Graphs

\[ G = (V, E) \]

- \( V \) is a set of vertices (often called nodes)
- \( E \) is a set of edges between vertices

\[ |V| = \text{number of vertices} \]
\[ |E| = \text{number of edges} \]

**Undirected graphs**

Each edge specifies a connection between two vertices without any predecessor (source) / successor (sink) relationship.

- \( V = \{1, 2, 3, 4, 5\} \)
- \( E = \{ (1, 2), (1, 4), (2, 4), (2, 5), (2, 3), (3, 5), (4, 5) \} \)

\[ |V| = 5 \]
\[ |E| = 7 \]

**Directed graphs**

Each edge specifies a directed connection from one vertex (source) to another vertex (sink).

- \( V = \{1, 2, 3, 4, 5, 6\} \)
- \( E = \{ (1, 2), (1, 4), (2, 5), (3, 5), (3, 6), (4, 2), (5, 4), (6, 6) \} \)

\[ |V| = 6 \]
\[ |E| = 8 \]
Graph Representation

adjacency matrix (better for dense graphs)

\[
\begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 1 & 0 \\
5 & 1 & 0 & 1 & 0 & 0 \\
\end{bmatrix}
\]

| v | x | \( |v| \) matrix
\[a_{ij} = \begin{cases} 
1 & \text{if } (i,j) \in E \\
0 & \text{otherwise}
\end{cases}\]

adjacency list representation (preferred)

list (array) of vertices where each vertex contains a list (array) of vertices to which the vertex connects (i.e. edge)

\[
\begin{array}{c}
1 & 2 & 3 & 4 & 5 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
2 & 1 & 2 & 2 & 4 \\
5 & 5 & 4 & 5 & 1 \\
\end{array}
\]

\{ edges \}

\[
\begin{array}{c}
3 \\
\end{array}
\]
adjacency matrix

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 0 & 1 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

adjacency list representation

\[
1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \\
2 \rightarrow 5 \\
3 \rightarrow 6 \\
4 \rightarrow 5 \\
\{5\} \\
\]

Implementation note:

can be implemented with array of pointers
Graph Algorithms

Breadth-first search
given a graph $G = (V, E)$ and a source vertex $s$
compute distance to all vertices $u \in V[G] - \{s\}$
computes distance (smallest number of edges) from $s$ to each reachable vertex
disCOVERS every reachable vertex from $s$
can produce a breadth-first tree with root $s$
that contains reachable vertices

\[\text{BFS}(G, s)\]

1. for each vertex $u \in V[G] - \{s\}$
   2. do color $[u] \leftarrow \text{WHITE}$
   3. $d[u] \leftarrow \infty$
   4. $\pi[u] \leftarrow \text{NIL}$
   5. color $[s] \leftarrow \text{GRAY}$
   6. $d[s] \leftarrow 0$
   7. $\pi[s] \leftarrow \text{NIL}$
   8. $Q \leftarrow \emptyset$
   9. \text{ENQUEUE} ($Q$, $S$)
10. while $Q \neq \emptyset$
    11. do $u \leftarrow \text{DEQUEUE}(Q)$
       12. for each $v \in \text{Adj}[u]$
           13. do if color $[v] \leftarrow \text{WHITE}$
               14. then color $[v] \leftarrow \text{GRAY}$
               15. $d[v] \leftarrow d[u] + 1$
               16. $\pi[v] \leftarrow u$
               17. \text{ENQUEUE} ($Q$, $v$)
    18. color $[u] \leftarrow \text{BLACK}$
called BFS because it discovers vertices uniformly across breadth of the frontier (discover vertices at k before discovering vertices at k+1)

to track progress, color vertices

all vertices start at WHITE
when vertices discovered becomes non-white (GRAY or BLACK)

BLACK vertices - all adjacent have been discovered
GRAY vertices - some adjacent haven't been discovered (WHITE)
and represent edge of frontier between discovered and undiscovered vertices

color\[u\] data structure tracks color of each vertex \( u \in V \)

\( \pi [u] \) data structure stores predecessor of \( u \)
if \( u \) has no predecessor, stores NIL

d\[u\] data structure stores distance from source \( S \) to vertex \( u \)

\( Q \) first-in, first out queue to manage the set of gray vertices.

(see Fig 22.3)
Depth-first search searches deeper into graph before backtracking.

edges explored out of the most recently discovered vertex \( v \) that still has unexplored edges leaving it.

computes discovery time \( d[u, j] \) and finishing time \( f[u, j] \) for each vertex.

creates depth-first forest of depth-first trees when its discovered (GRAYED) when its done (BLACKENED).

**DFS(G)** // notice no source vertex

1. for each vertex \( u \in V[G] \)
2. do color\([u, j] \leftarrow \text{WHITE} \) // initialize all vertices to white
3. \( \pi[u, j] \leftarrow \text{NIL} \) // initialize color, pred
4. time \( \leftarrow 0 \) // reset global time counter
5. for each vertex \( u \in V[G] \)
6. do if color\([u, j] = \text{WHITE} \)
7. then DFS-VISIT\((u)\) // while found call DFS-VISIT // becoming root of new tree

**DFS-VISIT\((u)\)**

1. color\([u, j] \leftarrow \text{GRAY} \) // paint vertex gray
2. time \( \leftarrow \text{time} + 1 \) // update global time
3. \( d[u, j] \leftarrow \text{time} \) // set discovery time
4. for each \( v \in \text{Adj}[u, j] \)
5. do if color\([v, j] = \text{WHITE} \)
6. then \( \pi[v, j] \leftarrow u \)
7. DFS-VISIT\((v)\) // examine neighbors, if white visit it
8. color\([u, j] \leftarrow \text{BLACK} \) // after every edge leaving \( u \) visited, paint black
9. \( f[u, j] \leftarrow \text{time} \leftarrow \text{time} + 1 \) // \( f \) is finish time to highlight // recursion

(see fig 22.4)
Shortest Path

many variants exist: single-destination shortest path

single-pair shortest path

all-pairs shortest path

we consider single source path \( p = <v_0, v_1, \ldots, v_k> \)

**single source shortest path**

- given a graph \( G = (V, E) \) compute the shortest path from a source vertex \( s \in V \) to each vertex \( v \in V \)

- assumes a weighted edge such that from each edge \( (u, v) \) a weight \( w(u, v) \) is specified

\[
\text{weight of path } w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \text{ weighted path is the sum of weights of its constituent edges}
\]

```plaintext
INITIALIZE-SINGLE-SOURCE (G, s)

for each vertex \( v \in V[G] \)

do \( d[v] \leftarrow \infty \)

\( \pi[v] \leftarrow \text{NIL} \)

\( d[s] \leftarrow 0 \)

//upper bound weight of shortest path
```

```
RELAX \((u, v, w)\)

if \( d[v] > d[u] + w(u, v) \)

then \( d[v] \leftarrow d[u] + w(u, v) \)

\( \pi[v] \leftarrow u \)

//relaxing edge tests if we can improve shortest path so far
```

\[
\begin{array}{c}
\text{relaxation of edge } (u, v) \text{ with weight 2}
\end{array}
\]

\[
\begin{array}{c}
\text{RELAX}(5, 9, 2) \\
\text{RELAX}(5, 7, 2)
\end{array}
\]

\[
\begin{array}{c}
\text{RELAX}(5, 6, 2)
\end{array}
\]
Dijkstra's algorithm solves single-source shortest-path problem on weighted directed graph for the case when all edge weights are non-negative.

Alg maintains a set of $S$ vertices whose final shortest-path weights from the source $s$ have already been determined.

$Q$ is a min-priority queue of vertices, keyed by their $d$ values.

**DIJKSTRA**($G,w,s$)

1. INITIALIZE-SINGLE-SOURCE($G,s$)  

2. $S \leftarrow \emptyset$  

3. $Q \leftarrow V[G]$  

4. While $Q \neq \emptyset$

   5. $u \leftarrow$ EXTRACT-MIN($Q$)  

   6. $S \leftarrow S \cup \{u\}$

   7. For each vertex $v \in \text{Adj}[u]$

      8. $d$ do RELAX($u,v,w$)

(see fig 24.6)
Directed Acyclic Graph (DAG)
directed graph that contains no cycles

often used to represent procedural relationships
(inputs and outputs of connected components such as
logic circuits)

\[
\begin{align*}
\text{DAG } \checkmark \\
\text{DAG } \times \\
\text{v, c, d is a cycle.}
\end{align*}
\]

topological sort
computes linear ordering of all vertices in DAG
such that if G contains an edge (u, v) then
u appears before v in ordering

**TOPOLOGICAL-SORT(G)**
1. call DFS(G) to compute finishing times F[v] for each vertex v
2. as each vertex is finished, insert into front of linked list
3. return the linked list of vertices

**can also use stack:**

**TOPOLOGICAL-SORT(G)**
1. DFS(G, stack)
2. while stack ≠ φ
   1. u = stack.pop
   2. output u

**modified DFS-VISIT:**

**DFS-VISIT(u, stack)**
1. color[u] ← GRAY
2. time ← time + 1
3. d[u] ← time
4. for each v ∈ Adj[u]
   1. if color[v] = WHITE
      1. then π[v] ← u
      2. DFS-VISIT(v, stack)
6. color[u] ← BLACK
7. f[u] ← time
8. stack.push(u)
9. time ← time + 1

(see fig 22.7)
DAG gives us opportunity to take advantage of topological sort

later edge will not jump back to earlier node
- no need for extract
- ordering done, just call relax

DAG-SHORTEST-PATHS (G, w, s)
1   topologically sort vertices of G
2   INITIALIZE-SINGLE-SOURCE (G, s)
3   for each vertex u, taken in topological order
4       do for each vertex v E Adj [u]
5           do RELAX (u, v, w)

(see fig 245)