CSC3160: Design and Analysis of Algorithms

Week 2: Single Source Shortest Paths

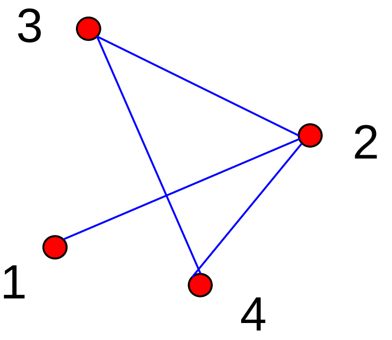
Instructor: Shengyu Zhang

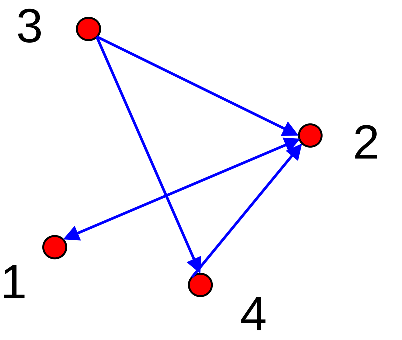
Content

- Graphs: model, size, distance.
- Problem: shortest path.
- Algorithms:
  - BFS: unweighted
  - Dijkstra: non-negative weights
  - Bellman-Ford: negative weights

Abstract model

- Graph: $G = (V, E)$
  - V: set of nodes/vertices/points
  - $E \subseteq V \times V$: set of edges





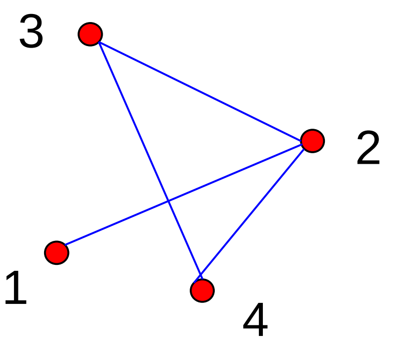
undirected graph:
Edges have no directions

directed graph:
Edges have directions

Graph, graph, graph…

- Why graph? There are lots of graph examples in our lives.
- Name one.
  - Information: WWW, citation
  - Social: co-actor, dating, messenger, communities
  - Technological: Internet, power grids, airline routes
  - Biological: Neural networks, food web, blood vessels
  - …

Representations of graphs



- Adjacency matrix:
  - $A = [a_{ij}]$, where

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$$

  - for general graphs

$$A = \begin{array}{cc} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} & \begin{array}{l} \cdots 1 \\ \cdots 2 \\ \cdots 3 \\ \cdots 4 \end{array} \\ \begin{array}{cccc} \vdots & \vdots & \vdots & \vdots \\ 1 & 2 & 3 & 4 \end{array} & \end{array}$$

- Adjacency list
  - for sparse graphs

1: 2
2: 1, 3, 4
3: 2, 4
4: 2, 3

Size of graph

- The size of a graph:
  - Adjacency matrix: $|V|^2$.

$$A = \begin{array}{cl} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} & \begin{matrix} \cdots 1 \\ \cdots 2 \\ \cdots 3 \\ \cdots 4 \end{matrix} \\ \begin{matrix} \vdots & \vdots & \vdots & \vdots \\ 1 & 2 & 3 & 4 \end{matrix} & \end{array}$$

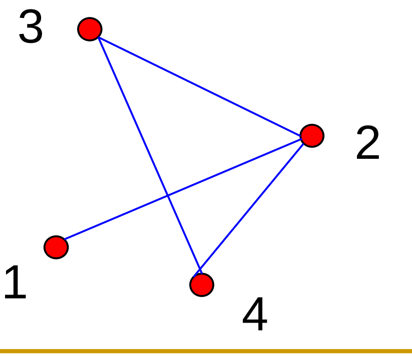
  - Adjacency list:
    - $|V| + 2|E|$ for undirected graphs.
      - Each undirected edge is counted twice.
    - $|V| + |E|$ for directed graphs.
      - Each directed edge is counted once.

$1\!: \ 2$
$2\!: \ 1, 3, 4$
$3\!: \ 2, 4$
$4\!: \ 2, 3$

Distance

- Next we focus on undirected graphs
  - Directed graphs are similarly handled.
- A path from i to j: $i \to v_1 \to v_2 \to \cdots \to v_k \to j$.
  - There may be more than one path from i to j.
- $d(i,j)$ = # edges of a shortest path from i to j

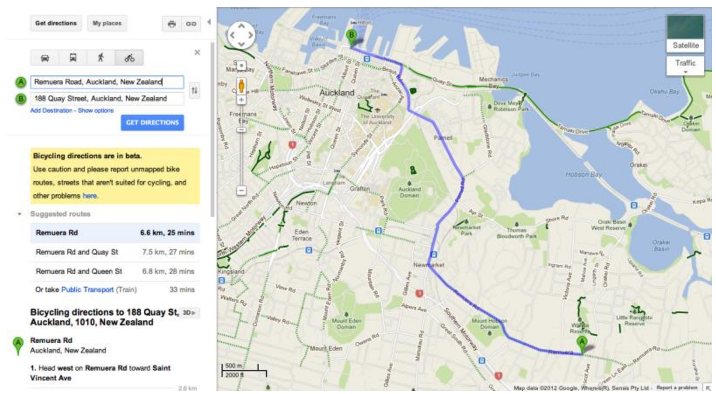
- $N(v)$ = {v's neighbors}
  $= \{u: d(v,u) = 1\}$



- A natural question: compute the distance and a shortest path between vertices
  - $s \to t$: st-distance
  - $s \to$ all other vertices: Single-Source Shortest Paths
  - all vertices $s \longrightarrow$ all other vertices t: All-Pair Shortest Paths

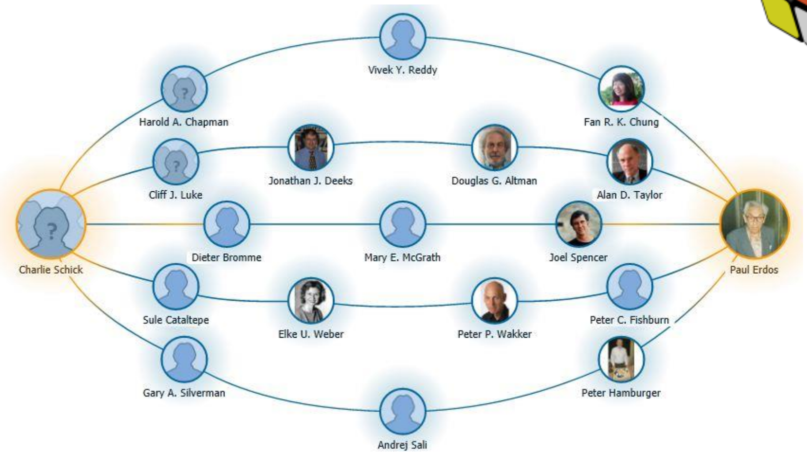
Why shortest paths?

- Google map for directions



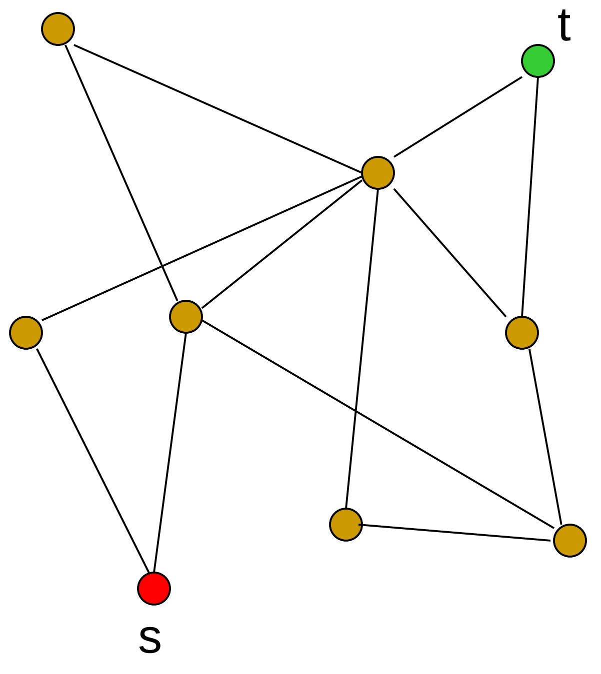
- Optimal solution of Rubik's cube.
  - Guess what's [the number]?

- Erdős number

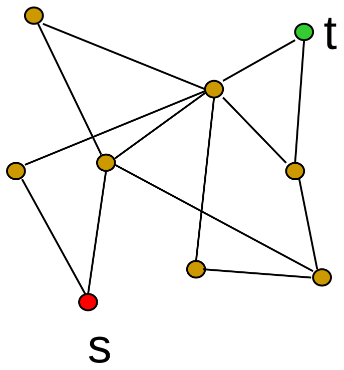


st-distance

- Let's consider the simplest case first: *st*-distance in an undirected graph.
- *How to do it?*
  - Even a very inefficient algorithm is ok.



BFS

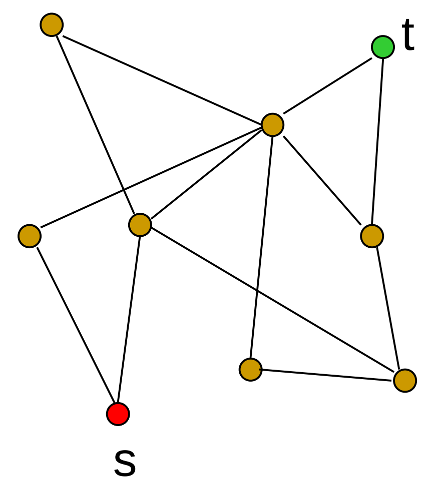


- One way of thinking:
- Methodology 1: Start from simple cases
  - Methodology 1.1: Start from the case in which some parameter is small
- Let's consider the following question:

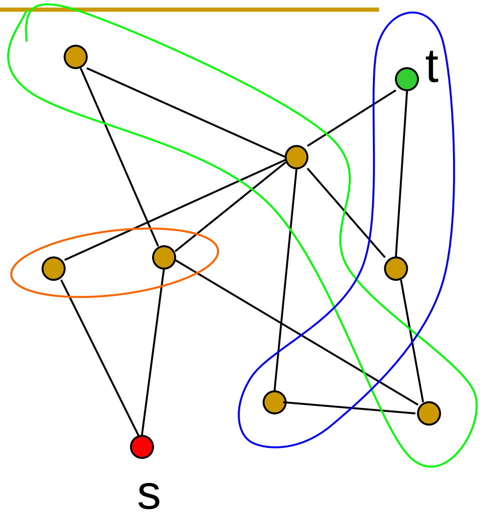
  Can we at least know whether $d(s,t) = 1$*?*
- This is very simple: just check whether t is a neighbor of s.

Little by little…

- Let's go slightly further: Can we know whether $d(s,t) = 2$?
- Not hard either: Just see whether t is a neighbor of some neighbor of s.
- Note that some neighbors of neighbors of s may have been seen before either as s itself or as a neighbor of s.



In general?



- $N_1(s) = \{$all neighbors of $s\}$
  - the vertices with distance 1 from s.
- $N_2(s) = \{$all neighbors of $N_1(s)\} - N_1(s) - \{s\}$
  - the vertices with distance 2 from s.
- $N_3(s) = \{$all neighbors of $N_2(s)\} - N_2(s) - N_1(s) - \{s\}$
  - the vertices with distance 3 from s.
- …
- $N_i(s) = \{$all neighbors of $N_{i-1}(s)\} - N_{i-1}(s) - \cdots - N_1(s) - \{s\}$
- If we find t in this step i, then $d(s,t) = i$.

BFS

- This is called the *breadth-first search* (BFS).
- Why it works?

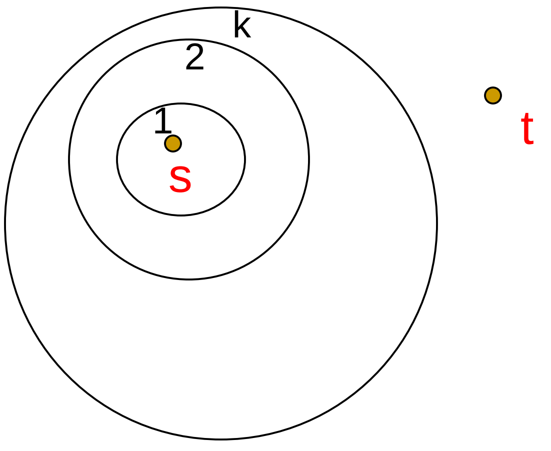
- [Thm] If we find t in Step k, then $d(s,t) = k$.

Or equivalently,

- [Thm] $N_k(s)$ contains exactly those vertices with distance k from s.

Proof of $N_k(s) = \{v: d(v,s) = k\}$

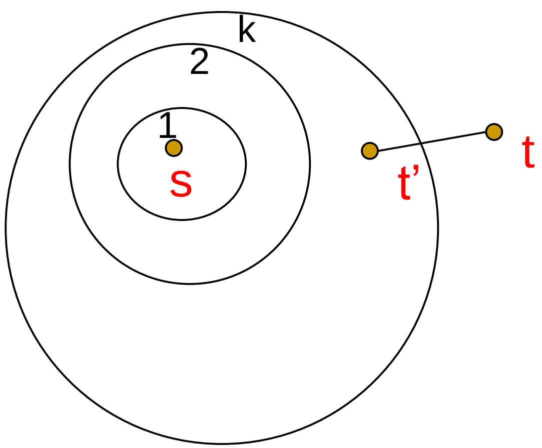
- Let's prove this by induction on k.
- $k = 1$: trivially true.
- Suppose k is correct, consider $k + 1$. Need:
  - 1. If $d(s,t) = k + 1$, then $t \in N_{k+1}(s)$
  - 2. If $t \in N_{k+1}(s)$, then $d(s,t) = k + 1$



1. If $d(s,t) = k + 1$, then $t \in N_{k+1}(s)$

Recall: $N_i(s) = \{\text{all neighbors of } N_{i-1}(s)\} - N_{i-1}(s) - \cdots - N_1(s) - \{s\}$

- A shortest path from s to t has length $k + 1$
- Just before reaching t, the path reaches some t' with $d(s,t') = k$ and $(t',t) \in E$.
- By induction, $t' \in N_k(s)$. So by algorithm, $t \in N_{k+1}(s)$ … …unless $t \in N_i(s)$ for some $i \leq k$
  - But the bad case won't happen since otherwise $d(s,t) \leq k$ by induction.



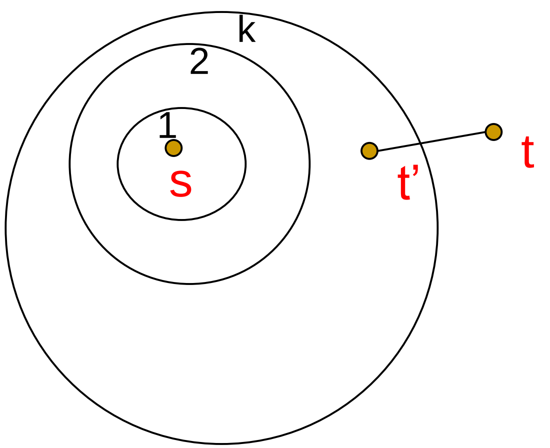
2. If $t \in N_{k+1}(s)$, then $d(s,t) = k+1$

Recall: $N_i(s) = \{$all neighbors of $N_{i-1}(s)\} - N_{i-1}(s) - \cdots - N_1(s) - \{s\}$

- $d(s,t) \le k+1$: Why?

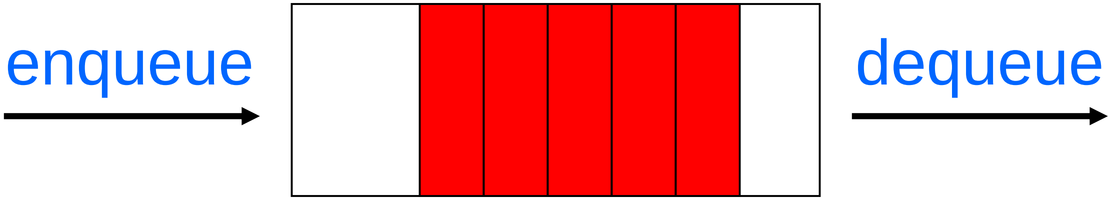
  since t is a neighbor of some vertex $t' \in N_k(s)$,
  - $d(s,t') = k$ by induction.
- $d(s,t) \ge k+1$: Why?

  $d(s,t)$ won't be $\le k$ since otherwise it'd have been covered by some $N_i(s)$ with $i \le k$. (By induction)



Implementation of the algorithm

- Queue: first in first out.
- Basic operations:
  - enqueue
  - dequeue



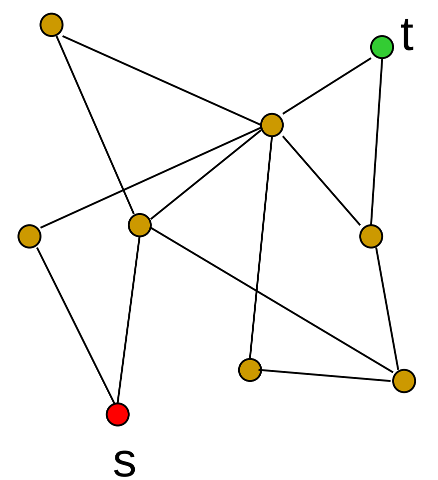
Algorithm for st-distance

- Initialize: $dist(s) = 0$; $dist(u) = \infty$ for all other u,
- $Q = [s]$
- While Q is not empty
  - Dequeue the top element u of Q
  - *// Enqueue all neighbors v of u that haven't been covered so far into Q, with $dist$ function updated*

    For all neighbors v of u, if $dist(v) = \infty$,
    - enqueue(v)
    - $dist(v) = dist(u) + 1$
  - If t is found, then stop and output $dist(t)$

Let's run it step by step together on the board!

- $dist(s) = 0$; $dist(u) = \infty$ for all other u,
- $Q = [s]$
- While Q is not empty
  - Dequeue the top element u of Q
  - For all neighbors v of u, if $dist(v) = \infty$,
    - enqueue(v)
    - $dist(v) = dist(u) + 1$
  - If t is found, then stop and output $dist(t)$



Complexity

- Initialize:
  - $dist(s) = 0$; $dist(u) = \infty$ for all other u - $|V|$
- $Q = [s]$ - 1
- While Q is not empty
  - Dequeue the top element u of Q - 1
  - For all neighbors v of u, if $dist(v) = \infty$, -$N(u)$
    - enqueue(v) - 1
    - $dist(v) = dist(u) + 1$ - 1
  - If t is found, then stop and output dist(t) - 1
- Total: $|V| + \sum_{u \in V} |N(u)| = O(|V| + |E|)$

One observation

- If we don’t stop when finding t, then eventually the algorithm finds the distances from s to all other nodes u.

Map

- Finished: On unweighted graphs, distance defined as the min # of edges
  - BFS
  - Complexity: $O(|V| + |E|)$
- Next:
  - non-negative weighted graphs.
  - Negative weighted graphs

Weighted edges

- More general: each edge has a non-negative length.
  - A length function $l(x, y)$ is given.
- $l(path)$ = sum of lengths of edges on $path$
- $l(s, t)$ = min $l(path)$ over all $paths$ from s to t

- *Question: How to do now?*
- Let's try BFS first.

BFS Algorithm for st-distance

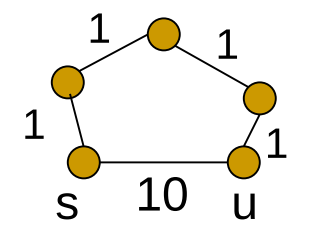
- Initialize: $dist(s) = 0$; $dist(u) = \infty$ for all other u,
- $Q = [s]$
- While Q is not empty
  - Dequeue the top element u of Q
  - (Enqueue all neighbors v of u that *haven't been covered so far* into Q, with $dist$ function adjusted)

    For all neighbors v of u, if $dist(v) = \infty$,
    - enqueue(v)
    - $dist(v) = dist(u) + \cancel{1}\ l(u, v).$

    Is this correct?
  - If t is found, then stop and output $dist(t)$

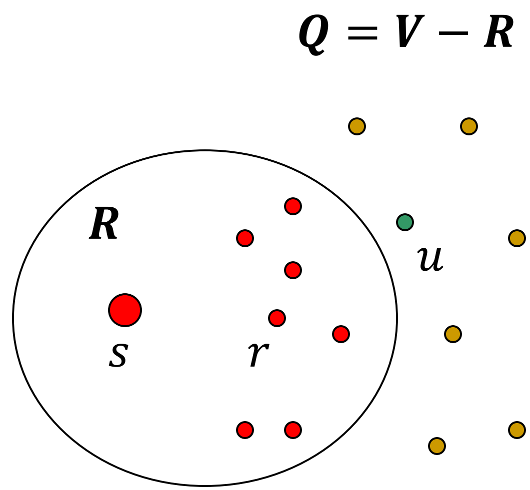
Problem of BFS

- Nodes collected at iteration i may have a shortest path with more than i edges.
- $dist(u)$, the "distance" we keep in algorithm, is only an upper bound of the real distance $l(s,u)$.
  - i.e. $dist(u) \geq l(s,u)$.
  - It's not necessary $l(s,u)$ yet since we may find better route later.
- As a result, after iteration i, we don't know $l(s,u)$ for $u \in N_i(s)$.
  - though we know an upper bound of $l(s,u)$.



Interesting things coming…

- The upper bound is tight for some vertices r.
  - $dist(r) = l(s, r)$.
- Suppose we maintain a set R of correct vertices
  - i.e. $r \in R \Rightarrow dist(r) = l(s, r)$
- We want to find another correct vertex u in $V - R$
  - s.t. we can put u into R (and then update u's neighbors).
- *Question: Which u to pick?*



When you want to pick something…

- Methodology 2: Good properties often happen at extremal points.
- Let's consider to pick the currently "best" one.
  - The u with the $\min_{u \in V-R} dist(u)$
- Recall that now $dist(u)$ is only an upper bound of $l(s, u)$
  - It corresponds to a path we've found so far, but there may be better routes found later.

Dijkstra's algorithm

- Initialize: $dist(x) = \infty$ for all $x \neq s$, and $dist(s) = 0$.
- Let Q contain all of V // $Q = V - R$
- **while** $Q \neq \emptyset$

  find a u with $\min_{u \in Q} dist(u)$

  delete u from Q

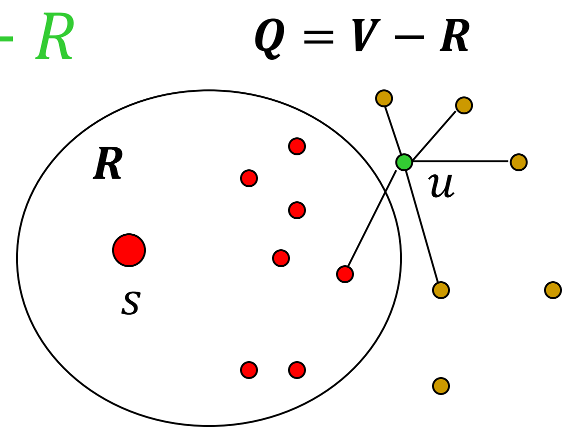
  for each $y \in N(u)$

  // update $N(u)$

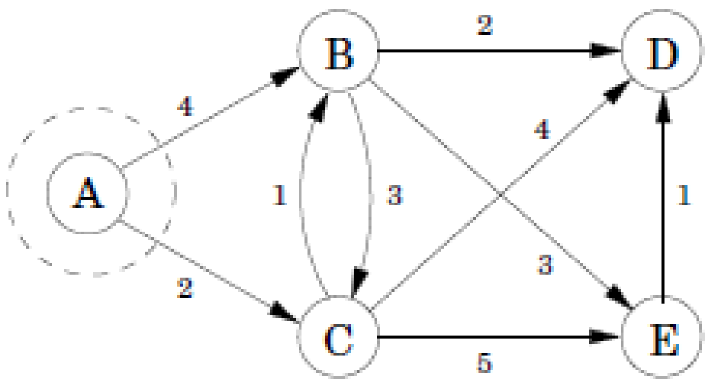
  if $dist(y) > dist(u) + l(u, y)$

  $dist(y) = dist(u) + l(u, y)$

  // update the estimated upper bound

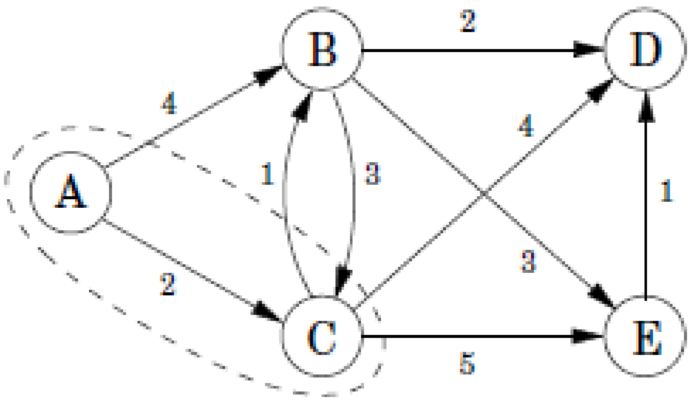


Running on an example



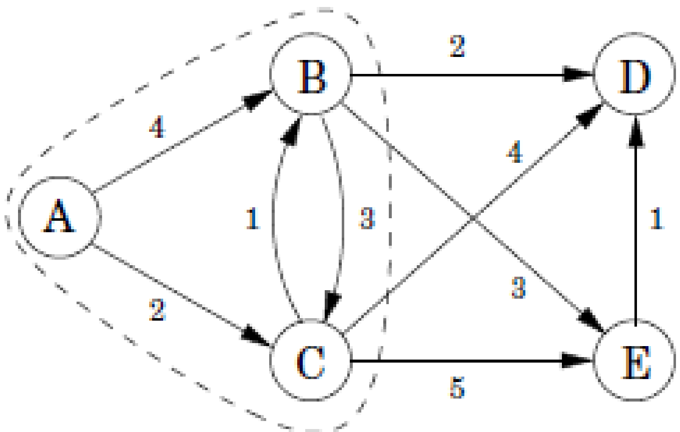
A: 0	D: ∞
B: 4	E: ∞
C: 2	

Running on an example (continued)



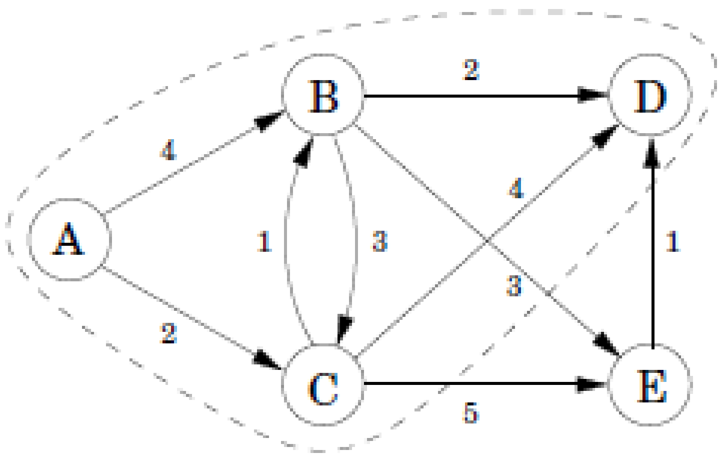
A: 0	D: 6
B: 3	E: 7
C: 2	

Running on an example (continued)



A: 0	D: 5
B: 3	E: 6
C: 2	

Running on an example (continued)

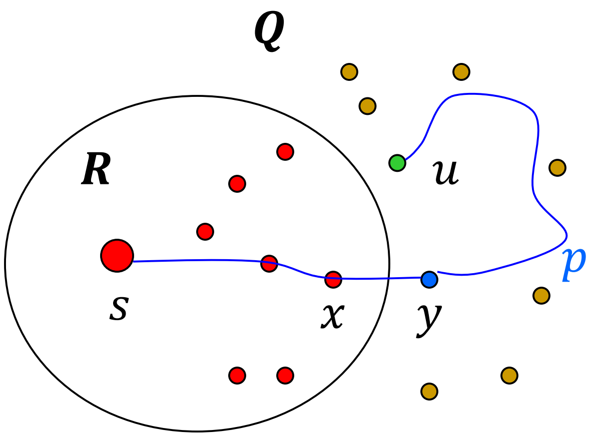


A: 0	D: 5
B: 3	E: 6
C: 2	

Key property in the proof

- Recall what we want: u achieving the minimum in $\min_{u \in V-R} dist(u)$ always has $dist(u) = l(s, u)$
- The whole idea and proof is in the next slide.

Proof of the key property: $dist(u) = l(s,u)$.



- Recall: $dist(u) \geq l(s,u)$.
- Will show: $dist(u) \leq l(s,u)$.
- Take a shortest path p from s to u
- Suppose p leaves R (for 1st time) by edge (x,y).
- [Claim] $dist(y) = l(s,y)$.
  - The part of p from s to y is a shortest path to y.
    - Any prefix of a shortest path ($s \rightarrow u$) is a shortest path itself ($s \rightarrow y$).
  - $dist(x) = l(s,x)$ since $x \in R$.
  - So $dist(y)$ has been tightened to $l(s,y)$ when x updates its neighbors
- So $dist(u) = \min_{w \in Q} dist(w) \underset{y \in Q}{\leq} dist(y) \underset{\text{Claim}}{=} l(s,y) \underset{\text{part} \leq \text{whole}}{\leq} l(p)$.

Map

- We've shown Dijkstra's algorithm for st-shortest path, and proved its correctness.

- Next:
  - Implementation (of min-finding) and complexity
  - Shortest path for negative weighted graphs

Complexity

- Initialize: $dist(x) = \infty$ for all $x \neq s$, and $dist(s) = 0$ - $|V|$
- Let Q contain all of V - $|V|$
- **while** $Q \neq \emptyset$

  find a u with min $dist(u)$, put it into R - delete-min cost

  for each $y \in N(u)$ - $|N(u)|$

  // update $N(u)$

  if $dist(y) > dist(u) + l(u, y)$

  $dist(y) = dist(u) + l(u, y)$ - decrease-key cost

  // update the estimated upper bound

- Total: $|V| \cdot$ (delete-min cost) $+ \ |V| + \ O(|E|) \cdot$ (decrease-key cost)

Implement of the queue

- We want a queue good for delete-min
- *priority queue*
- delete-min cost and decrease-key cost depend on the implementation of priority queue.
  - Array:
    - delete-min cost: length of Q, which is $\leq |V|$ in general.
    - decrease-key cost: $O(1)$
    - Total cost: $O(|V|^2)$.

Recall: Total cost = $|V| \cdot$ (delete-min cost) + $|V| + O(|E|) \cdot$ (decrease-key cost)

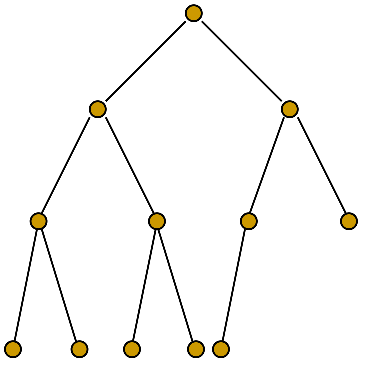
Other choices

- Binary heap
  - Much smaller delete-min cost: $\log(|V|)$
  - Slightly larger decrease-key cost: $\log(|V|)$.
- Total: $|V| \cdot$ (delete-min cost) $+\ |V| +\ O(|E|) \cdot$ (decrease-key cost)

  $= O(|V| \log|V| + |V| + |E| \log(|V|))$

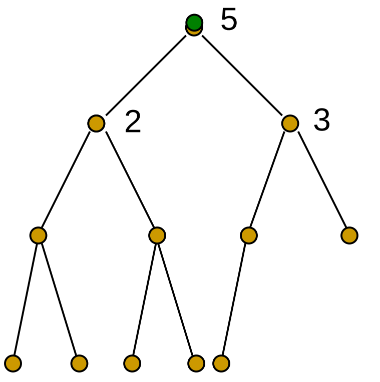
  $= O((|V| + |E|) \log(|V|)\ )$
  - Better than the array's cost $O(|V|^2)$ when $|E|$ is small
- d-ary heap: Similar except that it's now a complete d-ary tree.
- Fibonacci heap: even better decrease-key cost.
  - Details omitted; see the book.

Binary heap



- Complete binary tree: filled top-down, left-to-right
  - Depth: ≈ $\log_2(n)$, where n: # nodes
- A complete binary tree with the following property maintained:
  - Parent's value ≤ children's values
- The property implies that *the root has the min value*
- Good: really easy to find min.
- Bad: deleting the root makes it not a tree any more.

delete-min



- delete-min:
  - dequeue the root.
  - Put the last leaf at the root
  - Let it sift down
    - If it's bigger than either child's value
      - Swap it and the smaller child
- Property "Parent's value ≤ children's values" is kept.
- Cost: $\log_2(|V|)$. ($\because$ height of tree $\leq \log_2(|V|)$)

DecreaseKey

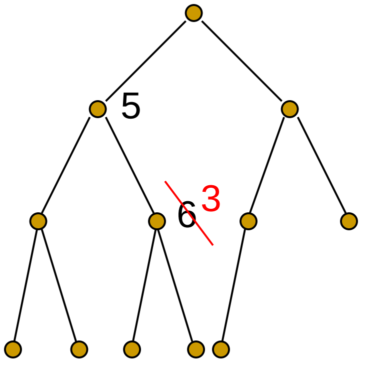
Recall:

if $dist(y) > dist(u) + l(u, y)$

$dist(y) = dist(u) + l(u, y)$

- cost decrease-key

- DecreaseKey:
  - After decreasing the key value,
  - Bubble it up:
    If it's smaller than its parent
    - Swap them.
- Property is maintained:
  - Parent's value ≤ children's values
- Cost: $\log(|V|)$



- Total: $|V| \cdot$ (delete-min cost) + $|V|$ + $O(|E|) \cdot$ (decrease-key cost)

  $= O(|V|\log(|V|) + |V| + |E|\log(|V|))$

  $= O((|V| + |E|)\log(|V|))$
- Better than the array's cost $O(|V|^2)$ when $|E|$ is small

Map

- We talked about Single Source Shortest Paths problem
  - On unweighted graphs, distance defined as the # of edges
    - BFS
  - On weighted graphs, distance defined as the sum of lengths of edges
    - Dijkstra's algorithm
- Next: on graphs with negative weights
  - Bellman-Ford

Further generalization

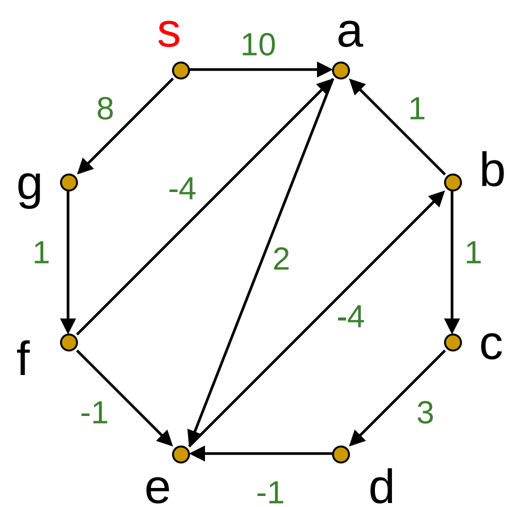
- Allow negative weights on edges?
- How to define the length of a path?
  - For $p = v_1 \to v_2 \to \cdots \to v_t$,
  - Naturally as before,

$$w(p) = \sum_{i=1,\dots,t-1} w(v_i, v_{i+1})$$

  - Only difference is that now some $w()$ may be < 0.
- Problem?

negative cycle

- For the graph as given, what's the shortest path from s to b
  - $s \to g \to f \to e \to b$: 4
  - ... $\to c \to d \to e \to b$: 3
- In general, negative cycles make "shortest paths" meaningless.
  - cycles with negative length
- So let's only consider graphs without negative cycle
- In particular, only directed graphs
  - undirected: negative edge = negative cycle



Requirements

- For a general graph, we thus desire an algorithm that
  - 1) tells whether the graph contains a negative cycle, and
  - 2) if not, computes the shortest paths
- Bellman-Ford's algorithm: achieve both!
- Let's first assume no negative cycle, and come back to this case later.

Idea of Bellman-Ford

- Methodology 3: Analyze properties of an optimal solution.
- For each point v, there is a shortest path from s to v:
  - $(s =)v_0 \to v_1 \to v_2 \to \cdots \to v_t(= v)$
- Recall: Prefix $(v_0 \cdots v_i)$ of any shortest path $(v_0 \cdots v_t)$ is a shortest path of $v_0 \to v_i$.
- So if we've found $v_0 \cdots v_i$, then updating v_i's neighbors' values finds shortest path of $v_0 \to v_{i+1}$.
  - Solved if we update $v_1, v_2, \ldots, v_t$ in this order ☺
- Issue: We don't know what these v_i's are.
- Solution: We update the whole graph
  - i.e. update $N(v)$'s values for all $v \in V$.

Bellman-Ford's algorithm

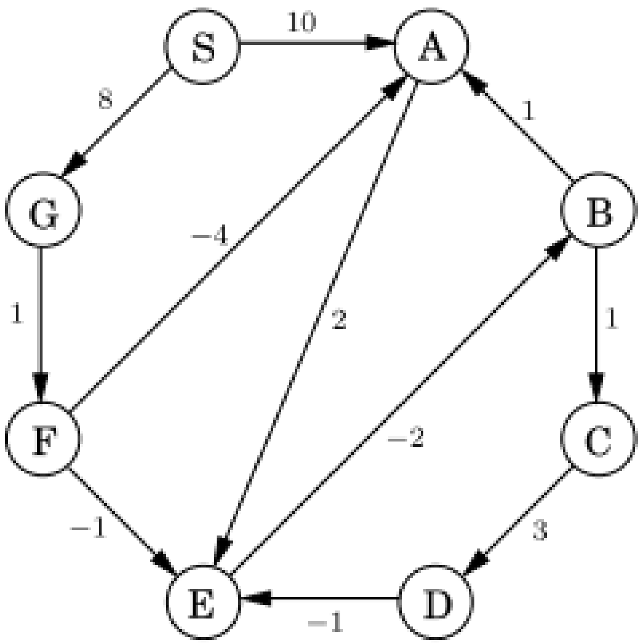
- $dist(s) = 0$ and $dist(u) = \infty$ for all $u \neq s$
- **for** $|V| - 1$ times

  for each $(x, y) \in E$,

  if $dist(y) > dist(x) + w(x, y)$ (1)

  $dist(y) = dist(x) + w(x, y)$ (2)

Execution on an example



	Iteration							
Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Correctness: suppose no negative cycle

- For each point v, there is a shortest path from s to v:
  - $(s =)v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_t(= v)$
- [Claim] After i steps, we have

  $dist(v_i) \leq w(s, v_i)$ // by induction
- [Claim] $dist(v_i) \geq w(s, v_i)$
  - $dist(v_i)$ is still an upper bound of $w(s, v_i)$
  - because $dist(v_i)$ is updated only based on paths found so far.
- Thus after t steps, we have $dist(v_t) = w(s, v_t)$.

How large could t be?

- [Obs] $t \leq |V| - 1$.
- Otherwise some vertex repeated twice in the path,
  - i.e. there is a cycle in the path
- We assume that all cycles have non-negative weights
- Deleting the cycle can never be worse.

Complexity

- $dist(s) = 0$ and $dist(u) = \infty, \forall u \neq s$
- **for** $|V| - 1$ times - $|V|$

  for each $(x, y) \in E$, - $|E|$

  if $dist(y) > dist(x) + w(x, y)$ - $O(1)$

  $dist(y) = dist(x) + w(x, y)$

- Total: $O(|V| \cdot |E|)$

Handling negative cycles

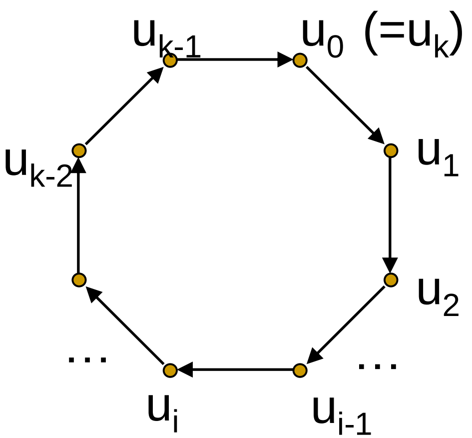
- Add one more round (after the $|V| - 1$ ones): **if** $dist(x)$ decreases for any x,
  report the existence of a negative cycle.
- [Claim] ∃negative cycle (reachable from s)
  ⇔ dist(x) decreases in the extra iteration
  - ⇐: trivial
  - ⇒: let's look at this part more carefully

$\exists$negative cycle $u_0 \to u_1 \to u_2 \to \cdots \to u_{k-1} \to u_k (= u_0)$ $\Rightarrow \exists i, dist(u_i)$ decreases in the extra iteration

- all $dist(u_i)$ don't decrease
  $\Rightarrow dist(u_i) \le dist(u_{i-1}) + w(u_{i-1}, u_i), \forall i$
- Sum up all these inequalities:

$$\begin{aligned} & dist(u_1) + \cdots + dist(u_k) \\ \le\ & dist(u_0) + \cdots + dist(u_{k-1}) \\ & + w(u_0, u_1) + \cdots + w(u_{k-1}, u_k) \end{aligned}$$

  - Note that $u_k = u_0$, thus the $dist()$ values cancel
- So $0 \le w(u_0, u_1) + \cdots + w(u_{k-1}, u_k)$, contradictory to our assumption of negative cycle.



In summary

- On unweighted graphs, distance defined as the min # of edges
  - BFS
  - Complexity: $O(|V| + |E|)$
- On non-negative weighted graphs, distance defined as the min sum of lengths of edges
  - Dijkstra's algorithm
  - Complexity: $O\big((|V| + |E|)\log|V|\big)$
- On general weighted graphs:
  - Bellman-Ford algorithm
  - Complexity: $O(|V| \cdot |E|)$

More algorithms (negative weight)?

- [Gabow and Tarjan] $O(\sqrt{|V|}|E|\log(VW))$
  - $W = \max_{(u,v)\in E}\{|w(u,v)|\}$.
  - *H. Gabow and R. Tarjan. Faster scaling algorithms for network problems. SIAM Journal on Computing, 18(5): 1013–1036, 1989.*
- [Goldberg] $O(\sqrt{|V|}|E|\log(W))$
  - *A. Goldberg. Scaling algorithms for the shortest paths problem. SIAM Journal on Computing, 24(3): 494–504, 1995.*
- An extensive overview of shortest path algorithms, in both theory and experiment.
  - *B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. Mathematical Programming, 73(2): 129–174, 1996.*