Graph Algorithms - Outline of Topics

- **Elementary Graph Algorithms** - Chapter 22
  - graph representation
  - breadth-first-search, depth-first-search, topological sort

- **Minimum Spanning Trees** - Chapter 23
  - Kruskal's and Prim's algorithms (greedy algorithms)

- **Single-Source Shortest Paths** - Chapter 24
  - Dijkstra's algorithm (greedy algorithm)

Undirected Graphs

An **undirected graph** $G = (V, E)$ consists of
- A set $V = |V|$ of nodes (vertices), and
- A set $E = |E|$ of undirected edges represented by node pairs

In this graph, both (b,c) and (c,b) are edges.
Weighted graph: each edge has a number, called a weight attached to it. The weight is usually a positive number and may represent distance, cost, etc.

Sparse graph: $|E| = o(|V|^2)$.

Dense graph: $|E| = \Theta(|V|^2)$.

Graph Terminology

A directed graph (digraph) $G = (V, E)$ consists of

- A set $V$ of nodes (vertices), and
- A set $E$ of unidirectional edges (represented by arrows)
- Self-loops are possible (as shown on node f)

Note: In this graph, (b,c) is an edge, but (c,b) is not an edge.
**Adjacency List Graph Representation**

*Adjacency list:* An array $A[1, |V|]$ of lists, one for each node $v \in V$ (vertex set). Each node $v$'s list contains pointers to all nodes adjacent to $v$ in $G$. Each edge repeated twice.

```
        a --> b --> d
         \   /\   /
          b \ /  \
          /   \   
         /     \
        c --> e --> f
         ↓      ↓
        d       f
```

*Complexity issues*
- advantage - storage is $O(V + E)$ (good for sparse graphs)
- drawback - list traversal to find edge

**Representing Undirected Graphs with Adjacency Matrices**

*Adjacency matrix:* An array $A[V, V]$ such that

$$A[i,j] = 1 \text{ if } (i,j) \in E \text{ and } 0 \text{ otherwise}$$

```
        a b c d e f g
        a 0 1 0 1 0 0 0
        b 1 0 1 0 0 0 0
        c 1 0 0 1 1 0 0
        d 0 0 0 1 0 1 0
        e 0 1 1 0 0 1 0
        f 0 0 0 0 0 0 1
        g 0 0 0 1 1 1 0
```

*Complexity issues*
- advantage - $O(1)$ time to check for edge
- drawback - storage is $O(V^2)$ (practical only for dense graphs)

In undirected graph, only the entries above the upper left to lower right diagonal need to be stored.
Representing Digraphs with Adjacency Lists

Complexity issues
· advantage - storage is $O(V + E)$
  Good for sparse graphs, and most graphs we will use are sparse
· drawback - list traversal to find edge
Only store outgoing edges in adj. lists

If $(u,v)$ is an edge, then it is **incident** on both $u$ and $v$ and we say vertex $v$ is **adjacent** to vertex $u$. The same terms hold for undirected graphs. Adjacent vertices are called **neighbors**

The **degree** of a node in an undirected graph is the number of edges incident on it.

The **in-degree** of a node in a digraph is the number of edges entering it and its **out-degree** is the number of edges leaving it.
A path of length $k$ from a node $u$ to a node $u'$ is a sequence $(v_0, v_1, ..., v_k)$ of nodes such that $u = v_0$, $u' = v_k$ and there is an edge between each $v_i$, $i = 0, 1, 2, ..., k$. In a digraph, a path exists between nodes $a$ and $b$ only if there is a sequence of outgoing edges from $a$ to $b$.

If there is a path $p$ between vertices $u$ and $v$, we say $v$ is reachable from $u$ via $p$.

A simple path has all distinct vertices.

The red edges in this graph trace a simple path between each pair of nodes.

An undirected graph is connected if there is a simple path between every pair of nodes. A graph may have several connected components that are disjoint subsets of nodes.

A completely connected graph is an undirected graph in which every pair of nodes is adjacent.
In a digraph, a path \((v_0, v_1, ..., v_k)\) forms a **cycle** if \(v_0 = v_k\) and the path contains at least one edge.

The cycle is **simple** if, in addition, \(v_1, v_2, ..., v_k\) are distinct.

A digraph with no cycles is called a **directed acyclic graph**, abbreviated DAG.

**Breadth-First Search**

**Breadth-First Search** finds the shortest-path distance (number of edges) between a source node and every other node in \(G\).

Called breadth-first because it discovers all vertices at distance \(k\) from a *source node* \(s\) before it discovers any vertices at distance \(k+1\) from \(s\), spanning the breadth before the depth of \(G\).

BFS finds all vertices \(v\) that are reachable from \(s\) by building a breadth-first tree, where the path in the tree from \(s\) to \(v\) has the fewest number of edges of all paths from \(s\) to \(v\).
Breadth-First Search

**Breadth-First Search** has time complexity of \( O(V + E) \) and is often used as a building block of other algorithms.

BFS is particularly useful in finding shortest paths on unweighted graphs.

BFS starts at a node \( s \) in a graph and explores all its neighbor nodes before moving to the next level (neighbors of neighbors).

Explores nodes in "layers".

Maintains a queue of nodes to keep track of which node it should visit next.

Breadth-First Search Implementation

The algorithm from our book maintains a FIFO queue, \( Q \), to manage the set of nodes and starts by enqueuing \( s \), the source node

BFS algorithm maintains the following information for each vertex \( u \):

- **u.c**: white, gray, or black to indicate status
  - white = not discovered yet; initially, all nodes except \( s \) are undiscovered.
  - gray = discovered, but not finished; initially only \( s \).
  - black = finished; initially none are finished.

- **u.d**: distance from \( s \) to \( u \); initially \( \infty \) for all but \( s.d = 0 \)

- **u.\pi**: predecessor of \( u \) in BF tree; initially NIL for all\((s.\pi = NIL and remains NIL)\)
BFS node

Each node has fields for predecessor ($\pi$), distance from source, and color. Each node also has an associated adjacency list with pointers to neighboring nodes.

Breadth-First Search

BFS (G, s):
0. s.c = gray; s.\pi = NIL
1. Q.enqueue (s) // Q is a FIFO ds
2. while Q ≠ ∅
3. u = Q.dequeue()
4. for each v adjacent to u
5. if v.c == white
6. v.c = gray
7. v.d = u.d + 1
8. v.\pi = u
9. Q.enqueue(v)
10. u.c = black

Q.enqueue(s) adds s to the rear of Q
Q.dequeue() removes and returns the item at the head of Q

Note: If G is not connected, then BFS will not visit the entire graph (without some extra provisions in the algorithm)
**Breadth-First Search**

BFS (G, s):

0. s.c = gray; s.\( \pi \) = NIL
1. Q.enqueue(s) // Q is a FIFO ds
2. while Q \( \neq \) \( \emptyset \)
3. \( u = Q.dequeue() \)
4. for each v adjacent to u
5. \( \text{if } v.c == \text{white} \)
6. \( v.c = \text{gray} \)
7. \( v.d = u.d + 1 \)
8. \( v.\pi = u \)
9. Q.enqueue(v)
10. u.c = black

**Complexity**

(Adjacency List)

- each node enqueued and dequeued once = \( O(V) \) time
- each edge considered once (in each direction on undirected G) = \( O(E) \) time

\( \bullet \) total = \( O(V + E) \)
\( \quad = O(V^2) \) (w-c)

---

**Analysis of Breadth-First Search**

Shortest-path distance \( \delta(s,v) \): minimum number of edges in any path from vertex s to v. If no path exists from s to v, then \( \delta(s,v) = \infty \).

The ultimate goal of the proof of correctness is to show that \( v.d = \delta(s,v) \) when the algorithm is done and that a path is found from s to all reachable vertices.

L. 22.1: Children of a node u are given a higher d value than u.
L. 22.2: For every edge \( (u,v) \), the shortest path from s to v can be no longer than the (shortest path from s to u) + 1.
L. 22.3: At any time, Q holds at most 2 distinct d values. I.e., the range of values in Q is at most 2. Why?
C. 22.4: The d values are monotonically increasing over time as the algorithm runs.
**Theorem 22.5: (Correctness of BFS)**

Let \( G = (V, E) \) be a directed or undirected graph, and suppose that BFS is run from a given source vertex \( s \in V \). Then, during execution, BFS discovers every vertex \( v \neq s \) that is reachable from the source \( s \), and upon termination, \( v.d = \delta(s,v) \) for every reachable or unreachable vertex \( v \).

Proof by contradiction.

Assume that for some vertex \( v \) that \( v.d \neq \delta(s,v) \) after running BFS. Also, assume that \( v \) is the vertex with minimum \( \delta(s,v) \) that receives an incorrect \( d \) value. By Lemma 22.2, it must be that \( v.d > \delta(s,v) \).

**Case 1:** \( v \) is not reachable from \( s \). This is a contradiction to the assumption that \( v \) is reachable, and the Thm holds.

**Case 2:** \( v \) is reachable from \( s \). Let \( u \) be the vertex immediately preceding \( v \) on a shortest path from \( s \) to \( v \), so that \( \delta(s,v) = \delta(s,u) + 1 \). Because \( \delta(s,u) < \delta(s,v) \) and because \( v \) is the vertex with the minimum \( \delta(s,v) \) that receives an incorrect \( d \) value, \( u.d = \delta(s,u) \).

So we have \( v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1 \).

Consider the time \( t \) when \( u \) is dequeued. At time \( t \), \( v \) is either white, gray, or black. We can derive a contradiction in each of these cases.

**Case 1:** \( v \) is white. Then in line 12, \( v.d = u.d + 1 \).

**Case 2:** \( v \) is black. Then \( v \) was already dequeued, and therefore \( v.d \leq u.d \) (by L. 22.3).

**Case 3:** \( v \) is gray. Then \( v \) turned gray when it was visited from some vertex \( w \), which was dequeued before \( u \). Then \( v.d = w.d + 1 \). Since \( w.d \leq u.d \) (by L. 22.3), \( v.d \leq u.d + 1 \).

Each of these cases is a contradiction to \( v.d > \delta(s,v) \), so we conclude that \( v.d = \delta(s,v) \).
Breadth-First Trees

BFS builds a breadth-first tree that can be identified by using the $\pi$ values at each node.

The edges defined by each $v.\pi$ are called tree edges.

```plaintext
Print-Path (G, s, v) // finds the tree edges between s and v,
1. if v == s       // starting at v
2.   print s
3. else
4.   if v.\pi == NIL
5.     print "no path from " s " to " v " exists"
6.   else
7.     Print-Path(G, s, v.\pi)
8.     print v
```

Breadth-First Search v2

BFS (G, s):

0. let marked be a boolean array of size $|V|$ // init all false
1. let edgeTo be an array of $|V|$ integers
2. Q.enqueue (s)
3. marked[s] = true
4. while Q ≠ ∅
5.   u = Q.dequeue()
6.   for each v adjacent to u
7.     if marked[v] == false
8.       edgeTo[v] = u
9.     marked[v] = true
10. Q.enqueue(v)
Enumerating shortest path, $s \rightarrow v$

pathTo(v):
1. if (!marked[v]) return false
2. Stack<Integer> path = new Stack<Integer>()
3. for (int x = v; x != s; x = edgeTo[x])
4.     path.push(x)
5. path.push(s)
6. return path

When pathTo finishes, the path will contain the path from s to v and they can be popped off the stack in order.

Analysis of Breadth-First Search

Proposition A1: For any vertex v reachable from s, BFS computes a shortest path from s to v such that no path from s to v has fewer edges.

Informal proof:
It is easy to prove by induction that Q always consists of zero or more vertices of distance k from s, followed by zero or more vertices of distance k+1 from s, for some integer k, starting with k = 0. This property implies, in particular, that vertices enter and leave Q in order of their increasing distance from s. When a vertex v enters Q, no shorter path to v will be found before v comes off Q, and no path to v that is discovered after v comes off Q can be shorter than the path length s to v.
Analysis of Breadth-First Search

Proposition A2: BFS takes time proportional to $V + E$ in the worst case.

Informal proof:
BFS marks all the vertices connected to $s$ in time proportional to the degree of $s$. If the graph is connected, this sum equals the sum of the degrees of all the vertices, or $2E$.

Initializing the marked[] and edgeTo[] arrays takes time proportional to $V$.

Example BFS Traversal

Order of visiting: $a_1 \ b_6 \ c_2 \ d_3 \ e_4 \ f_5 \ g_7 \ h_8 \ i_9 \ j_{10}$
Distance of vertex: $0 \ 1 \ 1 \ 1 \ 2 \ 2 \ \infty \ \infty \ \infty \ \infty$
Breadth-first Search Forest

Tree edges are solid lines and dashed lines are cross edges.

Bipartite Graphs

A graph is bipartite if all its vertices can be partitioned into two disjoint subsets $X$ and $Y$ so that every edge connects a vertex in $X$ with a vertex in $Y$, i.e., if its vertices can be colored in 2 colors so that every edge has its end points colored in different colors.
Bipartite Graphs

Explain how BFS could be used to detect a bipartite graph.

Mark the source, A, with color 1, mark the nodes at level 1 with color 2, and so on. Every node on an even numbered level will be color 1 and on every odd level color 2.

Is this graph bipartite?

Bipartite Graphs
Applications of BFS

Based upon the BFS, there are $O(V + E)$-time algorithms for the following problems:

- Testing whether graph is connected.
- Computing a spanning forest of graph.
- Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists.
- Computing a cycle in graph or reporting that no such cycle exists.