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)

Graphs

An undirected graph \( G = (V, E) \) consists of

- A set \( V \) of nodes (vertices), and
- A set \( E \) of bidirectional (undirected) edges (linked pairs of nodes)

For analysis of graph algorithms, we will use \( |V| \) (number of nodes) and \( |E| \) (number of edges)

- \( V \) is often called \( n \) and \( E \) is usually called \( m \)

In this graph, both \((b,c)\) and \((c,b)\) are edges.

Representing Undirected Graphs with Adjacency Lists

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

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

example:

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

Representing Undirected Graphs with Adjacency Matrices

Adjacency matrix: An array \( A[V, V] \) such that

\[ A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases} \]

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

In undirected graph, only the entries above the upper left to lower right diagonal need to be stored.

Representing Digraphs with Adjacency Lists

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 \( f \))

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

Note: In this graph, \((b,c)\) is an edge, but \((c,b)\) is not an edge.

Digraphs

Complexity issues
- advantage - storage is \( O(V + E) \) (good for sparse graphs)
- drawback - list traversal to find edge
  only store outgoing edges in lists
Representing a Digraph with an Adjacency Matrix

For a digraph, add a 1 to the matrix position row \( i \) and column \( j \) only where an edge is outgoing from \( i \) to \( j \).

The **degree** of a vertex in a graph is the number of edges incident on it, for both undirected and directed graphs. The **in-degree** of a vertex in a digraph is the number of edges entering it and its **out-degree** is the number of edges leaving it.

An undirected graph is **connected** if there is a simple path between every pair of vertices. A **completely connected** graph is an undirected graph in which every pair of vertices is adjacent.

A path of length \( k \) from a vertex \( u \) to a vertex \( u' \) is a sequence \((v_0, v_1, \ldots, v_k)\) of vertices such that \( u = v_0, u' = v_k \) and there is an edge between each \( v_i, i = 0,1,2,\ldots,k \). In a digraph, a path exists between vertices \( a \) 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 simple paths between each pair of nodes.

**A sparse graph** is one in which \(|E| \leq o(|V|^2)\)

**A dense graph** is one in which \(|E| = \Theta(|V|^2)\)

If \((u,v)\) is an edge in a graph, 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.
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 Problem**

**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.

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 \(v\).

Our book gives an object-oriented algorithm. These slides present a non-object oriented solution.

---

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

---

**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
  - total = \(O(V + E)\) = \(O(V^2)\) (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?

L. 22.4: the $d$ values are monotonically increasing over time as the algorithm runs.

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 $v.d = u.d$ (by L. 22.3).

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

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

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 $v \in V$ 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 $v.d = \delta(s,v)$ 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, $u.d = \delta(s,u)$.

Analysis of 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 $Q \neq \emptyset$
 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.

Breadth-First Search

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 ($v$.$π$ == NIL)
5. print "no path from " $s$ " to " $v$ " exists"
6. else
7. Print-Path($G$, $s$, $v$.$π$)
8. print $v$

Breadth-First Trees

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

The edges defined by each $π$ are called tree edges.
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.

Example BFS Traversal

Order of visiting:  a  c  d  e  f  b  g  h  i  j
Distance of vertex:  0  1  1  1  2  2  ∞  ∞  ∞  ∞

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 color1, mark the nodes at level 1 with color2, and so on. Every node on an even numbered level will be color1 and on every odd level color2.
Bipartite Graphs

Is this graph bipartite?

A — B
C — D

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.