Fundamental Algorithms

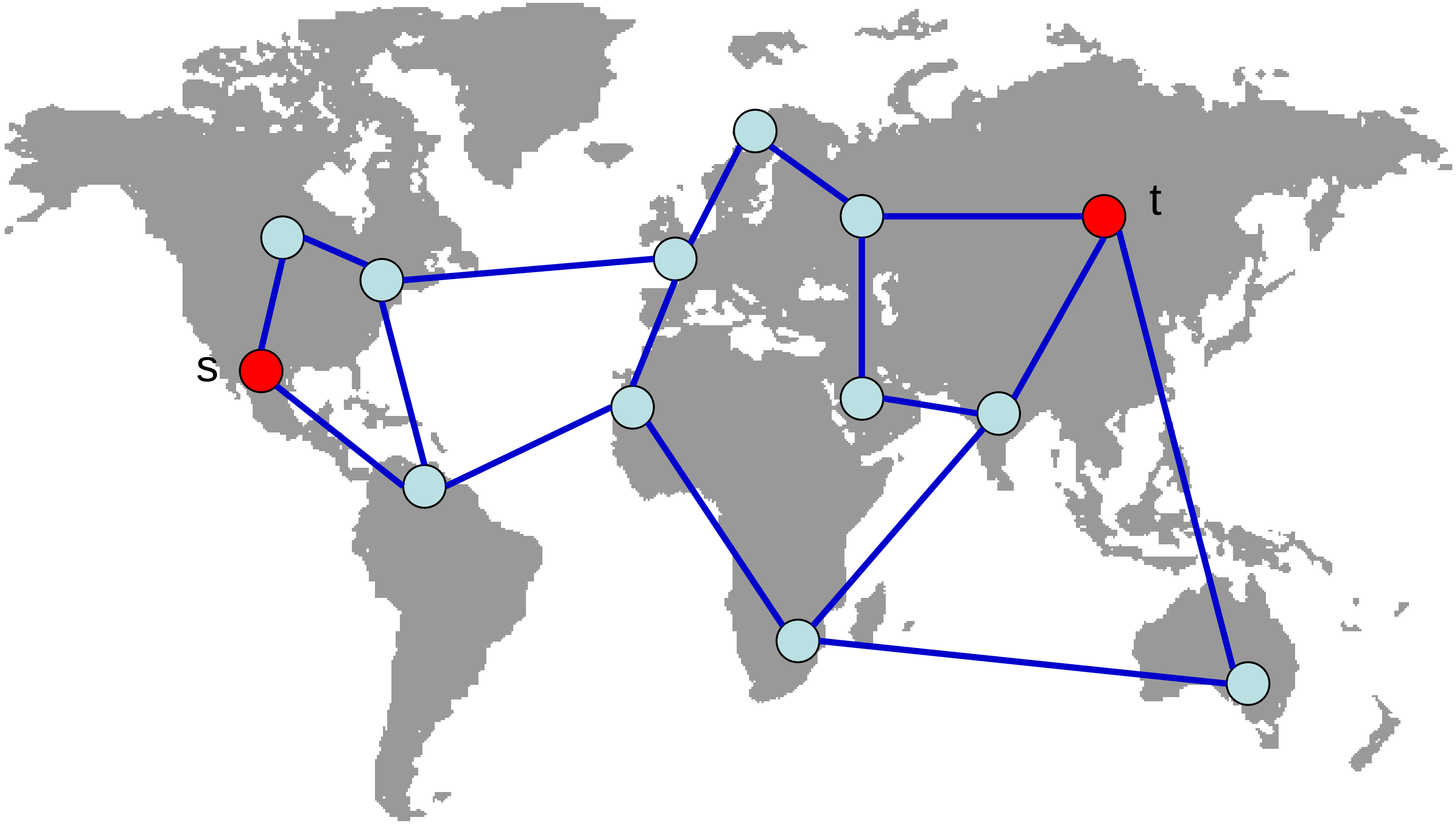
Chapter 4: Shortest Paths

Sevag Gharibian

(based on slides of Christian Scheideler)

WS 2018

Shortest Paths



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

Shortest Paths

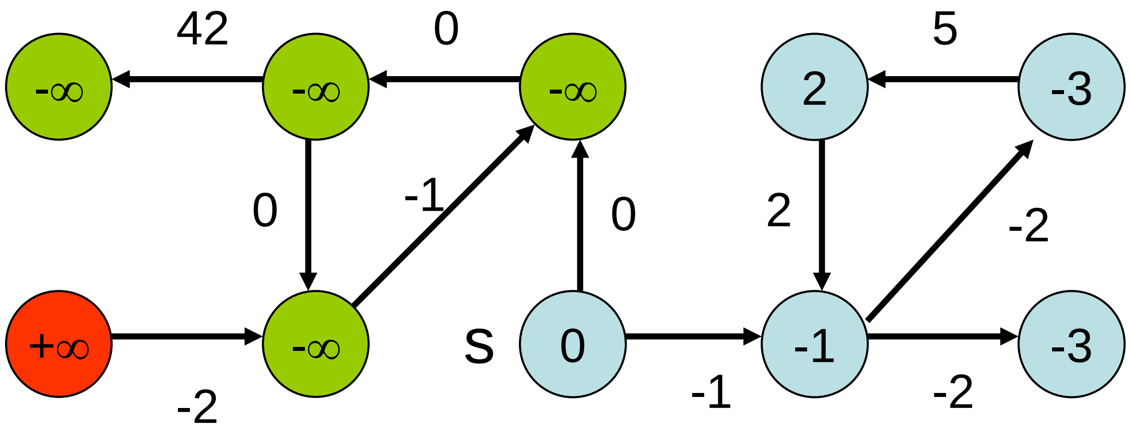
Shortest Path Problem:

- 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
  - In Lecture 2, we solved this using Dijkstra's algorithm and Priority Queues.
  - Q: What is different about the setting here?

- APSP (all pairs shortest path):
  find shortest paths between all pairs of nodes

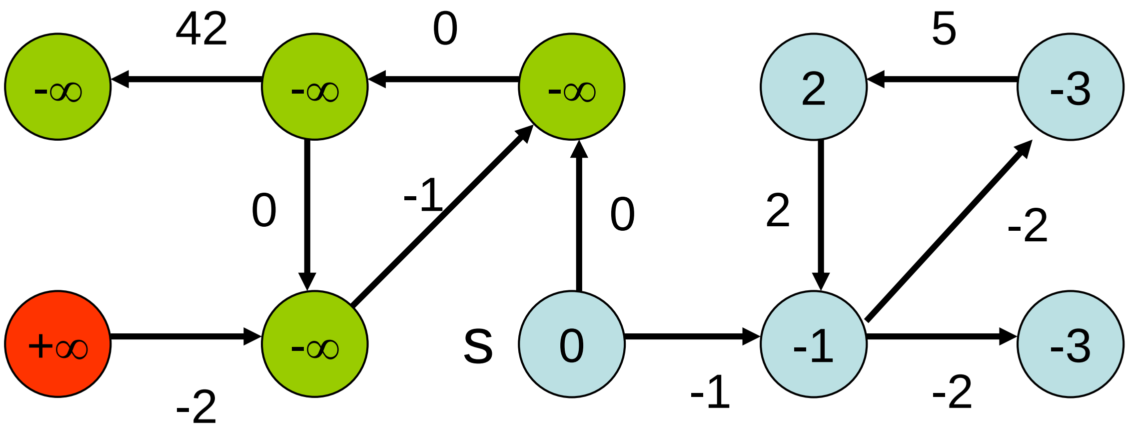
Shortest Paths



μ(s,v): distance between s and v

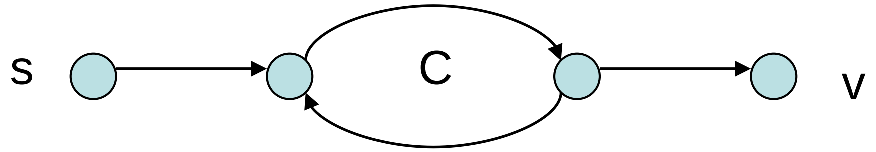
$$\mu(s,v) = \begin{cases} \infty & \text{no path from s to v} \\ -\infty & \text{path of arbitrarily low cost from s to v} \\ \min\{ c(p) \mid p \text{ is a path from s to v}\} & \end{cases}$$

Shortest Paths



When is the distance -∞?

If there is a negative cycle:

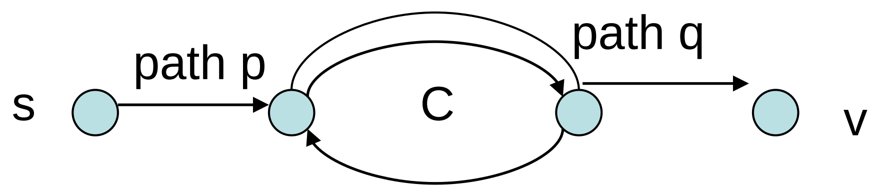


c(C)<0

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

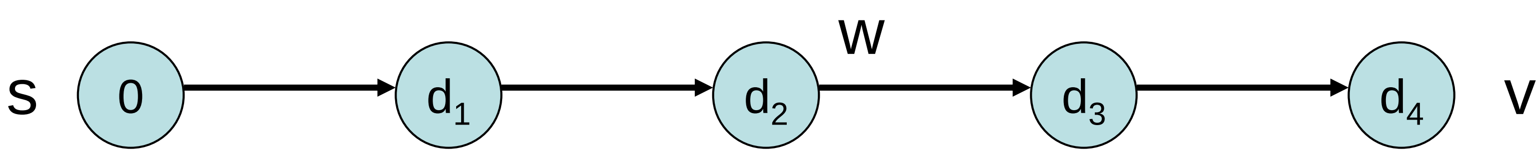
General Strategy:

- Initially, set $d(s):=0$ and $d(v):=\infty$ for all other nodes
- For every visited v, update distances to nodes w with $(v,w)\in E$, i.e., $d(w):= \min\{d(w), d(v)+c(v,w)\}$
- But what order to visit the edges E in?

Bellman-Ford Algorithm

Consider graphs with arbitrary (real) edge costs.

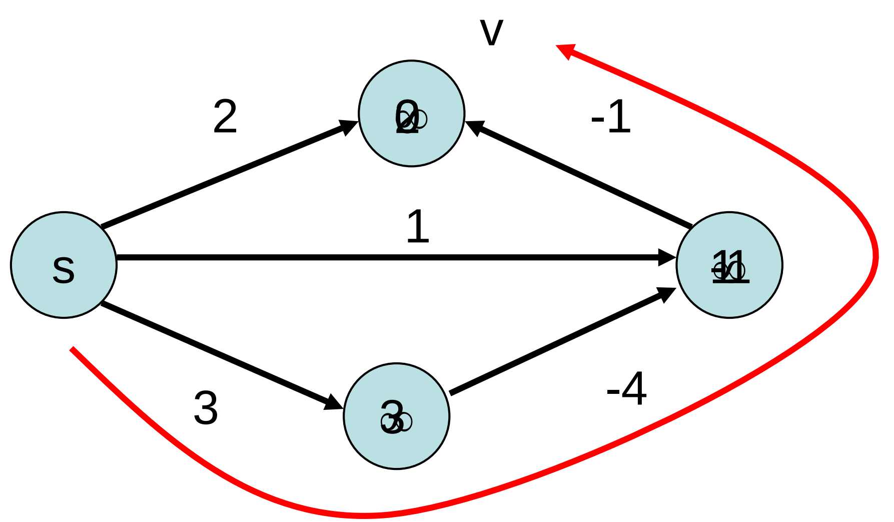
Problem: visit nodes along a shortest path from s to v in the „right order“



Dijkstra´s algorithm cannot be used in this case any more – let‘s do an example to see why.

Bellman-Ford Algorithm

Example (Dijkstra with negative weights):



Node v has wrong distance value! (Why?)

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.

Why is this important?

- It gives us a *stopping* criterion.
- Namely, takes at most n-1 steps to compute $\mu(s,v)$.

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: repeat the following n-1 times –

for each edge e in E, traverse e and update relevant node costs.

Claim: This will consider all simple paths of length n-1.

Bellman-Ford Algorithm

Strategy: repeat the following n-1 times –

for each edge e in E, traverse e and update relevant node costs.

Claim: This will consider all simple paths of length n-1. Why?

Proof sketch: For any simple path p, let e_i be its ith edge. Then, we view round i of the iteration as traversing edge e_i.

Bellman-Ford Algorithm

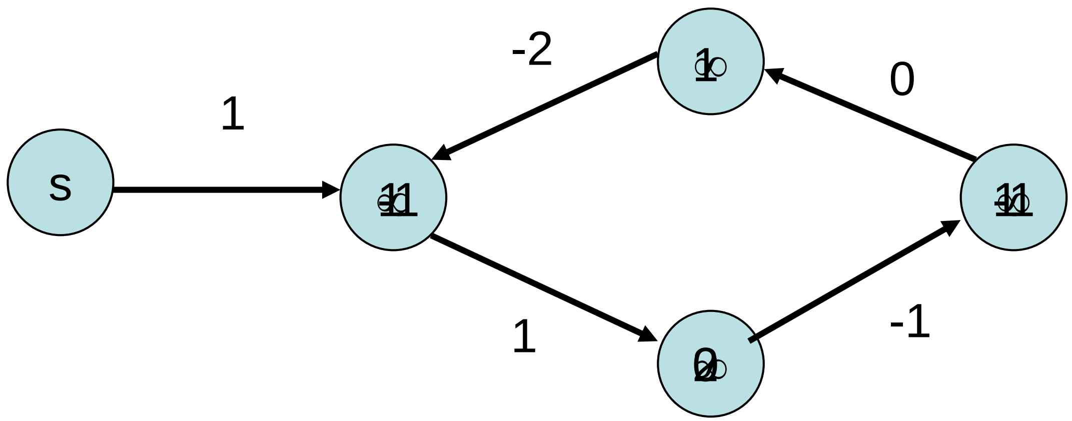
Proof sketch: For any simple path p, e_i be its ith edge. Then, we view round i of the loop as traversing edge e_i.

– Another viewpoint: Each iteration „applies all edges in parallel".

Aside: This idea is used in other places, too – e.g., if A is the adjacency matrix of a graph, then entry (i,j) of A^n contains number of walks of length n from vertex i to vertex j.

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:

- If no decrease of a distance in a round (i.e., $d[v]+c(v,w) \geq d[w]$ for all w), then $d[w]=\mu(s,w)$ for all w (i.e. reached correct values)
- If some node w's distance decreases in n-th round (i.e., $d[v]+c(v,w)<d[w]$ for some w): There are negative cycles for all such w, 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

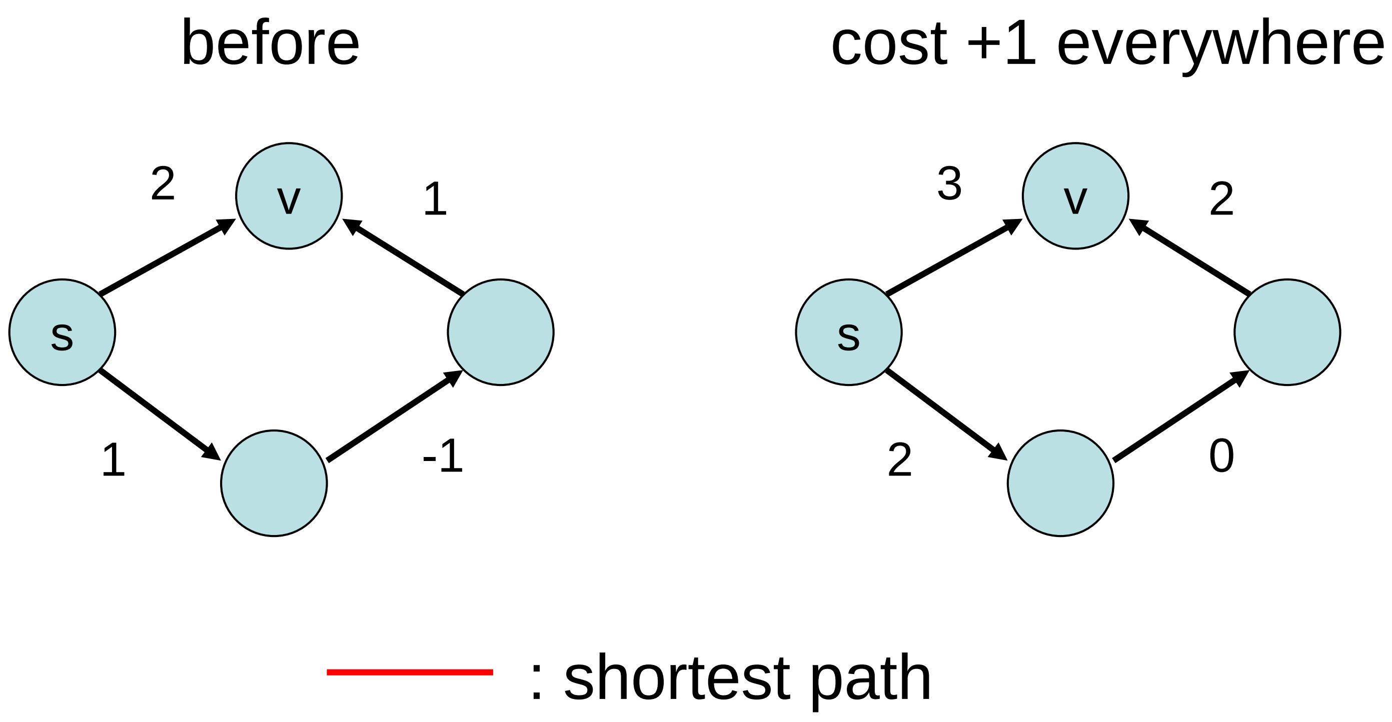
(Recall: Dijkstra requires O((m+n)logn) time using binary heaps.)

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:



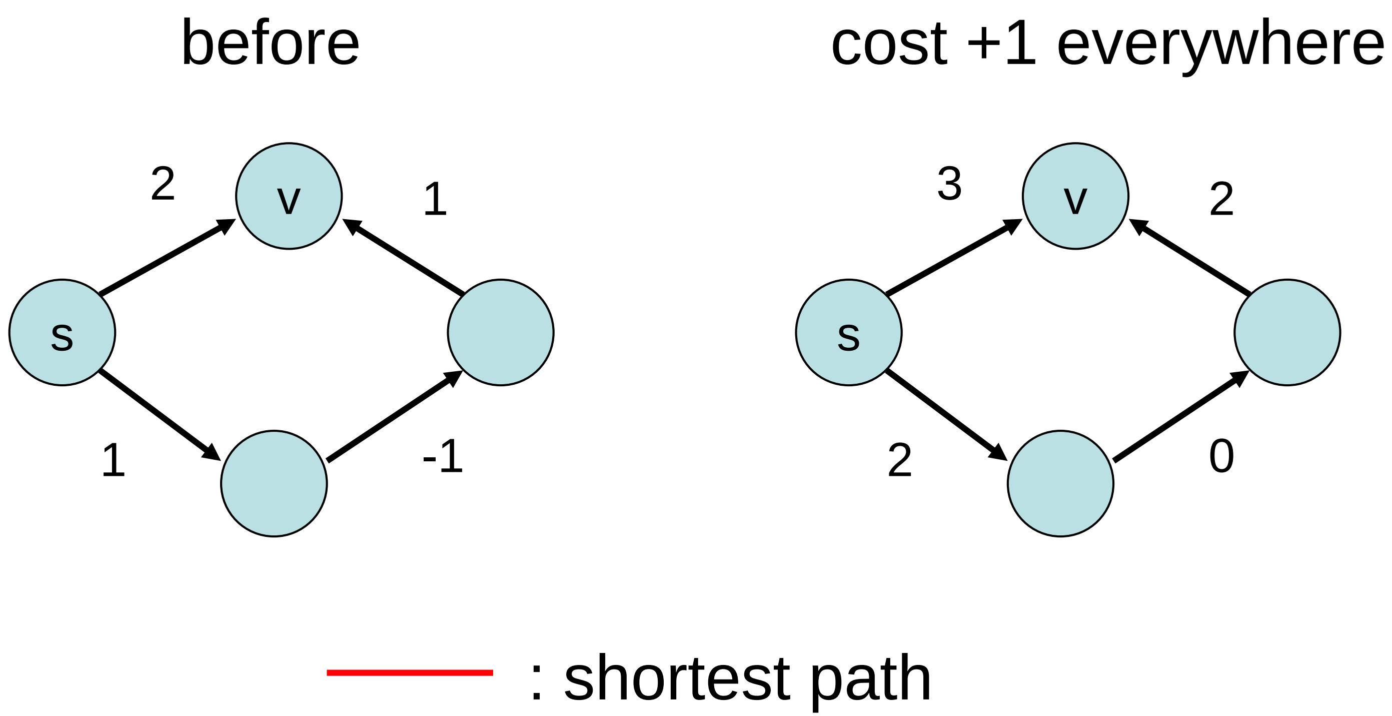
All Pairs Shortest Paths

Why does this counterexample work?

- Let p be a path in the original graph with cost c(p).
- After adding +1 to each edge, its new cost is c‘(p)=c(p)+|p|, for |p| the length of p.
- Thus, each *longer* paths are penalized disproportionately!
- Idea: New edge weights must not ``accumulate“ when added over any path.

All Pairs Shortest Paths

Counterexample to additive increase by c:



Johnson´s Method

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

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

- Claim: By choosing appropriate ϕ, the reduced cost can be used as non-negative edge labels.

For this, need two lemmas:

1. Prove that for any paths p and q mapping vertex v to w, and for any ϕ, $r(p)<r(q)$ iff $c(p)<c(q)$.
2. Give explicit ϕ such that all $r(e)>=0$.

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:

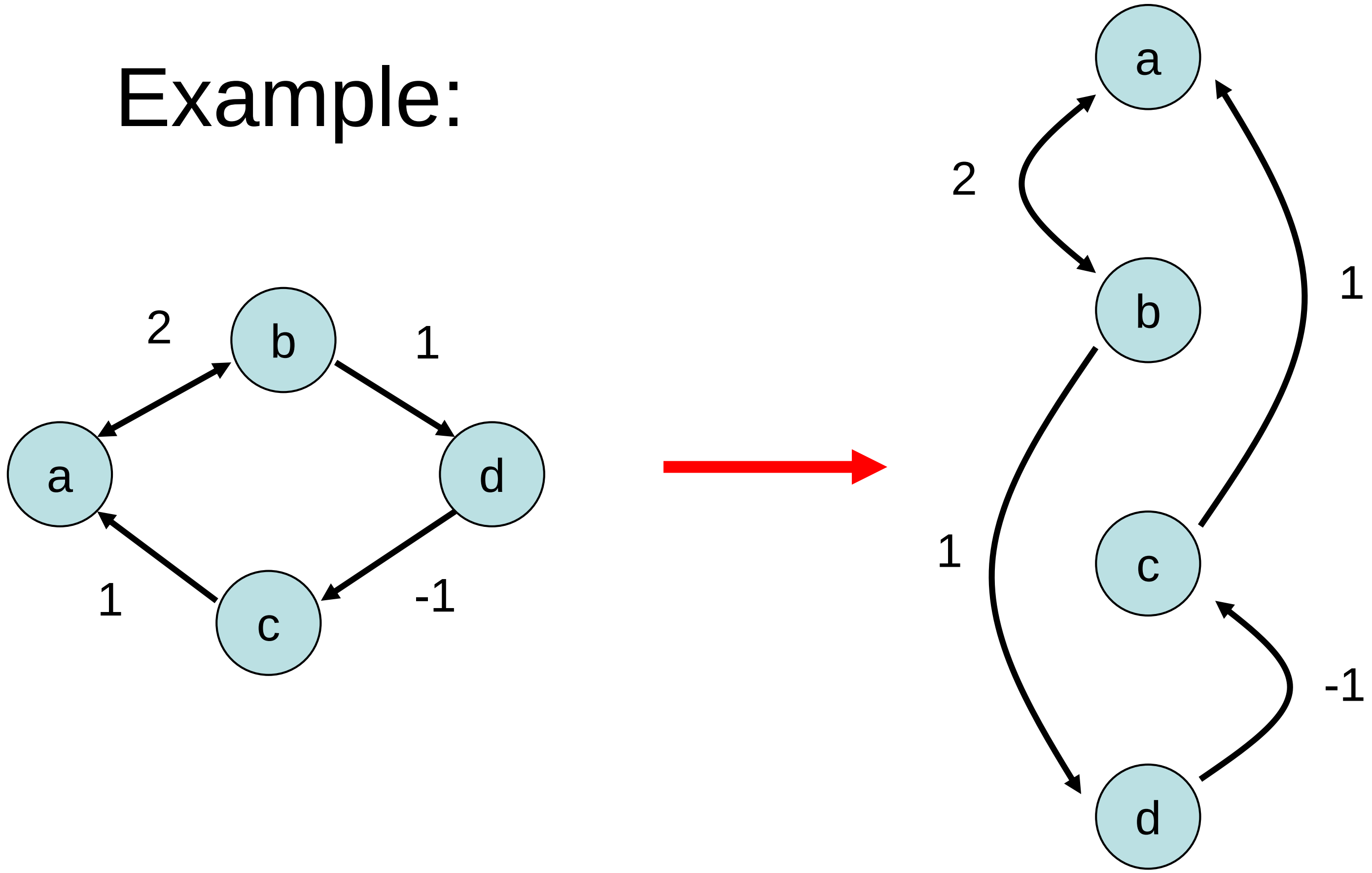
- According to our assumption, $\mu(s,v)\in\mathbb{R}$ for all v
- We know: for every edge $e=(v,w)$, $\mu(s,v)+c(e)\geq\mu(s,w)$ (otherwise, we have a contradiction to the definition of μ!)
- Therefore, $r(e) = \mu(s,v) + c(e) - \mu(s,w) \geq 0$

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!)
   - Fulfills assumption of Lemma 4.4 that such s exists
   - Cannot change cost of shortest path between pair of orig vertices v and w. (Why?)
2. Compute $\mu(s,v)$ using Bellman-Ford and set $\phi(v):=\mu(s,v)$ for all v
   - Needed to compute reduced costs r(e).
3. Compute the reduced costs r(e); make these the new edge costs.
   - Ensures edges have non-negative weight
4. Compute for all v distances $\mu(v,w)$ using Dijkstra with on graph G without node s
   - Finds least cost paths, but these costs need to be adjusted
5. Update path costs by setting $\mu(v,w):=\mu(v,w)+\phi(w)-\phi(v)$.

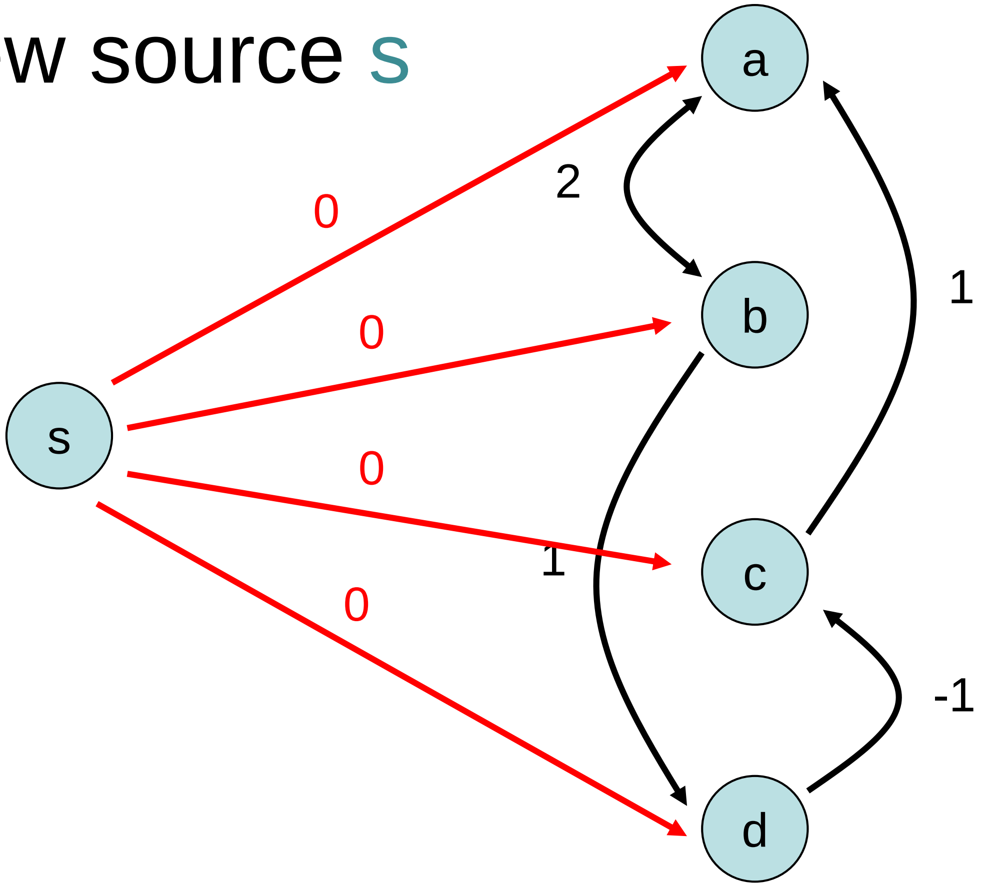
Johnson´s Method

Example:



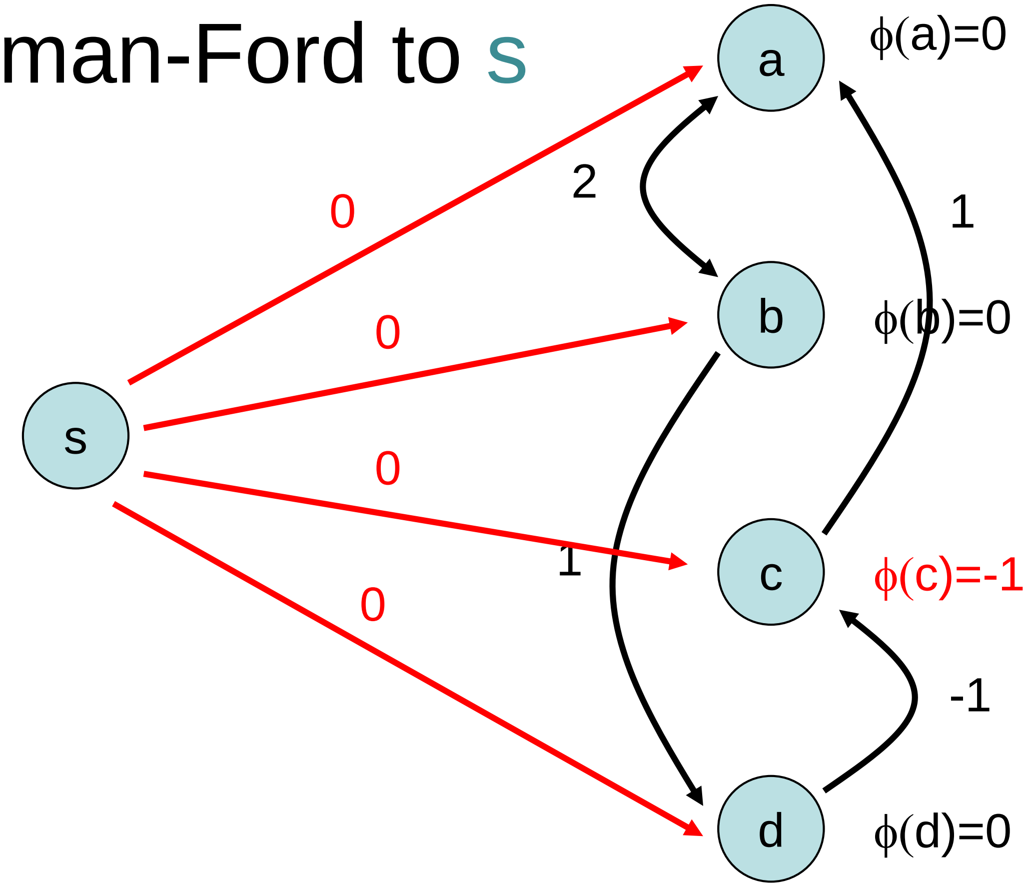
Johnson´s Method

Step 1: create new source s



Johnson´s Method

Step 2: apply Bellman-Ford to s

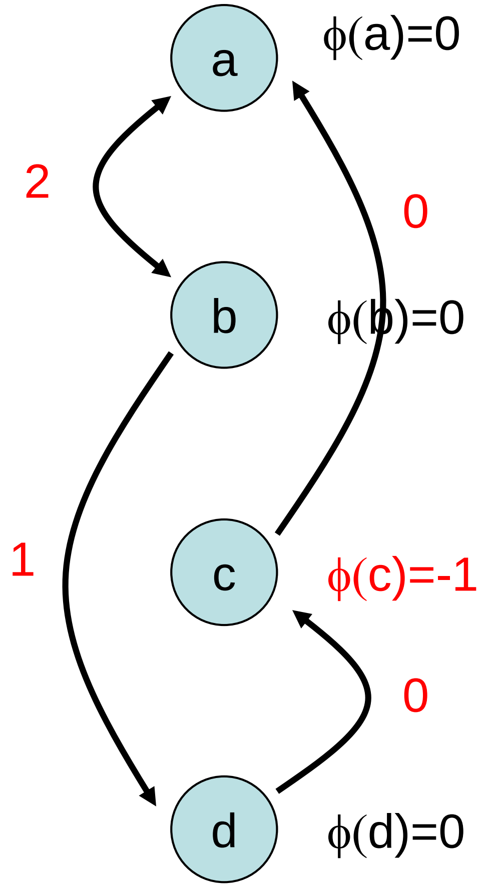


Johnson´s Method

Step 3: compute r(e)-values

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

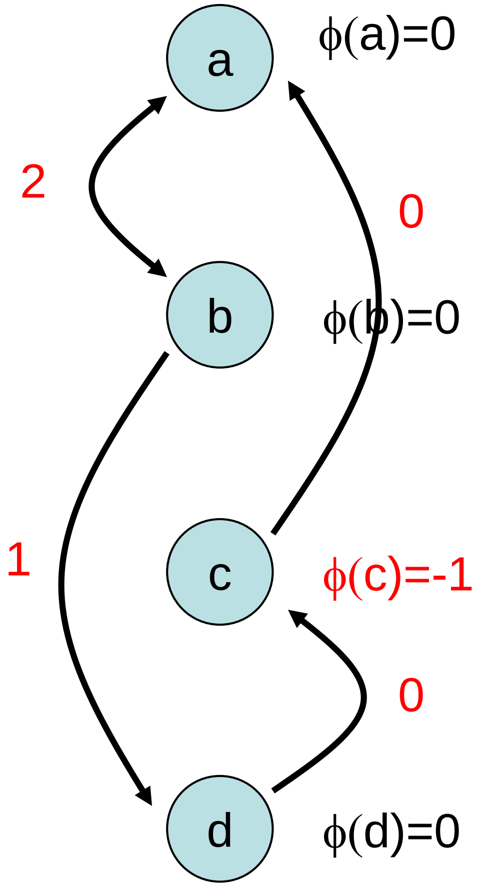
$$r(e) := \phi(v) + c(e) - \phi(w)$$



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

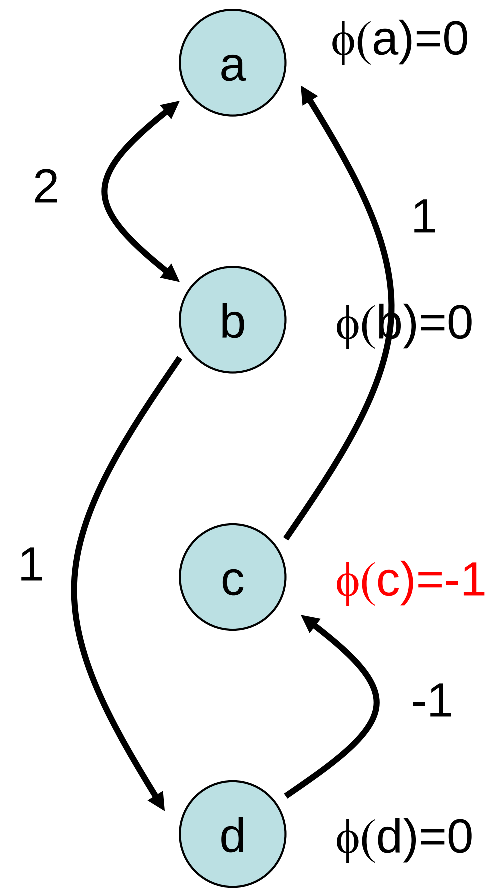


Johnson´s Method

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 (amortized runtime).

- 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?

Question

Can we “sparsify” an input graph, i.e. reduce the number of edges, while approximately preserving distances between vertices?

If so, the runtime for Johnson’s method can brought down from worst case n^3 :

$$O(n \cdot m + n^2 \log n)$$

Rest of lecture: How to quickly construct good „graph spanners“.

Graph Spanners

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

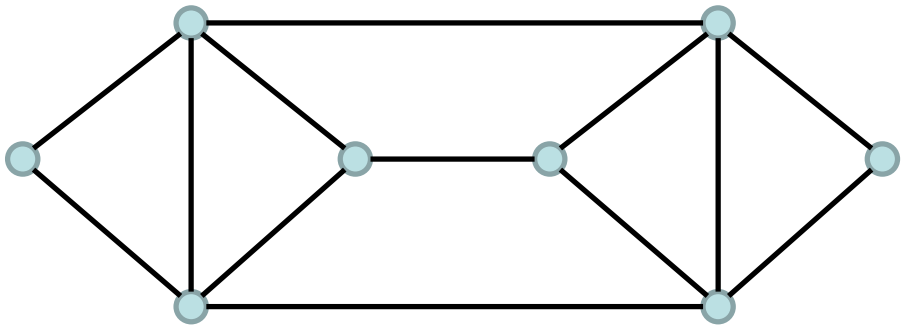
$$d_H(u,v) \le \alpha\cdot d_G(u,v) + \beta$$

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

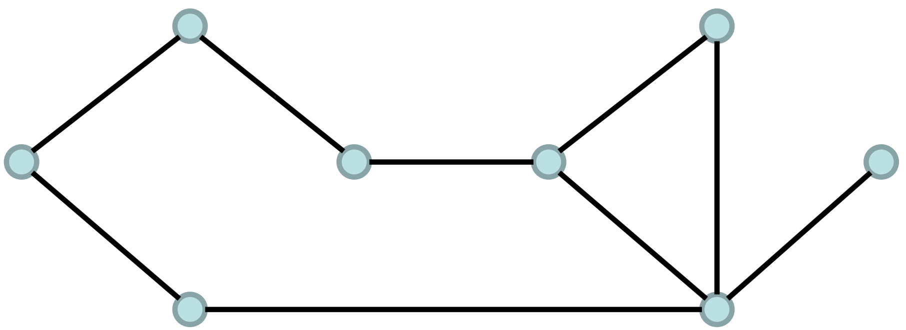
Graph Spanners

Example: all edge costs are 1

G



H



Graph Spanners

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 $u,v\in V$,

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

Observations:

1. Why can we assume WLOG edge costs are non-negative? (Hint: Problem is trivial in this case.)
2. Why is this problem not the same as computing a minimum spanning tree?

Graph Spanners

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

```
E(H):=∅
for each e={u,v}∈E(G) in the order of non-decreasing edge costs do
   //if taking e as a shortcut is „a lot“ cheaper than d_H(u,v), then add e
   if (2k-1)·c(e)<d_H(u,v) then
      add e to E(H)
```

Q: What is a naive runtime bound on the code above?

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 shortest path p from some node a to b in G.
- Case 1: If all edges of p exist in H, then $d_H(u,v)= d_G(u,v)$.
- Case 2: For any edge {u,v} in p but not in E(H):
  - Since {u,v} was rejected by the algorithm, $d_H(u,v)\leq(2k-1)\cdot c(\{u,v\})$. Thus, there is a (u,v)-path P of length at most $(2k-1)\cdot c(\{u,v\})$ in H.
  - Replacing edge {u,v} (in G) of p by P (in H) yields the claim.

Note:

1. We are implicitly using the fact that in the algorithm, $d_H(u,v)$ is monotonically non-increasing as we loop through edges. (Where do we use this assumption?)
2. This proof works if 2k-1 is replaced by any D>=1.

Graph Spanners

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

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

Q: What is a naive runtime bound on the code above?

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.

Left to prove: $|E(H)|=O(n^{1+1/k})$

Graph Spanners

Let's prove: $|E(H)|=O(n^{1+1/k})$. Need two lemmas for this. The first roughly says „a sparse graph shouldn't have short cycles".

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

Proof:

- By contradiction.
- 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.
- Since alg is greedy, $\{u,v\}$ has largest cost of all edges in C.
- Claim 1: By assumption, $C\setminus\{u,v\}$ results in a path of length at most $(2k-1)$ from u to v. Thus, $d_H(u,v)\leq(2k-1)\cdot c(u,v)$.
- Claim 2: 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.
- Claim 1 and 2 contradict one another.

Graph Spanners

Let‘s prove: $|E(H)|=O(n^{1+1/k})$. Here is the second lemma we need.

Lemma 4.8 implies that H has girth (defined as the minimum cycle length in H) 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:

- If H has at most $n+2n^{1+1/k}$ edges, claim is vacuously true.
- So assume H is a 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 $n\cdot(n^{1/k}+1)$. (Why?)
- 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.

Note: Unlike Theorem 4.6, this result cannot be scaled, i.e. we cannot trade sparsity for runtime.

Proof: Requires notion of hitting sets!

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.

Ex. V={1,2,3}, M={{1,2}, {1,3}, {2,3}}. What is a min-size hitting set? Is V itself a hitting set?

Note: Finding min-size hitting set is NP-complete.

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:

  $|S|:=\emptyset$ //stores the hitting set we are building
  for each $1\leq j\leq n$, set counter $c(j)=|\{S_i\in M: j\in S_i\}|$ //number of sets j appears in
  while $M\neq\emptyset$ do
      $k:=\text{argmax}_j\, c(j)$ //which element j appears in largest number of sets?
      $S:=S\cup\{k\}$
      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 $c(j)$, update key/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 $j\in V\setminus\{k_1,\dots,k_{j-1}\}$ must be $m_{j-1}R$. Since the algorithm is greedy, $c(k_j)$ must be at least the average count, which is $m_{j-1}R/(n-j+1)$.
- Therefore,

  $m_j \le (1-R/(n-j+1))\cdot m_{j-1} \le n\ \Pi_{l=0}^{j-1}\ (1-R/(n-l))$ (Hint: Apply bound recursively!)

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

Graph Spanners

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

Question: How to use hitting sets to build spanner?

Observe:

- If all vertices have degree $\leq \sqrt{n}$, then Thm 4.10 is vacuously true. So high-degree vertices are the „problem“.

- Idea: High-degree vertices have many neighbors through which they can be reached. So, only keep only a cleverly chosen small number of edges to these neighbors.
  - Do this using hitting sets!

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 (at most) 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.

- Question: Why does the = above hold? Hint: Why did we do BFS on s?

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 info 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 ($\Theta(n^2)$) (Why?)

Alternative approach, if approximate answers are sufficient:

- Compute additive or multiplicative spanner, and run an APSP algorithm on that spanner.
  $\rightarrow$ 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
  $p_A(v) := \mathrm{argmin}_{y \in A}\, d(v,y)$ //find vertex in A closest to v
  // find vertices in V closer to v than those in A
  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

A helpful lemma to bound storage requirements:

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.

Storage

- Lemma 4.17 implies that the storage space needed by our distance oracle is $O(n\cdot\sqrt{n}\log n)$.

Query time

- A query (u,v) is processed as follows:

  if $d(u,v)$ is stored in the hash table then

  return $d(u,v) =: D(u,v)$

  else

  return $d(u,p_A(u))+d(v,p_A(u)) =: D(u,v)$
- This obviously takes constant time.

Query accuracy/quality

For current case k=2, 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

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

Distance Oracles

Proof of Theorem 4.16 (continued):

Preprocessing runtime

- 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 $O(n \cdot \sqrt{n} \log n)$
- One can run a modified version of Dijkstra to compute $A(v) := \{x \in V \mid d(v,x) < d(v,p_A(v))\}$ for each $v \in V \setminus A$ in time $O(|E(A(v))| + |A(v)| \log n)$, where $E(A(v))$ is the set of all edges containing vertices of $A(v)$.

  One can show that $\Sigma_{w \in V} |E(A(w))| = O(m \cdot \sqrt{n})$
- Thus, 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 $1\leq i\leq k-1$, 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 $i\leq k-1$, $p_i(v)$.

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…