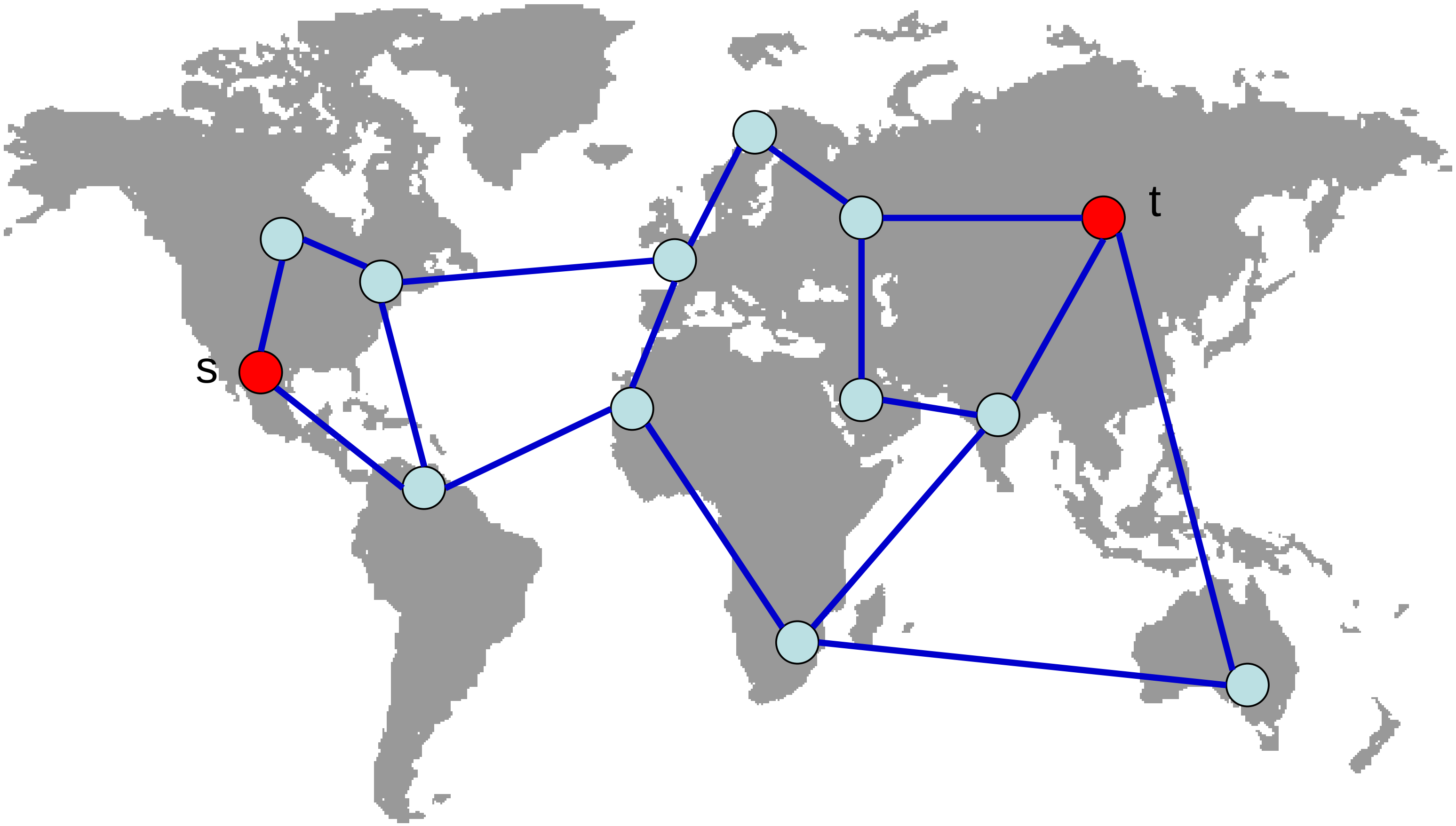
Fundamental Algorithms

Chapter 4: Shortest Paths

Christian Scheideler

WS 2017

Shortest Paths



Central question: Determine fastest way to get from s to t.

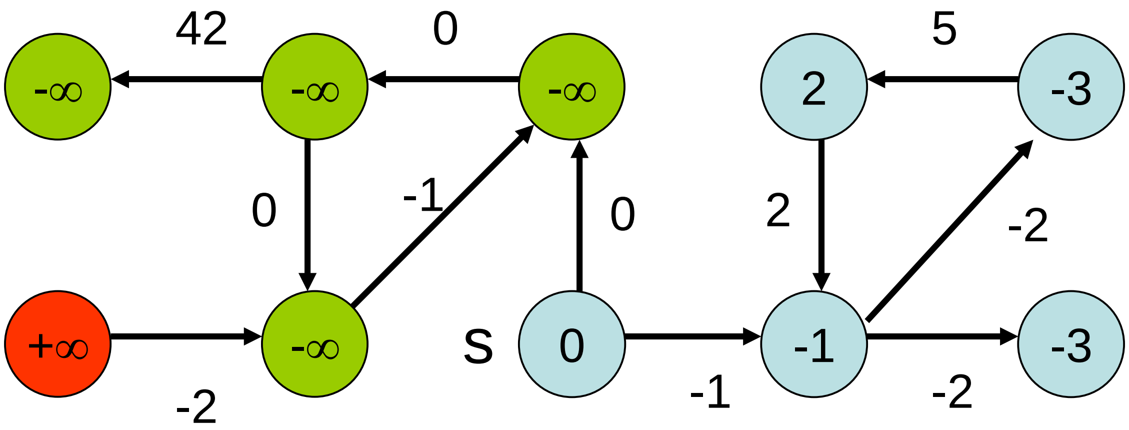
Shortest Paths

Shortest Path Problem:

- directed/undirected graph $G=(V,E)$
- edge costs $c:E\rightarrow\mathbb{R}$

- SSSP (single source shortest path): find shortest paths from a source node to all other nodes
- APSP (all pairs shortest path): find shortest paths between all pairs of nodes

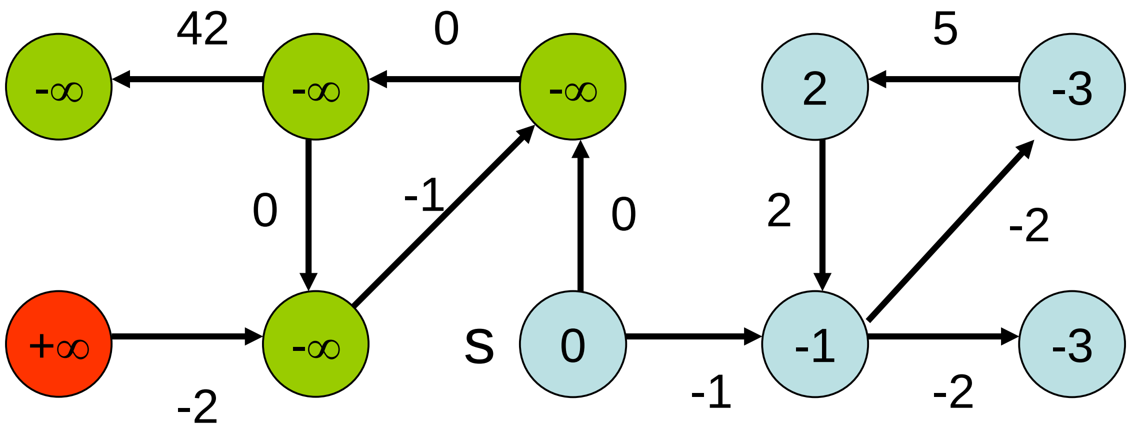
Shortest Paths



μ(s,v): distance between s and v

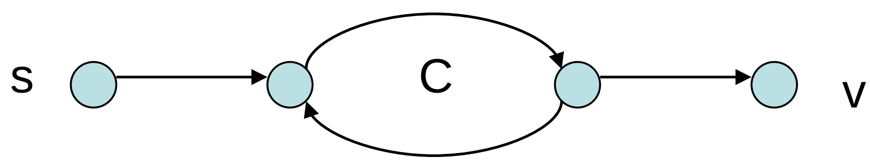
$$\mu(s,v) = \begin{cases} \infty & \text{no path from s to v} \\ -\infty & \text{path of arbitrarily low cost from s to v} \\ \min\{ c(p) \mid p \text{ is a path from s to v}\} & \end{cases}$$

Shortest Paths



When is the distance -∞?

If there is a negative cycle:

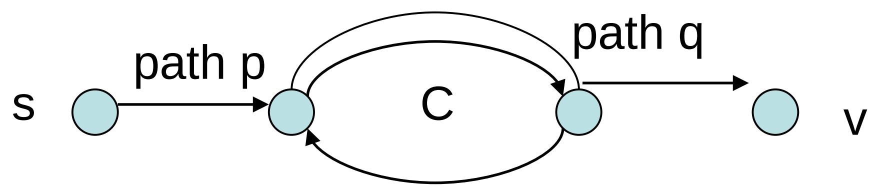


c(C)<0

Shortest Paths

Negative cycle necessary and sufficient for a distance of $-\infty$.

Negative cycle sufficient:



$c(C)<0$

Cost for i-fold traversal of C:

$c(p) + i \cdot c(C) + c(q)$

For $i \to \infty$ this expression approaches $-\infty$.

Shortest Paths

Negative cycle necessary and sufficient for a distance of $-\infty$.

Negative cycle necessary:

- l: minimal cost of a simple path from s to v
- suppose there is a non-simple path p from s to v with cost $c(r)<l$
- p non-simple: continuously remove a cycle C till we are left with a simple path
- since $c(p) < l$, there must be a cycle C with $c(C)<0$

Shortest Paths in Arbitrary Graphs

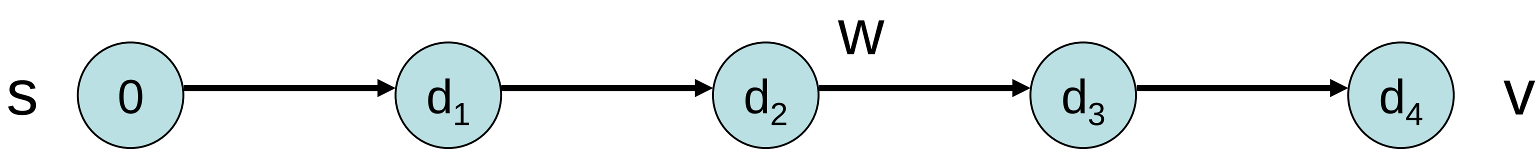
General Strategy:

- Initially, set $d(s):=0$ and $d(v):=\infty$ for all other nodes
- Visit nodes in an order that ensures that at least one shortest path from s to every v is visited in the order of its nodes
- For every visited v, update distances to nodes w with $(v,w)\in E$, i.e., $d(w):= \min\{d(w), d(v)+c(v,w)\}$

Bellman-Ford Algorithm

Consider graphs with arbitrary edge costs.

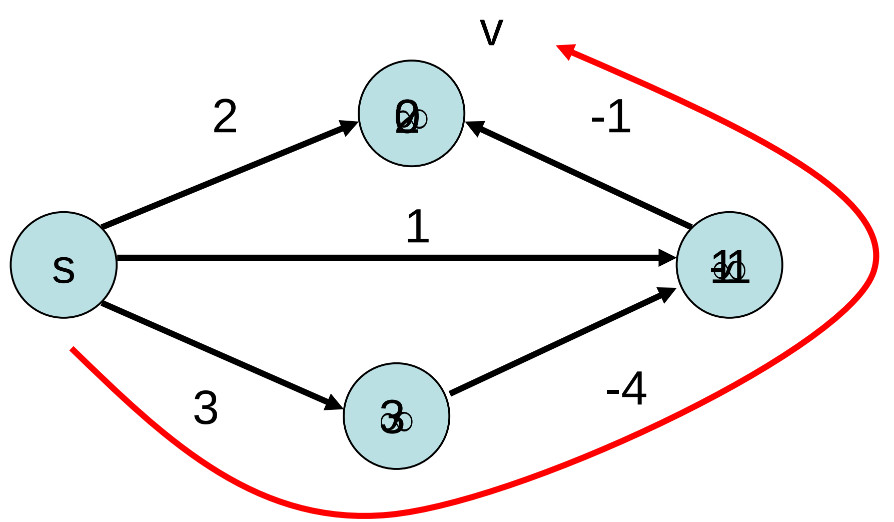
Problem: visit nodes along a shortest path from s to v in the right order



Dijkstra´s algorithm cannot be used in this case any more.

Bellman-Ford Algorithm

Example:



Node v has wrong distance value!

Bellman-Ford Algorithm

Lemma 4.1: For every node v with $\mu(s,v)>-\infty$ there is a simple path (without cycle!) from s to v of length $\mu(s,v)$.

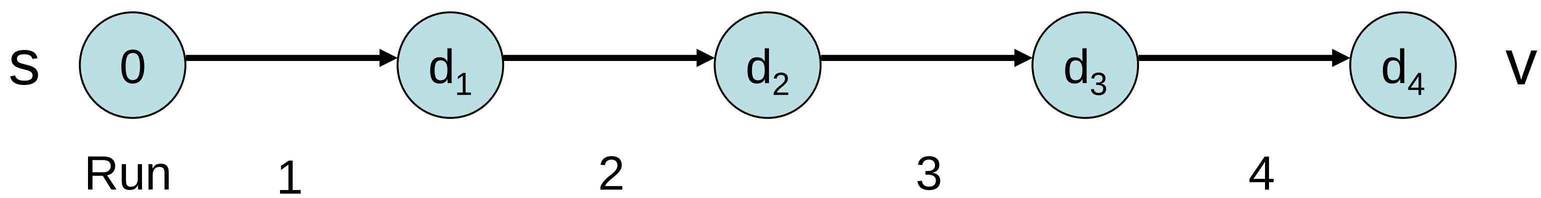
Proof:

- Path with cycle of length ≥ 0: removing the cycle does not increase the path length
- Path with cycle of length <0: distance from s is $-\infty$!

Bellman-Ford Algorithm

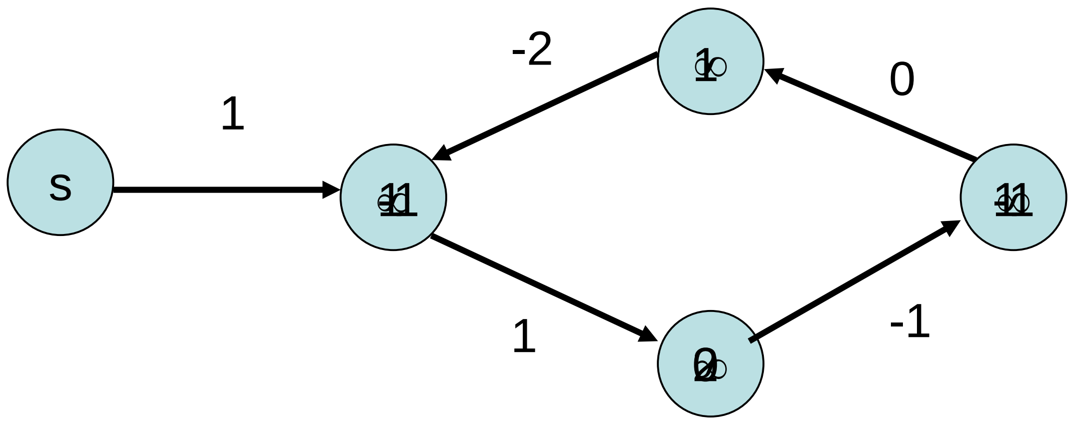
Conclusion: (graph with n nodes)
For every node v with $\mu(s,v) > -\infty$ there is a shortest path along <n nodes to v.

Strategy: visit (n-1)-times all nodes in the graph and update distances. Then all shortest paths have been considered.



Bellman-Ford Algorithm

Problem: detection of negative cycles



Conclusion: in a negative cycle, distance of at least one node keeps decreasing in each round, starting with a round <n

Bellman-Ford Algorithm

Lemma 4.2:

- No decrease of a distance in a round (i.e., $d[v]+c(v,w) \geq d[w]$ for all w):
  Done because $d[w]=\mu(s,w)$ for all w
- Decrease of a distance even in n-th round (i.e., $d[v]+c(v,w)<d[w]$ for some w):
  There are negative cycles for all of these nodes, so node w has distance $\mu(s,w)=-\infty$. If this is true for w, then also for all nodes reachable from w.

Proof: exercise

Bellman-Ford Algorithm

```
Procedure BellmanFord(s: NodeId)
  d=<∞,…,∞>: NodeArray of ℝ∪{-∞, ∞}
  parent=<⊥,…, ⊥>: NodeArray of NodeId
  d[s]:=0; parent[s]:=s
  for i:=1 to n-1 do  // update distances for n-1 rounds
    forall e=(v,w)∈E do
      if d[w] > d[v]+c(e) then  // better distance?
        d[w]:=d[v]+c(e); parent[w]:=v
  forall e=(v,w) ∈E do        // still better in n-th round?
    if d[w] > d[v]+c(e) then infect(w)

Procedure infect(v)  // set -∞-distance starting with v
  if d[v]>-∞ then
    d[v]:=-∞
    forall (v,w)∈E do infect(w)
```

Bellman-Ford Algorithm

Runtime: $O(n \cdot m)$

Improvements:

- Check in each update round if we still have $d[v]+c[v,w]<d[w]$ for some $(v,w) \in E$. No: done!
- Visit in each round only those nodes w with some edge $(v,w) \in E$ where $d[v]$ has decreased in the previous round.

All Pairs Shortest Paths

Assumption: graph with arbitrary edge costs, but no negative cycles

Naive Strategy for a graph with n nodes: run n times Bellman-Ford Algorithm (once for every node as the source)

Runtime: $O(n^2 m)$

All Pairs Shortest Paths

Better Strategy: Reduce n Bellman-Ford applications to n Dijkstra applications

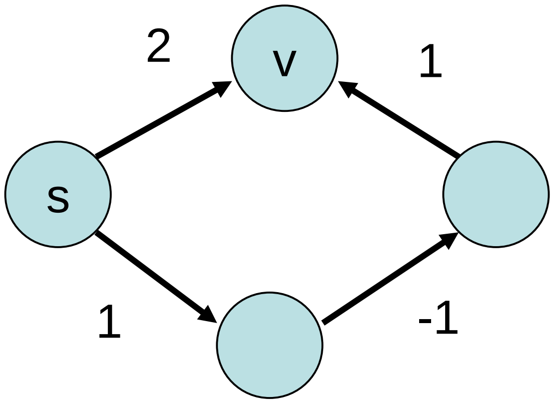
Problem: we need non-negative edge costs

Solution: convert edge costs into non-negative edge costs without changing the shortest paths (not so easy!)

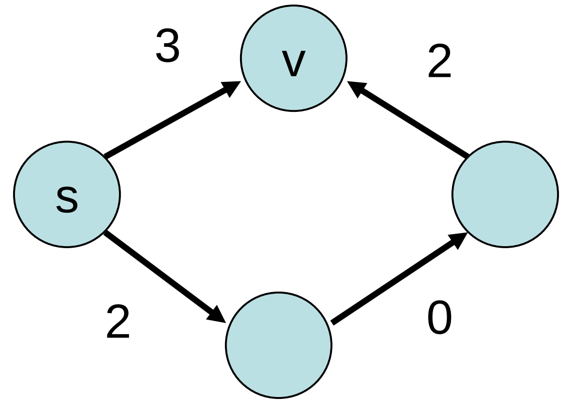
All Pairs Shortest Paths

Counterexample to additive increase by c:

before



cost +1 everywhere



: shortest path

Johnson´s Method

- Let $\phi: V \rightarrow \mathbb{R}$ be a function that assigns a potential to every node.
- The reduced cost of $e=(v,w)$ is:

$$r(e) := c(e) + \phi(v) - \phi(w)$$

Lemma 4.3: Let p and q be paths connecting the same endpoints in G. Then for every potential ϕ: $r(p)<r(q)$ if and only if $c(p)<c(q)$.

Johnson´s Method

Lemma 4.3: Let p and q be paths connecting the same endpoints in G. Then for every potential ϕ: $r(p)<r(q)$ if and only if $c(p)<c(q)$.

Proof: Let $p=(v_1,\dots,v_k)$ be an arbitrary path and $e_i=(v_i,v_{i+1})$ for all i. It holds:

$$
\begin{aligned}
r(p) &= \textstyle\sum_i r(e_i) \\
&= \textstyle\sum_i (\phi(v_i) + c(e_i) - \phi(v_{i+1})) \\
&= \phi(v_1) + c(p) - \phi(v_k)
\end{aligned}
$$

Johnson´s Method

Lemma 4.4: Suppose that G has no negative cycles and that all nodes can be reached from s. Let $\phi(v)=\mu(s,v)$ for all $v \in V$. With this ϕ, $r(e) \geq 0$ for all e.

Proof:

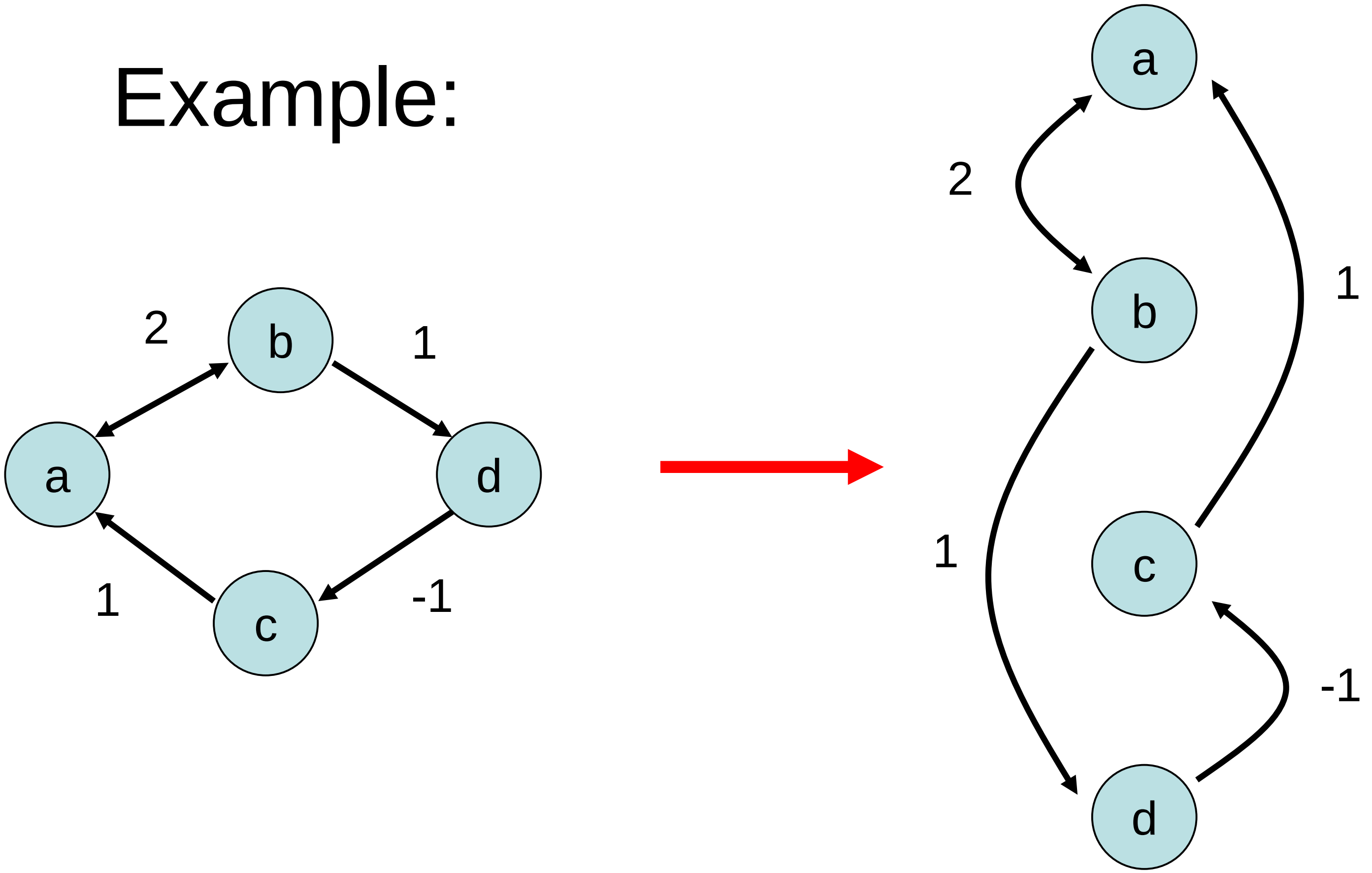
- According to our assumption, $\mu(s,v) \in \mathbb{R}$ for all v
- We know: for every edge $e=(v,w)$, $\mu(s,v)+c(e) \geq \mu(s,w)$ (otherwise, we have a contradiction to the definition of μ!)
- Therefore, $r(e) = \mu(s,v) + c(e) - \mu(s,w) \geq 0$

Johnson´s Method

1. Create new node s and new edges (s,v) for all v in G with $c(s,v)=0$ (all nodes reachable!)
2. Compute $\mu(s,v)$ using Bellman-Ford and set $\phi(v):=\mu(s,v)$ for all v
3. Compute the reduced costs $r(e)$
4. Compute for all nodes v the distances $\bar{\mu}(v,w)$ using Dijkstra with the reduced costs on graph G without node s
5. Compute the correct distances $\mu(v,w)$ via $\mu(v,w):=\bar{\mu}(v,w)+\phi(w)-\phi(v)$

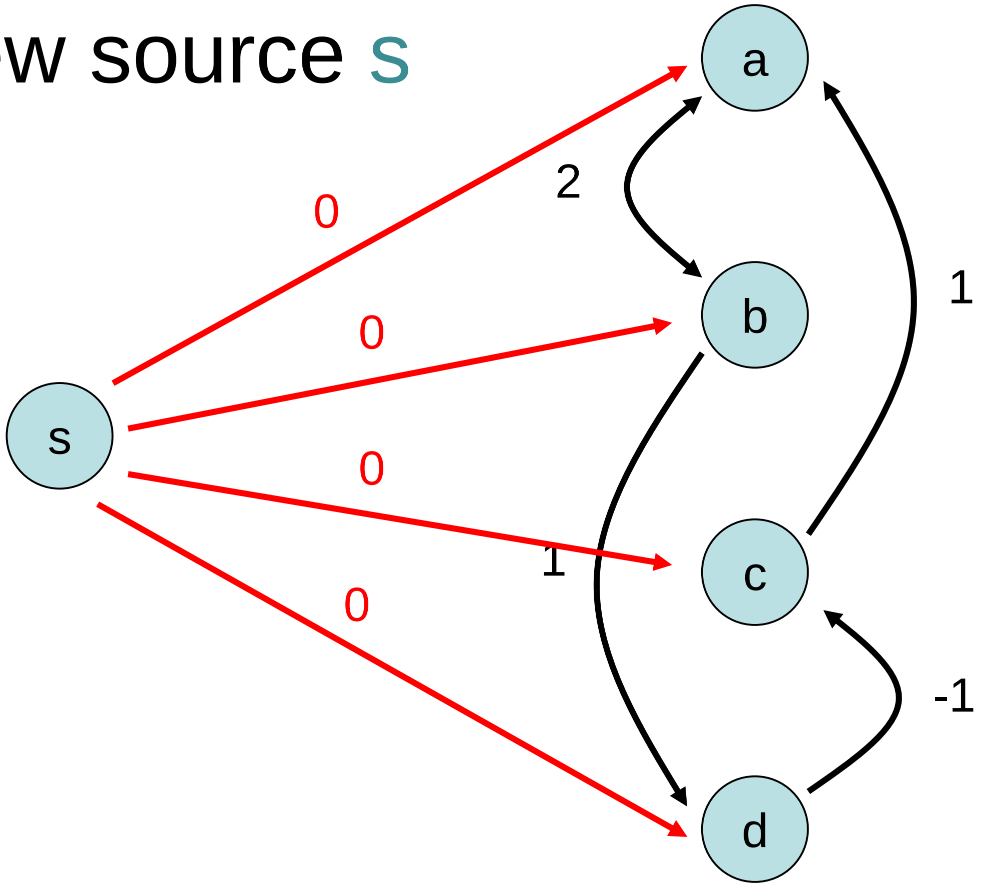
Johnson´s Method

Example:



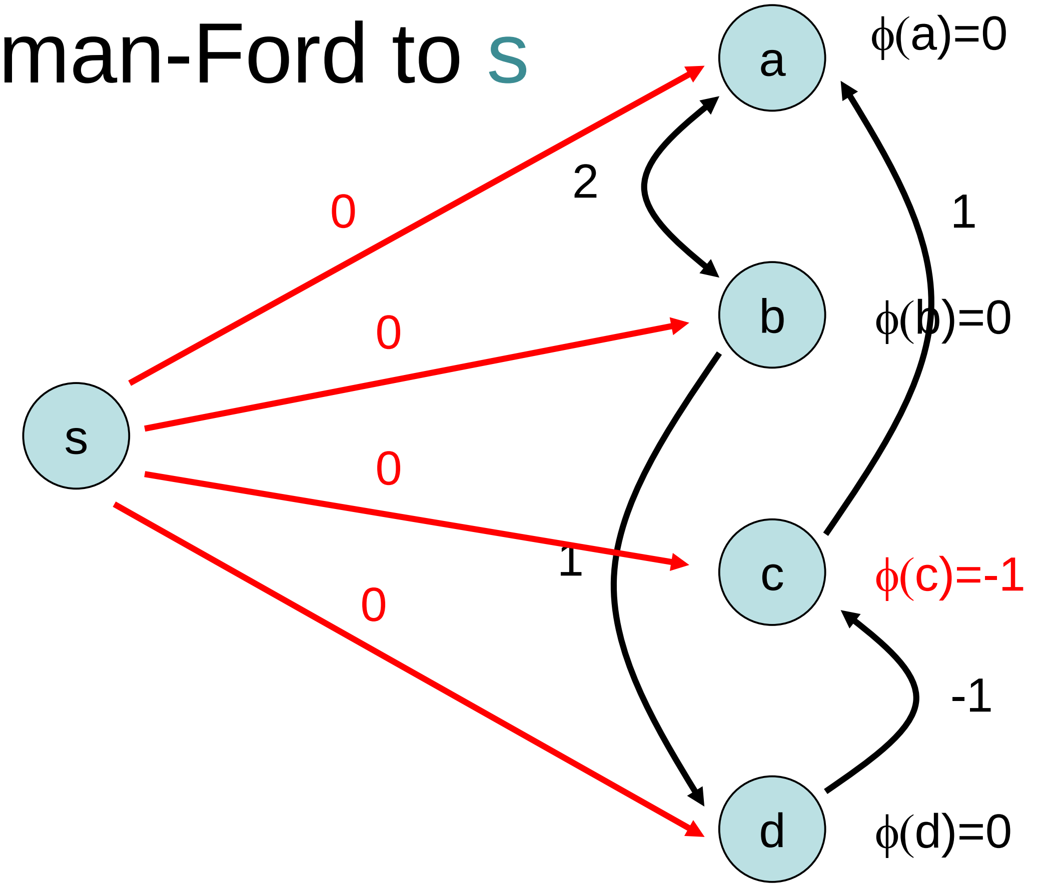
Johnson´s Method

Step 1: create new source s



Johnson´s Method

Step 2: apply Bellman-Ford to s

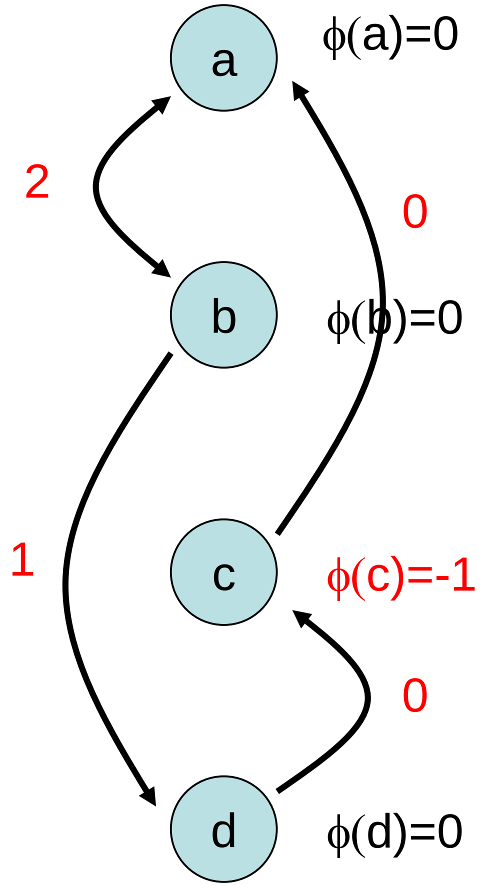


Johnson´s Method

Step 3: compute r(e)-values

The reduced cost of e=(v,w) is:

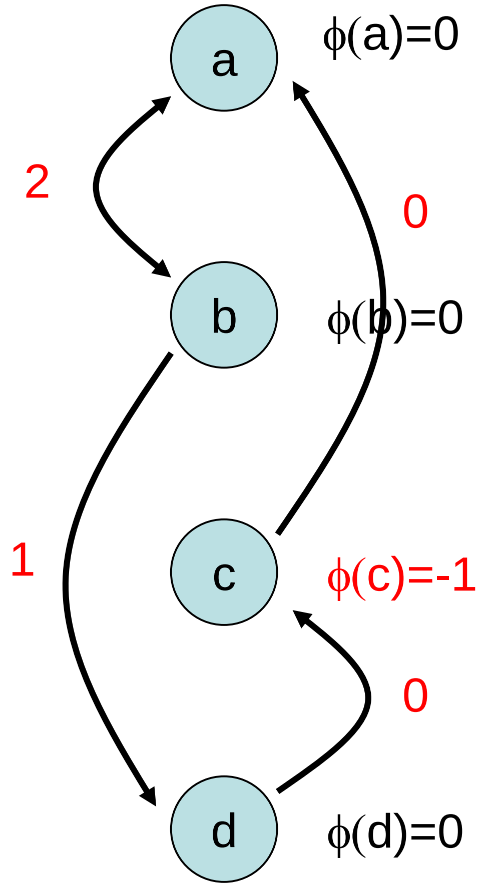
$$r(e) := \phi(v) + c(e) - \phi(w)$$



Johnson´s Method

Step 4: compute all distances $\bar{\mu}(v,w)$ via Dijkstra

$\bar{\mu}$	a	b	c	d
a	0	2	3	3
b	1	0	1	1
c	0	2	0	3
d	0	2	0	0

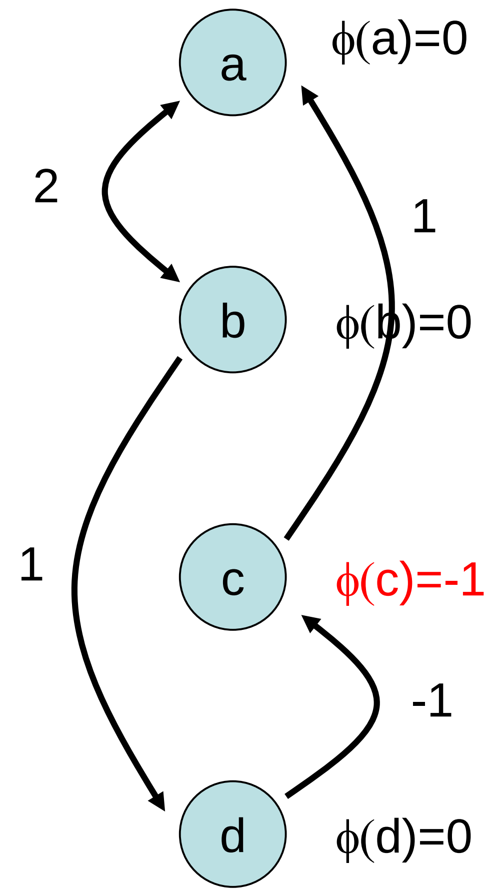


Johnson´s Method

Step 5: compute correct distances via the formula

$\mu(v,w)=\bar{\mu}(v,w)+\phi(w)-\phi(v)$

μ	a	b	c	d
a	0	2	2	3
b	1	0	0	1
c	1	3	0	4
d	0	2	-1	0



All Pairs Shortest Paths

Runtime of Johnson´s Method:

$O(T_{Bellman\text{-}Ford}(n,m) + n \cdot T_{Dijkstra}(n,m))$
$= O(n \cdot m + n(n \log n + m))$
$= O(n \cdot m + n^2 \log n)$

when using Fibonacci heaps.

- Problem with the runtime bound: m can be quite large in the worst case (up to $\sim n^2$)
- Can we significantly reduce m if we are fine with computing approximate shortest paths?

Graph Spanners

Definition 4.5: Given an undirected graph $G=(V,E)$ with edge costs $c:E\rightarrow\mathbb{R}$, a subgraph $H\subseteq G$ is an (α,β)-spanner of G iff for all $u,v\in V$,

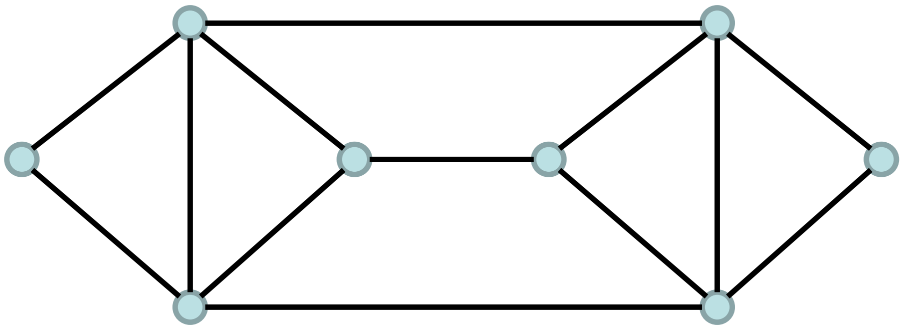
$$d_H(u,v) \leq \alpha\cdot d_G(u,v) + \beta$$

- $d_G(u,v)$: distance of u and v in G
- α: multiplicative stretch
- β: additive stretch

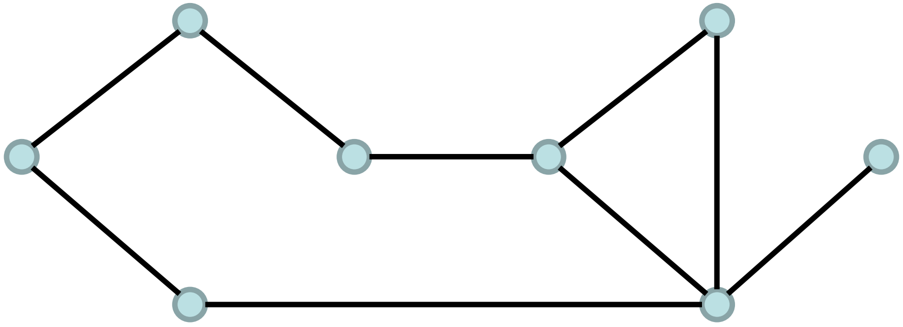
Graph Spanners

Example: all edge costs are 1

G



H



Graph Spanners

Consider the following Greedy algorithm by Althöfer et al. (Discrete Computational Geometry,1993):

$E(H) := \emptyset$
for each $\{u,v\} \in E(G)$ in the order of non-decreasing edge costs do
  if $(2k-1) \cdot c(u,v) < d_H(u,v)$ then
    add $\{u,v\}$ to $E(H)$

Theorem 4.6: For any $k \geq 1$, $|E(H)| = O(n^{1+1/k})$ and the graph H constructed by the Greedy algorithm is a $(2k-1,0)$-spanner.

Thorup and Zwick have shown that for any graph G with non-negative edge costs a structure related to H can be built in expected time $O(k \cdot m \cdot n^{1/k})$, which implies that we can then solve the $(2k-1)$-approximate APSP in time

$$O(k \cdot m \cdot n^{1/k} + n^{2+1/k}).$$

We will get back to that when we talk about distance oracles.

Graph Spanners

Proof of Theorem 4.6:

Lemma 4.7: H is a $(2k-1,0)$-spanner of G.

Proof:

- Consider any edge $\{u,v\} \in E(G) \setminus E(H)$.
- Since $\{u,v\}$ was rejected by the algorithm, $d_H(u,v) \leq (2k-1) \cdot c(u,v)$.
- Consider now any shortest path p from node a to b in G.
- For every edge $\{u,v\}$ in p, there is a path of length at most $(2k-1) \cdot c(u,v)$ in H.
- Replacing each edge $\{u,v\}$ in p by this path results in a path from a to b in H of length at most $(2k-1) \cdot c(p)$.

Graph Spanners

Proof of Theorem 4.6:

Lemma 4.8: Let C be any cycle in H. Then $|C|>2k$.

Proof:

- Assume that there is a cycle C of length at most $2k$ in H.
- Let $\{u,v\}$ be the last edge in C that was added by the algorithm.
- Clearly, $\{u,v\}$ has the largest cost of all edges in C.
- Also, when $\{u,v\}$ was considered, $(2k-1)\cdot c(u,v)<d_H(u,v)$ as otherwise $\{u,v\}$ would not have been added to H.
- However, since $C\setminus\{u,v\}$ results in a path of length at most $(2k-1)\cdot c(u,v)$ from u to v, $d_H(u,v)\leq(2k-1)\cdot c(u,v)$, leading to a contradiction.
- Hence, the lemma is true.

Graph Spanners

Proof of Theorem 4.6:

Lemma 4.8 implies that H has a girth (defined as the minimum cycle length in H) of more than 2k.

Lemma 4.9: Let H be a graph of size n with girth >2k. Then $|E(H)|=O(n^{1+1/k})$.

Proof:

- Let H be be any graph with girth >2k and at least $n+2n^{1+1/k}$ edges.
- Repeatedly remove any node from H of degree at most $\lceil n^{1/k} \rceil$ and any edges incident to that node, until no such node exists.
- The total number of edges removed in this way is at most $n \cdot (n^{1/k}+1)$.
- Hence, we obtain a subgraph H´ of H of minimum degree more than $\lceil n^{1/k} \rceil$ with at least $n^{1+1/k}$ edges connecting at most n nodes.
- Exercise: show that there cannot be a graph G of size n with girth >2k and minimum degree more than $\lceil n^{1/k} \rceil$.
- Thus, H´ must have a girth of at most 2k, and therefore also the original graph H. This, however, is a contradiction.

Graph Spanners

If we restrict ourselves to unweighted graphs (i.e., all edges have a cost of 1), we can also construct good additive spanners.

Theorem 4.10: Any n-node graph G has a $(1,2)$-spanner with $O(n^{3/2} \log n)$ edges.

Proof:

We first need the notion of hitting sets.

Definition 4.11: Given a collection M of subsets of V, a subset $S \subseteq V$ is a hitting set of M if it intersects every set in M.

Lemma 4.12: Let $M=(S_1,\ldots,S_n)$ be a collection of subsets of $V=\{1,\ldots,n\}$ with $|S_i| \geq R$ for all i. There is an algorithm running in $O(nR \log n + (n/R)\log^2 n)$ time that finds a hitting set S of M with $|S| \leq (n/R)\ln n$.

Graph Spanners

Lemma 4.12: Let $M=(S_1,\ldots,S_n)$ be a collection of subsets of $V=\{1,\ldots,n\}$ with $|S_i|\geq R$ for all i. There is an algorithm running in $O(nR \log n + (n/R) \log^2 n)$ time that finds a hitting set S of M with $|S|\leq(n/R)\ln n$.

Proof:

- Assume w.l.o.g. that $|S_i|=R$ for all i. Run the following greedy algorithm:

  $|S|:=\emptyset$
  for each $1\leq j\leq n$, keep a counter $c(j)=|\{S_i\in M: j\in S_i\}|$
  while $M\neq\emptyset$ do
      $k:=\text{argmax}_j\ c(j)$
      $S:=S\cup\{k\}$
      remove any subsets from M containing k and
      update the counters $c(j)$ accordingly

- To obtain the runtime, we store the counts $c(j)$ in a data structure that can support the following operations in $O(\log n)$ time: insert an element, return element j with maximum $c(j)$, decrement a given $c(j)$.

Graph Spanners

$|S|:=\emptyset$
for each $1 \le j \le n$, keep a counter $c(j)=|\{S_i \in M: j \in S_i\}|$
while $M \neq \emptyset$ do
  $k:=\text{argmax}_j\ c(j)$
  $S:=S \cup \{k\}$
  remove any subsets from M containing k and
  update the counters $c(j)$ accordingly

- total number of inserts: n because of n counters
  $\rightarrow$ runtime $O(n \log n)$
- total number of decrements: nR because each of the n sets contains just R elements and each of them can only cause one decrement
  $\rightarrow$ runtime $O(nR \log n)$
- total number of argmax calls: depends on number of iterations of while loop

Graph Spanners

Proof of Lemma 4.12 (continued):

- We still need an upper bound on $|S|$ (which gives an upper bound on while loop)
- Let m_j be the number of sets remaining in M after j passes of the while loop. Then $m_0=n$.
- Let k_j be the j-th element added to S, so $m_j=m_{j-1}-c(k_j)$.
- Just before we add k_j, the sum of $c(j)$ over all $j\in V\setminus\{k_1,\ldots,k_{j-1}\}$ must be $m_{j-1}R$, so $c(k_j)$ must be at least the average count, which is $m_{j-1}R/(n-j+1)$.
- Therefore,

  $m_j \leq (1-R/(n-j+1))\cdot m_{j-1} \leq n\ \Pi_{l=0}^{j-1}\ (1-R/(n-l))$

  $< n\cdot(1-R/n)^j \leq n\cdot e^{-Rj/n}$ (using the fact that $1-x\leq e^{-x}$ for all $x\in[0,1]$)
- Taking $j=(n/R)\ \ln n$ gives $m_j<1$, and therefore $m_j=0$.
- Hence, $|S|\leq(n/R)\ \ln n$.
- Thus, the total runtime over all argmax calls is $O((n/R)\log^2 n)$.

Graph Spanners

Proof of Theorem 4.10 (continued):

- Let S be a hitting set of minimal size for $M=\{ N(v) \mid deg(v) \geq \sqrt{n} \}$.
- From Lemma 4.12 ($R= \sqrt{n}$) we know that $|S|=O(\sqrt{n} \log n)$.
- Do a BFS search from each $s \in S$ and add the resulting n edges of the BFS tree to $E(H)$.
- For every $u \in V$ with $deg(u)<\sqrt{n}$ (the low-degree nodes), add all edges incident to u to $E(H)$.
- By construction, $|E(H)| = |S| \cdot n + n\sqrt{n} = O(n^{3/2} \log n)$.
- Consider any pair $u,v \in V$ with shortest path p in G. We have two cases:
- (a): p contains only low-degree nodes. Then p is also contained in H, so $d_H(u,v) = d(u,v)$.
- (b): p contains a high-degree node x. Let $s \in S$ be a node adjacent to x. Then we append the shortest paths from u to s and s to v in H to obtain a path from u to v in H. It holds:

$$
\begin{aligned}
d_H(u,v) &\leq d_H(u,s)+d_H(s,v) = d(u,s)+d(v,s) \\
&\leq (d(u,x)+1)+(d(v,x)+1) = d(u,v)+2
\end{aligned}
$$

- Hence, H is indeed a $(1,2)$-spanner.

Graph Spanners

Runtime of the algorithm for (1,2)-spanner:

- $O(n^{3/2} \log n)$: construction of hitting set S
- $O(\sqrt{n} \log n\ (n+m))$: BFS for all nodes in S
- $O(n^{3/2})$: adding all edges of low-degree nodes to H

Total runtime: $O(\sqrt{n}\ (m+n) \log n))$.

Runtime of approximate APSP algorithm for an unweighted graph G based on (1,2)-spanner H:

$$O(\sqrt{n}\ (m+n) \log n)) + O(n \cdot n^{3/2} \log n + n^2 \log n)$$

$$= O(\ n^{5/2} \log n)$$

With a more complex approach the runtime can be reduced to $O(\ n^{7/3} \log n)$. For the details see:

D. Dor, S. Halperin, and U. Zwick. All-pairs almost shortest paths. SIAM Journal of Computing, 29(5): 1740-1759, 2000.

Graph Spanners

Interestingly, the following two results are known:

Theorem 4.13: Any n-node graph G has a $(1,6)$-spanner with $O(n^{4/3})$ edges.

Theorem 4.14: In general, there is no additive spanner with $O(n^{4/3-\varepsilon})$ edges for n-node graphs for any $\varepsilon>0$.

For more information on that see:

Amir Abboud and Greg Bodwin. The 4/3 additive spanner exponent is tight. Proc. of the 48th ACM Symposium on Theory of Computing (STOC), 2016.

Distance Oracles

How to quickly answer distance requests?

Naive approach:

- Run an APSP algorithm and store all answers in a matrix

Problems:

- High runtime ($O(nm + n^2 \log n)$)
- High storage space ($\Theta(n^2)$)

Alternative approach, if approximate answers are sufficient:

- Compute additive or multiplicative spanner, and run an APSP algorithm on that spaner.
  $\rightarrow$ lower runtime
- But storage space is still high

Better solutions concerning the storage space have been investigated under the concept of distance oracles.

Distance Oracles

Definition 4.15: An α-approximate distance oracle is defined by two algorithms:

- a preprocessing algorithm that takes as its input a graph $G=(V,E)$ and returns a summary of G, and
- a query algorithm based on the summary of G that takes as its input two vertices $u,v \in V$ and returns an estimate $D(u,v)$ such that $d(u,v) \leq D(u,v) \leq \alpha \cdot d(u,v)$.

The quality of an α-approximate distance oracle is defined by its query time $q(n)$, preprocessing time $p(m,n)$, and storage space $s(n)$. The goal is to minimize all of these quantities.

Thorup and Zwick (STOC 2001) have shown the following result for graphs of non-negative edge costs:

Theorem 4.16: For all $k \geq 1$ there exists a $(2k-1)$-approximate distance oracle using $O(k \cdot n^{1+1/k})$ space and $O(m \cdot n^{1/k})$ time for preprocessing that can answer queries in $O(k)$ time (where we hide logarithmic factors in the O-notation).

Distance Oracles

Theorem 4.16: For all $k \geq 1$ there exists a $(2k-1)$-approximate distance oracle using $O(k \cdot n^{1+1/k})$ space and $O(m \cdot n^{1/k})$ time for preprocessing that can answer queries in $O(k)$ time (where we hide logarithmic factors in the O-notation).

Proof:

$k=1$: trivial. Just run our APSP algorithm.

$k=2$: Consider the following preprocessing algorithm:

> $A :=$ random subset $S \subseteq V$ of size $O(\sqrt{n} \log n)$
> for each $a \in A$ run Dijkstra to compute $d(a,v)$ for all $v \in V$
> for each $v \in V \setminus A$ do
> $p_A(v) := \text{argmin}_{y \in A}\, d(v,y)$
>   run Dijkstra to compute $A(v) := \{x \in V \mid d(v,x) < d(v,p_A(v))\}$
> $B(v) := A \cup A(v)$
>   store $p_A(v)$ under v
>   for all $x \in B(v)$, store $d(v,x)$ under key (v,x) in a hash table

- It is easy to show that with high probability A is a hitting set of $M = \{ N_{\sqrt{n}}(v) \mid v \in V\}$, where $N_{\sqrt{n}}(v)$ is the set containing the closest $\sqrt{n}$ nodes to v.

Distance Oracles

Lemma 4.17: $|B(v)| \leq O(\sqrt{n} \log n)$.

Proof:

- It suffices to show that $|A(v)| \leq \sqrt{n}$ since by construction $|A|=O(\sqrt{n} \log n)$.
- Because A hits the closest $\sqrt{n}$ nodes to v with high probability, some node $a \in A$ is also in $N_{\sqrt{n}}(v)$.
- Thus, all nodes closer to v than a are also in $N_{\sqrt{n}}(v)$.
- By the definition of $A(v)$, this implies that $A(v) \subseteq N_{\sqrt{n}}(v)$, and therefore, $|A(v)| \leq \sqrt{n}$.

Distance Oracles

Proof of Theorem 4.16 (continued):

Now we can bound the quality of the distance oracle.

- Lemma 4.17 implies that the storage space needed by our distance oracle is $O(n\cdot\sqrt{n}\log n)$.
- A query (u,v) is processed as follows:

if $d(u,v)$ is stored in the hash table then
    return $d(u,v)$
else
    return $d(u,p_A(u))+d(v,p_A(u))$

- This obviously takes constant time. We also need to show that $d(u,v)\leq D(u,v)\leq 3d(u,v)$.

Distance Oracles

Claim: $d(u,v) \leq D(u,v) \leq 3d(u,v)$.

- Suppose that $v \notin B(u)$, as otherwise $D(u,v)=d(u,v)$.
- By the triangle inequality we have

$$d(u,v) \leq d(u,p_A(u))+d(v,p_A(u)) = D(u,v)$$

- In order to show that $D(u,v) \leq 3d(u,v)$, we use the triangle inequality again to obtain

$$\begin{aligned} D(u,v) &= d(u,p_A(u))+d(p_A(u),v) \\ &\leq d(u,p_A(u))+(d(p_A(u),u)+d(u,v)) \\ &= 2d(u,p_A(u)) + d(u,v) \\ &\leq 3d(u,v) \end{aligned}$$

Distance Oracles

Proof of Theorem 4.16 (continued):

It remains to determine the runtime for the preprocessing.

- For each $a \in A$ run Dijkstra to compute $d(a,v)$ for all $v \in V$: runtime $O((m+n \log n) \cdot \sqrt{n} \log n)$.
- Determine $p_A(v)$ for each $v \in V \setminus A$: total runtime $O(n \cdot \sqrt{n} \log n)$
- Run Dijkstra to compute $A(v) := \{x \in V \mid d(v,x) < d(v,p_A(v))\}$: We need to modify Dijkstra so that we start with a Fibonacci heap only containing v and we only include a node y into the heap (and therefore process it) if $d(x)+c(x,y)<d(v,p_A(v))$ for some processed x. Then the runtime for each $v \in V \setminus A$ is $O(|E(A(v))|+|A(v)| \log n)$, where $E(A(v))$ is the set of all edges containing vertices of $A(v)$.
- Let $C(v)=\{ w \in V \mid v \in A(w) \}$. One can show (similarly to $A(v)$) that the expected size of $C(v)$ ist at most $2\sqrt{n}$. Hence,

$$\Sigma_{w \in V} |E(A(w))| \leq \Sigma_{w \in V,\ v \in A(w)} |E(v)| = \Sigma_{v \in V,\ w \in C(v)} |E(v)|$$
$$= \Sigma_{v \in V} (|C(v)| \cdot |E(v)|) = O(m \cdot \sqrt{n})$$

- Thus, the overall runtime for the $A(v)$´s is $O(m \cdot \sqrt{n} + n\sqrt{n} \log n)$.
- Therefore, the preprocessing needs $O((m + n \log n) \cdot \sqrt{n} \log n)$ time.

Distance Oracles

Proof of Theorem 4.16 (continued):

The algorithm for general k proceeds by taking many related samples $A_0,\ldots,A_k$ instead of just a single sample A. Concretely, it does the following:

- Let $A_0=V$ and $A_k=\emptyset$. For each $1\le i\le k-1$, choose a random $A_i\subseteq A_{i-1}$ of size $(|A_{i-1}|/n^{1/k})\log n = O(n^{1-i/k}\log n)$.
- Let $p_i(v)$ be the closest node to v in A_i. If $d(v,p_i(v))=d(v,p_{i+1}(v))$ then let $p_i(v)=p_{i+1}(v)$.
- For all $v\in V$ and $i<k-1$, define

  $A_i(v) = \{ x\in A_i \mid d(v,x) < d(v,p_{i+1}(v)) \}$

  $B(v) = A_{k-1}\cup(\bigcup_{i=0}^{k-2} A_i(v))$
- For all $v\in V$ and all $x\in B(v)$, store $d(v,x)$ in a hash table. Also store for each $v\in V$ and each $i\le k-1$, $p_i(v)$.

A query for (u,v) then works as follows:

$w:=p_0=v$
for $i=1$ to k do
  if $w\in B(u)$ then return $d(u,w)+d(w,v)$
  else $w:=p_i(u)$; swap u and v

For more information see:

Mikkel Thorup and Uri Zwick. Approximate distance oracles. In Proc. of the 33rd ACM Symposium on Theory of Computing (STOC), 2001.

Next Chapter

Matching algorithms…