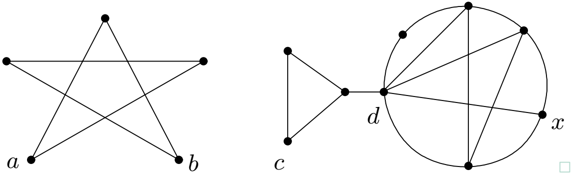
# 3 Distance in Graphs

While the previous lecture studied just the connectivity properties of a graph, now we are going to investigate how "long" (short, actually) a connection in a graph is.

This naturally leads to the concept of graph distance, which has two variants: the simple one considering only the number of edges, while the weighted one having a "length" for each edge.




## Brief outline of this lecture

- Distance in a graph, basic properties, triangle inequality.
- Graph metrics: all-pairs shortest distances.
- Dijkstra's algorithm for the shortest weighted distance in a graph.
- Route planning: a sketch of some advanced ideas.

## 3.1 Graph distance (unweighted)

Recall that a walk of length $n$ in a graph $G$ is an alternating sequence of vertices and edges $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$ such that each $e_i$ has the ends $v_{i-1}, v_i$.

**Definition 3.1.** **Distance $d_G(u,v)$** between two vertices $u, v$ of a graph $G$ is defined as the length of the shortest walk between $u$ and $v$ in $G$.
If there is now walk between $u, v$, then we declare $d_G(u,v) = \infty$. □

Informally and naturally, the distance between $u, v$ equals *the least possible number of edges* traversed from $u$ to $v$. Specially $d_G(u,u) = 0$.

Recall, moreover, that the shortest walk is always a path – Theorem 2.2.

**Fact**: The distance in an undirected graph is symmetric, i.e. $d_G(u,v) = d_G(v,u)$. □

**Lemma 3.2.** *The graph distance satisfies the triangle inequality:*

$$\forall u, v, w \in V(G) \;\;:\;\; d_G(u,v) + d_G(v,w) \geq d_G(u,w)\,.$$ □

**Proof**. Easily; starting with a walk of length $d_G(u,v)$ from $u$ to $v$, and appending a walk of length $d_G(v,w)$ from $v$ to $w$, results in a walk of length $d_G(u,v) + d_G(v,w)$ from $u$ to $w$. This is an upper bound on the real distance from $u$ to $w$. □

## How to find the distance

**Theorem 3.3.** *Let $u, v, w$ be vertices of a connected graph $G$ such that $d_G(u,v) < d_G(u,w)$. Then the breadth-first search algorithm on $G$, starting from $u$, finds the vertex $v$ before $w$.* □

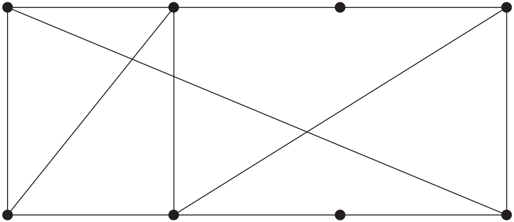
**Proof**. We apply induction on the distance $d_G(u,v)$: If $d_G(u,v) = 0$, i.e. $u = v$, then it is trivial that $v$ is found first. So let $d_G(u,v) = d > 0$ and $v'$ be a neighbour of $v$ closer to $u$, which means $d_G(u,v') = d-1$. Analogously choose $w'$ a neighbour of $w$ closer to $u$. Then

$$d_G(u,w') \geq d_G(u,w) - 1 > d_G(u,v) - 1 = d_G(u,v'),$$

and so $v'$ has been found before $w'$ by the inductive assumption. Hence $v'$ has been stored into $U$ before $w'$, and (cf. FIFO) the neighbours of $v'$ ($v$ among them, but not $w$) are found before the neighbours of $w'$ (such as $w$). □ □

**Corollary 3.4.** *The breadth-first search algorithm on $G$ correctly determines graph distances from the starting vertex.*

## Other related terms



**Definition 3.5.** Let $G$ be a graph. We define, with resp. to $G$, the following notions:

- The *excentricity* of a vertex $\text{exc}(v)$ is the largest distance from $v$ to another vertex; $\text{exc}(v) = \max_{x \in V(G)} d_G(v, x)$. □
- The *diameter* $\text{diam}(G)$ of $G$ is the largest excentricity over its vertices, and the *radius* $\text{rad}(G)$ of $G$ is the smallest excentricity over its vertices. □
- The *center* of $G$ is the subset $U \subseteq V(G)$ of vertices such that their excentricity equals $\text{rad}(G)$.

## 3.2 All-pairs shortest distances

**Definition**: The *metrics* of a graph is the collection of distances between all pairs of its vertices. In other words, the metrics is a matrix `d[,]` such that `d[i,j]` is the distance from $i$ to $j$. □

**Method 3.6. Dynamic programming for all-pairs distances**
*in a graph $G$ on the vertex set $V(G) = \{v_0, v_1, \dots, v_{N-1}\}$.*

- Initially, let `d[i,j]` be $1$ (alternatively, the edge length of $\{v_i, v_j\}$), or $\infty$ if $v_i, v_j$ are not adjacent. □
- After step $t \geq 0$ let it hold that `d[i,j]` is the shortest length of a walk between $v_i, v_j$ such that its internal vert. are from $\{v_0, v_1, \dots, v_{t-1}\}$ (empty for $t = 0$). □
- Moving from step $t$ to $t + 1$, we update all the distances as:
  - Either `d[i,j]` from the previous step is still optimal (the vertex $v_t$ does not help to obtain a shorter walk from $v_i$ to $v_j$), or
  - there is a shorter $v_i$ to $v_j$ walk using (also) the vertex $v_t$ which is, by the assumption at step $t$, of length `d[i,t]+d[t,j]` $\rightarrow$ `d[i,j]`. □

**Theorem 3.7.** *Method 3.6 correctly computes the distance* `d[i,j]` *between each pair of vertices $v_i, v_j$ in $N = |V(G)|$ steps.*

**Remark**: In a practical implementation we may use, say, `MAX_INT/2` in place of $\infty$.

**Algorithm 3.8. Floyd–Warshall algorithm (cf. 3.6)**

input < *the adjacency matrix* `G[,]` *of an* `N`*-vertex graph, such that the vertices of* `G` *are indexed as* `0...N-1`*, and* `G[i,j]=1` *if* $i,j$ *adjacent and* `G[i,j]=0` *otherwise*;

```
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));
for (t=0; t<N; t++) {
    for (i=0; i<N; i++) for (j=0; j<N; j++)
        d[i,j] = min(d[i,j], d[i,t]+d[t,j]);
}
return 'The distance matrix d[,]'; □
```

Notice that this Algorithm 3.8 is extremely simple and relatively fast — it needs about $N^3$ steps to get the whole distance matrix.

Its only problem is that all-pairs distances must be computed at the same time, even if we need to know just one distance...

## 3.3 Weighted distance in graphs

**Definition**: A *weighted graph* is a pair of a graph $G$ together with a weighting $w$ of the edges by real numbers $w : E(G) \to \boldsymbol{R}$ (edge lengths in this case).
A *positively weighted graph* $(G, w)$ is such that $w(e) > 0$ for all edges $e$. □

**Definition 3.9. (Weighted distance)** Consider a positively weighted graph $(G, w)$. The length of the weighted walk $S = v_0, e_1, v_1, e_2, v_2, \dots, e_n, v_n$ in $G$ is the sum

$$d^w_G(S) = w(e_1) + w(e_2) + \cdots + w(e_n)\,.$$

The *weighted distance* in $(G, w)$ between a pair of vertices $u, v$ is

$$d^w_G(u, v) = \min\{d^w_G(S) : S \text{ is a walk from } u \text{ to } v\}\,.$$ □

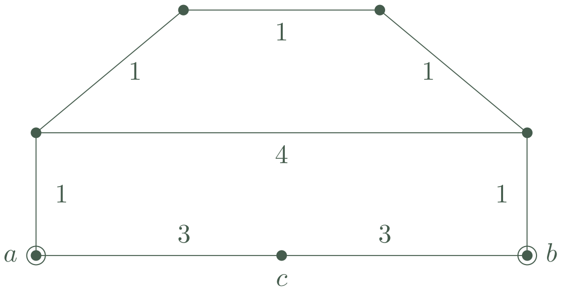
All these terms naturally extend from graphs to directed graphs. □

Analogously to Section 3.1 we get:

**Fact**: The shortest walk in a positively weighted (di)graph is always a path. □

**Lemma 3.10.** *The weighted distance in a positively weighted (di)graph satisfies the triangle inequality.*

See an example. . .



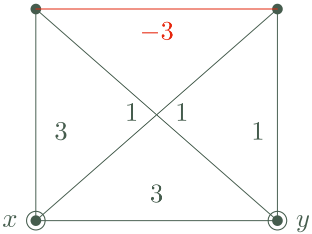

The distances between $a$–$c$ and between $b$–$c$ are $3$. What about the $a$–$b$ distance?

Is it $6$? No, the distance from $a$ to $b$ in the graph is $5$ (traverse the "upper path").

Furthermore, notice that this example also shows that simple BFS cannot correctly compute the shortest weighted distance.

## Negative edge-lengths?

What is the reason we are avoiding negative edge lengths?




Hence, what is the $x$–$y$ distance this graph? Say, $3$ or $1$? □

No, it is $-\infty$, precisely by Definition 3.9, and this answer does not sound nice. . . □

Hence we have got a good reason not to consider negative edges in general.

## 3.4 Single-source shortest paths problem

This section deals with the more specific problem of finding the shortest distance between one pair of terminals in a graph (or, from a single source to all other vertices).

**Remark**: The coming Dijkstra's algorithm is, on one hand, slightly more involved than Algorithm 3.8, but it is significantly faster in the computation of *single-source shortest distances*, on the other hand. □

### Dijkstra's algorithm:

- Is a variant of graph searching (related to BFS), in which every discovered vertex carries a *variable keeping its temporary distance*—the length of the shortest so far discovered walk reaching this vertex from the starting vertex. □
- We always pick from the depository the vertex with the shortest temporary distance. This is because no shorter walk may reach this vertex (assuming nonnegative edge lengths). □
- At the end of processing, the temporary distances become final shortest distances from the starting vertex (cf. Theorem 3.13). □
- Notice that this algorithm works as-is in directed graphs.

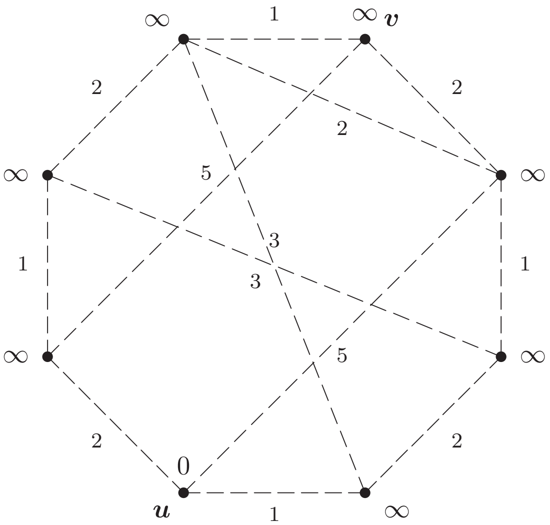
**Algorithm 3.12. Computing the single-source shortest paths (Dijkstra),**
*i.e. finding the shortest walk from $u$ to $v$, or from $u$ to all other vertices.*

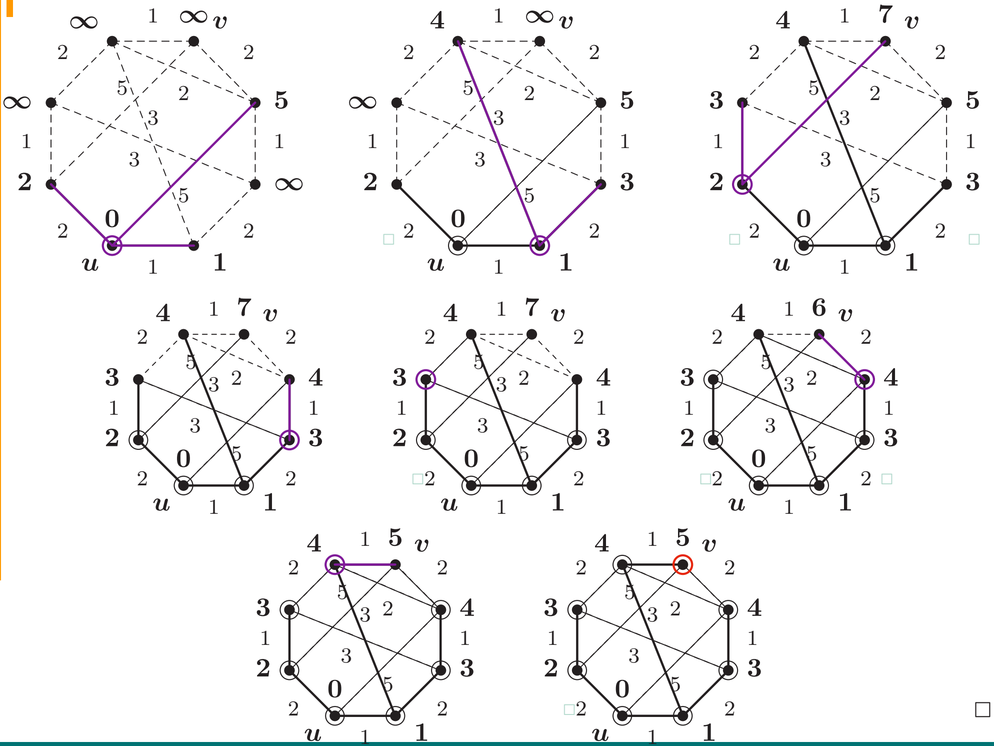
```
input < N-vertex graph G given by adjacency-length matrix len[,] ≥ 0,
        where len[i,j]=∞ if j is not an out-neighbour of i;
input < u,v, where u is the starting vertex and v the destination;

// state[i] records the vertex processing state, dist[i] is the temporary distance
for (i=0; i<N; i++) { dist[i] = MAX; state[i] = init; }
dist[u] = 0;  depository D = {u};
while (state[v]!=processed) {
    if (D==∅)  return 'No path';
    select m ∈ D  with minimal dist[m];
    // now updating all neighbours of m  and their temporary distances
    foreach (k out-neighbour of m) {
        D = D ∪ {k};
        if (dist[m]+len[m,k]<dist[k]) {
            income[k] = m;
            dist[k] = dist[m]+len[m,k];
        }
    }
    state[m] = processed;   D = D \ {m};
}
output 'A u-v path of length dist[v], stored in income[] reversely';
```

## Simple example

**Example 3.15.** *An illustration run of Dijkstra's Algorithm 3.12 from $u$ to $v$ in the following graph.*





**Fact**: The number of steps performed by Algorithm 3.12 to find the shortest path from `u` to `v` is about $N^2$ in rough impl., where $N$ is the number of vertices (not so good. . .).

On the other hand, with a better implementation of the depository, one can achieve on sparse graphs almost linear runtime; $O\big(|E(G)| + N \log N\big)$.

**Theorem 3.13.** *Every iteration of Algorithm 3.12 (since just after finishing the first* `while()` *loop) maintains an invariant that*

- `dist[i]` *is the length of a shortest path from* `u` *to* `i` *using only those internal vertices* $x$ *of* `state[x]==`*processed.*

**Proof**: Briefly using *mathematical induction*:

- In the first iteration, the first vertex `m=u` is picked and processed, and its neighbours receive the correct straight distances (edge lengths).
- In every next iteration, the picked vertex `m` is the nearest unprocessed one to the starting vertex `u`. Assuming nonnegative costs `len[,]`, this certifies that no shorter walk from `u` to `m` may exist in the graph.

  On the other hand, any improved path from `u` to an unfinished vertex `k` passing through `m` has `mk` as the last edge (since the distance of `m` is not smaller than of the other finished vertices). Hence `dist[k]` is updated correctly in the algorithm.

□

## 3.5 Advanced route planning

- Although being quite fast and, actually, "almost optimal" for the shortest path problem in weighted graphs, *Dijkstra's algorithm* turns out to be too slow for practical route planning applications in navigation devices containing map data of tens or hundreds millions of edges. □

- So, what can be done better? □

- An answer lies in *preprocessing* of the graph:

  It is quite natural to assume that the graph (of a road network) is relatively stable, and hence it can be thoroughly preprocessed on powerful computers. □However, where the preprocessing results can be stored? It is, say, completely unrealistic to store all the optimal routes in advance... □

- Two perhaps simplest approaches will be briefly sketched next.

First, a better alternative to Dijkstra's alg.—the *Algorithm $A^*$*, which uses a suitable *potential function* to direct the search "towards the goal". Whenever we have a good "sense of direction" (e.g. in a topo-map navigation), $A^*$ can perform much better!

## Algorithm $A^*$

- It re-implements Dijkstra with suitably modified edge costs. □
- Let $p_v(x)$ be a potential function giving an arbitrary lower bound on the distance from $x$ to the destination $v$. E.g., in a map navigation, $p_v(x)$ may be the Euclidean distance from $x$ to $v$. □
- Each directed(!) edge $xy$ of the weighted graph $(G, w)$ gets a new cost

$$\boldsymbol{w'(xy) = w(xy) + p_v(y) - p_v(x)\,.}$$

  The potential $p_v$ is *admissible* when all $w'(xy) \geq 0$, i.e. $w(xy) \geq p_v(x) - p_v(y)$. The above Euclidean potential is always admissible. □
- The modified length of any $u$-$v$ walk $S$ then is $d_G^{w'}(S) = d_G^w(S) + p_v(v) - p_v(u)$, which is a constant difference from $d_G^w(S)$! Hence some $S$ is optimal for the weighting $w$ iff $S$ is optimal for $w'$.

  Here the Euclidean potential "strongly prefers" edges in the destin. direction. Other (preprocessed) potential functions are possible as well, though.

Second, . . .

## The idea of a "reach"

- It is based on a natural observation that for long-distance route planning, vaste majority of edges of real-world road maps are basically irrelevant.

**Definition**: Let $S_{u,v}$ denote a shortest walk from $u$ to $v$ in weighted $G$. For $e \in E(S_{u,v})$ let $prefix(S_{u,v}, e)$, $suffix(S_{u,v}, e)$ denote the starting (ending) segment of $S_{u,v}$ up to (after) $e$. The *reach of an edge* $e \in E(G)$ is given as

$$reach_G(e) = \max\Big\{ \min\big( d_G^w(prefix(S_{u,v}, e)), d_G^w(suffix(S_{u,v}, e))\big) : \\ \forall u, v \in V(G) \wedge e \in E(S_{u,v}) \Big\}.$$

The reach of $e$ mathematically quantifies (ir)relevance of $e$ for route planning; the smaller $reach_G(e)$ is, the closer to the start or end of an optimal route $e$ has to be.

The immediate use of precomputed reach values is as follows:

- The line "foreach ( k *out-neighbour of* m)" (Algorithm 3.12) simply takes only those neighbours k such that $reach_G(\mathtt{mk}) \geq \mathtt{dist[m]}$.
- It is then important to employ the so-called bidirectional variant of Dijkstra / $A^*$.