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)
- **Dynamic Programming Revisited** - Chapter 25
  - All Pairs Shortest Paths (Floyd-Warshall Alg.)

Graphs

An **undirected graph** $G = (V, E)$ consists of
- A set $V$ of nodes (vertices), and
- A set $E$ of bidirectional edges (linked pairs of nodes)
  - $|V|$ is often labeled $n$ and $|E|$ is labeled $m$

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

**Digraphs**

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.

Adjacent vertices are called **neighbors**

If $(u,v)$ is an edge in a graph, then it is *incident* on both $u$ and $v$ and vertex $v$ is *adjacent* to vertex $u$.

The **degree** of a vertex in a graph is the number of edges incident on it.

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.

A path of length $k$ from a vertex $u$ to a vertex $u'$ is a sequence $(v_0, v_1, ..., v_k)$ of vertices such that $u=v_0$, $u'=v_k$ and there is an edge between each $v_i, i = 1,2,...,k$. 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.
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 **dense graph** is one in which \(|E|\) is \(\Theta(|V|^2)\)

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

The least number of edges possible in a connected graph is \(n-1\). In this case, the graph is called a **tree**.

In a digraph, a path \((v_0, v_1, \ldots, v_k)\) forms a **cycle** if \(v_0 = v_k\) and the path contains at least one edge.

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

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

Representing Undirected Graphs with Adjacency Lists

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

**example:**

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

**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 \(A[1\ldots|V|, 1\ldots|V|]\) such that \(A[i,j] = 1\) if \((i,j) \in E\) and 0 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  0 1 0 0 1 0 0
  d  1 0 0 0 1 0 1
  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 edge
- **drawback:** - storage is \(O(|V|^2)\) (practical for dense graphs)

In an undirected graph, only the entries on and above the diagonal need to be stored.
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.

...Next: Algorithms that search graphs

Breadth-First Search

Breadth-First Search is an algorithm for searching a graph in a "wave" form and is the basis for these graph algorithms:
- Prim’s minimum spanning tree algorithm
- Dijkstra’s single-source shortest path algorithm

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.

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

The algorithm maintains a queue (Q) to manage the set of vertices and starts with s, the source vertex
Initially, all vertices except s are colored white, and s is gray.

BFS algorithm maintains the following information for each vertex u:
- u.color (white, gray, or black) : indicates status
  white = not discovered yet
  gray = discovered, but not finished
  black = finished
- u.d : distance from s to u; initially ∞ for all but s=0
- u.π : predecessor of u in BF tree

Breadth-First Search

BFS (G, s)
1. Enqueue (Q,s)
2. while Q ≠ ∅
3.   u = Dequeue(Q)
4.   for each v adjacent to u
5.     if v.color == white
6.        v.color = gray
7.        v.d = u.d + 1
8.        v.π = u
9.        Enqueue(Q,v)
10.   u.color = black

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

Enqueue(Q,s) adds s to the rear of Q
Dequeue(Q) removes and returns the item at the head of Q

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 v.π are called tree edges.

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
**Depth-First Search**

Depth-First Search is another algorithm for searching a graph. Called depth-first because it searches "deeper" in the graph whenever possible.

Edges are explored out of the most recently discovered vertex v that still has unexplored edges. When all of v’s edges have been explored, the search "backtracks" to explore the edges incident on the vertex from which v was discovered.

First investigated in the 19th century by French mathematician Charles Pierre Trémaux as a strategy for solving mazes. Later it became a search technique in AI.

DFS finds a spanning tree of a graph G. A spanning tree of a graph contains all nodes and some or all of the edges.

**Algorithm**

- **DFS (G)**
  1. for each v ∈ G
  2. if v.color == white
  3. DFS-Visit (G,v)

After execution, for every vertex u, u.d < u.f

**Note:** If G = (V, E) is not connected, then DFS will still visit the entire graph.

**Complexity (Adjacency List Representation)**

- check all edges adjacent to each node - O(|E|) time
- total = O(|V| + |E|)

---

**Depth-First Search (recursive version)**

**DFS (G)**

1. for each v ∈ G
2. if v.color == white
3. DFS-Visit (G,v)

**DFS-Visit (G,u)**

1. u.color = gray
2. u.d = time
3. time = time + 1
4. for each v adjacent to u
5. if v.color == white
6. v..π = u
7. DFS-Visit(G,v)
8. u.color = black
9. u.f = time
10. time = time + 1

Complexity (Adjacency List Representation)

- check all edges adjacent to each node - O(|E|) time
- total = O(|V| + |E|)

---

**Non-recursive version of DFS using a stack, S.**

Let N_i be set of neighbors of node i.

In this algorithm, vertices may be pushed on the stack multiple times. The color of each vertex when it is popped from the stack determines the code executed in the while loop.

**DFS-Visit (G,u)**

1. S.push(u)
2. while not S.isempty()
3. u = S.top
4. if u.color == WHITE
5. u.color = GRAY
6. u.d = time
7. time = time + 1
8. for all v in N_i
   if v.color == white
      S.push(v)
      v..π = u
   else if u.color == GRAY
      u = S.pop()
9. u = S.pop()
DFS Tree

DFS builds a depth-first tree whose edges can be identified by using the π values at each node.

The DFS algorithm defines a depth-first forest $G_s$.

Classification of DFS Tree Edges

1. Tree edges: Edges included in depth-first forest. Edge $(u,v)$ is tree edge if $v$ was first discovered by edge $(u,v)$.
2. Back edges: Edge $(u,v)$ connects a vertex $u$ to an ancestor (non-parent) $v$ in a depth-first tree.
3. Forward edges: Edge $(u,v)$ connects a vertex $u$ to a descendant (non-child) $v$ in a depth-first tree.
4. Cross edges: All other edges, i.e., between sibling nodes (e.g., nodes on different branches) of the same depth-first tree or between nodes in different depth-first trees.

If the original graph is undirected then all of its edges are tree edges or back edges. Cross and forward edges apply only to directed graphs.

Topological Sort - Application of DFS

Produces a linear ordering of all reachable nodes in a DAG.

**input:** directed acyclic graph (DAG)
**output:** ordering of nodes s.t. if $(u,v) \in E$, then $u$ comes before $v$ in ordering

Topological-Sort $(G)$
1. call DFS$(G)$ to compute finishing times $f[v]$ for each $v$
2. as each vertex is finished, insert it at head of a linked list
3. return the linked list of vertices

Complexity (Adjacency List Representation) - $O(|V| + |E|)$

Topologically sorted vertices appear in reverse order of their finishing times. An application of this type of sorting algorithm is to indicate precedence among ordered events represented in a DAG.

Topological Sort - Application of DFS

The canonical application of topological sorting (topological order) is in scheduling a sequence of jobs or tasks.

If the graph has any cycles, a topological ordering is impossible.