Lecture Note (Part 3)

CSCI 4470/6470 Algorithms, Fall 2023

Liming Cai
Department of Computer Science, UGA

October 3, 2023

Part 3. Algorithms on graphs (Chapters 3 and 4)

Topics to be discussed:

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

1. Fundamentals of graphs

Terminologies:

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

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

Computer representations of graphs:

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

Computer representations of graphs:

- adjacency list

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

Computer representations of graphs:

- adjacency list
- adjacency matrix

Recursive definition for trees

Recursive definition for trees

- set pair $(\{x\}, \emptyset)$ is a **tree**;

Recursive definition for trees

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

Recursive definition for trees

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

Recursive definition for trees

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

Recursive definition for trees

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

Recursive definition for graphs

- set pair $(\{x\}, \emptyset)$ is a **graph**;

1. Fundamentals of graphs

Recursive definition for graphs

- set pair $(\{x\}, \emptyset)$ is a **graph**;
- if (V, E) is a **graph**, $U \subseteq V$, and $v \notin V$,

Recursive definition for graphs

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

1. Fundamentals of graphs

Recursive definition for graphs

- set pair $(\{x\}, \emptyset)$ is a **graph**;
- if (V, E) is a **graph**, $U \subseteq V$, and $v \notin V$,
  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

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 $E' \backslash E$).

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.

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 $(v, u) \in E$.

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,

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) explore(G, y);
```

2. Depth-first search

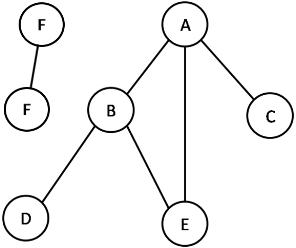
Assume graph G that was created with x being any vertex
Explore all vertices reachable from vertex x:

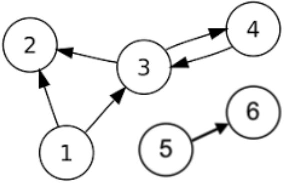
```
function explore(G: graph; x: vertex)
visited(x) = true;
for each edge (x, y) in G
  if not visited(y) explore(G, y);
```

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;
```

Examples for DFS





2. Depth-first search

2. Depth-first search

- DFS on a graph yields a **DF-search tree**:

2. Depth-first search

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

2. Depth-first search

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

  brackets patterns (look familiar?)

2. Depth-first search

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

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

  - tree edges
  - back edges

2. Depth-first search

DFS on directed graphs

DFS on directed graphs

- types of edges in DFS tree
  - tree edges
  - back edges
  - forward edges
  - cross 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 post$(x) >$ post(y)

2. Depth-first search

Applications of DFS

2. Depth-first search

Applications of DFS

- determine if the input graph is connected;

2. Depth-first search

Applications of DFS

- determine if the input graph is connected;
- determine if two given vertices are connected on input graph;

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;

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;

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;

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;

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;

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: $O(|E| + |V|)$

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
$\forall i < j, (v_j, v_i) \notin E$.

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 $\forall i < j, (v_j, v_i) \notin E$.

- DFS can be used to solve this problem. how?

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 $\forall i < j, (v_j, v_i) \notin E$.

- DFS can be used to solve this problem. how?

  examine the post time stamps

Strongly Connected Components (SCC) problem

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

Idea:

2. Depth-first search

Strongly Connected Components (SCC) problem

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

Idea:

- DFS on G;

Strongly Connected Components (SCC) problem

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

Idea:

- DFS on G;
- generated G^T, transpose of G (reversed edges directions);

Strongly Connected Components (SCC) problem

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

Idea:

- 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;

More about graph traversal algorithms

2. Depth-first search

More about graph traversal algorithms

- non-recursive version of DFS; using (?)

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

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 (?)

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 ?

More about graph traversal algorithms

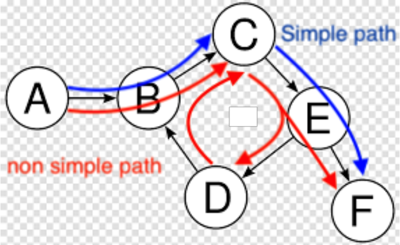
- non-recursive version of DFS; using (?)
- breadth first search (BFS); explores vertices in the order of their distance from the source vertex
- non-recursive BFS? using (?)
- recursive version ?
- time complexity

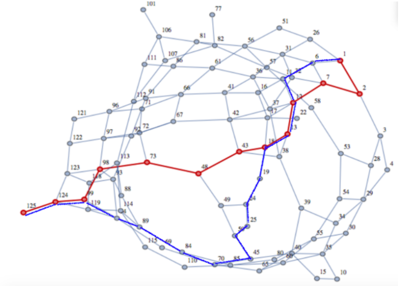
3. Shortest path problems

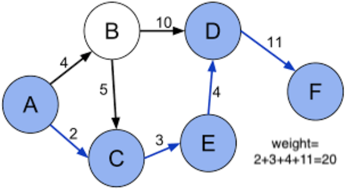
Paths on graphs:

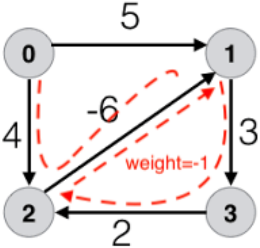
3. Shortest path problems

Paths on graphs:









3. Shortest path problems

Problems about 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;

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.

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.
- s-t **shortest path**: given $G = (V, E)$, vertices $s, t \in V$; find a shortest path $s \rightsquigarrow t$;

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.
- 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 $s \rightsquigarrow v$;

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.
- 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 $u \rightsquigarrow v$;

Shortest Distance problem:

3. Shortest path problems

Shortest Distance problem:

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

3. Shortest path problems

Shortest Distance problem:

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

- G is a DAG:

3. Shortest path problems

Shortest Distance problem:

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

- G is a DAG: DFS-based algorithm;

3. Shortest path problems

Shortest Distance problem:

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

- G is a DAG: DFS-based algorithm;
- G does not contain negative edges:

3. Shortest path problems

Shortest Distance problem:

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

- G is a DAG: DFS-based algorithm;
- G does not contain negative edges: Dijkstra's algorithm

Shortest Distance problem:

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

- G is a DAG: DFS-based algorithm;
- G does not contain negative edges: Dijkstra's algorithm
- G may contain negative cycles;

3. Shortest path problems

Shortest Distance problem:

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

- G is a DAG: DFS-based algorithm;
- G does not contain negative edges: Dijkstra's algorithm
- G may contain negative cycles; Bellman-Ford algorithm

3. Shortest path problems

Shortest Distance problem on DAG

3. Shortest path problems

Shortest Distance problem on DAG

- what does DFS tree on DAG look like?

Shortest Distance problem on DAG

- what does DFS tree on DAG look like?
- are there back, forward, and crossing edges?

3. Shortest path problems

Shortest Distance problem on DAG

- what does DFS tree on DAG look like?
- are there back, forward, and crossing edges?
- Solving the shortest distance problem
  with DFS, i.e, Topological Sort

3. Shortest path problems

Shortest Distance problem on DAG

- what does DFS tree on DAG look like?
- are there back, forward, and crossing edges?
- Solving the shortest distance problem
  with DFS, i.e, Topological Sort
- So vertices can be arranged linearly, beginning from source s
  how to calculate all the shortest distances from s?

3. Shortest path problems

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

3. Shortest path problems

Input: DAG $G = (V, E)$, edge lengths $l : E \to R_{\geq 0}$, 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: `Dijkstra's algorithm`

on general directed graphs, without negative weights

3. Shortest path problems

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

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

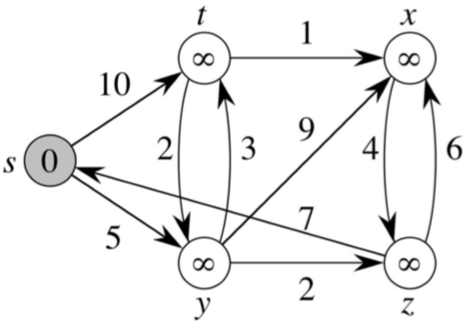
3. Shortest path problems

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

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

```
function Dijkstra(G, l, s)
for all u in V
  dist(u) = infinity;
  prev(u) = nil;
dist(s) = 0;
H = makequeue(V);
While H is not empty
u = dequeue(H);
for all edges (u, v) in E
    if dist(v) > dist(u) + l(u, v);
       dist(v) = dist(u) + l(u, v);
       prev(v) = u;
return (prev)
```

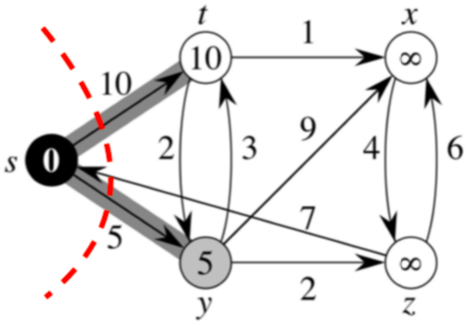
3. Shortest path problems



(a)

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

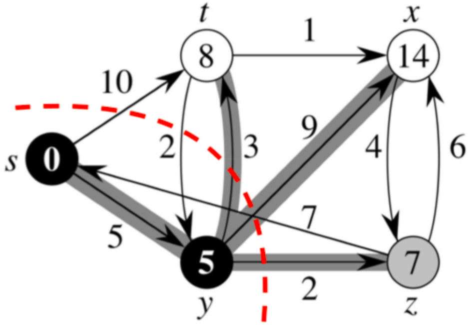
3. Shortest path problems



(b)

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

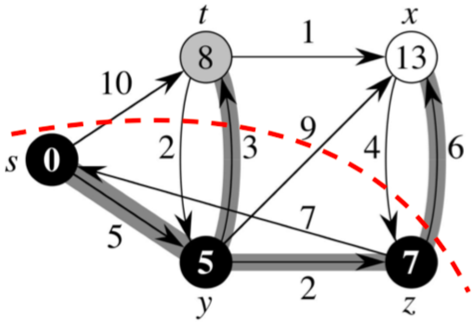
3. Shortest path problems



(c)

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

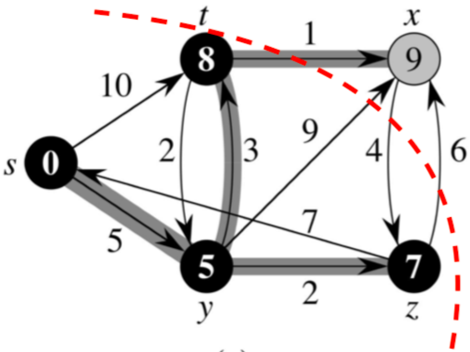
3. Shortest path problems



(d)

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

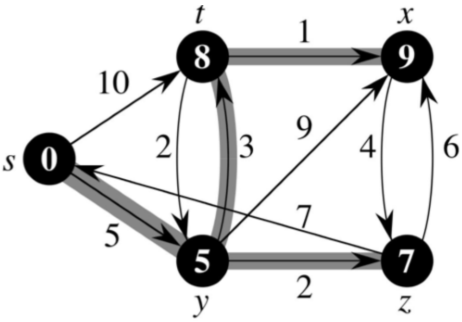
3. Shortest path problems



(e)

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

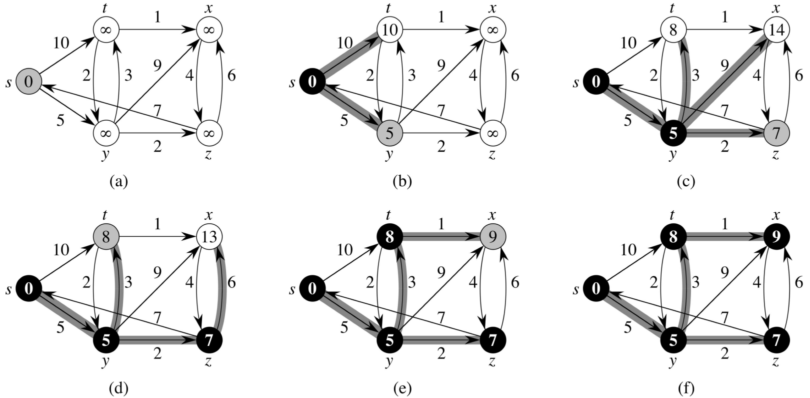
3. Shortest path problems



(f)

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

3. Shortest path problems



Note: while the black-colored vertices are in set S.

3. Shortest path problems

`Dijkstra's algorithm`

3. Shortest path problems

`Dijkstra's algorithm`

- more examples (textbook), keep track the priority queue

3. Shortest path problems

`Dijkstra's algorithm`

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

Dijkstra's algorithm

- more examples (textbook), keep track the priority queue
- what would happen if negative edges are present?
- edge relaxation on edge (u, v):

```
if dist(v) > dist(u) + l(u, v)
    dist(v) = dist(u) + l(u, v)
    prev(v) = u
```

3. Shortest path problems

Shortest path (distance) problem is solved by repeatedly relaxing edges coming out from selected vertices.

3. Shortest path problems

Shortest path (distance) problem is solved by repeatedly relaxing edges coming out from selected vertices.

- on DAGs, vertices can be chosen in a linear order;

3. Shortest path problems

Shortest path (distance) problem is solved by repeatedly relaxing edges coming out from selected vertices.

- on DAGs, vertices can be chosen in a linear order;
- on graphs without negative edges (`Dijkstra's`), what order?

Shortest path (distance) problem is solved by repeatedly relaxing edges coming out from selected vertices.

- on DAGs, vertices can be chosen in a linear order;
- on graphs without negative edges (`Dijkstra's`), what order?
- on arbitrary graphs, what order? (`Bellman-Ford`)

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;
if there is an edge (u, v):  dist(u) + l(u,v) < dist(v)
  then return "G contains a negative cycle"
  else return (dist, prev)
```

3. Shortest path problems

`Bellman-Ford` algorithm correctly detectes negative cycles.

3. Shortest path problems

`Bellman-Ford` algorithm correctly detectes negative cycles.

- Assume shortest path $p : s \rightsquigarrow v$.

3. Shortest path problems

`Bellman-Ford` algorithm correctly detectes negative cycles.

- Assume shortest path $p : s \rightsquigarrow v$.
- If after the $n-1$ rounds of relaxations are done, $\exists (u, v)$ with $dist(u) + l(u, v) < dist(v)$

`Bellman-Ford` algorithm correctly detectes 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.

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

`Bellman-Ford` algorithm correctly detectes 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?

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

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`: $O(|V| + |E|)$

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`: $O(|V| \log_2 |V| + E)$

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`: $O(|V||E|)$.