Part VI. Graph Algorithms
Part VI. Graph Algorithms

- Chapter 22 Elementary graph algorithms
- Chapter 23. Minimum spanning trees
- Chapter 24. Single-source shortest paths
- Chapter 25. All-pairs shortest paths
Chapter 22. Elementary graph algorithms
Chapter 22. Elementary graph algorithms

- Representations of graphs
Chapter 22. Elementary graph algorithms

- Representations of graphs
- Traverse graphs:
  - breadth-first-search (BFS)
  - depth-first-search (DFS)
- Applications:
  - topological sort
  - strongly connected components
Chapter 22. Elementary graph algorithms

- Representations of graphs
- Traverse graphs:
  1. breadth-first-search (BFS);
  2. depth-first-search (DFS)
Chapter 22. Elementary graph algorithms

- Representations of graphs
- Traverse graphs:
  1. breadth-first-search (BFS);
  2. depth-first-search (DFS)
- Applications:
Chapter 22. Elementary graph algorithms

Chapter 22. Elementary graph algorithms

- Representations of graphs
- Traverse graphs:
  1. breadth-first-search (BFS);
  2. depth-first-search (DFS)
- Applications:
  1. topological sort
Chapter 22. Elementary graph algorithms

• Representations of graphs

• traverse graphs:
  (1) breadth-first-search (BFS);
  (2) depth-first-search (DFS)

• applications:
  (1) topological sort
  (2) strongly connected components
Chapter 22. Elementary graph algorithms

Graph: $G = (V, E)$
Terminologies and notations:

• graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$.

• weight $w$: $E \rightarrow \mathbb{R}$, e.g., $w((1, 2)) = 4$, $w((5, 6)) = 3$, etc.

• degree: $\deg(v) =$ the number of edges incident on $v$, e.g., $\deg(3) = 4$, $\deg(7) = 2$.

• path: there is a path $a \xrightarrow{} b$, if $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$ and $v_1 = a$ and $v_k = b$. The path is a simple path if $v_1, \ldots, v_k$ are all different.

• cycle: when $v_1 = v_k$.

• It is a self-loop, if when $k = 1$ and $(v_1, v_k) \in E$. 

Chapter 22. Elementary graph algorithms

Terminologies and notations:

- **graph** $G = (V,E)$, where $V = \{v_1,\ldots,v_n\}$ and $E \subseteq V \times V$
- **weight** $w: E \rightarrow \mathbb{R}$, e.g., $w(1,2) = 4$, $w(5,6) = 3$, etc.
- **degree** $\deg(v)$ is the number of edges incident on $v$, e.g., $\deg(3) = 4$, $\deg(7) = 2$. The path $a \xrightarrow{} b$, if $(v_1,v_2),\ldots,(v_{k-1},v_k) \in E$ and $v_1 = a$ and $v_k = b$. The path is a simple path if $v_1,\ldots,v_k$ are all different.
- **cycle**: when $v_1 = v_k$. It is a self-loop, if when $k = 1$ and $(v_1,v_k) \in E$. 

![Graph diagram](image)
Chapter 22. Elementary graph algorithms

Terminologies and notations:

- graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$
Chapter 22. Elementary graph algorithms

Terminologies and notations:

- **graph** $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
Terminologies and notations:

- **graph** $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
  
  $V = \{1, 2, 3, 4, 5, 6, 7\}$
  
  $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}$
Chapter 22. Elementary graph algorithms

Terminologies and notations:

- **graph** $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
  
  $V = \{1, 2, 3, 4, 5, 6, 7\}$
  
  $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}$

- **weight** $w : E \rightarrow R$, 

The diagram above visualizes a graph with labeled nodes and edges.
Terminologies and notations:

- **graph** $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
  
  - $V = \{1, 2, 3, 4, 5, 6, 7\}$
  
  - $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}$

- **weight** $w : E \rightarrow R$, e.g., $w(1, 2) = 4$, $w(5, 6) = 3$, etc.
**Chapter 22. Elementary graph algorithms**

Terminologies and notations:

- **Graph** $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
  
  $V = \{1, 2, 3, 4, 5, 6, 7\}$
  
  $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}$

- **Weight** $w : E \rightarrow \mathbb{R}$, e.g., $w(1, 2) = 4$, $w(5, 6) = 3$, etc.

- **Degree**: $\text{deg}(v) = \text{the number of edges incident on } v$, e.g., $\text{deg}(3) = 4$, $\text{deg}(7) = 2$
Chapter 22. Elementary graph algorithms

Terminologies and notations:

- **graph** $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
  
  $V = \{1, 2, 3, 4, 5, 6, 7\}$
  
  $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}$

- **weight** $w : E \rightarrow \mathbb{R}$, e.g., $w(1, 2) = 4$, $w(5, 6) = 3$, etc.

- **degree**: $\text{deg}(v)$ = the number of edges incident on $v$, e.g., $\text{deg}(3) = 4$, $\text{deg}(7) = 2$

- **path**: there is a path $a \rightsquigarrow b$, if $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$
Terminologies and notations:

- **graph** $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
  
  $V = \{1, 2, 3, 4, 5, 6, 7\}$
  
  $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}$

- **weight** $w : E \rightarrow R$, e.g., $w(1, 2) = 4$, $w(5, 6) = 3$, etc.

- **degree**: $\text{deg}(v) =$ the number of edges incident on $v$, e.g., $\text{deg}(3) = 4$, $\text{deg}(7) = 2$

- **path**: there is a path $a \rightsquigarrow b$, if $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$
  and $v_1 = a$ and $v_k = b$. 
Terminologies and notations:

- **graph** \( G = (V, E) \), where \( V = \{v_1, \ldots, v_n\} \) and \( E \subseteq V \times V \)
  
  \( \begin{align*}
  V &= \{1, 2, 3, 4, 5, 6, 7\} \\
  E &= \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}
  \end{align*} \)

- **weight** \( w : E \rightarrow \mathbb{R} \), e.g., \( w(1, 2) = 4 \), \( w(5, 6) = 3 \), etc.

- **degree**: \( \text{deg}(v) \) = the number of edges incident on \( v \), e.g., \( \text{deg}(3) = 4 \), \( \text{deg}(7) = 2 \)

- **path**: there is a path \( a \leadsto b \), if \( (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E \)
  
  and \( v_1 = a \) and \( v_k = b \). The path is a **simple path** if \( v_1, \ldots v_k \) are all different.
Terminologies and notations:

- **graph** $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
  
  $V = \{1, 2, 3, 4, 5, 6, 7\}$
  
  $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}$

- **weight** $w : E \to \mathbb{R}$, e.g., $w(1, 2) = 4$, $w(5, 6) = 3$, etc.

- **degree**: $\text{deg}(v) = \text{the number of edges incident on } v$, e.g., $\text{deg}(3) = 4$, $\text{deg}(7) = 2$

- **path**: there is a path $a \rightsquigarrow b$, if $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$
  and $v_1 = a$ and $v_k = b$. The path is a **simple path** if $v_1, \ldots v_k$ are all different.

- **cycle**: when $v_1 = v_k$. 
Chapter 22. Elementary graph algorithms

Terminologies and notations:

• graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E \subseteq V \times V$
  $V = \{1, 2, 3, 4, 5, 6, 7\}$
  $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7)\}$

• weight $w: E \rightarrow R$, e.g., $w(1, 2) = 4$, $w(5, 6) = 3$, etc.

• degree: $deg(v) =$ the number of edges incident on $v$, e.g., $deg(3) = 4$, $deg(7) = 2$

• path: there is a path $a \rightsquigarrow b$, if $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$
  and $v_1 = a$ and $v_k = b$. The path is a simple path if $v_1, \ldots, v_k$ are all different.

• cycle: when $v_1 = v_k$.
  It is a self-loop, if when $k = 1$ and $(v_1, v_k) \in E$. 
Chapter 22. Elementary graph algorithms

- **digraphs**: directed graphs

- **complete graphs**: $K_n$, e.g., $K_6$

- **bipartite graphs**: $G = (V_1 \cup V_2, E)$, $V_1 \cap V_2 = \emptyset$, $K_3, 3$

- **planar graphs**: embedded in the plane without crossing edges: However, $K_5$ is not planar, neither is $K_{3,3}$.
Chapter 22. Elementary graph algorithms

- **digraphs**: directed graphs
Chapter 22. Elementary graph algorithms

- **digraphs**: directed graphs

- **complete graphs**: $K_n$, $K_5$ is not planar, neither is $K_3,3$.
Chapter 22. Elementary graph algorithms

- **digraphs**: directed graphs

- **complete graphs**: $K_n$, e.g., $K_6$
Chapter 22. Elementary graph algorithms

• **digraphs**: directed graphs

• **complete graphs**: $K_n$, e.g., $K_6$

• **bipartite graphs**: $G = (V_1 \cup V_2, E), V_1 \cap V_2 = \emptyset$, 
Chapter 22. Elementary graph algorithms

- **digraphs**: directed graphs

- **complete graphs**: $K_n$, e.g., $K_6$

- **bipartite graphs**: $G = (V_1 \cup V_2, E), \ V_1 \cap V_2 = \emptyset, K_{3,3}$,


Chapter 22. Elementary graph algorithms

- **digraphs**: directed graphs

- **complete graphs**: $K_n$, e.g., $K_6$

- **bipartite graphs**: $G = (V_1 \cup V_2, E)$, $V_1 \cap V_2 = \emptyset$, $K_{3,3}$

- **planar graphs**: embedded in the plane without crossing edges:
Chapter 22. Elementary graph algorithms

- **digraphs**: directed graphs

- **complete graphs**: $K_n$, e.g., $K_6$

- **bipartite graphs**: $G = (V_1 \cup V_2, E)$, $V_1 \cap V_2 = \emptyset$, $K_{3,3}$

- **planar graphs**: embedded in the plane without crossing edges:

  However, $K_5$ is not planar, neither is $K_{3,3}$
Chapter 22. Elementary graph algorithms

- **trees**: graphs that do not contain cycles; e.g.,
Chapter 22. Elementary graph algorithms

- **trees**: graphs that do not contain cycles; e.g.,
- **$k$-trees**: 

![Diagram of a graph with nodes 1 to 6 connected in a tree-like structure]
Chapter 22. Elementary graph algorithms

- **trees**: graphs that do not contain cycles; e.g.,
- **k-trees**:
  
  1-tree is tree;
Chapter 22. Elementary graph algorithms

- **trees**: graphs that do not contain cycles; e.g.,
- **$k$-trees**:
  
  1-tree is a tree;

  2-tree is a graph but with **tree width** = 2
Chapter 22. Elementary graph algorithms

Representations of graphs

adjacency-matrix
adjacency-list
Chapter 22. Elementary graph algorithms

Representations of graphs

adjacency-matrix
adjacency-list

(a)

(b)

(c)
Chapter 22. Elementary graph algorithms

adjacency-matrix for a weighted graph
Chapter 22. Elementary graph algorithms

Traverse graphs

basic ideas of depth-first-search (DFS) and breadth-first-search (BFS)

Both methods yield "search trees"
    or "search forest" (if the graph is not connected)
Chapter 22. Elementary graph algorithms

DFS on directed graphs, search tree
Chapter 22. Elementary graph algorithms

DFS on directed graphs, search tree
Chapter 22. Elementary graph algorithms

DFS on directed graphs, search tree

DFS on non-directed graphs, search tree
Chapter 22. Elementary graph algorithms

DFS on directed graphs, search tree

DFS on non-directed graphs, search tree
Chapter 22. Elementary graph algorithms

DFS on directed graphs, search tree

DFS on non-directed graphs, search tree
Chapter 22. Elementary graph algorithms

Traversal on graphs is an important task:
Traversal on graphs is an important task:

- navigating the whole graph;
Chapter 22. Elementary graph algorithms

Traversal on graphs is an important task:

- navigating the whole graph;
- for connectivity check;
Traversals on graphs is an important task:
- navigating the whole graph;
- for connectivity check;
- for circle check;
- etc.
Traversal on graphs is an important task:

• navigating the whole graph;
• for connectivity check;
• for circle check;
• etc

DFS and BFS are two fundamental algorithms for graph traversal!
Chapter 22. Elementary graph algorithms

First recursive DFS algorithm, assuming $G$ is connected.

How does the algorithm start?

- Initially set $u.\text{visit} = \text{false}$ for every vertex $u \in G.V$;
- $s.\pi = \text{NULL}$ for some specific $s \in G.V$;
- Call $\text{Recursive-DFS}(G,s)$.

But if $G$ is not connected, what should we do?
Chapter 22. Elementary graph algorithms

First recursive DFS algorithm, assuming $G$ is connected.

```
RECURSIVE-DFS(G, u);
```

How does the algorithm start?

- Initially set $u.visit = false$ for every vertex $u \in G.V$;
- $s.\pi = NULL$ for some specific $s \in G.V$;
- Call `RECURSIVE-DFS(G, s)`.

But if $G$ is not connected, what should we do?
First recursive DFS algorithm, assuming $G$ is connected.

\textbf{Recursive-DFS}(G, u);
1. \textbf{if not} $u.visit$

But if $G$ is not connected, what should we do?
Chapter 22. Elementary graph algorithms

First recursive DFS algorithm, assuming $G$ is connected.

**Recursive-DFS** ($G, u$);

1. if not $u.visit$
2. $u.visit = true$; \{ mark $u$ ”visited” \}
3. for each $v \in \text{Adj}[u]$ and not $v.visit$;
4. $v.\pi = u$; \{ set $v$’s parent to be $u$ \}
5. Recursive-DFS ($G, v$);
6. return ( );

How does the algorithm start?

- initially set $u.visit = false$ for every vertex $u \in G.V$;
- $s.\pi = NULL$ for some specific $s \in G.V$;
- call **Recursive-DFS** ($G, s$).

But if $G$ is not connected, what should we do?
First recursive DFS algorithm, assuming $G$ is connected.

Recursive-DFS($G, u$);
1. if not $u.visit$
2. $u.visit = true$; \{ mark $u$ "visited" \}
3. for each $v \in Adj[u]$ and not $v.visit$; \{ $u$'s unvisited neighbors \}
4. $v.\pi = u$; \{ set $v$'s parent to be $u$ \}
5. Recursive-DFS($G, v$);
6. return ($\cdot$)

How does the algorithm start?
• initially set $u.visit = false$ for every vertex $u \in G.V$;
• $s.\pi = NULL$ for some specific $s \in G.V$;
• call Recursive-DFS($G, s$).

But if $G$ is not connected, what should we do?
First recursive DFS algorithm, assuming $G$ is connected.

**Recursive-DFS** $(G, u);$
1. if not $u.visit$
2. $u.visit = true;$ \{ mark $u$ ”visited” \}
3. for each $v \in \text{Adj}[u]$ and not $v.visit;$ \{ $u$’s unvisited neighbors \}
4. $v.\pi = u;$ \{ set $v$’s parent to be $u$ \}

How does the algorithm start?
• initially set $u.visit = false$ for every vertex $u \in G.V$;
• $s.\pi = NULL$ for some specific $s \in G.V$;
• call Recursive-DFS $(G, s)$.

But if $G$ is not connected, what should we do?
First recursive DFS algorithm, assuming $G$ is connected.

**Recursive-DFS**($G, u$);
1.  if not $u.visit$
2.    $u.visit = true; \{ \text{mark } u \text{ "visited" } \}$
3.    for each $v \in \text{Adj}[u]$ and not $v.visit$; \{ $u$’s unvisited neighbors \}
4.      $v.\pi = u; \{ \text{set } v\text{’s parent to be } u \}$
5.    **Recursive-DFS**($G, v$);
Chapter 22. Elementary graph algorithms

First recursive DFS algorithm, assuming $G$ is connected.

**Recursive-DFS**($G, u$);
1. if not $u.visit$
2. $u.visit = true$; \{ mark $u$ "visited" \}
3. for each $v \in Adj[u]$ and not $v.visit$; \{ $u$'s unvisited neighbors \}
4. $v.\pi = u$; \{ set $v$'s parent to be $u$ \}
5. **Recursive-DFS**($G, v$);
6. return ();
First recursive DFS algorithm, assuming $G$ is connected.

**Recursive-DFS** $(G, u)$;

1. if not $u.visit$
2. $u.visit = \text{true};$ \hfill \{ mark $u$ "visited" \}
3. for each $v \in \text{Adj}[u]$ and not $v.visit;$ \hfill \{ $u$’s unvisited neighbors \}
4. \hspace{1em} $v.\pi = u;$ \hfill \{ set $v$’s parent to be $u$ \}
5. \hspace{1em} **Recursive-DFS** $(G, v);$ 
6. return ( );

How does the algorithm start?
First recursive DFS algorithm, assuming $G$ is connected.

\textbf{Recursive-DFS}(G, u);
1. \textbf{if} not \textit{u.visit} \textbf{then}
2. \hspace{1em} \textit{u.visit} = true; \hspace{1em} \{ mark \textit{u} ”visited” \}
3. \textbf{for} each \textit{v} $\in$ Adj[\textit{u}] and not \textit{v.visit} \textbf{do}
4. \hspace{1em} \textit{v.\pi} = u; \hspace{1em} \{ \textit{u’s unvisited neighbors} \}
5. \hspace{1em} \textbf{Recursive-DFS}(G, v); \hspace{1em} \{ \textit{set v’s parent to be \textit{u}} \}
6. \textbf{return} ( );

How does the algorithm start?

- initially set \textit{u.visit} = false for every vertex \textit{u} $\in$ \textit{G.V};
Chapter 22. Elementary graph algorithms

First recursive DFS algorithm, assuming $G$ is connected.

**Recursive-DFS**$(G, u);$  
1. if not $u.\text{visit}$  
2. \hspace{1cm} $u.\text{visit} = \text{true}$; \{ mark $u$ "visited" \}  
3. \hspace{1cm} for each $v \in \text{Adj}[u]$ and not $v.\text{visit}$; \{ $u$‘s unvisited neighbors \}  
4. \hspace{1cm} $v.\pi = u$; \{ set $v$‘s parent to be $u$ \}  
5. \hspace{1cm} **Recursive-DFS**$(G, v);$  
6. return ( );

How does the algorithm start?  

- initially set $u.\text{visit} = \text{false}$ for every vertex $u \in G.V$;  
- $s.\pi = NULL$ for some specific $s \in G.V$;
First recursive DFS algorithm, assuming $G$ is connected.

**Recursive-DFS** ($G, u$);

1. if not $u.visit$
2. $u.visit = true$; { mark $u$ ”visited” }
3. for each $v \in Adj[u]$ and not $v.visit$; { $u$’s unvisited neighbors }
4. $v.\pi = u$; { set $v$’s parent to be $u$ }
5. Recursive-DFS($G, v$);
6. return ( );

How does the algorithm start?

- initially set $u.visit = false$ for every vertex $u \in G.V$;
- $s.\pi = NULL$ for some specific $s \in G.V$;
- call **Recursive-DFS**($G, s$).
Chapter 22. Elementary graph algorithms

First recursive DFS algorithm, assuming $G$ is connected.

**Recursive-DFS**($G, u$);
1. **if** not $u$.visit
2. $u$.visit = **true**; \{ mark $u$ ”visited” \}
3. **for** each $v \in Adj[u]$ and not $v$.visit; \{ $u$’s unvisited neighbors \}
4. $v.\pi = u$; \{ set $v$’s parent to be $u$ \}
5. **Recursive-DFS**($G, v$);
6. **return** ( );

How does the algorithm start?

- initially set $u$.visit = **false** for every vertex $u \in G.V$;
- $s.\pi = NULL$ for some specific $s \in G.V$;
- call **Recursive-DFS**($G, s$).

But if $G$ is not connected, what should we do?
Chapter 22. Elementary graph algorithms

**To-Start-DFS**($G$)

1. for each $s \in G.V$
   
   initialize visit values

2. $s.visit = false$;

3. $s.\pi = NULL$;

4. for each $s \in G.V$ and not $s.visit$

5. Recursive-DFS($G,s$)

**Recursive-DFS**($G,u$)

1. if not $u.visit$

2. $u.visit = true$;

   mark $u$ "visited"

3. for each $v \in Adj[u]$ and not $v.visit$

4. $v.\pi = u$;

   set $v$'s parent to be $u$

5. Recursive-DFS($G,v$);

6. return $()$
Chapter 22. Elementary graph algorithms

\textbf{To-Start-DFS}(G)
1. \textbf{for} each \( s \in G.V \) \hspace{1cm} \{ initialize visit values \}
2. \( s.\text{visit} = \text{false}; \)
Chapter 22. Elementary graph algorithms

**To-Start-DFS** \((G)\)

1. **for** each \(s \in G.V\) \{ initialize \(visit\) values \}
2. \(s.visit = false;\)
3. \(s.\pi = NULL;\)
Chapter 22. Elementary graph algorithms

**TO-START-DFS**\( (G) \)

1. \textbf{for each} \( s \in G.V \) ~ \{ \text{initialize visit values} \}
2. \( s.visit = \text{false}; \)
3. \( s.\pi = \text{NULL}; \)
4. \textbf{for each} \( s \in G.V \) \textbf{and} \textbf{not} \( s.visit \)
5. \textbf{RECURSIVE-DFS}\( (G, s) \)
Chapter 22. Elementary graph algorithms

**To-Start-DFS**($G$)
1. for each $s \in G.V$ \{ initialize visit values \}
2. $s.visit = \text{false}$;
3. $s.\pi = \text{NULL}$;
4. for each $s \in G.V$ and not $s.visit$
5. **Recursive-DFS**($G, s$)

**Recursive-DFS**($G, u$);
1. if not $u.visit$
2. $u.visit = \text{true}$; \{ mark u ”visited” \}
3. for each $v \in \text{Adj}[u]$ and not $v.visit$; \{ u’s unvisited neighbors \}
4. $v.\pi = u$; \{ set v’s parent to be u \}
5. **Recursive-DFS**($G, v$);
6. return ( );
Chapter 22. Elementary graph algorithms

**To-Start-DFS**\((G)\)

1. for each \(s \in G.V\) \{ initialize \(visit\) values \}
2. \(s.visit = \text{false}\);
3. \(s.\pi = NULL\);
4. for each \(s \in G.V\) and \(\text{not} \ s.visit\)
5. \(\text{Recursive-DFS}(G, s)\)

**Recursive-DFS**\((G, u)\);

1. if \(\text{not} \ u.visit\)
2. \(u.visit = \text{true}\); \{ mark \(u\) ”visited” \}
3. for each \(v \in Adj[u]\) and \(\text{not} \ v.visit\); \{ u’s unvisited neighbors \}
4. \(v.\pi = u\); \{ set \(v\)’s parent to be \(u\) \}
5. \(\text{Recursive-DFS}(G, v)\);
6. return ( );
Chapter 22. Elementary graph algorithms

DFS (from the textbook) computes discover and finish time stamps \((u.d \text{ and } u.f)\) for every visited vertex \(u\).

DFS\((G)\)
1. for each vertex \(u \in G.V\)
2. \(u.color = \text{WHITE}\)
3. \(u.\pi = \text{NIL}\)
4. \(time = 0\)
5. for each vertex \(u \in G.V\)
6. if \(u.color == \text{WHITE}\)
7. DFS-\text{VISIT}(G, u)

DFS-\text{VISIT}(G, u)
1. \(time = time + 1\) // white vertex \(u\) has just been discovered
2. \(u.d = time\)
3. \(u.color = \text{GRAY}\)
4. for each \(v \in G.Adj[u]\) // explore edge \((u, v)\)
5. if \(v.color == \text{WHITE}\)
6. \(v.\pi = u\)
7. DFS-\text{VISIT}(G, v)
8. \(u.color = \text{BLACK}\) // blacken \(u\); it is finished
9. \(time = time + 1\)
10. \(u.f = time\)
Chapter 22. Elementary graph algorithms

DFS-Visit(G, u)
1  time = time + 1 // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u] // explore edge (u, v)
5       if v.color == WHITE
6         v.π = u
7         DFS-Visit(G, v)
8  u.color = BLACK // blacken u; it is finished
9  time = time + 1
10  u.f = time

→: edge being explored;
→: edge path taken by DFS
Chapter 22. Elementary graph algorithms

DFS-Visit(G, u)
1. time = time + 1  // white vertex u has just been discovered
2. u.d = time
3. u.color = GRAY
4. for each v ∈ G.Adj[u]  // explore edge (u, v)
5.  
6.    if v.color == WHITE  
7.      v.π = u
8.      DFS-Visit(G, v)
9.    u.color = BLACK  // blacken u; it is finished
10. time = time + 1
11. u.f = time

→: edge being explored;
→: edge path taken by DFS
Chapter 22. Elementary graph algorithms

DFS-Visit(G, u)
1 \( \text{time} = \text{time} + 1 \)  // white vertex \( u \) has just been discovered
2 \( u.d = \text{time} \)
3 \( u.color = \text{GRAY} \)
4 \( \text{for each } v \in G.Aadj[u] \)  // explore edge \((u, v)\)
5 \( \quad \text{if } v.color == \text{WHITE} \)
6 \( \quad \quad v.\pi = u \)
7 \( \quad \quad \text{DFS-Visit}(G, v) \)
8 \( u.color = \text{BLACK} \)  // blacken \( u \); it is finished
9 \( \text{time} = \text{time} + 1 \)
10 \( u.f = \text{time} \)

\( \rightarrow \): edge being explored;
\( \rightarrow \): edge path taken by DFS
Chapter 22. Elementary graph algorithms

DFS-Visit(G, u)
1 \( time = time + 1 \)  \hspace{1cm} // white vertex \( u \) has just been discovered
2 \( u.d = time \)
3 \( u.color = \text{GRAY} \)
4 \textbf{for} each \( v \in G.\text{Adj}[u] \)
5 \hspace{1cm} \textbf{if} \ \( v.color == \text{WHITE} \)
6 \hspace{2cm} \( v.\pi = u \)
7 \hspace{2cm} DFS-Visit(G, v)
8 \( u.color = \text{BLACK} \)  \hspace{1cm} // blacken \( u \); it is finished
9 \( time = time + 1 \)
10 \( u.f = time \)

→: edge being explored;
→: edge path taken by DFS
Chapter 22. Elementary graph algorithms
Chapter 22. Elementary graph algorithms

Another example of DFS execution (page 605)
Chapter 22. Elementary graph algorithms

Time complexity of DFS algorithm

\[ \Theta(|E| + |V|) \]
**Time complexity of DFS algorithm**

\[ \Theta(|E| + |V|) \]

where \(|E|\) is the number of edges in \(G\).

---

**DFS**

1. for each vertex \(u \in G.V\)
2. \(u.color = \text{WHITE}\)
3. \(u.\pi = \text{NIL}\)
4. \(time = 0\)
5. for each vertex \(u \in G.V\)
6. if \(u.color == \text{WHITE}\)
7. \(\text{DFS-VISIT}(G, u)\)

---

**DFS-VISIT**

1. \(time = time + 1\) // white vertex \(u\) has just been discovered
2. \(u.d = time\)
3. \(u.color = \text{GRAY}\)
4. for each \(v \in G.Adj[u]\) // explore edge \((u, v)\)
5. if \(v.color == \text{WHITE}\)
6. \(v.\pi = u\)
7. \(\text{DFS-VISIT}(G, v)\)
8. \(u.color = \text{BLACK}\) // blacken \(u\); it is finished
9. \(time = time + 1\)
10. \(u.f = time\)
Chapter 22. Elementary graph algorithms

Time complexity of DFS algorithm

```
DFS(G)
1   for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4   time = 0
5   for each vertex u ∈ G.V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1   time = time + 1       // white vertex u has just been discovered
2   u.d = time
3   u.color = GRAY
4   for each v ∈ G.Adj[u]  // explore edge (u, v)
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK       // blacken u; it is finished
9   time = time + 1
10  u.f = time
```

\( \Theta(|E| + |V|) \), where \(|E|\) is the number of edges in \(G\).
Chapter 22. Elementary graph algorithms

Properties of depth-first-search:

(1) \( u = v.\pi \text{ iff } \text{DFS-Visit}(G,v) \text{ is called.} \)

(2) Theorem 22.7 (Parenthesis Theorem): for any \( u, v \), exactly one of the following three conditions holds:

- \([u.d, u.f]\) and \([v.d, v.f]\) are entirely disjoint, and neither \( u \) nor \( v \) is a descendant of the other in the search tree.
- \([u.d, u.f]\) is contained entirely within \([v.d, v.f]\) and \( u \) is a descendant of \( v \), or
- \([v.d, v.f]\) is contained entirely within \([u.d, u.f]\) and \( v \) is a descendant of \( u \).

Corollary 22.8 (Nesting of descendants' intervals) Vertex \( v \) is a proper descendant of \( u \) in the depth-first search forest if and only if \( u.d < v.d < v.f < u.f \).
Chapter 22. Elementary graph algorithms

Properties of depth-first-search:

(1) \( u = v.\pi \)
Properties of depth-first-search:

(1) \( u = v.\pi \) iff DFS-VISIT\((G, v)\) is called.
Properties of depth-first-search:

(1) \( u = v.\pi \) iff \( \text{DFS-Visit}(G, v) \) is called.

(2) Theorem 22.7 (Parenthesis Theorem): for any \( u, v \), exactly one of the following three conditions holds:

- \([u.d, u.f]\) and \([v.d, v.f]\) are entirely disjoint, and neither \( u \) nor \( v \) is a descendant of the other in the search tree.
- \([u.d, u.f]\) is contained entirely within \([v.d, v.f]\) and \( u \) is a descendant of \( v \), or
- \([v.d, v.f]\) is contained entirely within \([u.d, u.f]\) and \( v \) is a descendant of \( u \).
Chapter 22. Elementary graph algorithms

Properties of depth-first-search:

(1) \( u = v.\pi \) iff \( \text{DFS-Visit}(G, v) \) is called.

(2) **Theorem 22.7 (Parenthesis Theorem):** for any \( u, v \), exactly one of the following three conditions holds:

- \([u.d, u.f]\) and \([v.d, v.f]\) are entirely disjoint, and neither \( u \) nor \( v \) is a descendant of the other in the search tree.

**Corollary 22.8 (Nesting of descendants' intervals)** Vertex \( v \) is a proper descendant of \( u \) in the depth-first search forest if and only if \( u.d < v.d < v.f < u.f \).
Chapter 22. Elementary graph algorithms

Properties of depth-first-search:

(1) $u = v.\pi$ iff $\text{DFS-VISIT}(G, v)$ is called.

(2) **Theorem 22.7 (Parenthesis Theorem):** for any $u, v$, exactly one of the following three conditions holds:

- $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither $u$ nor $v$ is a descendant of the other in the search tree.

- $[u.d, u.f]$ is contained entirely within $[v.d, v.f]$ and $u$ is a descendant of $v$, or
Chapter 22. Elementary graph algorithms

Properties of depth-first-search:

(1) \( u = v.\pi \) iff \( \text{DFS-Visit}(G, v) \) is called.

(2) **Theorem 22.7 (Parenthesis Theorem):** for any \( u, v \), exactly one of the following three conditions holds:

- \([u.d, u.f]\) and \([v.d, v.f]\) are entirely disjoint, and neither \( u \) nor \( v \) is a descendant of the other in the search tree.

- \([u.d, u.f]\) is contained entirely within \([v.d, v.f]\) and \( u \) is a descendant of \( v \), or

- \([v.d, v.f]\) is contained entirely within \([u.d, u.f]\) and \( v \) is a descendant of \( u \).
Properties of depth-first-search:

(1) $u = v.\pi$ iff DFS-VISIT($G,v$) is called.

(2) **Theorem 22.7 (Parenthesis Theorem):** for any $u, v$, exactly one of the following three conditions holds:

- $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither $u$ nor $v$ is a descendant of the other in the search tree.

- $[u.d, u.f]$ is contained entirely within $[v.d, v.f]$ and $u$ is a descendant of $v$, or

- $[v.d, v.f]$ is contained entirely within $[u.d, u.f]$ and $v$ is a descendant of $u$.

**Corollary 22.8 (Nesting of descendants’ intervals)** Vertex $v$ is a proper descendant of $u$ in the depth-first search forest if and only if $u.d < v.d < v.f < u.f$. 

Chapter 22. Elementary graph algorithms
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

**Proof:** ⇒
Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$
- case 1: $u = v$, apparently the claim is true;
Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;
Chapter 22. Elementary graph algorithms

Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

**Proof:** $\Rightarrow$
- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$
- Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$. 
Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. 
Chapter 22. Elementary graph algorithms

Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$

1. Case 1: $u = v$, apparently the claim is true;
2. Case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$)
Chapter 22. Elementary graph algorithms

Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$). Because at time $u.d$, $x$ is of WHITE color, $u.d < x.d$. 
Chapter 22. Elementary graph algorithms

Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$

case 1: $u = v$, apparently the claim is true;

case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$)

Because at time $u.d$ $x$ is of WHITE color, $u.d < x.d$.

Because $(w, x)$ is an edge, there are two possible scenarios:

(1) when $(w, x)$ is being explored, $x$ has already been discovered;
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

**Proof:**

$\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$). Because at time $u.d$ $x$ is of WHITE color, $u.d < x.d$.

Because $(w, x)$ is an edge, there are two possible scenarios:

1. When $(w, x)$ is being explored, $x$ has already been discovered; we thus have $x.d < w.f$;
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

**Proof:** $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$)

Because at time $u.d$ $x$ is of WHITE color, $u.d < x.d$.

Because $(w, x)$ is an edge, there are two possible scenarios:

1. when $(w, x)$ is being explored, $x$ has already been discovered;
   - we thus have $x.d < w.f$;
2. when $(w, x)$ is being explored, $x$ has WHITE color but will then be discovered
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

**Proof:** $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$).

Because at time $u.d$, $x$ is of WHITE color, $u.d < x.d$.

Because $(w, x)$ is an edge, there are two possible scenarios:

1. when $(w, x)$ is being explored, $x$ has already been discovered; we thus have $x.d < w.f$;
2. when $(w, x)$ is being explored, $x$ has WHITE color but will then be discovered we thus also have $x.d < w.f$;
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

**Proof:** $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$)

Because at time $u.d$ $x$ is of WHITE color, $u.d < x.d$.

Because $(w, x)$ is an edge, there are two possible scenarios:

- (1) when $(w, x)$ is being explored, $x$ has already been discovered; we thus have $x.d < w.f$;
- (2) when $(w, x)$ is being explored, $x$ has WHITE color but will then be discovered we thus also have $x.d < w.f$;

According to Corollary 22.8, $u.d < x.d < w.f < u.f$. 
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

**Proof:** ⇒

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

⇐

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$)

Because at time $u.d x$ is of WHITE color, $u.d < x.d$.

Because $(w, x)$ is an edge, there are two possible scenarios:

1. when $(w, x)$ is being explored, $x$ has already been discovered;
   - we thus have $x.d < w.f$;
2. when $(w, x)$ is being explored, $x$ has WHITE color but will then be discovered we thus also have $x.d < w.f$;

According to Corollary 22.8, $u.d < x.d < w.f < u.f$. Thus, $u.d < x.d < u.f$
Chapter 22. Elementary graph algorithms

**Theorem 22.9 (White-path Theorem)** In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

**Proof:** $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$)

Because at time $u.d$ $x$ is of WHITE color, $u.d < x.d$.

Because $(w, x)$ is an edge, there are two possible scenarios:

1. when $(w, x)$ is being explored, $x$ has already been discovered;
   - we thus have $x.d < w.f$;
2. when $(w, x)$ is being explored, $x$ has WHITE color but will then be discovered
   - we thus also have $x.d < w.f$;

According to Corollary 22.8, $u.d < x.d < w.f < u.f$. Thus, $u.d < x.d < u.f$

By Theorem 22.7, the interval $[x.d, x.f]$ is entirely contained within interval $[u.d, u.f]$. 
Chapter 22. Elementary graph algorithms

Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$

- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$

Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$).

Because at time $u.d$ $x$ is of WHITE color, $u.d < x.d$.

Because $(w, x)$ is an edge, there are two possible scenarios:

- (1) when $(w, x)$ is being explored, $x$ has already been discovered; we thus have $x.d < w.f$;
- (2) when $(w, x)$ is being explored, $x$ has WHITE color but will then be discovered we thus also have $x.d < w.f$;

According to Corollary 22.8, $u.d < x.d < w.f < u.f$. Thus, $u.d < x.d < u.f$

By Theorem 22.7, the interval $[x.d, x.f]$ is entirely contained within interval $[u.d, u.f]$. Therefore, $x$ is a descendant of $u$. Contradicts to the earlier assumption.
Chapter 22. Elementary graph algorithms

Theorem 22.9 (White-path Theorem) In a depth-first search forest of graph $G$, vertex $v$ is a descendant of $u$ if and only if at the time $u.d$ that the search discovers $u$, there is a path from $u$ to $v$ consisting of entirely of white vertices.

Proof: $\Rightarrow$
- case 1: $u = v$, apparently the claim is true;
- case 2: $v$ is a proper descendant of $u$, use Corollary 22.8 on every vertex on the path from $u$ to $v$; the claim is true;

$\Leftarrow$
Assume that at the time $u.d$, there is a path from $u$ to $v$ as stated in the theorem but $v$ is not descendant of $u$.

Assume $(w, x)$ be an edge on the path and $x$ is the first vertex on the path which is not descendant of $u$. Note that $w$ is a descendant of $u$ (or just $u$)
Because at time $u.d$ $x$ is of WHITE color, $u.d < x.d$.
Because $(w, x)$ is an edge, there are two possible scenarios:
- (1) when $(w, x)$ is being explored, $x$ has already been discovered; we thus have $x.d < w.f$;
- (2) when $(w, x)$ is being explored, $x$ has WHITE color but will then be discovered we thus also have $x.d < w.f$;

According to Corollary 22.8, $u.d < x.d < w.f < u.f$. Thus, $u.d < x.d < u.f$
By Theorem 22.7, the interval $[x.d, x.f]$ is entirely contained within interval $[u.d, u.f]$. Therefore, $x$ is a descendant of $u$. Contradicts to the earlier assumption. $v$ should be a descendant of $u$. 
Chapter 22. Elementary graph algorithms

Classification of edges (for directed graphs)

- Tree edges: those in the search tree (forest);
- Back edges: those connecting a vertex to an ancestor; a self-loop, in a directed graph, can be a back edge;
- Forward edges: those connecting a vertex to a descendant;
- Cross edges: all other edges;
Classification of edges (for **directed graphs**)

- **Tree edges**: those in the search tree (forest);
- **Back edges**: those connecting a vertex to an ancestor; a selfloop, in a directed graph, can be a back edge;
- **Forward edges**: those connecting a vertex to a descendant;
- **Cross edges**: all other edges;

**DFS forest with back, forward, & cross edges.**
Chapter 22. Elementary graph algorithms

Classification of edges (for **directed graphs**)

- **Tree edges**: those in the search tree (forest);
- **Back edges**: those connecting a vertex to an ancestor; a selfloop, in a directed graph, can be a back edge;
- **Forward edges**: those connecting a vertex to a descendant;
- **Cross edges**: all other edges.

- **tree edges**: those in the search tree (forest);
Classification of edges (for directed graphs)

- **tree edges**: those in the search tree (forest);
  $(u, v)$ is a tree edge if $v$ was discovered by exploring $(u, v)$;

- **back edges**: those connecting a vertex to an ancestor;
  a selfloop, in a directed graph, can be a back edge;

- **forward edges**: those connecting a vertex to a descendant;

- **cross edges**: all other edges;
Classification of edges (for directed graphs)

- **tree edges**: those in the search tree (forest); 
  
  \((u, v)\) is a tree edge if \(v\) was discovered by exploring \((u, v)\);

- **back edges**: those connecting a vertex to an ancestor;

- **forward edges**: those connecting a vertex to a descendant;

- **cross edges**: all other edges;

---

**DFS forest with back, forward, & cross edges.**
Classification of edges (for directed graphs)

- **tree edges**: those in the search tree (forest); 
  \((u, v)\) is a tree edge if \(v\) was discovered by exploring \((u, v)\);
- **back edges**: those connecting a vertex to an ancestor; 
  a selfloop, in a directed graph, can be a back edge;
Chapter 22. Elementary graph algorithms

Classification of edges (for directed graphs)

- **tree edges**: those in the search tree (forest); 
  \((u, v)\) is a tree edge if \(v\) was discovered by exploring \((u, v)\);
- **back edges**: those connecting a vertex to an ancestor; 
  a selfloop, in a directed graph, can be a back edge;
- **forward edges**: those connecting a vertex to a descendant;
Classification of edges (for directed graphs)

- **tree edges**: those in the search tree (forest);
  \((u, v)\) is a tree edge if \(v\) was discovered by exploring \((u, v)\);
- **back edges**: those connecting a vertex to an ancestor;
  a selfloop, in a directed graph, can be a back edge;
- **forward edges**: those connecting a vertex to a descendant;
- **cross edges**: all other edges;
To identify the type of edge \((u,v)\) with the color of \(v\):

- **WHITE**: tree edge;
- **GRAY**: back edge;
- **BLACK**: forward or cross edge;

First stages of a Directed DFS, showing **Edges**, the **DFS TREE**, a **Tree Edge**, a **Back Edge**, a **Forward Edge**, and a **Cross Edge**.
Chapter 22. Elementary graph algorithms

To identify the type of edge \((u, v)\) with the color of \(v\):

First stages of a Directed DFS, showing Edges, the DFS TREE, a Tree Edge, a Back Edge, a Forward Edge, and a Cross Edge.

To identify the type of edge \((u, v)\) with the color of \(v\):
Chapter 22. Elementary graph algorithms

To identify the type of edge \((u, v)\) with the color of \(v\):

WHITE: tree edge;
Chapter 22. Elementary graph algorithms

To identify the type of edge $(u, v)$ with the color of $v$:

WHITE: tree edge;

GRAY: back edge;

First stages of a Directed DFS, showing Edges, the DFS TREE, a Tree Edge, a Back Edge, a Forward Edge, and a Cross Edge.
Chapter 22. Elementary graph algorithms

To identify the type of edge \((u, v)\) with the color of \(v\):

- WHITE: tree edge;
- GRAY: back edge;
- BLACK: forward or cross edge;
Theorem 22.10 In a depth-first search of undirected graph $G$, every edge of $G$ is either tree edge or back edge.
**Theorem 22.10** In a depth-first search of undirected graph $G$, every edge of $G$ is either tree edge or back edge.

**Proof.** Let $(u, v)$ be an edge in $G$. Assume in the depth-first search of $G$, $u.d < v.d$. 


Chapter 22. Elementary graph algorithms

**Theorem 22.10** In a depth-first search of undirected graph $G$, every edge of $G$ is either tree edge or back edge.

**Proof.** Let $(u, v)$ be an edge in $G$. Assume in the depth-first search of $G$, $u.d < v.d$. Then there are two scenarios:
Theorem 22.10 In a depth-first search of undirected graph $G$, every edge of $G$ is either tree edge or back edge.

Proof. Let $(u, v)$ be an edge in $G$. Assume in the depth-first search of $G$, $u.d < v.d$. Then there are two scenarios:

1. $v$ is discovered by exploring edge $(u, v)$, then $(u, v)$ is a tree edge;
Chapter 22. Elementary graph algorithms

**Theorem 22.10** In a depth-first search of undirected graph $G$, every edge of $G$ is either a tree edge or a back edge.

**Proof.** Let $(u, v)$ be an edge in $G$. Assume in the depth-first search of $G$, $u.d < v.d$. Then there are two scenarios:

1. $v$ is discovered by exploring edge $(u, v)$, then $(u, v)$ is a tree edge;
2. $v$ is discovered not through exploring edge $(u, v)$. 
Chapter 22. Elementary graph algorithms

Theorem 22.10 In a depth-first search of undirected graph $G$, every edge of $G$ is either tree edge or back edge.

Proof. Let $(u, v)$ be an edge in $G$. Assume in the depth-first search of $G$, $u.d < v.d$. Then there are two scenarios:

1. $v$ is discovered by exploring edge $(u, v)$, then $(u, v)$ is a tree edge;
2. $v$ is discovered not through exploring edge $(u, v)$.

Because $(u, v)$ is an edge, $v$ is discovered when $u$ is in gray color.
Chapter 22. Elementary graph algorithms

Theorem 22.10 In a depth-first search of undirected graph $G$, every edge of $G$ is either tree edge or back edge.

Proof. Let $(u, v)$ be an edge in $G$. Assume in the depth-first search of $G$, $u.d < v.d$. Then there are two scenarios:

1. $v$ is discovered by exploring edge $(u, v)$, then $(u, v)$ is a tree edge;
2. $v$ is discovered not through exploring edge $(u, v)$.

Because $(u, v)$ is an edge, $v$ is discovered when $u$ is in gray color. Since $u$ is in the adjacency list of $v$, $(v, u)$ will eventually be explored and thus a back edge.
Chapter 22. Elementary graph algorithms

Breadth First Search (BFS)
Chapter 22. Elementary graph algorithms

Breadth First Search (BFS)
Chapter 22. Elementary graph algorithms

Breadth First Search Algorithm  
(with a queue)

Time complexity of BFS: \( O(|V|+|E|) \)

Note: BFS can find a shortest path from \( s \) to all other nodes (non-weighted). (Why?)
Chapter 22. Elementary graph algorithms

Breadth First Search Algorithm (with a queue)
Chapter 22. Elementary graph algorithms

Breadth First Search Algorithm (with a queue)

```
BFS(G, s)
1   for each vertex u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   ENQUEUE(Q, s)
10  while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13        if v.color == WHITE
14           v.color = GRAY
15           v.d = u.d + 1
16           v.π = u
17           ENQUEUE(Q, v)
18     u.color = BLACK
```
Chapter 22. Elementary graph algorithms

Breadth First Search Algorithm (with a queue)

\[
\begin{align*}
BFS(G, s) \\
1 & \text{for each vertex } u \in G.V - \{s\} \\
2 & \quad u.color = \text{WHITE} \\
3 & \quad u.d = \infty \\
4 & \quad u.\pi = \text{NIL} \\
5 & \quad s.color = \text{GRAY} \\
6 & \quad s.d = 0 \\
7 & \quad s.\pi = \text{NIL} \\
8 & \quad Q = \emptyset \\
9 & \quad \text{ENQUEUE}(Q, s) \\
10 & \text{while } Q \neq \emptyset \\
11 & \quad u = \text{DEQUEUE}(Q) \\
12 & \quad \text{for each } v \in G.\text{Adj}[u] \\
13 & \quad \quad \text{if } v.color == \text{WHITE} \\
14 & \quad \quad \quad v.color = \text{GRAY} \\
15 & \quad \quad \quad v.d = u.d + 1 \\
16 & \quad \quad \quad v.\pi = u \\
17 & \quad \quad \text{ENQUEUE}(Q, v) \\
18 & \quad u.color = \text{BLACK}
\end{align*}
\]

Time complexity of BFS:
Chapter 22. Elementary graph algorithms

Breadth First Search Algorithm (with a queue)

```
BFS(G, s)
1  for each vertex u ∈ G.V - {s}
2     u.color = WHITE
3     u.d = ∞
4     u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = Ø
9  ENQUEUE(Q, s)
10  while Q ≠ Ø
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

Time complexity of BFS: \( O(|V| + |E|) \)
Chapter 22. Elementary graph algorithms

Breadth First Search Algorithm (with a queue)

```
BFS(G, s)
1   for each vertex u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   ENQUEUE(Q, s)
10  while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13        if v.color == WHITE
14           v.color = GRAY
15           v.d = u.d + 1
16           v.π = u
17           ENQUEUE(Q, v)
18     u.color = BLACK
```

Time complexity of BFS: $O(|V| + |E|)$
Note: BFS can find a shortest path from $s$ to all other nodes (non-weighted). (Why?)
Chapter 22. Elementary graph algorithms

Applications

Reachability Problem

Input: \( G = (V,E) \), and \( s,t \in V \);

Output: YES if and only there is a path \( s \to t \) in \( G \).

• The problem can be solved with DFS and BFS by search on the graph from \( s \) until \( t \) shows up.
Chapter 22. Elementary graph algorithms

Applications

Reachability Problem

Input: \( G = (V, E) \), and \( s, t \in V \);

Output: YES if and only there is a path \( s \Rightarrow t \) in \( G \).

• The problem can be solved with DFS and BFS by search on the graph from \( s \) until \( t \) shows up.
Applications

Reachability Problem
Chapter 22. Elementary graph algorithms

Applications

Reachability Problem

**Input:** $G = (V, E)$, and $s, t \in V$,
 Applications

 Reachability Problem

 **Input:** $G = (V, E)$, and $s, t \in V$;
 **Output:** YES if and only there is a path $s \leadsto t$ in $G$. 
Applications

Reachability Problem

**Input:** \( G = (V, E) \), and \( s, t \in V \), ;
**Output:** \textbf{YES} if and only if there is a path \( s \leadsto t \) in \( G \).

- The problem can be solved with DFS and BFS
Applications

Reachability Problem

**Input:** $G = (V, E)$, and $s, t \in V$, ;
**Output:** YES if and only there is a path $s \sim t$ in $G$.

- The problem can be solved with DFS and BFS
  - by search on the graph from $s$ until $t$ shows up.
Applications

Reachability Problem

**Input:** $G = (V, E)$, and $s, t \in V$;

**Output:** YES if and only there is a path $s \sim t$ in $G$.

- The problem can be solved with DFS and BFS
  - by search on the graph from $s$ until $t$ shows up.
Chapter 22. Elementary graph algorithms

Reachability Problem

Reachability \((G, u, t)\);

1. \(u\).\text{visit} = \text{true};
2. \(\text{for each} \ v \in \text{Adj}[u] \text{and not} \ v\text{.visit};
3. \(\text{if} \ v = t \text{then} \text{reachable} = \text{Yes}; \text{exit};
4. \text{else} \ v\text{.π} = u;
5. \text{Reachability}(G, v, t);
6. \text{return} ( );

Main()

\text{reachable} = \text{No}; \text{Reachability}(G, s, t);
\text{print} (\text{reachable});
Reachability Problem

1. $u \text{.visit} = \text{true}$
2. For each $v \in \text{Adj}[u]$ and not $v \text{.visit}$
3. If $v = t$ then \text{reachable} = \text{Yes}; \text{exit}
4. Else $v.\pi = u$
5. Reachability$(G, v, t)$
6. Return $( )$

Main() 
reachable = \text{No}$
Reachability$(G, s, t)$;
print (reachable)
Reachability Problem

**REACHABILITY**\((G, u, t)\);
1. \(u.visit = \text{true}\);
2. for each \(v \in \text{Adj}[u]\) and not \(v.visit\);
3. if \(v = t\) then reachable = Yes; exit;
4. else \(v.\pi = u\);
5. \text{REACHABILITY}(G, v, t);
6. return ( );
Chapter 22. Elementary graph algorithms

Reachability Problem

**REACHABILITY**\((G, u, t)\);

1. \(u.visit = \textbf{true};\)
2. \(\textbf{for each } v \in Adj[u] \textbf{ and not } v.visit;\)
3. \(\textbf{if } v = t \textbf{ then } \textbf{reachable } = \textbf{Yes}; \textbf{exit};\)
4. \(\textbf{else } v.\pi = u;\)
5. \( \textbf{REACHABILITY}(G, v, t);\)
6. \(\textbf{return } ( );\)

**MAIN()**

\(\textbf{reachable } = \textbf{No};\)

**REACHABILITY**(\(G, s, t\));

**print**(reachable);
Path Counting Problem

Input:

\( G = (V, E) \), and \( s, t \in V \);

Output: the number of paths from \( s \) ⇝ \( t \) in \( G \).

- we modify Reachability to count paths.

PathCounting \((G, u, t)\):

1. \( u.\text{visit} = \text{true} \);
2. for each \( v \in \text{Adj}[u] \);
3. if \( v.\text{visit} \) then \( u.c = u.c + v.c \);
4. else \( v.\pi = u \);
5. PathCounting \((G, v, t)\);
6. \( u.c = u.c + v.c \);
7. return ()

Main :

1. for each \( u \in G \)
2. \( u.c = 0 \);
3. PathCounting \((G, s, t)\);
4. print \((s.c)\)
Chapter 22. Elementary graph algorithms

Path Counting Problem

Input: \( G = (V,E) \), and \( s,t \in V \);

Output: the number of paths from \( s \) \( \rightarrow \) \( t \) in \( G \).

- we modify Reachability to count paths.

PathCounting \( (G,u,t) \):

1. \( u.\text{visit} = \text{true} \);
2. for each \( v \in \text{Adj}[u] \);
3. if \( v.\text{visit} \) then \( u.\text{c} = u.\text{c} + v.\text{c} \);
4. else \( v.\pi = u \);
5. PathCounting \( (G,v,t) \);
6. \( u.\text{c} = u.\text{c} + v.\text{c} \);
7. return ( );

Main ()

1. for each \( u \in G \);
2. \( u.\text{c} = 0 \);
3. PathCounting \( (G,s,t) \);
4. print ( \( s.\text{c} \) )


Path Counting Problem

**Input:** $G = (V, E)$, and $s, t \in V$;
Chapter 22. Elementary graph algorithms

Path Counting Problem

**Input:** \( G = (V, E) \), and \( s, t \in V \);

**Output:** the number of paths from \( s \leadsto t \) in \( G \).
Path Counting Problem

**Input:** \( G = (V, E) \), and \( s, t \in V \);

**Output:** the number of paths from \( s \leadsto t \) in \( G \).

- we modify \texttt{Reachability} to count paths.

```python
PathCounting(G, u, t):
1. u.visit = True;
2. for each v \in \text{Adj}[u];
3.   if v.visit then u.c = u.c + v.c;
4.   else v.\pi = u;
5.   PathCounting(G, v, t);
6. u.c = u.c + v.c
7. return ()
```

```python
Main()
1. for each u \in G
2.   u.c = 0
3. PathCounting(G, s, t);
4. print(s.c)
```
Path Counting Problem

Input: \( G = (V, E) \), and \( s, t \in V \);  
Output: the number of paths from \( s \leadsto t \) in \( G \).

- we modify Reachability to count paths.

PathCounting\((G, u, t)\);
1. \( u.visit = \text{true} \);
2. \( \text{for each } v \in \text{Adj}[u] \); 
3. \( \text{if } v.visit \text{ then } u.c = u.c + v.c \); 
4. \( \text{else } v.\pi = u \); 
5. \( \text{PathCounting}(G, v, t) \); 
6. \( u.c = u.c + v.c \) 
7. \( \text{return } ( ) \);
Path Counting Problem

**INPUT:** $G = (V, E)$, and $s, t \in V$ ;

**OUTPUT:** the number of paths from $s \leadsto t$ in $G$.

- we modify Reachability to count paths.

**PathCounting**($G, u, t$);
1. $u.visit = \text{true}$;
2. for each $v \in \text{Adj}[u]$ ;
3. \quad if $v.visit$ then $u.c = u.c + v.c$;
4. \quad else $v.\pi = u$;
5. \quad **PathCounting**($G, v, t$);
6. \quad $u.c = u.c + v.c$
7. return ( );

**Main**()
1. for each $u \in G$
2. \quad $u.c = 0$;
3. **PathCounting**($G, s, t$);
4. **print** ($s.c$)
Chapter 22. Elementary graph algorithms

Topological sorting

• On directed acyclic graphs (DAGs)

A sorted order:
socks, shorts, pants, shoes, shirt, tie, belt, jacket, watch.
Chapter 22. Elementary graph algorithms

Topological sorting
Chapter 22. Elementary graph algorithms

Topological sorting

- On directed acyclic graphs (DAGs)
Chapter 22. Elementary graph algorithms

Topological sorting

• On directed acyclic graphs (DAGs)

A sorted order: socks, shorts, pants, shoes, shirt, tie, belt, jacket, watch.
Chapter 22. Elementary graph algorithms

• apply DFS algorithm.
• reversed order of finish times: p,n,o,s,m,r,y,v,x,w,z,u,q,t
• Correctness proof?
Chapter 22. Elementary graph algorithms

- apply DFS algorithm.
Chapter 22. Elementary graph algorithms

- apply DFS algorithm.
Chapter 22. Elementary graph algorithms

- apply DFS algorithm.

- reversed order of finish times:

```
p, n, o, s, m, r, y, v, x, w, z, u, q, t
```

- Correctness proof?
Chapter 22. Elementary graph algorithms

• apply DFS algorithm.

• reversed order of finish times:

  \[ p, n, o, s, m, r, y, v, x, w, z, u, q, t \]

• Correctness proof?
Chapter 22. Elementary graph algorithms

### Strongly connected components (SCC)

Let $G = (V,E)$ be a digraph. A strongly connected component (SCC) is a maximal subgraph $H = (V_H,E_H)$ of $G$ such that for every two nodes $v,u \in V_H$,

1. there is a directed path $v \rightarrow u$ consisting of edges in $E_H$; and
2. there is a directed path $u \rightarrow v$ consisting of edges in $E_H$. 
Chapter 22. Elementary graph algorithms

Strongly connected components (SCC)
Strongly connected components (SCC)

Let $G = (V, E)$ be a digraph. A strongly connected component is a maximal subgraph $H = (V_H, E_H)$ of $G$.
Chapter 22. Elementary graph algorithms

Strongly connected components (SCC)

Let $G = (V, E)$ be a digraph. A *strongly connected component* is a maximal subgraph $H = (V_H, E_H)$ of $G$ such that for every two nodes $v, u \in V_H$, 

(1) there is a directed path $v \rightarrow u$ consisting of edges in $E_H$;
(2) there is a directed path $u \rightarrow v$ consisting of edges in $E_H$. 

Strongly connected components (SCC)

Let $G = (V, E)$ be a digraph. A strongly connected component is a maximal subgraph $H = (V_H, E_H)$ of $G$ such that for every two nodes $v, u \in V_H$,

(1) there is a directed path $v \Rightarrow u$ consisting of edges in $E_H$; and
Chapter 22. Elementary graph algorithms

Strongly connected components (SCC)

Let $G = (V, E)$ be a digraph. A strongly connected component is a maximal subgraph $H = (V_H, E_H)$ of $G$ such that for every two nodes $v, u \in V_H$, 
(1) there is a directed path $v \rightarrow u$ consisting of edges in $E_H$; and 
(2) there is a directed path $u \rightarrow v$ consisting of edges in $E_H$. 


Chapter 22. Elementary graph algorithms

Strongly connected components (SCC)

Let $G = (V, E)$ be a digraph. A strongly connected component is a maximal subgraph $H = (V_H, E_H)$ of $G$ such that for every two nodes $v, u \in V_H$,

1. there is a directed path $v \rightarrow u$ consisting of edges in $E_H$; and
2. there is a directed path $u \rightarrow v$ consisting of edges in $E_H$. 

Idea of an algorithm to use DFS to solve SCC problem.

- use DFS to generate DFS forest;
- each search tree $T_u$ (rooted at $u$) consists of vertices $v$ such that $u \Rightarrow v$;
- use DFS again on $T_u$; hope to search from every one $v$ within $T_u$ to make sure $v \Rightarrow u$ as well.
- however, this may be difficult (proof is left as an exercise).
Idea of an algorithm to use DFS to solve SCC problem.

• use DFS to generate DFS forest; each search tree $T_u$ (rooted at $u$) consists of vertices $v$ such that $u \rightarrow v$;

• use DFS again on $T_u$; hope to search from every one $v$ within $T_u$ to make sure $v \rightarrow u$ as well.

• however, this may be difficult (proof is left as an exercise).
Idea of an algorithm to use DFS to solve SCC problem.

• use DFS to generate DFS forest;
Idea of an algorithm to use DFS to solve SCC problem.

- use DFS to generate DFS forest; each search tree $T_u$ (rooted at $u$) consists of vertices $v$ such that $u \rightarrow v$;
Idea of an algorithm to use DFS to solve SCC problem.

- use DFS to generate DFS forest; each search tree $T_u$ (rooted at $u$) consists of vertices $v$ such that $u \leadsto v$;
- use DFS again on $T_u$;
Idea of an algorithm to use DFS to solve SCC problem.

- use DFS to generate DFS forest; each search tree $T_u$ (rooted at $u$) consists of vertices $v$ such that $u \leadsto v$;
- use DFS again on $T_u$; hope to search from every one $v$ within $T_u$ to make sure $v \leadsto u$ as well.
Idea of an algorithm to use DFS to solve SCC problem.

- use DFS to generate DFS forest; each search tree $T_u$ (rooted at $u$) consists of vertices $v$ such that $u \leadsto v$;
- use DFS again on $T_u$; hope to search from every one $v$ within $T_u$ to make sure $v \leadsto u$ as well.
- however, this may be difficult (proof is left as an exercise).
Chapter 22. Elementary graph algorithms

Algorithm

Strongly Connected Components (G)

1. call DFS(G) to compute u.f for each u ∈ G.V
2. compute G_T the transpose of G {reverse all edges in G}
3. call DFS(G_T) (vertices are considered in the decreasing order of finish times computed in step 1)
4. output each tree in the depth-first forest produced by step 3.
Chapter 22. Elementary graph algorithms

Algorithm **STRONGLY CONNECTED COMPONENTS**($G$)
Algorithm \textbf{Strongly Connected Components}(G)

1. \textbf{call} \text{DFS}(G) to compute \(u.f\) for each \(u \in G.V\)
Chapter 22. Elementary graph algorithms

Algorithm STRONGLY CONNECTED COMPONENTS(G)
1. call DFS(G) to compute $u.f$ for each $u \in G.V$
2. compute $G^T$ the transpose of $G$
Chapter 22. Elementary graph algorithms

Algorithm **Strongly Connected Components**($G$)

1. call DFS($G$) to compute $u.f$ for each $u \in G.V$
2. compute $G^T$ the transpose of $G$ \{ reverse all edges in $G$ \}
Algorithm **Strongly Connected Components**(\(G\))
1. **call** DFS(G) to compute \(u.f\) for each \(u \in G.V\)
2. compute \(G^T\) the transpose of \(G\) \{reverse all edges in \(G\}\)
3. **call** DFS\((G^T)\) (vertices are considered in the decreasing order of finish times computed in step 1)
Algorithm Strongly Connected Components($G$)

1. call DFS($G$) to compute $u.f$ for each $u \in G.V$
2. compute $G^T$ the transpose of $G$ \{ reverse all edges in $G$ \}
3. call DFS($G^T$) (vertices are considered in the decreasing
   order of finish times computed in step 1)
4. output each tree in the depth-first forest produced by step 3.
Chapter 22. Elementary graph algorithms

Algorithm **Strongly Connected Components**($G$)
1. **call** DFS($G$) to compute $u.f$ for each $u \in G.V$
2. compute $G^T$ the transpose of $G$ \{ reverse all edges in $G$ \}
3. **call** DFS($G^T$) (vertices are considered in the decreasing order of finish times computed in step 1)
4. output each tree in the depth-first forest produced by step 3.
Chapter 22. Elementary graph algorithms

Ideas behind the algorithm:

• the first pass DFS results in a DFS forest, let $T$ be a tree with root $r$; for every vertex $u \in T$, $r \xrightarrow{} u$, so an SCC can only be produced from some tree in the forest;

• let vertex $v \in T$ but $v \neq r$, we are not sure $v \xrightarrow{} u$;

• instead, we would like to check if $v \xrightarrow{} r$ for $v \in T$.
  (because $r \xrightarrow{} u$, $v \xrightarrow{} r$ implies $v \xrightarrow{} u$);

• that is the same as to use second-DFS (starting from $r$) to check if $r \xrightarrow{} v$ after edge directions are reversed;

• only those in the same second-DFS tree belongs to the same SCC.
Ideas behind the algorithm:

- the first pass DFS results in a DFS forest, let $T$ be a tree with root $r$;
Chapter 22. Elementary graph algorithms

Ideas behind the algorithm:

• the first pass DFS results in a DFS forest, let $T$ be a tree with root $r$; for every vertex $u \in T$, $r \leadsto u$,
Ideas behind the algorithm:

- the first pass DFS results in a DFS forest, let $T$ be a tree with root $r$; for every vertex $u \in T$, $r \sim u$,
  so an SCC can only be produced from some tree in the forest;
Ideas behind the algorithm:

• the first pass DFS results in a DFS forest, let $T$ be a tree with root $r$; for every vertex $u \in T$, $r \leadsto u$,
  so an SCC can only be produced from some tree in the forest;

• let vertex $v \in T$ but $v \neq r$, we are not sure $v \leadsto u$;

• that is the same as to use second-DFS (starting from $r$) to check if $r \leadsto v$ after edge directions are reversed;

• only those in the same second-DFS tree belongs to the same SCC.
Ideas behind the algorithm:

- the first pass DFS results in a DFS forest, let $T$ be a tree with root $r$; for every vertex $u \in T$, $r \rightsquigarrow u$, so an SCC can only be produced from some tree in the forest;
- let vertex $v \in T$ but $v \neq r$, we are not sure $v \rightsquigarrow u$;
- instead, we would like to check if $v \rightsquigarrow r$ for $v \in T$. 
Ideas behind the algorithm:

- the first pass DFS results in a DFS forest, let $T$ be a tree with root $r$; for every vertex $u \in T$, $r \leadsto u$, so an SCC can only be produced from some tree in the forest;

- let vertex $v \in T$ but $v \neq r$, we are not sure $v \leadsto u$;

- instead, we would like to check if $v \leadsto r$ for $v \in T$. (because $r \leadsto u$, $v \leadsto r$ implies $v \leadsto u$);
Chapter 22. Elementary graph algorithms

Ideas behind the algorithm:

• the first pass DFS results in a DFS forest, let \( T \) be a tree with root \( r \); for every vertex \( u \in T \), \( r \leadsto u \), so an SCC can only be produced from some tree in the forest;

• let vertex \( v \in T \) but \( v \neq r \), we are not sure \( v \leadsto u \); instead, we would like to check if \( v \leadsto r \) for \( v \in T \).
  (because \( r \leadsto u \), \( v \leadsto r \) implies \( v \leadsto u \));

• that is the same as to use second-DFS (starting from \( r \)) to check if \( r \leadsto v \) after edge directions are reversed;
Chapter 22. Elementary graph algorithms

Ideas behind the algorithm:

• the first pass DFS results in a DFS forest, let $T$ be a tree with root $r$; for every vertex $u \in T$, $r \rightsquigarrow u$, so an SCC can only be produced from some tree in the forest;

• let vertex $v \in T$ but $v \neq r$, we are not sure $v \rightsquigarrow u$;

• instead, we would like to check if $v \rightsquigarrow r$ for $v \in T$. (because $r \rightsquigarrow u$, $v \rightsquigarrow r$ implies $v \rightsquigarrow u$);

• that is the same as to use second-DFS (starting from $r$) to check if $r \rightsquigarrow v$ after edge directions are reversed;

• only those in the same second-DSF tree belongs to the same SCC.
Chapter 22. Elementary graph algorithms

Properties from algorithm

Strongly Connected Components \((G)\)

(1) Component graph: \(G_{SCC} = (V_{SCC}, E_{SCC})\)

Let \(C_1, C_2, ..., C_k\) be \(k\) distinct SCCs for \(G\). Then \(V_{SCC} = \{v_1, v_2, ..., v_k\}\); \(E_{SCC} = \{(v_i, v_j) : \exists u \in C_i, v \in C_j, (u, v) \in E\}\).

Then \(G_{SCC}\) is a DAG (directed acyclic graph).

Proof. Assume the opposite to the claim that, for some \(v_i, v_j \in V_{SCC}\), there is a path \(v_i \rightarrow v_j\) and another path \(v_j \rightarrow v_i\), forming a cycle in \(V_{SCC}\).

By the definition of \(G_{SCC}\), there must be a path in \(G\), from some vertex in \(C_i\) to some vertex in \(C_j\); at the same time, there is a path in \(G\), from some vertex in \(C_j\) to some vertex in \(C_i\). Then \(C_i\) and \(C_j\) should form a single SCC, not two distinct SCCs. Contradicts.
Properties from algorithm \textsc{Strongly Connected Components}(G)

\textbf{Component graph:}

\(G_{\text{SCC}} = (V_{\text{SCC}}, E_{\text{SCC}})\)

\(V_{\text{SCC}} = \{v_1, v_2, \ldots, v_k\}\)

\(E_{\text{SCC}} = \{(v_i, v_j) : \exists u \in C_i, v \in C_j, (u, v) \in E\}\)

Then \(G_{\text{SCC}}\) is a DAG (directed acyclic graph).

**Proof.** Assume the opposite to the claim that, for some \(v_i, v_j \in V_{\text{SCC}}\), there is a path \(v_i \xrightarrow{} v_j\) and another path \(v_j \xrightarrow{} v_i\), forming a cycle in \(V_{\text{SCC}}\). By the definition of \(G_{\text{SCC}}\), there must be a path in \(G\), from some vertex in \(C_i\) to some vertex in \(C_j\); at the same time, there is a path in \(G\), from some vertex in \(C_j\) to some vertex in \(C_i\). Then \(C_i\) and \(C_j\) should form a single SCC, not two distinct SCCs. Contradicts.
Chapter 22. Elementary graph algorithms

Properties from algorithm `STRONGLY CONNECTED COMPONENTS(G)`

1. Component graph: \( G^{SCC} = (V^{SCC}, E^{SCC}) \) is defined as follow:

\[
\begin{align*}
\text{let } & C_1, C_2, \ldots, C_k \text{ be } k \text{ distinct SCCs for } G. \\
\text{then } & V^{SCC} = \{v_1, v_2, \ldots, v_k\}; \\
\text{and } & E^{SCC} = \{(v_i, v_j) : \exists u \in C_i, v \in C_j, (u, v) \in E\}.
\end{align*}
\]

Then \( G^{SCC} \) is a DAG (directed acyclic graph).

Proof. Assume the opposite to the claim that, for some \( v_i, v_j \in V^{SCC} \), there is a path \( v_i \Rightarrow v_j \) and another path \( v_j \Rightarrow v_i \), forming a cycle in \( V^{SCC} \).

By the definition of \( G^{SCC} \), there must be a path in \( G \), from some vertex in \( C_i \) to some vertex in \( C_j \); at the same time, there is a path in \( G \), from some vertex in \( C_j \) to some vertex in \( C_i \). Then \( C_i \) and \( C_j \) should form a single SCC, not two distinct SCCs. Contradicts.
Chapter 22. Elementary graph algorithms

Properties from algorithm \textsc{Strongly Connected Components}(G)

(1) Component graph: $G^{SCC} = (V^{SCC}, E^{SCC})$ is defined as follow:

\begin{itemize}
  \item let $C_1, C_2, \ldots, C_k$ be $k$ distinct SCCs for $G$. Then
\end{itemize}
Chapter 22. Elementary graph algorithms

Properties from algorithm **Strongly Connected Components** \(G\)

1. Component graph: \(G^{SCC} = (V^{SCC}, E^{SCC})\) is defined as follow:

   let \(C_1, C_2, \ldots, C_k\) be \(k\) distinct SCCs for \(G\). Then

   \[
   V^{SCC} = \{v_1, v_2, v_k\};
   \]

   \[
   E^{SCC} = \{(v_i, v_j) : \exists u \in C_i, v \in C_j, (u, v) \in E\}.
   \]
Properties from algorithm \textbf{Strongly Connected Components}(G)

(1) Component graph: $G^{SCC} = (V^{SCC}, E^{SCC})$ is defined as follow:

let $C_1, C_2, \ldots, C_k$ be $k$ distinct SCCs for $G$. Then

$V^{SCC} = \{v_1, v_2, v_k\}$;
$E^{SCC} = \{(v_i, v_j) : \exists u \in C_i, v \in C_j, (u, v) \in E\}$.

Then $G^{SCC}$ is a DAG (directed acyclic graph).
Properties from algorithm Strongly Connected Components \((G)\)

(1) Component graph: \(G^{SCC} = (V^{SCC}, E^{SCC})\) is defined as follow:

let \(C_1, C_2, \ldots, C_k\) be \(k\) distinct SCCs for \(G\). Then

\[
V^{SCC} = \{v_1, v_2, v_k\};
\]

\[
E^{SCC} = \{(v_i, v_j) : \exists u \in C_i, v \in C_j, (u, v) \in E\}.
\]

Then \(G^{SCC}\) is a DAG (directed acyclic graph).

Proof. Assume the opposite to the claim that, for some \(v_i, v_j \in V^{SCC}\), there is a path \(v_i \leadsto v_j\) and another path \(v_j \leadsto v_i\), forming a cycle in \(V^{SCC}\).
Chapter 22. Elementary graph algorithms

Properties from algorithm \textbf{Strongly Connected Components}(G)

(1) Component graph: $G^{SCC} = (V^{SCC}, E^{SCC})$ is defined as follow:

   let $C_1, C_2, \ldots, C_k$ be $k$ distinct SCCs for $G$. Then

   $V^{SCC} = \{v_1, v_2, v_k\}$;

   $E^{SCC} = \{(v_i, v_j) : \exists u \in C_i, v \in C_j, (u, v) \in E\}$.

   Then $G^{SCC}$ is a DAG (directed acyclic graph).

\textbf{Proof}. Assume the opposite to the claim that, for some $v_i, v_j \in V^{SCC}$, there is a path $v_i \leadsto v_j$ and another path $v_j \leadsto v_i$, forming a cycle in $V^{SCC}$.

By the definition of $G^{SCC}$, there must be a path in $G$, from some vertex in $C_i$ to some vertex in $C_j$; at the same time, there is a path in $G$, from some vertex in $C_j$ to some vertex in $C_i$. Then $C_i$ and $C_j$ should form a single SCC, not two distinct SCCs. \textbf{Contradicts}. 
Chapter 22. Elementary graph algorithms

...
Chapter 22. Elementary graph algorithms

Let $C$ be a SCC, define $f(C) = \max_{u \in C \{u.f\}}$, (with the finish times from the first DFS call).
Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u, v \in C$ and $v \in C'$, then $f(C) > f(C')$. 
Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u,v) \in E$, where $u, v \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$. 

Chapter 22. Elementary graph algorithms

Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u, v \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$.
Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$. 
Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{ u.f \}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u,v) \in E$, where $u \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$. Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$.

Now consider the first DFS call, there are two situations:
Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u, v \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$. Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$.

Now consider the first DFS call, there are two situations:

(1) $y$ was searched before $x$:
Chapter 22. Elementary graph algorithms

Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$. Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$.

Now consider the first DFS call, there are two situations:

(1) $y$ was searched before $x$:
   by property (1) there is no path from $y$ to $x$,
Chapter 22. Elementary graph algorithms

Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14:** Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof:** Assume the opposite, i.e., $f(C) < f(C')$. Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$.

Now consider the first DFS call, there are two situations:

(1) $y$ was searched before $x$:
   by property (1) there is no path from $y$ to $x$, $x.f > y.f$. 
Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u, \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$. Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$.

Now consider the first DFS call, there are two situations:

(1) $y$ was searched before $x$:
   by property (1) there is no path from $y$ to $x$, $x.f > y.f$.

(2) $y$ was search after $x$:
Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u, v \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$.
Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$.

Now consider the first DFS call, there are two situations:

(1) $y$ was searched before $x$:
   by property (1) there is no path from $y$ to $x$, $x.f > y.f$.

(2) $y$ was search after $x$:
   since there is a path from $x$ to $y$ because of $(u, v)$,
Let $C$ be a SCC, define $f(C) = \max_{u \in C}\{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14** : Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u,v) \in E$, where $u, v \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$.
Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$.

Now consider the first DFS call, there are two situations:

(1) $y$ was searched before $x$:
   by property (1) there is no path from $y$ to $x$, $x.f > y.f$.

(2) $y$ was search after $x$:
   since there is a path from $x$ to $y$ because of $(u,v)$, $x.f > y.f$. 
Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) **Lemma 22.14**: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u, v \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof**: Assume the opposite, i.e., $f(C) < f(C')$. Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$. Now consider the first DFS call, there are two situations:

(1) $y$ was searched before $x$:
   by property (1) there is no path from $y$ to $x$, $x.f > y.f$.

(2) $y$ was search after $x$:
   since there is a path from $x$ to $y$ because of $(u, v)$, $x.f > y.f$.

Both cases contradicts the assumption.
Chapter 22. Elementary graph algorithms

Let $C$ be a SCC, define $f(C) = \max_{u \in C} \{u.f\}$, (with the finish times from the first DFS call).

(2) Lemma 22.14: Let $C$ and $C'$ be distinct strongly connected components for directed graph $G$. If $(u, v) \in E$, where $u, v \in C$ and $v \in C'$, then $f(C) > f(C')$.

Proof: Assume the opposite, i.e., $f(C) < f(C')$. Then there must be vertices $x \in C$ and $y \in C'$ such that $x.f < y.f$.

Now consider the first DFS call, there are two situations:

(1) $y$ was searched before $x$:
   by property (1) there is no path from $y$ to $x$, $x.f > y.f$.

(2) $y$ was search after $x$:
   since there is a path from $x$ to $y$ because of $(u, v)$, $x.f > y.f$.

Both cases contradicts the assumption. So $f(C) > f(C')$. 
Chapter 22. Elementary graph algorithms

The algorithm

Strongly Connected Components

\(G\)
correctly computes the strongly connected components for a directed graph \(G\).

We need to prove two statements:

1. If \(v \rightarrow u\) and \(u \rightarrow v\) in \(G\), then \(u\) and \(v\) belong to the same component \(C\) produced by the algorithm.

2. If \(u, v \in C\), then we have \(v \rightarrow u\) and \(u \rightarrow v\) in \(G\).
Chapter 22. Elementary graph algorithms

(3) The algorithm \texttt{Strongly Connected Components}(G) correctly computes the strongly connected components for a directed graph $G$. 
(3) The algorithm Strongly Connected Components\((G)\) correctly computes the strongly connected components for a directed graph \(G\).

We need to prove two statements:
(3) The algorithm \texttt{Strongly Connected Components}(G) correctly computes the strongly connected components for a directed graph G.

We need to prove two statements:

(1) If \( v \leadsto u \) and \( u \leadsto v \) in G, then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.
(3) The algorithm $\text{Strongly Connected Components}(G)$ correctly computes the strongly connected components for a directed graph $G$.

We need to prove two statements:

(1) If $v \leadsto u$ and $u \leadsto v$ in $G$, then $u$ and $v$ belong to the same component $C$ produced by the algorithm.

(2) If $u, v \in C$, then we have $v \leadsto u$ and $u \leadsto v$ in $G$. 
Proof:

(1) If \( v \rightarrow u \) and \( u \rightarrow v \) in \( G \), then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.

Sketch of proof:

• Assume in 1st DFS, \( v \) was discovered before \( u \) (or opposite);
• As \( v \rightarrow u \) in \( G \), \( u \) and \( v \) belong to the same search tree rooted at \( r \) with \( r.f \geq v.f > u.f \) (note: \( r \) could be just \( v \))
• As \( u \rightarrow v \) in \( G \), \( v \rightarrow u \) in \( G_T \);
• Now consider the 2nd DFS; there are 2 situations:
  (1) searching from some \( w \) with \( w.f \geq v.f \) (note: \( w \) could be \( v \)) finds \( v \) first; then it finds \( u \);
  (2) the search finds \( u \) first; because \( v \rightarrow u \) in \( G \), \( u \rightarrow v \) is in \( G_T \), it finds also \( v \).

In both situations, \( u \) and \( v \) belongs to the same search tree in the 2nd DFS search. Therefore, \( u \) and \( v \) belong to the same component.
Chapter 22. Elementary graph algorithms

Proof:

• Assume in 1st DFS, v was discovered before u (or opposite);
• As v⇝u in G, u and v belong to the same search tree rooted at r with r.f ≥ v.f > u.f (note: r could be just v);
• As u⇝v in G, v⇝u in G_T;
• Now consider the 2nd DFS; there are 2 situations:
  (1) searching from some w with w.f ≥ v.f (note: w could be v) finds v first; then it finds u;
  (2) the search finds u first; because v⇝u in G, u⇝v is in G_T, it finds also v.
In both situations, u and v belong to the same search tree in the 2nd DFS search. Therefore, u and v belong to the same component.
Chapter 22. Elementary graph algorithms

Proof:

(1) If \( v \sim u \) and \( u \sim v \) in \( G \) in \( G \), then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.
Proof:

(1) If $v \leadsto u$ and $u \leadsto v$ in $G$ in $G$, then $u$ and $v$ belong to the same component $C$ produced by the algorithm.

Sketch of proof:
Chapter 22. Elementary graph algorithms

Proof:

(1) If \( v \sim u \) and \( u \sim v \) in \( G \), then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.

Sketch of proof:

- assume in 1st DFS, \( v \) was discovered before \( u \) (or opposite);
Chapter 22. Elementary graph algorithms

Proof:

(1) If \( v \leadsto u \) and \( u \leadsto v \) in \( G \), then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.

Sketch of proof:

- assume in 1st DFS, \( v \) was discovered before \( u \) (or opposite);
- as \( v \leadsto u \) in \( G \), \( u \) and \( v \) belong to the same search tree rooted at \( r \) with \( r.f \geq v.f > u.f \) (note: \( r \) could be just \( v \))


Proof:

(1) If \( v \leadsto u \) and \( u \leadsto v \) in \( G \), then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.

Sketch of proof:

- assume in 1st DFS, \( v \) was discovered before \( u \) (or opposite);
- as \( v \leadsto u \) in \( G \), \( u \) and \( v \) belong to the same search tree rooted at \( r \) with \( r.f \geq v.f > u.f \) (note: \( r \) could be just \( v \))
- as \( u \leadsto v \) in \( G \), \( v \leadsto u \) in \( G^T \).
Proof:

(1) If $v \leadsto u$ and $u \leadsto v$ in $G$, then $u$ and $v$ belong to the same component $C$ produced by the algorithm.

Sketch of proof:

• assume in 1st DFS, $v$ was discovered before $u$ (or opposite);
• as $v \leadsto u$ in $G$, $u$ and $v$ belong to the same search tree rooted at $r$ with $r.f \geq v.f > u.f$ (note: $r$ could be just $v$)
• as $u \leadsto v$ in $G$, $v \leadsto u$ in $G^T$;
• now consider the 2nd DFS; there are 2 situations:
Chapter 22. Elementary graph algorithms

Proof:

(1) If \( v \leadsto u \) and \( u \leadsto v \) in \( G \), then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.

Sketch of proof:

- assume in 1st DFS, \( v \) was discovered before \( u \) (or opposite);
- as \( v \leadsto u \) in \( G \), \( u \) and \( v \) belong to the same search tree rooted at \( r \) with \( r.f \geq v.f > u.f \) (note: \( r \) could be just \( v \))
- as \( u \leadsto v \) in \( G \), \( v \leadsto u \) in \( G^T \);
- now consider the 2nd DFS; there are 2 situations:

  (1) searching from some \( w \) with \( w.f \geq v.f \) (note: \( w \) could be \( v \)) finds \( v \) first; then it finds \( u \);
Chapter 22. Elementary graph algorithms

Proof:

(1) If \( v \leadsto u \) and \( u \leadsto v \) in \( G \) in \( G \), then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.

Sketch of proof:

• assume in 1st DFS, \( v \) was discovered before \( u \) (or opposite);
• as \( v \leadsto u \) in \( G \), \( u \) and \( v \) belong to the same search tree rooted at \( r \) with \( r.f \geq v.f > u.f \) (note: \( r \) could be just \( v \))
• as \( u \leadsto v \) in \( G \), \( v \leadsto u \) in \( G^T \);
• now consider the 2nd DFS; there are 2 situations:

  (1) searching from some \( w \) with \( w.f \geq v.f \) (note: \( w \) could be \( v \))
      finds \( v \) first; then it finds \( u \);

  (2) the search finds \( u \) first; because \( v \leadsto u \) in \( G \), \( u \leadsto v \) is in \( G^T \),
      it finds also \( v \).
Chapter 22. Elementary graph algorithms

Proof:

(1) If \( v \sim u \) and \( u \sim v \) in \( G \) in \( G \), then \( u \) and \( v \) belong to the same component \( C \) produced by the algorithm.

Sketch of proof:

- assume in 1st DFS, \( v \) was discovered before \( u \) (or opposite);
- as \( v \sim u \) in \( G \), \( u \) and \( v \) belong to the same search tree rooted at \( r \) with \( r.f \geq v.f > u.f \) (note: \( r \) could be just \( v \))
- as \( u \sim v \) in \( G \), \( v \sim u \) in \( G^T \);
- now consider the 2nd DFS; there are 2 situations:

  (1) searching from some \( w \) with \( w.f \geq v.f \) (note: \( w \) could be \( v \)) finds \( v \) first; then it finds \( u \);

  (2) the search finds \( u \) first; because \( v \sim u \) in \( G \), \( u \sim v \) is in \( G^T \), it finds also \( v \).

In both situations, \( u \) and \( v \) belongs to the same search tree in the 2nd DFS search. Therefore, \( u \) and \( u \) belong to the same component.
Chapter 22. Elementary graph algorithms

(2) If \( u, v \in C \), then we have \( v \Rightarrow u \) and \( u \Rightarrow v \) in \( G \).

Sketch of proof:
(1) Assume \( u, v \) belong to the same tree of root \( r \) in 2nd DFS;
(2) then \( r.f > u.f \) and \( r.f > v.f \) in 1st DFS;
(3) the assumption in (1) also implies:
• \( r \Rightarrow u \) and \( r \Rightarrow v \) in \( G_T \);
• that is, \( u \Rightarrow r \) and \( v \Rightarrow r \) in \( G_T \);
• then \( u.f > r.f \) and \( v.f > r.f \) in 1st DFS, which conflict with conclusions in (2), UNLESS \( r \Rightarrow u \) and \( r \Rightarrow v \) in \( G_T \) also.
(4) This means: through \( r \), \( v \Rightarrow u \) and \( u \Rightarrow v \) in \( G \).
(2) If $u, v \in C$, then we have $v \leadsto u$ and $u \leadsto v$ in $G$. 

Sketch of proof: 

(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS; 

(2) then $r.f > u.f$ and $r.f > v.f$ in 1st DFS; 

(3) the assumption in (1) also implies: 

• $r \leadsto u$ and $r \leadsto v$ in $G_T$; 

• that is, $u \leadsto r$ and $v \leadsto r$ in $G$; 

• then $u.f > r.f$ and $v.f > r.f$ in 1st DFS, 

which conflict with conclusions in (2), UNLESS $r \leadsto u$ and $r \leadsto v$ in $G$ also. 

(4) This means: through $r$, $v \leadsto u$ and $u \leadsto v$ in $G$. 

Chapter 22. Elementary graph algorithms
(2) If $u, v \in C$, then we have $v \preceq u$ and $u \preceq v$ in $G$.

Sketch of proof:
(2) If $u, v \in C$, then we have $v \sim u$ and $u \sim v$ in $G$.

Sketch of proof:
(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS;
(2) If \( u, v \in C \), then we have \( v \Rightarrow u \) and \( u \Rightarrow v \) in \( G \).

Sketch of proof:

(1) assume \( u, v \) belong to the same tree of root \( r \) in 2nd DFS;
(2) then \( r.f > u.f \); and \( r.f > v.f \) in 1st DFS;
(2) If $u, v \in C$, then we have $v \sim u$ and $u \sim v$ in $G$.

Sketch of proof:

(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS;
(2) then $r.f > u.f$; and $r.f > v.f$ in 1st DFS;
(3) the assumption in (1) also implies:
(2) If $u, v \in C$, then we have $v \Rightarrow u$ and $u \Rightarrow v$ in $G$.

Sketch of proof:

(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS;
(2) then $r.f > u.f$; and $r.f > v.f$ in 1st DFS;
(3) the assumption in (1) also implies:
   • $r \Rightarrow u$ and $r \Rightarrow v$ in $G^T$;
(2) If $u, v \in C$, then we have $v \preceq u$ and $u \preceq v$ in $G$.

Sketch of proof:

(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS;
(2) then $r.f > u.f$; and $r.f > v.f$ in 1st DFS;
(3) the assumption in (1) also implies:
   • $r \preceq u$ and $r \preceq v$ in $G^T$;
   • that is, $u \preceq r$ and $v \preceq r$ in $G$;
(2) If $u, v \in C$, then we have $v \rightsquigarrow u$ and $u \rightsquigarrow v$ in $G$.

Sketch of proof:

(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS;
(2) then $r.f > u.f$; and $r.f > v.f$ in 1st DFS;
(3) the assumption in (1) also implies:
   • $r \rightsquigarrow u$ and $r \rightsquigarrow v$ in $G^T$;
   • that is, $u \rightsquigarrow r$ and $v \rightsquigarrow r$ in $G$;
   • then $u.f > r.f$ and
(2) If \( u, v \in C \), then we have \( v \mapsto u \) and \( u \mapsto v \) in \( G \).

Sketch of proof:

(1) assume \( u, v \) belong to the same tree of root \( r \) in 2nd DFS;
(2) then \( r.f > u.f \); and \( r.f > v.f \) in 1st DFS;
(3) the assumption in (1) also implies:
   • \( r \mapsto u \) and \( r \mapsto v \) in \( G^T \);
   • that is, \( u \mapsto r \) and \( v \mapsto r \) in \( G \);
   • then \( u.f > r.f \) and
     \( v.f > r.f \) in 1st DFS,
(2) If $u, v \in C$, then we have $v \leadsto u$ and $u \leadsto v$ in $G$.

Sketch of proof:

(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS;
(2) then $r.f > u.f$; and $r.f > v.f$ in 1st DFS;
(3) the assumption in (1) also implies:
   - $r \leadsto u$ and $r \leadsto v$ in $G^T$;
   - that is, $u \leadsto r$ and $v \leadsto r$ in $G$;
   - then $u.f > r.f$ and $v.f > r.f$ in 1st DFS,
     which conflict with conclusions in (2),
(2) If $u, v \in C$, then we have $v \leadsto u$ and $u \leadsto v$ in $G$.

Sketch of proof:

(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS;
(2) then $r.f > u.f$; and $r.f > v.f$ in 1st DFS;
(3) the assumption in (1) also implies:
   - $r \leadsto u$ and $r \leadsto v$ in $G^T$;
   - that is, $u \leadsto r$ and $v \leadsto r$ in $G$;
   - then $u.f > r.f$ and $v.f > r.f$ in 1st DFS,
   which conflict with conclusions in (2), UNLESS
(2) If $u, v \in C$, then we have $v \leadsto u$ and $u \leadsto v$ in $G$.

Sketch of proof:

(1) assume $u, v$ belong to the same tree of root $r$ in 2nd DFS;
(2) then $r.f > u.f$; and $r.f > v.f$ in 1st DFS;
(3) the assumption in (1) also implies:
    • $r \leadsto u$ and $r \leadsto v$ in $G^T$;
    • that is, $u \leadsto r$ and $v \leadsto r$ in $G$;
    • then $u.f > r.f$ and
      • $v.f > r.f$ in 1st DFS,
      which conflict with conclusions in (2), UNLESS
      $r \leadsto u$ and $r \leadsto v$ in $G$ also.
(2) If $u, v \in C$, then we have $v \rightsquigarrow u$ and $u \rightsquigarrow v$ in $G$.

Sketch of proof:

(1) assume $u$, $v$ belong to the same tree of root $r$ in 2nd DFS;
(2) then $r.f > u.f$; and $r.f > v.f$ in 1st DFS;
(3) the assumption in (1) also implies:
    • $r \rightsquigarrow u$ and $r \rightsquigarrow v$ in $G^T$;
    • that is, $u \rightsquigarrow r$ and $v \rightsquigarrow r$ in $G$;
    • then $u.f > r.f$ and $v.f > r.f$ in 1st DFS,
      which conflict with conclusions in (2), UNLESS $r \rightsquigarrow u$ and $r \rightsquigarrow v$ in $G$ also.
(4) This means: through $r$, $v \rightsquigarrow u$ and $u \rightsquigarrow v$ in $G$. 
Chapter 22. Elementary graph algorithms

Reachability Problem

Input: \(G = (V,E)\), and \(s,t \in V\);
Output: YES if and only there is a path \(s \rightarrow t\) in \(G\).

- The problem can be solved with DFS and BFS by search on the graph from \(s\) until \(t\) shows up.
- Linear time \(O(|E| + |V|)\). Can we do better?
- But first answer the following question: Can you write an SQL program to solve Reachability?
- It appears that a loop is needed to solve Reachability. Why?
- Inherent difficulty in parallel computation. P-complete, it cannot be solved in time \(O(\log n)\) even if \(\Theta(n)\) CPUs are used.
Chapter 22. Elementary graph algorithms

Reachability Problem

Input: \( G = (V, E) \), and \( s, t \in V \); Output: YES if and only there is a path \( s \to t \) in \( G \).

- The problem can be solved with DFS and BFS by search on the graph from \( s \) until \( t \) shows up.
- Linear time \( O(|E| + |V|) \). Can we do better?
- But first answer the following question: Can you write an SQL program to solve Reachability?
- It appears that a loop is needed to solve Reachability. Why?
  - Inherent difficulty in parallel computation.
  - \( P \)-complete, it cannot be solved in time \( O(\log n) \) even if \( \Theta(n) \) CPUs are used.
Reachability Problem

Input: $G = (V, E)$, and $s, t \in V$;
Reachability Problem

**Input:** $G = (V, E)$, and $s, t \in V$;

**Output:** YES if and only there is a path $s \leadsto t$ in $G$.
Reachability Problem

**Input:** $G = (V, E)$, and $s, t \in V$;

**Output:** YES if and only there is a path $s \leadsto t$ in $G$.

- The problem can be solved with DFS and BFS.

- The problem can be solved with DFS and BFS.

- Can we do better?

  - But first answer the following question:

    - Can you write an SQL program to solve Reachability?

      - It appears that a loop is needed to solve Reachability.

        - Why?

          - Inherent difficulty in parallel computation.

          - P-complete, it cannot be solved in time $O(\log n)$ even if $\Theta(n)$ CPUs are used.
Reachability Problem

**Input:** $G = (V, E)$, and $s, t \in V$;
**Output:** YES if and only there is a path $s \sim t$ in $G$.

- The problem can be solved with DFS and BFS by search on the graph from $s$ until $t$ shows up.

Linear time $O(|E| + |V|)$. Can we do better?

- But first answer the following question: Can you write an SQL program to solve Reachability?

It appears that a loop is needed to solve Reachability. Why?

Inherent difficulty in parallel computation. P-complete, it cannot be solved in time $O(\log n)$ even if $\Theta(n)$ CPUs are used.
Reachability Problem

**Input:** $G = (V, E)$, and $s, t \in V$.

**Output:** YES if and only there is a path $s \leadsto t$ in $G$.

- The problem can be solved with DFS and BFS by search on the graph from $s$ until $t$ shows up. Linear time $O(|E| + |V|)$. Can we do better?

But first answer the following question: Can you write an SQL program to solve Reachability? Why? Inherent difficulty in parallel computation. P-complete, it cannot be solved in time $O(\log n)$ even if $\Theta(n)$ CPUs are used.
Reachability Problem

**INPUT:** $G = (V, E)$, and $s, t \in V$;

**OUTPUT:** YES if and only there is a path $s \sim t$ in $G$.

- The problem can be solved with DFS and BFS by search on the graph from $s$ until $t$ shows up. Linear time $O(|E| + |V|)$. Can we do better?

- But first answer the following question:
  Can you write an SQL program to solve Reachability?
Reachability Problem

**INPUT:** \( G = (V, E) \), and \( s, t \in V \).

**OUTPUT:** \( \text{YES} \) if and only if there is a path \( s \rightsquigarrow t \) in \( G \).

- The problem can be solved with DFS and BFS by search on the graph from \( s \) until \( t \) shows up. Linear time \( O(|E| + |V|) \). Can we do better?

- But first answer the following question:
  
  *Can you write an SQL program to solve Reachability?*

- It appears that a loop is needed to solve Reachability.
Reachability Problem

**Input:** $G = (V, E)$, and $s, t \in V$;

**Output:** YES if and only there is a path $s \sim t$ in $G$.

- The problem can be solved with DFS and BFS by search on the graph from $s$ until $t$ shows up.
  Linear time $O(|E| + |V|)$. Can we do better?

- But first answer the following question:
  Can you write an SQL program to solve Reachability?

- It appears that a loop is needed to solve Reachability. Why?
Reachability Problem

**Input:** \( G = (V, E) \), and \( s, t \in V \);  
**Output:** YES if and only there is a path \( s \leadsto t \) in \( G \).

- The problem can be solved with DFS and BFS  
  by search on the graph from \( s \) until \( t \) shows up.  
  Linear time \( O(|E| + |V|) \). Can we do better?

- But first answer the following question:  
  Can you write an SQL program to solve Reachability?

- It appears that a loop is needed to solve Reachability. **Why?**  
  Inherent difficulty in parallel computation.
Reachability Problem

Input: \( G = (V, E) \), and \( s, t \in V \);  
Output: YES if and only there is a path \( s \leadsto t \) in \( G \).

- The problem can be solved with DFS and BFS by search on the graph from \( s \) until \( t \) shows up.  
  Linear time \( O(|E| + |V|) \). Can we do better?

- But first answer the following question:  
  Can you write an SQL program to solve Reachability?

- It appears that a loop is needed to solve Reachability. Why?  
  Inherent difficulty in parallel computation.

  \textbf{P-complete}, it cannot be solved in time \( O(\log n) \) even if \( \Theta(n) \) CPUs are used.
Chapter 23. Minimum Spanning Trees

A spanning tree of a graph $G = (V, E)$ is a tree as a subgraph in $G$ which contains all vertices in $V$.

A minimum spanning tree (MST) of an edge-weighted graph $G$ is a spanning tree with the least edge weight sum.
Chapter 23. Minimum Spanning Trees

A spanning tree of a graph $G = (V, E)$ is a tree as a subgraph in $G$ which contains all vertices in $V$. A minimum spanning tree (MST) of an edge-weighted graph $G$ is a spanning tree with the least edge weight sum.
A spanning tree of a graph $G = (V, E)$ is a tree as a subgraph in $G$ which contains all vertices in $V$. 

\begin{center}
\begin{tikzpicture}
  \foreach \x in {0,1,2,3,4,5,6,7,8} {
    \foreach \y in {0,1,2,3,4,5,6,7} {
      \draw[gray,thick] \x-\y;
    }
  }
  \foreach \y in {0,1,2,3,4,5,6,7} {
    \filldraw[red] 0-\y circle(2pt);
  }
  \foreach \x in {0,1,2,3,4,5,6,7} {
    \foreach \y in {0,1,2,3,4,5,6,7} {
      \draw[blue,thick] \x-\y;
    }
  }
\end{tikzpicture}
\end{center}
Chapter 23. Minimum Spanning Trees

- A spanning tree of a graph $G = (V, E)$ is a tree as a subgraph in $G$ which contains all vertices in $V$. 

![Graph Image]
Chapter 23. Minimum Spanning Trees

• A spanning tree of a graph $G = (V, E)$ is a tree as a subgraph in $G$ which contains all vertices in $V$.

• A minimum spanning tree (MST) of an edge-weighted graph $G$ is a spanning tree with the least edge weight sum.
Chapter 23. Minimum Spanning Trees

Chapter 23. Minimum Spanning Trees

- A spanning tree of a graph $G = (V, E)$ is a tree as a subgraph in $G$ which contains all vertices in $V$.

- A minimum spanning tree (MST) of an edge-weighted graph $G$ is a spanning tree with the least edge weight sum.
Chapter 23. Minimum Spanning Trees

• A spanning tree of a graph \( G = (V, E) \) is a tree as a subgraph in \( G \) which contains all vertices in \( V \).

• A minimum spanning tree (MST) of an edge-weighted graph \( G \) is a spanning tree with the least edge weight sum.
Chapter 23. Minimum Spanning Trees

The MST problem
Chapter 23. Minimum Spanning Trees

The MST problem

**Input**: connected, undirected graph $G = (V, E)$ with weight $w : E \rightarrow R$, 

**Output**: a spanning tree $T = (V, E')$ such that $W(T) = \sum_{(u,v) \in E'} w(u,v)$ is the minimum.
The MST problem

**Input:** connected, undirected graph $G = (V, E)$ with weight $w : E \to \mathbb{R}$,

**Output:** a spanning tree $T = (V, E')$ such that

$$W(T) = \sum_{(u,v) \in E'} w(u,v)$$

is the minimum
Chapter 23. Minimum Spanning Trees

The MST problem

**Input:** connected, undirected graph $G = (V, E)$ with weight $w : E \rightarrow R$,

**Output:** a spanning tree $T = (V, E')$ such that

$$W(T) = \sum_{(u,v) \in E'} w(u, v)$$ is the minimum

We will introduce two **greedy algorithms**: (1) Kruskal’s and (2) Prim’s
Chapter 23. Minimum Spanning Trees

The MST problem

**Input**: connected, undirected graph $G = (V, E)$ with weight $w : E \rightarrow R$,

**Output**: a spanning tree $T = (V, E')$ such that

$$W(T) = \sum_{(u,v) \in E'} w(u,v)$$

is the minimum.

We will introduce two **greedy algorithms**: (1) Kruskal’s and (2) Prim’s

- They have the same generic process to grow a spanning tree;
Chapter 23. Minimum Spanning Trees

The MST problem

**Input:** connected, undirected graph $G = (V, E)$ with weight $w : E \rightarrow R$,

**Output:** a spanning tree $T = (V, E')$ such that

$$W(T) = \sum_{(u,v) \in E'} w(u,v)$$

is the minimum

We will introduce two **greedy algorithms**: (1) Kruskal’s and (2) Prim’s

- They have the same generic process to grow a spanning tree;
- but differ in which edge to add the partially grown tree.
Chapter 23. Minimum Spanning Trees

Growing an MST

Generic MST \( (G, w) \) {
1. \( A = \emptyset \);
2. while \( A \) does not form a spanning tree
3. find an edge \((u,v)\) that is safe for \( A \)
4. \( A = A \cup \{(u,v)\} \)
5. return \( (A) \)

Loop invariant: \( A \) is always a subset of some MST;
Note: when the loop terminates, \( A \) is a MST.

safe edge: edge \((u,v)\) is safe for \( A \) if it does not violate the loop invariant,
i.e, \( A \cup \{(u,v)\} \) is a subset of some MST.
Growing an MST

A generic process to grow an MST.
**Growing an MST**

A generic process to grow an MST.

\[ \text{ GENERIC MST}(G, w) \quad \{ \text{ given graph } G \text{ and weight function } w \} \]
Chapter 23. Minimum Spanning Trees

Growing an MST

A generic process to grow an MST.

\textsc{Generic MST}(G, w) \{ given graph } G \text{ and weight function } w \} \\
1. \quad A = \emptyset;
Growing an MST

A generic process to grow an MST.

\[
\text{Generic MST}(G, w) \quad \{ \text{given graph } G \text{ and weight function } w \} \\
1. \quad A = \emptyset; \\
2. \quad \textbf{while } A \text{ does not form a spanning tree} \\
3. \quad \text{find an edge } (u, v) \text{ that is safe for } A
\]
Chapter 23. Minimum Spanning Trees

Growing an MST

A generic process to grow an MST.

\[
\text{GENERIC MST}(G, w) \quad \{ \text{given graph } G \text{ and weight function } w \} \\
1. \quad A = \emptyset; \\
2. \quad \textbf{while} A \text{ does not form a spanning tree} \\
3. \quad \text{find an edge } (u, v) \text{ that is safe for } A \\
4. \quad A = A \cup \{(u, v)\}
\]
Growing an MST

A generic process to grow an MST.

**Generic MST** \( (G, w) \) \{ given graph \( G \) and weight function \( w \) \}

1. \( A = \emptyset \);
2. **while** \( A \) does not form a spanning tree
3. find an edge \((u, v)\) that is safe for \( A \)
4. \( A = A \cup \{(u, v)\} \)
5. **return** \( (A) \)
Chapter 23. Minimum Spanning Trees

Growing an MST

A generic process to grow an MST.

**Generic MST**\((G,w)\) \{ given graph \(G\) and weight function \(w\) \}

1. \(A = \emptyset\);
2. while \(A\) does not form a spanning tree
3. find an edge \((u,v)\) that is safe for \(A\)
4. \(A = A \cup \{(u,v)\}\)
5. return \((A)\)

Loop invariant:
Chapter 23. Minimum Spanning Trees

Growing an MST

A generic process to grow an MST.

\textbf{Generic MST}(G, w) \quad \{ \text{given graph } G \text{ and weight function } w \} \\
1. \quad A = \emptyset; \\
2. \quad \textbf{while } A \text{ does not form a spanning tree} \\
3. \quad \text{find an edge } (u, v) \text{ that is safe for } A \\
4. \quad A = A \cup \{(u, v)\} \\
5. \quad \textbf{return } (A)

Loop invariant: \( A \) is always a subset of some MST;
Chapter 23. Minimum Spanning Trees

Growing an MST

A generic process to grow an MST.

\textbf{Generic MST} \((G, w)\) \{ given graph \(G\) and weight function \(w\) \}

1. \(A = \emptyset\);
2. \textbf{while} \(A\) does not form a spanning tree
3. find an edge \((u, v)\) that is safe for \(A\)
4. \(A = A \cup \{(u, v)\}\)
5. \textbf{return} \((A)\)

Loop invariant: \(A\) is always a subset of some MST;
Chapter 23. Minimum Spanning Trees

Growing an MST

A generic process to grow an MST.

**Generic MST** \((G, w)\) \{ given graph \(G\) and weight function \(w\) \}

1. \(A = \emptyset\);
2. **while** \(A\) does not form a spanning tree
3. find an edge \((u, v)\) that is **safe** for \(A\)
4. \(A = A \cup \{(u, v)\}\)
5. **return** \((A)\)

Loop invariant: \(A\) is always a subset of some MST;

Note: when the loop terminates, \(A\) is a MST.
Growing an MST

A generic process to grow an MST.

**Generic MST** \( (G,w) \) \{ given graph \( G \) and weight function \( w \) \}

1. \( A = \emptyset \);
2. while \( A \) does not form a spanning tree
3. find an edge \((u,v)\) that is safe for \( A \)
4. \( A = A \cup \{(u,v)\} \)
5. return \((A)\)

Loop invariant: \( A \) is always a subset of some MST;

Note: when the loop terminates, \( A \) is a MST.

safe edge:
Growing an MST

A generic process to grow an MST.

** Generic MST \((G, w)\) \{ given graph \(G\) and weight function \(w\) \}

1. \(A = \emptyset\);
2. \textbf{while} \(A\) does not form a spanning tree
3. find an edge \((u, v)\) that is \textbf{safe} for \(A\)
4. \(A = A \cup \{(u, v)\}\)
5. \textbf{return} \((A)\)

Loop invariant: \(A\) is always a subset of some MST;

Note: when the loop terminates, \(A\) is a MST.

safe edge:

edge \((u, v)\) is safe for \(A\) if \textbf{does not violate the loop invariant},
Chapter 23. Minimum Spanning Trees

Growing an MST

A generic process to grow an MST.

**Generic MST**($G, w$) { given graph $G$ and weight function $w$ }

1. $A = \emptyset$;
2. while $A$ does not form a spanning tree
3. find an edge $(u, v)$ that is safe for $A$
4. $A = A \cup \{(u, v)\}$
5. return $(A)$

Loop invariant: $A$ is always a subset of some MST;

Note: when the loop terminates, $A$ is a MST.

safe edge:
edge $(u, v)$ is safe for $A$ if does not violate the loop invariant, i.e, $A \cup \{(u, v)\}$ is a subset of some MST.
We first need some terminologies

• cut: $(S, V - S)$, a partition of $V$

• crossing: $(u, v)$ crosses cut $(S, V - S)$ if $u$ and $v$ are in $S$ and $V - S$, respectively
Chapter 23. Minimum Spanning Trees

We first need some terminologies

- **cut**: \((S, V - S)\), a partition of \(V\)
Chapter 23. Minimum Spanning Trees

We first need some terminologies

- **cut**: \((S, V-S)\), a partition of \(V\)
Chapter 23. Minimum Spanning Trees

We first need some terminologies

- **cut**: \((S, V - S)\), a partition of \(V\)

- **crossing**: \((u, v)\) crosses cut \((S, V - S)\)
  if \(u\) and \(v\) are in \(S\) and \(V - S\), respectively
Chapter 23. Minimum Spanning Trees

Some more terminologies
Some more terminologies

- **respect**: a cut respects a set $A$ of edges if no edge in $A$ crosses the cut.
Some more terminologies

- **respect**: a cut respects a set $A$ of edges if no edge in $A$ crosses the cut.

- **light edge**: an edge is a light edge crossing a cut if its weight is the minimum of any edge that crosses the cut.
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$. 

...
Theorem 23.1 Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$. 

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$. 

**Sketch of proof:**

1. If $A \cup \{(u,v)\}$ forms a cycle there must have been another edge in $A$ that crosses cut $(S, V \setminus S)$ (WHY?), implying the cut did not respect $A$.
   Contradicts.

2. Assume that some MST $T$, $A \subset T$.
   First, $T \cup \{(u,v)\}$ forms a circle! (WHY?)
   There must be another edge $(x,y)$ that crosses the cut $(S, V \setminus S)$.
   Since $(u,v)$ is a light edge, $T' = T \setminus \{(x,y)\} \cup \{(u,v)\}$ is an MST.
   Now $A \cup \{(u,v)\} \subseteq T'$ because $(x,y) \not\in A$ (otherwise, the cut would not respect $A$).
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$. 
Theorem 23.1 Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Theorem 23.1 Let $G = (V, E)$. Let $A \subseteq E$, contained in some MST for $G$. Let $(S, V - S)$ be any cut of $G$ that respects $A$. Let $(u, v)$ be a light edge crossing the cut. Then edge $(u, v)$ is safe for $A$. 

Sketch of proof:
(1) If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$ that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$. Contradicts.

(2) Assume that some MST $T$, $A \subset T$. First, $T \cup \{(u, v)\}$ forms a cycle! Why? There must be another edge $(x, y)$ that crosses the cut $(S, V - S)$. Since $(u, v)$ is light edge, $T' = T - \{(x, y)\} \cup \{(u, v)\}$ is an MST. Now $A \cup \{(u, v)\} \subseteq T'$ because $(x, y) \not\in A$ (otherwise, the cut would not respect $A$).
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is **safe** for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$. Let $A \subseteq E$, contained in some MST for $G$. Let $(S, V - S)$ be any cut of $G$ that respects $A$. Let $(u, v)$ be a light edge crossing the cut. Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:

1. $(u, v)$ does not form a cycle;
2. $A$, after including $(u, v)$, is still a subset of some MST.
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
(2) $A$, after including $(u, v)$, is still a subset of some MST.

**Sketch of proof:**
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
(2) $A$, after including $(u, v)$, is still a subset of some MST.

**Sketch of proof:**

(1) If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$
that crosses cut $(S, V - S)$ (WHY?),
**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
1. $(u, v)$ does not form a cycle;
2. $A$, after including $(u, v)$, is still a subset of some MST.

**Sketch of proof:**

1. If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$
   that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$.  
   **Contradicts.**
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
(2) $A$, after including $(u, v)$, is still a subset of some MST.

**Sketch of proof:**

(1) If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$
that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$.
Contradicts.

(2) Assume that some MST $T$, $A \subset T$. 

Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
(2) $A$, after including $(u, v)$, is still a subset of some MST.

**Sketch of proof:**

(1) If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$
that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$.
Contradicts.

(2) Assume that some MST $T$, $A \subset T$.
First, $T \cup \{(u, v)\}$ forms a circle!
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
(2) $A$, after including $(u, v)$, is still a subset of some MST.

**Sketch of proof:**

(1) If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$ that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$. **Contradicts.**

(2) Assume that some MST $T$, $A \subset T$.
First, $T \cup \{(u, v)\}$ forms a circle! **Why?**
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
(2) $A$, after including $(u, v)$, is still a subset of some MST.

**Sketch of proof:**

(1) If $A \cup \{(u,v)\}$ forms a cycle there must have been another edge in $A$ that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$. Contradicts.

(2) Assume that some MST $T$, $A \subset T$.
First, $T \cup \{(u,v)\}$ forms a circle! Why?

There must be another edge $(x, y)$ cross the cut $(S, V - S)$. 
Chapter 23. Minimum Spanning Trees

Theorem 23.1 Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
(2) $A$, after including $(u, v)$, is still a subset of some MST.

Sketch of proof:

(1) If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$
that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$.
Contradicts.

(2) Assume that some MST $T$, $A \subset T$.
First, $T \cup \{(u, v)\}$ forms a circle! Why?
There must be another edge $(x, y)$ cross the cut $(S, V - S)$.
Since $(u, v)$ is light edge, $T' = T - \{(x, y)\} \cup \{(u, v)\}$ is an MST.
Theorem 23.1 Let $G = (V, E)$. Let $A \subseteq E$, contained in some MST for $G$. Let $(S, V - S)$ be any cut of $G$ that respects $A$. Let $(u, v)$ be a light edge crossing the cut. Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
(1) $(u, v)$ does not form a cycle;
(2) $A$, after including $(u, v)$, is still a subset of some MST.

Sketch of proof:

(1) If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$ that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$. Contradicts.

(2) Assume that some MST $T$, $A \subset T$.
First, $T \cup \{(u, v)\}$ forms a circle! Why?

There must be another edge $(x, y)$ cross the cut $(S, V - S)$. Since $(u, v)$ is light edge, $T' = T - \{(x, y)\} \cup \{(u, v)\}$ is an MST.

Now $A \cup \{(u, v)\} \subseteq T'$
Chapter 23. Minimum Spanning Trees

**Theorem 23.1** Let $G = (V, E)$.
Let $A \subseteq E$, contained in some MST for $G$.
Let $(S, V - S)$ be any cut of $G$ that respects $A$.
Let $(u, v)$ be a light edge crossing the cut.
Then edge $(u, v)$ is safe for $A$.

For the theorem, we need to prove:
1. $(u, v)$ does not form a cycle;
2. $A$, after including $(u, v)$, is still a subset of some MST.

**Sketch of proof:**

1. If $A \cup \{(u, v)\}$ forms a cycle there must have been another edge in $A$ that crosses cut $(S, V - S)$ (WHY?), implying the cut did not respect $A$. **Contradicts.**

2. Assume that some MST $T$, $A \subset T$.
First, $T \cup \{(u, v)\}$ forms a circle! **Why?**

   There must be another edge $(x, y)$ cross the cut $(S, V - S)$.
   Since $(u, v)$ is light edge, $T' = T - \{(x, y)\} \cup \{(u, v)\}$ is an MST.

Now $A \cup \{(u, v)\} \subseteq T'$ **because** $(x, y) \not\in A$ (otherwise, the cut would not respect $A$.****
Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the GENERIC MST algorithm work.
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the GENERIC MST algorithm work.

- specific algorithms can be produced from GENERIC MST based on how the set $A$ is grown.

MST-Kruskal $((G,w))$

1. $A = \emptyset$
2. for each vertex $v \in G.V$
3. Make-Set ($v$)
4. sort edges in $E$ into non-decreasing order by their weight $w$
5. for each edge $\{(u,v)\} \in E$, taken in the order
6. if Find Set ($u$) $\neq$ Find Set ($v$)
7. $A = A \cup \{(u,v)\}$
8. Union ($u,v$)
9. return ($A$)
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the Generic MST algorithm work.

• specific algorithms can be produced from Generic MST based on how the set $A$ is grown.

• $A$ may always be a tree (Prim’s algorithm)
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the **Generic MST** algorithm work.

- specific algorithms can be produced from **Generic MST** based on how the set $A$ is grown.

- $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

**MST-Kruskal** $(G,w)$

1. $A = \emptyset$
2. for each vertex $v \in G.V$
3. Make-Set ($v$)
4. sort edges in $E$ into non-decreasing order by their weight
5. for each edge $(u,v) \in E$, taken in the order
6. if Find Set ($u$) $\neq$ Find Set ($v$)
7. $A = A \cup \{(u,v)\}$
8. Union ($u,v$)
9. return ($A$)
Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the \textsc{Generic MST} algorithm work.

- specific algorithms can be produced from \textsc{Generic MST} based on how the set $A$ is grown.

- $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

\textsc{MST-Kruskal}(G, w)
Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the **Generic MST** algorithm work.

- specific algorithms can be produced from **Generic MST** based on how the set $A$ is grown.
- $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

**MST-Kruskal** ($G, w$)
1. $A = \emptyset$;
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the Generic MST algorithm work.

• specific algorithms can be produced from Generic MST based on how the set $A$ is grown.

• $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

$\text{MST-Kruskal}(G, w)$
1. $A = \emptyset$;
2. for each vertex $v \in G.V$
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the Generic MST algorithm work.

- specific algorithms can be produced from Generic MST based on how the set $A$ is grown.
- $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

MST-Kruskal($G, w$)
1. $A = \emptyset$;
2. for each vertex $v \in G.V$
3. MAKE-SET($v$)
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the **Generic MST** algorithm work.

- specific algorithms can be produced from **Generic MST** based on how the set $A$ is grown.
- $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

**MST-Kruskal**($G, w$)
1. $A = \emptyset$;
2. for each vertex $v \in G.V$
3.  **Make-Set**(v)
4.  **sort** edges in $E$ into non-decreasing order by their weight $w$

Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the Generic MST algorithm work.

• specific algorithms can be produced from Generic MST based on how the set $A$ is grown.

• $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

**MST-Kruskal**($G, w$)
1. $A = \emptyset$;
2. for each vertex $v \in G.V$
3. Make-Set$(v)$
4. sort edges in $E$ into non-decreasing order by their weight $w$
5. for each edge $(u, v) \in E$, taken in the order
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the Generic MST algorithm work.

- specific algorithms can be produced from Generic MST based on how the set $A$ is grown.

- $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

MST-Kruskal($G, w$)
1. $A = \emptyset$;
2. for each vertex $v \in G.V$
3.   Make-Set$(v)$
4. sort edges in $E$ into non-decreasing order by their weight $w$
5. for each edge $(u, v) \in E$, taken in the order
6.   if Find Set $(u) \neq$ Find Set$(v)$
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the Generic MST algorithm work.

• specific algorithms can be produced from Generic MST based on how the set $A$ is grown.

• $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

MST-Kruskal($G, w$)
1. $A = \emptyset$;
2. for each vertex $v \in G.V$
3. \hspace{1em} Make-Set($v$)
4. sort edges in $E$ into non-decreasing order by their weight $w$
5. for each edge $(u, v) \in E$, taken in the order
6. \hspace{1em} if Find Set ($u$) $\neq$ Find Set($v$)
7. \hspace{2em} $A = A \cup \{(u, v)\}$
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the Generic MST algorithm work.

• specific algorithms can be produced from Generic MST based on how the set $A$ is grown.

• $A$ may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

MST-Kruskal$(G, w)$
1. $A = \emptyset$;
2. for each vertex $v \in G.V$
3. \hspace{1em} Make-Set$(v)$
4. sort edges in $E$ into non-decreasing order by their weight $w$
5. for each edge $(u, v) \in E$, taken in the order
6. \hspace{1em} if Find Set $(u) \neq$ Find Set$(v)$
7. \hspace{2em} $A = A \cup \{(u, v)\}$
8. Union$(u, v)$
Chapter 23. Minimum Spanning Trees

Theorem 23.1 gives a sufficient information for how to identify a safe edge to make the \texttt{Generic MST} algorithm work.

- specific algorithms can be produced from \texttt{Generic MST} based on how the set \( A \) is grown.

- \( A \) may always be a tree (Prim’s algorithm) or could be a forest (Kruskal’s algorithm).

\texttt{MST-Kruskal}(G, w)

\begin{enumerate}
\item \( A = \emptyset \);
\item \textbf{for} each vertex \( v \in G.V \)
\item \textsc{Make-Set}(v)
\item \textbf{sort} edges in \( E \) into non-decreasing order by their weight \( w \)
\item \textbf{for} each edge \( (u, v) \in E \), taken in the order
\item \textbf{if} \textsc{Find Set}(u) \neq \textsc{Find Set}(v)
\item \( A = A \cup \{(u, v)\} \)
\item \textsc{Union}(u, v)
\item \textbf{return} \( (A) \)
\end{enumerate}
Chapter 23. Minimum Spanning Trees

Execution of Kruskal's algorithm for MST

disjoint sets
[A] [B] [C] [D] [E] [F] [G] [H]
[A] [B] [C] [D] [E] [F] [G] [H]
[A] [B] [C] [D] [E] [F, G] [H]
[A] [B] [D] [E] [C, F, G] [H]
[A] [D] [E] [B, C, F, G] [H]
[D] [E] [A, B, C, F, G] [H]
[D, E] [A, B, C, F, G] [H]
[D, E, H] [A, B, C, F, G]
[A, B, C, D, E, F, G, H]
Chapter 23. Minimum Spanning Trees

At each iteration of the for loop, e.g., identify

\[ \{A, B, C, F, G\} \]

cut that respects \( A \):
\[ \{A, B, C, D, F, G\} \]
\[ \{E, H\} \]
light edge \( (D, E) \) crosses the cut;

\[ \{D, E\} \]
\[ \{A, B, C, F, G\} \]
\[ \{H\} \]
Chapter 23. Minimum Spanning Trees

At each iteration of the for loop, e.g., identify

• \( A = \{(A, F), (B, F), (C, G), (F, G), (D, E)\} \), cut that respects \( A \):
  \( S = \{A, B, C, D, F, G\} \), \( V - S = \{E, H\} \), light edge \( (D, E) \) crosses the cut;

• \( A = \{(A, F), (B, F), (C, G), (F, G), (D, E)\} \), cut that respects \( A \):
  \( S = \{A, B, C, F, G, H\} \), \( V - S = \{D, E\} \), light edge \( (E, H) \) crosses the cut;

\( A = \)
Chapter 23. Minimum Spanning Trees

At each iteration of the for loop, e.g., identify

- $A = \{(A, F), (B, F), (C, G), (F, G)\}$,
Chapter 23. Minimum Spanning Trees

At each iteration of the for loop, e.g., identify

- \( A = \{(A, F), (B, F), (C, G), (F, G)\} \),
- cut that respects \( A \): \( S = \{A, B, C, D, F, G\} \), \( V-S = \{E, H\} \),

[Diagrams showing the graphs with edges and cut sets]
Chapter 23. Minimum Spanning Trees

At each iteration of the for loop, e.g., identify

$A = \{(A, F), (B, F), (C, G), (F, G)\}$,

cut that respects $A$: $S = \{A, B, C, D, F, G\}$, $V - S = \{E, H\}$,

light edge $(D, E)$ crosses the cut;
Chapter 23. Minimum Spanning Trees

At each iteration of the `for` loop, e.g., identify

\[ A = \{(A, F), (B, F), (C, G), (F, G)\}, \]

\[ S = \{A, B, C, D, F, G\}, \quad V - S = \{E, H\}, \]

light edge \((D, E)\) crosses the cut;

\[ A = \]

\[ [D, E] \quad [A, B, C, F, G] \quad [H] \]

\[ [D, E] \quad [A, B, C, F, G] \quad [H] \]
Chapter 23. Minimum Spanning Trees

At each iteration of the for loop, e.g., identify

- \( A = \{(A, F), (B, F), (C, G), (F, G)\} \),
  cut that respects \( A \): \( S = \{A, B, C, D, F, G\} \), \( V-S = \{E, H\} \),
  light edge \((D, E)\) crosses the cut;

- \( A = \{(A, F), (B, F), (C, G), (F, G), (D, E)\} \),
Chapter 23. Minimum Spanning Trees

At each iteration of the for loop, e.g., identify

- $A = \{(A, F), (B, F), (C, G), (F, G)\}$,
  cut that respects $A$: $S = \{A, B, C, D, F, G\}$, $V-S = \{E, H\}$,
  light edge $(D, E)$ crosses the cut;

- $A = \{(A, F), (B, F), (C, G), (F, G), (D, E)\}$,
  cut that respect $A$: $S = \{A, B, C, F, G, H\}$, $V-S = \{D, E\}$,
At each iteration of the for loop, e.g., identify

- $\mathcal{A} = \{(A, F), (B, F), (C, G), (F, G)\}$,
  cut that respects $\mathcal{A}$: $S = \{A, B, C, D, F, G\}$, $V-S = \{E, H\}$, light edge $(D, E)$ crosses the cut;

- $\mathcal{A} = \{(A, F), (B, F), (C, G), (F, G), (D, E)\}$,
  cut that respect $\mathcal{A}$: $S = \{A, B, C, F, G, H\}$, $V-S = \{D, E\}$, light edge $(E, H)$ crosses the cut;
Chapter 23. Minimum Spanning Trees

The Kruskal’s algorithm uses disjoint-set data structures (where elements are partitioned into disjoint sets)
Chapter 23. Minimum Spanning Trees

The Kruskal’s algorithm uses disjoint-set data structures (where elements are partitioned into disjoint sets)

- **Make Set**($x$): create a set of single element $x$;
Chapter 23. Minimum Spanning Trees

The Kruskal’s algorithm uses disjoint-set data structures (where elements are partitioned into disjoint sets)

- **Make Set**($x$): create a set of single element $x$;
- **Find Set**($x$): identify the set that contains element $x$;

Implementations (left: linked lists, Right: disjoint-set forest)

Time complexity:

$O(\log n)$ for Make Set($x$), Find Set($x$), Union($x,y$)

Time complexity of Kruskal's algorithm: $O(|E| \log |V|) + |V|$. 
Chapter 23. Minimum Spanning Trees

The Kruskal’s algorithm uses disjoint-set data structures (where elements are partitioned into disjoint sets)

- **Make Set**($x$): create a set of single element $x$;
- **Find Set**($x$): identify the set that contains element $x$;
- **Union**($x, y$): union the two sets containing $x$ and $y$ into one;

Implementations (left: linked lists, Right: disjoint-set forest)

Time complexity: $O(\log n)$ for $\text{Make Set}(x)$, $\text{Find Set}(x)$, $\text{Union}(x, y)$ with disjoint-set forest implementation.

Time complexity of Kruskal’s algorithm: $O(|E| \log |V| + |V|)$. 
Chapter 23. Minimum Spanning Trees

The Kruskal’s algorithm uses disjoint-set data structures (where elements are partitioned into disjoint sets)

- **Make Set**($x$): create a set of single element $x$;
- **Find Set**($x$): identify the set that contains element $x$;
- **Union**($x$, $y$): union the two sets containing $x$ and $y$ into one;

Implementations (left: linked lists, Right: disjoint-set forest)
Chapter 23. Minimum Spanning Trees

The Kruskal’s algorithm uses disjoint-set data structures (where elements are partitioned into disjoint sets)

- **Make Set**($x$): create a set of single element $x$;
- **Find Set**($x$): identify the set that contains element $x$;
- **Union**($x, y$): union the two sets containing $x$ and $y$ into one;

Implementations (left: linked lists, Right: disjoint-set forest)

Time complexity:
Chapter 23. Minimum Spanning Trees

The Kruskal’s algorithm uses disjoint-set data structures (where elements are partitioned into disjoint sets)

- **Make Set**(\(x\)): create a set of single element \(x\);
- **Find Set**(\(x\)): identify the set that contains element \(x\);
- **Union**(\(x, y\)): union the two sets containing \(x\) and \(y\) into one;

Implementations (left: linked lists, Right: disjoint-set forest)

Time complexity: \(O(\log n)\) for **Make Set**(\(x\)), **Find Set**(\(x\)), **Union**(\(x, y\)) with disjoint-set forest implementation.
Chapter 23. Minimum Spanning Trees

The Kruskal’s algorithm uses disjoint-set data structures (where elements are partitioned into disjoint sets)

- **Make Set** \( (x) \): create a set of single element \( x \);
- **Find Set** \( (x) \): identify the set that contains element \( x \);
- **Union** \( (x,y) \): union the two sets containing \( x \) and \( y \) into one;

Implementations (left: linked lists, Right: disjoint-set forest)

Time complexity: \( O(\log n) \) for **Make Set** \( (x) \), **Find Set** \( (x) \), **Union** \( (x,y) \) with disjoint-set forest implementation.

Time complexity of Kruskal’s algorithm: \( O(|E| \log |V| + |V|) \).
Chapter 23. Minimum Spanning Trees

\[ \text{MST-Prim}(G, w, r) \]
Chapter 23. Minimum Spanning Trees

MST-Prim\((G, w, r)\)
1. for each \(u \in G.V\)
Chapter 23. Minimum Spanning Trees

MST-Prim \( (G, w, r) \)
1. \textbf{for each} \( u \in G.V \)
2. \( u.key = \infty \) \{ \textit{u.key} is the \( u \)'s shortest distance to set \( A = V-Q \) \}
Chapter 23. Minimum Spanning Trees

MST-Prim\( (G, w, r) \)
1. for each \( u \in G.V \)
2. \( u.\text{key} = \infty \) \{ \( u.\text{key} \) is the \( u \)'s shortest distance to set \( A = V - Q \)\}
3. \( u.\pi = NULL \)
MST-Prim \((G, w, r)\)

1. for each \(u \in G.V\) 
2. \(u\.key = \infty\) \{ \(u\.key\) is the \(u\)'s shortest distance to set \(A = V-Q\)\} 
3. \(u\.\pi = NULL\) 
4. \(r\.key = 0\) \{ start from vertex \(r\) \} 

Running time \(O(|E| + |V| \log |V|)\).
Chapter 23. Minimum Spanning Trees

MST-Prim($G$, $w$, $r$)
1. for each $u \in G.V$
2. $u.key = \infty$ \{ $u.key$ is the $u$'s shortest distance to set $A = V - Q$ \}
3. $u.\pi = NULL$
4. $r.key = 0$ \{ start from vertex $r$ \}
5. $Q = G.V$ \{ establish priority queue $Q$ with $key$ values \}
Chapter 23. Minimum Spanning Trees

MST-Prim\((G, w, r)\)
1. for each \(u \in G.V\)
2. \(u.key = \infty\) \{ \(u.key\) is the \(u\)'s shortest distance to set \(A = V-Q\)\}
3. \(u.\pi = NULL\)
4. \(r.key = 0\) \{ start from vertex \(r\) \}
5. \(Q = G.V\) \{ establish priority queue \(Q\) wit key values\}
6. while \(Q \neq \emptyset\)
Chapter 23. Minimum Spanning Trees

MST-Prim\((G, w, r)\)
1. for each \(u \in G.V\) \{ \(u.key\) is the \(u\)'s shortest distance to set \(A = V-Q\)\}
2. \(u.key = \infty\)
3. \(u.\pi = NULL\)
4. \(r.key = 0\) \{ start from vertex \(r\) \}
5. \(Q = G.V\) \{ establish priority queue \(Q\) wit key values\}
6. while \(Q \neq \emptyset\)
7. \(u = \text{Extract Min}(Q)\)
Chapter 23. Minimum Spanning Trees

MST-Prim($G, w, r$)

1. for each $u \in G.V$
2. $u.key = \infty$ \hspace{1cm} \{ $u.key$ is the $u'$s shortest distance to set $A = V - Q$\}
3. $u.\pi = NULL$
4. $r.key = 0$ \hspace{1cm} \{ start from vertex $r$ \}
5. $Q = G.V$ \hspace{1cm} \{ establish priority queue $Q$ with key values\}
6. while $Q \neq \emptyset$
7. \hspace{0.5cm} $u = \text{Extract Min}(Q)$
8. \hspace{0.5cm} for each $v \in Adj[u]$
Chapter 23. Minimum Spanning Trees

MST-Prim\((G, w, r)\)

1. \textbf{for} each \(u \in G.V\)
2. \(u.\text{key} = \infty\) \{ \(u.\text{key}\) is the \(u\)'s shortest distance to set \(A = V-Q\)\}
3. \(u.\pi = NULL\)
4. \(r.\text{key} = 0\) \{ start from vertex \(r\) \}
5. \(Q = G.V\) \{ establish priority queue \(Q\) wit \text{key} values\}
6. \textbf{while} \(Q \neq \emptyset\)
7. \(u = \text{EXTRACT MIN}(Q)\)
8. \textbf{for} each \(v \in \text{Adj}[u]\)
9. \textbf{if} \(v \in Q\) and \(w(u, v) < v.\text{key}\) \{ for those not in \(A\), update distances\}
Chapter 23. Minimum Spanning Trees

MST-Prim\((G, w, r)\)
1. \textbf{for} each \(u \in G.V\) \{ \(u\).\textit{key} is the \(u\)'s shortest distance to set \(A = V-Q\) \}
2. \(u\).\textit{key} = \(\infty\)
3. \(u\).\pi = \text{\textit{NULL}}
4. \(r\).\textit{key} = 0 \{ \text{start from vertex } r \} \}
5. \(Q = G.V\) \{ \text{establish priority queue } Q \text{ wit key values} \}
6. \textbf{while} \(Q \neq \emptyset\)
7. \(u = \text{\textbf{Extract Min}}(Q)\)
8. \textbf{for} each \(v \in \text{\textit{Adj}}[u]\)
9. \textbf{if} \(v \in Q \text{ and } w(u, v) < v\).\textit{key} \{ \text{for those not in } A, \text{ update distances} \}
10. \textbf{then} \(v\).\pi = u\)

\textit{usage of Priority queue: } \(Q\), \textbf{Extract Min} takes \(O(\log n)\) \time\).  
\textit{running time} \(O(|E| + |V| \log |V|)\).
Chapter 23. Minimum Spanning Trees

MST-Prim(G, w, r)
1. for each \( u \in G.V \) \{ \( u.key \) is the \( u \)'s shortest distance to set \( A = V-Q \) \}
2. \( u.key = \infty \)
3. \( u.\pi = NULL \)
4. \( r.key = 0 \) \{ start from vertex \( r \) \}
5. \( Q = G.V \) \{ establish priority queue \( Q \) wit key values \}
6. while \( Q \neq \emptyset \)
7. \( u = \text{Extract Min}(Q) \)
8. for each \( v \in Adj[u] \)
9. if \( v \in Q \) and \( w(u, v) < v.key \) \{ for those not in \( A \), update distances \}
10. then \( v.\pi = u \)
11. \( v.key = w(u, v) \)
Chapter 23. Minimum Spanning Trees

MST-Prim\((G, w, r)\)
1. \textbf{for each} \(u \in G.V\) \{ \(u\).key is the \(u\)'s shortest distance to set \(A = V\setminus Q\) \}
2. \(u\).key = \(\infty\)
3. \(u\).\(\pi = NULL\)
4. \(r\).key = 0 \{ start from vertex \(r\) \}
5. \(Q = G.V\) \{ establish priority queue \(Q\) wit key values \}
6. \textbf{while} \(Q \neq \emptyset\)
7. \(u = \text{Extract Min}(Q)\)
8. \textbf{for each} \(v \in Adj[u]\)
9. \textbf{if} \(v \in Q\) and \(w(u, v) < v\).key \{ for those not in \(A\), update distances \}
10. \textbf{then} \(v\).\(\pi = u\)
11. \(v\).key = \(w(u, v)\)
12. \textbf{return} \(\pi\)

usage of Priority queue: \(Q\), \(\text{Extract Min}\) takes \(O(\log n)\) time.

running time \(O(\mid E\mid + \mid V\mid \log \mid V\mid)\).
MST-Prim\((G, w, r)\)
1. \textbf{for} each \(u \in G.V\) \{ \(u.key\) is the \(u\)'s shortest distance to set \(A = V-Q\) \}
2. \(u.key = \infty\) \{ start from vertex \(r\) \}
3. \(u.\pi = NULL\)
4. \(r.key = 0\) \{ establish priority queue \(Q\) wit key values\}
5. \(Q = G.V\)
6. \textbf{while} \(Q \neq \emptyset\)
7. \(u = \text{EXTRACT MIN}(Q)\)
8. \textbf{for} each \(v \in Adj[u]\)
9. \hspace{1em} \textbf{if} \(v \in Q\) and \(w(u, v) < v.key\) \{ for those not in \(A\), update distances\}
10. \hspace{2em} \textbf{then} \(v.\pi = u\)
11. \hspace{2em} \(v.key = w(u, v)\)
12. \textbf{return} \(\pi\)

usage of Priority queue: \(Q\), \text{EXTRACT MIN} takes \(O(\log n)\) time.
Chapter 23. Minimum Spanning Trees

MST-Prim($G, w, r$)
1. for each $u \in G.V$
2. $u.key = \infty$ \{ $u.key$ is the $u$'s shortest distance to set $A = V - Q$ \}
3. $u.\pi = NULL$
4. $r.key = 0$ \{ start from vertex $r$ \}
5. $Q = G.V$ \{ establish priority queue $Q$ wit key values \}
6. while $Q \neq \emptyset$
7. $u = \text{Extract Min}(Q)$
8. for each $v \in Adj[u]$
9. if $v \in Q$ and $w(u, v) < v.key$ \{ for those not in $A$, update distances \}
10. then $v.\pi = u$
11. $v.key = w(u, v)$
12. return $\pi$

usage of Priority queue: $Q$, $\text{Extract Min}$ takes $O(\log n)$ time.

running time $O(|E| + |V| \log |V|)$. 

Chapter 23. Minimum Spanning Trees
Chapter 23. Minimum Spanning Trees

Summary of Kruskal’s and Prim’s algorithms:
Summary of Kruskal’s and Prim’s algorithms:

- initialize parent array
Chapter 23. Minimum Spanning Trees

Summary of Kruskal’s and Prim’s algorithms:

- initialize parent array
  - initialize $A = \emptyset$ or initial vertex $r$;
Chapter 23. Minimum Spanning Trees

Summary of Kruskal’s and Prim’s algorithms:

- initialize parent array
  
  initialize \( A = \emptyset \) or initial vertex \( r \);

- repeatedly choosing from the remaining edges;
Chapter 23. Minimum Spanning Trees

Summary of Kruskal’s and Prim’s algorithms:

- initialize parent array
  
  initialize $A = \emptyset$ or initial vertex $r$;

- repeatedly choosing from the remaining edges;
  
  pick a light edge that respects a cut
Chapter 23. Minimum Spanning Trees

Summary of Kruskal’s and Prim’s algorithms:

• initialize parent array
  
  initialize $A = \emptyset$ or initial vertex $r$;

• repeatedly choosing from the remaining edges;
  
  pick a light edge that respects a cut
  add it to $A$
Summary of Kruskal’s and Prim’s algorithms:

- initialize parent array
  
  initialize $A = \emptyset$ or initial vertex $r$;

- repeatedly choosing from the remaining edges;
  
  pick a light edge that respects a cut
  
  add it to $A$,
Chapter 23. Minimum Spanning Trees

Summary of Kruskal’s and Prim’s algorithms:

- initialize parent array
  
  initialize $A = \emptyset$ or initial vertex $r$;

- repeatedly choosing from the remaining edges;
  
  pick a light edge that respects a cut
  add it to $A$,
  ensure that $A$ is a subset of some MST
Summary of Kruskal’s and Prim’s algorithms:

- initialize parent array
  - initialize $A = \emptyset$ or initial vertex $r$;
- repeatedly choosing from the remaining edges;
  - pick a light edge that respects a cut
  - add it to $A$,
  - ensure that $A$ is a subset of some MST
- until $A$ forms a spanning tree.
Some questions about MST
Chapter 23. Minimum Spanning Trees

Some questions about MST

• What are the "cuts" implied in Kruskal’s algorithm and in Prim’s algorithm, respectively?
Some questions about MST

• What are the "cuts" implied in Kruskal’s algorithm and in Prim’s algorithm, respectively?

• Can we develop a DP algorithm for the MST problem?
Chapter 23. Minimum Spanning Trees

Some questions about MST

• What are the "cuts" implied in Kruskal’s algorithm and in Prim’s algorithm, respectively?

• Can we develop a DP algorithm for the MST problem?

  the main issue: how solutions to subproblems help build solution for the problem
Some questions about MST

• What are the "cuts" implied in Kruskal’s algorithm and in Prim’s algorithm, respectively?

• Can we develop a DP algorithm for the MST problem?
  
  the main issue: how solutions to subproblems help build solution for the problem
  
  what are subproblems, or what do subsolutions look like?
Chapter 24. Single Source Shortest Paths

Chapter 24. Single-source shortest paths
Chapter 24. Single Source Shortest Path

Chapter 24. Single-source shortest paths

Given a graph \( G = (V, E) \), with weight \( w : E \rightarrow R \); and single vertex \( s \in V \);
Chapter 24. Single Source Shortest Paths

Chapter 24. Single-source shortest paths

Given a graph $G = (V, E)$, with weight $w : E \rightarrow R$; and single vertex $s \in V$;

for each vertex $v \in V$, find a shortest path $s \leadsto v$. 

• Shortest path is a simple path.
• “Distance” is measured by the total edge weight on the path, i.e., if the path $v_0 \ldots p \leadsto v_k$ is $p = (v_0, v_1, \ldots, v_k)$ then the path weight is $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.
• shortest distance between $u$ and $v$ is $\delta(u, v) = \min_{u \leadsto v} \{ w(p) \}$. 
Chapter 24. Single Source Shortest Paths

Chapter 24. Single-source shortest paths

Given a graph $G = (V, E)$, with weight $w : E \rightarrow R$; and single vertex $s \in V$; for each vertex $v \in V$, find a shortest path $s \rightsquigarrow v$.

- Shortest path is a simple path.
Chapter 24. Single-source shortest paths

Given a graph \( G = (V, E) \), with weight \( w : E \to \mathbb{R} \); and single vertex \( s \in V \);

for each vertex \( v \in V \), find a shortest path \( s \leadsto v \).

- Shortest path is a simple path.
- “Distance” is measured by the total edge weight on the path.
Chapter 24. Single-source shortest paths

Given a graph $G = (V, E)$, with weight $w : E \to \mathbb{R}$; and single vertex $s \in V$; for each vertex $v \in V$, find a shortest path $s \leadsto v$.

- **Shortest path is a simple path.**
- “Distance” is measured by the total edge weight on the path
  
  i.e., if the path $v_0 \overset{p}{\leadsto} v_k$ is $p = (v_0, v_1, \ldots, v_k)$

  then the path weight is $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
Chapter 24. Single Source Shortest Paths

Chapter 24. Single-source shortest paths

Given a graph \( G = (V, E) \), with weight \( w : E \rightarrow \mathbb{R} \); and single vertex \( s \in V \);

for each vertex \( v \in V \), find a shortest path \( s \rightsquigarrow v \).

- Shortest path is a simple path.
- “Distance” is measured by the total edge weight on the path
  i.e., if the path \( v_0 \overset{p}{\rightsquigarrow} v_k \) is \( p = (v_0, v_1, \ldots, v_k) \)
  then the path weight is \( w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \)
- Shortest distance between \( u \) and \( v \) is
  \[
  \delta(u, v) = \min_{u \overset{p}{\rightsquigarrow} v} \{ w(p) \} 
  \]
Chapter 24. Single Source Shortest Paths

- Single-source shortest paths: from $s$ to each vertex $v \in V$
Chapter 24. Single Source Shortest Paths

- **Single-source shortest paths**: from $s$ to each vertex $v \in V$
- a special case: **Single-pair shortest path**: from $s$ to $t$
Chapter 24. Single Source Shortest Paths

- **Single-source shortest paths**: from $s$ to each vertex $v \in V$
- a special case: **Single-pair shortest path**: from $s$ to $t$
- **All-pairs shortest paths**: from $s$ to $t$ for all pairs $s, t \in V$. 
Chapter 24. Single Source Shortest Paths

Lemma 24.1 (a subpath of a shortest path is a shortest path)

Proof idea: (proof by contradiction)
Assume that $p_{i,j}$ is not the shortest path from $v_i$ to $v_j$. Then there is a shorter path $q_{i,j}$ from $v_i$ to $v_j$.

Define path $q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)$ has weight $w(q) = \sum_{t=1}^{i} w(v_t - 1, v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1}) < \sum_{t=1}^{i} w(v_t - 1, v_t) + w(p_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1}) = w(p)$
contradicts to the assumption that $p$ is the shortest path from $v_0$ to $v_k$. 
Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph $G = (V, E)$ with edge weight function $w$. Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path $v_0 \xmapsto{p} v_k$. Then $p_{i,j} = (v_i, \ldots, v_j)$ is a shortest path $v_i \xmapsto{p_{i,j}} v_j$. 

Proof idea: (proof by contradiction) Assume that $p_{i,j}$ is not the shortest path from $v_i$ to $v_j$. Then there is a shorter path $q_{i,j}$ from $v_i$ to $v_j$. Define path $q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)$ has weight $w(q) = \sum_{t=1}^{i} w(v_{t-1}v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_tv_{t+1}) < \sum_{t=1}^{i} w(v_{t-1}v_t) + w(p_{i,j}) + \sum_{t=j}^{k-1} w(v_tv_{t+1}) = w(p)$ contradicts to the assumption that $p$ is the shortest path from $v_0$ to $v_k$. 

Chapter 24. Single Source Shortest Paths

Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph $G = (V, E)$ with edge weight function $w$. Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path $v_0 \overset{p}{\longrightarrow} v_k$. Then $p_{i,j} = (v_i, \ldots, v_j)$ is a shortest path $v_i \overset{p_{i,j}}{\longrightarrow} v_j$.

Proof idea: (proof by contradiction)
Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph $G = (V, E)$ with edge weight function $w$. Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path $v_0 \xrightarrow{p} v_k$. Then $p_{i,j} = (v_i, \ldots, v_j)$ is a shortest path $v_i \xrightarrow{p_{i,j}} v_j$.

Proof idea: (proof by contradiction) Assume that $p_{i,j}$ is not the shortest path from $v_i$ to $v_j$. Then there is a shorter path $q_{i,j}$ from $v_i$ to $v_j$. 

\[ w(p_{i,j}) < w(p) \]

This contradicts the assumption that $p$ is the shortest path from $v_0$ to $v_k$. Therefore, $p_{i,j}$ is a shortest path from $v_i$ to $v_j$. 


Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph \( G = (V, E) \) with edge weight function \( w \). Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path \( v_0 \xrightarrow{p} v_k \). Then \( p_{i,j} = (v_i, \ldots, v_j) \) is a shortest path \( v_i \xrightarrow{p_{i,j}} v_j \).

Proof idea: (proof by contradiction) Assume that \( p_{i,j} \) is not the shortest path from \( v_i \) to \( v_j \). Then there is a shorter path \( q_{i,j} \) from \( v_i \) to \( v_j \).

Define path

\[ q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k) \]

has weight

\[ w(q) = \]
Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph \( G = (V, E) \) with edge weight function \( w \). Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path \( v_0 \overset{p}{\rightarrow} v_k \). Then \( p_{i,j} = (v_i, \ldots, v_j) \) is a shortest path \( v_i \overset{p_{i,j}}{\rightarrow} v_j \).

Proof idea: (proof by contradiction) Assume that \( p_{i,j} \) is not the shortest path from \( v_i \) to \( v_j \). Then there is a shorter path \( q_{i,j} \) from \( v_i \) to \( v_j \).

Define path

\[
q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)
\]

has weight

\[
w(q) = \sum_{t=1}^{i} w(v_{t-1}, v_t)
\]
Chapter 24. Single Source Shortest Paths

**Lemma 24.1** (a subpath of a shortest path is a shortest path)

Given a weighted directed graph \( G = (V, E) \) with edge weight function \( w \). Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path \( v_0 \overset{p}{\rightarrow} v_k \). Then \( p_{i,j} = (v_i, \ldots, v_j) \) is a shortest path \( v_i \overset{p_{i,j}}{\rightarrow} v_j \).

**Proof idea:** (proof by contradiction) Assume that \( p_{i,j} \) is not the shortest path from \( v_i \) to \( v_j \). Then there is a shorter path \( q_{i,j} \) from \( v_i \) to \( v_j \).

Define path
\[
q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)
\]
has weight
\[
w(q) = \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(q_{i,j}) +
\]
Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph $G = (V, E)$ with edge weight function $w$. Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path $v_0 \xrightarrow{p} v_k$. Then $p_{i,j} = (v_i, \ldots, v_j)$ is a shortest path $v_i \xrightarrow{p_{i,j}} v_j$.

Proof idea: (proof by contradiction) Assume that $p_{i,j}$ is not the shortest path from $v_i$ to $v_j$. Then there is a shorter path $q_{i,j}$ from $v_i$ to $v_j$.

Define path

$$q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)$$

has weight

$$w(q) = \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1})$$

contradicts to the assumption that $p$ is the shortest path from $v_0$ to $v_k$. 
Chapter 24. Single Source Shortest Paths

Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph \( G = (V, E) \) with edge weight function \( w \). Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path \( v_0 \overset{p}{\rightarrow} v_k \). Then \( p_{i,j} = (v_i, \ldots, v_j) \) is a shortest path \( v_i \overset{p_{i,j}}{\rightarrow} v_j \).

Proof idea: (proof by contradiction) Assume that \( p_{i,j} \) is not the shortest path from \( v_i \) to \( v_j \). Then there is a shorter path \( q_{i,j} \) from \( v_i \) to \( v_j \).

Define path

\[
q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)
\]

has weight

\[
w(q) = \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1})
\]
Chapter 24. Single Source Shortest Paths

Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph \( G = (V, E) \) with edge weight function \( w \). Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path \( v_0 \xrightarrow{p} v_k \). Then \( p_{i,j} = (v_i, \ldots, v_j) \) is a shortest path \( v_i \xrightarrow{p_{i,j}} v_j \).

Proof idea: (proof by contradiction) Assume that \( p_{i,j} \) is not the shortest path from \( v_i \) to \( v_j \). Then there is a shorter path \( q_{i,j} \) from \( v_i \) to \( v_j \).

Define path

\[
q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)
\]

has weight

\[
w(q) = \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1})
\]

\[
< \sum_{t=1}^{i} w(v_{t-1}, v_t)
\]
Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph \( G = (V, E) \) with edge weight function \( w \). Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path \( v_0 \overset{p}{\rightarrow} v_k \). Then \( p_{i,j} = (v_i, \ldots, v_j) \) is a shortest path \( v_i \overset{p_{i,j}}{\rightarrow} v_j \).

Proof idea: (proof by contradiction) Assume that \( p_{i,j} \) is not the shortest path from \( v_i \) to \( v_j \). Then there is a shorter path \( q_{i,j} \) from \( v_i \) to \( v_j \).

Define path

\[ q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k) \]

has weight

\[
\begin{align*}
w(q) &= \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1}) \\
&< \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(p_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1})
\end{align*}
\]
**Chapter 24. Single Source Shortest Paths**

**Lemma 24.1** (a subpath of a shortest path is a shortest path)

Given a weighted directed graph $G = (V, E)$ with edge weight function $w$. Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path $v_0 \xrightarrow{p} v_k$. Then $p_{i,j} = (v_i, \ldots, v_j)$ is a shortest path $v_i \xrightarrow{p_{i,j}} v_j$.

**Proof idea:** (proof by contradiction) Assume that $p_{i,j}$ is not the shortest path from $v_i$ to $v_j$. Then there is a shorter path $q_{i,j}$ from $v_i$ to $v_j$.

Define path

$$q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)$$

has weight

$$w(q) = \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1})$$

$$< \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(p_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1})$$

contradicts to the assumption that $p$ is the shortest path from $v_0$ to $v_k$. 
Chapter 24. Single Source Shortest Paths

Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph $G = (V, E)$ with edge weight function $w$. Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path $v_0 \xrightarrow{p} v_k$. Then $p_{i,j} = (v_i, \ldots, v_j)$ is a shortest path $v_i \xrightarrow{p_{i,j}} v_j$.

Proof idea: (proof by contradiction) Assume that $p_{i,j}$ is not the shortest path from $v_i$ to $v_j$. Then there is a shorter path $q_{i,j}$ from $v_i$ to $v_j$.

Define path

$$ q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k) $$

has weight

$$ w(q) = \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1}) $$

$$< \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(p_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1}) = w(p)$$
Lemma 24.1 (a subpath of a shortest path is a shortest path)

Given a weighted directed graph $G = (V, E)$ with edge weight function $w$. Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path $v_0 \stackrel{p}{\rightarrow} v_k$. Then $p_{i,j} = (v_i, \ldots, v_j)$ is a shortest path $v_i \stackrel{p_{i,j}}{\rightarrow} v_j$.

Proof idea: (proof by contradiction) Assume that $p_{i,j}$ is not the shortest path from $v_i$ to $v_j$. Then there is a shorter path $q_{i,j}$ from $v_i$ to $v_j$.

Define path

$$q = (v_0, \ldots, v_i, q_{i,j}, v_j, \ldots, v_k)$$

has weight

$$w(q) = \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(q_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1})$$

$$< \sum_{t=1}^{i} w(v_{t-1}, v_t) + w(p_{i,j}) + \sum_{t=j}^{k-1} w(v_t, v_{t+1}) = w(p)$$

contradicts to the assumption that $p$ is the shortest path from $v_0$ to $v_k$. 
Chapter 24. Single Source Shortest Paths

Some terminologies:
Chapter 24. Single Source Shortest Paths

Some terminologies:

- **negative weights** are allowed;
Chapter 24. Single Source Shortest Paths

Some terminologies:

- **negative weights** are allowed;
- cycles on a path: not a simple path;
Some terminologies:

- **negative weights** are allowed;
- cycles on a path: not a simple path;
- **negative weight cycles**, 0 weight cycles

---

Chapter 24. Single Source Shortest Paths

Some terminologies:

- **negative weights** are allowed;
- cycles on a path: not a simple path;
- **negative weight cycles**, 0 weight cycles
Chapter 24. Single Source Shortest Paths

Some terminologies:

- **negative weights** are allowed;
- cycles on a path: not a simple path;
- **negative weight cycles**, 0 weight cycles
- representing shortest paths: predecessor $\pi$

[http://graphserver.sourceforge.net/gallery.html](http://graphserver.sourceforge.net/gallery.html)
Chapter 24. Single Source Shortest Paths

Some terminologies:

- **negative weights** are allowed;
- cycles on a path: not a simple path;
- **negative weight cycles**, 0 weight cycles
- representing shortest paths: predecessor $\pi$
  shortest path tree:
Some terminologies:

- **negative weights** are allowed;
- cycles on a path: not a simple path;
- **negative weight cycles**, 0 weight cycles
- representing shortest paths: predecessor $\pi$
  shortest path tree:

  http://graphserver.sourceforge.net/gallery.html
  (width $\propto 1$/distance)
Chapter 24. Single Source Shortest Paths

Technique: relaxation

- Intuition:
  
  if $s \xrightarrow{P} v$ has distance $v.d$ (computed so far),
Chapter 24. Single Source Shortest Paths

**Technique: relaxation**

- **Intuition:**
  
  if \( s \xrightarrow{p} v \) has distance \( v.d \) (computed so far),
  
  \( s \xrightarrow{q} u \) is **newly discovered**.
Technique: relaxation

- Intuition:
  
  if $s \xrightarrow{p} v$ has distance $v.d$ (computed so far),
  $s \xrightarrow{q} u$ is newly discovered. Then
Chapter 24. Single Source Shortest Paths

**Technique: relaxation**

- **Intuition:**
  
  if $s \xrightarrow{p} v$ has distance $v.d$ (computed so far),
  
  $s \xrightarrow{q} u$ is newly discovered. Then
  
  $$v.d = \min\{v.d, u.d + w(u, v)\}$$
Chapter 24. Single Source Shortest Paths

**Technique: relaxation**

- **Intuition:**
  
  if $s \xrightarrow{p} v$ has distance $v.d$ (computed so far),
  
  $s \xrightarrow{q} u$ is newly discovered. Then

  $$v.d = \min\{v.d, u.d + w(u,v)\}$$

- **In other words:**
Chapter 24. Single Source Shortest Paths

Technique: relaxation

• Intuition:

if \( s \rightarrow v \) has distance \( v.d \) (computed so far),
\( s \rightarrow u \) is newly discovered. Then

\[
v.d = \min\{v.d, u.d + w(u, v)\}
\]

• In other words:

Let \( v.d \) be an weight upper bound of a shortest path from \( s \) to \( v \),
Chapter 24. Single Source Shortest Paths

Technique: relaxation

• Intuition:

  if \( s \xrightarrow{p} v \) has distance \( v.d \) (computed so far),
  \( s \xrightarrow{q} u \) is newly discovered. Then

  \[
  v.d = \min\{v.d, u.d + w(u,v)\}
  \]

• In other words:

  Let \( v.d \) be an weight upper bound of a shortest path from \( s \) to \( v \),
  initialized \( \infty \).
Chapter 24. Single Source Shortest Paths

Technique: relaxation

• Intuition:

  if $s \xrightarrow{P} v$ has distance $v.d$ (computed so far),
  $s \xrightarrow{q} u$ is newly discovered. Then

  $$v.d = \min \{v.d, u.d + w(u, v)\}$$

• In other words:

  Let $v.d$ be an weight upper bound of a shortest path from $s$ to $v$, initialized $\infty$.

  The process of relaxing edge $(u, v)$: improves $v.d$ by taking the path through $u$, and update $v.d$ and $v.\pi$. 
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

\textsc{Bellman-Ford}(G, w, s)
Bellman-Ford algorithm

**Bellman-Ford**$(G, w, s)$
1. for each vertex $v \in G.V$ initialization

Running time: $O(|V| |E|)$
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

\textsc{Bellman-Ford}(G, w, s)
1. \textbf{for} each vertex \( v \in G.V \) \hspace{1cm} \text{initialization}
2. \hspace{1cm} \( v.d = \infty \)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

\textsc{Bellman-Ford}(G, w, s)

1. \textbf{for} each vertex \( v \in G.V \) \hspace{0.5cm} \text{initialization}
2. \hspace{0.5cm} \( v.d = \infty \)
3. \hspace{0.5cm} \( v.\pi = NULL \)

4. \hspace{0.5cm} \( s.d = 0 \)
5. \hspace{0.5cm} \textbf{for} \( i = 1 \) to \(|V| - 1\) \hspace{0.5cm} \text{relaxation}
6. \hspace{0.5cm} \textbf{for} each edge \((u, v) \in G.E\)
7. \hspace{0.5cm} \textbf{if} \( v.d > u.d + w(u, v) \)
8. \hspace{0.5cm} \hspace{0.5cm} \( v.d = u.d + w(u, v) \)
9. \hspace{0.5cm} \hspace{0.5cm} \( v.\pi = u \)

10. \hspace{0.5cm} \textbf{for} each edge \((u, v) \in G.E\) \hspace{0.5cm} \text{checking negative weight cycle}
11. \hspace{0.5cm} \hspace{0.5cm} \textbf{if} \( v.d > u.d + w(u, v) \)
12. \hspace{0.5cm} \hspace{0.5cm} \hspace{0.5cm} \textbf{return} \textit{FALSE}
13. \hspace{0.5cm} \hspace{0.5cm} \textbf{return} \textit{TRUE}

\textbf{Running time:} \( O(|V||E|) \)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

\textbf{BELLMAN-FORD}(G, w, s)

1. \textbf{for} each vertex \( v \in G.V \) \textbf{ initialization} \\
2. \hspace{1cm} \( v.d = \infty \) \\
3. \hspace{1cm} \( v.\pi = NULL \) \\
4. \hspace{1cm} \( s.d = 0 \)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

\textbf{Bellman-Ford}(G, w, s)

1. \textbf{for each vertex } v \in G.V \textbf{ initialization}
2. \hspace{1cm} v.d = \infty
3. \hspace{1cm} v.\pi = \text{NULL}
4. \hspace{1cm} s.d = 0
5. \hspace{1cm} \textbf{for } i = 1 \textbf{ to } |V| - 1 \textbf{ relaxation}

\text{Running time : } O(|V||E|)
Bellman-Ford algorithm

\textsc{Bellman-Ford}(G, w, s)

1. \textbf{for} each vertex \( v \in G.V \) \hspace{1cm} \text{initialization}
2. \hspace{1cm} \( v.d = \infty \)
3. \hspace{1cm} \( v.\pi = NULL \)
4. \hspace{1cm} \( s.d = 0 \)
5. \textbf{for} \( i = 1 \) \textbf{to} \( |V| - 1 \) \hspace{1cm} \text{relaxation}
6. \hspace{1cm} \textbf{for} each edge \((u, v) \in G.E\)

\textbf{Running time:} \( O(|V||E|) \)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

\textsc{Bellman-Ford}(G, w, s)
1. \textbf{for} each vertex \( v \in G.V \) \hspace{1cm} \textit{initialization}
2. \hspace{1cm} \( v.d = \infty \)
3. \hspace{1cm} \( v.\pi = \text{NULL} \)
4. \hspace{1cm} \( s.d = 0 \)
5. \textbf{for} \( i = 1 \) \textbf{to} \( |V| - 1 \) \hspace{1cm} \textit{relaxation}
6. \hspace{1cm} \textbf{for} each edge \( (u, v) \in G.E \)
7. \hspace{1cm} \textbf{if} \( v.d > u.d + w(u,v) \)

Running time: \( O(|V||E|) \)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

**Bellman-Ford**\((G, w, s)\)

1. **for** each vertex \(v \in G.V\) \text{ initialization}
2. \(v.d = \infty\)
3. \(v.\pi = NULL\)
4. \(s.d = 0\)
5. **for** \(i = 1\) to \(|V| - 1\) \text{ relaxation}
6. **for** each edge \((u, v) \in G.E\)
7. \(\text{if } v.d > u.d + w(u, v)\)
8. \(v.d = u.d + w(u, v)\)

Running time: \(O(|V||E|)\)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

**Bellman-Ford**($G, w, s$)

1. **for** each vertex $v \in G.V$ initialization
2. $v.d = \infty$
3. $v.\pi = NULL$
4. $s.d = 0$
5. **for** $i = 1$ to $|V| - 1$ relaxation
6. **for** each edge $(u, v) \in G.E$
7. if $v.d > u.d + w(u, v)$
8. $v.d = u.d + w(u, v)$
9. $v.\pi = u$

Running time: $O(|V||E|)$
Bellman-Ford algorithm

**Bellman-Ford**($G, w, s$)

1. for each vertex $v \in G.V$ initialization
2. \( v.d = \infty \)
3. \( v.\pi = NULL \)
4. \( s.d = 0 \)
5. for $i = 1$ to $|V| - 1$ relaxation
6. for each edge $(u, v) \in G.E$
7. if $v.d > u.d + w(u, v)$
8. \( v.d = u.d + w(u, v) \)
9. \( v.\pi = u \)
10. for each edge $(u, v) \in G.E$ checking negative weight cycle

Running time: \( O(|V||E|) \)
Bellman-Ford algorithm

\textsc{Bellman-Ford}(G, w, s)

1. \textbf{for each vertex }v \in G.V \textbf{ \ add initialization}
   \hspace{1cm} \textit{initialization}
2. \quad v.d = \infty
3. \quad v.\pi = \text{NULL}
4. \quad s.d = 0
5. \quad \textbf{for } i = 1 \textbf{ \ to } |V| - 1 \textbf{ \ add relaxation}
6. \quad \textbf{for each edge } (u, v) \in G.E
7. \quad \hspace{1cm} \textbf{if } v.d > u.d + w(u, v)
8. \quad \hspace{1cm} v.d = u.d + w(u, v)
9. \quad \hspace{1cm} v.\pi = u
10. \textbf{for each edge } (u, v) \in G.E \textbf{ \ add checking negative weight cycle}
11. \quad \textbf{if } v.d > u.d + w(u, v)

\textbf{Running time :} \ O(|V||E|)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

\[
\text{Bellman-Ford}(G, w, s)
\]
1. for each vertex \( v \in G.V \) initialization
2. \( v.d = \infty \)
3. \( v.\pi = \text{NULL} \)
4. \( s.d = 0 \)
5. for \( i = 1 \) to \(|V| - 1\) relaxation
6. for each edge \((u, v) \in G.E\)
7. if \( v.d > u.d + w(u, v) \)
8. \( v.d = u.d + w(u, v) \)
9. \( v.\pi = u \)
10. for each edge \((u, v) \in G.E\) checking negative weight cycle
11. if \( v.d > u.d + w(u, v) \)
12. return (FALSE)

Running time: \(O(|V||E|)\)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

**Bellman-Ford**\((G, w, s)\)

1. for each vertex \(v \in G.V\) initialization
2. \(v.d = \infty\)
3. \(v.\pi = NULL\)
4. \(s.d = 0\)
5. for \(i = 1\) to \(|V| - 1\) relaxation
6. for each edge \((u, v) \in G.E\)
7. if \(v.d > u.d + w(u, v)\)
8. \(v.d = u.d + w(u, v)\)
9. \(v.\pi = u\)
10. for each edge \((u, v) \in G.E\) checking negative weight cycle
11. if \(v.d > u.d + w(u, v)\)
12. return (FALSE)
13. return (TRUE)

Running time: \(O(|V||E|)\)
Chapter 24. Single Source Shortest Paths

Bellman-Ford algorithm

\textsc{Bellman-Ford}(G, w, s)

1. \textbf{for} each vertex \( v \in G.V \) \hspace{1cm} \text{initialization}
2. \hspace{1cm} \( v.d = \infty \)
3. \hspace{1cm} \( v.\pi = NULL \)
4. \hspace{1cm} \( s.d = 0 \)
5. \textbf{for} \( i = 1 \) to \( |V| - 1 \) \hspace{1cm} \text{relaxation}
6. \hspace{1cm} \textbf{for} each edge \( (u, v) \in G.E \)
7. \hspace{2cm} \textbf{if} \( v.d > u.d + w(u, v) \)
8. \hspace{2cm} \hspace{1cm} \( v.d = u.d + w(u, v) \)
9. \hspace{2cm} \hspace{1cm} \( v.\pi = u \)
10. \hspace{1cm} \textbf{for} each edge \( (u, v) \in G.E \) \hspace{1cm} \text{checking negative weight cycle}
11. \hspace{2cm} \textbf{if} \( v.d > u.d + w(u, v) \)
12. \hspace{2cm} \hspace{1cm} \textbf{return} (FALSE)
13. \hspace{2cm} \textbf{return} (TRUE)

Running time : \( O(|V||E|) \)
Chapter 24. Single Source Shortest Paths
Chapter 24. Single Source Shortest Paths

1.

2.

3.

4.

5.
Properties of shortest paths and relaxation
Properties of shortest paths and relaxation

\texttt{RELAX}(u, v, w)
Properties of shortest paths and relaxation

\texttt{RELAX}(u, v, w)

1. \textbf{if} \( v.d > u.d + w(u, v) \)
Properties of shortest paths and relaxation

**RELAX**\((u, v, w)\)
1. \(v.d > u.d + w(u, v)\)
2. \(v.d = u.d + w(u, v)\)
Chapter 24. Single Source Shortest Paths

Properties of shortest paths and relaxation

\text{RELAX}(u, v, w)

1. \textbf{if } v.d > u.d + w(u, v)
2. \quad v.d = u.d + w(u, v)
3. \quad v.\pi = u
Properties of shortest paths and relaxation

**RELAX**\((u, v, w)\)

1. **if** \(v.d > u.d + w(u, v)\)
2. \(v.d = u.d + w(u, v)\)
3. \(v.\pi = u\)

**Lemma 24.14, Convergence property:** Let \(s \leadsto u \rightarrow v\) is a shortest path. If \(u.d = \delta(s, u)\) holds before **RELAX**\((u, v, w)\) is called, then \(v.d = \delta(s, v)\) after the call.
Properties of shortest paths and relaxation

RELAX\((u, v, w)\)
1. \( \text{if } v.d > u.d + w(u, v) \)
2. \( v.d = u.d + w(u, v) \)
3. \( v.\pi = u \)

Lemma 24.14, Convergence property: Let \( s \leadsto u \rightarrow v \) is a shortest path. If \( u.d = \delta(s, u) \) holds before RELAX\((u, v, w)\) is called, then \( v.d = \delta(s, v) \) after the call.

Proof:
Properties of shortest paths and relaxation

\textsc{Relax}(u, v, w)
1. if \( v.d > u.d + w(u, v) \)
2. \( v.d = u.d + w(u, v) \)
3. \( v.\pi = u \)

\textbf{Lemma 24.14, Convergence property:} Let \( s \rightsquigarrow u \rightarrow v \) is a shortest path. If \( u.d = \delta(s, u) \) holds before \textsc{Relax}(u, v, w) is called, then \( v.d = \delta(s, v) \) after the call.

\textbf{Proof:} \( v.d \leq u.d + w(u, v) \)
Properties of shortest paths and relaxation

\texttt{RELAX}(u, v, w)

1. \texttt{if } v.d > u.d + w(u, v) \\
2. \hspace{1cm} v.d = u.d + w(u, v) \\
3. \hspace{1cm} v.\pi = u \\

**Lemma 24.14, Convergence property:** Let \( s \xrightarrow{} u \rightarrow v \) is a shortest path. If \( u.d = \delta(s, u) \) holds before \texttt{RELAX}(u, v, w) is called, then \( v.d = \delta(s, v) \) after the call.

**Proof:** \( v.d \leq u.d + w(u, v) = \delta(s, u) + w(u, v) \)
Properties of shortest paths and relaxation

\texttt{RELAX}(u, v, w)
1. \textbf{if} $v.d > u.d + w(u, v)$
2. $v.d = u.d + w(u, v)$
3. $v.\pi = u$

\textbf{Lemma 24.14, Convergence property}: Let $s \leadsto u \rightarrow v$ is a shortest path. If $u.d = \delta(s, u)$ holds before \texttt{RELAX}(u, v, w) is called, then $v.d = \delta(s, v)$ after the call.

\textbf{Proof}: $v.d \leq u.d + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v)$. 
Chapter 24. Single Source Shortest Paths

Properties of shortest paths and relaxation

\textsc{Relax}(u, v, w)
1. if \( v.d > u.d + w(u, v) \)
2. \( v.d = u.d + w(u, v) \)
3. \( v.\pi = u \)

**Lemma 24.14, Convergence property:** Let \( s \leadsto u \rightarrow v \) is a shortest path. If \( u.d = \delta(s, u) \) holds before \textsc{Relax}(u, v, w) is called, then \( v.d = \delta(s, v) \) after the call.

**Proof:** \( v.d \leq u.d + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v) \). So \( v.d = \delta(s, v) \).
We want to prove that, if a shortest path \( s \leadsto v \) consists of \( k \) edges, Bellman-Fold obtains value \( v.d = \delta(s, v) \) after the \( k \)th round of relaxation (assuming there is no negative cycle).
We want to prove that, if a shortest path $s \leadsto v$ consists of $k$ edges, Bellman-Fold obtains value $v.d = \delta(s, v)$ after the $k$th round of relaxation (assuming there is no negative cycle).

Proof idea: Induction on $k$. 
We want to prove that, if a shortest path $s \rightsquigarrow v$ consists of $k$ edges, Bellman-Fold obtains value $v.d = \delta(s,v)$ after the $k$th round of relaxation (assuming there is no negative cycle).

**Proof idea:** Induction on $k$.

- $k = 0$, $v$ can only be $s$. Proved!
Chapter 24. Single Source Shortest Paths

We want to prove that, if a shortest path $s \leadsto v$ consists of $k$ edges, Bellman-Fold obtains value $v.d = \delta(s,v)$ after the $k$th round of relaxation (assuming there is no negative cycle).

Proof idea: Induction on $k$.

- $k = 0$, $v$ can only be $s$. Proved!

- Assume the claim is proved for all vertices $v$ that have a shortest path of length $k$. 
Chapter 24. Single Source Shortest Paths

We want to prove that, if a shortest path $s \leadsto v$ consists of $k$ edges, Bellman-Fold obtains value $v.d = \delta(s, v)$ after the $k$th round of relaxation (assuming there is no negative cycle).

Proof idea: Induction on $k$.

- $k = 0$, $v$ can only be $s$. Proved!

- Assume the claim is proved for all vertices $v$ that have a shortest path of length $k$. What claim again??
We want to prove that, if a shortest path $s \rightarrow v$ consists of $k$ edges, Bellman-Fold obtains value $v.d = \delta(s, v)$ after the $k$th round of relaxation (assuming there is no negative cycle).

Proof idea: Induction on $k$.

- $k = 0$, $v$ can only be $s$. Proved!
- Assume the claim is proved for all vertices $v$ that have a shortest path of length $k$. What claim again??
- Let $v$ be any vertex that has a shortest path $s \rightarrow u \rightarrow v$, consisting of $k + 1$ edges;
Chapter 24. Single Source Shortest Paths

We want to prove that, if a shortest path $s \leadsto v$ consists of $k$ edges, Bellman-Fold obtains value $v.d = \delta(s, v)$ after the $k$th round of relaxation (assuming there is no negative cycle).

Proof idea: Induction on $k$.

- $k = 0$, $v$ can only be $s$. Proved!
- Assume the claim is proved for all vertices $v$ that have a shortest path of length $k$. What claim again??
- Let $v$ be any vertex that has a shortest path $s \leadsto u \to v$, consisting of $k + 1$ edges;
  Then $s \leadsto u$ is a shortest path for $u$ consisting of $k$ edges;
We want to prove that, if a shortest path \( s \leadsto v \) consists of \( k \) edges, Bellman-Fold obtains value \( v.d = \delta(s, v) \) after the \( k \)th round of relaxation (assuming there is no negative cycle).

**Proof idea:** Induction on \( k \).

- \( k = 0 \), \( v \) can only be \( s \). Proved!

- Assume the **claim** is proved for all vertices \( v \) that have a shortest path of length \( k \). What claim again??

- Let \( v \) be any vertex that has a shortest path \( s \leadsto u \rightarrow v \), consisting of \( k + 1 \) edges;

  Then \( s \leadsto u \) is a shortest path for \( u \) consisting of \( k \) edges;

  Now by assumption, \( u.d = \delta(s, u) \) after \( k \) round of relaxation.
We want to prove that, if a shortest path \( s \leadsto v \) consists of \( k \) edges, Bellman-Fold obtains value \( v.d = \delta(s, v) \) after the \( k \)th round of relaxation (assuming there is no negative cycle).

\[ \text{Proof idea: Induction on } k. \]

- \( k = 0 \), \( v \) can only be \( s \). Proved!
- Assume the \textbf{claim} is proved for all vertices \( v \) that have a shortest path of length \( k \). What claim again??
- Let \( v \) be any vertex that has a shortest path \( s \leadsto u \rightarrow v \), consisting of \( k + 1 \) edges;

\[ \text{Then } s \leadsto u \text{ is a shortest path for } u \text{ consisting of } k \text{ edges}; \]

\[ \text{Now by assumption, } u.d = \delta(s, u) \text{ after } k \text{ round of relaxation.} \]

By \textbf{Convergence property Lemma}, \( v.d = \delta(s, v) \)
after another round of relaxation.
Lemma 24.15, Path-relaxation property: Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path from \( s = v_0 \) to \( v_k \). If a sequence relaxation steps occur that includes, in order, relaxing the edges \((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\), then \( v_k.d = \delta(s, v_k) \) after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of these edges.
Lemma 24.15, Path-relaxation property: Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path from $s = v_0$ to $v_k$. If a sequence relaxation steps occur that includes, in order, relaxing the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of these edges.

Proof:
Lemma 24.15, Path-relaxation property: Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path from $s = v_0$ to $v_k$. If a sequence relaxation steps occur that includes, in order, relaxing the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of these edges.

Proof: We prove by induction on $i$ that after the $i$th edge $(v_{i-1}, v_i)$ on path $p$ is relaxed, $v_i.d = \delta(s, v_i)$. 
**Chapter 24. Single Source Shortest Paths**

**Lemma 24.15, Path-relaxation property:** Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path from \( s = v_0 \) to \( v_k \). If a sequence relaxation steps occur that includes, in order, relaxing the edges \((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\), then \( v_k.d = \delta(s, v_k) \) after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of these edges.

**Proof:** We prove by induction on \( i \) that after the \( i \)th edge \((v_{i-1}, v_i)\) on path \( p \) is relaxed, \( v_i.d = \delta(s, v_i) \).

**basis:** \( i = 0 \). \( v_0 = s \), \( s.d = 0 = \delta(s, s) \)!
Lemma 24.15, Path-relaxation property: Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path from \( s = v_0 \) to \( v_k \). If a sequence relaxation steps occur that includes, in order, relaxing the edges \( (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \), then \( v_k.d = \delta(s, v_k) \) after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of these edges.

Proof: We prove by induction on \( i \) that after the \( i \)th edge \( (v_{i-1}, v_i) \) on path \( p \) is relaxed, \( v_i.d = \delta(s, v_i) \).

basis: \( i = 0 \). \( v_0 = s \), \( s.d = 0 = \delta(s, s) \)!
Assume: \( v_{i-1}.d = \delta(s, v_{i-1}) \).
Lemma 24.15, Path-relaxation property: Let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path from $s = v_0$ to $v_k$. If a sequence relaxation steps occur that includes, in order, relaxing the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of these edges.

Proof: We prove by induction on $i$ that after the $i$th edge $(v_{i-1}, v_i)$ on path $p$ is relaxed, $v_i.d = \delta(s, v_i)$.

basis: $i = 0$. $v_0 = s$, $s.d = 0 = \delta(s, s)$ !
Assume: $v_{i-1}.d = \delta(s, v_{i-1})$.
Induction: After we relax edge $(v_{i-1}, v_i)$, by convergence property, we have $v_i.d = \delta(s, v_i)$. 


Lemma 24.15, Path-relaxation property: Let \( p = (v_0, v_1, \ldots, v_k) \) be a shortest path from \( s = v_0 \) to \( v_k \). If a sequence relaxation steps occur that includes, in order, relaxing the edges \( (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \), then \( v_k.d = \delta(s, v_k) \) after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of these edges.

**Proof:** We prove by induction on \( i \) that after the \( i \)th edge \( (v_{i-1}, v_i) \) on path \( p \) is relaxed, \( v_i.d = \delta(s, v_i) \).

**basis:** \( i = 0 \). \( v_0 = s, s.d = 0 = \delta(s, s) \)!

**Assume:** \( v_{i-1}.d = \delta(s, v_{i-1}) \).

**Induction:** After we relax edge \( (v_{i-1}, v_i) \), by convergence property, we have \( v_i.d = \delta(s, v_i) \). And this holds for all times afterward.
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \rightarrow R$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$. 
Chapter 24. Single Source Shortest Paths

Correctness of \texttt{Bellman-Ford} algorithm

1. On graphs without negative cycles)

\textbf{Lemma 24.2} Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \rightarrow \mathbb{R}$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

\textbf{Proof}: (By induction on $k$,}

Chapter 24. Single Source Shortest Paths

Correctness of \textbf{Bellman-Ford} algorithm

1. On graphs without negative cycles

\textbf{Lemma 24.2} Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \to \mathbb{R}$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

\textbf{Proof:} (By induction on $k$, the number of edges on the computed path $p: s \xrightarrow{p} v$, to prove the claim to be true).
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

**Lemma 24.2** Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \rightarrow R$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

**Proof:** (By induction on $k$, the number of edges on the computed path $p : s \leadsto v$, to prove the claim to be true).
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \rightarrow R$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

Proof: (By induction on $k$, the number of edges on the computed path $p: s \xrightarrow{p} v$, to prove the claim to be true).

Base: $k = 0$. $v = s$. It is true.
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \rightarrow R$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

Proof: (By induction on $k$, the number of edges on the computed path $p: s \xrightarrow{P} v$, to prove the claim to be true).

Base: $k = 0$. $v = s$. It is true.
Assume: the claim is true for $k - 1$. 
Chapter 24. Single Source Shortest Paths

Correctness of \textbf{Bellman-Ford} algorithm

1. On graphs without negative cycles)

\textbf{Lemma 24.2} Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \to \mathbb{R}$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

\textbf{Proof:} (By induction on $k$, the number of edges on the computed path $p: s \leadsto v$, to prove the claim to be true).

- \textbf{Base:} $k = 0$. $v = s$. It is true.
- \textbf{Assume:} the claim is true for $k - 1$.
- \textbf{Induction:} computed path $p: s \leadsto v$ has $k$ edges and
  - $p$ arrives at $x$ before reaching $v$ via $(x, v)$.
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \to \mathbb{R}$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

Proof: (By induction on $k$, the number of edges on the computed path $p: s \xrightarrow{p} v$, to prove the claim to be true).

Base: $k = 0$. $v = s$. It is true.
Assume: the claim is true for $k - 1$.
Induction: computed path $p: s \xrightarrow{p} v$ has $k$ edges and $p$ arrives at $x$ before reaching $v$ via $(x, v)$. So $v.d = x.d + w(x, v)$
Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let \( G = (V, E) \) be a weighted, directed graph with source \( s \) and weight function \( w : E \rightarrow \mathbb{R} \) and assume that \( G \) contains no negative weight cycles that can be reached from \( s \). Then after \(|V| - 1\) iterations of line 5 in the algorithm, \( v.d = \delta(s, v) \) for all vertices \( v \) that are reachable from \( s \).

Proof: (By induction on \( k \), the number of edges on the computed path \( p: s \xrightarrow{p} v \), to prove the claim to be true).

Base: \( k = 0 \). \( v = s \). It is true.

Assume: the claim is true for \( k - 1 \).

Induction: computed path \( p: s \xrightarrow{p} v \) has \( k \) edges and

\( p \) arrives at \( x \) before reaching \( v \) via \((x, v)\). So \( v.d = x.d + w(x, v) \)

By Lemma 24.1, \( \delta(s, v) = \delta(s, y) + w(y, v) \) for some \( y \).
Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let \( G = (V, E) \) be a weighted, directed graph with source \( s \) and weight function \( w : E \to R \) and assume that \( G \) contains no negative weight cycles that can be reached from \( s \). Then after \(|V| - 1\) iterations of line 5 in the algorithm, \( v.d = \delta(s, v) \) for all vertices \( v \) that are reachable from \( s \).

Proof: (By induction on \( k \), the number of edges on the computed path \( p: s \xrightarrow{P} v \), to prove the claim to be true).

- **Base**: \( k = 0 \). \( v = s \). It is true.
- **Assume**: the claim is true for \( k - 1 \).
- **Induction**: computed path \( p: s \xrightarrow{P} v \) has \( k \) edges and \( p \) arrives at \( x \) before reaching \( v \) via \((x, v)\). So \( v.d = x.d + w(x, v) \)

By Lemma 24.1, \( \delta(s, v) = \delta(s, y) + w(y, v) \) for some \( y \).

Since after \( k \) iterations, \( v.d \) has been updated with the statement

\[ \text{if } v.d > u.d + w(u, v) \text{ then } v.d = u.d + w(u, v), \text{ for all } u, \text{ including } x, y \]
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \to R$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

Proof: (By induction on $k$, the number of edges on the computed path $p: s \xrightarrow{p} v$, to prove the claim to be true).

Base: $k = 0$. $v = s$. It is true.
Assume: the claim is true for $k - 1$.
Induction: computed path $p: s \xrightarrow{p} v$ has $k$ edges and $p$ arrives at $x$ before reaching $v$ via $(x, v)$. So $v.d = x.d + w(x, v)$

By Lemma 24.1, $\delta(s, v) = \delta(s, y) + w(y, v)$ for some $y$.

Since after $k$ iterations, $v.d$ has been updated with the statement if $v.d > u.d + w(u, v)$ then $v.d = u.d + w(u, v)$, for all $u$, including $x$, $y$ By the assumption, for every $u$, including $x$ and $y$, $u.d = \delta(s, u)$ because the computed path $s \xrightarrow{\sim} u$ contains $k - 1$ edges. So we have
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \to R$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

Proof: (By induction on $k$, the number of edges on the computed path $p: s \leadsto v$, to prove the claim to be true).

Base: $k = 0$. $v = s$. It is true.
Assume: the claim is true for $k - 1$.
Induction: computed path $p: s \leadsto v$ has $k$ edges and $p$ arrives at $x$ before reaching $v$ via $(x, v)$. So $v.d = x.d + w(x, v)$

By Lemma 24.1, $\delta(s, v) = \delta(s, y) + w(y, v)$ for some $y$.

Since after $k$ iterations, $v.d$ has been updated with the statement if $v.d > u.d + w(u, v)$ then $v.d = u.d + w(u, v)$, for all $u$, including $x$, $y$.

By the assumption, for every $u$, including $x$ and $y$, $u.d = \delta(s, u)$ because the computed path $s \leadsto u$ contains $k - 1$ edges. So we have

$v.d = x.d + w(x, v)$
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \to \mathbb{R}$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

Proof: (By induction on $k$, the number of edges on the computed path $p: s \xrightarrow{p} v$, to prove the claim to be true).

Base: $k = 0$. $v = s$. It is true.

Assume: the claim is true for $k - 1$.

Induction: computed path $p: s \xrightarrow{p} v$ has $k$ edges and $p$ arrives at $x$ before reaching $v$ via $(x, v)$. So $v.d = x.d + w(x, v)$

By Lemma 24.1, $\delta(s, v) = \delta(s, y) + w(y, v)$ for some $y$.

Since after $k$ iterations, $v.d$ has been updated with the statement if $v.d > u.d + w(u, v)$ then $v.d = u.d + w(u, v)$, for all $u$, including $x, y$. By the assumption, for every $u$, including $x$ and $y$, $u.d = \delta(s, u)$ because the computed path $s \xrightarrow{} u$ contains $k - 1$ edges. So we have

$v.d = x.d + w(x, v) \leq y.d + w(y, v)$
Chapter 24. Single Source Shortest Paths

Correctness of Bellman-Ford algorithm

1. On graphs without negative cycles)

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \rightarrow R$ and assume that $G$ contains no negative weight cycles that can be reached from $s$. Then after $|V| - 1$ iterations of line 5 in the algorithm, $v.d = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

Proof: (By induction on $k$, the number of edges on the computed path $p: s \leadsto v$, to prove the claim to be true).

Base: $k = 0$. $v = s$. It is true.
Assume: the claim is true for $k - 1$.
Induction: computed path $p: s \leadsto v$ has $k$ edges and $p$ arrives at $x$ before reaching $v$ via $(x, v)$. So $v.d = x.d + w(x, v)$

By Lemma 24.1, $\delta(s, v) = \delta(s, y) + w(y, v)$ for some $y$.

Since after $k$ iterations, $v.d$ has been updated with the statement if $v.d > u.d + w(u, v)$ then $v.d = u.d + w(u, v)$, for all $u$, including $x$, $y$.

By the assumption, for every $u$, including $x$ and $y$, $u.d = \delta(s, u)$ because the computed path $s \leadsto u$ contains $k - 1$ edges. So we have

$v.d = x.d + w(x, v) \leq y.d + w(y, v) = \delta(s, y) + w(y, v) = \delta(s, v)$
Theorem 24.4 Bellman-Ford algorithm is correct on weighted, directed graphs.
Chapter 24. Single Source Shortest Paths

Theorem 24.4 Bellman-Ford algorithm is correct on weighted, directed graphs.

Proof: By Lemma 24.2, we only need to show, when $G$ contains a negative weight cycle reachable from $s$, the algorithm returns FALSE.
Chapter 24. Single Source Shortest Paths

**Theorem 24.4** Bellman-Ford algorithm is correct on weighted, directed graphs.

**Proof:** By Lemma 24.2, we only need to show, when $G$ contains a negative weight cycle reachable from $s$, the algorithm returns FALSE.

Let the cycle to be $c = (v_0, v_1, \cdots, v_k)$, where $v_0 = v_k$ and
Theorem 24.4 Bellman-Ford algorithm is correct on weighted, directed graphs.

Proof: By Lemma 24.2, we only need to show, when $G$ contains a negative weight cycle reachable from $s$, the algorithm returns FALSE.

Let the cycle to be $c = (v_0, v_1, \cdots, v_k)$, where $v_0 = v_k$ and

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$
Chapter 24. Single Source Shortest Paths

**Theorem 24.4** Bellman-Ford algorithm is correct on weighted, directed graphs.

**Proof:** By Lemma 24.2, we only need to show, when $G$ contains a negative weight cycle reachable from $s$, the algorithm returns FALSE.

Let the cycle to be $c = (v_0, v_1, \cdots, v_k)$, where $v_0 = v_k$ and

$$
\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0
$$

Assume for all $i$, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$. 
Theorem 24.4 Bellman-Ford algorithm is correct on weighted, directed graphs.

Proof: By Lemma 24.2, we only need to show, when \( G \) contains a negative weight cycle reachable from \( s \), the algorithm returns FALSE.

Let the cycle to be \( c = (v_0, v_1, \cdots, v_k) \), where \( v_0 = v_k \) and
\[
\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0
\]

Assume for all \( i \), \( v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i) \).
\[
\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + w(v_{i-1}, v_i),
\]
Chapter 24. Single Source Shortest Paths

**Theorem 24.4** Bellman-Ford algorithm is correct on weighted, directed graphs.

**Proof**: By Lemma 24.2, we only need to show, when $G$ contains a negative weight cycle reachable from $s$, the algorithm returns FALSE.

Let the cycle to be $c = (v_0, v_1, \cdots, v_k)$, where $v_0 = v_k$ and

$$
\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0
$$

Assume for all $i$, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$.

$$
\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + w(v_{i-1}, v_i),
$$
Chapter 24. Single Source Shortest Paths

**Theorem 24.4** Bellman-Ford algorithm is correct on weighted, directed graphs.

**Proof:** By Lemma 24.2, we only need to show, when $G$ contains a negative weight cycle reachable from $s$, the algorithm returns FALSE.

Let the cycle to be $c = (v_0, v_1, \cdots, v_k)$, where $v_0 = v_k$ and

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

Assume for all $i$, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$.

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + w(v_{i-1}, v_i),$$

But

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d,$$
Chapter 24. Single Source Shortest Paths

**Theorem 24.4** Bellman-Ford algorithm is correct on weighted, directed graphs.

**Proof:** By Lemma 24.2, we only need to show, when $G$ contains a negative weight cycle reachable from $s$, the algorithm returns FALSE.

Let the cycle to be $c = (v_0, v_1, \cdots, v_k)$, where $v_0 = v_k$ and

$$
\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0
$$

Assume for all $i$, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$.

$$
\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + w(v_{i-1}, v_i),
$$

But

$$
\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d, \text{ implying } \sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0
$$
Chapter 24. Single Source Shortest Paths

**Theorem 24.4** Bellman-Ford algorithm is correct on weighted, directed graphs.

**Proof:** By Lemma 24.2, we only need to show, when \( G \) contains a negative weight cycle reachable from \( s \), the algorithm returns FALSE.

Let the cycle to be \( c = (v_0, v_1, \cdots, v_k) \), where \( v_0 = v_k \) and

\[
\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0
\]

Assume for all \( i \), \( v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i) \).

But

\[
\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d, \quad \text{implying} \quad \sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0
\]

contradicting to \( c \) being a negative cycle where \( \sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \)
Finding shortest paths on DAGs (directed acyclic graphs)
Chapter 24. Single Source Shortest Paths

Finding shortest paths on DAGs (directed acyclic graphs)
• Algorithms can take the advantage of the non-cyclicity.
Chapter 24. Single Source Shortest Paths

Finding shortest paths on DAGs (directed acyclic graphs)

- Algorithms can take the advantage of the non-cyclicity.
- How would your algorithm be?
Chapter 24. Single Source Shortest Paths

Finding shortest paths on DAGs (directed acyclic graphs)

- Algorithms can take the advantage of the non-cyclicity.
- How would your algorithm be?

topological order of vertices!
Chapter 24. Single Source Shortest Paths

Dag-Shortest Paths($G, w, s$)

1. topologically sort the vertices of $G.V$
2. for each vertex $v \in G.V$
3. $v.d = \infty$
4. $v.\pi = \text{NULL}$
5. $s.d = 0$
6. for each $u \in G.V$, in the topologically sorted order
7. for each vertex $v \in \text{Adj}[u]$
8. if $v.d > u.d + w(u,v)$
9. $v.d = u.d + w(u,v)$
10. $v.\pi = u$
11. return $(d, \pi)$

• Should we improve lines 6-7?
• Running time: ?
Chapter 24. Single Source Shortest Paths

DAG-SHORTEST PATHS\( (G, w, s) \)
1. topologically sort the vertices of \( G.V \)
Chapter 24. Single Source Shortest Paths

**DAG-Shortest Paths**($G, w, s$)

1. topologically sort the vertices of $G.V$
2. for each vertex $v \in G.V$

...
Chapter 24. Single Source Shortest Paths

**DAG-Shortest Paths** \((G, w, s)\)
1. topologically sort the vertices of \(G.V\)
2. for each vertex \(v \in G.V\)
3. \(v.d = \infty\)
Chapter 24. Single Source Shortest Paths

**DAG-SHORTEST PATHS**\((G, w, s)\)
1. topologically sort the vertices of \(G.V\)
2. for each vertex \(v \in G.V\)
3. \(v.d = \infty\)
4. \(v.\pi = NULL\)

• Should we improve lines 6-7?
• Running time: ?
Chapter 24. Single Source Shortest Paths

\[ \text{DAG-Shortest Paths}(G, w, s) \]

1. topologically sort the vertices of \( G.V \)
2. for each vertex \( v \in G.V \)
3. \( v.d = \infty \)
4. \( v.\pi = NULL \)
5. \( s.d = 0 \)
Chapter 24. Single Source Shortest Paths

DAG-SHORTEST PATHS\((G, w, s)\)
1. topologically sort the vertices of \(G.V\)
2. for each vertex \(v \in G.V\)
3. \(v.d = \infty\)
4. \(v.\pi = NULL\)
5. \(s.d = 0\)
6. for each \(u \in G.V\), in the topologically sorted order

• Should we improve lines 6-7?
• Running time: ?
Chapter 24. Single Source Shortest Paths

**DAG-Shortest Paths**$(G, w, s)$

1. topologically sort the vertices of $G.V$
2. for each vertex $v \in G.V$
3. $v.d = \infty$
4. $v.\pi = NULL$
5. $s.d = 0$
6. for each $u \in G.V$, in the topologically sorted order
7. for each vertex $v \in Adj[u]$
Chapter 24. Single Source Shortest Paths

DAG-SHORTEST PATHS\((G, w, s)\)
1. topologically sort the vertices of \(G.V\)
2. for each vertex \(v \in G.V\)
3. \(v.d = \infty\)
4. \(v.\pi = NULL\)
5. \(s.d = 0\)
6. for each \(u \in G.V\), in the topologically sorted order
7. for each vertex \(v \in Adj[u]\)
8. if \(v.d > u.d + w(u, v)\)

• Should we improve lines 6-7?
• Running time: ?
DAG-SHORTEST PATHS\((G, w, s)\)
1. topologically sort the vertices of \(G.V\)
2. for each vertex \(v \in G.V\)
3. \(v.d = \infty\)
4. \(v.\pi = NULL\)
5. \(s.d = 0\)
6. for each \(u \in G.V\), in the topologically sorted order
7. for each vertex \(v \in Adj[u]\)
8. if \(v.d > u.d + w(u, v)\)
9. \(v.d = u.d + w(u, v)\)


**Dag-Shortest Paths**\((G,w,s)\)
1. topologically sort the vertices of \(G.V\)
2. for each vertex \(v \in G.V\)
   3. \(v.d = \infty\)
   4. \(v.\pi = NULL\)
5. \(s.d = 0\)
6. for each \(u \in G.V\), in the topologically sorted order
   7. for each vertex \(v \in Adj[u]\)
5. if \(v.d > u.d + w(u,v)\)
9. \(v.d = u.d + w(u,v)\)
10. \(v.\pi = u\)

**Should we improve lines 6-7?**

**Running time:**?
Chapter 24. Single Source Shortest Paths

**DAG-Shortest Paths**($G, w, s$)
1. topologically sort the vertices of $G.V$
2. for each vertex $v \in G.V$
   3. $v.d = \infty$
   4. $v.\pi = NULL$
5. $s.d = 0$
6. for each $u \in G.V$, in the topologically sorted order
   7. for each vertex $v \in Adj[u]$
   8. if $v.d > u.d + w(u, v)$
   9. $v.d = u.d + w(u, v)$
   10. $v.\pi = u$
11. return $(d, \pi)$
Chapter 24. Single Source Shortest Paths

**DAG-Shortest Paths** ($G, w, s$)
1. topologically sort the vertices of $G.V$
2. for each vertex $v \in G.V$
3. $v.d = \infty$
4. $v.\pi = NULL$
5. $s.d = 0$
6. for each $u \in G.V$, in the topologically sorted order
7. for each vertex $v \in Adj[u]$
8. if $v.d > u.d + w(u, v)$
9. $v.d = u.d + w(u, v)$
10. $v.\pi = u$
11. return $(d, \pi)$

• Should we improve lines 6-7?

Running time: ?
Chapter 24. Single Source Shortest Paths

DAG-SHORTEST PATHS($G, w, s$)
1. topologically sort the vertices of $G.V$
2. for each vertex $v \in G.V$
3. $v.d = \infty$
4. $v.\pi = NULL$
5. $s.d = 0$
6. for each $u \in G.V$, in the topologically sorted order
7. for each vertex $v \in Adj[u]$
8. if $v.d > u.d + w(u, v)$
9. $v.d = u.d + w(u, v)$
10. $v.\pi = u$
11. return $(d, \pi)$

- Should we improve lines 6-7?
- Running time: ?
Chapter 24. Single Source Shortest Paths

note: the root is \( s \).
Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.
Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

Dijkstra(G, w, s)
Chapter 24. Single Source Shortest Paths

Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\[ \text{Dijkstra}(G, w, s) \]

1. for each vertex \( v \in G.V \)
Chapter 24. Single Source Shortest Paths

Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\textsc{Dijkstra}(G, w, s)

1. for each vertex \( v \in G.V \)
   \( v.d = \infty \)

2. \( s.d = 0 \)

3. \( S = \emptyset \)

4. \( Q = G.V \)

5. while \( Q \) is not empty

6. \( u = \text{Extract Min}(Q) \)

7. \( S = S \cup \{ u \} \)

8. for each vertex \( v \in \text{Adj}[u] \)

9. if \( v.d > u.d + w(u,v) \)

10. \( v.d = u.d + w(u,v) \)

11. \( v.\pi = u \)

12. return \((d, \pi)\)

Running time: ?
Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\text{Dijkstra}(G, w, s)
1. for each vertex \( v \in G.V \)
2. \( v.d = \infty \)
3. \( v.\pi = NULL \)

Running time:
Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\textsc{Dijkstra}(G, w, s)
1. for each vertex \( v \in G.V \)
2. \( \quad v.d = \infty \)
3. \( \quad v.\pi = \text{NULL} \)
4. \( \quad s.d = 0 \)
Chapter 24. Single Source Shortest Paths

Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\textsc{Dijkstra} (G, w, s)

1. for each vertex \( v \in G.V \)
2. \( v.d = \infty \)
3. \( v.\pi = NULL \)
4. \( s.d = 0 \)
5. \( S = \emptyset \)
Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\textsc{Dijkstra}(G, w, s)
1. for each vertex \( v \in G.V \)
2. \( v.d = \infty \)
3. \( v.\pi = NULL \)
4. \( s.d = 0 \)
5. \( S = \emptyset \)
6. \( Q = G.V \)
Chapter 24. Single Source Shortest Paths

Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\textbf{Dijkstra}(G, w, s)
1. for each vertex \( v \in G.V \)
2. \( v.d = \infty \)
3. \( v.\pi = NULL \)
4. \( s.d = 0 \)
5. \( S = \emptyset \)
6. \( Q = G.V \)
7. \textbf{while} \( Q \) is not empty

[Algorithms and Code]

Running time:?
Chapter 24. Single Source Shortest Paths

**Dijkstra’s algorithm**

On weighted, directed graphs in which each edge has non-negative weight.

**DIJKSTRA***(G, w, s)*

1. for each vertex \( v \in G.V \)
2. \( v.d = \infty \)
3. \( v.\pi = NULL \)
4. \( s.d = 0 \)
5. \( S = \emptyset \)
6. \( Q = G.V \)
7. while \( Q \) is not empty
8. \( u = EXTRACT\ MIN\ (Q) \)
Chapter 24. Single Source Shortest Paths

Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\[
\text{DIJKSTRA}(G, w, s) \\
1. \text{ for each vertex } v \in G.V \\
2. \quad v.d = \infty \\
3. \quad v.\pi = NULL \\
4. \quad s.d = 0 \\
5. \quad S = \emptyset \\
6. \quad Q = G.V \\
7. \text{ while } Q \text{ is not empty} \\
8. \quad u = \text{EXTRACT MIN}(Q) \\
9. \quad S = S \cup \{u\}
\]
Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\[ \text{DIJKSTRA}(G, w, s) \]
1. for each vertex \( v \in G.V \)
2. \( v.d = \infty \)
3. \( v.\pi = NULL \)
4. \( s.d = 0 \)
5. \( S = \emptyset \)
6. \( Q = G.V \)
7. while \( Q \) is not empty
8. \( u = \text{EXTRACT-MIN}(Q) \)
9. \( S = S \cup \{u\} \)
10. for each vertex \( v \in \text{Adj}[u] \)
Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

**Dijkstra**\( (G, w, s) \)
1. for each vertex \( v \in G.V \)
2. \( v.d = \infty \)
3. \( v.\pi = NULL \)
4. \( s.d = 0 \)
5. \( S = \emptyset \)
6. \( Q = G.V \)
7. while \( Q \) is not empty
8. \( u = \text{Extract Min} (Q) \)
9. \( S = S \cup \{u\} \)
10. for each vertex \( v \in Adj[u] \)
11. if \( v.d > u.d + w(u, v) \)
Chapter 24. Single Source Shortest Paths

Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

Dijkstra\((G, w, s)\)
1. for each vertex \(v \in G.V\)
2. \(v.d = \infty\)
3. \(v.\pi = NULL\)
4. \(s.d = 0\)
5. \(S = \emptyset\)
6. \(Q = G.V\)
7. while \(Q\) is not empty
8. \(u = \text{Extract Min} (Q)\)
9. \(S = S \cup \{u\}\)
10. for each vertex \(v \in Adj[u]\)
11. if \(v.d > u.d + w(u, v)\)
12. \(v.d = u.d + w(u, v)\)
Chapter 24. Single Source Shortest Paths

Dijkstra’s algorithm

On weighted, directed graphs in which each edge has non-negative weight.

Dijkstra(G, w, s)
1. for each vertex v ∈ G.V
2. v.d = ∞
3. v.π = NULL
4. s.d = 0
5. S = ∅
6. Q = G.V
7. while Q is not empty
8. u = Extract Min (Q)
9. S = S ∪ {u}
10. for each vertex v ∈ Adj[u]
11. if v.d > u.d + w(u, v)
12. v.d = u.d + w(u, v)
13. v.π = u
Chapter 24. Single Source Shortest Paths

**Dijkstra’s algorithm**

On weighted, directed graphs in which each edge has non-negative weight.

**Dijkstra**\((G, w, s)\)
1. for each vertex \(v \in G.V\)
2. \(v.d = \infty\)
3. \(v.\pi = \text{NULL}\)
4. \(s.d = 0\)
5. \(S = \emptyset\)
6. \(Q = G.V\)
7. while \(Q\) is not empty
8. \(u = \text{Extract Min}(Q)\)
9. \(S = S \cup \{u\}\)
10. for each vertex \(v \in \text{Adj}[u]\)
11. if \(v.d > u.d + w(u, v)\)
12. \(v.d = u.d + w(u, v)\)
13. \(v.\pi = u\)
14. return \((d, \pi)\)
Chapter 24. Single Source Shortest Paths

Dijkstra's algorithm

On weighted, directed graphs in which each edge has non-negative weight.

\[ \text{DIJKSTRA}(G, w, s) \]
1. for each vertex \( v \in G.V \)
2. \( v.d = \infty \)
3. \( v.\pi = \text{NULL} \)
4. \( s.d = 0 \)
5. \( S = \emptyset \)
6. \( Q = G.V \)
7. while \( Q \) is not empty
8. \( u = \text{EXTRACT MIN} (Q) \)
9. \( S = S \cup \{u\} \)
10. for each vertex \( v \in \text{Adj}[u] \)
11. if \( v.d > u.d + w(u, v) \)
12. \( v.d = u.d + w(u, v) \)
13. \( v.\pi = u \)
14. return \( (d, \pi) \)

Running time:?
Note: the black-colored vertices are in set \( S \).
Chapter 24. Single Source Shortest Paths

Correctness of algorithm Dijkstra

Theorem 24.6

Dijkstra's algorithm, run on a weighted, directed graph $G = (V,E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s,u)$ for all vertices $u \in V$.

Proof: We need to show the while loop has loop invariant: $u.d = \delta(s,u)$ for each $u \in S$.

Assume $u$ to be the first such vertex that $u.d > \delta(s,u)$ when it is being added to $S$.

Then there must be a shortest path $p: s \rightarrow x \rightarrow y \rightarrow u$, for some $x \in S$ and some $y \not\in S$.

$y.d = \delta(s,y)$ when $u$ is being added to $S$. This is because $x \in S$, $x.d = \delta(s,x)$ when $x$ was added to $S$.

Edge $(x,y)$ was related at that time, and $y.d = \delta(s,y)$ by Convergence-property.

So when $u$ was chosen, $u.d \leq y.d = \delta(s,y) \leq \delta(s,u)$. Contradicts the choice of $u$.

So $u.d = \delta(s,u)$ when it is being included to $S$. 
Chapter 24. Single Source Shortest Paths

Correctness of algorithm \textsc{Dijkstra}

\textbf{Theorem 24.6} Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$. 

\begin{proof}

We need to show the \texttt{while} loop has loop invariant:

\begin{enumerate}
  \item $u.d = \delta(s, u)$ for each $u \in S$
\end{enumerate}

Assume $u$ to be the first such vertex that $u.d > \delta(s, u)$ when it is being added to $S$ then there must be a shortest path $p: s \rightarrow x \rightarrow y \rightarrow u$, for some $x \in S$ and some $y \not\in S$.

$\ y.d = \delta(s, y) \ $when $\ u \ $is being added to $\ S$.

This is because $x \in S$, $x.d = \delta(s, x)$ when $x$ was added to $S$.

So when $u$ was chosen, $u.d \leq y.d = \delta(s, y) \leq \delta(s, u)$.

Contradicts the choice of $u$.

So $u.d = \delta(s, u)$ when it is being included to $S$.
\end{proof}
Chapter 24. Single Source Shortest Paths

Correctness of algorithm Dijkstra

Theorem 24.6 Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof: We need to show the while loop has loop invariant:
Correctness of algorithm \textsc{Dijkstra}

\textbf{Theorem 24.6} Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

\textbf{Proof}: We need to show the \textbf{while} loop has \textit{loop invariant}: $u.d = \delta(s, u)$ for each $u \in S$. 

Correctness of algorithm Dijkstra

Theorem 24.6 Dijkstra’s algorithm, run on a weighted, directed graph \( G = (V, E) \) with non-negative weight function \( w \) and source \( s \), terminates with \( u.d = \delta(s, u) \) for all vertices \( u \in V \).

Proof: We need to show the while loop has loop invariant:
\( u.d = \delta(s, u) \) for each \( u \in S \)

Assume \( u \) to be the first such vertex that \( u.d > \delta(s, u) \) when it is being added to \( S \).
Chapter 24. Single Source Shortest Paths

Correctness of algorithm \textsc{Dijkstra}

\textbf{Theorem 24.6} Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

\textbf{Proof:} We need to show the \textbf{while} loop has loop invariant:

$u.d = \delta(s, u)$ for each $u \in S$

Assume $u$ to be the first such vertex that $u.d > \delta(s, u)$ when it is being added to $S$ then there must be a shortest path $p: s \leadsto x \rightarrow y \leadsto u$, for some $x \in S$ and some $y \notin S$. 
Correctness of algorithm **Dijkstra**

**Theorem 24.6** Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

**Proof**: We need to show the **while** loop has loop invariant: $u.d = \delta(s, u)$ for each $u \in S$

Assume $u$ to be the first such vertex that $u.d > \delta(s, u)$ when it is being added to $S$ then there must be a shortest path $p$: $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$, for some $x \in S$ and some $y \notin S$.

$y.d = \delta(s, y)$ **when** $u$ is being added to $S$. This is because $x \in S$, $x.d = \delta(s, x)$ when $x$ was added to $S$. 
Chapter 24. Single Source Shortest Paths

Correctness of algorithm Dijkstra

Theorem 24.6 Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof: We need to show the while loop has loop invariant:

$u.d = \delta(s, u)$ for each $u \in S$

Assume $u$ to be the first such vertex that $u.d > \delta(s, u)$ when it is being added to $S$ then there must be a shortest path $p$: $s \leadsto x \rightarrow y \leadsto u$, for some $x \in S$ and some $y \notin S$.

$y.d = \delta(s, y)$ when $u$ is being added to $S$. This is because $x \in S$, $x.d = \delta(s, x)$ when $x$ was added to $S$. Edge $(x, y)$ was related at that time, and $y.d = \delta(s, y)$ by Convergence-property.
Chapter 24. Single Source Shortest Paths

Correctness of algorithm Dijkstra

**Theorem 24.6** Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

**Proof**: We need to show the while loop has loop invariant:
$u.d = \delta(s, u)$ for each $u \in S$

Assume $u$ to be the first such vertex that $u.d > \delta(s, u)$ when it is being added to $S$ then there must be a shortest path $p$: $s \leadsto x \rightarrow y \leadsto u$, for some $x \in S$ and some $y \notin S$.

$y.d = \delta(s, y)$ when $u$ is being added to $S$. This is because $x \in S$, $x.d = \delta(s, x)$ when $x$ was added to $S$. Edge $(x, y)$ was related at that time, and $y.d = \delta(s, y)$ by Convergence-property. So
Chapter 24. Single Source Shortest Paths

Correctness of algorithm Dijkstra

Theorem 24.6 Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof: We need to show the while loop has loop invariant: 
$u.d = \delta(s, u)$ for each $u \in S$

Assume $u$ to be the first such vertex that $u.d > \delta(s, u)$ when it is being added to $S$ then there must be a shortest path $p$: $s \leadsto x \rightarrow y \leadsto u$, for some $x \in S$ and some $y \notin S$.

$y.d = \delta(s, y)$ when $u$ is being added to $S$. This is because $x \in S$, $x.d = \delta(s, x)$ when $x$ was added to $S$. Edge $(x, y)$ was related at that time, and $y.d = \delta(s, y)$ by Convergence-property. So

When $u$ was chosen, $u.d \leq y.d = \delta(s, y) \leq \delta(s, u)$. Contradicts the choice of $u$. 

**Chapter 24. Single Source Shortest Paths**

**Correctness of algorithm Dijkstra**

**Theorem 24.6** Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function $w$ and source $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

**Proof:** We need to show the while loop has loop invariant: $u.d = \delta(s, u)$ for each $u \in S$

Assume $u$ to be the first such vertex that $u.d > \delta(s, u)$ when it is being added to $S$ then there must be a shortest path $p$: $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$, for some $x \in S$ and some $y \notin S$.

$y.d = \delta(s, y)$ when $u$ is being added to $S$. This is because $x \in S$, $x.d = \delta(s, x)$ when $x$ was added to $S$. Edge $(x, y)$ was related at that time, and $y.d = \delta(s, y)$ by **Convergence-property**. So

When $u$ was chosen, $u.d \leq y.d = \delta(s, y) \leq \delta(s, u)$. **Contradicts** the choice of $u$. So $u.d = \delta(s, u)$ when it is being included to $S$. 
• Running time of Dijkstra?
Chapter 24. Single Source Shortest Paths

- Running time of Dijkstra?
- Can Dijkstra deals with negative edges or cycles?
Chapter 24. Single Source Shortest Paths

- Running time of Dijkstra?
- Can Dijkstra deal with negative edges or cycles?
- Fundamental differences between Bellman-Ford and Dijkstra?
Chapter 24. Single Source Shortest Paths

- Running time of Dijkstra?
- Can Dijkstra deals with negative edges or cycles?
- Fundamental differences between Bellman-Ford and Dijkstra?

**Dijkstra**(G, w, s)
1. for each vertex v ∈ G.V
2. v.d = ∞
3. v.π = NULL
4. s.d = 0
5. S = ∅
6. Q = G.V
7. while Q is not empty
8. u = EXTRACT MIN (Q)
9. S = S ∪ {u}
10. for each vertex v ∈ Adj[u]
11. if v.d > u.d + w(u, v)
12. v.d = u.d + w(u, v)
13. v.π = u
14. return (d, π)

**Bellman-Ford**(G, w, s)
1. for each vertex v ∈ G.V
2. v.d = ∞
3. v.π = NULL
4. s.d = 0
5. for i = 1 to |V| − 1
6. for each edge (u, v) ∈ G.E
7. if v.d > u.d + w(u, v)
8. v.d = u.d + w(u, v)
9. v.π = u
10. for each edge (u, v) ∈ G.E
11. if v.d > u.d + w(u, v)
12. return (FALSE)
13. return (TRUE)
Chapter 24. Single Source Shortest Paths
Chapter 24. Single Source Shortest Paths

- Fundamental differences between Dijkstra and MST-Prim?
Chapter 24. Single Source Shortest Paths

- Fundamental differences between Dijkstra and MST-Prim?

\begin{align*}
\text{Dijkstra}(G, w, s) & : \\
1. & \text{for each vertex } v \in G.V \\
2. & v.d = \infty \\
3. & v.\pi = NULL \\
4. & s.d = 0 \\
5. & S = \emptyset \\
6. & Q = G.V \\
7. & \text{while } Q \text{ is not empty} \\
8. & u = \text{EXTRACT MIN}(Q) \\
9. & S = S \cup \{u\} \\
10. & \text{for each vertex } v \in \text{Adj}[u] \\
11. & \text{if } v.d > u.d + w(u, v) \\
12. & v.d = u.d + w(u, v) \\
13. & v.\pi = u \\
14. & \text{return } (d, \pi)
\end{align*}

\begin{align*}
\text{MST-Prim}(G, w, r) & : \\
1. & \text{for each } u \in G.V \\
2. & u.key = \infty \quad \{ u.key \text{ is } \infty \}
3. & u.\pi = NULL \quad \{ \text{start from vertex } r \}
4. & r.key = 0 \\
5. & Q = G.V \\
6. & \text{while } Q \neq \emptyset \\
7. & u = \text{EXTRACT MIN}(Q) \\
8. & \text{for each } v \in \text{Adj}[u] \\
9. & \text{if } v \in Q \text{ and } w(u, v) < v.key \\
10. & \quad \text{then } v.\pi = u \\
11. & \quad v.key = w(u, v) \\
12. & \text{return } \pi
\end{align*}
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem

**Input**: A weighted graph \( G = (V, E) \) with edge weight function \( w \);

**Output**: Shortest paths between every pair of vertices in \( G \).

- **Dijkstra**: runs in time \( O(|V|^2 \log |V| + |V||E|) \) on non-negative edges.
- **Bellman-Ford**: runs in time \( O(|V|^2 |E|) \) for general graphs, but \( O(|V|^4) \) on "dense" graphs.

**New algorithms**

- **Dynamic programming algorithm**: \( O(|V|^4) \), improved to \( O(|V|^3 \log |V|) \).
- **Floyd-Warshall algorithm**: \( O(|V|^3) \).

**Graph representation**: adjacency matrix \( W = (w_{ij}) \).
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem

**INPUT:** A weighted graph $G = (V, E)$ with edge weight function $w$;
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem

**Input:** A weighted graph $G = (V, E)$ with edge weight function $w$;

**Output:** Shortest paths between every pair of vertices in $G$. 

• Dijkstra would run in time $O(|V|^2 \log |V| + |V||E|)$ on non-negative edges

• Bellman-Ford would run in time $O(|V|^2 |E|)$ for general graphs, but $O(|V|^4)$ on “dense” graphs

New algorithms

• A dynamic programming algorithm $O(|V|^4)$, improved to $O(|V|^3 \log |V|)$

• Floyd-Warshall algorithm: $O(|V|^3)$.

Graph representation: adjacency matrix $W = (w_{ij})$
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem

**Input:** A weighted graph $G = (V, E)$ with edge weight function $w$;

**Output:** Shortest paths between every pair of vertices in $G$.

- **Dijkstra** would run in time $O(|V|^2 \log |V| + |V||E|)$ on non-negative edges

Graph representation: adjacency matrix $W = (w_{ij})$
All Pair Shortest Paths Problem

**Input:** A weighted graph $G = (V, E)$ with edge weight function $w$;

**Output:** Shortest paths between every pair of vertices in $G$.

- **Dijkstra** would run in time $O(|V|^2 \log |V| + |V||E|)$ on non-negative edges
- **Bellman-Ford** would run in time $O(|V|^2|E|)$ for general graphs, but $O(|V|^4)$ on "dense" graphs
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem

**Input:** A weighted graph \( G = (V, E) \) with edge weight function \( w \);

**Output:** Shortest paths between every pair of vertices in \( G \).

- **Dijkstra** would run in time \( O(|V|^2 \log |V| + |V||E|) \) on non-negative edges
- **Bellman-Ford** would run in time \( O(|V|^2 |E|) \) for general graphs, but \( O(|V|^4) \) on "dense" graphs

New algorithms
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem

**Input:** A weighted graph \( G = (V, E) \) with edge weight function \( w \);

**Output:** Shortest paths between every pair of vertices in \( G \).

- **Dijkstra** would run in time \( O(|V|^2 \log |V| + |V||E|) \) on non-negative edges
- **Bellman-Ford** would run in time \( O(|V|^2|E|) \) for general graphs, but \( O(|V|^4) \) on "dense" graphs

New algorithms

- A dynamic programming algorithm \( O(|V|^4) \), improved to \( O(|V|^3 \log |V|) \)
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem

**Input:** A weighted graph \( G = (V, E) \) with edge weight function \( w \);

**Output:** Shortest paths between every pair of vertices in \( G \).

- **Dijkstra** would run in time \( O(|V|^2 \log |V| + |V||E|) \) on non-negative edges
- **Bellman-Ford** would run in time \( O(|V|^2|E|) \) for general graphs, but \( O(|V|^4) \) on "dense" graphs

New algorithms

- A dynamic programming algorithm \( O(|V|^4) \), improved to \( O(|V|^3 \log |V|) \)
- **Floyd-Warshall** algorithm: \( O(|V|^3) \).
Chapter 25. All-pairs shortest paths

All Pair Shortest Paths Problem

**Input:** A weighted graph $G = (V, E)$ with edge weight function $w$;  
**Output:** Shortest paths between every pair of vertices in $G$.

- **Dijkstra** would run in time $O(|V|^2 \log |V| + |V||E|)$ on non-negative edges.
- **Bellman-Ford** would run in time $O(|V|^2|E|)$ for general graphs, but $O(|V|^4)$ on "dense" graphs.

New algorithms

- A dynamic programming algorithm $O(|V|^4)$, improved to $O(|V|^3 \log |V|)$.
- Floyd-Warshall algorithm: $O(|V|^3)$.

Graph representation: adjacency matrix $W = (w_{ij})$.
Chapter 25. All-pairs shortest paths

A dynamic programming approach
Chapter 25. All-pairs shortest paths

A dynamic programming approach

- Optimal substructure
- Objective function
Chapter 25. All-pairs shortest paths

A dynamic programming approach

- Optimal substructure
- Objective function

Define $l_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$
A dynamic programming approach

- Optimal substructure
- Objective function

Define $l_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$

**does not work!** having a data dependency issue.
Chapter 25. All-pairs shortest paths

A dynamic programming approach

- Optimal substructure
- Objective function

Define $l_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ does not work! having a data dependency issue.

Define $l^m_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ that contains at most $m$ edges.
A dynamic programming approach

- Optimal substructure
- Objective function

Define $l_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ does not work! having a data dependency issue.

Define $l_{ij}^m$ be the minimum weight of any path from $v_i$ to $v_j$ that contains at most $m$ edges.

or alternatively,
Chapter 25. All-pairs shortest paths

A dynamic programming approach

- Optimal substructure
- Objective function

Define $l_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ does not work! having a data dependency issue.

Define $l^m_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ that contains at most $m$ edges.

or alternatively,

Define $l^k_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ in which intermediate vertices have indexes $\leq k$. 
Define $l^m_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ that contains at most $m$ edges.
Define $l^m_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ that contains at most $m$ edges.

$$l^m_{ij} = \min(l^{m-1}_{ij}, \min_{1\leq k\leq n} \{l^{m-1}_{ik} + w_{kj}\})$$
Define $l^m_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ that contains at most $m$ edges.

$$l^m_{ij} = \min(l^{m-1}_{ij}, \min_{1 \leq k \leq n} \{l^{m-1}_{ik} + w_{kj}\})$$

If $w_{jj} = 0$, we can rewrite

$$l^m_{ij} = \min_{1 \leq k \leq n} \{l^{m-1}_{ik} + w_{kj}\}$$
Chapter 25. All-pairs shortest paths

Define $l^m_{ij}$ be the minimum weight of any path from $v_i$ to $v_j$ that contains at most $m$ edges.

$$l^m_{ij} = \min(l^{m-1}_{ij}, \min_{1 \leq k \leq n} \{l^m_{ik} + w_{kj}\})$$

If $w_{jj} = 0$, we can rewrite

$$l^m_{ij} = \min_{1 \leq k \leq n} \{l^{m-1}_{ik} + w_{kj}\}$$

and base cases:

$$l^1_{ij} = w_{ij}$$
Chapter 25. All-pairs shortest paths

Define $l_{ij}^m$ be the minimum weight of any path from $v_i$ to $v_j$ that contains at most $m$ edges.

$$l_{ij}^m = \min(l_{ij}^{m-1}, \min_{1 \leq k \leq n} \{l_{ik}^{m-1} + w_{kj}\})$$

If $w_{jj} = 0$, we can rewrite

$$l_{ij}^m = \min_{1 \leq k \leq n} \{l_{ik}^{m-1} + w_{kj}\}$$

and base cases:

$$l_{ij}^1 = w_{ij}$$

Adjacency matrix $W = (w_{ij})$ is the default.
Chapter 25. All-pairs shortest paths

DP table filling algorithm:
Chapter 25. All-pairs shortest paths

DP table filling algorithm:
For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$;
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

**Extended Shortest Paths**($L, W$)
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

**Extended Shortest Paths**($L, W$)
1. $n = \text{rows}[L]$;
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

**Extended Shortest Paths** ($L, W$)
1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;

```plaintext
for i = 1 to n
  for j = 1 to n
    $L'[i,j] = \infty$  ($L'[i,j] = L[i,j]$ in case $w_{a,a} \neq 0$)
  for k = 1 to n
    $L'[i,j] = \min \{L'[i,j], L[i,k] + w[k,j]\}$
return $(L')$
```
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

Extended Shortest Paths ($L, W$)

1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;
3. for $i = 1$ to $n$
DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

**Extended Shortest Paths**$(L, W)$

1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;
3. for $i = 1$ to $n$
4. for $j = 1$ to $n$
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

Extended Shortest Paths($L, W$)
1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;
3. for $i = 1$ to $n$
4. for $j = 1$ to $n$
5. $L'[i,j] = \infty$ \quad ($L'[i,j] = L[i,j]$ in case $w_{a,a} \neq 0$)
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

**Extended Shortest Paths** $(L, W)$
1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;
3. for $i = 1$ to $n$
4. \hspace{1em} for $j = 1$ to $n$
5. \hspace{2em} $L'[i, j] = \infty \quad (L'[i, j] = L[i, j]$ in case $w_{a,a} \neq 0)$
6. \hspace{1em} for $k = 1$ to $n$
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

Extended Shortest Paths $(L, W)$
1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;
3. for $i = 1$ to $n$
4.    for $j = 1$ to $n$
5.        $L'[i, j] = \infty$ \hspace{1em} ($L'[i, j] = L[i, j]$ in case $w_{a,a} \neq 0$)
6.    for $k = 1$ to $n$
7.        $L'[i, j] = \min \{L'[i, j], L[i, k] + w[k, j]\}$
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

Extended Shortest Paths($L, W$)
1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;
3. for $i = 1$ to $n$
4.     for $j = 1$ to $n$
5.         $L'[i, j] = \infty$ \hspace{1cm} ($L'[i, j] = L[i, j]$ in case $w_{a,a} \neq 0$)
6.     for $k = 1$ to $n$
7.         $L'[i, j] = \min\{L'[i, j], L[i, k] + w[k, j]\}$
8. return ($L'$)
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

**Extended Shortest Paths** $(L,W)$

1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;
3. for $i = 1$ to $n$
   4. for $j = 1$ to $n$
   5. $L'[i,j] = \infty$ (in case $w_{a,a} \neq 0$)
   6. for $k = 1$ to $n$
   7. $L'[i,j] = \min\{L'[i,j], L[i,k] + w[k,j]\}$
8. return $L'$

Call Extended Shortest Paths for $m = 2, 3, \ldots, n - 1$
Chapter 25. All-pairs shortest paths

DP table filling algorithm:

For $L^1 = W$ and $m = 2, \ldots, n - 1$, compute table $L^m$ from table $L^{m-1}$; technically two tables are enough.

**Extended Shortest Paths** $(L, W)$
1. $n = \text{rows}[L]$;
2. let $L'$ be an $n \times n$ table;
3. for $i = 1$ to $n$
4. \hspace{1em} for $j = 1$ to $n$
5. \hspace{2em} $L'[i, j] = \infty \quad (L'[i, j] = L[i, j] \text{ in case } w_{a, a} \neq 0)$
6. \hspace{1em} for $k = 1$ to $n$
7. \hspace{2em} $L'[i, j] = \min\{L'[i, j], L[i, k] + w[k, j]\}$
8. return $(L')$

Call **Extended Shortest Paths** for $m = 2, 3, \ldots, n - 1$

\[ L^m \leftarrow \text{Extended Shortest Paths}(L^{m-1}, W) \]
Running on an example:
Running on an example:

\[ W = L^1 = \text{the first matrix.} \]

\[
l^2_{0,0} = \min \begin{cases} 
  l^1_{0,0} & \text{value} = 8 \\
  l^1_{0,0} + l^1_{0,0} & k = 0, \text{value} = 8 + 8 = 16 \\
  l^1_{0,1} + l^1_{1,0} & k = 1, \text{value} = 1 + 6 = 7 \\
  l^1_{0,2} + l^1_{2,0} & k = 2, \text{value} = 1 + 3 = 4^* 
\end{cases}
\]
Chapter 25. All-pairs shortest paths

• running time: $\Theta(n^4)$.
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.
- improving the running time by repeatedly squaring:
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.
- improving the running time by repeatedly squaring:
  
  compute: $L^1, L^2, L^4, \ldots, L^{2^k}$. 

\[2^k = n - 1 \implies k = \lceil \log_2(n - 1) \rceil.\]
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.
- improving the running time by repeatedly squaring:
  
  compute: $L^1, L^2, L^4, \ldots, L^{2^k}$.

  what is $k$ here?
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.
- improving the running time by repeatedly squaring:
  compute: $L^1, L^2, L^4, \cdots, L^{2^k}$.
  what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$. 

Faster All Pair Shortest Paths

1. $n = \text{rows}[W];$
2. $L = W;$
3. $m = 1;$
4. while $m < n - 1$
5. $L = \text{Extended Shortest Paths}(L, L);$ 
6. $m = 2 \times m;$
7. return $(L)$. 

Chapter 25. All-pairs shortest paths

• running time: $\Theta(n^4)$.

• improving the running time by repeatedly squaring:

  compute: $L^1, L^2, L^4, \ldots, L^{2^k}$.

  what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$.

Faster All Pair Shortest Paths($W$)
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.
- improving the running time by repeatedly squaring:
  compute: $L^1, L^2, L^4, \cdots, L^{2^k}$.

  what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$.

Faster All Pair Shortest Paths($W$)
1. $n = rows[W]$;
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.
- improving the running time by repeatedly squaring:

  compute: $L^1, L^2, L^4, \cdots, L^{2^k}$.

  what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$.

**Faster All Pair Shortest Paths($W$)**

1. $n = \text{rows}[W]$;
2. $L = W$;
Chapter 25. All-pairs shortest paths

• running time: $\Theta(n^4)$.

• improving the running time by repeatedly squaring:

  compute: $L^1, L^2, L^4, \ldots, L^{2^k}$.

  what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$.

Faster All Pair Shortest Paths($W$)
1. $n = \text{rows}[W]$;
2. $L = W$;
3. $m = 1$;
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.

- improving the running time by repeatedly squaring:

  compute: $L^1, L^2, L^4, \ldots, L^{2^k}$.

  what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$.

**Faster All Pair Shortest Paths** ($W$)

1. $n = \text{rows}[W]$;
2. $L = W$;
3. $m = 1$;
4. **while** $m < n - 1$
• running time: $\Theta(n^4)$.

• improving the running time by repeatedly squaring:
  
  compute: $L^1$, $L^2$, $L^4$, $\ldots$, $L^{2^k}$.

  what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$.

Faster All Pair Shortest Paths($W$)
1. $n = \text{rows}[W]$;
2. $L = W$;
3. $m = 1$;
4. while $m < n - 1$
5. $L = \text{Extended Shortest Paths}(L, L)$
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.
- improving the running time by repeatedly squaring:
  - compute: $L^1, L^2, L^4, \cdots, L^{2^k}$.
  - what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$.

**Faster All Pair Shortest Paths** $(W)$
1. $n = \text{rows}[W]$;
2. $L = W$;
3. $m = 1$;
4. **while** $m < n - 1$
5. $L = \text{Extended Shortest Paths}(L, L)$
6. $m = 2 \times m$
Chapter 25. All-pairs shortest paths

- running time: $\Theta(n^4)$.
- improving the running time by repeatedly squaring:
  
  compute: $L^1, L^2, L^4, \ldots, L^{2^k}$.

  what is $k$ here? Let $2^k = n - 1$. Then $k = \lceil \log_2(n - 1) \rceil$.

Faster All Pair Shortest Paths ($W$)
1. $n = \text{rows}[W]$;
2. $L = W$;
3. $m = 1$;
4. while $m < n - 1$
5.   $L = \text{Extended Shortest Paths}(L, L)$
6.   $m = 2 \times m$
7. return ($L$)
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

Define: $d(k)_{ij}$ to be the shortest path distance from $v_i$ to $v_j$ with no intermediate vertices of indexes higher than $k$.

Thus $d(k)_{ij} = \min\{d(k-1)_{ij}, d(k-1)_{ik} + d(k-1)_{kj}\}$ with base case: $d(0)_{ij} = w_{ij}$.

Floyd-Warshall (W)

1. $n = \text{rows}[W]$
2. $D(0) = W$
3. for $k = 1$ to $n$
4. for $i = 1$ to $n$
5. for $j = 1$ to $n$
6. $D(k)_{ij} = \min\{D(k-1)_{ij}, D(k-1)_{ik} + D(k-1)_{kj}\}$
7. return ($D(n)$)
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

intermediate vertices on a path $v_i \leadsto v_j$: those other than $v_i$ and $v_j$. 

Define:

$$d(k)_{ij}$$

to be the shortest path distance from $v_i$ to $v_j$ with no intermediate vertices of indexes higher than $k$.

Thus

$$d(k)_{ij} = \min\{d(k-1)_{ij}, d(k-1)_{ik} + d(k-1)_{kj}\}$$

with base case:

$$d(0)_{ij} = w_{ij}.$$ 

Floyd-Warshall ($W$)

1. $n = \text{rows}[W]$
2. $D(0) = W$
3. for $k = 1$ to $n$
4. for $i = 1$ to $n$
5. for $j = 1$ to $n$
6. $D(k)_{ij} = \min\{D(k-1)_{ij}, D(k-1)_{ik} + D(k-1)_{kj}\}$
7. return $(D(n))$
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

intermediate vertices on a path \( v_i \leadsto v_j \): those other than \( v_i \) and \( v_j \).

Define: \( d_{ij}^{(k)} \) to be the shortest path distance from \( v_i \) to \( v_j \).
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

intermediate vertices on a path $v_i \rightsquigarrow v_j$: those other than $v_i$ and $v_j$.

Define: $d_{ij}^{(k)}$ to be the shortest path distance from $v_i$ to $v_j$ with no intermediate vertices of indexes higher than $k$. 
Floyd-Warshall algorithm

Intermediate vertices on a path $v_i \sim v_j$: those other than $v_i$ and $v_j$.

Define: $d_{i,j}^{(k)}$ to be the shortest path distance from $v_i$ to $v_j$ with no intermediate vertices of indexes higher than $k$. Thus

$$d_{i,j}^{(k)} = \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$$

with base case: $d_{i,j}^{(0)} = w_{i,j}$. 
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

intermediate vertices on a path $v_i \rightsquigarrow v_j$: those other than $v_i$ and $v_j$.

Define: $d_{i,j}^{(k)}$ to be the shortest path distance from $v_i$ to $v_j$ with no intermediate vertices of indexes higher than $k$. Thus

$$d_{i,j}^{(k)} = \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$$

with base case: $d_{i,j}^{(0)} = w_{i,j}$.

Floyd-Warshall($W$)
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

intermediate vertices on a path \( v_i \leadsto v_j \): those other than \( v_i \) and \( v_j \).

Define: \( d_{ij}^{(k)} \) to be the shortest path distance from \( v_i \) to \( v_j \) with no intermediate vertices of indexes higher than \( k \). Thus

\[
d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})
\]

with base case: \( d_{ij}^{(0)} = w_{ij} \).

\textsc{Floyd-Warshall}(W)

1. \( n = \text{rows}[W] \)
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

Intermediate vertices on a path \( v_i \sim v_j \): those other than \( v_i \) and \( v_j \).

Define: \( d_{ij}^{(k)} \) to be the shortest path distance from \( v_i \) to \( v_j \) with no intermediate vertices of indexes higher than \( k \). Thus

\[
d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})
\]

with base case: \( d_{ij}^{(0)} = w_{ij} \).

\text{Floyd-Warshall}(W)

1. \( n = \text{rows}[W] \)
2. \( D^{(0)} = W \)
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

Intermediate vertices on a path \( v_i \rightarrow v_j \): those other than \( v_i \) and \( v_j \).

Define: \( d_{ij}^{(k)} \) to be the shortest path distance from \( v_i \) to \( v_j \) with no intermediate vertices of indexes higher than \( k \). Thus

\[
d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})
\]

with base case: \( d_{ij}^{(0)} = w_{ij} \).

\text{Floyd-Warshall}(W)
1. \( n = \text{rows}[W] \)
2. \( D^{(0)} = W \)
3. \text{for } k = 1 \text{ to } n
Floyd-Warshall algorithm

intermediate vertices on a path \( v_i \leadsto v_j \): those other than \( v_i \) and \( v_j \).

Define: \( d^{(k)}_{ij} \) to be the shortest path distance from \( v_i \) to \( v_j \) with no intermediate vertices of indexes higher than \( k \). Thus

\[
d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})
\]

with base case: \( d^{(0)}_{ij} = w_{ij} \).

**Floyd-Warshall**\((W)\)

1. \( n = \text{rows}[W] \)
2. \( D^{(0)} = W \)
3. \( \textbf{for } k = 1 \textbf{ to } n \)
4. \( \textbf{for } i = 1 \textbf{ to } n \)
Floyd-Warshall algorithm

intermediate vertices on a path $v_i \leadsto v_j$: those other than $v_i$ and $v_j$.

Define: $d_{i,j}^{(k)}$ to be the shortest path distance from $v_i$ to $v_j$
with no intermediate vertices of indexes higher than $k$. Thus

$$d_{i,j}^{(k)} = \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$$

with base case: $d_{i,j}^{(0)} = w_{ij}$.

**Floyd-Warshall**($W$)
1. $n = rows[W]$
2. $D^{(0)} = W$
3. for $k = 1$ to $n$
4. for $i = 1$ to $n$
5. for $j = 1$ to $n$
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

intermediate vertices on a path \( v_i \leadsto v_j \): those other than \( v_i \) and \( v_j \).

Define: \( d^{(k)}_{ij} \) to be the shortest path distance from \( v_i \) to \( v_j \) with no intermediate vertices of indexes higher than \( k \). Thus

\[
d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})
\]

with base case: \( d^{(0)}_{ij} = w_{ij} \).

\text{Floyd-Warshall}(W)
1. \( n = \text{rows}[W] \)
2. \( D^{(0)} = W \)
3. \( \text{for } k = 1 \text{ to } n \)
4. \( \quad \text{for } i = 1 \text{ to } n \)
5. \( \quad \quad \text{for } j = 1 \text{ to } n \)
6. \( \quad \quad \quad D^{(k)}[i,j] = \min\{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\} \)
Chapter 25. All-pairs shortest paths

Floyd-Warshall algorithm

intermediate vertices on a path \( v_i \leadsto v_j \): those other than \( v_i \) and \( v_j \).

Define: \( d_{ij}^{(k)} \) to be the shortest path distance from \( v_i \) to \( v_j \) with no intermediate vertices of indexes higher than \( k \). Thus

\[
d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})
\]

with base case: \( d_{ij}^{(0)} = w_{ij} \).

FLOYD-WARSHALL(\( W \))
1. \( n = \text{rows}[W] \)
2. \( D^{(0)} = W \)
3. \( \text{for } k = 1 \text{ to } n \)
4. \( \quad \text{for } i = 1 \text{ to } n \)
5. \( \quad \quad \text{for } j = 1 \text{ to } n \)
6. \( \quad \quad \quad D^{(k)}[i, j] = \min\{D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j]\} \)
7. \( \text{return } (D^{(n)}) \)
Chapter 25. All-pairs shortest paths

\[
D^{(0)} = \begin{pmatrix}
0 & 3 & 8 & \infty & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & \infty & \infty \\
2 & \infty & -5 & 0 & \infty \\
\infty & \infty & \infty & 6 & 0
\end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix}
\text{NIL} & 1 & 1 & \text{NIL} & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\
\text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\
4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\
\text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL}
\end{pmatrix}
\]

\[
D^{(1)} = \begin{pmatrix}
0 & 3 & 8 & \infty & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & \infty & \infty \\
2 & 5 & -5 & 0 & -2 \\
\infty & \infty & \infty & 6 & 0
\end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix}
\text{NIL} & 1 & 1 & \text{NIL} & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\
\text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\
4 & 1 & 4 & \text{NIL} & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL}
\end{pmatrix}
\]

\[
D^{(2)} = \begin{pmatrix}
0 & 3 & 8 & 4 & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & 5 & 11 \\
2 & 5 & -5 & 0 & -2 \\
\infty & \infty & \infty & 6 & 0
\end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix}
\text{NIL} & 1 & 1 & 2 & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\
\text{NIL} & 3 & \text{NIL} & 2 & 2 \\
4 & 1 & 4 & \text{NIL} & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL}
\end{pmatrix}
\]

\[
D^{(3)} = \begin{pmatrix}
0 & 3 & 8 & 4 & -4 \\
\infty & 0 & \infty & 1 & 7 \\
\infty & 4 & 0 & 5 & 11 \\
2 & -1 & -5 & 0 & -2 \\
\infty & \infty & \infty & 6 & 0
\end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix}
\text{NIL} & 1 & 1 & 2 & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\
\text{NIL} & 3 & \text{NIL} & 2 & 2 \\
4 & 3 & 4 & \text{NIL} & 1 \\
\text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL}
\end{pmatrix}
\]

\[
D^{(4)} = \begin{pmatrix}
0 & 3 & 1 & 4 & -4 \\
2 & 0 & -4 & 1 & -1 \\
7 & 4 & 0 & 5 & 3 \\
2 & -1 & -5 & 0 & -2 \\
8 & 5 & 1 & 6 & 0
\end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix}
\text{NIL} & 1 & 4 & 2 & 1 \\
4 & \text{NIL} & 4 & 2 & 1 \\
4 & 3 & \text{NIL} & 2 & 1 \\
4 & 3 & 4 & \text{NIL} & 1 \\
4 & 3 & 4 & 5 & \text{NIL}
\end{pmatrix}
\]

\[
D^{(5)} = \begin{pmatrix}
0 & 1 & 3 & 2 & -4 \\
3 & 0 & -4 & 1 & -1 \\
7 & 4 & 0 & 5 & 3 \\
2 & -1 & -5 & 0 & -2 \\
8 & 5 & 1 & 6 & 0
\end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix}
\text{NIL} & 3 & 4 & 5 & 1 \\
4 & \text{NIL} & 4 & 2 & 1 \\
4 & 3 & \text{NIL} & 2 & 1 \\
4 & 3 & 4 & \text{NIL} & 1 \\
4 & 3 & 4 & 5 & \text{NIL}
\end{pmatrix}
\]
• Constructing a shortest path
Chapter 25. All-pairs shortest paths

• Constructing a shortest path

• for each $v_i$ and each $v_j$, to remember the last step to reach $j$. 

\[
\pi(0)_{ij} = \begin{cases} 
\text{NULL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\
\text{i} & \text{if } i \neq j \text{ and } w_{ij} < \infty.
\end{cases}
\]

\[
\pi(k)_{ij} = \begin{cases} 
\pi(k-1)_{ij} & \text{if } d(k-1)_{ij} \leq d(k-1)_{ik} + d(k-1)_{kj}, \\
\pi(k-1)_{kj} & \text{if } d(k-1)_{ij} > d(k-1)_{ik} + d(k-1)_{kj}.
\end{cases}
\]
Chapter 25. All-pairs shortest paths

- Constructing a shortest path
- for each $v_i$ and each $v_j$, to remember the last step to reach $j$. predecessor matrix $\pi$, recursively defined as

\[
\pi(0)_{ij} = \begin{cases} 
\text{NULL} & \text{if } i = j \text{ or } w_{ij} = \infty \\
i & \text{if } i \neq j \text{ and } w_{ij} < \infty 
\end{cases}
\]

\[
\pi(k)_{ij} = \pi(k-1)_{ij} \quad \text{if } d(k-1)_{ij} \leq d(k-1)_{ik} + d(k-1)_{kj},
\]

\[
\pi(k)_{ij} = \pi(k-1)_{kj} \quad \text{if } d(k-1)_{ij} > d(k-1)_{ik} + d(k-1)_{kj}.
\]
Chapter 25. All-pairs shortest paths

- Constructing a shortest path
- for each $v_i$ and each $v_j$, to remember the last step to reach $j$. predecessor matrix $\pi$, recursively defined as

$$
\pi_{ij}^{(0)} = NULL \text{ if } i = j \text{ or } w_{ij} = \infty, \text{ or }
$$
Chapter 25. All-pairs shortest paths

- Constructing a shortest path
- for each $v_i$ and each $v_j$, to remember the last step to reach $j$. predecessor matrix $\pi$, recursively defined as

\[
\pi^{(0)}_{ij} = \text{NULL} \text{ if } i = j \text{ or } w_{ij} = \infty, \text{ or }
\]

\[
\pi^{(0)}_{ij} = i \text{ if } i \neq j \text{ and } w_{ij} < \infty.
\]
Chapter 25. All-pairs shortest paths

- Constructing a shortest path
- for each $v_i$ and each $v_j$, to remember the last step to reach $j$.

predecessor matrix $\pi$, recursively defined as

\[
\pi^{(0)}_{ij} = \text{NULL} \text{ if } i = j \text{ or } w_{ij} = \infty, \text{ or } \\
\pi^{(0)}_{ij} = i \text{ if } i \neq j \text{ and } w_{ij} < \infty.
\]

\[
\pi^{(k)}_{ij} = \pi^{(k-1)}_{ij} \text{ if } d^{(k-1)}_{ij} \leq d^{(k-1)}_{ik} + d^{(k-1)}_{kj}, \text{ or }
\]

\[
\pi^{(k)}_{ij} = \pi^{(k-1)}_{kj} \text{ if } d^{(k-1)}_{ij} > d^{(k-1)}_{ik} + d^{(k-1)}_{kj}.
\]
Chapter 25. All-pairs shortest paths

- Constructing a shortest path
- for each \( v_i \) and each \( v_j \), to remember the last step to reach \( j \).

predecessor matrix \( \pi \), recursively defined as

\[
\pi^{(0)}_{ij} = \text{NULL} \quad \text{if} \quad i = j \quad \text{or} \quad w_{ij} = \infty, \quad \text{or}
\]

\[
\pi^{(0)}_{ij} = i \quad \text{if} \quad i \neq j \quad \text{and} \quad w_{ij} < \infty.
\]

\[
\pi^{(k)}_{ij} = \pi^{(k-1)}_{ij} \quad \text{if} \quad d^{(k-1)}_{ij} \leq d^{(k-1)}_{ik} + d^{(k-1)}_{kj}, \quad \text{or}
\]

\[
\pi^{(k)}_{ij} = \pi^{(k-1)}_{kj} \quad \text{if} \quad d^{(k-1)}_{ij} > d^{(k-1)}_{ik} + d^{(k-1)}_{kj}
\]
Chapter 25. All-pairs shortest paths

Summary of shortest path algorithms
Summary of shortest path algorithms

1. Bellman-Ford’s algorithm (able to detect negative weight cycles)
Chapter 25. All-pairs shortest paths

Summary of shortest path algorithms

1. Bellman-Ford’s algorithm (able to detect negative weight cycles)
2. DAG Shortest Paths (use topological sorting) [Lawler]
Chapter 25. All-pairs shortest paths

Summary of shortest path algorithms

1. Bellman-Ford’s algorithm (able to detect negative weight cycles)
2. DAG Shortest Paths (use topological sorting) [Lawler]
3. Dijkstra's algorithm (assuming non-negative weights)
Chapter 25. All-pairs shortest paths

Summary of shortest path algorithms

1. Bellman-Ford’s algorithm (able to detect negative weight cycles)
2. DAG Shortest Paths (use topological sorting) [Lawler]
3. Dijkstra’s algorithm (assuming non-negative weights)
4. Matrix multiplication (DP) [Lawler, folklore]
Chapter 25. All-pairs shortest paths

Summary of shortest path algorithms

1. Bellman-Ford’s algorithm (able to detect negative weight cycles)
2. DAG Shortest Paths (use topological sorting) [Lawler]
3. Dijkstra's algorithm (assuming non-negative weights)
4. Matrix multiplication (DP) [Lawler, folklore]
5. Floyd-Warshall algorithm (DP)