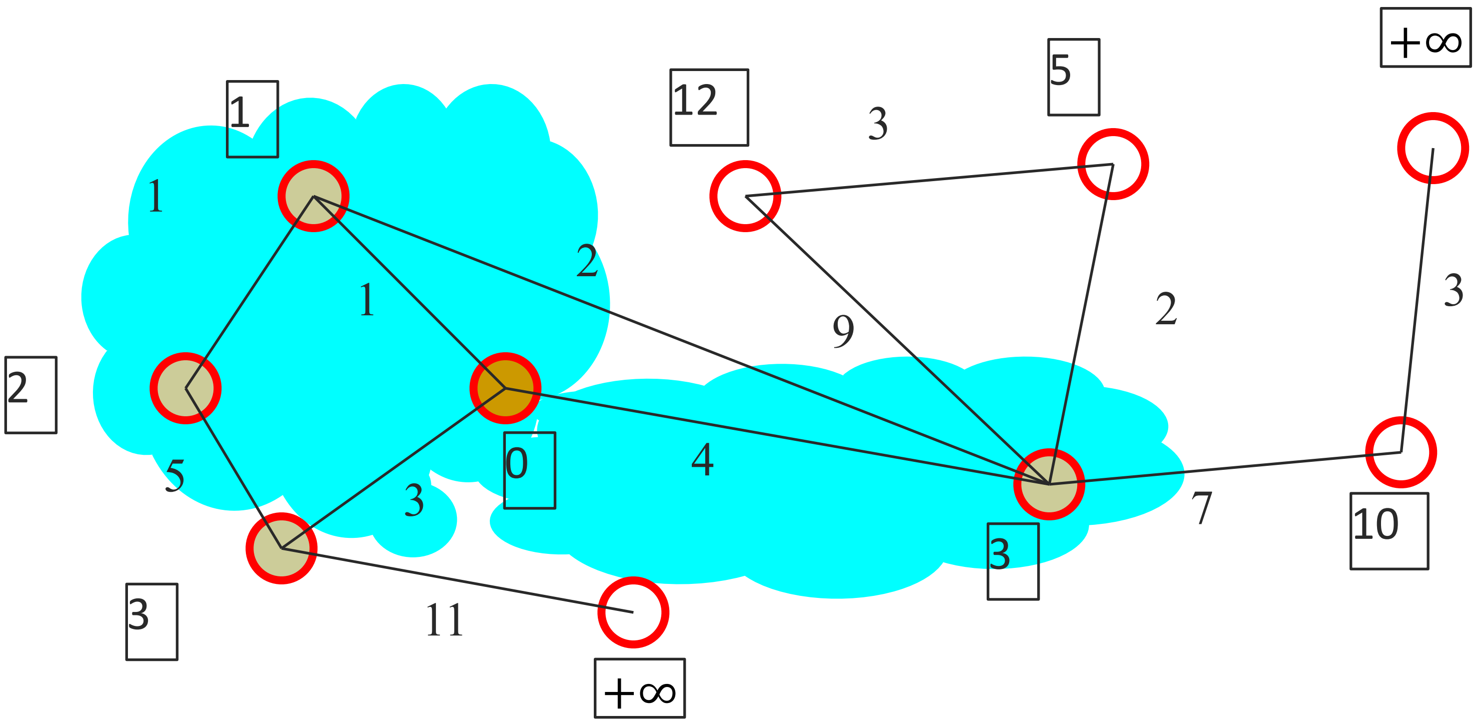
Dijkstra's algorithm: pull u into C



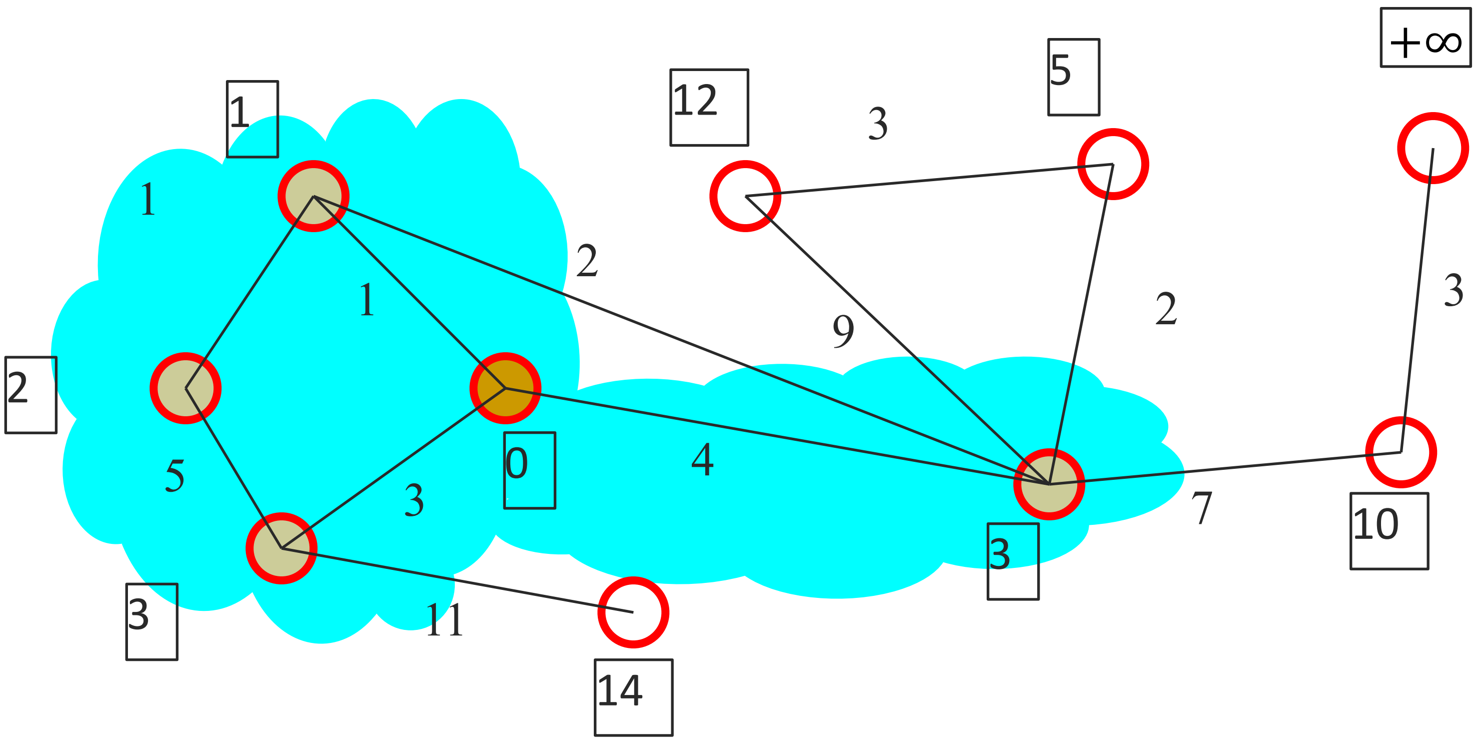
Dijkstra's algorithm: update *C's* neighborhood



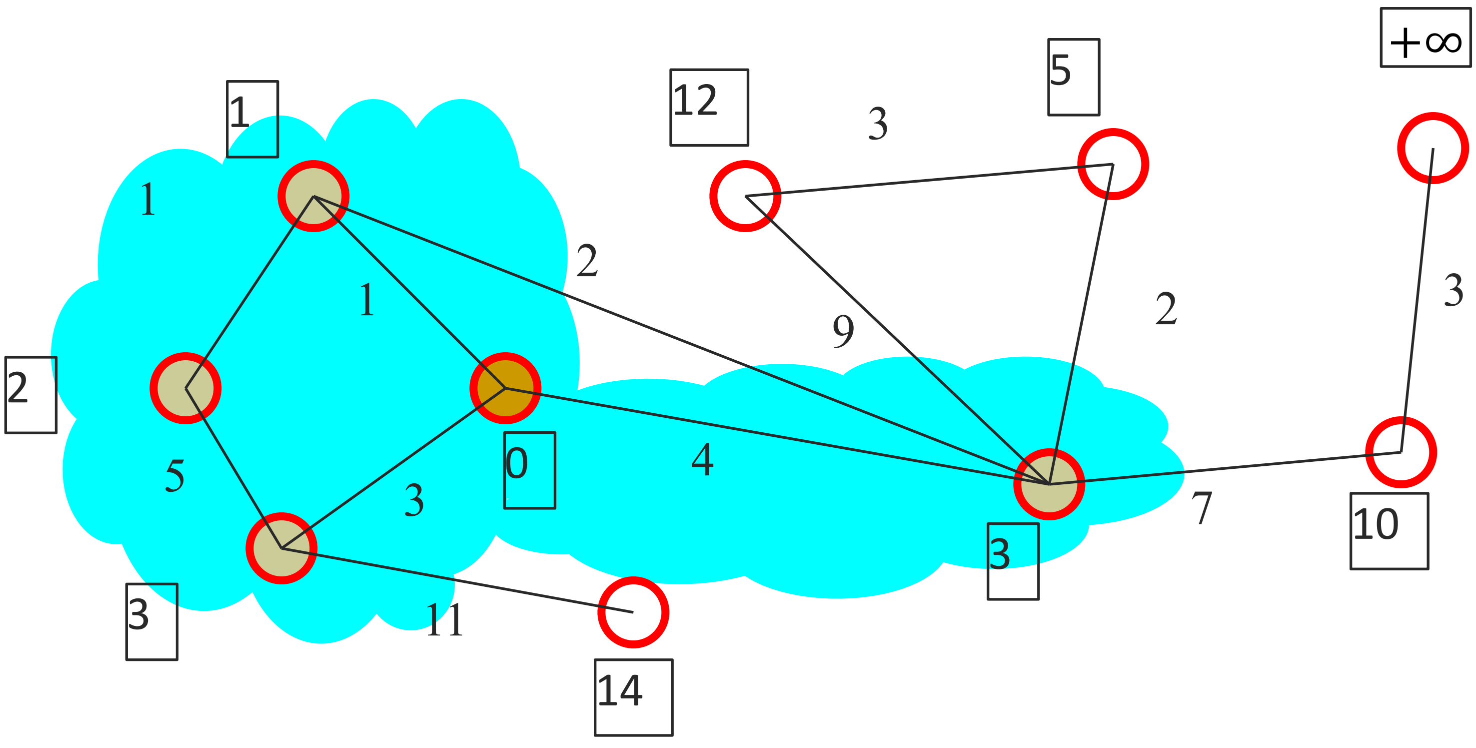
Dijkstra's algorithm: pick closest vertex u outside C



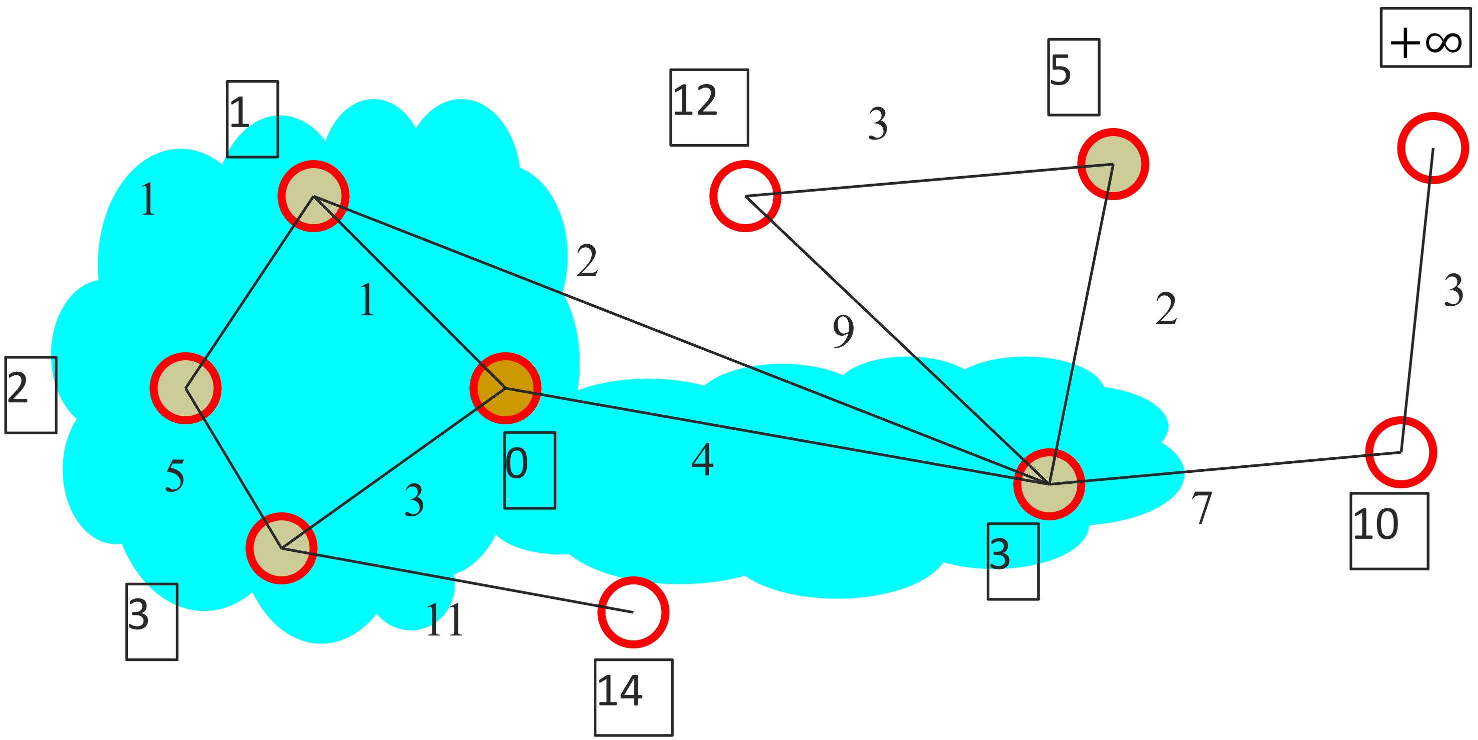
Dijkstra's algorithm: pull u into C



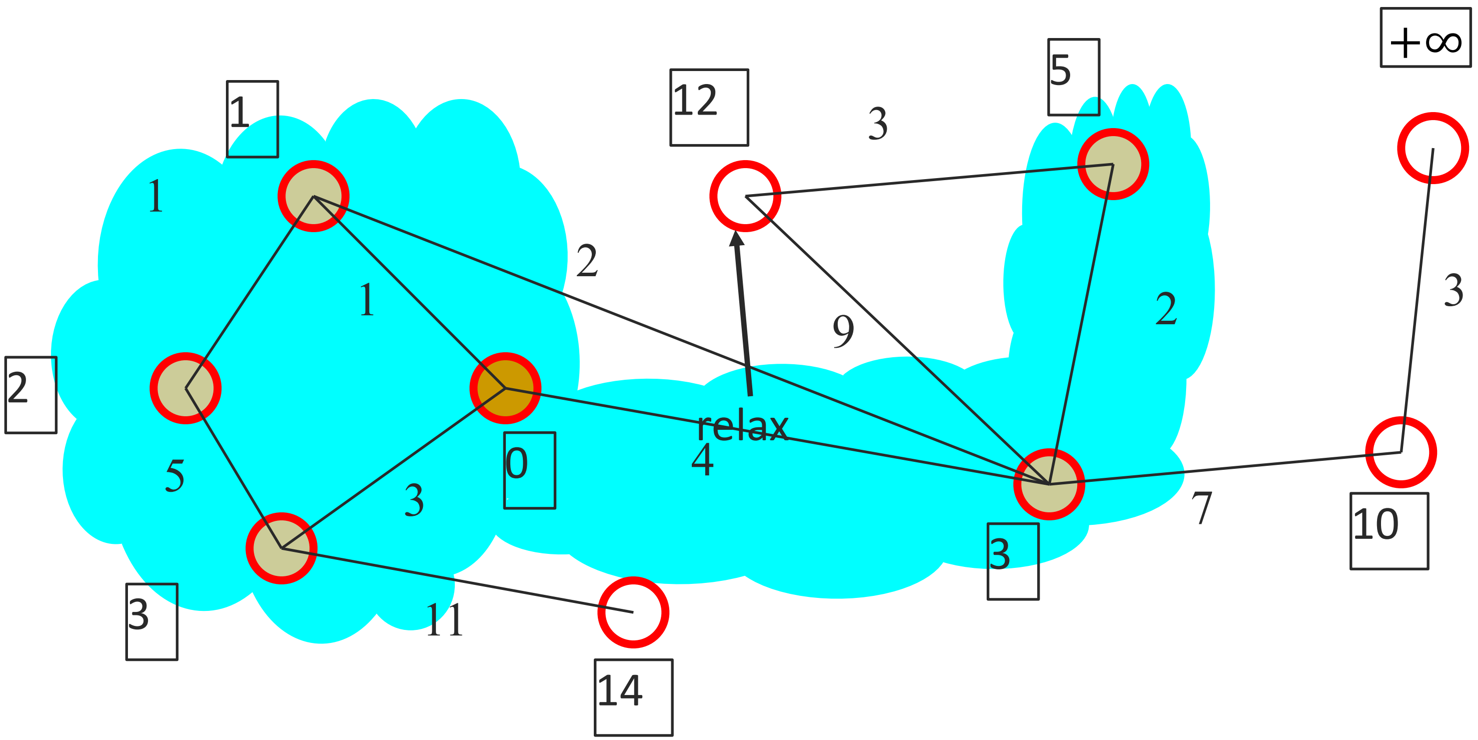
Dijkstra's algorithm: update *C's* neighborhood



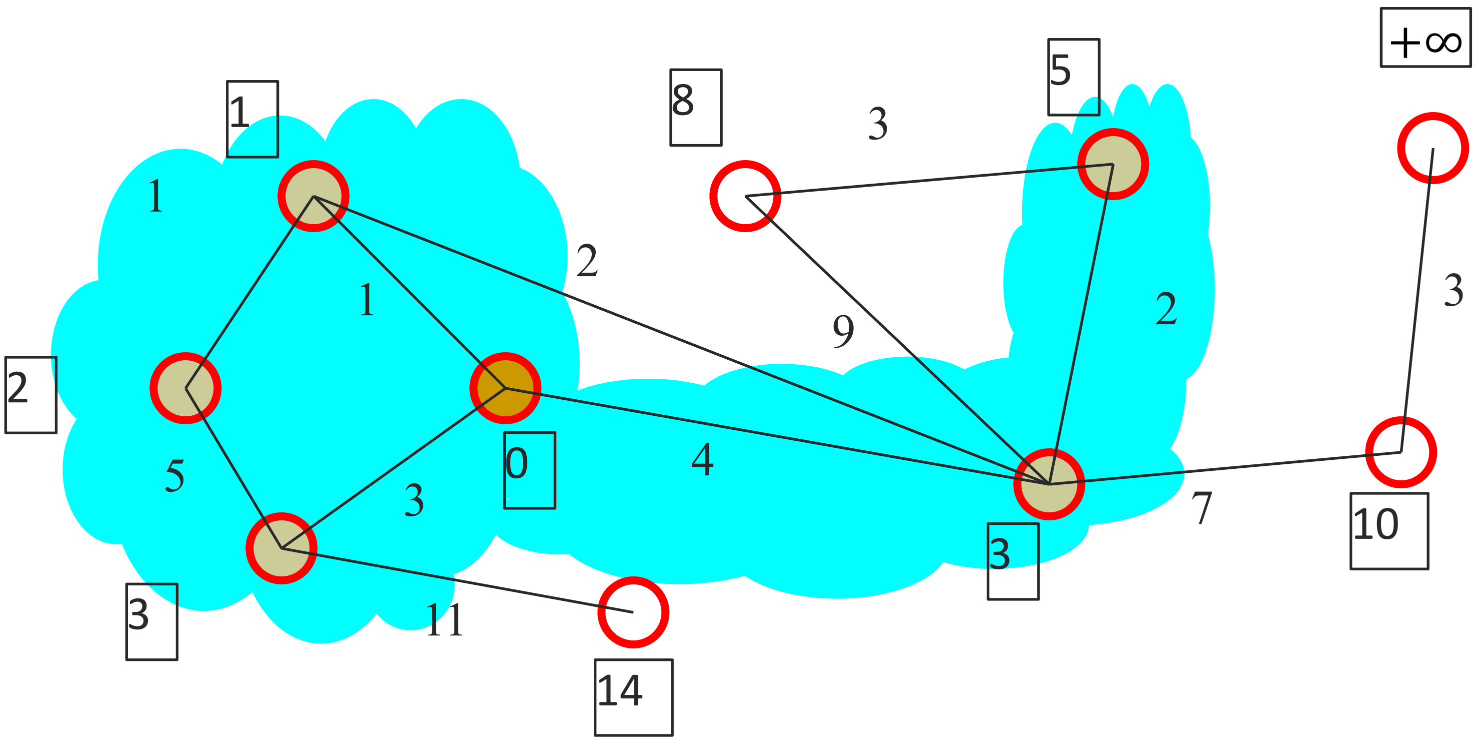
Dijkstra's algorithm: pick closest vertex u outside C



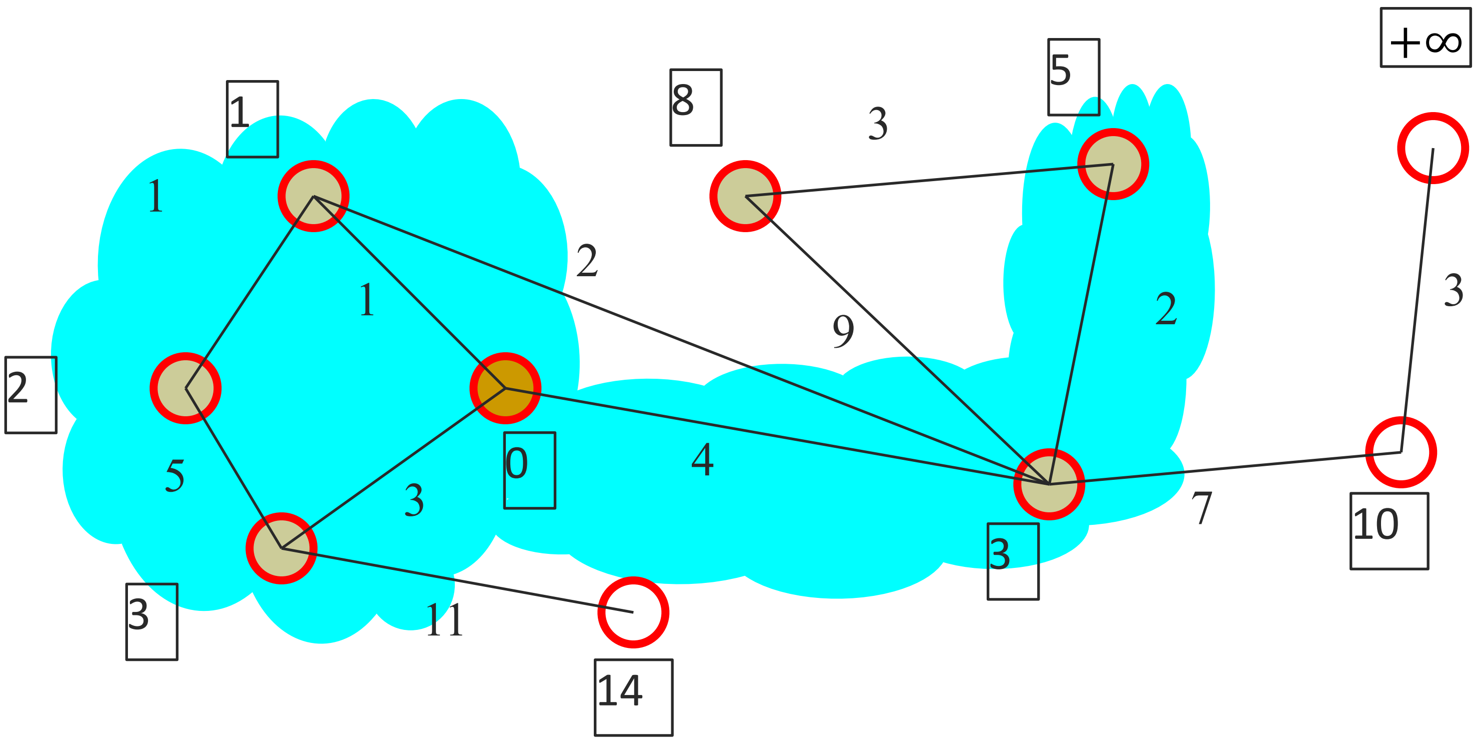
Dijkstra's algorithm: pull u into C



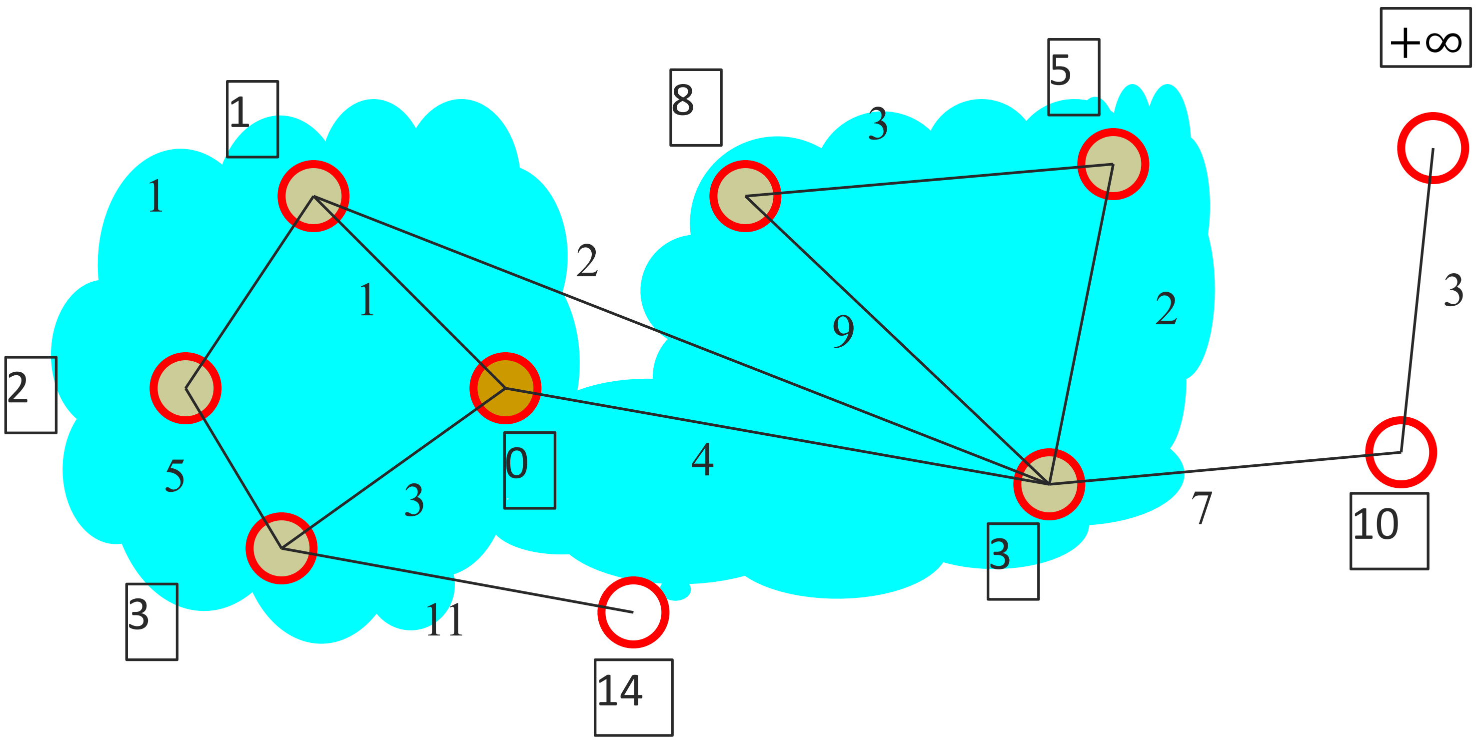
Dijkstra's algorithm: update C's neighborhood



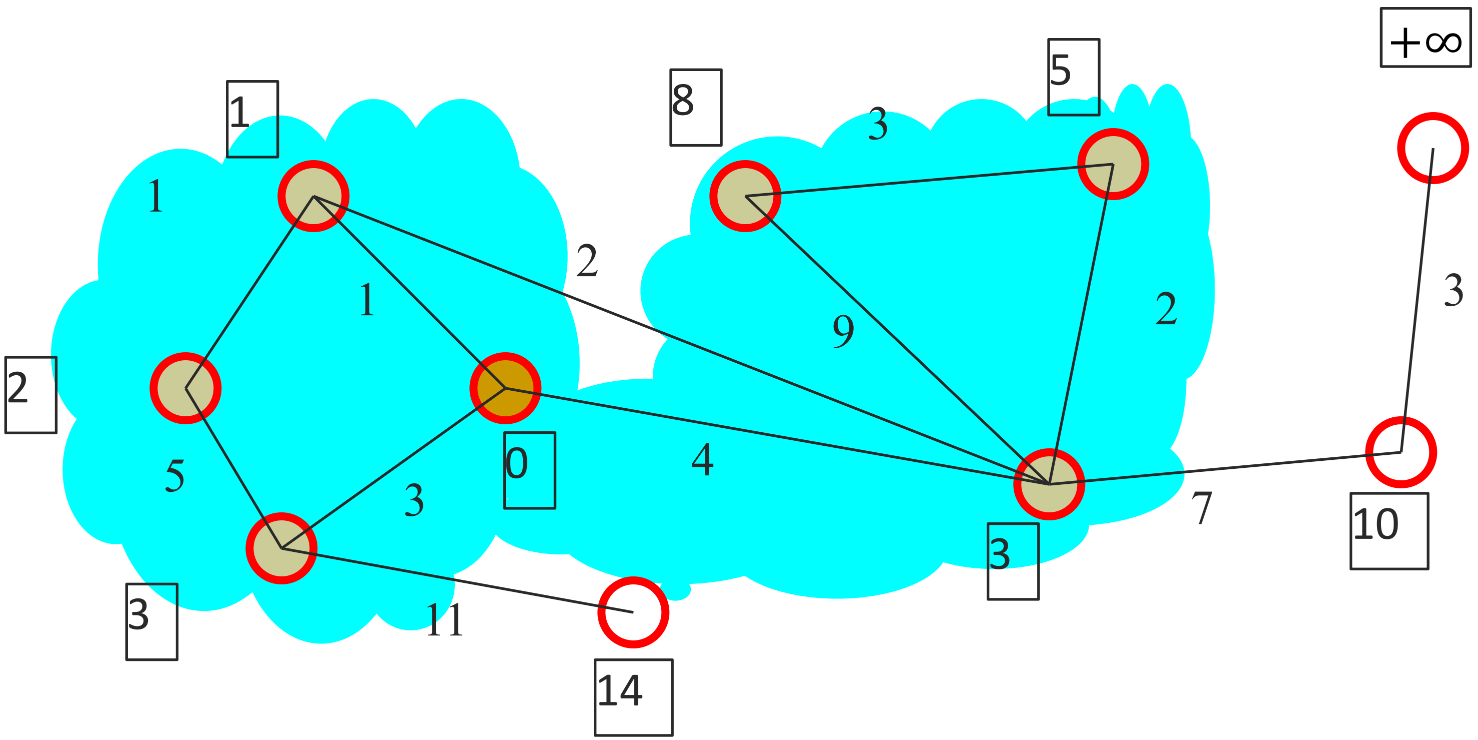
Dijkstra’s algorithm: pick closest vertex u outside C



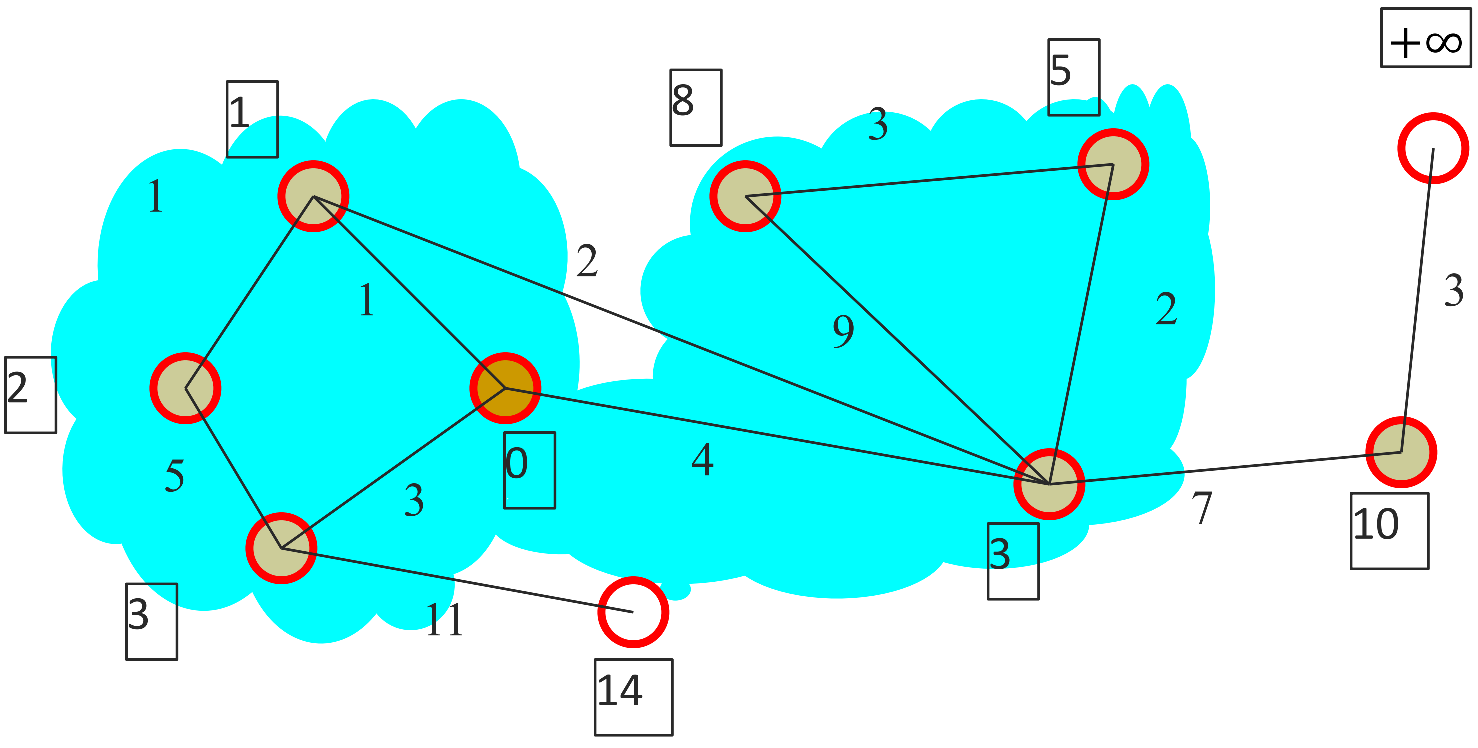
Dijkstra’s algorithm: pull u into C



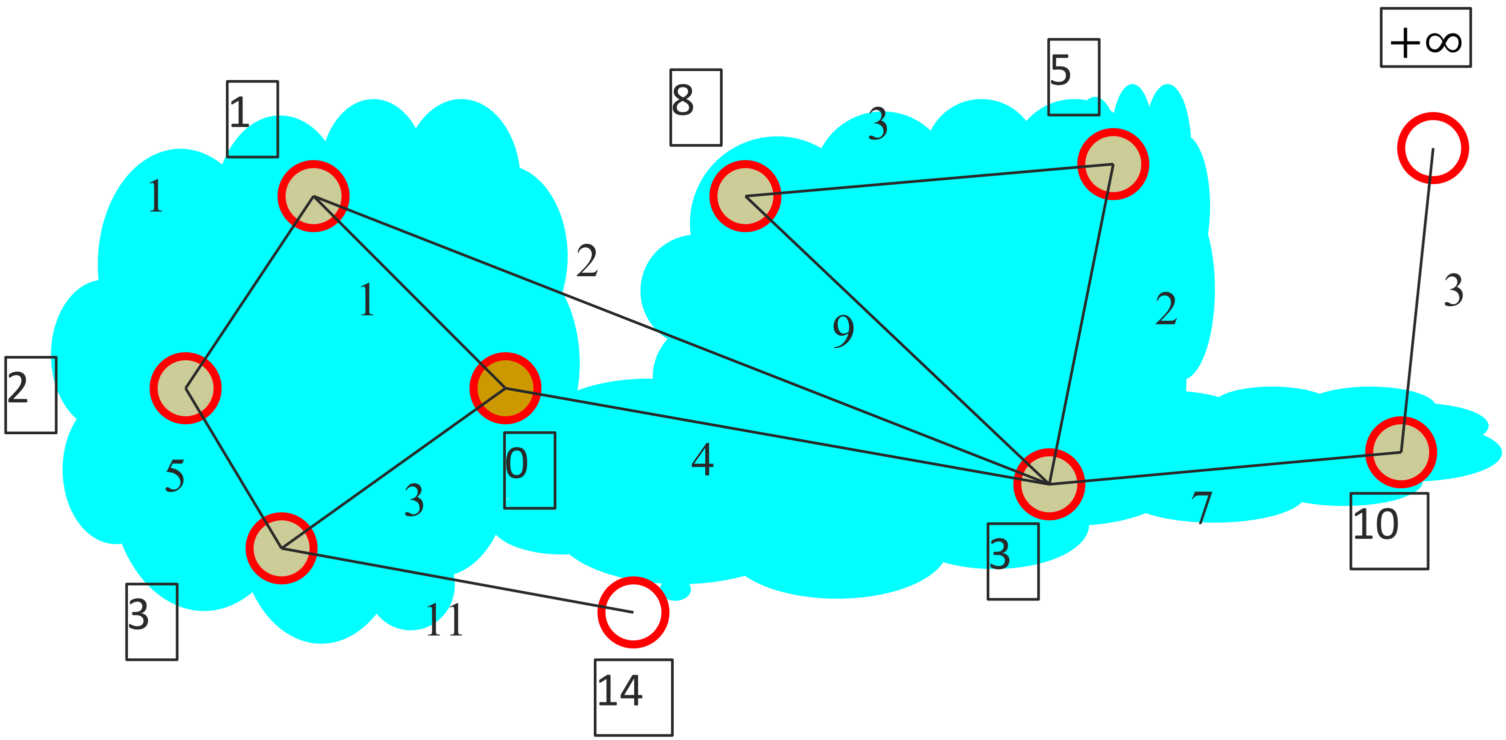
Dijkstra's algorithm: update *C's* neighborhood



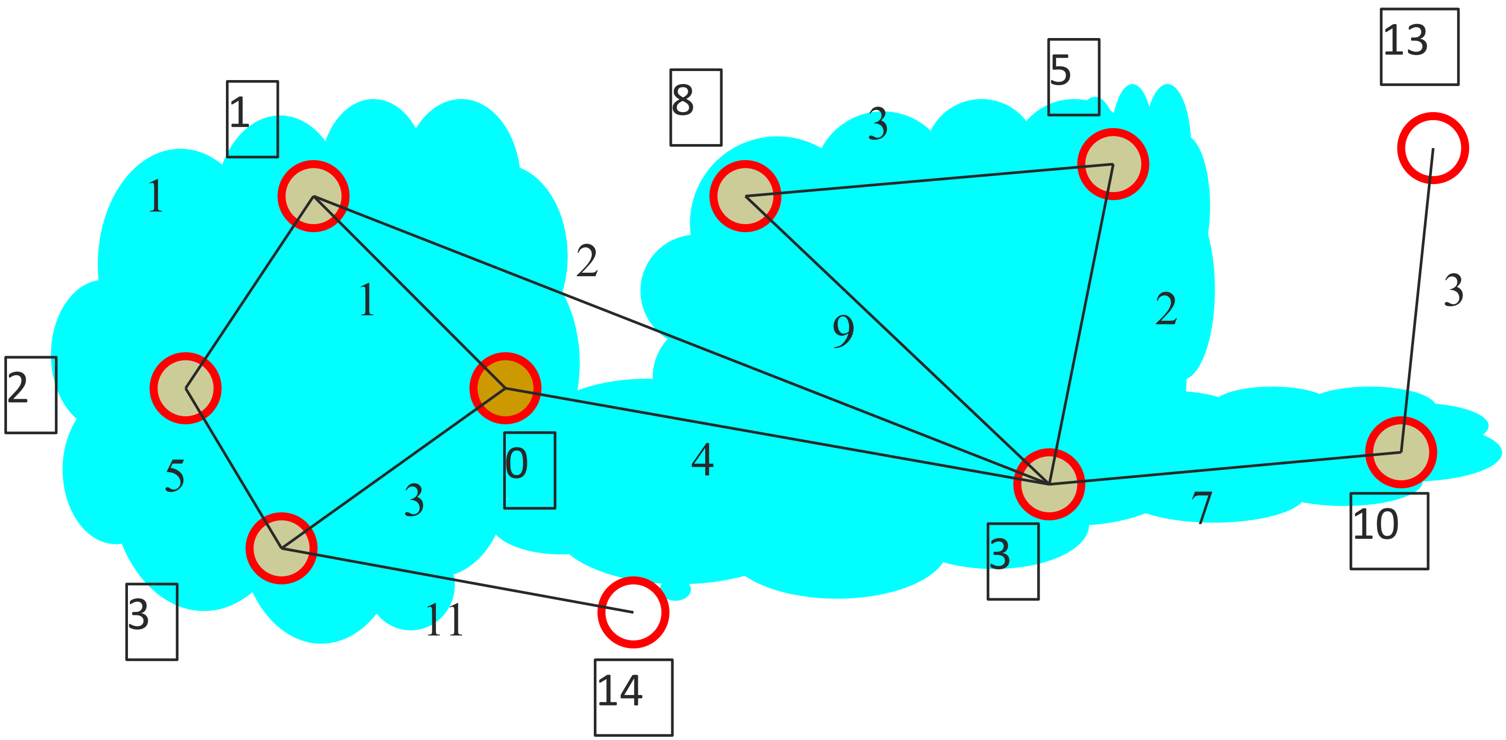
Dijkstra's algorithm: pick closest vertex u outside C



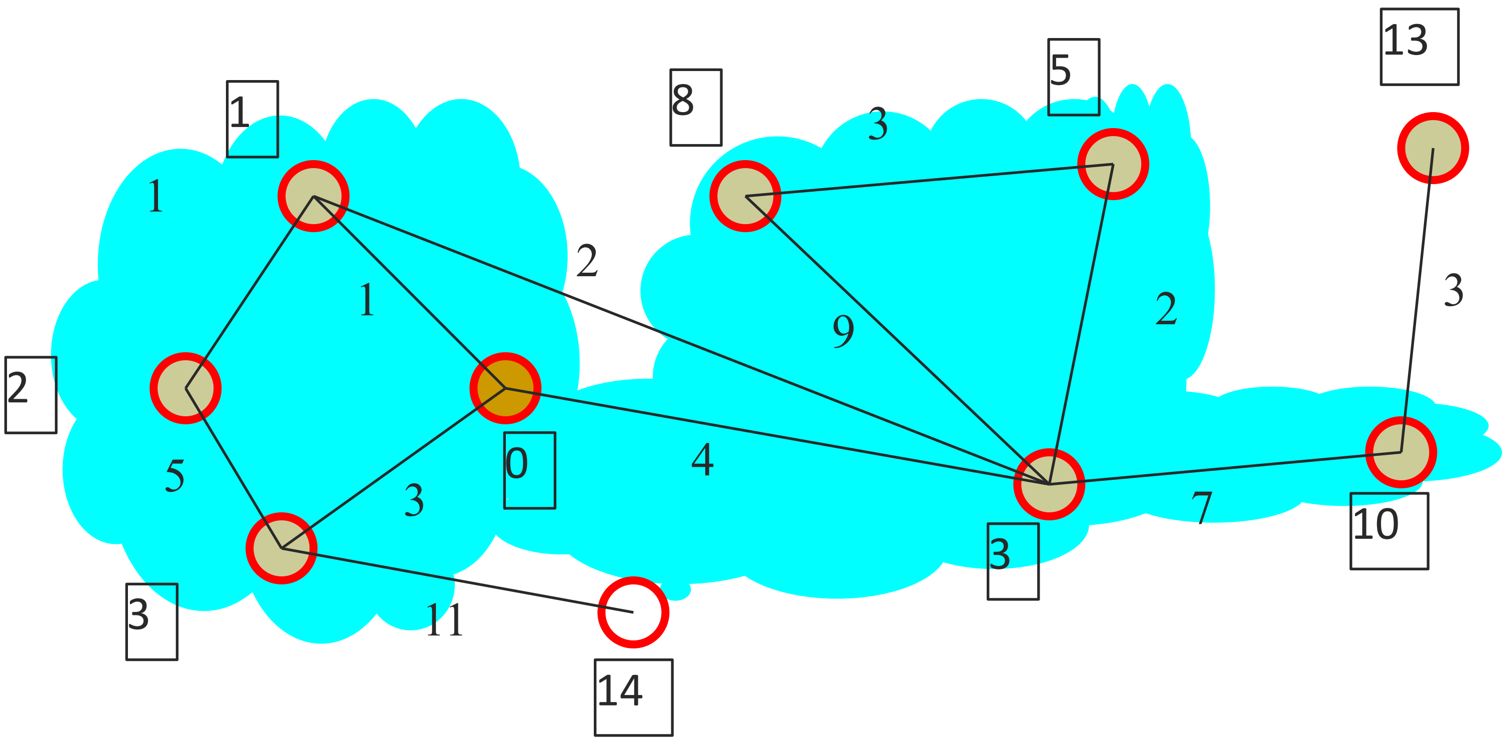
Dijkstra's algorithm: pull u into C



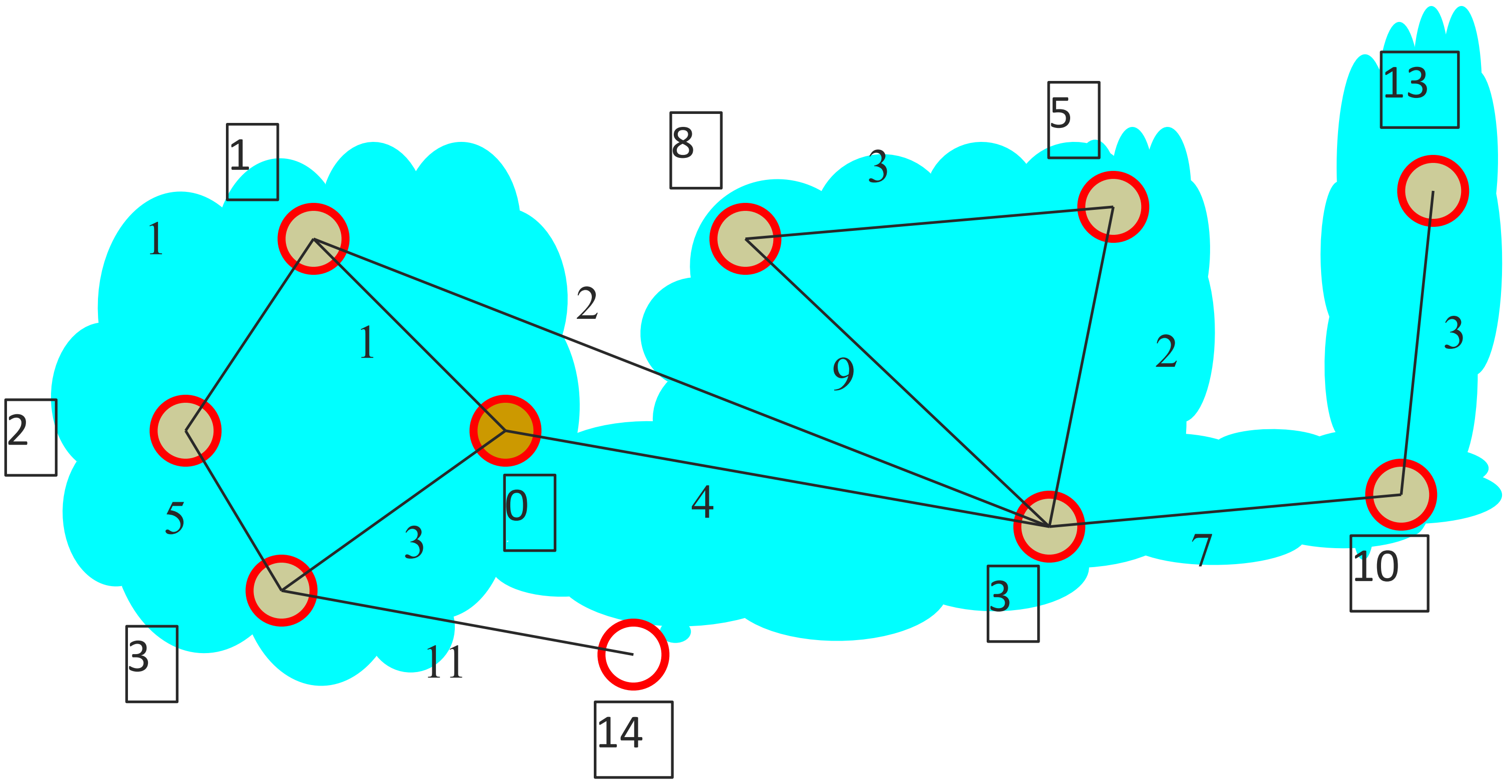
Dijkstra's algorithm: update *C's* neighborhood



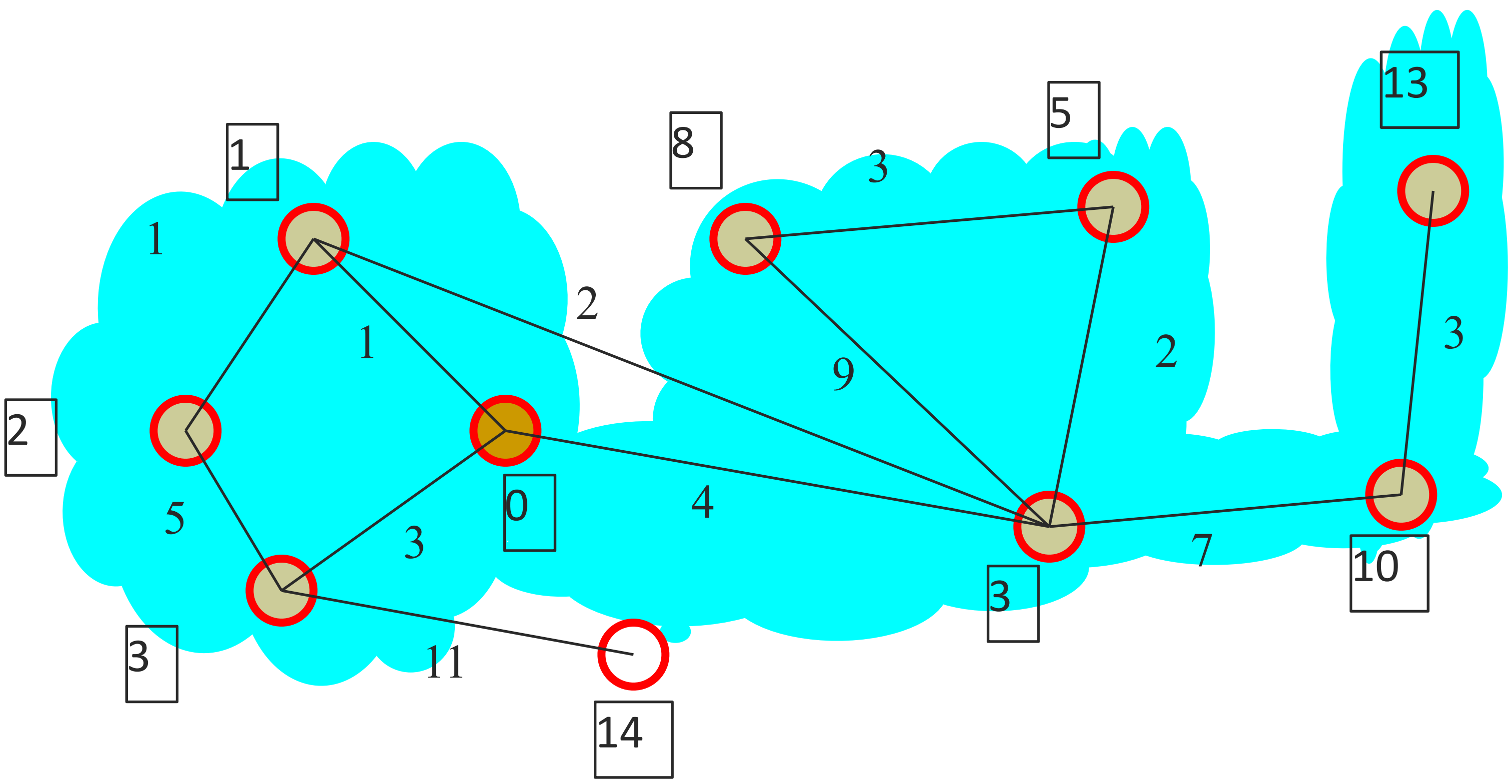
Dijkstra's algorithm: pick closest vertex u outside C



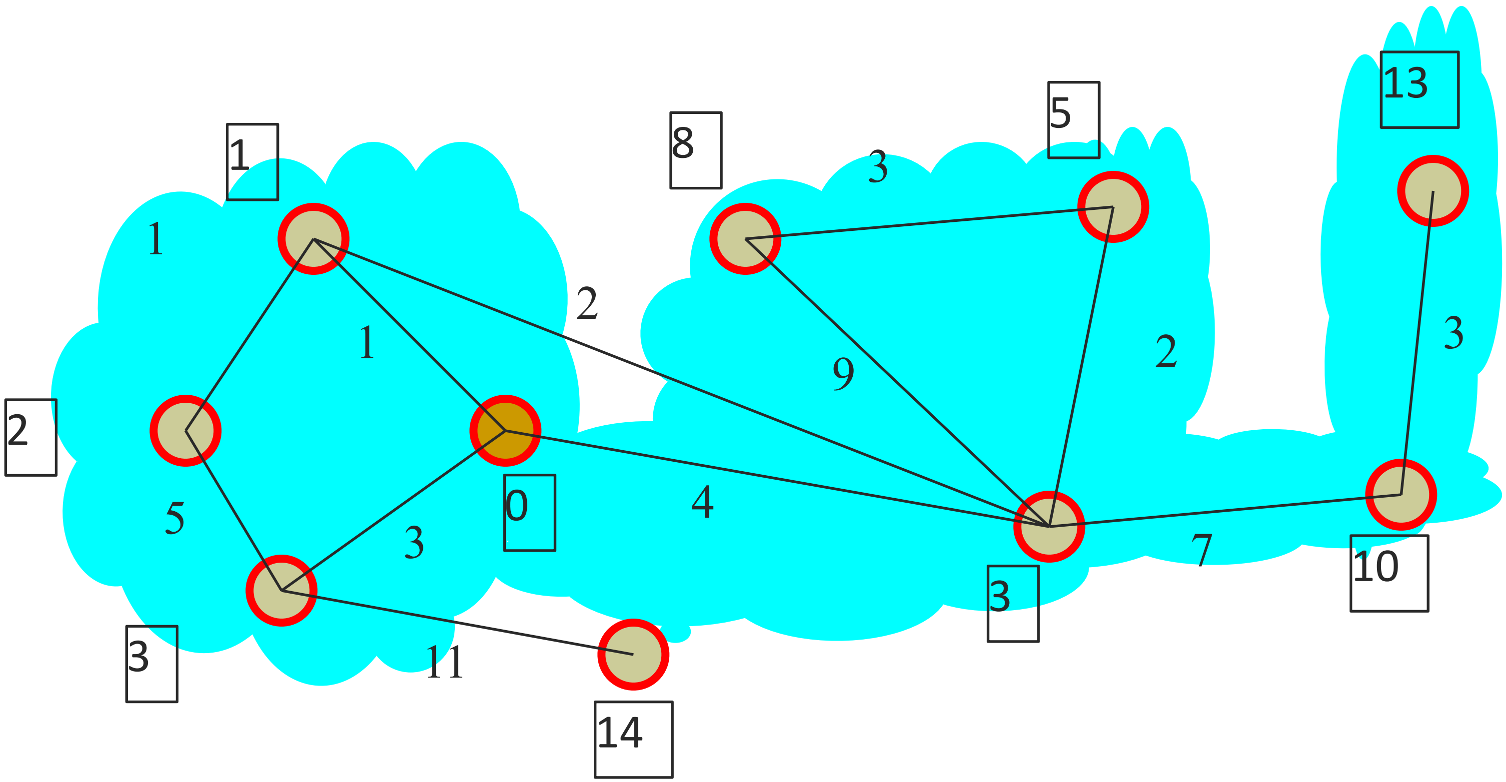
Dijkstra's algorithm: pull u into C



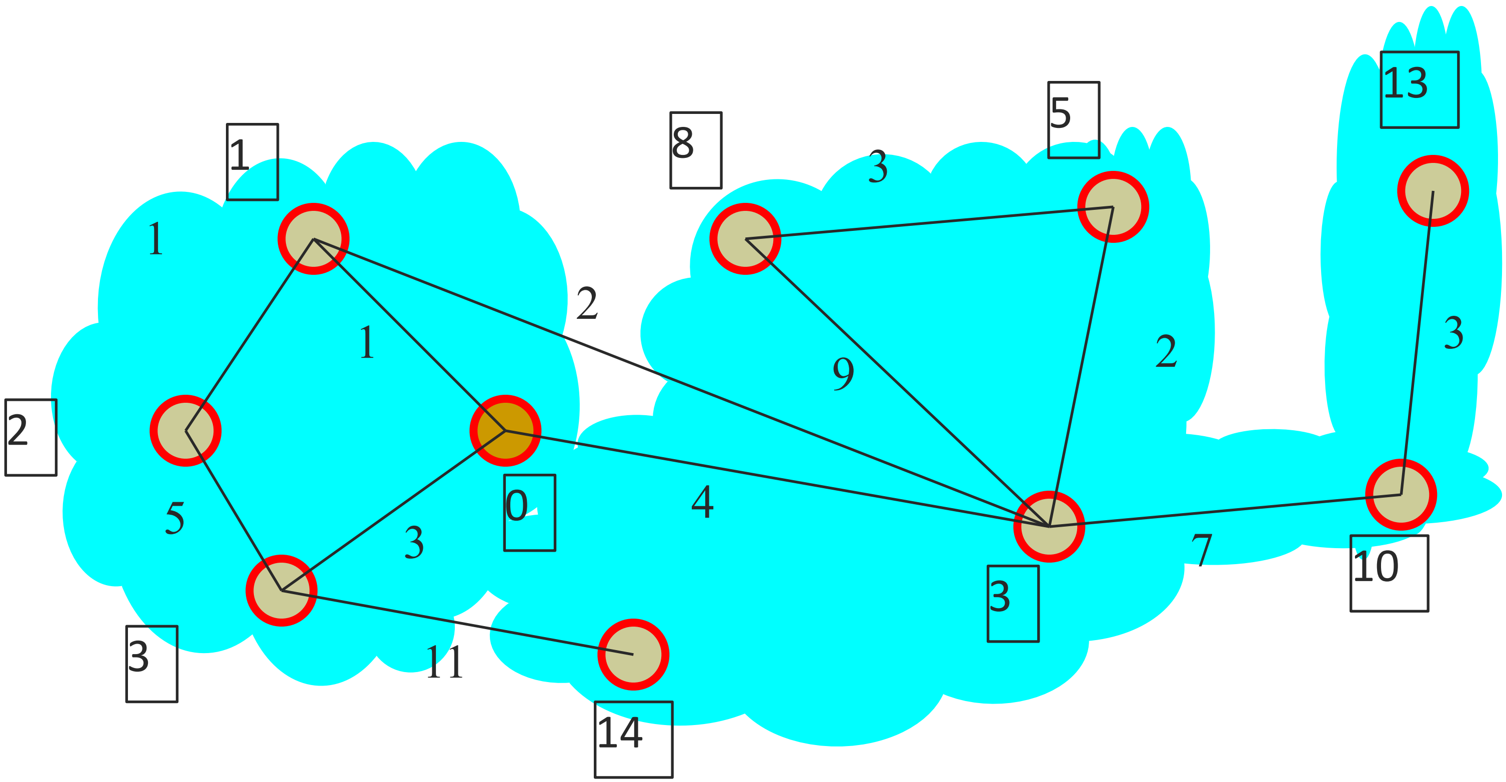
Dijkstra's algorithm: update *C's* neighborhood



Dijkstra’s algorithm: pick closest vertex u outside C



Dijkstra's algorithm: pull u into C

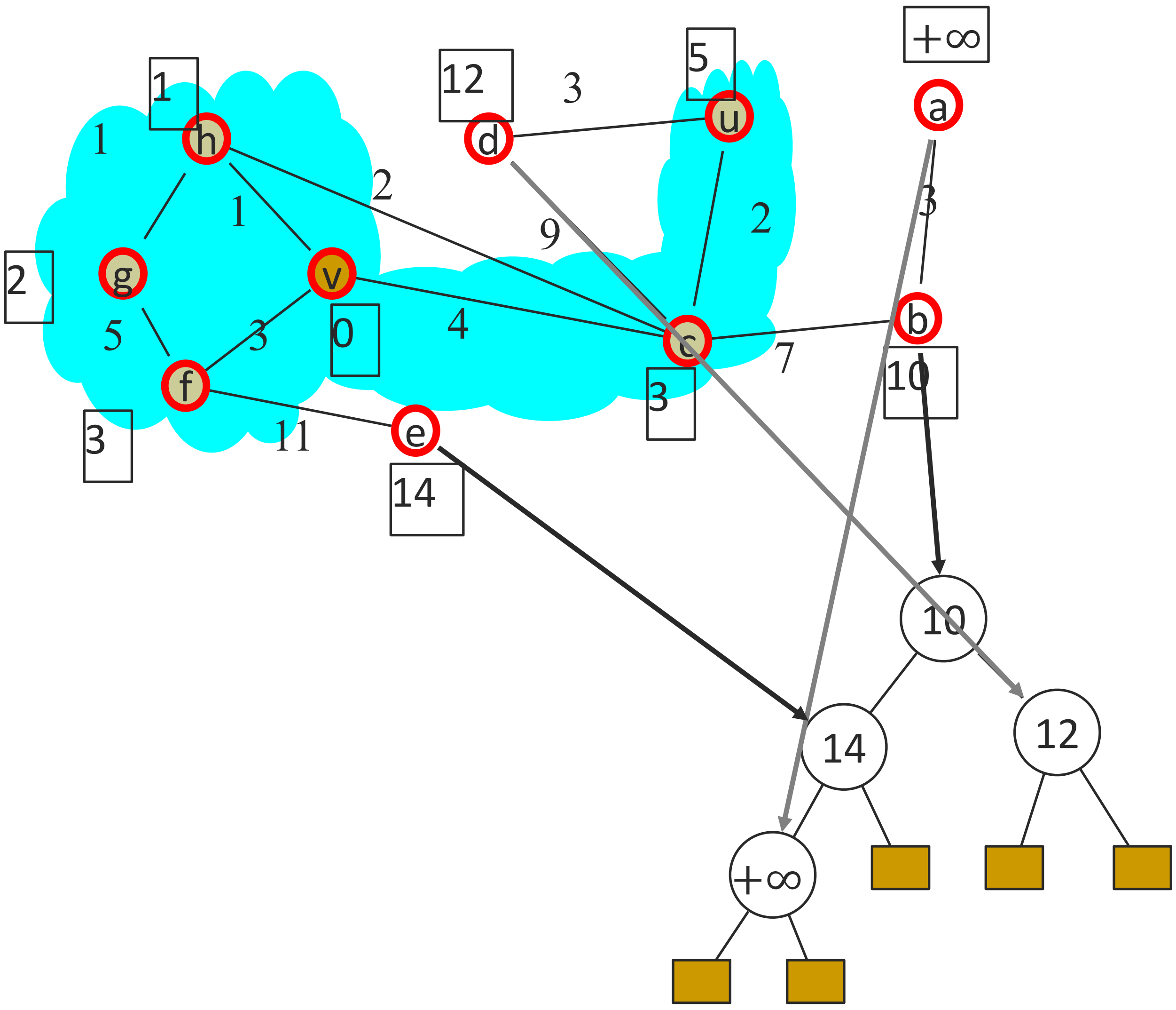


Algorithm
DijkstraShortestPaths(G,v)

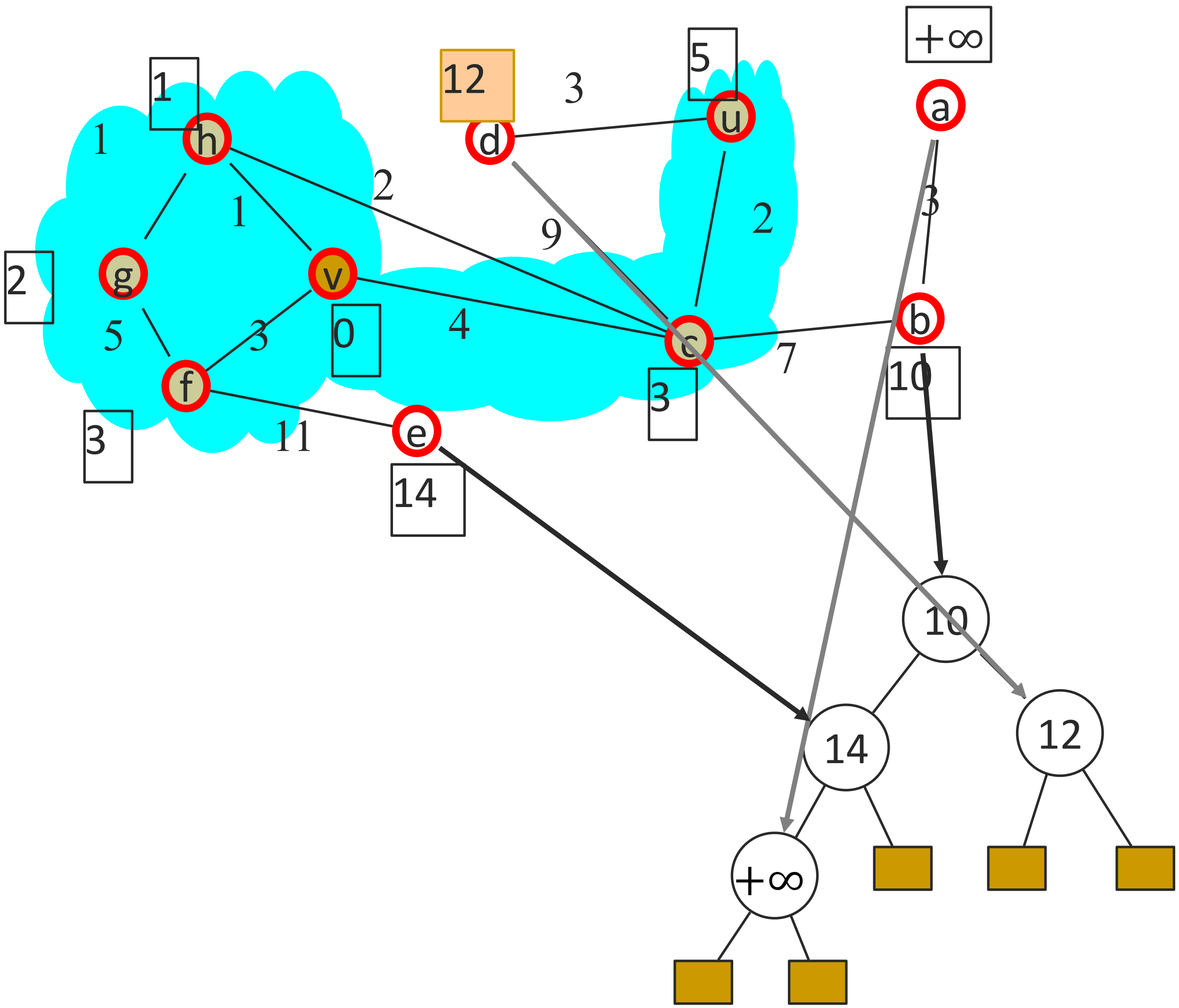
```
D[v]←0
for each vertex u≠v of G do
   D[u]← +∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to T
   for each vertex z∈N(u) with z∈Q do
      if D[u]+w((u,z)) < D[z] then
         D[z]←D[u]+w((u,z))
         update z's key in Q to D[z]
return D
```

Relaxation

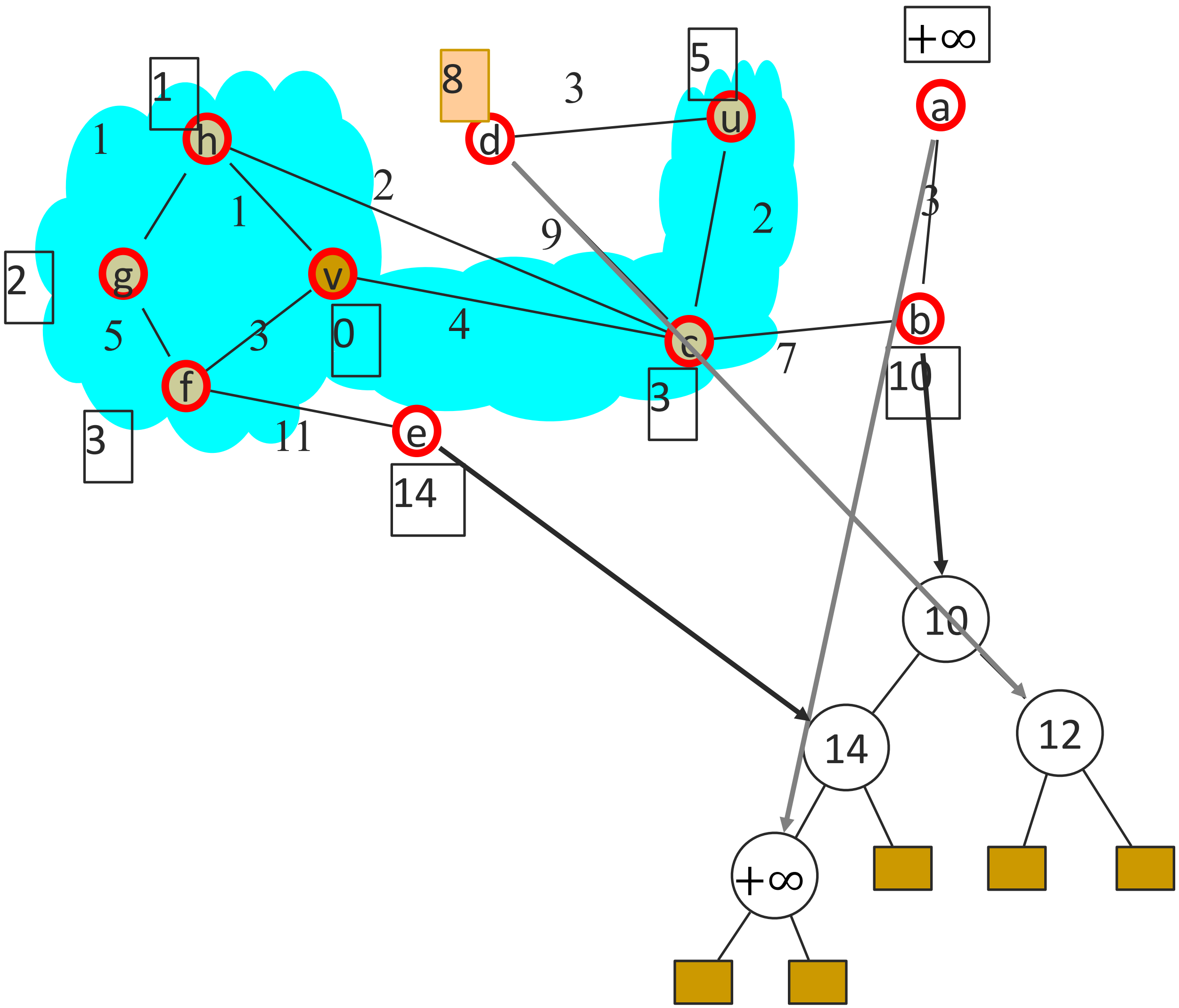
Dijkstra’s algorithm using heaps



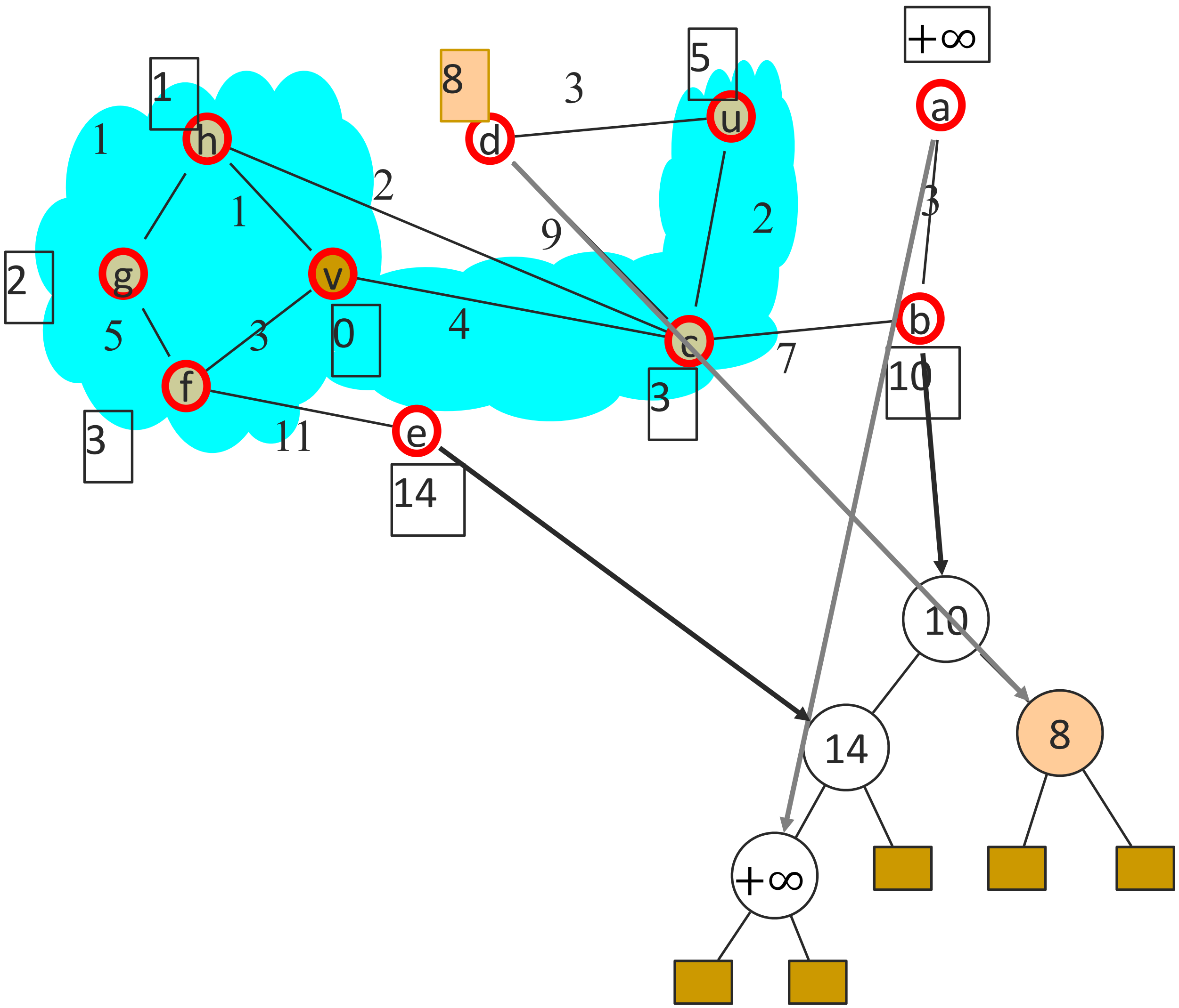
Dijkstra's algorithm using heaps



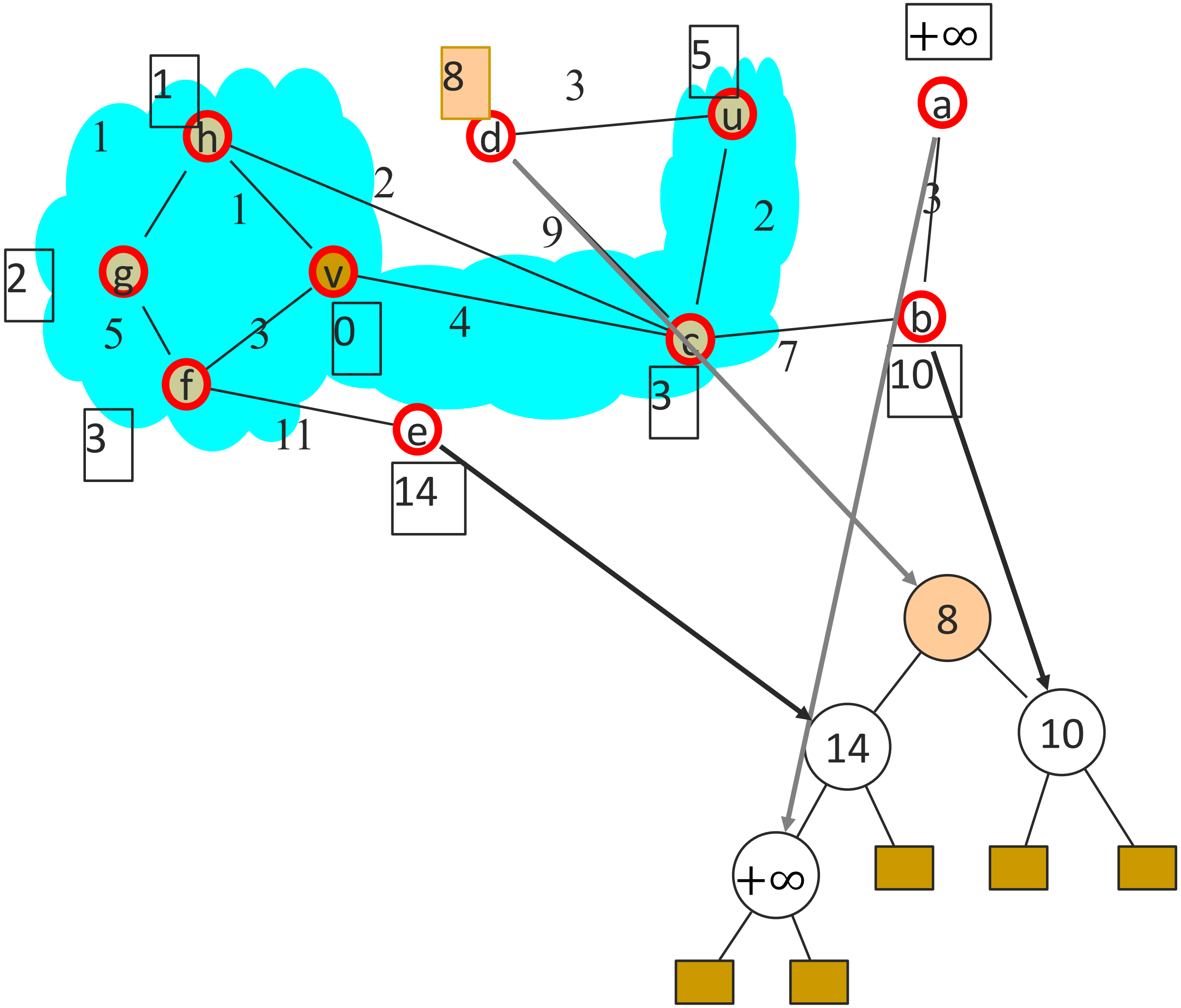
Dijkstra's algorithm using heaps



Dijkstra's algorithm using heaps



Dijkstra’s algorithm using heaps



Running time

```
D[v]←0
for each vertex u≠v of G do
   D[u]← +∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to cloud
   for each vertex z∈N(u) with z∈Q do
      if D[u]+w((u,z)) < D[z] then
         D[z]←D[u]+w((u,z))
         update z's key in Q to D[z]
return D
```

Relaxation

Running time for G=(V,E) with $|V|$=n and $|E|$=m

- Insertion of vertices in priority queue Q
  - $O(n)$ when using bottom-up heap construction
- While loop:
  - Per iteration:
    - Remove vertex from Q $O(\log n)$
    - Relaxation $O(\deg(u) \log(n))$
  - $\sum_{u \in G}(1 + \deg(u)) \log n$ is $O((n + m)\log n)$
- Overall running time: $O(m \log n)$

In real life applications

- Often the graphs are <u>sparse</u>
- Then $O(m \log n)$ may be $O(n \log n)$

Algorithm DijkstraShortestPaths(G,v)

Correctness

```
D[v]←0
for each vertex u≠v of G do
   D[u]← +∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to T
   for each vertex z∈adj(u) with z∈Q do
      if D[u]+w((u,z)) < D[z] then
         D[z]←D[u]+w((u,z))
         update z's key in Q to D[z]
return D
```

Relaxation

Invariants

Let d(u)=distance of *u* from *v*

1. For each node u *in* T, $D[u] =$ d(u)
2. For each node *u* not in T, *D[u]*= length of shortest path from v to u without the use of other nodes outside of T

   $+\infty$ denotes that the node cannot be reached yet from v via T nodes only

→ For all nodes u *in* T, $D[u] \geq$ d(u)

Proof that invariants hold:

Initially they hold:
$D(v)=0$, $D(u)=\infty$

Consider the first time they fail:
CASE A: If Invariant 1 holds, i.e, $D(u)=d(u)$ for all u in T , but Invariant 2 fails for some z outside T when we add u to T, i.e, $D(z)$ fails to be the length of the shortest path from v to z passing only through nodes in T.

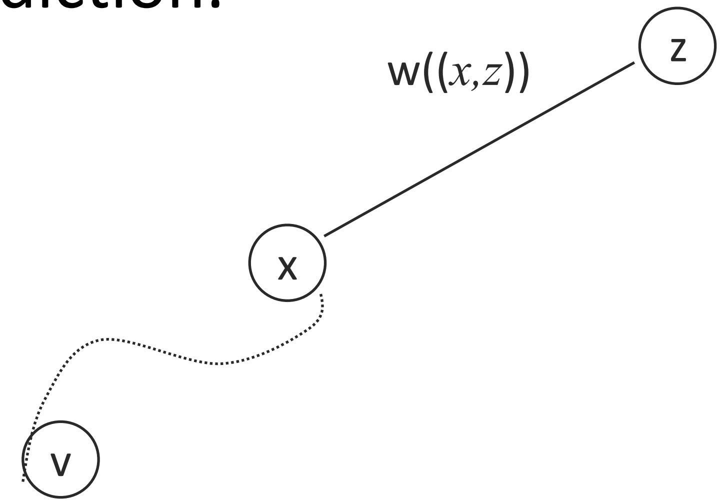
<u>Claim:</u> This can only happen if there is a shorter path of length $d'<D(z)$ from v to z whose last edge is $\{u,z\}$:
Why?

Claim: Assuming Invariant I holds, Invariant 2 is violated only if there is a shorter path from v to z whose last edge is {u,z}: Why?

Proof of Claim: Suppose to the contrary, the last edge of the shorter path is {x,z}, x≠u, x in T.

We would have previously set D(z) to D(x)+w(x,z) when we relaxed x.

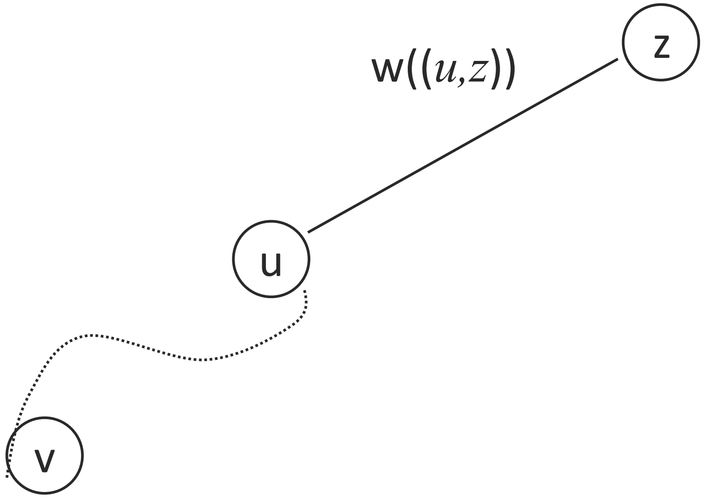
By Invariant 1, D(x)=d(x) since x is in T, so D(z) would have been set to d(x)+w(x,z)=d' < D(z) giving a contradiction.



Since u is the last node in the shortest path to z passing through only T, then
the relaxation step correctly sets

- $D(z) = d(v,u) + w((u,z))$ so Invariant 2 is NOT violated

after the relaxation with respect to u is completed.



Proof that invariants hold (cont'd):

<u>CASE B: </u>(Invariant 1 fails first)

Consider the first time Invariant 1 fails:

u is added to T, D(u) is the length of the shortest path from u to v through T but D(u) ≠ d(u)

Let P =shortest path from v to u

- Note: For all u, $D(u) \geq d(u)$
- Let y =last node in T on path P
- By Invariant 1, $D[y] = d(y)$

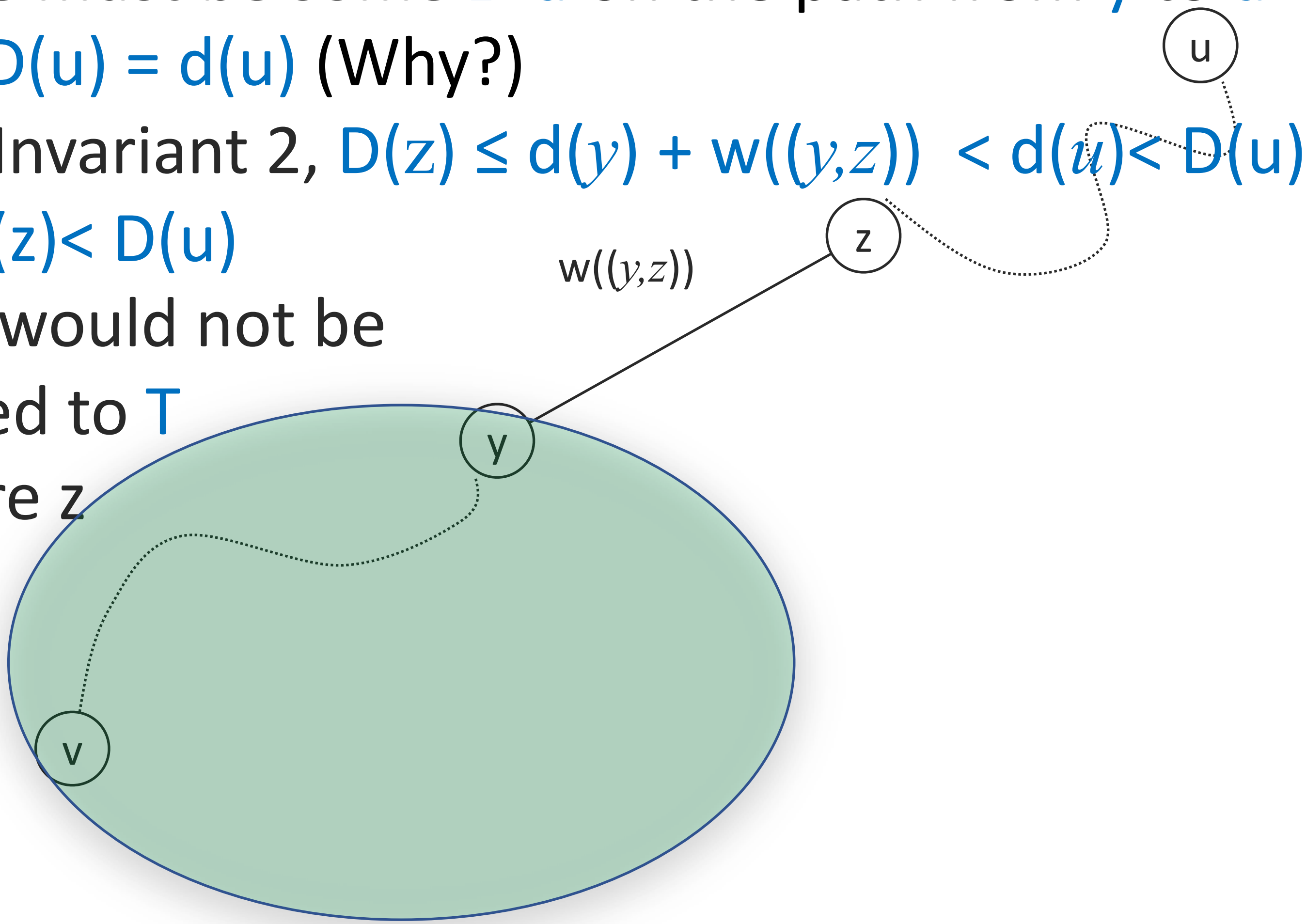
There must be some $z \neq u$ on the path from y to u

Else $D(u) = d(u)$ (Why?)

- By Invariant 2, $D(z) \leq d(y) + w((y,z)) < d(u) < D(u)$

→ $D(z) < D(u)$

→ u would not be added to T before z



-

We've just show both invariants always hold

Let $d(u)$=distance of u from v

1. For each node u *in* T, $D[u] = d(u)$

2. For each node u not in T, D[u]= length of shortest path from v to u without the use of other nodes outside of T

$+\infty$ denotes that the node cannot be reached yet from v via T nodes only

→ When the algorithm is finished, all nodes are in T and all D[u]=d(u)