Graph Search Algorithms
A surprisingly large number of computational problems can be expressed as graph problems.
(a) A directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$. The edge $(2,2)$ is a self-loop.

(b) An undirected graph $G = (V,E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1,2), (1,5), (2,5), (3,6)\}$. The vertex 4 is isolated.

(c) The subgraph of the graph in part (a) induced by the vertex set $\{1,2,3,6\}$. 
Trees

A tree is a connected, acyclic, undirected graph.
A forest is a set of trees (not necessarily connected)
Running Time of Graph Algorithms

• Running time often a function of both $|V|$ and $|E|$.

• For convenience, drop the $|\cdot|$ in asymptotic notation, e.g. $O(V+E)$. 
End of Lecture 10

April 6, 2009
Representations: Undirected Graphs

Space complexity:

- Adjacency List: $\theta(V + E)$
- Adjacency Matrix: $\theta(V^2)$

Time to find all neighbours of vertex $u$:

- Adjacency List: $\theta(\text{degree}(u))$
- Adjacency Matrix: $\theta(V)$

Time to determine if $(u,v) \in E$:

- Adjacency List: $\theta(\text{degree}(u))$
- Adjacency Matrix: $\theta(1)$
Representations: Directed Graphs

Space complexity:
- Adjacency List: $\Theta(V + E)$
- Adjacency Matrix: $\Theta(V^2)$

Time to find all neighbours of vertex $u$:
- Adjacency List: $\Theta(\text{degree}(u))$
- Adjacency Matrix: $\Theta(V)$

Time to determine if $(u,v) \in E$:
- Adjacency List: $\Theta(\text{degree}(u))$
- Adjacency Matrix: $\Theta(1)$
Breadth-First Search

• **Goal:** To recover the shortest paths from a source node \( s \) to all other reachable nodes \( v \) in a graph.
  - The length of each path and the paths themselves are returned.

• **Notes:**
  - There are an exponential number of possible paths
  - This problem is harder for general graphs than trees because of cycles!
Breadth-First Search

Input: Graph $G = (V,E)$ (directed or undirected) and source vertex $s \in V$.

Output:
\[ d[v] = \text{shortest path distance} \, \delta(s,v) \, \text{from} \, s \, \text{to} \, v, \, \forall v \in V. \]
\[ \pi[v] = u \, \text{such that} \, (u,v) \, \text{is last edge on a shortest path from} \, s \, \text{to} \, v. \]

- **Idea:** send out search ‘wave’ from $s$.
- **Keep track of progress by colouring vertices:**
  - **Undiscovered** vertices are coloured **black**
  - **Just discovered** vertices (on the wavefront) are coloured **red**.
  - **Previously discovered** vertices (behind wavefront) are coloured **grey**.
BFS

First-In First-Out (FIFO) queue stores ‘just discovered’ vertices.

---

**Found**

**Not Handled**

**Queue**
BFS

Found
Not Handled
Queue

d=0
BFS

Found
Not Handled
Queue

d=0

d=1

d=2

d=1
g
b
c
f

d=2

d=2

d=2

d=2

a

s

d

b

e

g

e

f

h

i

j

k

l

m

n

CSE 3101
BFS

Found
Not Handled
Queue

d=0
d=1
d=2
d=1
d=2

d=0
d=1
d=2

d=1

d=2

d=0

d=1

d=2

d=0

d=1

d=2

d=0

d=1

d=2

CSE 3101
BFS

Not Handled
Queue

Found

Queue

d=0

d=1

d=2

d=3
BFS

Found
Not Handled
Queue

d=0

d=1

d=2

d=3

d=3
BFS

Found
Not Handled
Queue

d=0

d=1

d=2

d=3
BFS

Found
Not Handled
Queue

d=0
d=1
d=2
d=3
d=2
d=3
BFS

Queue

Found
Not Handled

d=0
d=1
d=2
d=3

h
i
l

CSE 3101
BFS

Not Handled
Queue

Found

s
a
c
b
d
d=0
d=1
d=4
d=3
d=2
d=3

d=4

l
j
g
f
c
h
k
l
m

CSE 3101
BFS

Not Handled
Queue

Found

d=0

d=1

d=2

d=3

d=4

d=5
Breadth-First Search Algorithm

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

- Q is a FIFO queue.
- Each vertex assigned finite \( d \) value at most once.
- Q contains vertices with \( d \) values \( \{i, \ldots, i, i+1, \ldots, i+1\} \)
- \( d \) values assigned are monotonically increasing over time.
Breadth-First-Search is **Greedy**

- Vertices are handled:
  - in order of their discovery (FIFO queue)
  - Smallest $d$ values first
Correctness

Basic Steps:

The shortest path to $u$ has length $d$ & there is an edge from $u$ to $v$.

There is a path to $v$ with length $d+1$. 
Correctness

- Vertices are discovered in order of their distance from the source vertex \( s \).

- When we discover \( v \), how do we know there is not a shorter path to \( v \)?
  - Because if there was, we would already have discovered it!
Correctness

Input: Graph $G = (V, E)$ (directed or undirected) and source vertex $s \in V$.

Output:
- $d[v] =$ distance from $s$ to $v$, $\forall v \in V$.
- $\pi[v] = u$ such that $(u,v)$ is last edge on shortest path from $s$ to $v$.

Two-step proof:

On exit:
1. $d[v] \geq \delta(s,v) \forall v \in V$
2. $d[v] \neq \delta(s,v) \forall v \in V$
Claim 1. $d$ is never too small: $d[v] \geq \delta(s,v) \forall v \in V$

Proof: There exists a path from $s$ to $v$ of length $d[v]$.

By Induction:
Suppose it is true for all vertices thus far discovered (red and grey). $v$ is discovered from some adjacent vertex $u$ being handled.

$$\rightarrow d[v] = d[u] + 1$$
$$\geq \delta(s,u) + 1$$
$$\geq \delta(s,v)$$

since each vertex $v$ is assigned a $d$ value exactly once, it follows that on exit, $d[v] \geq \delta(s,v) \forall v \in V$. 
Claim 1. \( d \) is never too small: \( d[v] \geq \delta(s,v) \forall v \in V \)

Proof: There exists a path from \( s \) to \( v \) of length \( d[v] \).

BFS \((G, s)\)

\[
\begin{align*}
&1 \quad \text{for each vertex } u \in V[G] - \{s\} \\
&2 \quad \text{do } color[u] \leftarrow \text{BLACK} \\
&3 \quad d[u] \leftarrow \infty \\
&4 \quad \pi[u] \leftarrow \text{NIL} \\
&5 \quad color[s] \leftarrow \text{RED} \\
&6 \quad d[s] \leftarrow 0 \\
&7 \quad \pi[s] \leftarrow \text{NIL} \\
&8 \quad Q \leftarrow \emptyset \\
&9 \quad \text{ENQUEUE}(Q, s) \\
&10 \quad \text{while } Q \neq \emptyset \quad \text{<LI>: } d[v] \geq \delta(s,v) \forall 'discovered' \text{ (red or grey) } v \in V \\
&11 \quad \text{do } u \leftarrow \text{DEQUEUE}(Q) \\
&12 \quad \text{for each } v \in \text{Adj}[u] \\
&13 \quad \text{do if } color[v] = \text{BLACK} \\
&14 \quad \text{then } color[v] \leftarrow \text{RED} \\
&15 \quad d[v] \leftarrow d[u] + 1 \geq \delta(s,u) + 1 \geq \delta(s,v) \\
&16 \quad \pi[v] \leftarrow u \\
&17 \quad \text{ENQUEUE}(Q, v) \\
&18 \quad color[u] \leftarrow \text{GRAY}
\end{align*}
\]
Claim 2. \( d \) is never too big: \( d[v] \leq \delta(s, v) \forall v \in V \)

Proof by contradiction:

Suppose one or more vertices receive a \( d \) value greater than \( \delta \).

Let \( v \) be the vertex with minimum \( \delta(s, v) \) that receives such a \( d \) value.

Suppose that \( v \) is discovered and assigned this \( d \) value when vertex \( x \) is dequeued.

Let \( u \) be \( v \)'s predecessor on a shortest path from \( s \) to \( v \).

Then
\[
\delta(s, v) < d[v] \\
\rightarrow \delta(s, v) - 1 < d[v] - 1 \\
\rightarrow d[u] < d[x]
\]

Recall: vertices are dequeued in increasing order of \( d \) value.

\( \rightarrow u \) was dequeued before \( x \).

\( \rightarrow d[v] = d[u] + 1 = \delta(s, v) \quad \text{Contradiction!} \)
Correctness

Claim 1. \( d \) is never too small: \( d[v] \geq \delta(s,v) \forall v \in V \)

Claim 2. \( d \) is never too big: \( d[v] \leq \delta(s,v) \forall v \in V \)

\[ \Rightarrow d \text{ is just right: } d[v] = \delta(s,v) \forall v \in V \]
Progress?

- On every iteration one vertex is processed (turns gray).

BFS(\(G, s\))

1. for each vertex \(u \in V[G] - \{s\}\)
2. \hspace{1em} do color\([u]\) \leftarrow BLACK
3. \hspace{1em} d\([u]\) \leftarrow \infty
4. \hspace{1em} \pi\([u]\) \leftarrow NIL
5. \hspace{1em} color\([s]\) \leftarrow RED
6. d\([s]\) \leftarrow 0
7. \pi\([s]\) \leftarrow NIL
8. Q \leftarrow \emptyset
9. ENQUEUE\((Q, s)\)
10. while Q \neq \emptyset
11. \hspace{1em} do u \leftarrow DEQUEUE\((Q)\)
12. \hspace{1em} for each v \in Adj\([u]\)
13. \hspace{2em} do if color\([v]\) = BLACK
14. \hspace{2em} then color\([v]\) \leftarrow RED
15. \hspace{2em} d\([v]\) \leftarrow d\([u]\) + 1
16. \hspace{2em} \pi\([v]\) \leftarrow u
17. \hspace{2em} ENQUEUE\((Q, v)\)
18. color\([u]\) \leftarrow GRAY
Running Time

Each vertex is enqueued at most once $\rightarrow O(V)$

Each entry in the adjacency lists is scanned at most once $\rightarrow O(E)$

Thus run time is $O(V + E)$.

```
BFS(G, s)
1   for each vertex $u \in V[G] - \{s\}$
2       do color[$u$] $\leftarrow$ BLACK
3           d[$u$] $\leftarrow$ $\infty$
4           $\pi$[$u$] $\leftarrow$ NIL
5       color[$s$] $\leftarrow$ RED
6       d[$s$] $\leftarrow$ 0
7       $\pi$[$s$] $\leftarrow$ NIL
8       Q $\leftarrow$ $\emptyset$
9       ENQUEUE(Q, s)
10      while Q $\neq$ $\emptyset$
11          do u $\leftarrow$ DEQUEUE(Q)
12              for each v $\in$ Adj[$u$]
13                  do if color[v] = BLACK
14                     then color[v] $\leftarrow$ RED
15                     d[v] $\leftarrow$ d[$u$] + 1
16                     $\pi$[v] $\leftarrow$ u
17                     ENQUEUE(Q, v)
18                 color[u] $\leftarrow$ GRAY
```
The shortest path problem has the **optimal substructure property**:
- Every subpath of a shortest path is a shortest path.

The **optimal substructure property**
- is a hallmark of both greedy and dynamic programming algorithms.
- allows us to compute both shortest path distance and the shortest paths themselves by storing only one $d$ value and one predecessor value per vertex.
Recovering the Shortest Path

For each node $v$, store predecessor of $v$ in $\pi(v)$.

Predecessor of $v$ is $\pi(v) = u$. 
Recovering the Shortest Path

PRINT-PATH(G, s, v)
Precondition: s and v are vertices of graph G
Postcondition: the vertices on the shortest path from s to v have been printed in order
if v = s then
    print s
else if π[v] = NIL then
    print "no path from" s "to" v "exists"
else
    PRINT-PATH(G, s, π[v])
    print v
Colours are actually not required

\begin{align*}
\text{BFS}(V, E, s) \\
\text{for each } u \in V - \{s\} & \quad \text{do } d[u] \leftarrow \infty \\
\quad d[s] & \leftarrow 0 \\
Q & \leftarrow \emptyset \\
\text{ENQUEUE}(Q, s) \\
\text{while } Q \neq \emptyset & \quad \text{do } u \leftarrow \text{DEQUEUE}(Q) \\
\quad \text{for each } v \in \text{Adj}[u] & \quad \text{do if } d[v] = \infty \\
\quad & \quad \text{then } d[v] \leftarrow d[u] + 1 \\
& \quad \text{ENQUEUE}(Q, v)
\end{align*}
Depth First Search (DFS)

• Idea:
  – Continue searching “deeper” into the graph, until we get stuck.
  – If all the edges leaving $v$ have been explored we “backtrack” to the vertex from which $v$ was discovered.

• Does not recover shortest paths, but can be useful for extracting other properties of graph, e.g.,
  – Topological sorts
  – Detection of cycles
  – Extraction of strongly connected components
Depth-First Search

Input: Graph $G = (V,E)$ (directed or undirected)

Output: 2 timestamps on each vertex:

- $d[v] =$ discovery time.
- $f[v] =$ finishing time.

$1 \leq d[v] < f[v] \leq 2 \mid V \mid$

- Explore every edge, starting from different vertices if necessary.
- As soon as vertex discovered, explore from it.
- Keep track of progress by colouring vertices:
  - Black: undiscovered vertices
  - Red: discovered, but not finished (still exploring from it)
  - Gray: finished (found everything reachable from it).
DFS

Found
Not Handled
Stack

<node,# edges>

Note: Stack is Last-In First-Out (LIFO)
DFS

Found
Not Handled
Stack

\[<\text{node},\#\text{ edges}>\]

\[s,0\]
DFS

Found
Not Handled
Stack

<node,# edges>

a,0
s,1
DFS

Found
Not Handled
Stack

\(<\text{node, \# edges}>\)

c,0
a,1
s,1

CSE 3101
DFS

Found
Not Handled
Stack

<node,# edges>

h,0
c,1
a,1
s,1

CSE 3101
DFS

Found
Not Handled
Stack

<node,# edges>

k,0
h,1
c,1
a,1
s,1
DFS

Found
Not Handled
Stack

⟨node,# edges⟩

Path on Stack

Tree Edge

CSE 3101
DFS

Found Not Handled

Stack

<node,# edges>

CSE 3101

56
DFS

Cross Edge to handled node: d[h]<d[i]

Found
Not Handled
Stack

<node,# edges>

i,1
c,2
a,1
s,1
DFS

Found
Not Handled
Stack

<node,# edges>

/s,1
a,1
c,2
i,2
b,1
e,1
g,2
j,1
m,1

CSE 3101

58
DFS

Not Handled

Stack

<node,# edges>

1,0
i,3
c,2
a,1
s,1
DFS

Found
Not Handled
Stack

\(<\text{node,}\#\text{ edges}>\)

- a, 1
- c, 2
- i, 3
- b
- d
- g
- e
- j
- m
- s
- l

Stack:

- 1, 1
- i, 3
- c, 2
- a, 1
- s, 1

Graph:

- s
- b
- e
- g
- j
- m
- l
- i
- k
- h
- c
- a
- f

Edges:

- 1/1
- 2/1
- 3/1
- 4/7
- 5/6
- 60
- 8/1
- 9/1

CSE 3101
DFS

Found
Not Handled
Stack

<node, # edges>

i, 3
j, 3
h, 2
c, 2
a, 1
s, 1

CSE 3101
DFS

Found
Not Handled
Stack

<node,# edges>

j,0
g,1
i,4
c,2
a,1
s,1
DFS

Back Edge to node on Stack:

Not Handled
Stack

<node,# edges>

j, 1
g, 1
i, 4
c, 2
a, 1
s, 1
DFS

Found
Not Handled
Stack

<node, # edges>

a, 1
c, 2
j, 2
g, 1
i, 4
c, 2
a, 1
s, 1

CSE 3101
DFS
Found
Not Handled
Stack

<node, # edges>

m, 1
j, 2
g, 1
i, 4
c, 2
a, 1
s, 1

a
b
c
d
e
f

g
h

1/2
3/8
5/6
9/10
11/12
13/
DFS

Found
Not Handled
Stack
<node,# edges>

j,2
g,1
i,4
c,2
a,1
s,1

CSE 3101
DFS

Stack

<node,# edges>

a,1
c,2
g,1
8/1/
9/10
5/6
4/7
2/
3/

1/11/12/15

Found Not Handled

g,1
i,4
c,2
a,1
s,1

CSE 3101
DFS

Found
Not Handled
Stack

<node,# edges>

a,1
b,1
s,1
DFS

Not Handled
Stack

\langle node, \# \text{edges}\rangle

\begin{align*}
&\text{f, 1} \\
&\text{i, 5} \\
&\text{c, 2} \\
&\text{a, 1} \\
&\text{s, 1}
\end{align*}
DFS

Not Handled
Stack

\(<node, \# \text{edges}>\)

Found

\(a, 1\)
\(c, 2\)
\(s, 1\)
\(l, 10\)
\(m, 14\)
\(j, 15\)
\(g, 16\)
\(d, 7\)
\(f, 18\)
\(11/16\)
\(5/6\)
\(4/7\)
\(8/\)
\(3/\)
\(2/\)
\(1/\)
DFS

Found
Not Handled
Stack

<node,# edges>

s, 1
a, 1
c, 2
b, /
e, /
g, 11/16
h, 4/7
f, 17/18
d, /

i, 8/19
j, 12/15
k, 5/6
l, 9/10
m, 13/14

CSE 3101

73
DFS

Found
Not Handled
Stack

<node,# edges>

c,3
a,1
s,1

Forward Edge
DFS

Found
Not Handled
Stack

\(<\text{node}, \# \text{ edges}>\)

a, 1
s, 1

CSE 3101
DFS

Found
Not Handled
Stack

<node,# edges>

a,2
s,1

CSE 3101
DFS

Found
Not Handled
Stack
<node,# edges>

s, l
DFS

Found
Not Handled
Stack

<node,# edges>

d,0
s,2

CSE 3101
DFS

Found
Not Handled
Stack

<node, # edges>

d, 1
s, 2
DFS

Not Handled

Stack

<node, # edges>

d, 2
s, 2
DFS

Found
Not Handled
Stack

<node,# edges>

e,0
d,3
s,2

CSE 3101
DFS

Found
Not Handled
Stack

<node,# edges>

e,1
d,3
s,2

CSE 3101
DFS

Found
Not Handled
Stack

<node,# edges>
DFS

Not Handled
Found
Stack

<node,# edges>

b,0
s,4
DFS

Found
Not Handled
Stack

<node,# edges>

b, l
s, 4
DFS

Found
Not Handled
Stack

<node,# edges>
DFS

Found
Not Handled
Stack

\langle \text{node}, \# \text{edges} \rangle

\begin{align*}
\text{Nodes:} & \quad a, b, c, d, e, f, g, h, i, j, k, l, m, s \\
\text{Edges:} & \quad (a, b), (b, e), (c, f), (d, i), (e, j), (f, g), (g, j), (h, k), (i, l), (j, k), (k, l), (l, m), (m, s)
\end{align*}
DFS

Found
Not Handled
Stack

<node,# edges>

Tree Edges
Back Edges
Forward Edges
Cross Edges
Classification of Edges in DFS

1. **Tree edges** are edges in the depth-first forest $G_\pi$. Edge $(u, v)$ is a tree edge if $v$ was first discovered by exploring edge $(u, v)$.

2. **Back edges** are those edges $(u, v)$ connecting a vertex $u$ to an ancestor $v$ in a depth-first tree.

3. **Forward edges** are non-tree edges $(u, v)$ connecting a vertex $u$ to a descendant $v$ in a depth-first tree.

4. **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other.
Classification of Edges in DFS

1. **Tree edges**: Edge \((u, v)\) is a **tree edge** if \(v\) was **black** when \((u, v)\) traversed.

2. **Back edges**: \((u, v)\) is a **back edge** if \(v\) was **red** when \((u, v)\) traversed.

3. **Forward edges**: \((u, v)\) is a **forward edge** if \(v\) was **gray** when \((u, v)\) traversed and \(d[v] > d[u]\).

4. **Cross edges** \((u,v)\) is a **cross edge** if \(v\) was **gray** when \((u, v)\) traversed and \(d[v] < d[u]\).

Classifying edges can help to identify properties of the graph, e.g., a graph is acyclic iff DFS yields no **back** edges.
In a depth-first search of an undirected graph, every edge is either a tree edge or a back edge. Why?
Undirected Graphs

- **Suppose that** \((u,v)\) **is a forward edge** or a cross edge in a DFS of an undirected graph.

- \((u,v)\) **is a forward edge** or a cross edge when \(v\) is already handled (grey) when accessed from \(u\).

- This means that all vertices reachable from \(v\) have been explored.

- Since we are currently handling \(u\), \(u\) must be red.

- Clearly \(v\) is reachable from \(u\).

- Since the graph is undirected, \(u\) must also be reachable from \(v\).

- Thus \(u\) must already have been handled: \(u\) must be grey.

- **Contradiction!**
End of Lecture 11

Apr 8, 2009
Depth-First Search Algorithm

**DFS(G)**

1. for each vertex \( u \in V[G] \)
2. \( \text{do } color[u] \leftarrow \text{BLACK} \)
3. \( \pi[u] \leftarrow \text{NIL} \)
4. \( \text{time } \leftarrow 0 \)
5. for each vertex \( u \in V[G] \)
6. \( \text{do if } color[u] = \text{BLACK} \)
7. \( \text{then DFS-Visit}(u) \)

**DFS-Visit (u)**

Precondition: vertex \( u \) is undiscovered

Postcondition: all vertices reachable from \( u \) have been processed

1. \( color[u] \leftarrow \text{RED} \)  
   \( \triangleright \) BLACK vertex \( u \) has just been discovered.
2. \( \text{time } \leftarrow \text{time} + 1 \)
3. \( d[u] \leftarrow \text{time} \)
4. for each \( v \in Adj[u] \)  
   \( \triangleright \) Explore edge \((u, v)\).
5. \( \text{do if } color[v] = \text{BLACK} \)
6. \( \text{then } \pi[v] \leftarrow u \)
7. \( \text{then DFS-Visit}(v) \)
8. \( color[u] \leftarrow \text{GRAY} \)  
   \( \triangleright \) GRAY \( u \); it is finished.
9. \( f[u] \leftarrow \text{time} \leftarrow \text{time} + 1 \)
Depth-First Search Algorithm

DFS(G)

1. for each vertex \( u \in V[G] \)
2. do color[\( u \)] \( \leftarrow \) BLACK
3. \( \pi[u] \leftarrow \text{NIL} \)
4. \( \text{time} \leftarrow 0 \)
5. for each vertex \( u \in V[G] \)
6. do if color[\( u \)] = BLACK
7. then DFS-VISIT(\( u \))

DFS-Visit(\( u \))

Precondition: vertex \( u \) is undiscovered
Postcondition: all vertices reachable from \( u \) have been processed

1. color[\( u \)] \( \leftarrow \) RED \( \triangleright \) BLACK vertex \( u \) has just been discovered.
2. \( \text{time} \leftarrow \text{time} + 1 \)
3. \( d[u] \leftarrow \text{time} \)
4. for each \( v \in Adj[u] \) \( \triangleright \) Explore edge \((u, v)\).
5. do if color[\( v \)] = BLACK
6. then \( \pi[v] \leftarrow u \)
7. DFS-VISIT(\( v \))
8. color[\( u \)] \( \leftarrow \) GRAY \( \triangleright \) GRAY \( u \); it is finished.
9. \( f[u] \leftarrow \text{time} \leftarrow \text{time} + 1 \)

\text{total work} = \( \theta(V) \)

Thus running time = \( \theta(V + E) \)

\text{total work} = \sum_{v \in V} |Adj[\( v \)]| = \( \theta(E) \)
Topological Sorting
(e.g., putting tasks in linear order)

An application of Depth-First Search
Linear Order

underwear
pants

socks
shoes

underwear
pants
socks
shoes

socks
underwear
pants
shoes
Linear Order

underwear → pants → shoes → socks

Too many video games?
Linear Order

Precondition:
A Directed Acyclic Graph (DAG)

Post Condition:
Find one valid linear order

Algorithm:
• Find a terminal node (sink).
• Put it last in sequence.
• Delete from graph & repeat $\Theta(V^2)$

We can do better!

..... l
Linear Order

Alg: DFS

Found
Not Handled
Stack

a
b
c
d
e
g
f
h
i
j
k
l
When node is popped off stack, insert at front of linearly-ordered “to do” list.

**Linear Order:**

```
..... f
```
Linear Order

Alg: DFS

Found
Not Handled
Stack: ged

Linear Order:

l,f
Linear Order

Alg: DFS

Found
Not Handled
Stack

g, l, f
Linear Order

Alg: DFS

Found Not Handled Stack

d

Linear Order:
e, g, l, f
Linear Order

Alg: DFS

Found
Not Handled
Stack

d, e, g, l, f
Linear Order

Alg: DFS

Found
Not Handled
Stack

k
j
i

Linear Order:
d, e, g, l, f
Linear Order
Alg: DFS

Found
Not Handled
Stack

j
i

Linear Order:
k, d, e, g, l, f
Linear Order

Alg: DFS

Not Handled Stack

Found

Linear Order:

j,k,d,e,g,l,f
Linear Order

Alg: DFS

Linear Order:

i,j,k,d,e,g,l,f
Linear Order
Alg: DFS

Stack
Found
Not Handled

Linear Order:
i,j,k,d,e,g,l,f
Linear Order

Alg: DFS

c, i, j, k, d, e, g, l, f
Linear Order

Alg: DFS

Linear Order:

b,c,i,j,k,d,e,g,l,f
Linear Order

Alg: DFS

Linear Order:

b, c, i, j, k, d, e, g, l, f
Linear Order

Alg: DFS

Linear Order:

h, b, c, i, j, k, d, e, g, l, f
Linear Order

**Alg:** DFS

Linear Order:

\[ a, h, b, c, i, j, k, d, e, g, l, f \]  

**Done!**
Linear Order

Proof: Consider each edge

• Case 1: u goes on stack first before v.
  • Because of edge,
    v goes on before u comes off
  • v comes off before u comes off
  • v goes after u in order. 😊
Proof: Consider each edge
• Case 1: u goes on stack first before v.
• Case 2: v goes on stack first before u.
  v comes off before u goes on.
  • v goes after u in order. 😊
Linear Order

Proof: Consider each edge

• Case 1: \( u \) goes on stack first before \( v \).
• Case 2: \( v \) goes on stack first before \( u \).
  \( v \) comes off before \( u \) goes on.
Case 3: \( v \) goes on stack first before \( u \).
  \( u \) goes on before \( v \) comes off.
• Panic: \( u \) goes after \( v \) in order.
• Cycle means linear order is impossible

The nodes in the stack form a path starting at \( s \).

- \( u \rightarrow v \)

- \( v \ldots u \ldots \)
Linear Order

Alg: DFS

Found
Not Handled
Stack

Analysis: $\Theta(V+E)$

Linear Order:

Done!
Shortest Paths Revisited
Back to Shortest Path

• BFS finds the **shortest paths** from a source node $s$ to every vertex $v$ in the graph.

• Here, the **length** of a path is simply the number of edges on the path.

• But what if edges have different ‘costs’?

\[ \delta(s, v) = 3 \]

\[ \delta(s, v) = 12 \]
Single-Source (Weighted) Shortest Paths
The Problem

• What is the shortest driving route from Toronto to Ottawa? (e.g. MAPQuest, Google Maps)

• Input:
  Directed Graph \( G = (V,E) \)
  Edge weights \( w : E \to \mathbb{R} \)

  Weight of path \( p = \langle v_0, v_1, \ldots, v_k \rangle = \sum_{i=1}^{k} w(v_{i-1}, v_i) \)

  Shortest-path weight from \( u \) to \( v \):

  \[
  \delta(u,v) = \begin{cases} 
  \min \{ w(p) : u \to^p \to v \} & \text{if } \exists \text{ a path } u \to \cdots \to v, \\
  \infty & \text{otherwise.}
  \end{cases}
  \]

  Shortest path from \( u \) to \( v \) is any path \( p \) such that \( w(p) = \delta(u,v) \).
Example

Single-source shortest path search induces a search tree rooted at $s$. This tree, and hence the paths themselves, are not necessarily unique.
Shortest path variants

- **Single-source shortest-paths problem**: the shortest path from s to each vertex v. (e.g. BFS)

- **Single-destination shortest-paths problem**: Find a shortest path to a given destination vertex t from each vertex v.

- **Single-pair shortest-path problem**: Find a shortest path from u to v for given vertices u and v.

- **All-pairs shortest-paths problem**: Find a shortest path from u to v for every pair of vertices u and v.
**Negative-weight edges**

- OK, as long as no negative-weight cycles are reachable from the source.
  - If we have a negative-weight cycle, we can just keep going around it, and get $w(s, v) = -\infty$ for all $v$ on the cycle.
  - But OK if the negative-weight cycle is not reachable from the source.
  - Some algorithms work only if there are no negative-weight edges in the graph.
Optimal substructure

- Lemma: Any subpath of a shortest path is a shortest path
- Proof: Cut and paste.

Suppose this path $p$ is a shortest path from $u$ to $v$. Then $\delta(u,v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{vy})$.

Now suppose there exists a shorter path $x \rightarrow \cdots \rightarrow y$.

Then $w(p'_{xy}) < w(p_{xy})$.

Construct $p'$:

Then $w(p') = w(p_{ux}) + w(p'_{xy}) + w(p_{vy}) < w(p_{ux}) + w(p_{xy}) + w(p_{vy}) = w(p)$.

So $p$ wasn't a shortest path after all!
Cycles

• Shortest paths can’t contain cycles:
  – Already ruled out negative-weight cycles.
  – Positive-weight: we can get a shorter path by omitting the cycle.
  – Zero-weight: no reason to use them \rightarrow assume that our solutions won’t use them.
Output of a single-source shortest-path algorithm

- For each vertex $v$ in $V$:
  - $d[v] = \delta(s, v)$.
    - Initially, $d[v] = \infty$.
    - Reduce as algorithm progresses. But always maintain $d[v] \geq \delta(s, v)$.
    - Call $d[v]$ a shortest-path estimate.
  - $\pi[v] = \text{predecessor of } v \text{ on a shortest path from } s$.
    - If no predecessor, $\pi[v] = \text{NIL}$.
    - $\pi$ induces a tree — **shortest-path tree**.
Initialization

• All shortest-paths algorithms start with the same initialization:

\[\text{INIT-SINGLE-SOURCE}(V, s)\]

for each \( v \) in \( V \)

\[
\begin{align*}
\text{do } d[v] & \leftarrow \infty \\
\pi[v] & \leftarrow \text{NIL} \\
\end{align*}
\]

\[d[s] \leftarrow 0\]
Relaxing an edge

- Can we improve shortest-path estimate for v by going through u and taking (u,v)?

\[
\text{RELAX}(u, v, w) \\
\quad \text{if } d[v] > d[u] + w(u, v) \text{ then} \\
\quad \quad d[v] \leftarrow d[u] + w(u, v) \\
\quad \pi[v] \leftarrow u
\]
General single-source shortest-path strategy

1. Start by calling INIT-SINGLE-SOURCE
2. Relax Edges

Algorithms differ in the order in which edges are taken and how many times each edge is relaxed.
Example: Single-source shortest paths in a directed acyclic graph (DAG)

- Since graph is a DAG, we are guaranteed no negative-weight cycles.
Algorithm

\textsc{Dag-Shortest-Paths} (G, w, s)
1. topologically sort the vertices of G
2. \textsc{Initialize-Single-Source} (G, s)
3. \textbf{for} each vertex \(u\), taken in topologically sorted order
4. \hspace{1em} \textbf{do for} each vertex \(v \in \text{Adj}[u]\)
5. \hspace{2em} \textbf{do} \textsc{Relax} (u, v, w)

Time: \(\Theta(V + E)\)
Example
Example
Example
Example
Example
Example
Correctness: Path relaxation property (Lemma 24.15)

Let \( p = \langle v_0, v_1, \ldots, v_k \rangle \) be a shortest path from \( s = v_0 \) to \( v_k \). If we relax, in order, \( (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \), even intermixed with other relaxations, then \( d[v_k] = \delta(s, v_k) \).
Correctness of DAG Shortest Path Algorithm

• Because we process vertices in topologically sorted order, edges of any path are relaxed in order of appearance in the path.
  – → Edges on any shortest path are relaxed in order.
  – → By path-relaxation property, correct.
Example: Dijkstra’s algorithm

- Applies to general weighted directed graph (may contain cycles).
- But weights must be non-negative.
- Essentially a weighted version of BFS.
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weights \(d[v]\).
- Maintain 2 sets of vertices:
  - \(S\) = vertices whose final shortest-path weights are determined.
  - \(Q\) = priority queue = \(V-S\).
Dijkstra’s algorithm

\[
\text{Dijkstra}\left(G, w, s\right)
\]

1. \text{Initialize-Single-Source}\left(G, s\right)
2. \( S \leftarrow \emptyset \)
3. \( Q \leftarrow V[G] \)
4. \textbf{while} \( Q \neq \emptyset \)
5. \hspace{1em} \textbf{do} \( u \leftarrow \text{Extract-Min}(Q) \)
6. \hspace{1em} \( S \leftarrow S \cup \{u\} \)
7. \hspace{1em} \textbf{for each vertex} \( v \in Adj[u] \)
8. \hspace{1em} \hspace{1em} \textbf{do} \text{Relax}\left(u, v, w\right)

- Dijkstra’s algorithm can be viewed as greedy, since it always chooses the “lightest” vertex in \( V - S \) to add to \( S \).
Dijkstra’s algorithm: Analysis

- Analysis:
  - Using minheap, queue operations takes $O(\log V)$ time

\[ \text{Dijkstra}(G, w, s) \]
1. Initialize-Single-Source($G, s$) $O(V)$
2. $S \leftarrow \emptyset$
3. $Q \leftarrow V[G]$
4. while $Q \neq \emptyset$
5. \hspace{1em} do $u \leftarrow \text{Extract-Min}(Q)$ $O(\log V) \times O(V)$ iterations
6. \hspace{1em} do $S \leftarrow S \cup \{u\}$
7. \hspace{1em} for each vertex $v \in \text{Adj}[u]$ $O(\log V) \times O(E)$ iterations
8. \hspace{1em} do Relax($u, v, w$)

→ Running Time is $O(E \log V)$
Example

Key:
White ⇔ Not Found
Grey ⇔ Handling
Black ⇔ Handled
Example
Example
Example
Example
Example
Correctness of Dijkstra’s algorithm

\begin{verbatim}
Dijkstra(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 S ← ∅
3 Q ← V[G]
4 while Q ≠ ∅
5 do u ← EXTRACT-MIN(Q)
6 S ← S ∪ {u}
7 for each vertex v ∈ Adj[u]
8 do RELAX(u, v, w)
\end{verbatim}

- **Loop invariant:** d[v] = δ(s, v) for all v in S.
  - **Initialization:** Initially, S is empty, so trivially true.
  - **Termination:** At end, Q is empty → S = V → d[v] = δ(s, v) for all v in V.
  - **Maintenance:**
    - Need to show that
      - d[u] = δ(s, u) when u is added to S in each iteration.
      - d[u] does not change once u is added to S.
Correctness of Dijkstra’s Algorithm: Upper Bound Property

• Upper Bound Property:
  
  1. \( d[v] \geq \delta(s,v) \forall v \in V \)
  2. Once \( d[v] = \delta(s,v) \), it doesn't change

• Proof:

  By induction.

  **Base Case:** \( d[v] \geq \delta(s,v) \forall v \in V \) immediately after initialization, since

  \( d[s] = 0 = \delta(s,s) \)

  \( d[v] = \infty \forall v \neq s \)

  **Inductive Step:**

  Suppose \( d[x] \geq \delta(s,x) \forall x \in V \)

  Suppose we relax edge \((u,v)\).

  If \( d[v] \) changes, then \( d[v] = d[u] + w(u,v) \)

  \[
  \geq \delta(s,u) + w(u,v) \\
  \geq \delta(s,v)
  \]
Correctness of Dijkstra’s Algorithm

Claim: When \( u \) is added to \( S \), \( d[u] = \delta(s,u) \)

Proof by Contradiction: Let \( u \) be the first vertex added to \( S \) such that \( d[u] \neq \delta(s,u) \) when \( u \) is added.

Let \( y \) be first vertex in \( V - S \) on shortest path to \( u \)
Let \( x \) be the predecessor of \( y \) on the shortest path to \( u \)

Claim: \( d[y] = \delta(s,y) \) when \( u \) is added to \( S \).

Proof:
\( d[x] = \delta(s,x) \), since \( x \in S \).

\((x,y)\) was relaxed when \( x \) was added to \( S \) \( \rightarrow d[y] = \delta(s,x) + w(x,y) = \delta(s,y) \)
Correctness of Dijkstra’s Algorithm

Thus $d[y] = \delta(s, y)$ when $u$ is added to $S$.

$\rightarrow d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$ (upper bound property)

But $d[u] \leq d[y]$ when $u$ added to $S$

Thus $d[y] = \delta(s, y) = \delta(s, u) = d[u]!$

Thus when $u$ is added to $S$, $d[u] = \delta(s, u)$

Consequences:
There is a shortest path to $u$ such that the predecessor of $u$ $\pi[u] \in S$ when $u$ is added to $S$. The path through $y$ can only be a shortest path if $w[p_2] = 0$. 

\[ \text{DIJKSTRA}(G, w, s) \]
1. \text{INITIALIZE-SINGLE-SOURCE}(G, s)
2. $S \leftarrow \emptyset$
3. $Q \leftarrow V[G]$
4. \textbf{while }$Q \neq \emptyset$
5. \hspace{1em} \textbf{do }$u \leftarrow \text{EXTRACT-MIN}(Q)$
6. \hspace{2em} $S \leftarrow S \cup \{u\}$
7. \hspace{2em} \textbf{for each vertex } $v \in \text{Adj}[u]$
8. \hspace{3em} \textbf{do }\text{RELAX}(u, v, w)
Correctness of Dijkstra’s algorithm

\textbf{Dijkstra}(G, w, s)
1 \textbf{Initialize-Single-Source}(G, s)
2 \text{S} \leftarrow \emptyset
3 \text{Q} \leftarrow V[G]
4 \textbf{while} \ Q \neq \emptyset
5 \quad \textbf{do} \ u \leftarrow \textbf{Extract-Min}(Q)
6 \quad \text{S} \leftarrow \text{S} \cup \{u\}
7 \quad \textbf{for each vertex} \ v \in \text{Adj}[u]
8 \quad \hspace{1cm} \textbf{do} \ \textbf{Relax}(u, v, w)

\textbf{Relax}(u, v, w) \text{ can only decrease } d[v].

\text{By the upper bound property, } d[v] \geq \delta(s,v).

Thus once \( d[v] = \delta(s,v) \), it will not be changed.

\begin{itemize}
\item \textbf{Loop invariant:} \( d[v] = \delta(s,v) \) for all \( v \) in \( S \).
\item \textbf{Maintenance:}
\begin{itemize}
\item Need to show that
\begin{itemize}
\item \( d[u] = \delta(s,u) \) when \( u \) is added to \( S \) in each iteration.
\item \( d[u] \) does not change once \( u \) is added to \( S \).
\end{itemize}
\end{itemize}
\end{itemize}