

### 3 Graph Distance and Path Finding

In some other applications, graphs are used to model distances; e.g., as in road networks and in workflow diagrams. The basic task then is to find shortest paths or routes, and the optimal distance.



#### Brief outline of this lecture

- Distance in a graph, basic properties, BFS.
- Weighted distance in digraphs; the problem of negative cycles and Bellman–Ford’s algorithm.
- Dijkstra’s algorithm for the single-source shortest paths.
- A sketch of some advanced ideas in practical path planning.

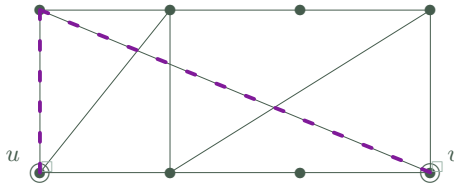
## 3.1 Unit Distance in Graphs

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

**Definition 3.1.** The *distance*  $d_G(u, v)$  between two vertices  $u, v$  of a graph  $G$  is defined as the length of a *shortest walk* between  $u$  and  $v$  in  $G$ .

If there is **no** walk between  $u, v$ , then we declare  $d_G(u, v) = \infty$ .  $\square$

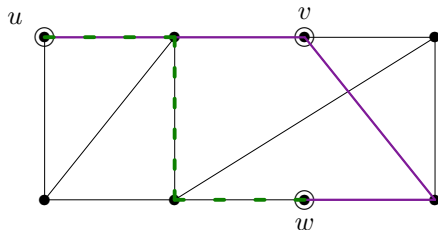
Naturally, the distance between  $u, v$  equals *the least possible number of edges* travelled from  $u$  to  $v$ , and it is always **achieved by a path**, as shown in Lemma 2.6. Spec.  $d_G(u, u) = 0$ .



**Remark:** Distance can be analogously defined for **digraphs**, using directed walks or paths.

A more general view in Section 3.2 will consider also **non-unit lengths** of edges in  $G$ .

## Triangle inequality



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

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

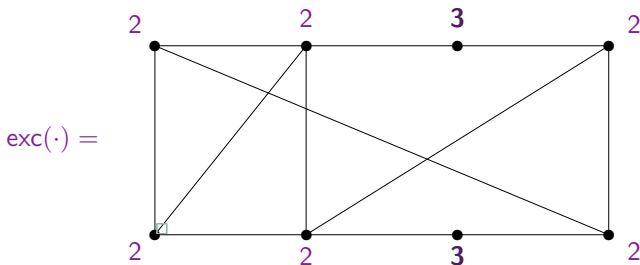
**Proof.** Easily; starting with a walk of length  $d_G(u, v)$  from  $u$  to  $v$ , and appending a walk of length  $d_G(v, w)$  from  $v$  to  $w$ , results in a walk of length  $d_G(u, v) + d_G(v, w)$  from  $u$  to  $w$ . This is an upper bound on the distance from  $u$  to  $w$ .  $\square$

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

## Other related terms

**Definition 3.3.** 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;  $\text{exc}(v) = \max_{x \in V(G)} d_G(v, x)$ .  $\square$
- 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.



It always holds  $\text{diam}(G) \leq 2 \cdot \text{rad}(G)$ .  $\square$

- The **center** of  $G$  is the subset  $U \subseteq V(G)$  of vertices such that their excentricity equals  $\text{rad}(G)$ .

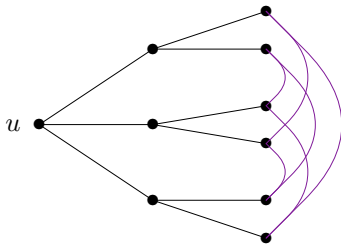
*diameter = průměr, radius = poloměr  
excentricity = excentricita*

## An exercise

**Example 3.4.** What is the largest possible number of vertices a *cubic* (i.e., 3-regular) graph of radius 2 may have?  $\square$

Let  $G$  be the graph. First of all, the definition of radius tells us that, for some vertex  $u \in V(G)$ , all the vertices of  $G$  are at distance  $\leq 2$  from  $u$ .  $\square$

Second, there can be  $\leq 10$  such vertices by the degree-3 condition:



$\square$

And third, we are able (or *lucky*?) to fill in the remaining six edges (in order to get all the degrees equal to 3) as in the picture. Hence, 10 vertices is possible, and this is the answer.  $\square$

**Remark:** Note, moreover, that we have actually constructed a graph of *diameter 2*, which is a stronger requirement than *radius 2*.

## Simple Computation of Distance (BFS)

Computing the (unit) distance from a given vertex  $u_0$  to any other vertex of a graph is a matter of an extremely simple algorithm, based on BFS:

**Algorithm 3.5.** *Computing all distances from a starting vertex  $u_0 \in V(G)$ .*  $\square$   
For a given graph (or digraph)  $G$  and any  $u_0 \in V(G)$ , we run Algorithm 2.1 with the implementation of `PROCESS(v; e)` as follows (and with void `PROCESS(e)`):

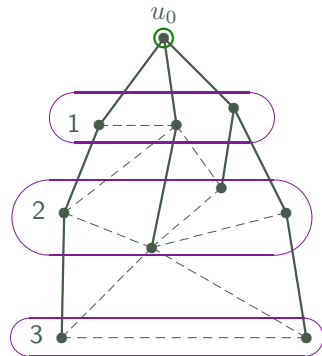
$U$  as a fifo queue (BFS), and

initialize  $\text{dist}[u_0, v] \leftarrow \infty$ , for all  $v \in V(G)$ ;

$\text{dist}[u_0, u_0] \leftarrow 0$ ;

...

```
PROCESS(v; e) {  
    u ← the starting vertex of 'e = uv';  
    dist[u_0, v] ← dist[u_0, u] + 1;  
}
```



\* jednoduchý výpočet vzdálenosti přes BFS \*

## BFS distance – the proof

**Theorem 3.6.** *Let  $u_0, v, w$  be vertices of a connected graph  $G$  such that  $d_G(u_0, v) < d_G(u_0, w)$ . Then the breadth-first search algorithm on  $G$ , starting from  $u_0$ , discovers the vertex  $v$  before  $w$ .  $\square$*

**Proof.** We apply induction on the distance  $d_G(u_0, v)$ : If  $d_G(u_0, v) = 0$ , i.e.  $u_0 = v$ , then it is trivial that  $v$  is found first. So let  $d_G(u_0, v) = d > 0$  and  $v'$  be a neighbour of  $v$  closer to  $u_0$ , which means  $d_G(u_0, v') = d - 1$ . Analogously choose  $w'$  a neighbour of  $w$  closer to  $u_0$ . Then

$$d_G(u_0, w') = d_G(u_0, w) - 1 > d_G(u_0, v) - 1 = d_G(u_0, v'), \square$$

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 discovered before the neighbours of  $w'$  (which include  $w$ ).  $\square$   $\square$

**Corollary 3.7.** *The search tree of the BFS Algorithm 2.1 on  $G$  determines the distances from  $u_0 \in V(G)$  to all vertices of  $G$ .*

*Hence, Alg. 3.5 is correct, meaning that  $\text{dist}(u_0, v) = d_G(u_0, v)$  for all  $v \in V(G)$ .*

## 3.2 Weighted Distance in Digraphs

Recall (Section 2.3): A *weighted (di)graph* is a pair of a (di)graph  $G$  together with a weighting  $w$  of the edges by real numbers  $w : E(G) \rightarrow \mathbf{R}$  (edge **lengths** in this case).

A *positively weighted (di)graph*  $(G, w)$  is such that  $w(e) > 0$  for all edges  $e$ .  $\square$

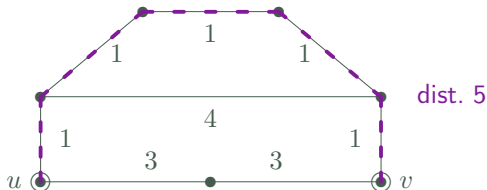
**Definition 3.8. Weighted distance (length)** in a weighted (di)graph  $(G, w)$ .

The *length* of a weighted (dir.) walk  $S = (v_0, e_1, v_1, e_2, v_2, \dots, e_n, v_n)$  in  $G$  is the sum

$$d_G^w(S) = w(e_1) + w(e_2) + \dots + w(e_n). \square$$

The *weighted distance* in  $(G, w)$  from a vertex  $u$  to a vertex  $v$  is

$$d_G^w(u, v) = \min\{d_G^w(S) : S \text{ is a (directed) walk from } u \text{ to } v\}. \square$$

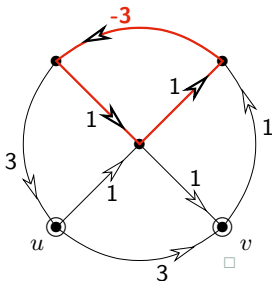


For **undir.** graphs  $G$ , the definition considers the symmetric orientation of the edges.



## Basic facts

- Weighted distance in a digraph  $(G, w)$  satisfies the **triangle inequality**.  
(The same statement and proof hold here as in Lemma 3.2.)  $\square$
- Ordinary **graph distance** is obtained for weights  $(G, w_1)$  s.t.  $w_1(e) = 1$  for all  $e$ .  $\square$
- If a weighted digraph  $(G, w)$  contains a cycle (a closed walk) of **negative length**, then the distance between a pair of vertices in  $G$  may **not** be defined (" $-\infty$ "):



**Proposition 3.9.** *If  $(G, w)$  is a weighted digraph containing no cycles of **negative length** (and hence no such closed walks), then  $\square$*

- *the weighted distance in  $(G, w)$  is always **well defined**, and  $\square$*
- *the weighted distance is achieved by a directed path in  $G$ .*

## Negative or positive weights?

- By the previous facts, negative-length edges may cause huge problems with graph distance. So, why to consider them at all?

(Do they make sense, anyway?) □

- For **undirected** graphs, the negative-length problem seems fatal, and hence we consider only positively weighted undirected graphs.

For **digraphs**, though, negative-length edges might be useful to consider, as long as there is **no cycle of negative length** (Prop. 3.9). E.g., for DAGs.

## Bellman–Ford Algorithm

**Definition:** A cycle of negative length in a weighted digraph is called a *negative cycle*. □

### Algorithm 3.10. *Computing the distance or detecting a negative cycle.*

For a given *weighted digraph*  $(G, w)$ , and a starting vertex  $u_0 \in V(G)$ , the task is to compute the *distance*  $\text{dist}[u_0, v] = d_G^w(u_0, v)$  from  $u_0$  to any vertex  $v \in V(G)$ .

```
initialize  $\text{dist}[u_0, v] \leftarrow \infty$ , for all  $v \in V(G)$ ;  
 $\text{dist}[u_0, u_0] \leftarrow 0$ ; □  
repeat  $|V(G)| - 1$  times {  
    foreach ( $e = tv \in E(G)$ ) {  
         $\text{dist}[u_0, v] \leftarrow \min(\text{dist}[u_0, v], \text{dist}[u_0, t] + w(e))$ ;    (*)  
    }  
} □  
foreach ( $e = tv \in E(G)$ ) {  
    if ( $\text{dist}[u_0, v] > \text{dist}[u_0, t] + w(e)$ )  
        output 'Error; a negative cycle exists in  $(G, w)$ .'  
}  
output 'Distances from  $u_0$  in  $\text{dist}[u_0, \cdot]$ .'
```

□

(One can also easily store the *predecessors* for the computed distances on line (\*). . .)

## Proof of the Bellman–Ford algorithm

**Proof.** To claim that  $\text{dist}[u_0, v] = d_G^w(u_0, v)$  if there is no negative cycle in  $(G, w)$ , and that a **negative cycle is detected** otherwise, we prove the following three steps.

1. At every step of Algorithm 3.10, it is  $\text{dist}[u_0, v] \geq d_G^w(u_0, v)$ :  $\square$

This holds at the beginning, and follows trivially by induction on the number of elementary steps ‘ $\text{dist}[u_0, v] \leftarrow \min(\text{dist}[u_0, v], \text{dist}[u_0, t] + w(e))$ ’. $\square$

2. Assume there is **no** negative dir. cycle in  $(G, w)$ . Let (cf. Prop. 3.9)  $V_k \subseteq V(G)$  be the subset of vertices  $v$  for which  $d_G^w(u_0, v)$  is **achieved** by a dir.  $u_0$ - $v$  path with  $\leq k$  edges. Then, after iteration no.  $k$  of ‘foreach ( $e = uv \in E(G)$ )’, the value of  $\text{dist}[u_0, v]$  equals  $d_G^w(u_0, v)$  for all  $v \in V_k$ :  $\square$

Again, this trivially holds for  $k = 0$  and follows easily by induction.  $\square$

3. Let  $C \subseteq G$  be any directed cycle. If no **negative cycle** is reported at the end of Alg. 3.10, i.e. ‘ $\text{dist}[u_0, v] \not\geq \text{dist}[u_0, u] + w(e)$ ’ for all  $e = uv \in E(C)$  in the last phase, then  $C$  is **not** a negative cycle in  $(G, w)$ :  $\square$

We have  $\text{dist}(u_0, v) - \text{dist}(u_0, t) \leq w(e)$  and summing these over all  $e \in E(C)$  we get  $0 \leq \sum_{e \in E(C)} w(e)$ . Consequently, negative cycles in  $(G, w)$  are detected in the algorithm (but only detected, they cannot be easily constructed).  $\square$

### 3.3 Positive-length Shortest Paths

In contrast to previous Algorithm 3.10, shortest paths may be computed **much faster** when all the edge lengths are **positive** (which is true, e.g., in practical *routing problems*).

For the typical, so-called *single-source positive-length shortest paths problem*, a nearly optimal algorithm is the following traditional one.

#### Dijkstra's algorithm:

- For a given **positively weighted digraph**  $(G, w)$ , and an arbitrary starting vertex  $u_0 \in V(G)$ , the algorithm computes  $dist[u_0, v]$  for all  $v \in V(G)$ . ◻
- In the graph-search scheme of Algorithm 2.1, one simply implements
  - ‘**choose**  $(e, u) \in U$ ’ by picking  $(e, u)$ ,  $e = tu$ , from  $U$  such that  $dist(u_0, t) + w(tu)$  is minimized, ◻
  - ‘**PROCESS**( $u; e=tu$ )’ as  $dist[u_0, u] \leftarrow dist[u_0, t] + w(tu)$ , ◻
  - ‘**PROCESS**( $e$ )’ as void, and
  - the search tree  $T$  then stores shortest paths from  $u_0$ . ◻
- This algorithm works in the same way for undirected as for directed graphs.

A self-contained exposition of **Dijkstra's algorithm** is quite simple:

**Algorithm 3.11. Dijkstra's for single-source shortest paths.**

For a positively weighted digraph  $(G, w)$ , and a vertex  $u_0 \in V(G)$ , compute shortest paths  $\text{predec}[\cdot]$  and distances  $\text{dist}[u_0, \cdot]$  in  $(G, w)$  from the source  $u_0$  to all of  $G$ .

```
initialize  $\text{dist}[u_0, v] \leftarrow \infty$ , for all  $v \in V(G)$ ;  
 $\text{dist}[u_0, u_0] \leftarrow 0$ ;  
 $U \leftarrow \{u_0\}$ ;  $\square$   
while ( $U \neq \emptyset$ ) {  
    choose  $u \in U$  minimizing  $\text{dist}[u_0, u]$ ;  
    foreach (edge  $f$  starting in  $u$ ) {  
         $v \leftarrow$  the opposite vertex of ' $f = uv$ ';  
        if ( $\text{dist}[u_0, u] + w(uv) < \text{dist}[u_0, v]$ ) {  
             $U \leftarrow U \cup \{v\}$ ;  
             $\text{predec}[v] \leftarrow u$ ;  
             $\text{dist}[u_0, v] \leftarrow \text{dist}[u_0, u] + w(uv)$ ;  
        }  
    }  
     $U \leftarrow U \setminus \{u\}$ ;  
}
```

output 'distances in  $\text{dist}[\cdot]$ , predecessors of shortest paths in  $\text{predec}[\cdot]$ ';

**Proposition 3.12.** If the stack  $U$  is implemented as a *minimum heap*, then the number of steps performed by Algorithm 3.11 is  $O(|E(G)| + |V(G)| \cdot \log |V(G)|)$ .  $\square$

A vertex  $u \in V(G)$  is called “*relaxed*” after it is removed in ‘ $U \leftarrow U \setminus \{u\}$ ’ above.

**Theorem 3.13.** Every iteration of Algorithm 3.11 maintains an invariant that

- $\text{dist}[u_0, v]$  is the length of a shortest path from  $u_0$  to  $v$  using only those *internal vertices which are relaxed*, and such a shortest path is stored in  $\text{predec}[\cdot]$ .  $\square$

Consequently, all the distances and shortest paths to reachable vertices are correct.

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

- In the first iteration of ‘ $\text{while } (U \neq \emptyset)$ ’,  $u_0$  is chosen and the straight distances (edge lengths) to its neighbours are stored.  $\square$
- Subsequently, for every chosen vertex  $u$  in ‘ $u \in U$  *minimizing*  $\text{dist}[u_0, u]$ ’, the current value of  $\text{dist}[u_0, u]$  is optimal since *no negative edges* exist in  $(G, w)$  (and so every possible detour via non-relaxed vertices would only be longer).  $\square$

Then, all working distances and the shortest-paths record are properly updated (wrt.  $u$ ) while “relaxing”  $u$ :

```
if (  $\text{dist}[u_0, u] + w(uv) < \text{dist}[u_0, v]$  ) {  
     $\text{predec}[v] \leftarrow u$ ;  $\text{dist}[u_0, v] \leftarrow \text{dist}[u_0, u] + w(uv)$ ;  
}
```

$\square$

## Bidirectional Dijkstra's algorithm

In some settings, the following improved variant may be significantly more efficient in the **single-pair shortest path problem** in a digraph  $(G, w)$ :  $\square$

- To find a shortest  $u_0$ - $v_0$  path, run **two instances** of Algorithm 3.11 concurrently:
  - $\mathcal{A}$  searches shortest paths from  $u_0$  in  $(G, w)$ , as usual, and  $\square$
  - $\mathcal{A}^\leftarrow$  searches shortest paths from  $v_0$  in  $(G^\leftarrow, w)$  where  $G^\leftarrow$  results from  $G$  by **reversing all edges**;  $e \in E(G)$  to  $e^\leftarrow \in E(G^\leftarrow)$  such that  $w(e^\leftarrow) = w(e)$ .  $\square$
- $\mathcal{A}$  and  $\mathcal{A}^\leftarrow$  may simply alternate their iterations, or better;
  - minima  $u \in U$  and  $u' \in U^\leftarrow$  are chosen concurrently, and the instance achieving **smaller value among  $dist(u_0, u)$  and  $dist^\leftarrow(v_0, u')$**  is run.  $\square$
- Termination condition;  
the whole algorithm stops when the search subtrees  $T$  and  $T^\leftarrow$  of  $\mathcal{A}$  and  $\mathcal{A}^\leftarrow$  meet each other. That is, whenever some vertex is relaxed in both  $\mathcal{A}$  and  $\mathcal{A}^\leftarrow$ .  $\square$

At this moment, a shortest  $u_0$ - $v_0$  path exists using only **relaxed vertices**.

Though, this does not mean that every shortest  $u_0$ - $v_0$  has to pass through the intersection of the two search trees – one still has to loop over the relevant edges  $uv$  to determine the minimum of  $dist(u_0, v) + w(uv) + dist^\leftarrow(v_0, v)$ .



## All-pairs Shortest Distances

The last algorithm we are going to present in this section is **extraordinarily simple**, although rather slow since it has to compute **all-pairs distances at once**.  $\square$

### Algorithm 3.14. *Floyd–Warshall’s algorithm for all-pairs distances*

For a positively weighted digraph  $(G, w)$ , compute distances  $\text{dist}[\cdot, \cdot]$  between all pairs of vertices of  $G$ .

```
initialize  $\text{dist}[u, v] \leftarrow \infty$ , for all  $u, v \in V(G)$ ;  
foreach  $(uv \in E(G))$   $\text{dist}[u, v] \leftarrow w(uv)$ ;  
foreach  $(t \in V(G))$  {  
    foreach  $(u, v \in V(G))$  {  
         $\text{dist}[u, v] \leftarrow \min(\text{dist}[u, v], \text{dist}[u, t] + \text{dist}[t, v])$ ;  
    }  
}
```

output 'The complete distance matrix of  $(G, w)$  in  $d[\cdot, \cdot]$ ';

The number of steps of this algorithm is  $O(|V(G)|^3)$ , which is quite slow compared to repeated Dijkstra in the case of sparse graphs.  $\square$

**Remark:** Floyd–Warshall’s algorithm has many shapes; it appears, e.g., in computation of the transitive closure and in the translation of a finite automaton to a regular expression.  $\square$

The algorithm is also related to *matrix multiplication*.

Algorithm 3.14 is based on the following beautifully simple dynamic-programming idea:

### Computing all-pairs distances dynamically

- Given is a weighted (di)graph  $(G, w)$  on  $n$  vertices;  $V(G) = \{t_0, t_1, \dots, t_{n-1}\}$ . $\square$   
Let  $dist^i(u, v)$  denote the length of a shortest  $u$ - $v$  walk  $S$  in  $G$  such that all vertices of  $S$  except the ends  $u, v$  are from the subset  $\{t_0, \dots, t_{i-1}\}$ .  $\square$
- For computing  $dist^{i+1}$ , the admissible walks are those as for  $dist^i$  plus those walks passing through  $t_i$  (“ $u$ - $t_i$ - $v$ ”).  $\square$

Consequently,

$$dist^{i+1}(u, v) = \min (dist^i(u, v), dist^i(u, t_i) + dist^i(t_i, v)) \square$$

and

$$dist[u, v] = dist^n(u, v). \square$$

- This algorithm works correctly also with negative edge lengths, as long as there is **no negative cycle** (same as Bellman–Ford):

**Proposition 3.15.** *Algorithm 3.14 correctly computes distances between all pairs of vertices in a weighted (di)graph  $(G, w)$ , provided that there is no negative cycle.*

## 3.4 Some Advanced Ideas in Path Finding

Based on the above comparison of approaches, *Dijkstra's algorithm* seems to be the ultimate tool for practical path finding (or *route planning*) problems.

- 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, e.g., 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, what of the preprocessing results **can be stored**? It is, say, completely unrealistic to store all the optimal routes in advance. . . □
- Two perhaps simplest practically usable approaches will be briefly sketched next.

First, an alternative to Dijkstra's alg. is 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 way much better!

## Algorithm *A\**

- In a basic setting, *A\** re-implements Dijkstra with suitably **modified edge costs** on digraphs. □
- Let  $p_v(x)$  be a potential function giving an arbitrary **lower bound** on the distance from  $x$  to the destination  $v$  (i.e.,  $p_v$  is *admissible*).  
E.g., in a map navigation,  $p_v(x)$  may be the Euclidean distance from  $x$  to  $v$ . □

- Each oriented edge  $xy$  of the weighted graph  $(G, w)$  gets a new cost

$$w'(xy) := w(xy) + p_v(y) - p_v(x).$$

The potential  $p_v$  is *consistent* when **all**  $w'(xy) \geq 0$ , i.e.  $w(xy) \geq p_v(x) - p_v(y)$ .  
The above Euclidean potential is always consistent. □

- The modif. 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  $d_G^w(S)$ . □

Consequently, 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 (also preprocessed) potential functions are possible as well, though.

Second, ...

## Idea of the “reach” parameter

- It is based on a natural observation that for long-distance route planning, vast 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

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

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:

- We must use the **bidirectional** variant of Dijkstra or  $A^*$ . □
- The line ‘foreach (edge  $f$  starting in  $u$ )’ in Algorithm 3.11 (in **each direction**) now takes only those edges  $f = uv$  such that  $reach_G(f) \geq dist[u_0, u]$ .

## 3.5 Appendix: An example run of Dijkstra's alg.

**Example 3.15.** An illustration run of Dijkstra's Algorithm 3.11 from  $u$  to  $v$  in the following graph.

