Discrete mathematics

Petr Kovář & Tereza Kovářová
petr.kovar@vsb.cz

VŠB – Technical University of Ostrava

Winter Term 2022/2023
DiM 470-2301/02, 470-2301/04, 470-2301/06

EUROPEAN UNION
European Structural and Investment Funds
Operational Programme Research,
Development and Education

MINISTRY OF EDUCATION,
YOUTH AND SPORTS

The translation was co-financed by the European Union and the Ministry of Education, Youth and Sports from the Operational Programme Research, Development and Education, project "Technology for the Future 2.0", reg. no. CZ.02.2.69/0.0/0.0/18_058/0010212.

About this file

This file is meant to be a guideline for the lecturer. Many important pieces of information are not in this file, they are to be delivered in the lecture: said, shown or drawn on board. The file is made available with the hope students will easier catch up with lectures they missed.

For study the following resources are better suitable:

- Meyer: Lecture notes and readings for an http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-fall-2005/readings/" (weeks 1-5, 8-10, 12-13), MIT, 2005.
- Diestel: Graph theory http://diestel-graph-theory.com/ (chapters 1-6), Springer, 2010.

See also http://homel.vsb.cz/~kov16/predmety_dm.php

Lecture overview

Chapter Distance and measuring in graphs

- motivation
- distance in graphs
- measuring in graphs
- weighted distance
- shortest path algorithm

Motivation

In many real life applications of graphs we need to "measure" distances.

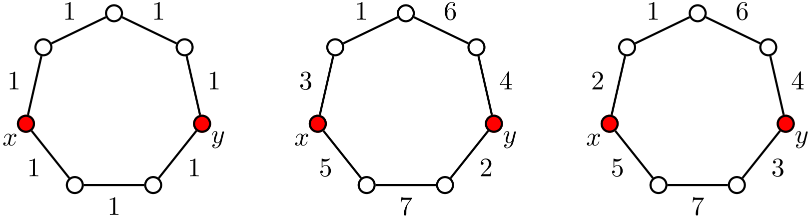
In a graph representing a road network it is natural to ask

> *"How far is it from vertex (place) u to vertex (place) v?"* or
>
> *"How long does it tak to travel from vertex u to vertex v?"*

The distance will not be just mere *number of edges* (number of roads traveled) but important will be their *length*. Notice that length have not been considered yet.

We will introduce the notion of labeling edges. The meaning of the labels may vary: length, width, capacity, color, . . .

Usually, for labels one can use natural numbers only (well chosen scale).



Different distances between vertices u and v in graph C_7.

In the graph on the left
the distance between vertices x and y is $3 =$ number of edges of the shorter path (walk).

In the graph in the middle
the distance between vertices x and y is $14 = 3 + 1 + 6 + 4 = 5 + 7 + 2$.

In the graph on the right
the distance between vertices x and y is $13 = 2 + 1 + 6 + 4$.

Distance in graphs

First for unlabeled graphs, i.e. each edge has length 1.

Length of a walk is the number of edges in the sequence of vertices and edges in a a walk

$$v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n,$$

where each edge e_i has end-vertices v_{i-1} and v_i.

Definition

Distance $\mathrm{dist}_G(u, v)$ between vertices u and v in a graph G is given by the length of the shortest walk between u and v in G. If no walk between u and v exists, we define the length to be $\mathrm{dist}_G(u, v) = \infty$.

Notice, that

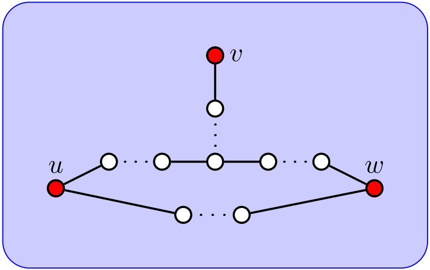
- the shortest walk (with the fewest edges) is always a path
- in unoriented graphs is $\mathrm{dist}_G(u, v) = \mathrm{dist}_G(v, u)$
- $\mathrm{dist}_G(u, u) = 0$
- if $\mathrm{dist}_G(u, v) = 1$, then edge $uv \in E(G)$

Lemma

Distance in a graph G satisfies the *triangle inequality*:

$$\forall u, v, w \in V(G)\colon\ \operatorname{dist}_G(u, w) \leq \operatorname{dist}_G(u, v) + \operatorname{dist}_G(v, w).$$

Proof The inequality follows from the observation, that the walk of length $\operatorname{dist}_G(u, v)$ between u, v joined with the walk of length $\operatorname{dist}_G(v, w)$ between v, w gives a walk of length $\operatorname{dist}_G(u, v) + \operatorname{dist}_G(v, w)$ between u, w. Never $\operatorname{dist}_G(u, w) > \operatorname{dist}_G(u, v) + \operatorname{dist}_G(v, w)$. Yet, a shorter walk from u to v can exist $\operatorname{dist}_G(u, w) \leq \operatorname{dist}_G(u, v) + \operatorname{dist}_G(v, w)$. □



Two walks u, v and v, w; a shorter walk between u, w.

Measuring in graphs (graph metrics)

When measuring distances one cannot simply choose among all possible paths.

Example

What is the number of all paths between u, v in a complete graph K_n.

1. If $u = v$, then there exists only one (trivial) path from u to v.
2. Of $u \neq v$ there exist $V(n-2,k) = \frac{(n-2)!}{(n-2-k)!}$ different paths from u to v with k internal vertices, $0 \leq k \leq n-2$.

The total number of different uv paths is $\displaystyle\sum_{k=0}^{n-2} \frac{(n-2)!}{(n-2-k)!}$.

There are $O((n-2)!)$ different paths ... too many possibilities.

For $n = 10$ there are 109 601 different paths in K_{10}.
For $n = 15$ there are already 16 926 797 486 different paths in K_{15}.
And for $n = 20$ there are already $1.74 \cdot 10^{16}$ different paths in K_{20}.
There are more than 670 tram-/bus-stops in Ostrava...

There is a simple modification of the breadth-first search algorithm (depository implemented as a queue Q).
We determine lengths of the shortest paths form a given vertex to every other vertex.

Each newly found vertex w will be assigned the distance by one greater than the processed vertex v.
Distances are stored an a one-dimensional array `dist[]`.

Algorithm: Distances from a given vertex

```
// on the input is the graph G
input < graph G;
status(all vertices of G) = initial;
queue Q = a given vertex u of G;
status(u) = found;
dist(u) = 0;                          // distance of u
```

Algorithm: Distances from a given vertex (continued)

```
// processing a selected component of G
while (Q is not empty) {
   pick a vertex v from the queue Q; Q = Q - v;
   for (edges e incident with v)        // for all edges
      w = other end-vertex of e = vw;  // known neighbor?
      if (status(w) == initial) {
         status(w) = found;
         add vertex w to queue: Q = Q + w;
         dist[w] = dist[v]+1;           // distance of w
      }
   }
   status(v) = processed;
}
// vertices in additional components are unreachable!
while (there are unprocessed vertices w in G) {
   dist[w] = MAX_INT;                 // infinity
   status(w) = processed;
}
```

Notice:

- The number of steps depends on the number of vertices and edges of the given graph.
  Complexity is $O(n+m)$, where n is the number of vertices and m is the number of edges.
  After the line `dist[w] = dist[v]+1;`
  add the line `pre[w] = v;`
- If we store for every vertex its *predecessor* on the shortest path, we can reconstruct the path:
  - the last vertex is w,
  - the next-to-the-last vertex is $pre[w]$,
  - the the next-to-the-next-to-the-last vertex is $pre[pre[w]]$,
  - ...
  - first (i.e. starting) vertex is $pre[\ldots pre[pre[w]]] = u$.

We assumed that vertices closer to u are processed before more distant vertices.
This can be proven and used to prove the validity of the algorithm.

Lemma

Let u, v, w be vertices of a connected graph G such, that $\mathrm{dist}_G(u, v) < \mathrm{dist}_G(u, w)$. In a *breadth-first* search in G starting at the vertex u the vertex v will always be found before the vertex w.

Proof By induction on $\mathrm{dist}_G(u, v)$.
Basis step: For $\mathrm{dist}_G(u, v) = 0$, i.e. $u = v$ the claim is obvious – the vertex u is found first.
Inductive step: Now for some $\mathrm{dist}_G(u, v) = d > 0$ we denote by v' the neighbor of v on the shortest walk u, v to u, obviously $d_G(u, v') = d - 1$. Similarly, by w' we denote the neighbor of w on the walk u, w to u, thus $\mathrm{dist}_G(u, v') < \mathrm{dist}_G(u, w')$.
By the induction hypothesis the vertex v' will be in a breadth-first search found before w'. This implies also, that v' will come to the queue of the depository before w', and thus the neighbors of v' (v is among them) will be found before the neighbors of w'. $\square$

Corollary

The basic algorithm for breadth-first search can be used to count distances from the vertex u to all other vertices.

Proof is in the textbook.

Questions

Why the depth-first search cannot be used instead the breadth-first search? Which part of the algorithm would fail?

Evaluating the metrics

By a metrics we understand the distance between any pair of vertices in a given graph. We expect the metrics to satisfy "common properties".

Formally: the set of vertices along with the distance function for every pair of vertices forms a metric space.

Definition

Metrics ρ on a given set A is such a mapping $\rho : A \times A \to \mathbb{R}$, that $\forall x, y \in A$ the following holds

1. $\rho(x, y) \geq 0$ while $\rho(x, y) = 0$ only for $x = y$,
2. $\rho(x, y) = \rho(y, x)$,
3. $\rho(x, y) + \rho(y, z) \geq \rho(x, z)$.

Informally: The metrics in G is a matrix (two-dimensional field) `d[][]`, where `d[i][j]` gives the distance between vertices i and j (vertices are 0, 1, . . . , $|V(G)| - 1$).

To find the metrics we can use the algorithm for measuring distances from a given starting vertex (repeat it for every starting vertex u).
There is a simpler algorithm:

Method: Counting the metric by joining paths

We denote the vertices of a graph by $0, 1, 2, \ldots, N-1$.

- Let `d[i][j]` equal 1 (optionaly to the length of edge ij), or ∞ if edge ij is not in the graph.
- After each iteration $t \geq 0$ contains `d[i][j]` the length of the shortest path between i, j which passes only through vertices in $\{0, 1, 2, \ldots, t\}$.
- During each iteration t we may modify the distance between every pair of vertices, there are two options:
  1) we find a shorter way through the newly added vertex t; we replace `d[i][j]` by a shorter length `d[i][t] + d[t][j]`, or
  2) adding the vertex t does not help to find a way shorter than `d[i][j]` obtained in the previous steps; then `d[i][j]` remains unaltered.

Floyd's Algorithm – shortest paths

```
input: adjacency matrix G[][] of a graph with N vertices,
       where G[i][j]=1 for edge ij and
             G[i][j]=0 otherwise;

// initialization (value MAX_INT/2 stands for "infinity")
for (i=0; i<N; i++)
     for (j=0; j<N; j++)
          d[i][j] = (i==j ? 0 : (G[i][j] ? 1 : MAX_INT/2));

// loop for every vertex t, index from [0,N-1]
for (t=0; t<N; t++)
   // traverse all pairs of vertices
     for (i=0; i<N; i++)
          for (j=0; j<N; j++)
         // is there a shorter path through t?
               d[i][j] = min(d[i][j], d[i][t]+d[t][j]);
```

In the computer we implement ∞ by a large constant, i.e. `MAX_INT/2`.

Advantages:

- easy implementation
- finds the distance between every pair of vertices

Disadvantages:

- even when searching only the distance of two vertices, we *have to* find the distance of every pair of vertices
- complexity of $O(n^3)$, where n is the number of vertices
- doesn't provide shortest paths, just distances
  (can't reconstruct the path based on the result only)

Weighted distance

We assign numbers to edges: length, width, capacity, color, ...

Definition

Labeling of a given graph G is a mapping $w : E(G) \to \mathbb{R}$, which assigns a real number $w(e)$ (called edge weight/label) to every edge of G. Weighted (or labeled) graph is a graph G along with a labeling.
In a positively weighted (labeled) graph G are all weights $w(e)$ positive ($w(e) > 0$ pro $\forall e \in E(G)$).

Edge weight – more commonly "labels".

In real life applications:

- labels are usually non-negative,
- we can use integers only when choosing a suitable scale (units).

Positively weighted (labeled) graph is a special case of a labeled graph.

Now we introduce distances in weighted graphs.

Definition

Let G be a weighted graph G with labeling w.
The length of a weighted walk $S = v_0, e_1, v_1, e_2, v_2, \dots, e_n, v_n$ in G is the sum

$$d_G^w(S) = w(e_1) + w(e_2) + \cdots + w(e_n)$$

(each edge is counted as may times as it appears in the walk S).
(Weighted) distance between two vertices u, v in a weighted (positively labeled) graph (G, w) is

$$\mathrm{dist}_G^w(u, v) = \min\{d_G^w(S), \text{where } S \text{ is a } \textbf{path} \text{ between } u \text{ and } v\}.$$

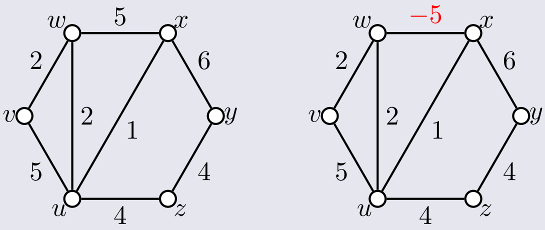
If vertices u and v are unreachable, we set $\mathrm{dist}_G^w(u, v) = \infty$.

Lemma

Weighted distance in positively weighted graphs satisfies the *triangle inequality* $\forall u, v, w \in V(G)\colon \ \mathrm{dist}_G^w(u, w) \leq \mathrm{dist}_G^w(u, v) + \mathrm{dist}_G^w(v, w)$.

Why only non-negative weights?

Example



Two different labelings of G.

Questions

- What is the distance between v and y in the graph on the left? 13? 12? 11? 10?
- What is the distance between w and z?
  We do not allow negative weights, since then no shortest walk has to exists.
- What is the distance between v and y in the graph on the right? 3?, 0?, -1?, 10? $-n$?

Shortest path algorithm

For finding a shortest (weighted) path between two vertices of a positively weighted graph Dijkstra's algorithm is used.

- more complex than the algorithm above
- is *significantly faster*; finds the distance from a particular vertex to all other vertices, not between all pairs of vertices

Dijkstra's algorithm is used while searching connections in on-line search engines.

Dijkstra's algorithm

- is a modification of the breadth-first search algorithm – for each vertex v found we store the value of *distance* (length of the shortest u, v-path) from the vertex u, as well as the last vertex on this path.
- From the depository we always pick the vertex v *with the smallest distance* from u (no shorter u, v-path exists).
- After the search we have the distance form u to all vertices of the graph.

Dijkstra's algorithm (initialization)

Finds the shortest path between u and v of a positively weighted graph G (given by the incidence matrix).

```
input: graph on N vertices, in an incidence matrix neig[][]
       and w[][], where neig[i][0], ..., neig[i][deg[i]-1]}
       are neighbors of vertex i with degree deg[i] and edge
       from i to neig[i][k] has length w[i][k] > 0;
input: u,v, we search path from u to v;
// state[i] stores the state of vertex i:
//    0 ... initial
//    1 ... processed
// dist[i] gives the shotest (so far) distance to i
// pre[i]  contains the predecessor of i

// initialization
for (i=0; i<=N; i++)   // MAX_INT also to dist[N]!
     { dist[i] = MAX_INT; state[i] = initial; }
dist[u] = 0;
```

Dijkstra's algorithm (continued)

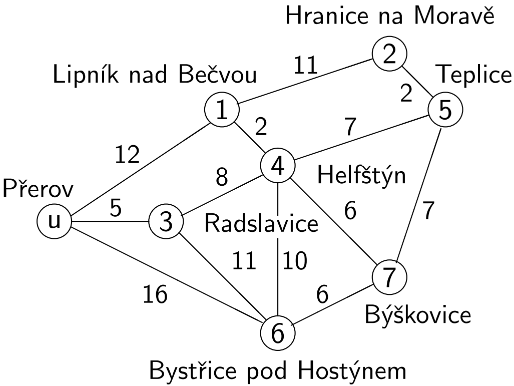
```
while (state[v] == initial) {
   for (i=0, j=N; i<N; i++)     // dist[N] = MAX_INT
      if (state[i] == initial && dist[i] < dist[j])
         j = i;
   // we have the closest unprocessed vertex j
   // process it
   if (dist[j] == MAX_INT) return NO_PATH;
   state[j] = processed;
   for (k=0; k<deg[j]; k++)
      if (dist[j]+w[j][k] < dist[neig[j][k]]) {
         dist[neig[j][k]] = dist[j]+w[j][k];
         pre[neig[j][k]] = j;
      }
      // field pre[] containfs information about
      // predecessors on the shortest path
}
output: Path of length dist[v] stored recursively in pre[];
```

Notes to Dijkstra's algorithm

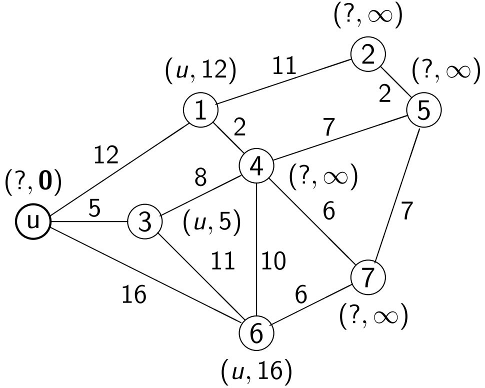
- Running the loop not with the condition `state[v] == initial`, but until all vertices are processed, the algorithm gives the shortest path and its length from u to all vertices. This information is stored in `dist[]` and `pre[]`.
- The total number of steps in Dijkstra's Algorithm for finding the shortest path from `u` to `v` is approximately N^2, where N in the number of vertices.
- Implementing the depository in a convenient way (e.g. heap with the distance as a key) an even faster implementation can be achieved on sparse graphs – running time is approximately the number of edges.
- Algorithm works also for *oriented graphs*.
- We can modify it easily also for *widest road*.

An example follows. . .

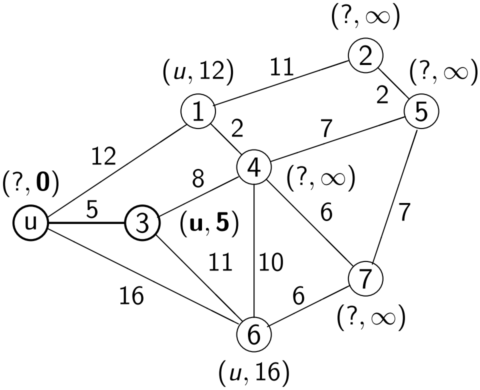
Take the road map close to Přerov. We search for distance from Přerov to all other places.



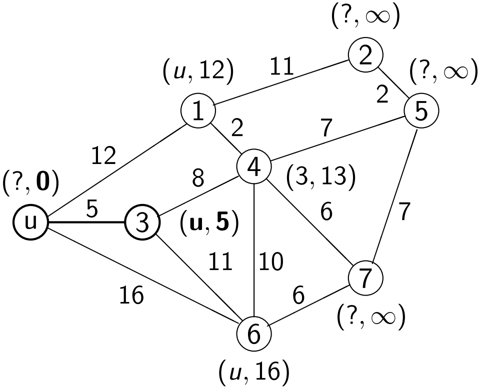
This is a graph representation of the road map. Edges in the graph are labeled by distances in kilometers. Vertices represent cities and roads are depicted by edged joining the corresponding vertices. Vertex i will be labeled by $(pre[i], dist[i])$.



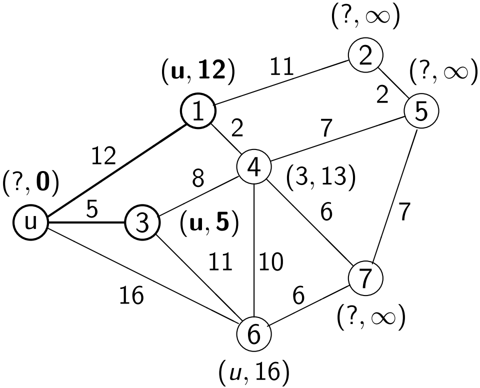
In the initial step of Dijkstra's algorithm each vertex will be in the state 0 (initial state). Only the starting vertex u will have distance 0, i.e. labeled by $(0, 0)$. All remaining vertices are labeled by $(?, \infty)$.
In the first step all vertices j, adjacent to u will be labeled by $(s, w[s][j])$.



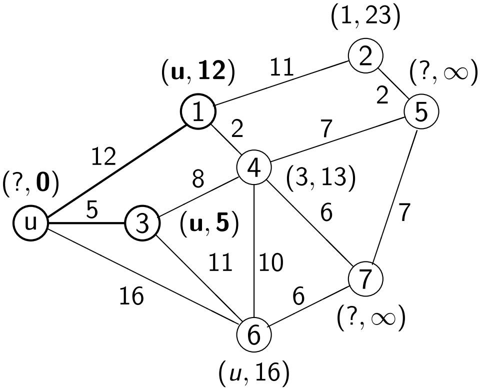
Next we choose the vertex j, which has from u the distance. This is the vertex 3.



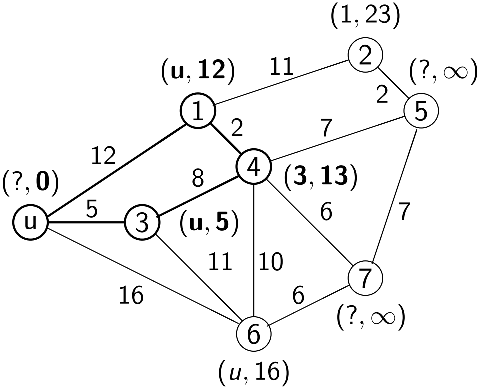
In the next step we modify the label of neighbors of 3 (the closest unprocessed vertex).
We modify the label of vertex 4. The new label of vertex 4 will be $(3, 13)$.
The label of vertex 6 will not be changed. The vertex u is also adjacent to 3, but it is processed and its label will change no more.



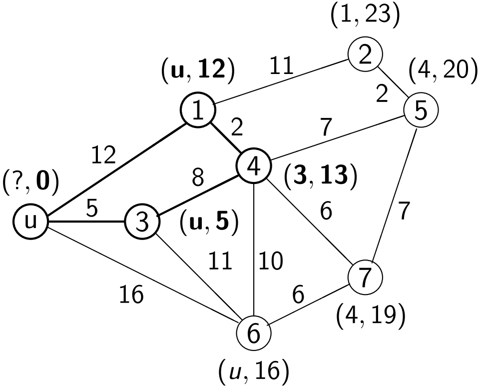
Next we pick the vertex j, with the closest distance from u.
This is the vertex 1 ($dist[1] = 12$).



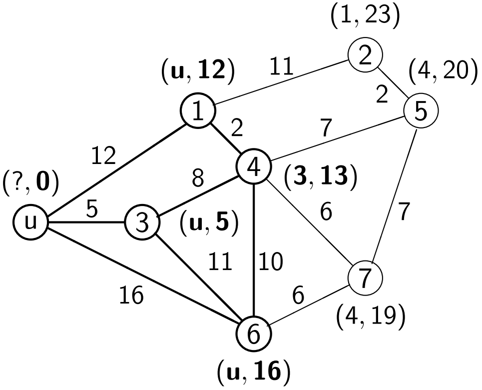
Now vertex 2 will be labeled $(1, 23)$, since $\infty > dist[1] + w[1][2]$. But the label of 4 will not be changed.



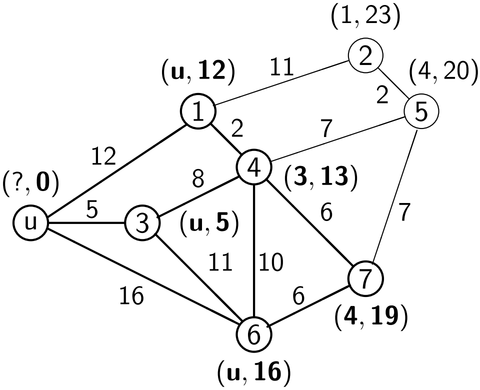
Vertex 4 is the closest to u, it will be processed next.



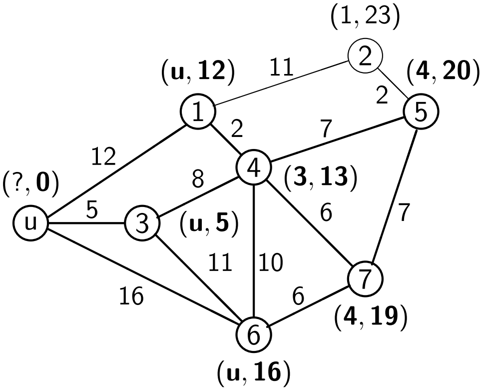
The unprocessed neighbors of vertex 4 are 5, 6 and 7. Since $dist[5] > dist[4] + w[4][5]$ $(\infty > 13 + 7)$, we label vertex 5 by $(4, 20)$. The label of vertex 6 will not change. The vertex 7 will be labeled by a new label $(4, 13 + 6)$.



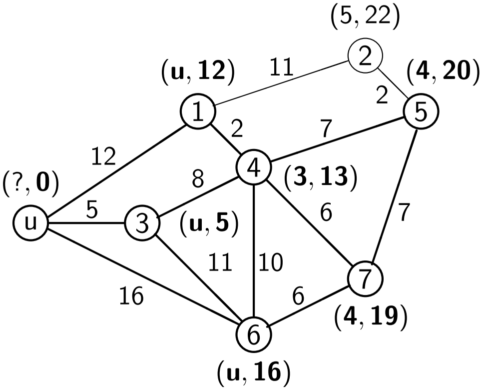
Closest to vertex u is now the vertex 6. The remaining unprocessed vertices 2, 5 and 7 have a higher $dist[i]$.
We will not modify any label, no distance can be improved!
Note: If there are more vertices with the same distance, we choose one arbitrarily.



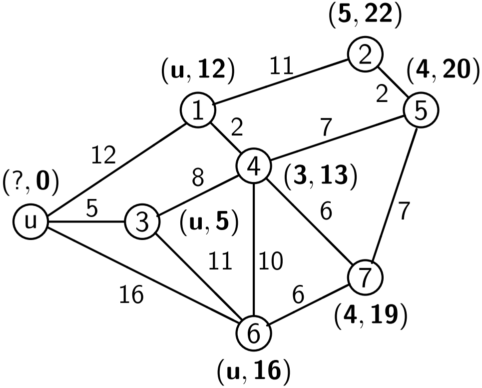
Closest to vertex u is now the vertex 7 ($dist[7] = 19$). Again no label will be modified.



Closest to vertex u is the vertex 5 since $dist[5] < dist[2]$ $(20 < 23)$. We process it.



Last unprocessed vertex is now vertex 2.
Since $dist[2] > dist[5] + w[5][2]$ $(23 > 22)$ the new label of vertex 2 will be $(5, 22)$.



Now (the only) vertex closest to u is vertex 2. We modify its state and the algorithm stops.
We have found the distance from u to all vertices in the graph.

Proof that Dijkstra's Algorithm works correct

Theorem

Let G be a (positively) weighted graph and let u and v be two vertices in G. Dijkstra's Algorithm finds the shortest path from vertex u to vertex v.

Proof

By S we denote the set of processed vertices.
Key observation is that after each iteration gives `dist[i]` the distance from `u` to `i` traversing only all *processed* vertices in S. These distances are the same when traversing any vertices in G.

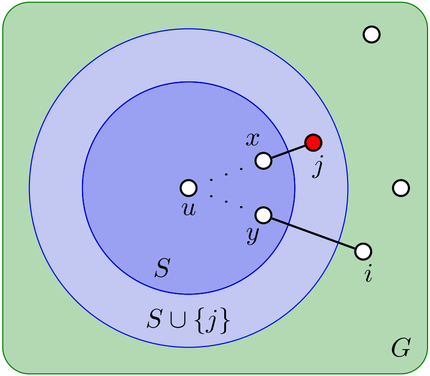
We proceed by induction on the number of iterations:

Basis step: In the first iteration of Dijkstra's Algorithm the only vertex in the depository is u. We process it and modify the distance to its neighbors based on edge weights adjacent to u.

The claim holds trivially, since after the iteration $S = \{u\}$ and all the distances through vertices in S only are minimal.

Proof (continued)

Inductive step: In every subsequent iteration we choose from the depository the vertex `j` with the distance to vertex `u`.
At the same time no shorter path to `j` exists, all paths through unprocessed vertices has to be longer, no shortcut through more distant vertices is not possible due the choice of `j`.



Here we use the that the weights `w[][]` are positive, through `i` the paths have to be longer than through `j`. The claim follows by induction. □

Next lecture

Chapter Trees and forest

- motivation
- basic tree properties
- rooted trees
- isomorphism of trees
- spanning trees of graphs