CSC 226

Algorithms and Data Structures: II

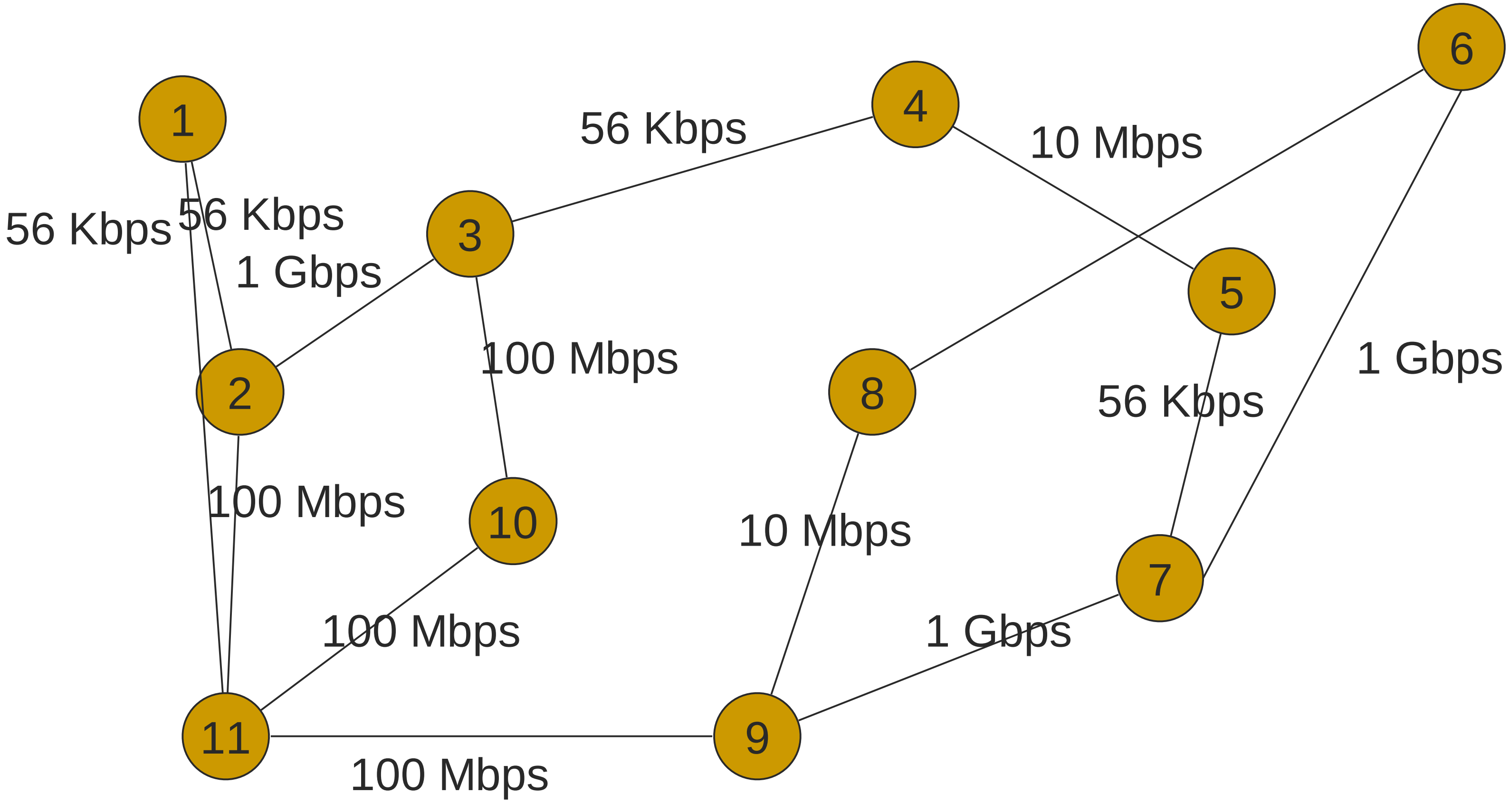
Rich Little
rlittle@uvic.ca
ECS 516

Shortest Paths

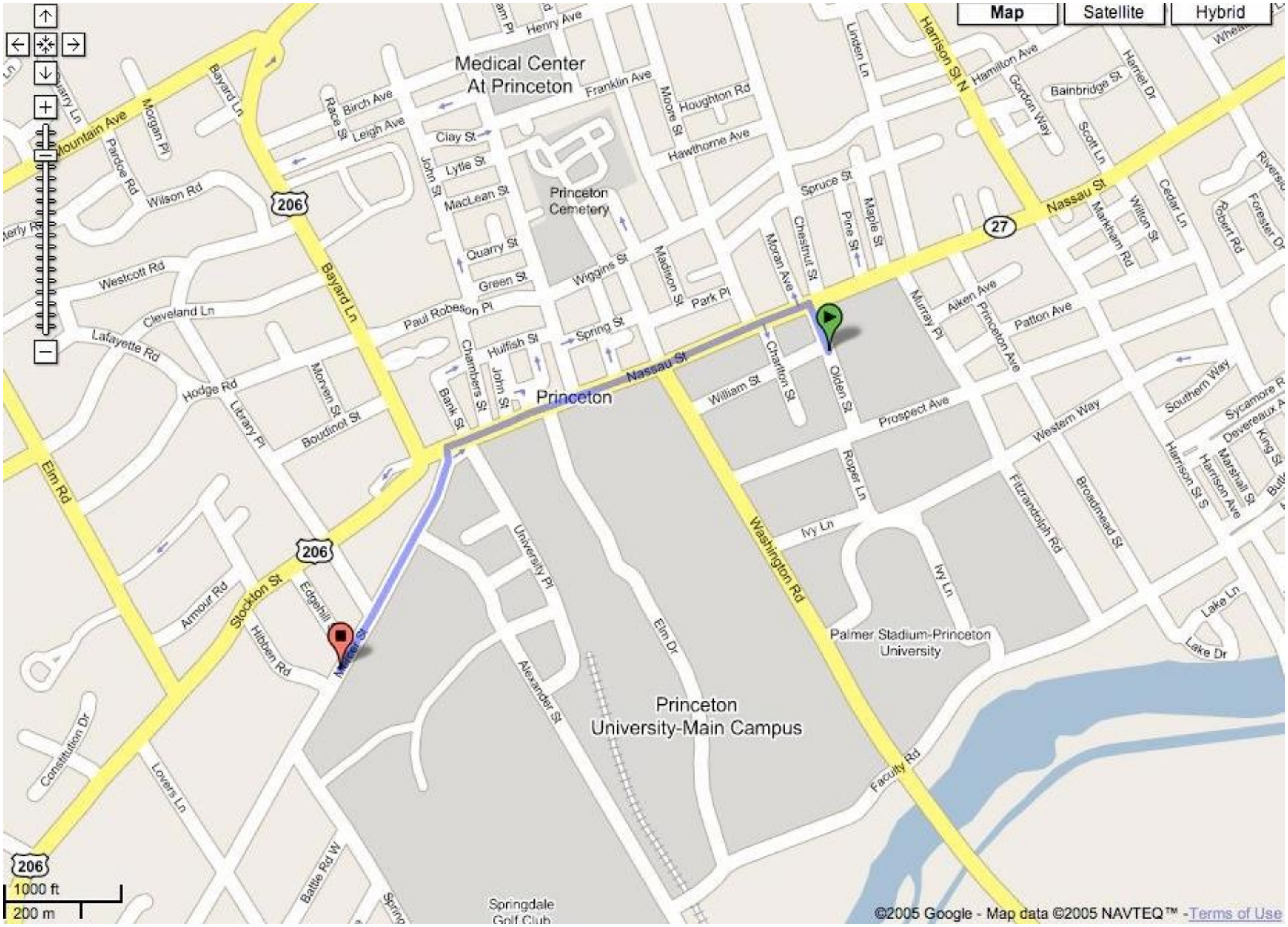
in edge-weighted graphs

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
Bayard Ln
Mountain Ave
Stockton St
Elm Rd
Mercer St
Olden St
Prospect Ave
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

- 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, … , 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).

Algorithm
`DijkstraShortestPaths(G,v)`

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

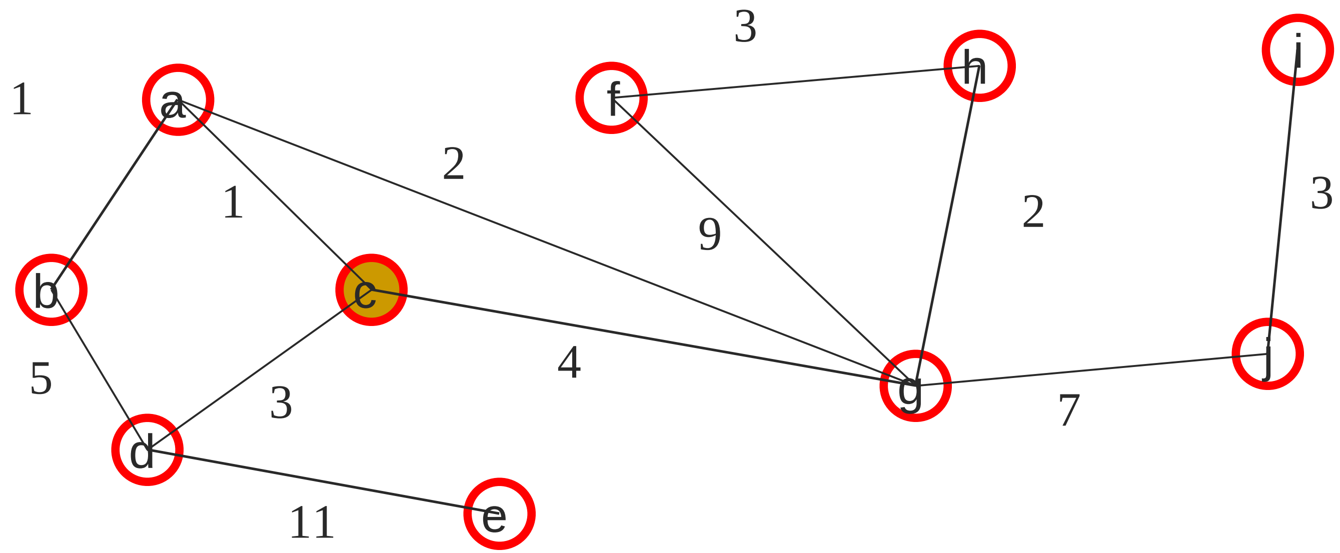
Output: A label D[u] for each vertex u in G such that D[u] is the shortest distance from v to u in G.

Algorithm
DijkstraShortestPaths(G,v)

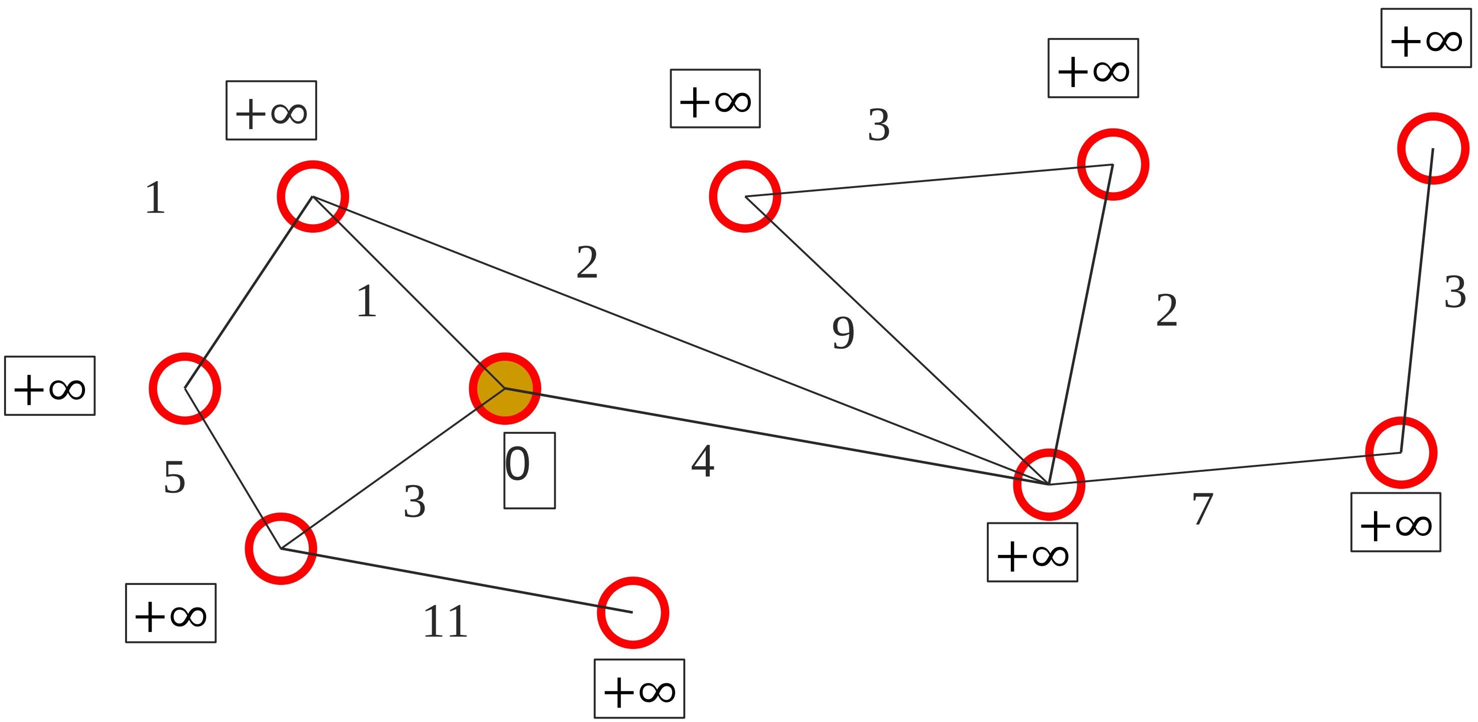
```
D[v]←0
for each vertex u≠v of G do
   D[u]← +∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to cloud
   for each vertex z∈N(u) with z∈Q do
     if D[u]+w((u,z)) < D[z] then
        D[z]←D[u]+w((u,z))
        update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

Relaxation

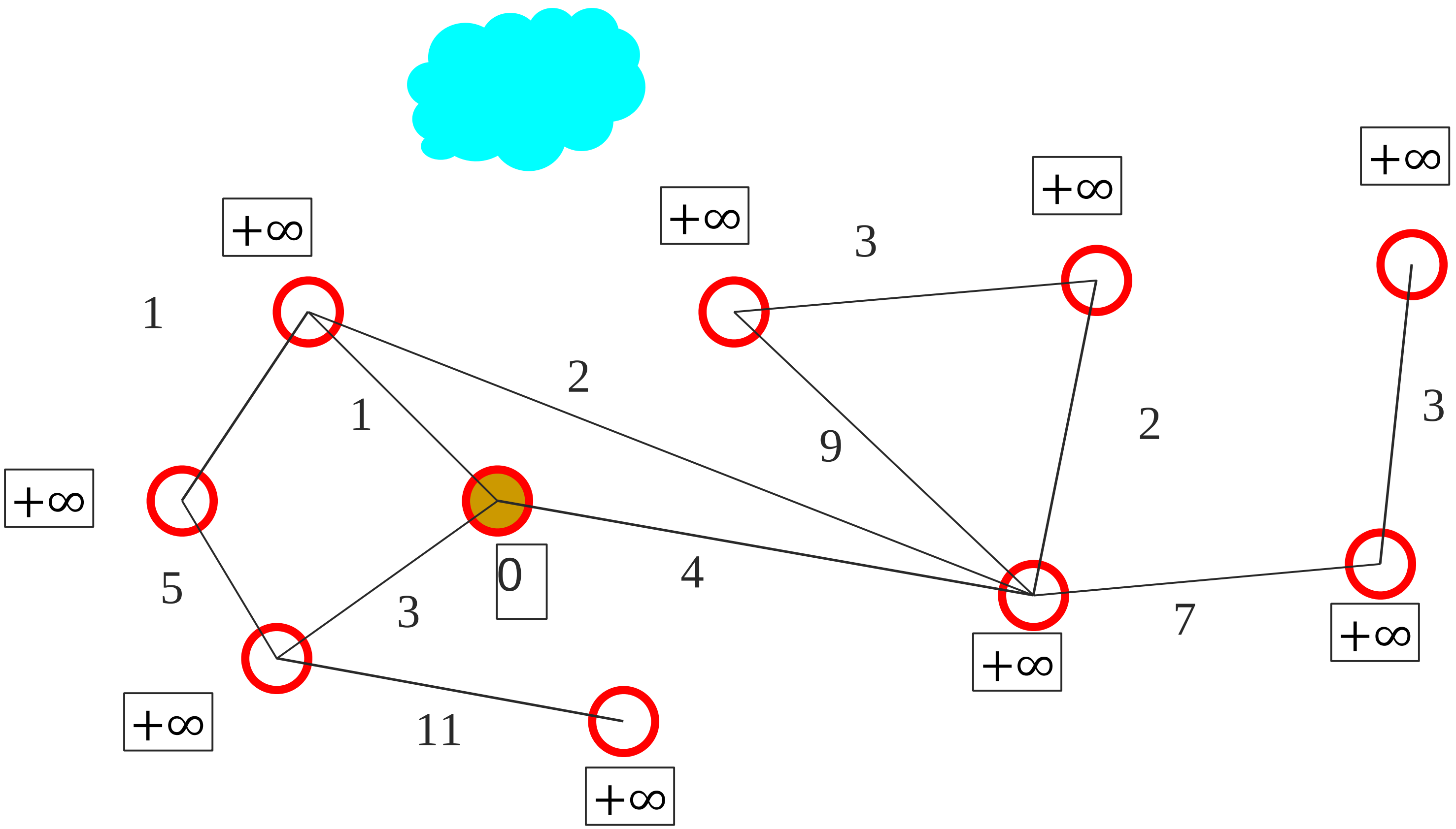
Dijkstra’s algorithm: a greedy algorithm



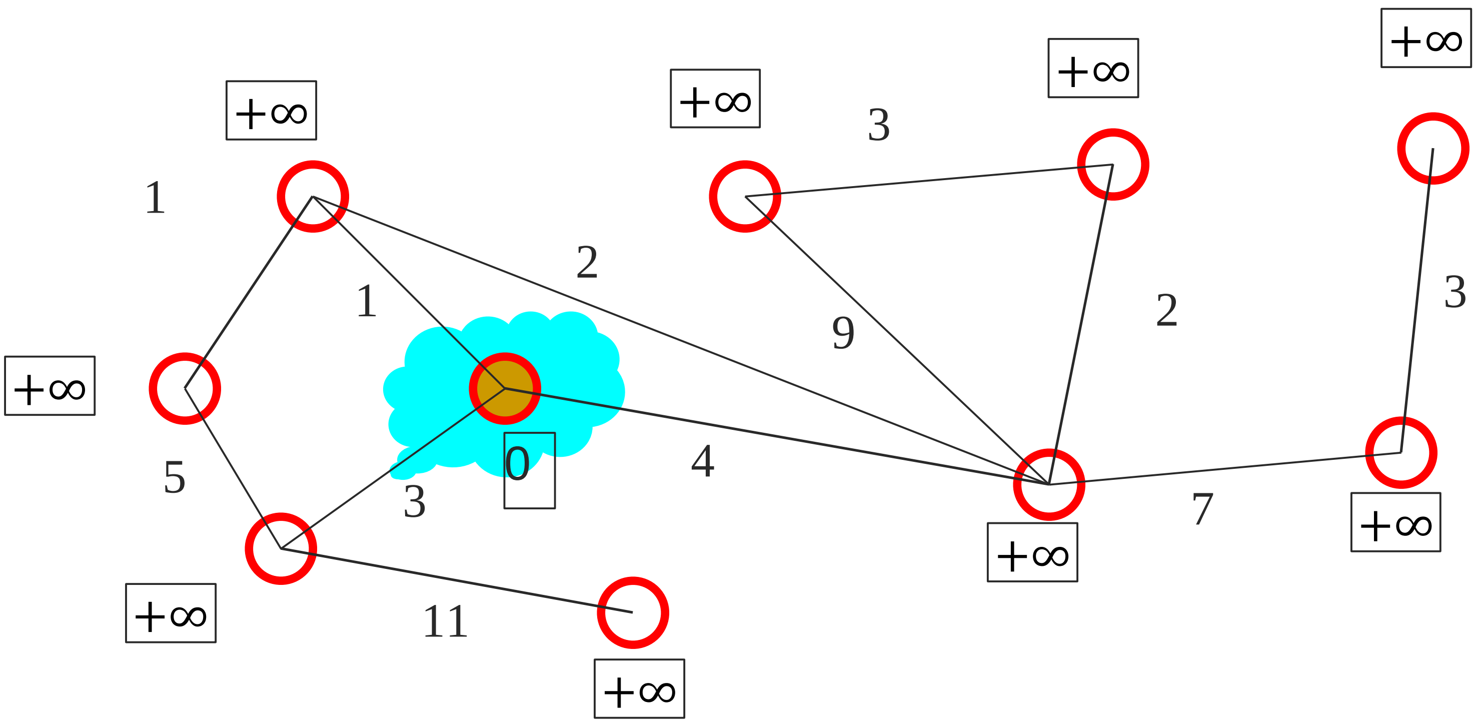
Dijkstra's algorithm: Initializing



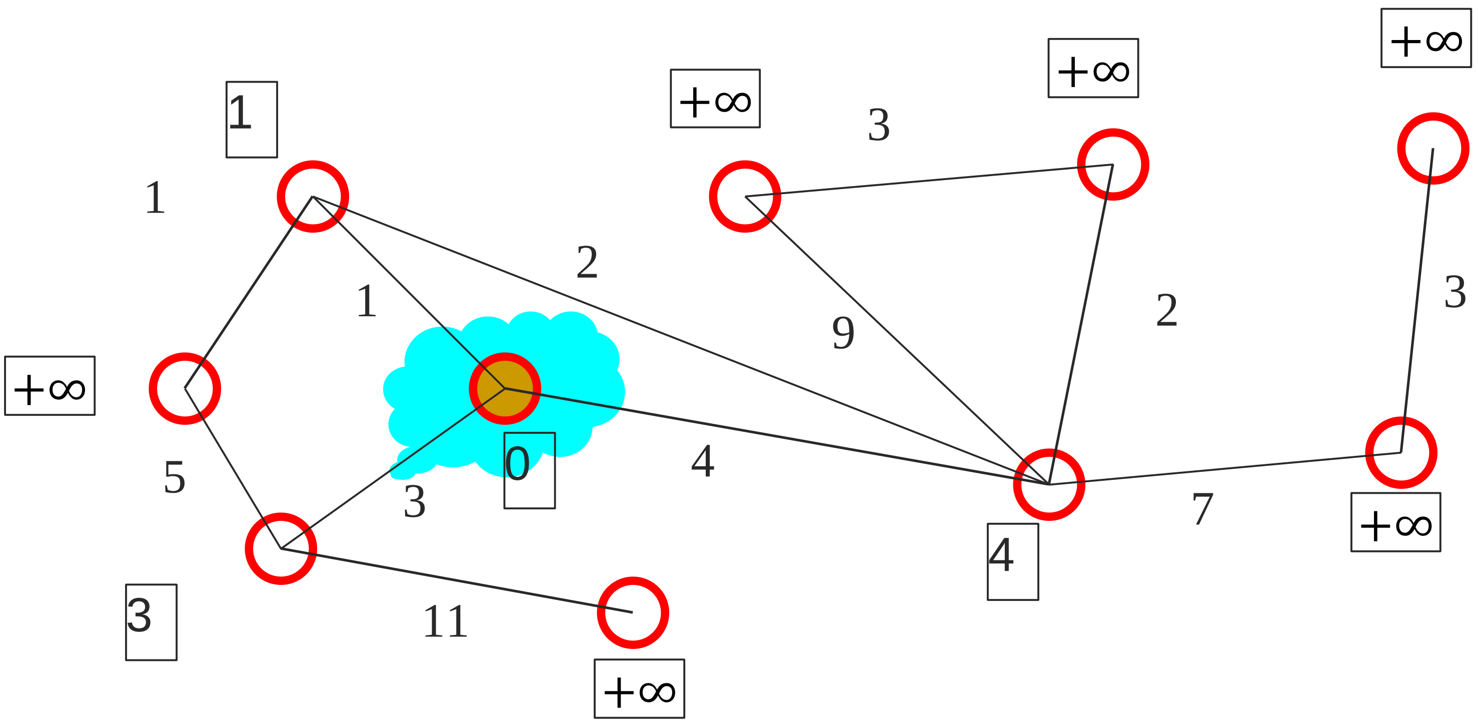
Dijkstra's algorithm: Initializing Cloud C (consisting of "solved" subgraph)



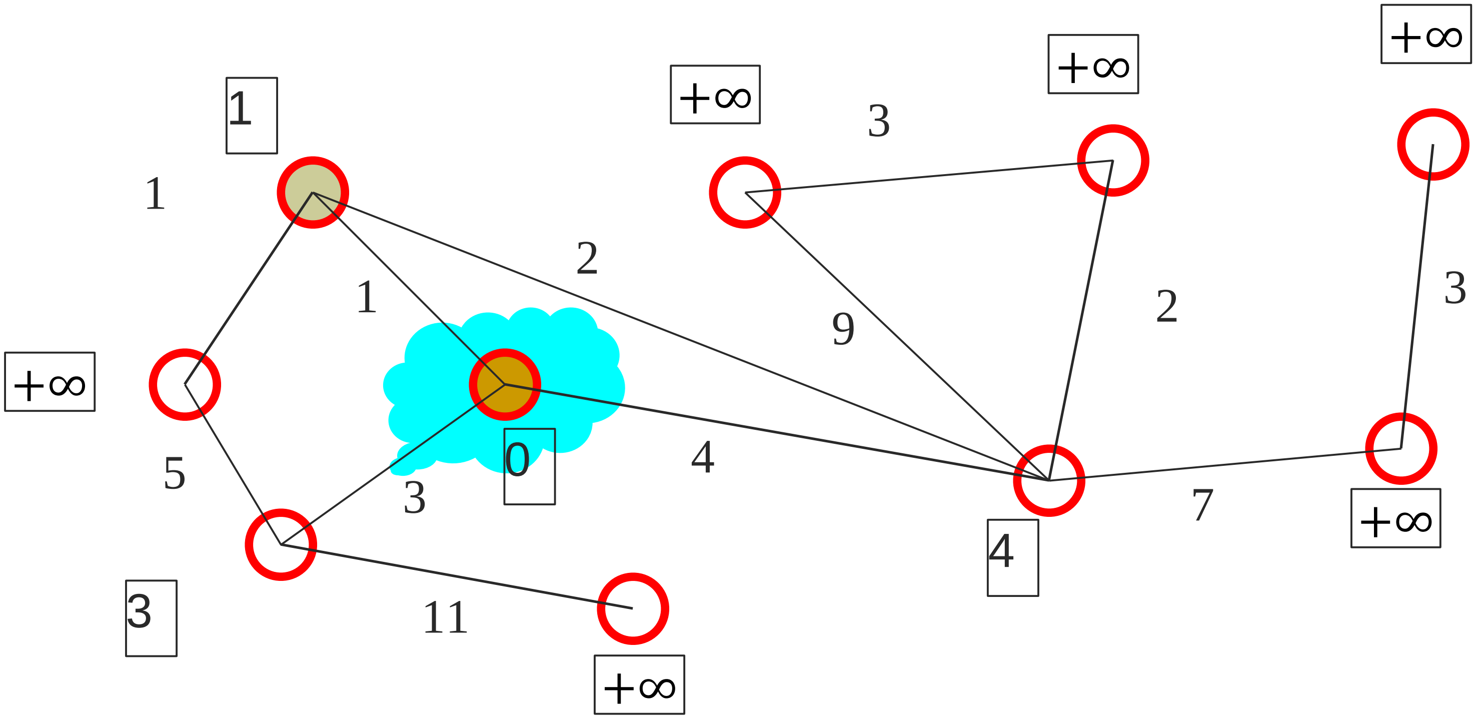
Dijkstra's algorithm: pull v into C



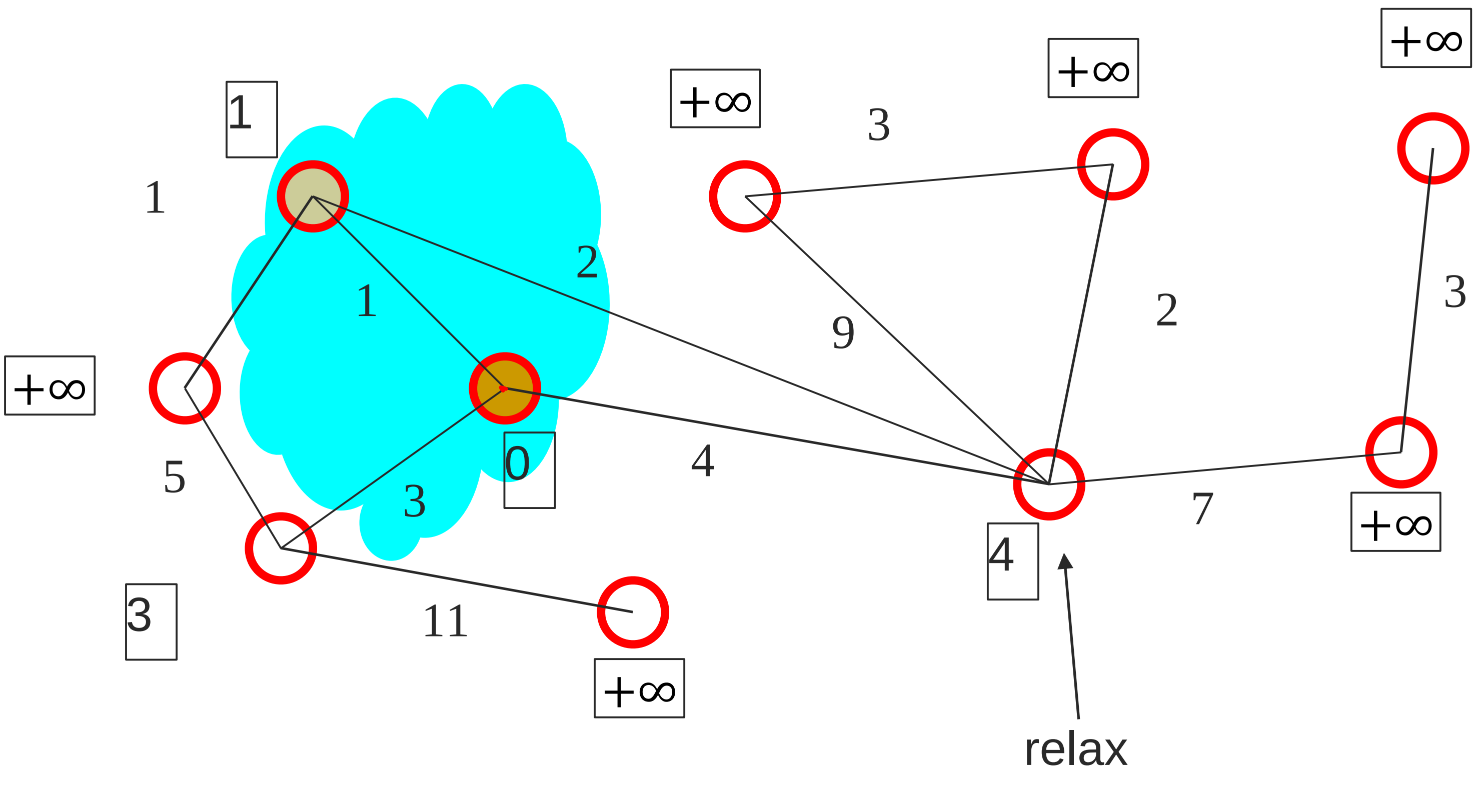
Dijkstra's algorithm: update *C's* neighborhood



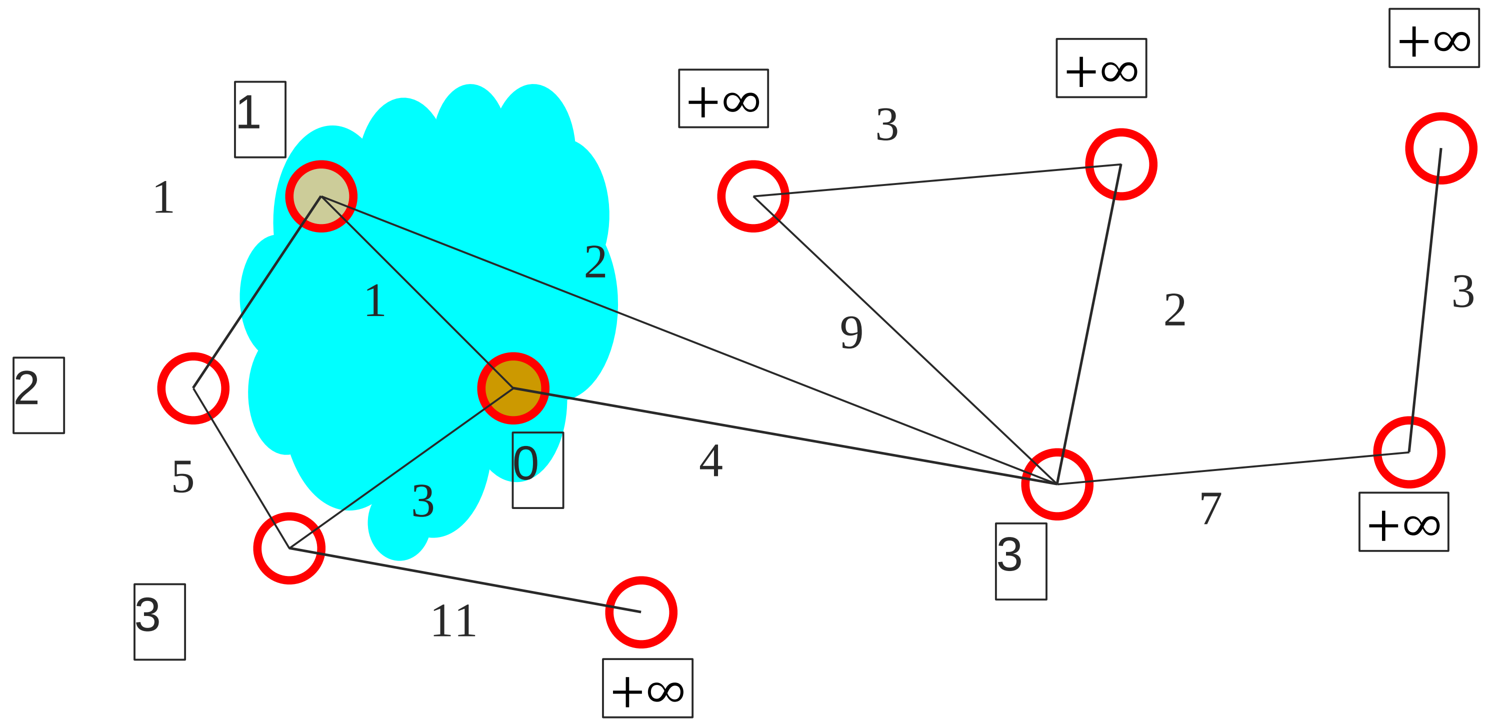
Dijkstra's algorithm: pick closest vertex u outside C



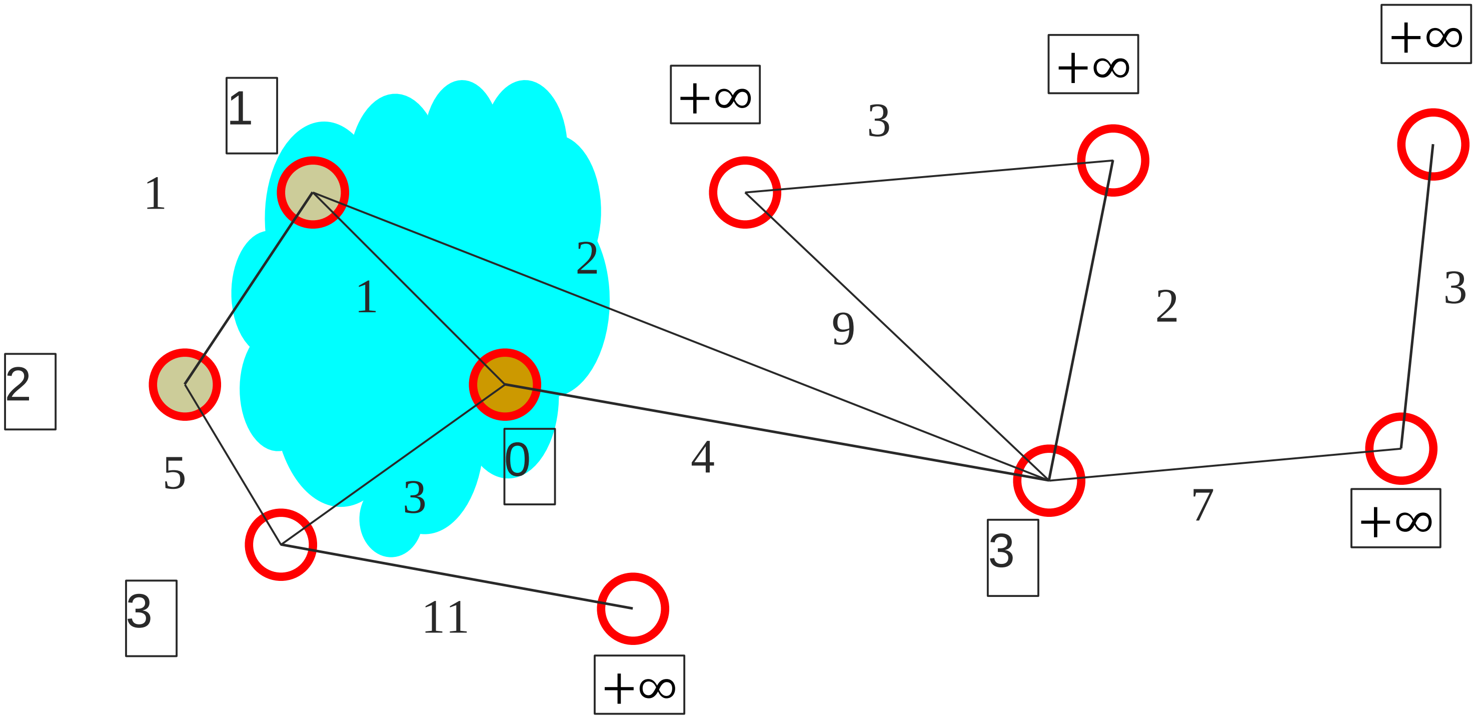
Dijkstra's algorithm: pull u into C



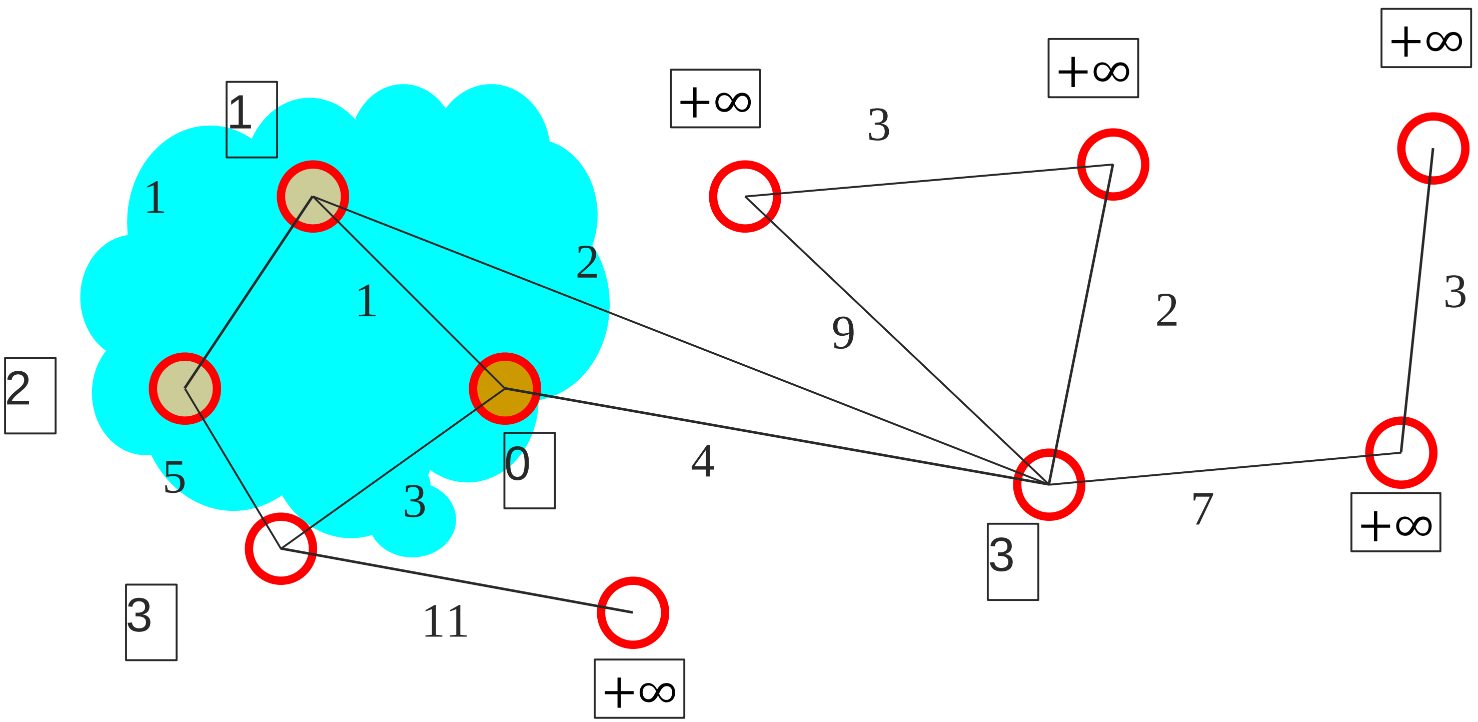
Dijkstra's algorithm: update *C's* neighborhood



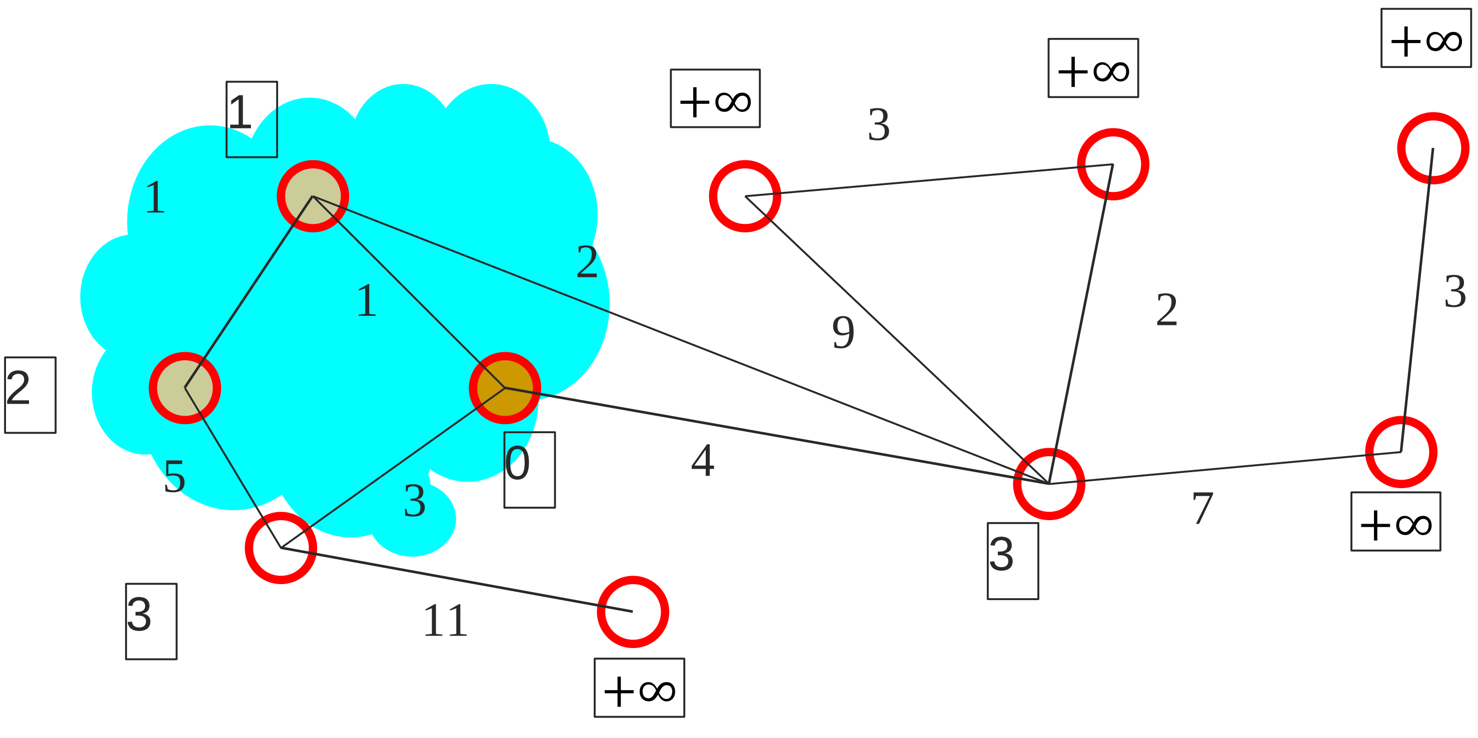
Dijkstra's algorithm: pick closest vertex u outside C



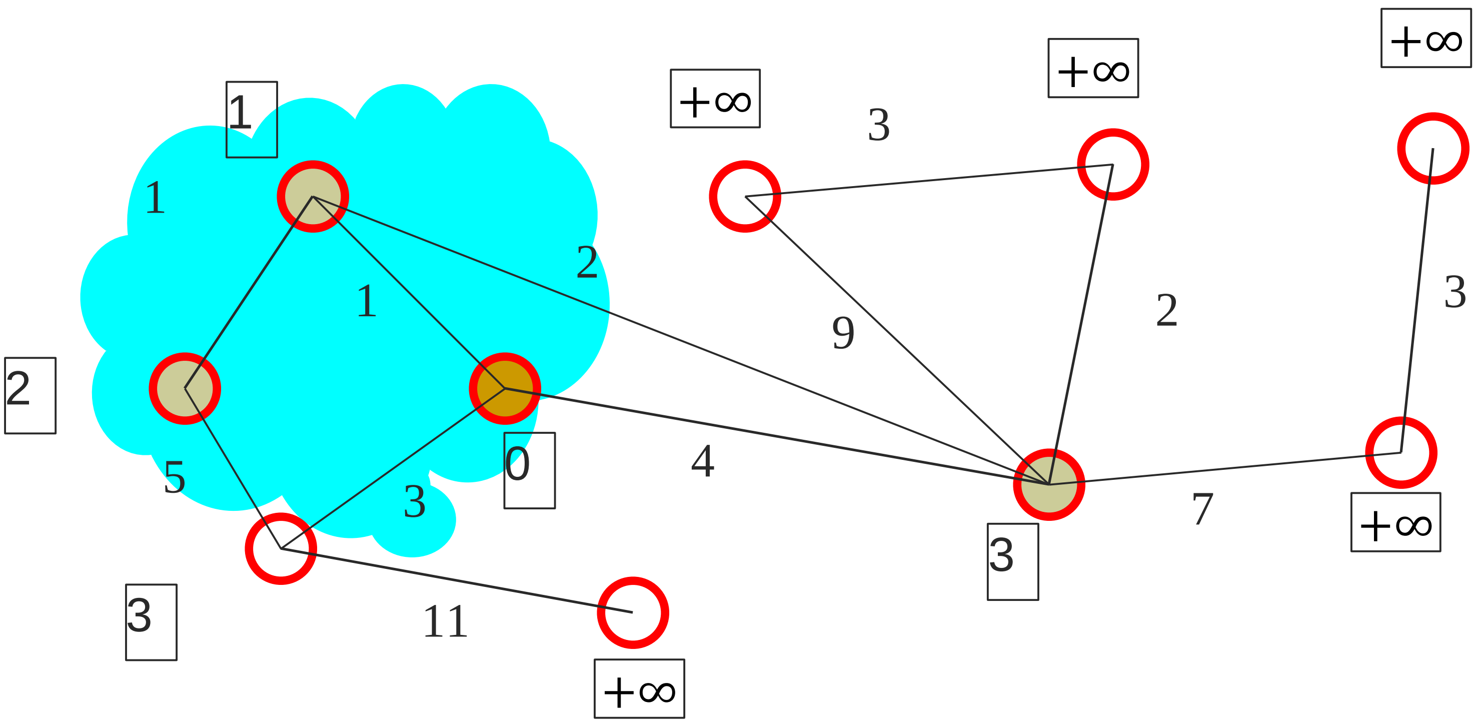
Dijkstra's algorithm: pull u into C



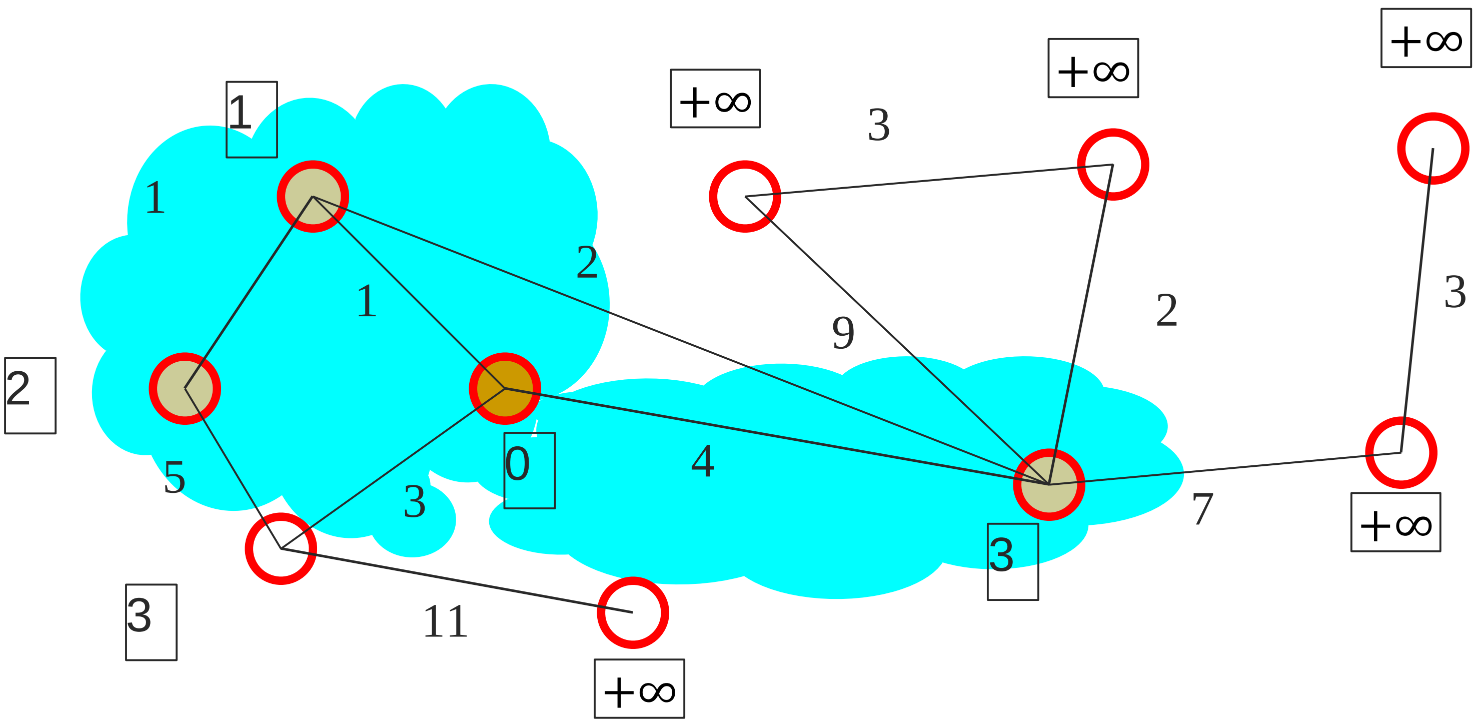
Dijkstra’s algorithm: update *C’s* neighborhood



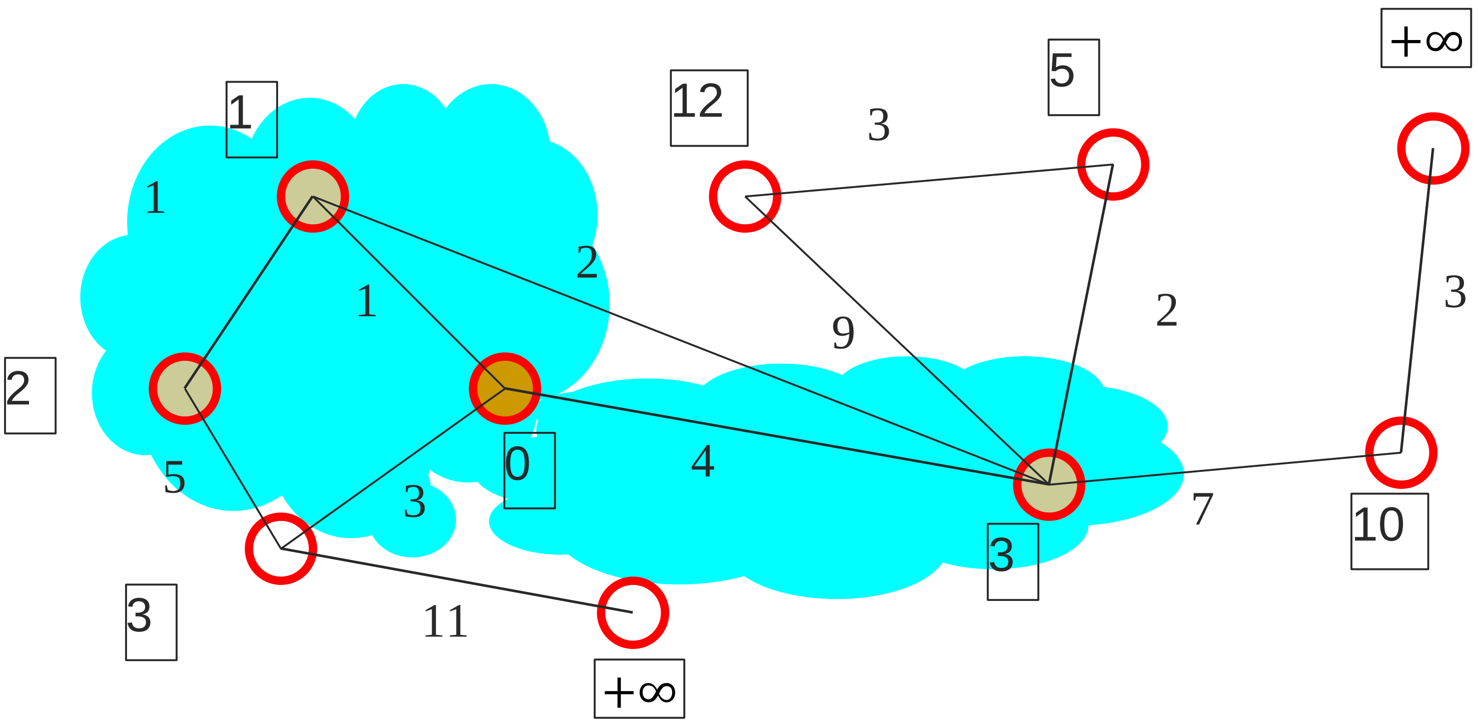
Dijkstra's algorithm: pick closest vertex u outside C



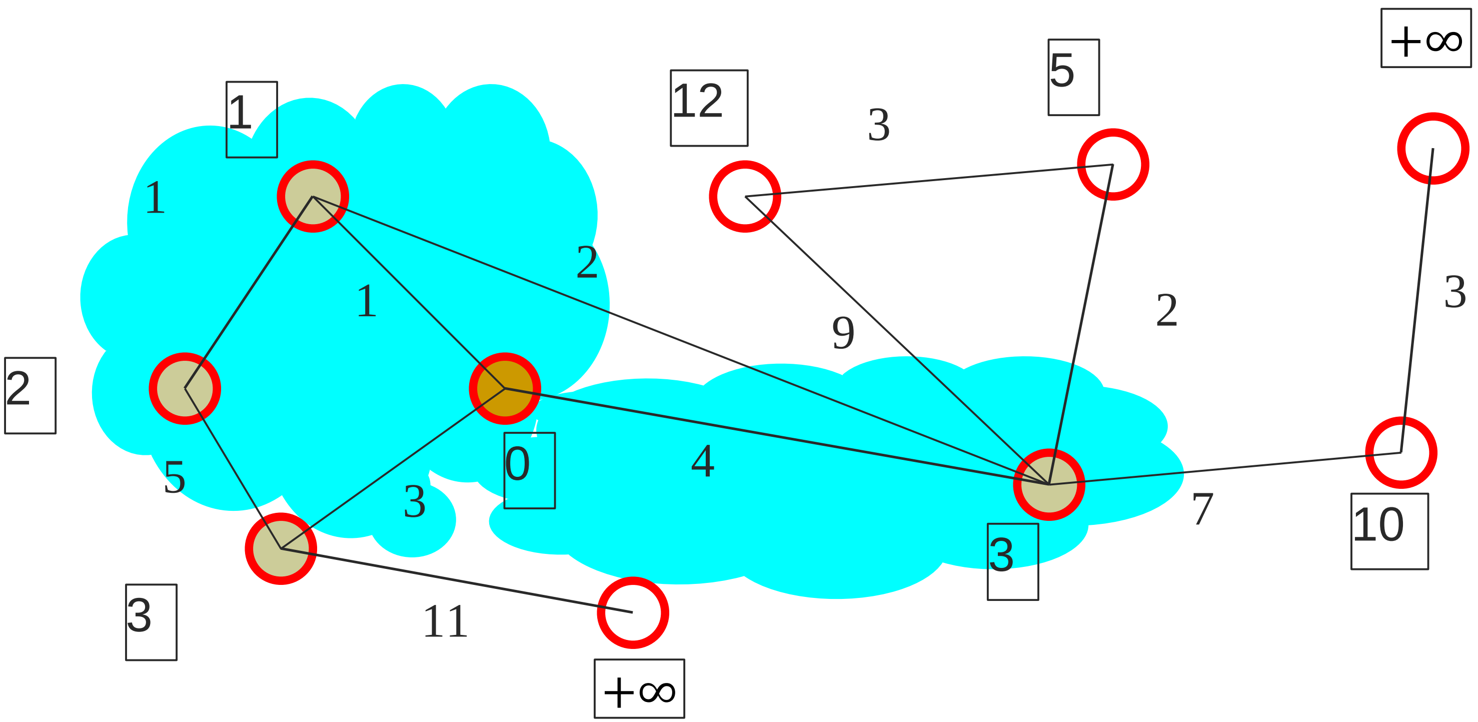
Dijkstra's algorithm: pull u into C



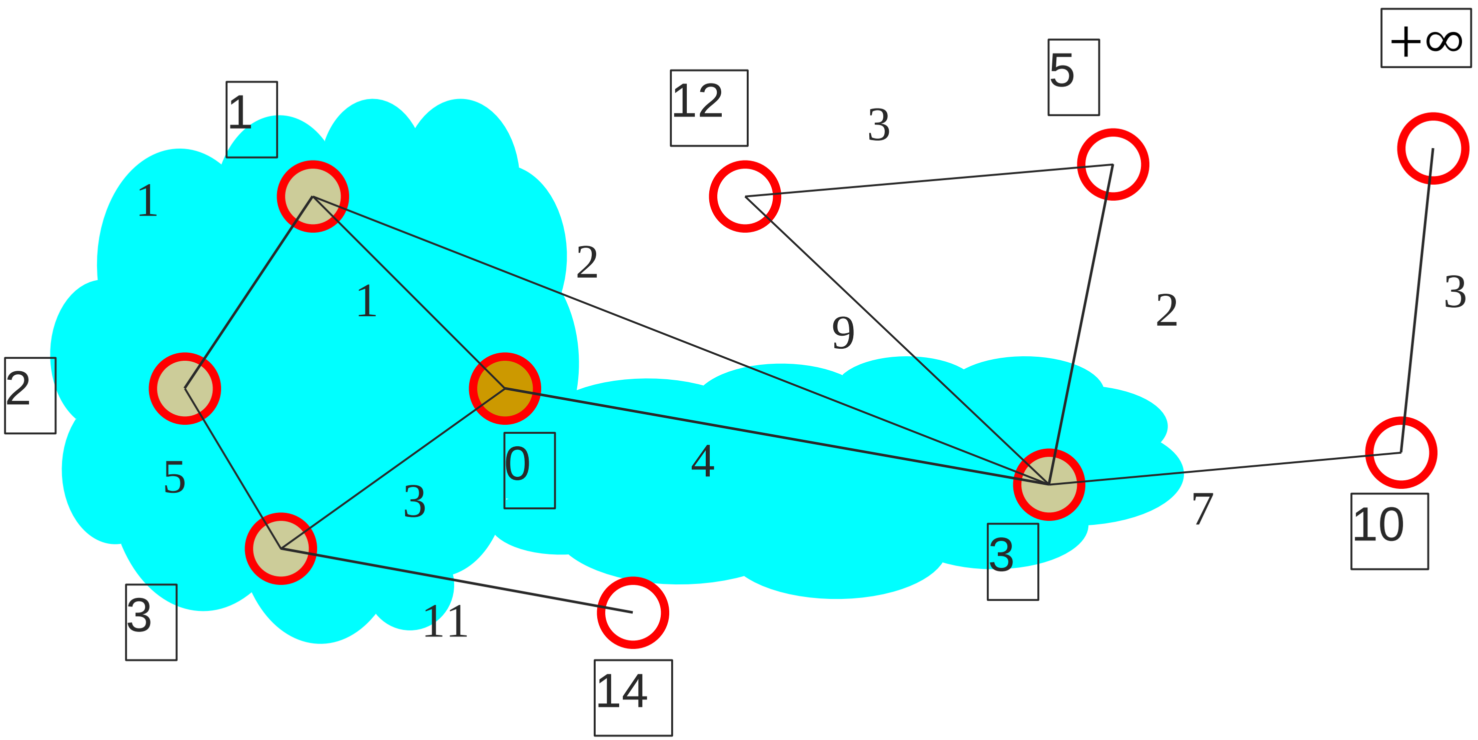
Dijkstra's algorithm: update *C's* neighborhood



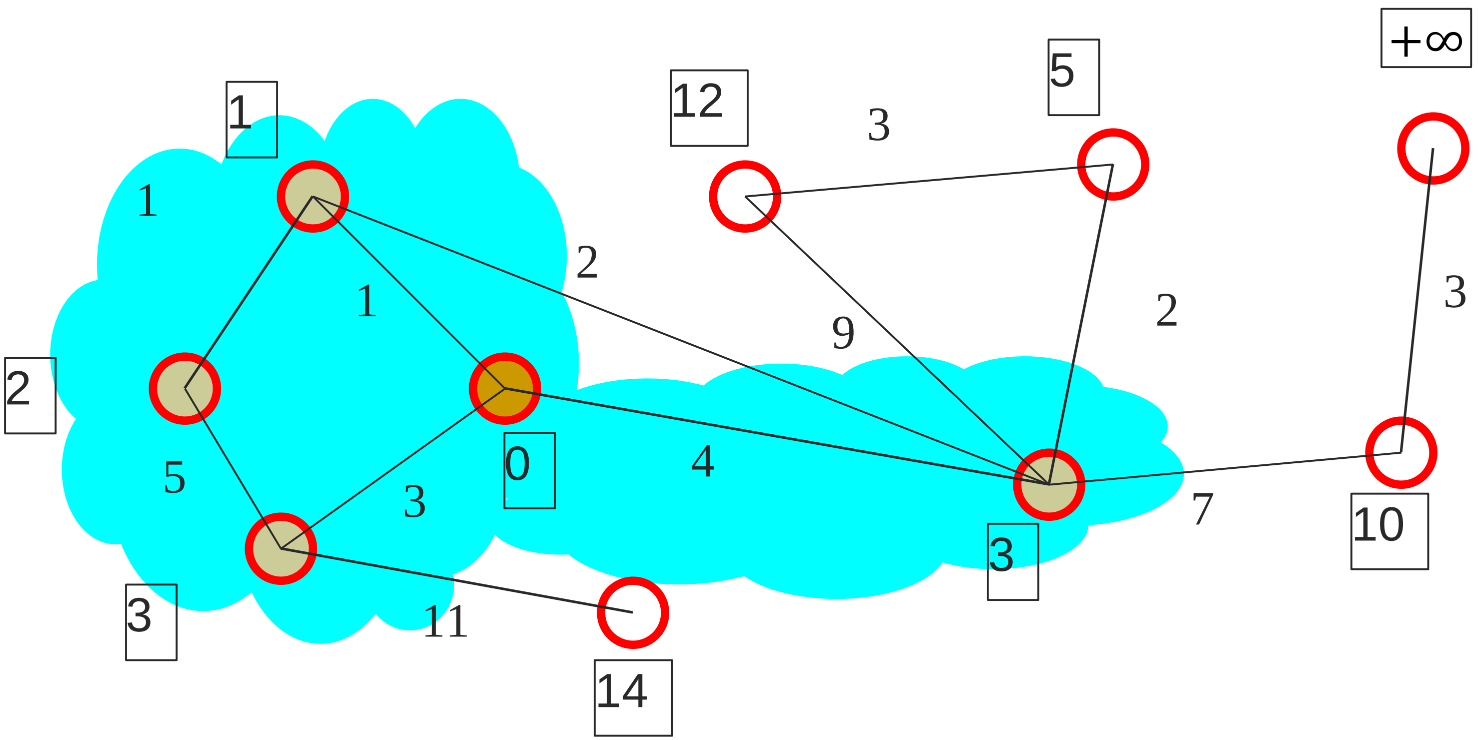
Dijkstra's algorithm: pick closest vertex u outside C



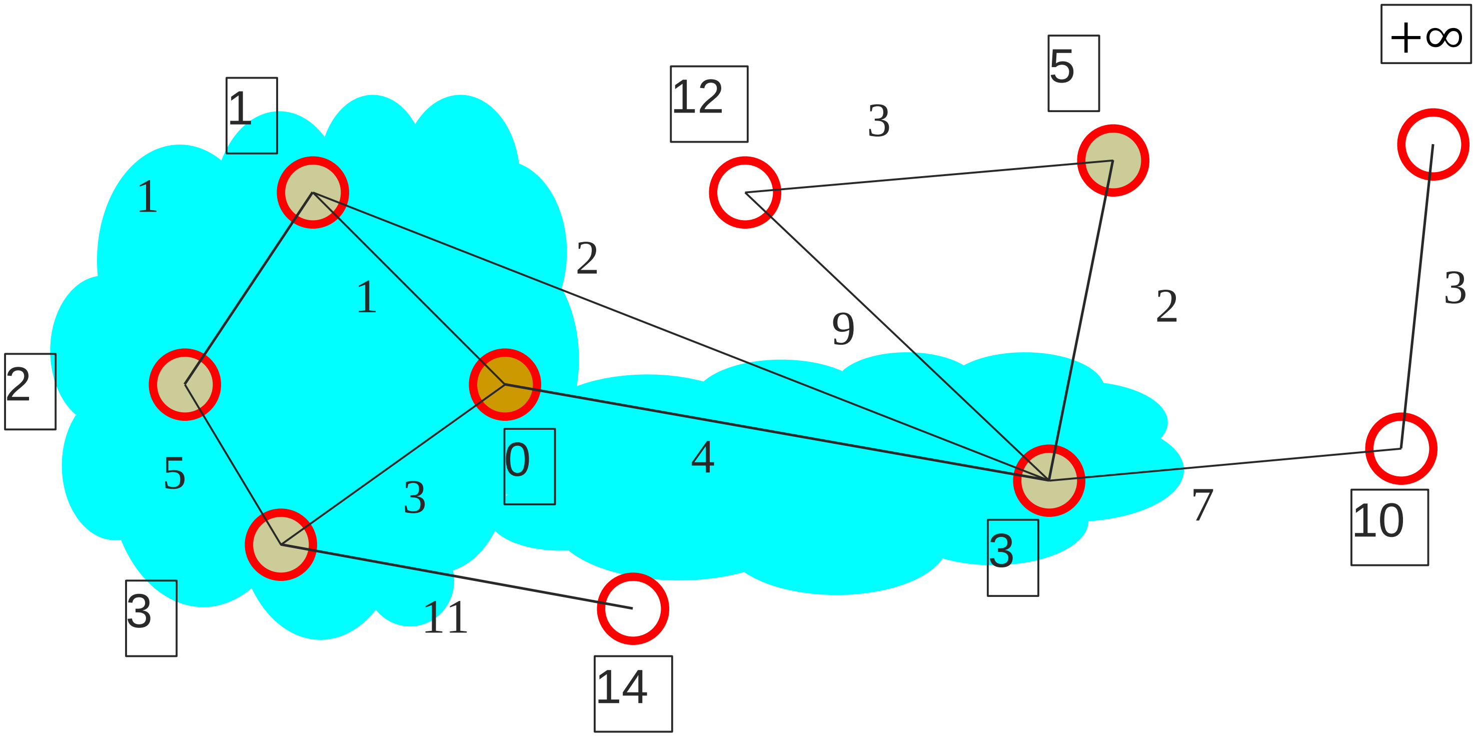
Dijkstra's algorithm: pull u into C



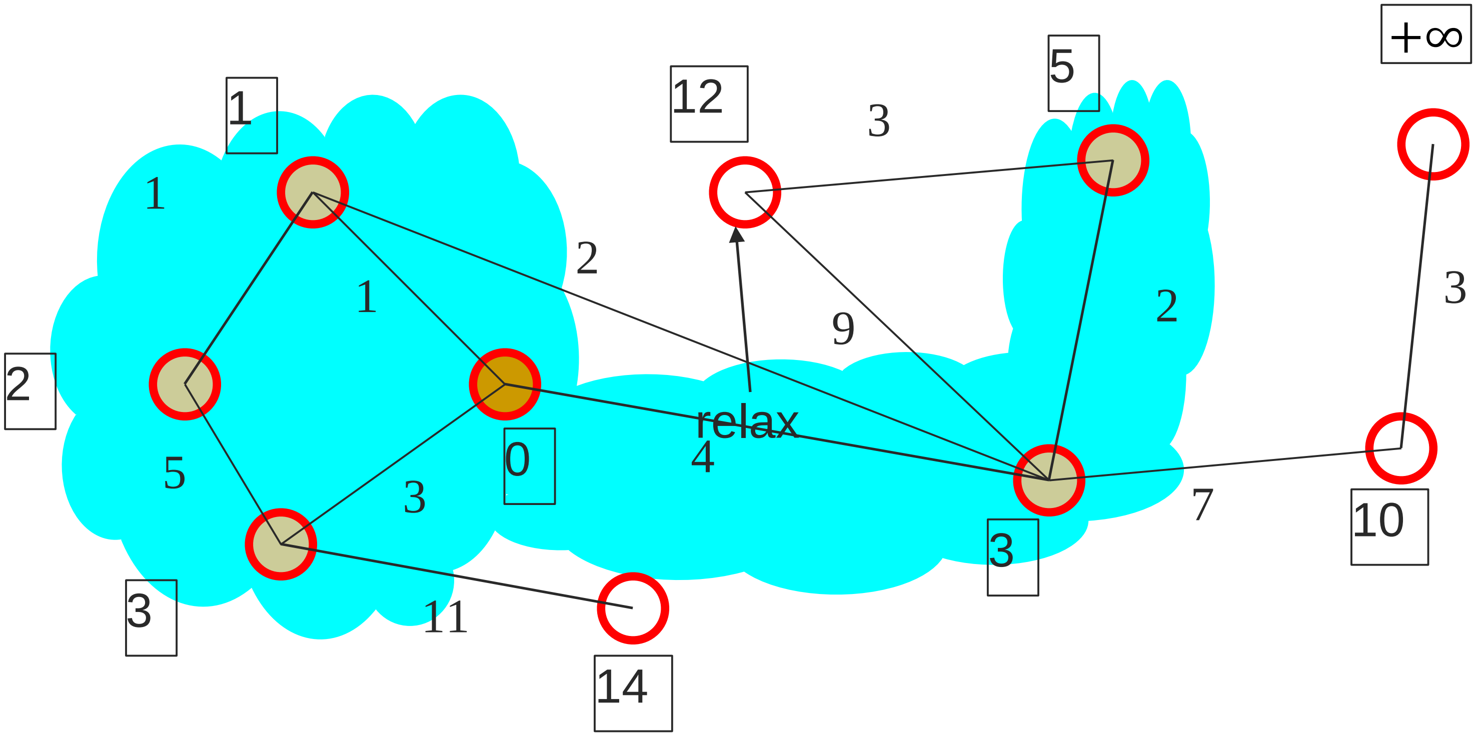
Dijkstra's algorithm: update *C's* neighborhood



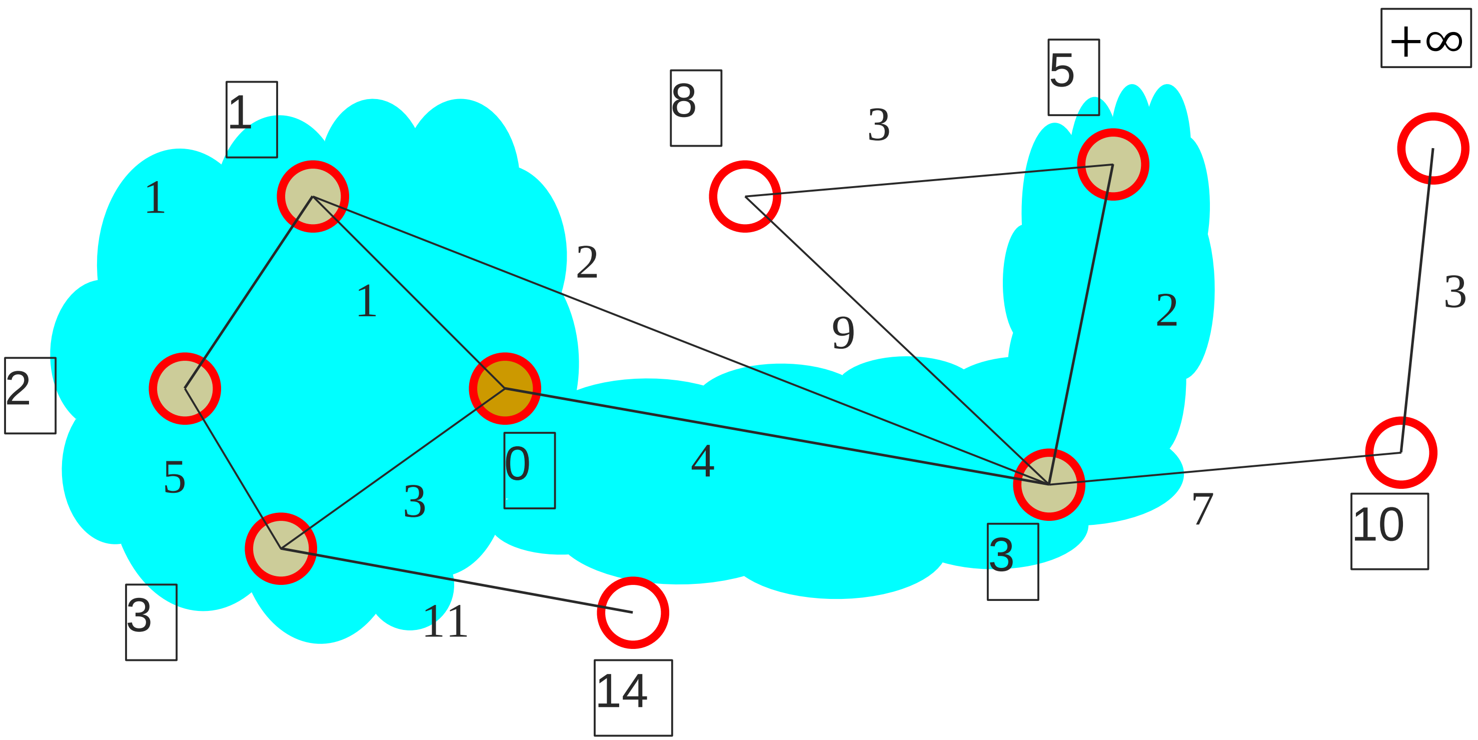
Dijkstra's algorithm: pick closest vertex u outside C



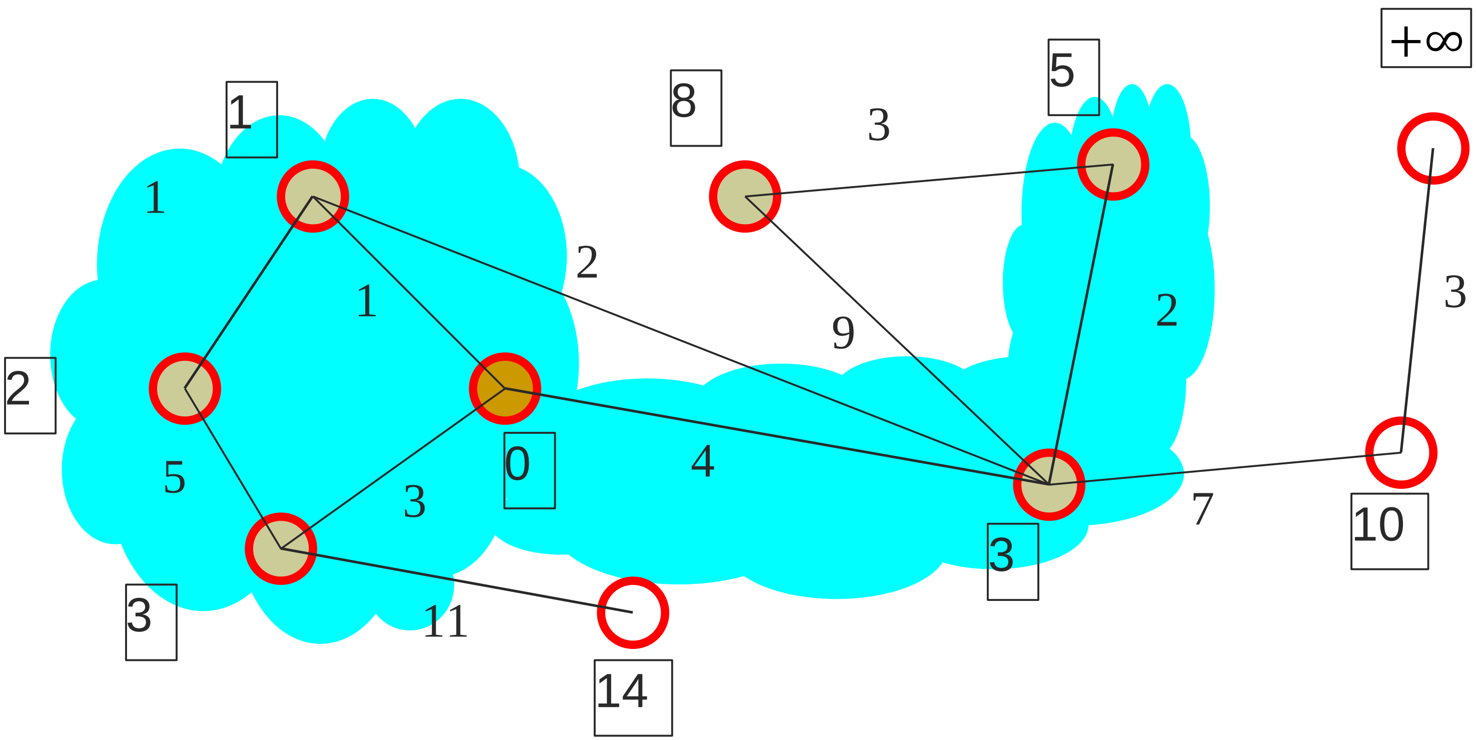
Dijkstra's algorithm: pull u into C



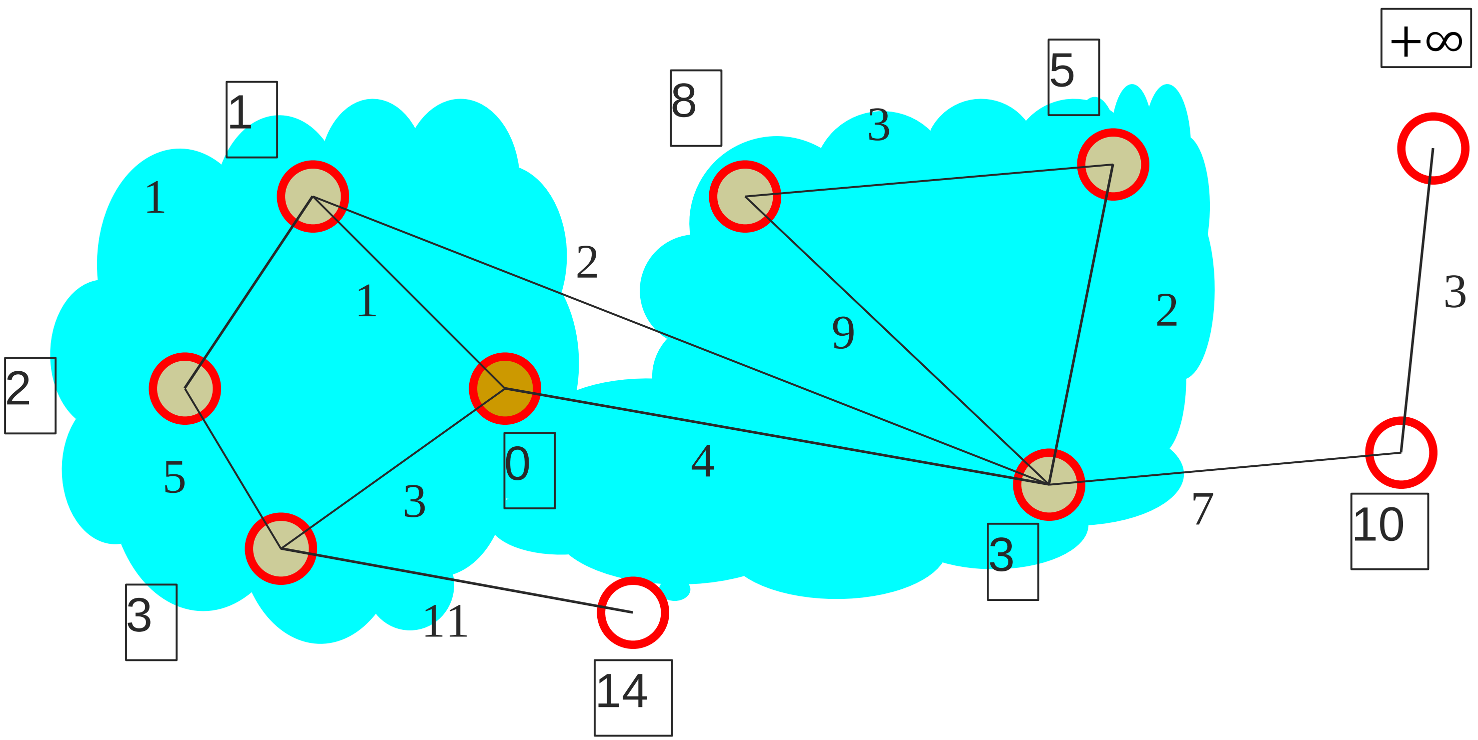
Dijkstra's algorithm: update *C*'s neighborhood



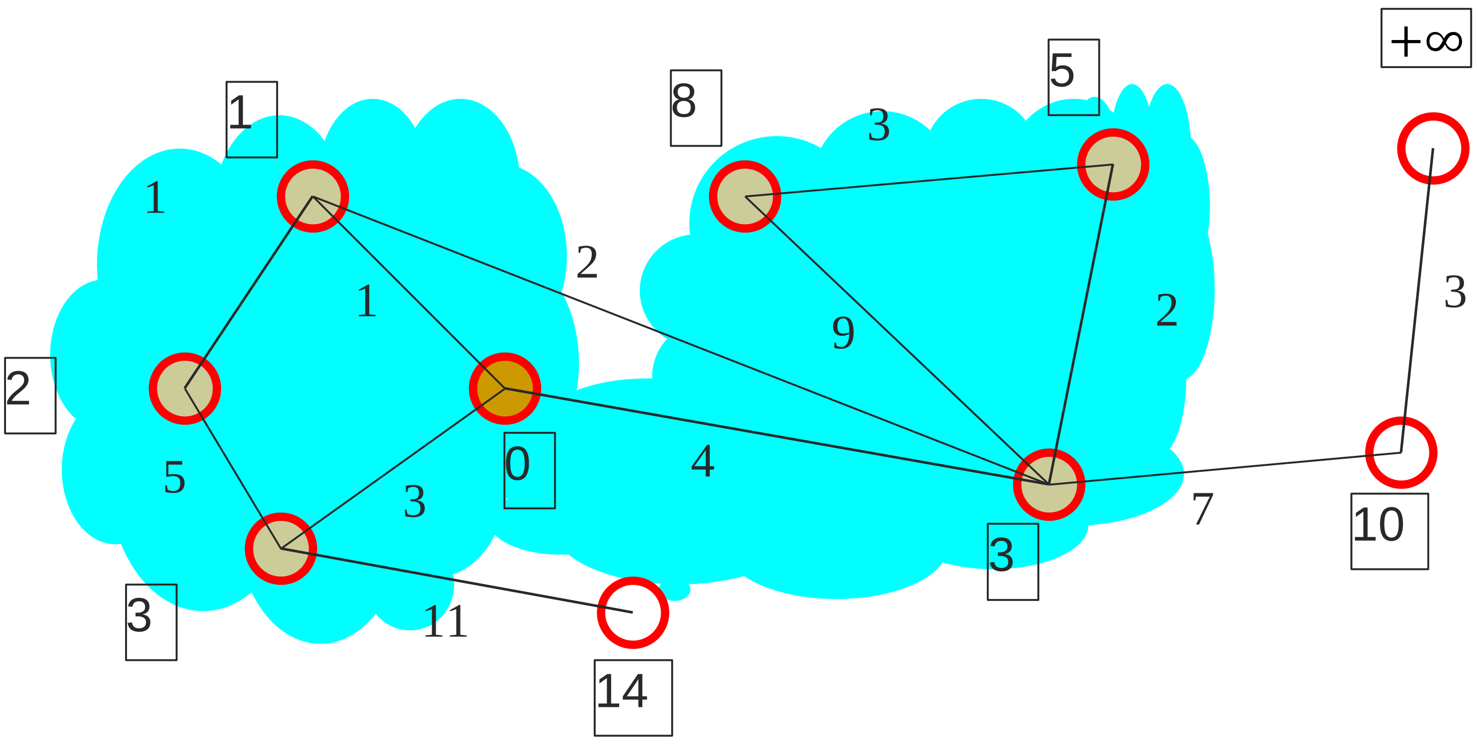
Dijkstra's algorithm: pick closest vertex u outside C



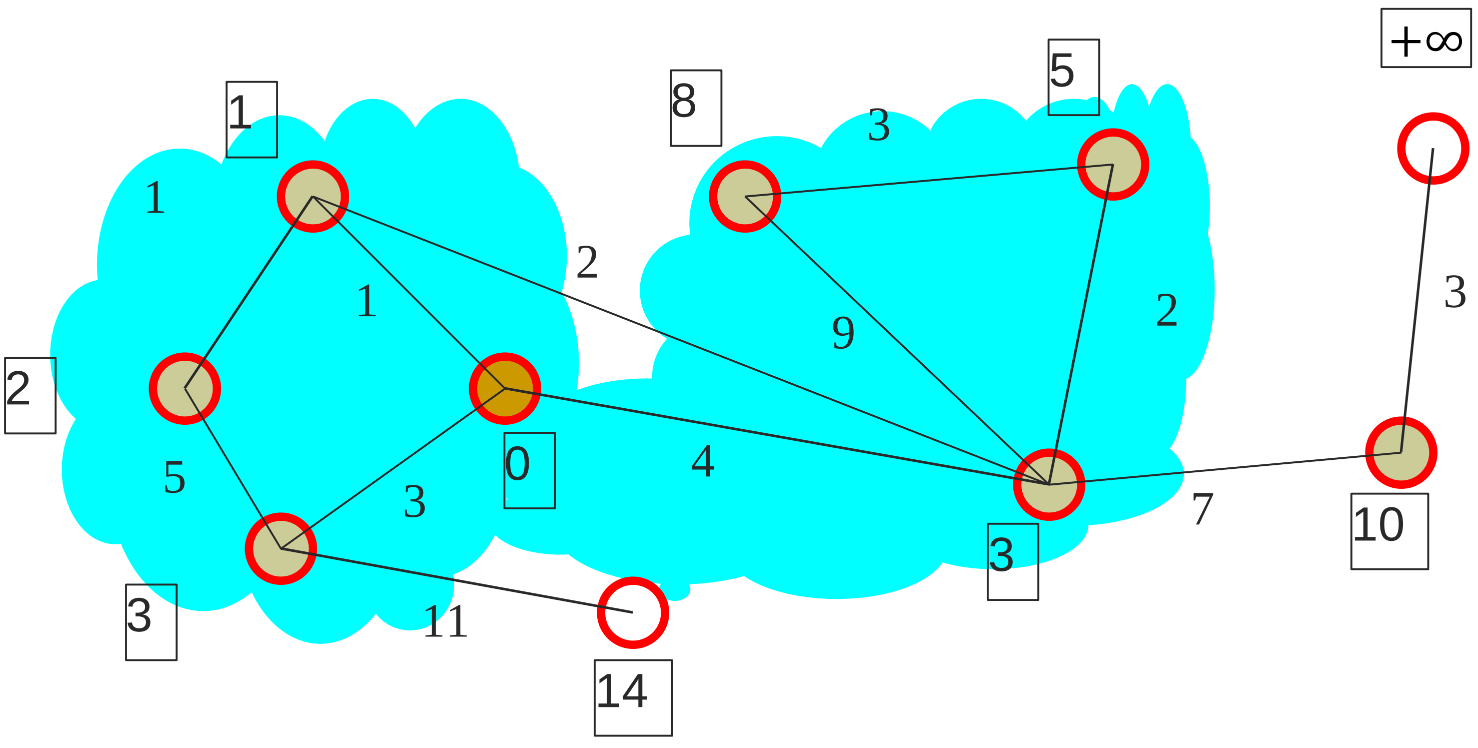
Dijkstra's algorithm: pull u into C



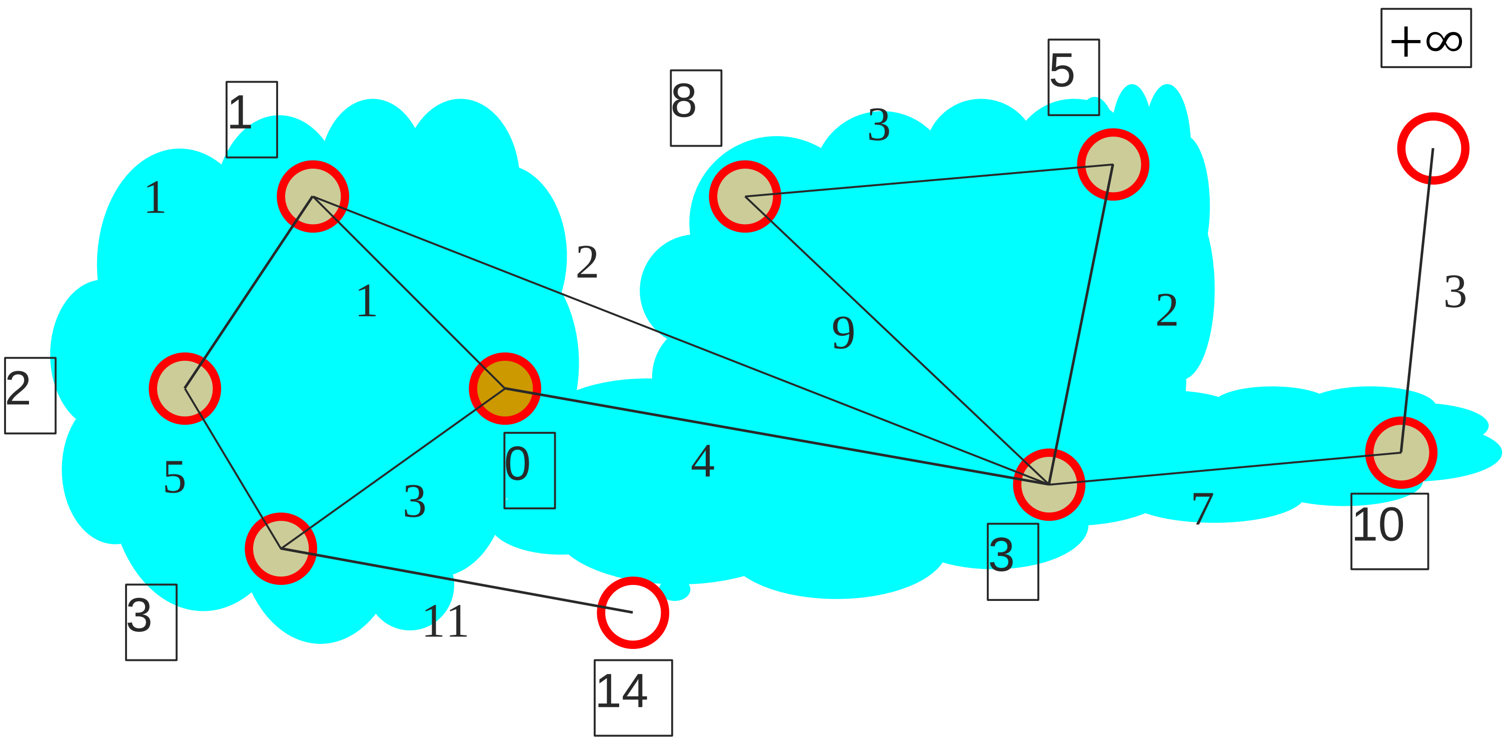
Dijkstra's algorithm: update *C's* neighborhood



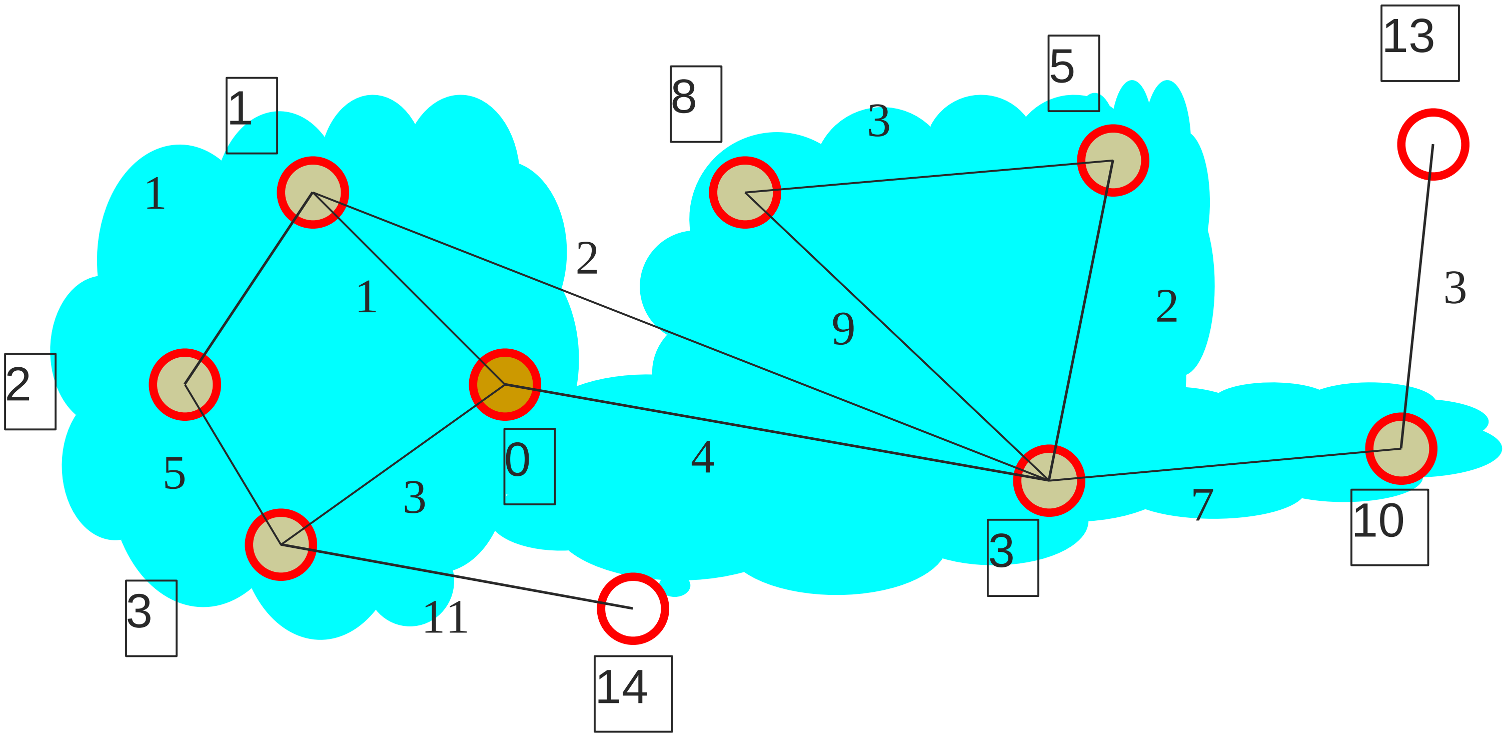
Dijkstra’s algorithm: pick closest vertex u outside C



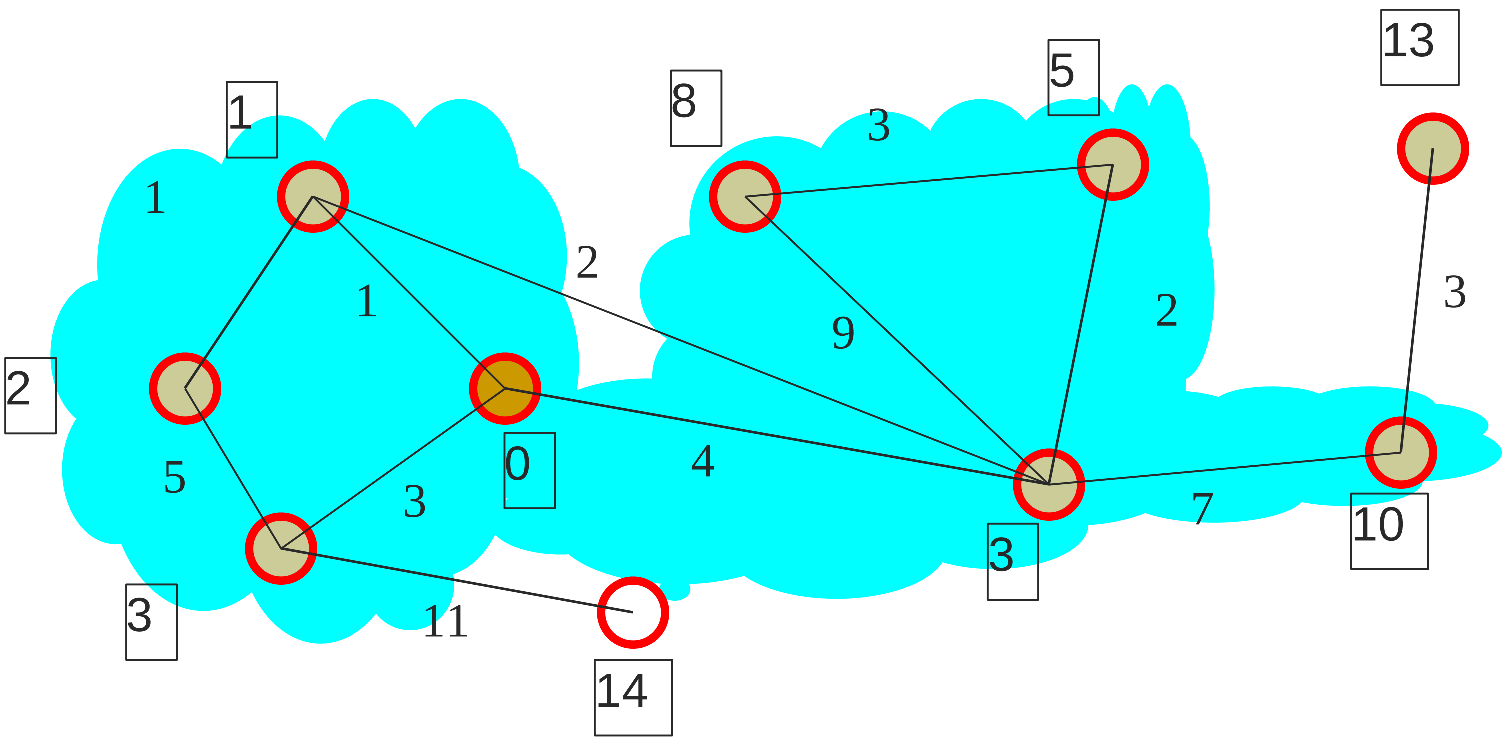
Dijkstra’s algorithm: pull u into C



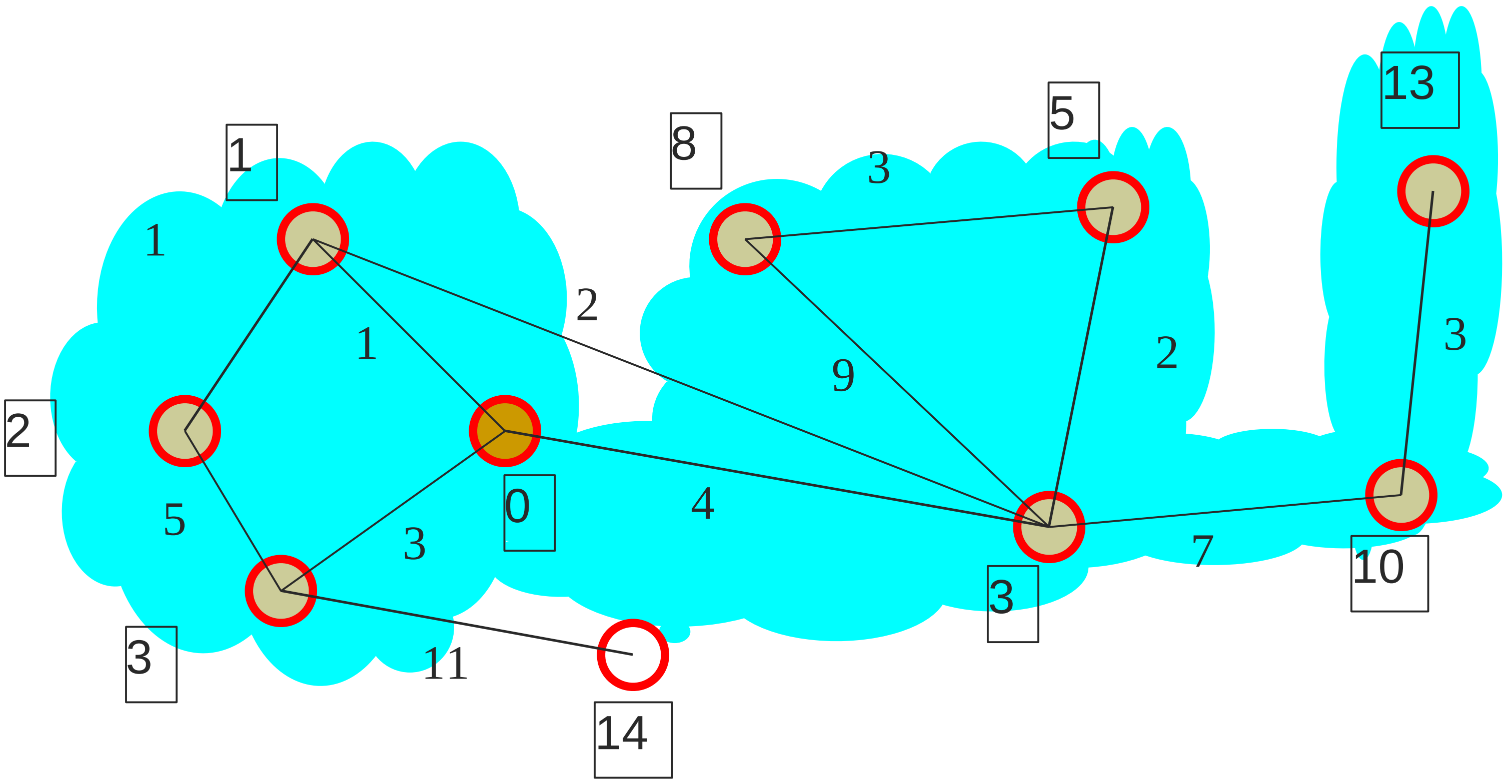
Dijkstra’s algorithm: update *C’s* neighborhood



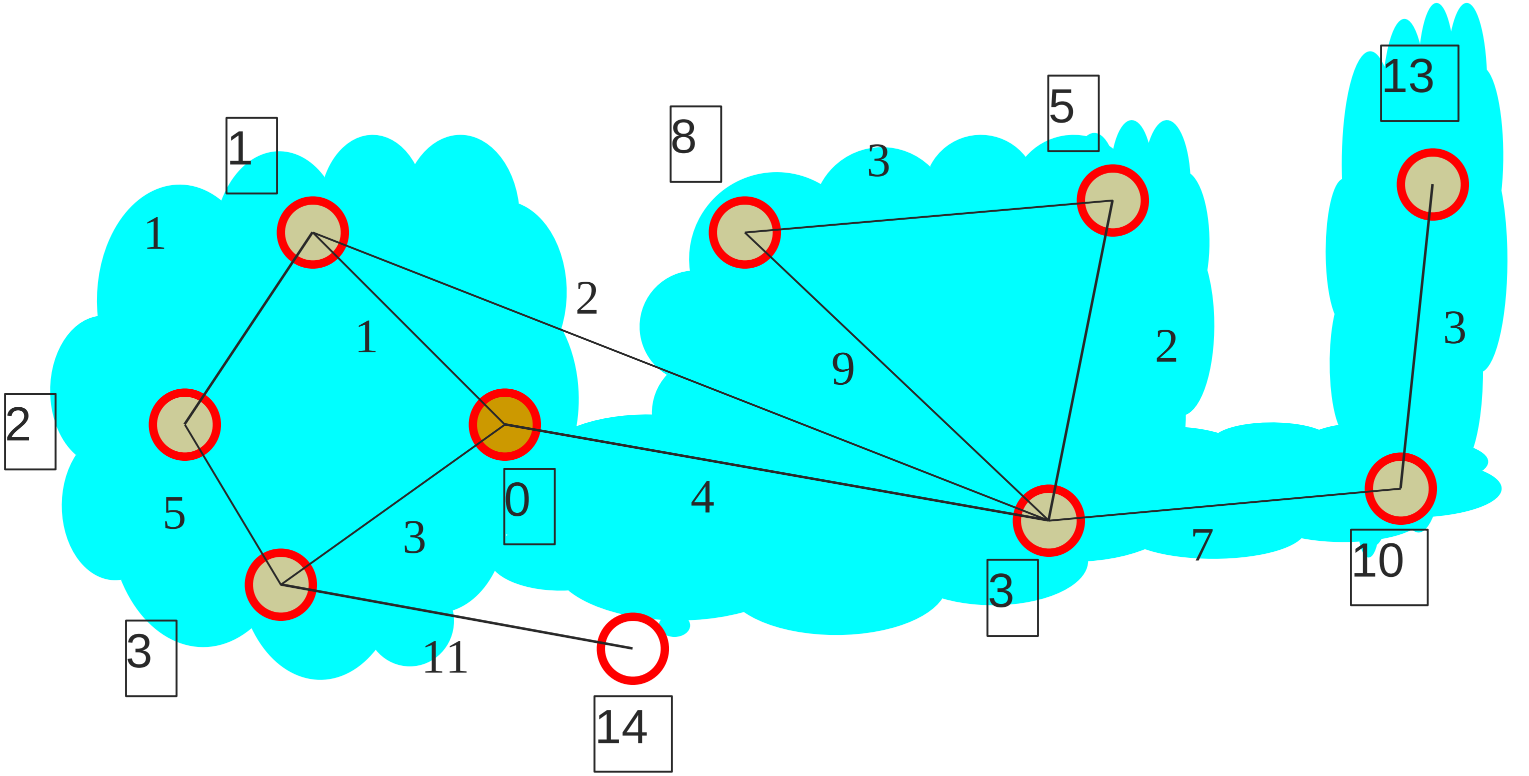
Dijkstra's algorithm: pick closest vertex u outside C



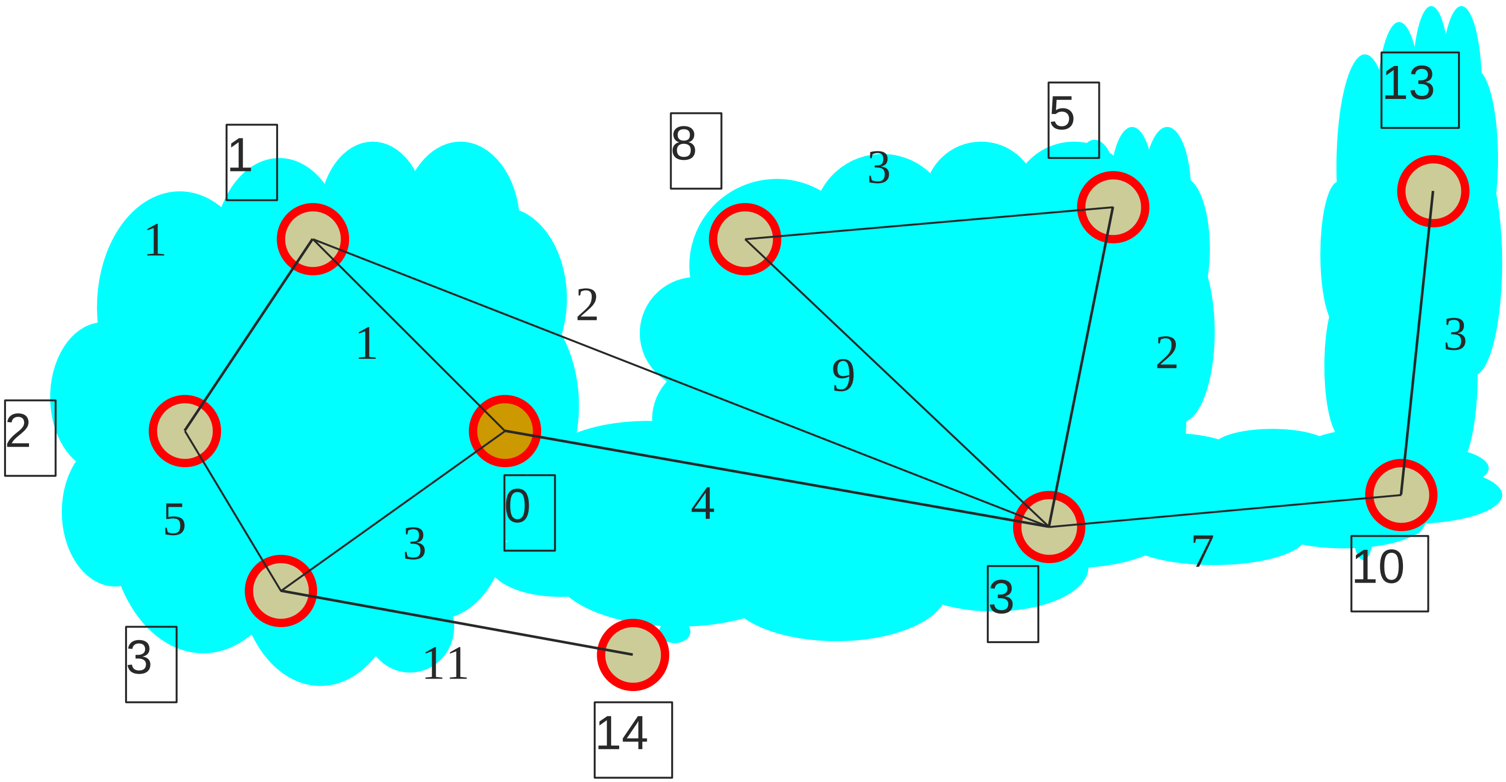
Dijkstra's algorithm: pull u into C



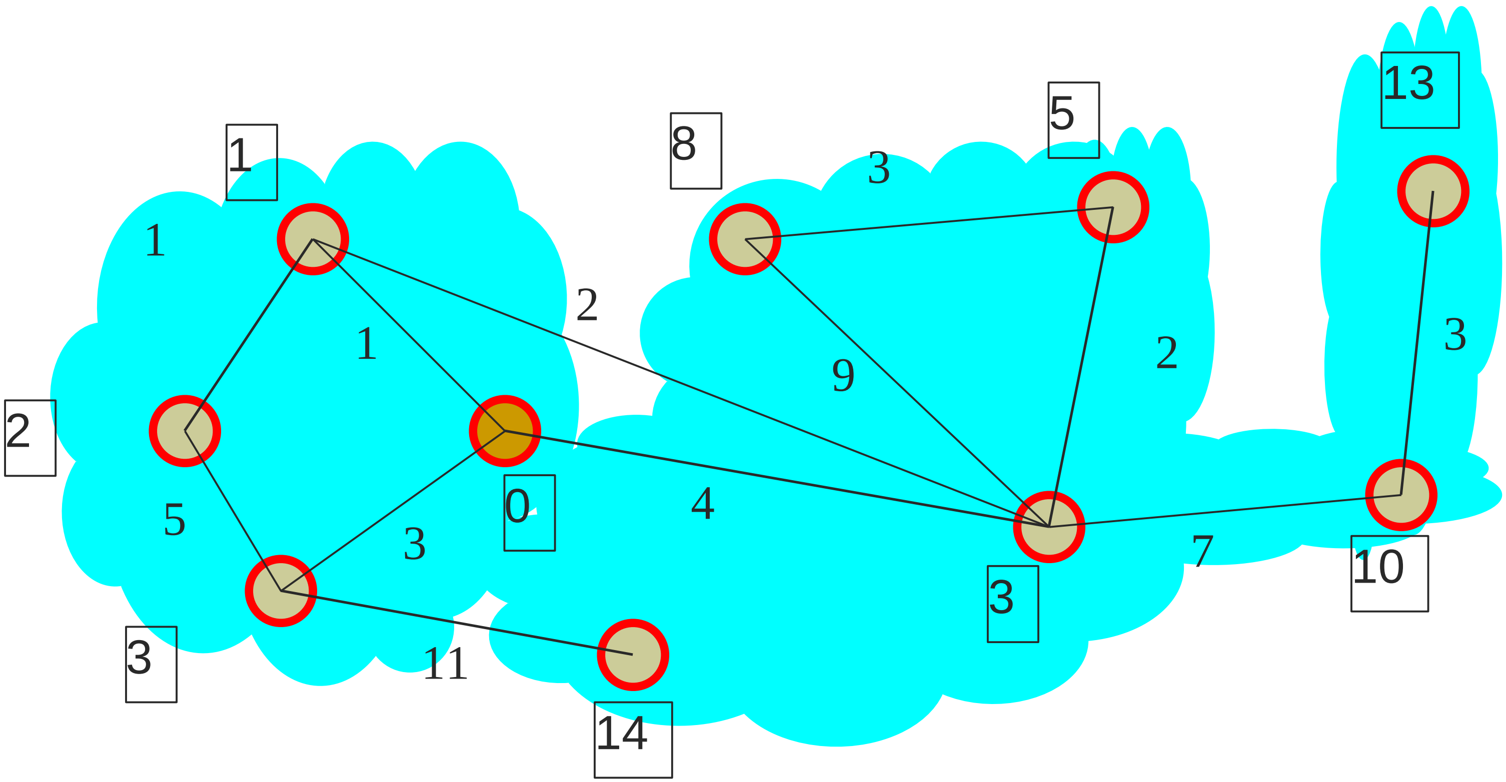
Dijkstra's algorithm: update *C's* neighborhood



Dijkstra's algorithm: pick closest vertex u outside C



Dijkstra's algorithm: pull u into C



When pulling a neighbour u of C into C

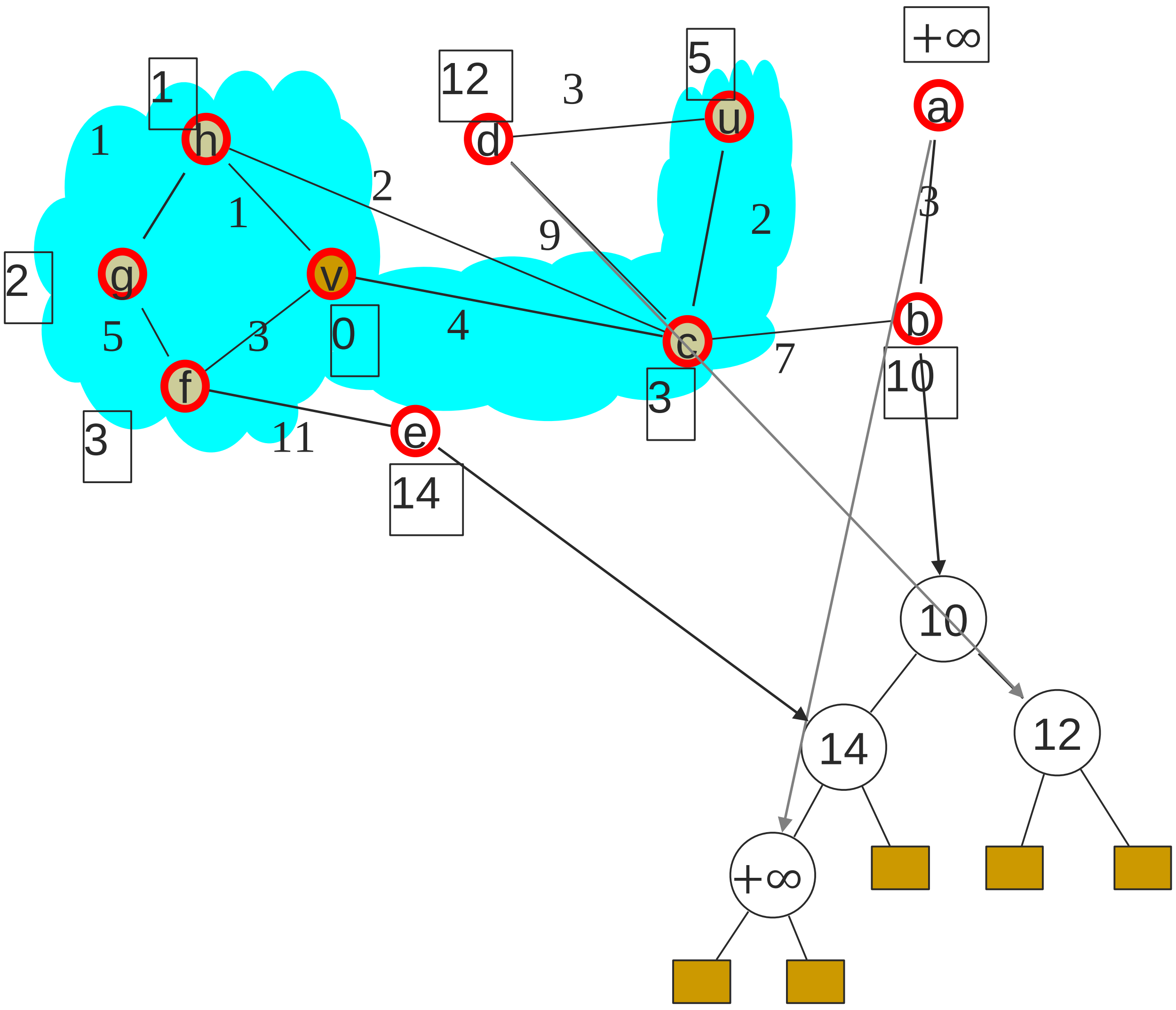
- The value associated with u denotes the length of a shortest path from v to u
- For any vertex x not in the cloud
  - the value associated with x denotes a shortest path from v to x without the use of other vertices outside of the cloud
  - $+\infty$ denotes that the vertex cannot be reached yet from v via cloud vertices only

Algorithm DijkstraShortestPaths(G,v)

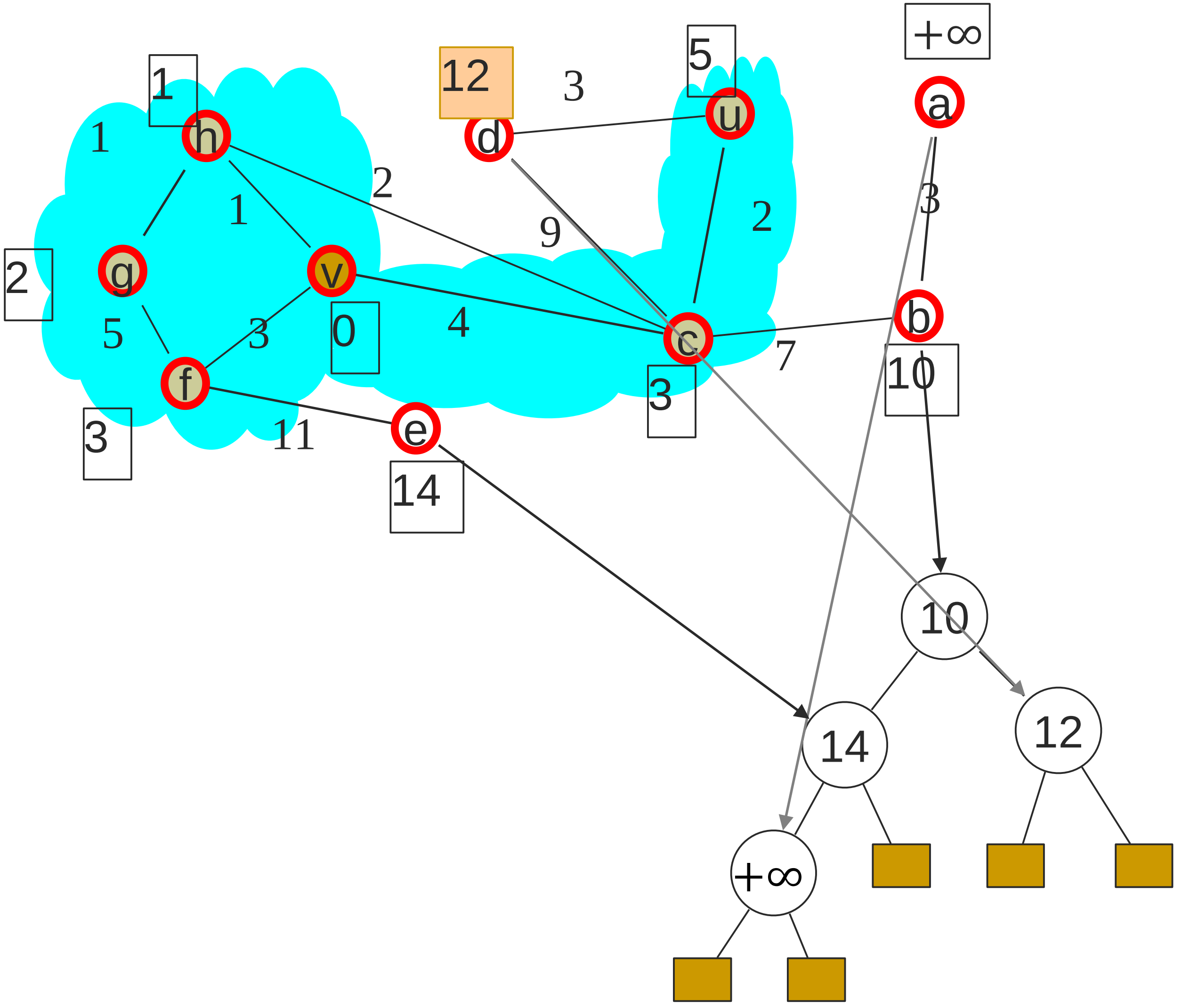
```
D[v]←0
for each vertex u≠v of G do
   D[u]← +∞
Let Q be a priority queue containing all
   vertices of G using D[.] as keys
while Q is not empty do
   u←Q.removeMin() //u is added to cloud
   for each vertex z∈N(u) with z∈Q do
     if D[u]+w((u,z)) < D[z] then
        D[z]←D[u]+w((u,z))
        update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

Relaxation

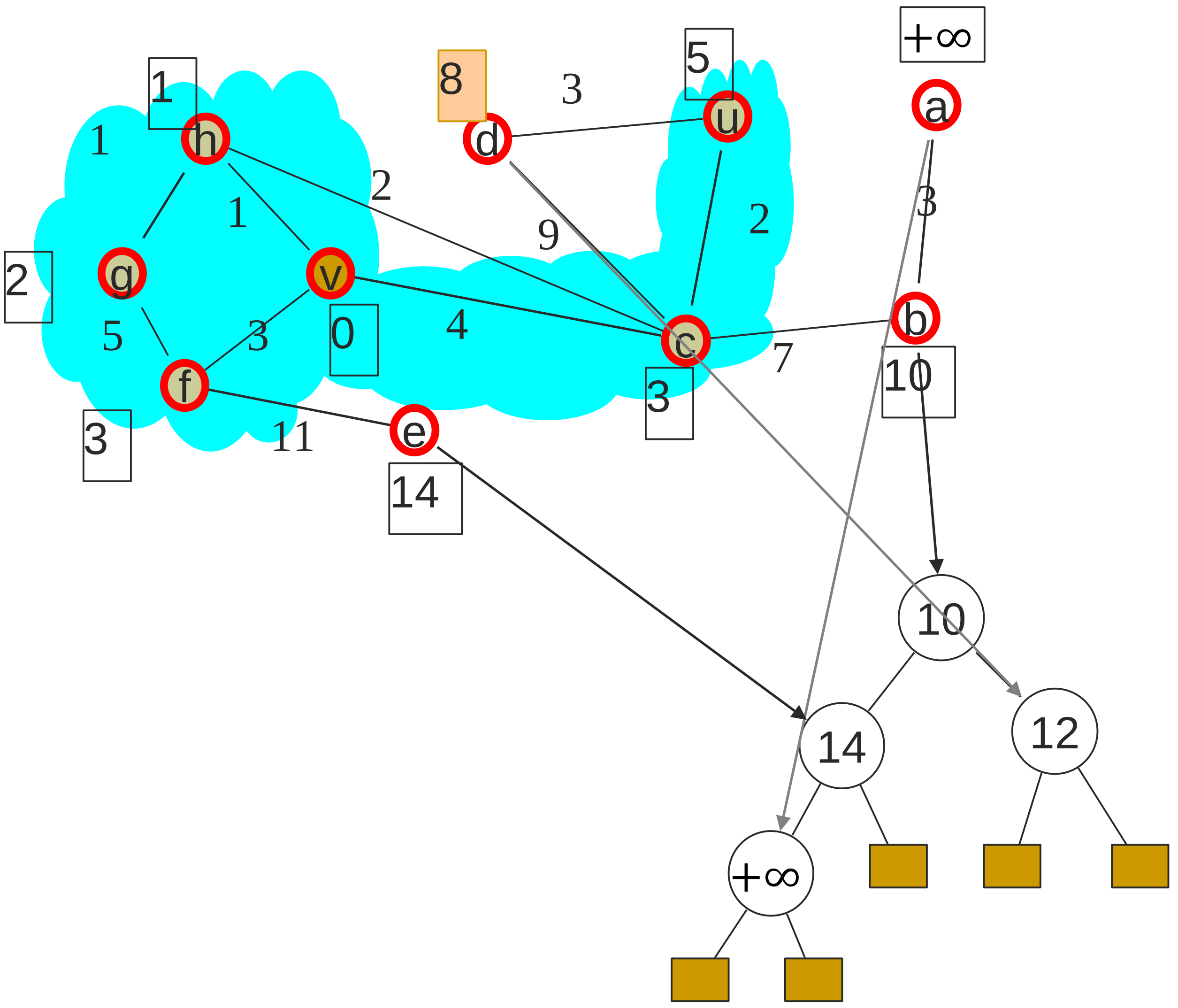
Dijkstra’s algorithm using heaps



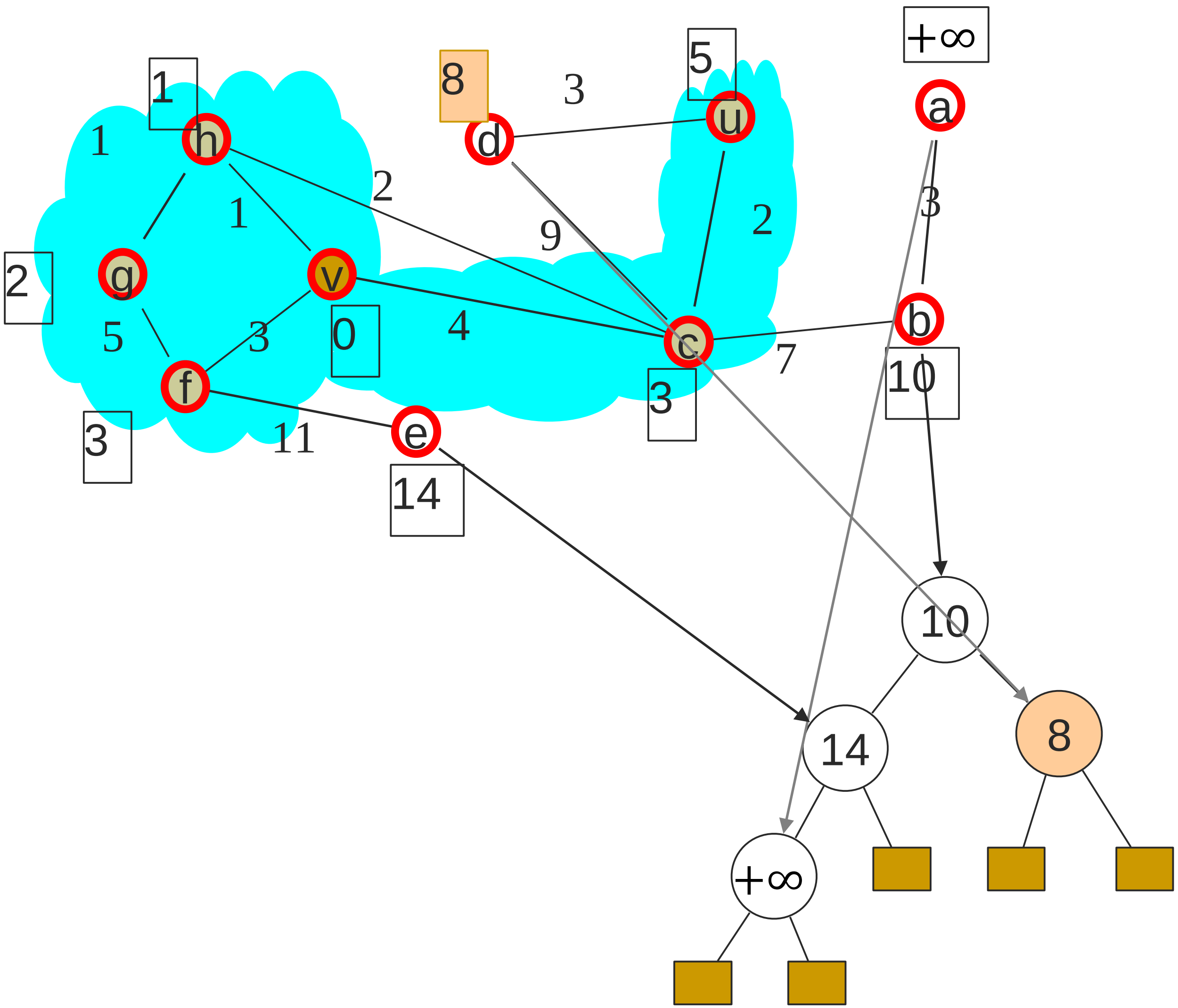
Dijkstra's algorithm using heaps



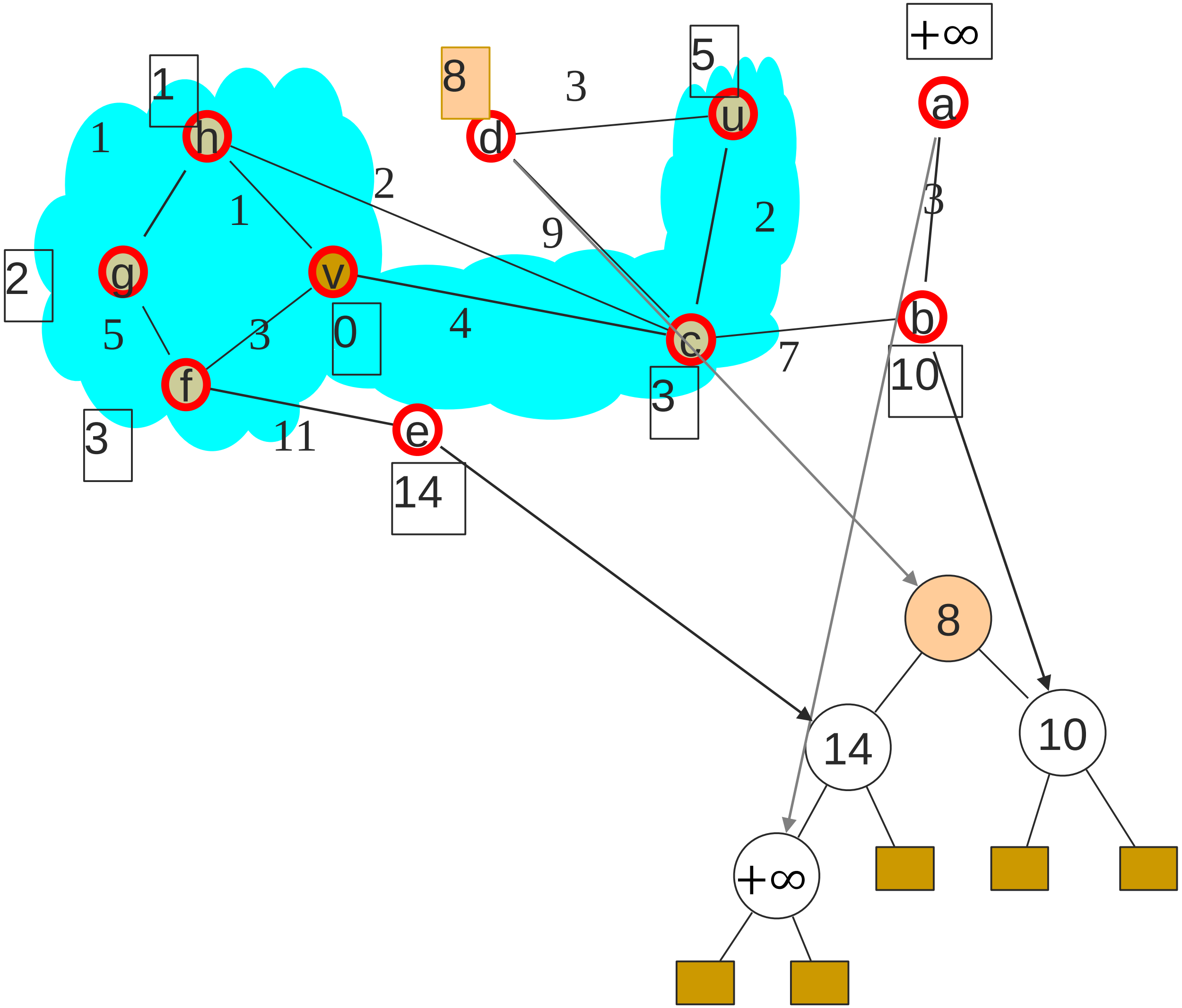
Dijkstra’s algorithm using heaps



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 $\quad O(\log n)$
    - Relaxation $\quad O(\deg(u)\log(n))$
  - $\sum_{u \in G}\big(1 + \deg(u)\big)\log n$ is $O((n+m)\log n)$

- Overall running time: $O(m \log n)$

In real life applications

- Often the graphs are sparse
- 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 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

Correctness of Dijkstra's algorithm

- **To show:** whenever u is pulled into cloud C, D[u] stores the length of a shortest path from v (the starting vertex) to u

- **Definition:** For vertices u and v in G, we denote with $d(v,u)$ the length of a shortest path from v to u.

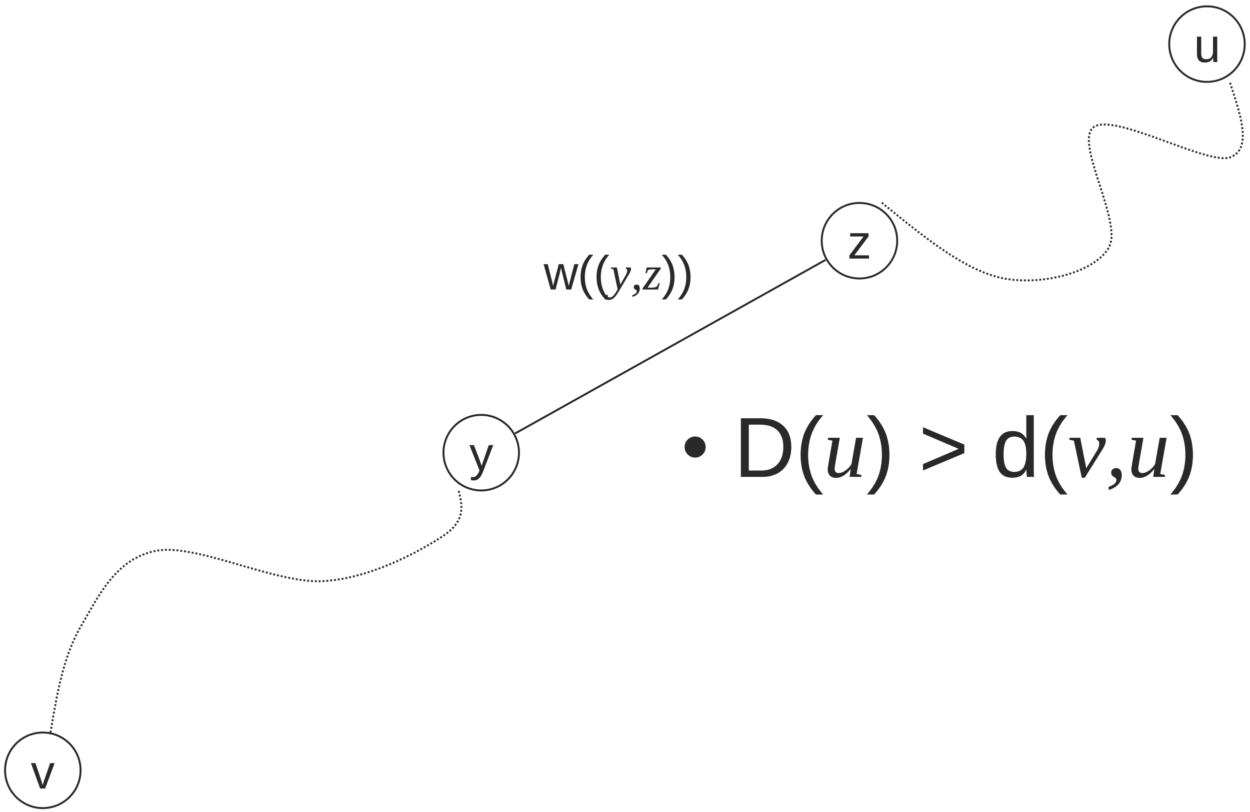
Whenever u is pulled into cloud C, D[u] stores the length of a shortest path from v to u

Proof. Assume: claim is wrong. Then:

there exists a vertex t that is pulled into cloud C and D[t] > d(v,t)

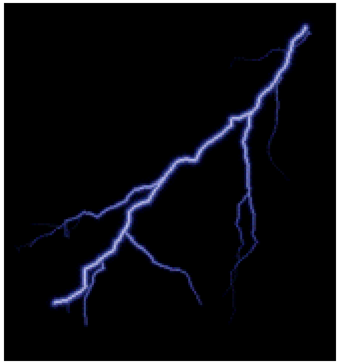
We define:

- u the first such vertex (currently) pulled into C
- P a shortest path in G from source v to vertex u
- y the last vertex that lies on P and is pulled “correctly” into C
- z the vertex closest to y that lies on P and is not in C



- $D(u) > d(v,u)$

- $D(u) > d(v,u)$
- $y \in C$, $D[y] = d(v,y)$
- $D(u) \le D(z)$
- $D(z) = d(v,z)$



- $D(u) \le D(z) = d(v,z) \le d(v,z) + d(z,u)$
  $= d(v,u)$

Prim's algorithm: eager implementation

```
public class PrimMST {
  private Edge[] edgeTo;                  // shortest edge from tree to vertex
  private double[] distTo;                // distTo[w] = edgeTo[w].weight()
  private boolean[] marked;               // true if v in mst
  private IndexMinPQ<Double> pq;          // eligible crossing edges

   public PrimMST(WeightedGraph G) {
     edgeTo = new Edge[G.V()];
     distTo = new double[G.V()];
     marked = new boolean[G.V()];
     for(int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
     pq = new IndexMinPQ<Double>(G.V());
     distTo[0] = 0.0;
     pq.insert(0, 0.0);
     while(!pq.isEmpty())
          visit(G, pq.delMin());
  }
```

assume G is connected

repeatedly delete the min weight edge e = v–w from PQ

Prim's algorithm: eager implementation

```
private void visit(WeightedGraph G, int v) {
   marked[v] = true;
   for (Edge e : G.adj(v)) {
      int w = e.other(v);
      if (marked[w]) continue;
      if (e.weight() < distTo[w]) {
           edgeTo[w] = e;
           distTo[w] = e.weight();
           if (pq.contains(w)) pq.changeKey(w, distTo[w]);
           else pq.insert(w, distTo[w]);
       }
   }
}
public Iterable<Edge> edges(){
   Queue<Edge> mst = new Queue<Edge>();
   for (int v = 0; v < edgeTo.length; v++)
      Edge e = edgeTo[v];
      if (e != null) {
         mst.enqueue(e);
      }
   }
   return mst; }
```

add v to T

for each edge e = v–w, add to PQ if w not already in T

add edge e to tree

Update distance to w or Insert distance to w

Create the mst

Djikstra's algorithm: eager implementation

```
public class DijkstraUndirectedSP {
  private Edge[] edgeTo;                  // last edge from on path to v
  private double[] distTo;                // distance to v from s
  private IndexMinPQ<Double> pq;     // eligible crossing edges

   public DijkstraUndirectedSP(WeightedGraph G, int s) {
      edgeTo = new Edge[G.V()];
      distTo = new double[G.V()];

      for(int v = 0; v < G.V(); v++)
           distTo[v] = Double.POSITIVE_INFINITY;
      pq = new IndexMinPQ<Double>(G.V());
      distTo[s] = 0.0;
      pq.insert(s, distTo[s]);
      while(!pq.isEmpty())
             relax(G, pq.delMin());
  }
```

assume G is connected

repeatedly delete the edge e = v–w from PQ that is closest to s.

Dijkstra's algorithm: eager implementation

```
private void relax(WeightedGraph G, int v) {

  for (Edge e : G.adj(v)) {
    int w = e.other(v);

    if (distTo[v] + e.weight() < distTo[w]) {
        edgeTo[w] = e;
        distTo[w] = distTo[v] + e.weight();
        if (pq.contains(w)) pq.changeKey(w, distTo[w]);
        else pq.insert(w, distTo[w]);
     }
  }
}
public Iterable<Edge> edges(){
  Queue<Edge> spt = new Queue<Edge>();
  for (int v = 0; v < edgeTo.length; v++)
    Edge e = edgeTo[v];
    if (e != null) {
      spt.enqueue(e);
    }
  }
  return spt; }
```

for each edge e = v–w, add to PQ if w not already in T

add edge e to tree

Update distance to w or Insert distance to w

Create the spt