Lecture Note 3

CSCI 6470 Algorithms (Fall 2024)

Liming Cai

School of Computing UGA

September 11, 2024

Chapter 3. Algorithms on graphs

Topics to be discussed:

- Basics and representations of graphs
- Depth-first search and applications
- Shortest path algorithms
- priority queue

1. Fundamentals of graphs

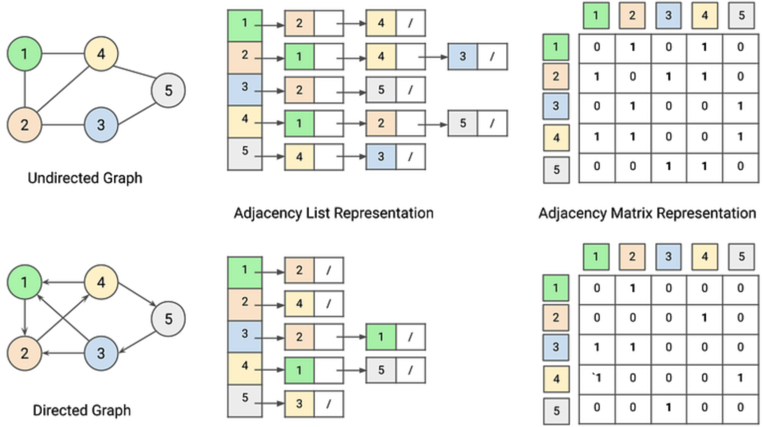
Terminologies:

- vertex, edge, graph, degree, neighbor, weight, directed edge,
- di-graph, subgraph, tree, path, cycle,
- connected component, strongly connected component,
- complete graph, planar graph, non-planar graph, bi-partite graph

1. Fundamentals of graphs

Computer representations of graphs:

adjacency list, adjacency matrix



Undirected Graph

Adjacency List Representation

Adjacency Matrix Representation

Directed Graph

1. Fundamentals of graphs

Recursive definition for trees

- set pair $(\{x\}, \emptyset)$ is a **tree**;
- if (V, E) is a **tree**, vertex $u \in V$, and vertex $v \notin V$, then $\big(V \cup \{v\}, E \cup \{(v, u)\}\big)$ is a **tree**.

Trees, created with these rules, are without a root. But the first vertex created can be designated as the root.

But why would non-biological trees need a root?

1. Fundamentals of graphs

Recursive definition for graphs

- set pair $(\{x\}, \emptyset)$ is a **graph**;
- if (V, E) is a **graph**, subset $U \subseteq V$, and vertex $v \notin V$, then (V', E') is a **graph**, where
  $V' = V \cup \{v\}$, $E' = E \cup \{(v, u) : u \in U\}$.

Proper definitions of graphs may incur some structural views on graphs and help solve various computational problems on graphs.

2. Depth-first search

Based on the recursive definition, a given graph (V', E') can be decomposed as

(1) a subgraph graph (V, E),
(2) a vertex $v \in V' - V$ (also written as $V' \backslash V$);
(3) a subset $U \subseteq V$;
(4) $\forall u \in U$, edges $(v, u) \in E' - E$ (also written as $E' \backslash E$).

Graph traversal by exploiting the recursive definition of graphs.

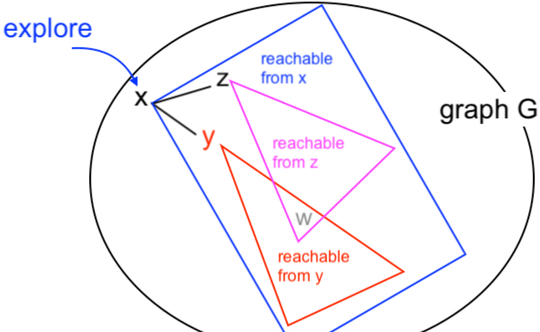
- traverse a graph:
  visit vertex v and then recursively **visit** u, for all $(v, u) \in E$.
- two different traversal methods: DFS and BFS,
  depending on which vertex is to visit next

2. Depth-first search

Assume graph G that was created with x being any vertex

Explore all vertices reachable from vertex x:

```
function explore(G: graph; x: vertex)
 1. visited(x) = true;
 2. for each edge (x, y) in G
 3.   if not visited(y)
 4.      explore(G, y);
```



2. Depth-first search

Adding time stamps:

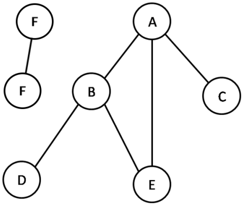
```
function explore(G: graph; x: vertex)
visited(x) = true;
pre(x) = time_stamp;          // pre-visit work
time_stamp = time_stamp + 1;
for each edge (x,y) in G
  if not visited(y)
     parent(y)=x;             // record tree edge
     explore(G, y);
post(x) = time_stamp;            // post-visit work
time_stamp = time_stamp + 1;
```

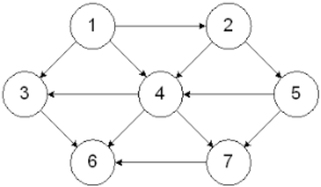
Main body

```
function main(G: graph);
for every vertex v in G
  visited(v) = false;
for every vertex v in G
  if not visited(v)
     explore(G, v);
```

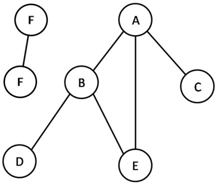
2. Depth-first search

Examples for DFS





2. Depth-first search



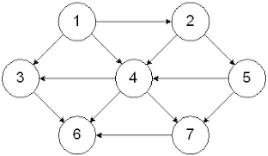
- DFS on a graph yields a **DF-search tree**:
- properties of $\mathtt{pre}(x)$ and $\mathrm{post}(x)$ values

  brackets patterns (look familiar?)
- type of edges in DFS tree

  - tree edges
  - back edges

2. Depth-first search

DFS on directed graphs



- types of edges in DFS tree

  - tree edges
  - back edges
  - forward edges
  - cross edges

- DFS on directed acyclic graphs (DAGs)

  Theorem If there is a path $x \rightsquigarrow y$, then $\text{post}(x) > \text{post}(y)$

2. Depth-first search

DFS algorithm time complexity

- `for` loops in both functions `main` and `explore` visit every every vertex (once), thus $O(|V|)$;
- for every vertex, all edges shared with its neighbors are checked at most twice (why?), thus $O(|E|)$;
- the total time is $O(|V| + |E|)$.

2. Depth-first search

Applications of DFS

- determine if the input graph is connected;
- determine if two given vertices are connected on input graph;
- determine if the input graph contains a cycle;
- find a topological order of vertices on the input DAG;
- find the strong connected components of the input graph;
- find single-source shorted paths on the input DAG;

All have time complexity: $O(|E| + |V|)$

2. Depth-first search

Determine if the input graph contains a cycle;

The graph has a cycle if, when encountered with edge `(x,y)` in function `explore`, `(x,y)` is a back edge, i.e.,

- for non-directed graphs,

  (1) `visited(y)` = `true`, and
  (2) `y` $\neq$ `parent(x)`

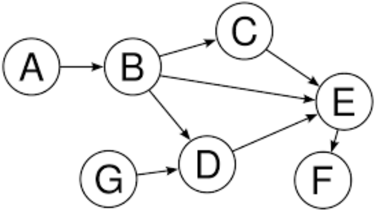
- for directed graphs,

  (1) `visited(y)` = `true`, and
  (2) `pre(y)` < `pre(x)` and `post(x)` is not defined yet

2. Depth-first search

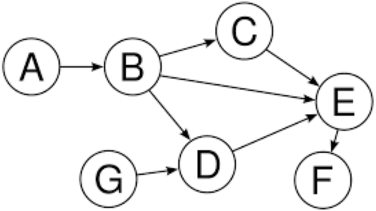
Topological Sort problem

Input: directed acyclic graph $G(V, E)$,
Output: vertices of V in order: $v_1, v_2, \ldots, v_n$ such that
$\forall i < j, (v_j, v_i) \notin E$.



Some topological sorted orders:

2. Depth-first search



- DFS can be used to solve this problem

  remember the following?

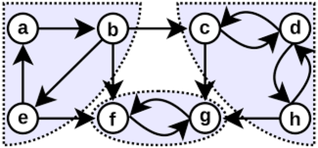
  Theorem If there is a path $x \rightsquigarrow y$, then $\text{post}(x) > \text{post}(y)$

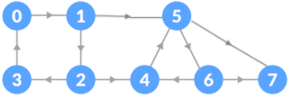
  Conclusion: Reversed order of `post` values is a topological sort order.

2. Depth-first search

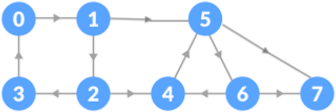
Strongly Connected Components (SCC) problem

Input: directed graph $G(V, E)$,
Output: strongly connected components for G.





2. Depth-first search



Idea:

2. Depth-first search

SCC algorithm:

- DFS on G;
- generated G^T, transpose of G (reversed edges directions);
- DFS on G^T from vertex `v` with the highest `post(v)` value;

2. Depth-first search

More about graph traversal algorithms

- non-recursive version of DFS; using (?)
- breadth first search (BFS); explores vertices in the order of their distance from the source vertex
- non-recursive BFS? using (?)
- recursive version ?
- time complexity

Review for Midterm

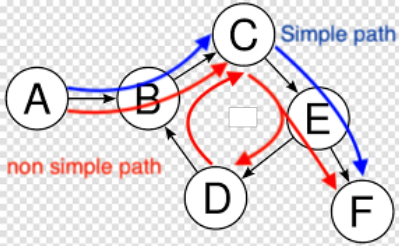
- big-O notation:
  - definition and usage to infer big-O for given time expressions
- recursive algorithms:
  - ability to read/understand, write, and to derive time function recurrence;
  - recursive algorithms for selection sort, insertion sort;
- divide-and-conquer:
  - merge sort, binary search, and variants;
  - quick sort, partition function;
  - derivation of time function recurrence for divide-and-conquer algorithms;
- proof-by-induction:
  - proof of big-O for time function with given recurrences;

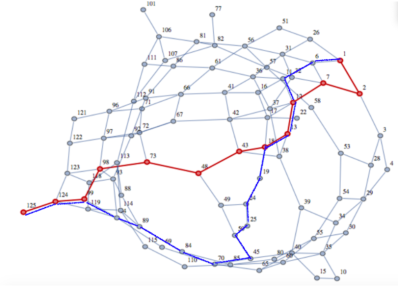
Review for Midterm

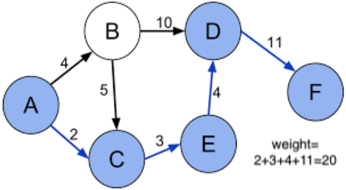
- randomized quick sort:
  - difference between randomized quick sort and deterministic quick sort;
  - averaged (expected) time for randomized quick sort, recurrence formula ;
- deterministic `selection`:
  - how the algorithm works, how time recurrence is derived;
- DFS and applications:
  - `explore` algorithm, DFS tree/forest, time stamps, edge types;
  - connectivity, cycle, topological sort, strongly connected component;

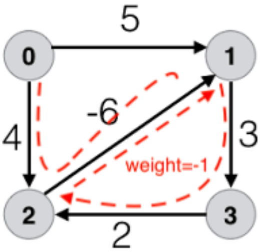
3. Shortest path problems

Paths on graphs:









3. Shortest path problems

Problems about paths on graphs:

- **Reachability** : given $G = (V, E)$, and vertices $s, t \in V$;
  asked if there is a path: $s \rightsquigarrow t$, from s to t;
- **Cycle**: given $G = (V, E)$,
  asked if there is a cycle in the graph. How exactly is this done?
- s-t **shortest path**: given $G = (V, E)$, vertices $s, t \in V$;
  find a shortest path $s \rightsquigarrow t$;
- **Single source shortest path**: given $G = (V, E)$,
  $\forall v \in V$, find a shortest path $s \rightsquigarrow v$;
- **All pair shortest path**: given $G = (V, E)$,
  $\forall u, v \in V$, find a shortest path $u \rightsquigarrow v$;
  [To be discussed in the subject of dynamic programming]

3. Shortest path problems

Shortest Distance problem:

Input: di-graph $G = (V, E)$, lengths $w : E \to R$; source $s \in V$
Output: $dist(v)$, shortest distance s to every vertex $v \in V$.

- Is the following problem easier?

  Given source s and target t, find a shortest path from s to t;

- Subpath on a shortest path is a shortest path

  if $p_{s,r,t} : s \rightsquigarrow r \rightsquigarrow t$ is a shortest path from s to t,
  then subpath $p_{s,r} : s \rightsquigarrow r$ is a shortest path from s to r. [proof?]

3. Shortest path problems

Shortest Distance problem:

Input: di-graph $G = (V, E)$, lengths $l : E \to R$; source $s \in V$
Output: $dist(v)$, shortest distance s to every vertex $v \in V$.

- Finding a shortest path from s to t implies finding shortest paths from source s to all other vertices.

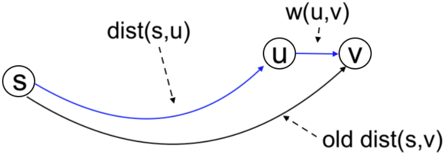
  shortest distance from s to one vertex may impact on another;

- The question becomes "which vertices should be considered before others in computing their shortest distances from s?"
- That is: is there an order of vertices whose shortest distances can be computed correctly and efficiently?

3. Shortest path problems

Shortest Distance problem:

- updating path distances,



$$\mathtt{dist}(\mathtt{v}) = \min \begin{cases} \mathtt{dist}(\mathtt{v}) \\ \mathtt{dist}(\mathtt{u}) + \mathtt{l}(\mathtt{u}, \mathtt{v}) \end{cases}$$

called relaxation of edges

3. Shortest path problems

Shortest Distance problem on 3 different types of graphs:

- G is a DAG: relying on DFS

  edges are relaxed according to topo sort order;

- G does not contain negative edges: Dijkstra's algorithm

  edges (u, v) are relaxed only after $dist(u)$ is shortest;

- G may contain negative cycles: Bellman-Ford algorithm

  edges are relaxed in an arbitrary order

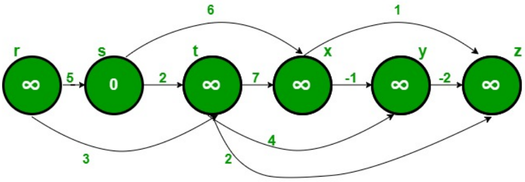
3. Shortest path problems

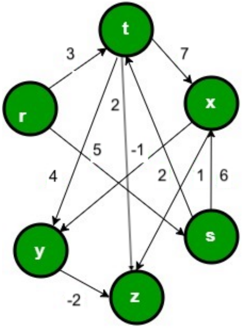
Input: DAG $G = (V, E)$, edge lengths $l : E \to R$, and s $\in V$,
Output: $\forall$ u $\in V$, the smallest distance `dist(u)` from s to u.

```
function dag-shortest-path(G, l, s)
for all u in V
  dist(u) = infinity;
  prev(u) = nil;  // predecessor of u in the path
dist(s) = 0;
topological sort V;
for all u in V in the sorted order
  for all edge (u, v) in E
      if dist(v) > dist(u) + l(u, v);
         dist(v) = dist(u) + l(u, v);
         prev(v) = u;
```

3. Shortest path problems

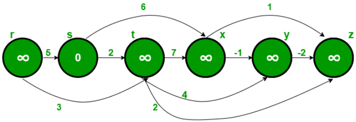
Shortest Distance problem on DAG



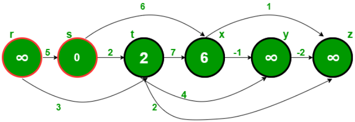


3. Shortest path problems

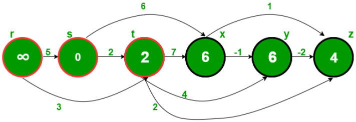
Shortest Distance problem on DAG



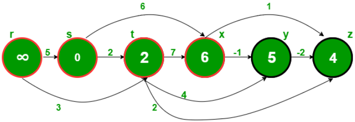
(c)



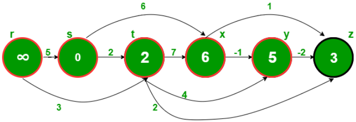
(d)



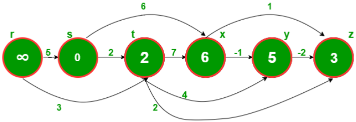
(e)



(f)



(g)



(h)

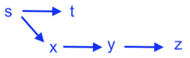
3. Shortest path problems

Shortest Distance problem on DAG

algorithm steps on the previous example

dist table							prev table					
r	s	t	x	y	z	step	r	s	t	x	y	z
inf	0	inf	inf	inf	inf	initial	nil	nil	nil	nil	nil	nil
inf	0	2	6	inf	inf	u = s	nil	nil	s	s	nil	nil
inf	0	2	6	6	4	u = t	nil	nil	s	s	t	t
inf	0	2	6	5	4	u = x	nil	nil	s	s	x	t
inf	0	2	6	5	3	u = y	nil	nil	s	s	x	y
inf	0	2	6	5	3	u = z	nil	nil	s	s	x	y

Shortest paths tree



shortest paths from s to all other vertices

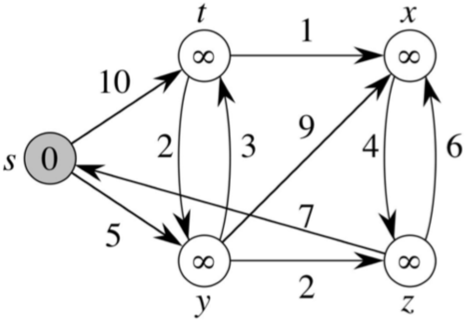
3. Shortest path problems

Shortest Distance problem: `Dijkstra's algorithm`
on general directed graphs, without negative weights

Input: $G = (V, E)$, edge lengths $l : E \rightarrow R_{\geq 0}$, and $\mathtt{s} \in V$,
Output: $\forall$ $\mathtt{u} \in V$, the smallest distance `dist(u)` from `s` to `u`.

```
function Dijkstra(G, l, s)
1. for all u in V
2.   dist(u) = infinity;
3.   prev(u) = nil;
4. dist(s) = 0;
5. H = makequeue(V);
6. While H is not empty  // body including lines 7 - 11
7.   u = dequeue(H);
8.   for all edges (u, v) in E
9.     if dist(v) > dist(u) + l(u, v);
10.       dist(v) = dist(u) + l(u, v);
11.       prev(v) = u;
12.return (prev)
```

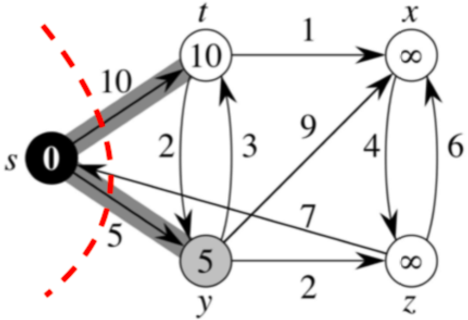
3. Shortest path problems



(a)

Note: the gray-colored vertex is to be dequeued
while the black-colored vertices are in set S.

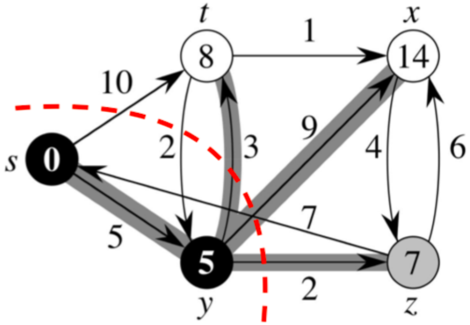
3. Shortest path problems



(b)

Note: the gray-colored vertex is to be dequeued
while the black-colored vertices are in set S.

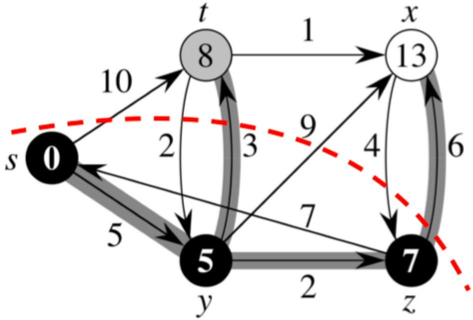
3. Shortest path problems



(c)

Note: the gray-colored vertex is to be dequeued
while the black-colored vertices are in set S.

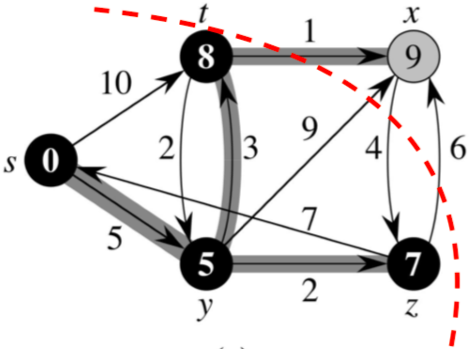
3. Shortest path problems



(d)

Note: the gray-colored vertex is to be dequeued
while the black-colored vertices are in set S.

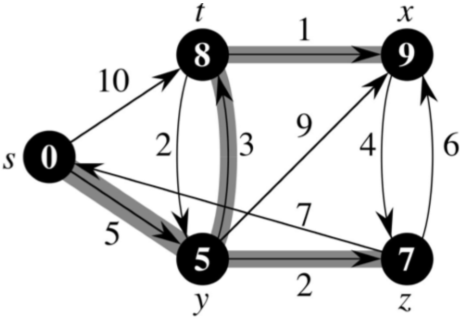
3. Shortest path problems



(e)

Note: the gray-colored vertex is to be dequeued
while the black-colored vertices are in set S.

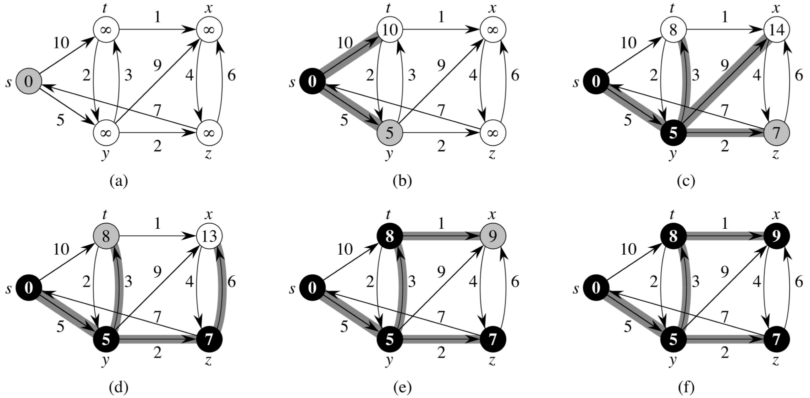
3. Shortest path problems



(f)

Note: the gray-colored vertex is to be dequeued while the black-colored vertices are in set S.

3. Shortest path problems



Note: the gray-colored vertex is to be dequeued while the black-colored vertices are in set S.

3. Shortest path problems

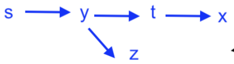
Shortest Distance problem on graphs without negative edges

Dijkstra's algorithm steps on the previous example

dist table | prev table

s	t	x	y	z	step	s	t	x	y	z
0	inf	inf	inf	inf	initial	nil	nil	nil	nil	nil
0	10	inf	5	inf	u = s	nil	s	nil	s	nil
0	8	14	5	7	u = y	nil	y	y	s	y
0	8	13	5	7	u = z	nil	y	z	s	y
0	8	9	5	7	u = t	nil	y	t	s	y
0	8	9	5	7	u = x	nil	y	t	s	y

Shortest paths tree



shortest paths from s to all other vertices

3. Shortest path problems

`Dijkstra's algorithm`

- more examples (textbook), keep track the priority queue
- what would happen if negative edges are present?

in classroom discussion

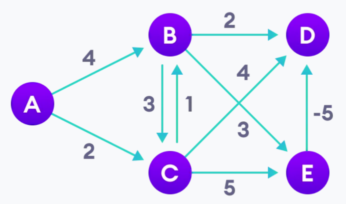
3. Shortest path problems

For graphs that may contain negative edges, edge relaxations cannot be done in specific order.

But every shortest path consists of at most $n-1$ edges; $n-1$ rounds of edge relaxations suffices

```
function Bellman-Ford(G=(V,E): graph; s: vertex)
for all u in V
  dist(u) = infinite;
  prev(u) = nil;
dist(s) = 0;
for k=1 to n-1
  for all edge (u,v) in E
    if dist(v) >  dist(u) + l(u,v)        // l(u,v) is the
       then dist(v) = dist(u) + l(u,v);   // weight of edge
       prev(v) = u;
for every an edge (u, v) in E
    if dist(u) + l(u,v) < dist(v)
       then return "G contains a negative cycle"
return (dist, prev)
```

3. Shortest path problems



The order of edges chosen to be relaxed at each round has impact on distances updated

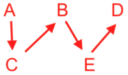
Source: A

	A	B	C	D	E
	0	∞	∞	∞	∞
Round 1	0	~~4~~ (A) 3 (C)	2 (A)	~~6~~ (B) 2 (E)	7 (C)
Round 2	0	3 (C)	2 (A)	1 (E)	6 (B)
Round 3	0	3 (C)	2 (A)	1 (E)	6 (B)
Round 4	0	3 (C)	2 (A)	1 (E)	6 (B)

Order of edges selected to relax : (A,B), (A,C), (B,C),(B,D),(B,E)(C,B), (C,D),(C,E),(E,D)

vertex within ( ) is the prev vertex

Shortest-paths tree:



3. Shortest path problems

`Bellman-Ford` algorithm correctly detects negative cycles.

- Assume shortest path $p : s \rightsquigarrow v$.
- If after the $n-1$ rounds of relaxations are done, $\exists(u,v)$ with $dist(u) + l(u,v) < dist(v)$
- then path $p : s \rightsquigarrow v$ consists of more than $n-1$ edges; i.e., p contains at least $n+1$ vertices, some vertex x occurs twice.

$$p : s \rightsquigarrow y \rightarrow x \rightsquigarrow x \rightarrow z \rightsquigarrow v$$

- the path cannot be shorter than $s \rightsquigarrow y \rightarrow x \rightarrow z \rightsquigarrow v$ **unless** cycle $x \rightsquigarrow x$ is negative.

3. Shortest path problems

Priority queue implementation using heap

what is a heap? physical implementation?
how to use a heap to realize a priority queue?

Time complexities of different shortest path algorithms

- `dag-shortest-path`: $O(|V| + |E|)$
- `Dijkstra's`: $O(|V| \log_2 |V| + E)$
- `Bellman-Ford`: $O(|V||E|)$.