Graph Algorithms

Textbook reading

Chapter 3
Overview

**Design principle:**
- Learn the structure of a graph by systematic exploration

**Proof techniques:**
- Proof by contradiction

**Problems:**
- Bipartiteness
- Connectivity
- Strong connectivity
- Topological sorting
A **graph** is an ordered pair \( G = (V, E) \).

- \( V \) is the set of **vertices** of \( G \).
- \( E \) is the set of **edges** of \( G \).
- The elements of \( E \) are pairs \((v, w)\) of vertices.
A **graph** is an ordered pair $G = (V, E)$.

- $V$ is the set of **vertices** of $G$.
- $E$ is the set of **edges** of $G$.
- The elements of $E$ are pairs $(v, w)$ of vertices.

For an edge $(v, w) \in E$, we call vertices $v$ and $w$ **adjacent**.

Edge $(v, w)$ is **incident** to $v$ and $w$.

Vertices $v$ and $w$ are the **endpoints** of edge $(v, w)$.

The **degree** of a vertex $v$ is the number of edges incident to $v$. 
A graph is **undirected** if its edges are *unordered* pairs \((v, w)\), that is, \((v, w) = (w, v)\).

A graph is **directed** if its edges are *ordered* pairs \((v, w)\), that is, \((v, w) \neq (w, v)\).

Edge \((v, w)\) is an **out-edge** of \(v\) and an **in-edge** of \(w\).

The **in-degree** and **out-degree** of a vertex \(v\) are the numbers of in-edges and out-edges incident to \(v\).
A path $P = (x = v_0, v_1, \ldots, v_k = y)$ from a vertex $x$ to a vertex $y$ is a sequence of vertices such that $(v_{i-1}, v_i)$ is an edge, for all $1 \leq i \leq k$.

A cycle is a path from a vertex $x$ back to itself.

A path or cycle is simple if it contains every vertex of $G$ at most once.
A graph is **connected** if there exists a path from $x$ to $y$, for any two vertices $x$ and $y$ of $G$. 

connected

not connected
Adjacency-List Representation of Graphs

- Doubly-linked vertex list
- Doubly-linked edge list
- One doubly-linked adjacency list per vertex
- Pointers from adjacency list entries to vertices
- Cross-pointers between edges and adjacency list entries
Many problems are quite naturally expressed as graph problems:

**Example:** Stable matching is a special case of bipartite perfect matching.

A graph is **bipartite** if its vertices can be divided into sets $X$ and $Y$ so that every edge has one endpoint in $X$ and the other in $Y$.

A **matching** of a graph is a subset of edges so that no two edges in the set share an endpoint.

A matching is **perfect** if every vertex is the endpoint of an edge.
Example: Airline scheduling

- There are $n$ lucrative flight segments to be serviced
- Flight segment = (source, destination, departure time, arrival time)

Question: Can we service all $n$ segments using the $k$ planes in our fleet?

Rules:
- Same plane can service two segments $(s_1, t_1, d_1, a_1)$ and $(s_2, t_2, d_2, a_2)$ if
  - $t_1 = s_2$
  - There is enough time for maintenance between arrival time $a_1$ and departure time $d_2$
- We can add other flight segments to get a plane that arrived at destination $t_1$ to service segment from destination $s_2$ by adding extra flight segments that get us from $t_1$ to $s_2$. The same rules about maintenance periods between arrivals and departures apply.
Graph-theoretic formulation:
Graph-theoretic formulation:
Graph-theoretic formulation:

Are there $k$ paths in this network whose union includes all solid edges?
Graph-theoretic formulation:

Are there $k$ paths in this network whose union includes all solid edges?
Graph-theoretic formulation:

Are there $k$ paths in this network whose union includes all solid edges?

This reduces to a flow problem. See textbook, Chapter 7.
Example: Ordering tasks under constraints

Building a shack

1. Buy boards
2. Buy nails
3. Buy hammer
4. Erect sides of shack
5. Add roof
6. Buy hinges
7. Assemble door
8. Insert door in door frame
Example: Ordering tasks under constraints

Building a shack

1. Buy boards
2. Buy nails
3. Buy hammer
4. Erect sides of shack
5. Add roof
6. Buy hinges
7. Assemble door
8. Insert door in door frame

Topological sorting:
Number the vertices so that, for every edge \((v, w)\), \(v < w\).
Modelling Real-World Problems (4)

More examples:
- Communication networks
- Transportation networks
- Data structures
- ...
**Graph Exploration (1)**

**EXPLORE-GRAPH**($G$)

1. Mark every vertex and every edge of $G$ as unexplored
2. for every vertex $v$ of $G$
3. do if $v$ is unexplored
4. then Explore-from-Vertex($G, v$)

**EXPLORE-FROM-VERTEX**($G, v$)

1. Mark $v$ as explored
2. $S ← \text{Adj}(v)$
3. while $S$ is not empty
4. do Remove an edge $(x, y)$ from $S$
5. if $(x, y)$ is not explored
6. then if $y$ is not explored
7. then Mark $(x, y)$ as a tree edge
8. Mark $y$ as explored
9. $S ← S \cup \text{Adj}(y)$
10. else Mark $(x, y)$ as a non-tree edge
Graph Exploration Variants (1): Depth-First Search

**DFS-FROM-VERTEX**(\(G, v\))

1. Mark \(v\) as explored
2. \(\triangleright S\) is a stack
3. for every edge \((v, w)\) incident to \(v\)
   4. do Push\((S, (v, w))\)
5. while \(S\) is not empty
   6. do \((x, y) \leftarrow\) Pop\((S)\)
   7. if \((x, y)\) is not explored
      8. then if \(y\) is not explored
         9. then Mark \((x, y)\) as a tree edge
      10. Mark \(y\) as explored
     11. for every edge \((y, z)\) incident to \(y\)
     12. do Push\((S, (y, z))\)
     13. else Mark \((x, y)\) as a non-tree edge
Graph Exploration Variants (1): Depth-First Search

**DFS-FROM-VERTEX**\((G, v)\)
1. Mark \(v\) as explored
2. \(\triangleright S\) is a stack
3. for every edge \((v, w)\) incident to \(v\)
   4. do Push\((S, (v, w))\)
5. while \(S\) is not empty
   6. do \((x, y) \leftarrow \text{Pop}(S)\)
   7. if \((x, y)\) is not explored
      8. then if \(y\) is not explored
         9. then Mark \((x, y)\) as a tree edge
      10. Mark \(y\) as explored
      11. for every edge \((y, z)\) incident to \(y\)
         12. do Push\((S, (y, z))\)
      13. else Mark \((x, y)\) as a non-tree edge

**Lemma:** Depth-first search takes \(O(n + m)\) time.
**Graph Exploration Variants (2): Breadth-First Search**

**BFS-from-Vertex**(\(G, v\))

1. Mark \(v\) as explored
2. \(\triangleright \) \(S\) is a queue
3. for every edge \((v, w)\) incident to \(v\)
   4. do Enqueue\((S, (v, w))\)
5. while \(S\) is not empty
   6. do \((x, y) \leftarrow \text{Dequeue}(S)\)
   7. if \((x, y)\) is not explored
      8. then if \(y\) is not explored
         9. then Mark \((x, y)\) as a tree edge
      10. Mark \(y\) as explored
   11. for every edge \((y, z)\) incident to \(y\)
      12. do Enqueue\((S, (y, z))\)
   13. else Mark \((x, y)\) as a non-tree edge
**BFS-FROM-VERTEX**(*G*, *v*)

1. Mark *v* as explored
2. ▷ *S* is a queue
3. for every edge (*v*, *w*) incident to *v* 
4. do Enqueue(*S*, (*v*, *w*))
5. while *S* is not empty 
6. do (*x*, *y*) ← Dequeue(*S*) 
7. if (*x*, *y*) is not explored 
8. then if *y* is not explored 
9. then Mark (*x*, *y*) as a tree edge 
10. Mark *y* as explored 
11. for every edge (*y*, *z*) incident to *y* 
12. do Enqueue(*S*, (*y*, *z*)) 
13. else Mark (*x*, *y*) as a non-tree edge

**Lemma:** *Breadth-first search takes* \( \mathcal{O}(n + m) \) *time.*
**Dijkstra-from-Vertex** \((G, v)\)

1. Mark \(v\) as explored
2. ▶ \(S\) is a priority queue
3. for every edge \((v, w)\) incident to \(v\)
   - do Insert \((S, (v, w), w(v, w))\)
4. while \(S\) is not empty
5. do \((x, y) \leftarrow\) Delete-Min \((S)\)
6. if \((x, y)\) is not explored
   - then if \(y\) is not explored
     - then Mark \((x, y)\) as a tree edge
     - Mark \(y\) as explored
   - Let dist\((v, y)\) be the priority of edge \((x, y)\)
   - for every edge \((y, z)\) incident to \(y\)
     - do Insert \((S, (y, z), \text{dist}(s, y) + w(y, z))\)
7. else Mark \((x, y)\) as a non-tree edge
Graph Exploration Variants (4): Prim’s Algorithm

**Prim-from-Vertex** \( (G, v) \)

1. Mark \( v \) as explored
2. \textit{▷ S is a priority queue}
3. for every edge \((v, w)\) incident to \( v \)
4. do Insert \((S, (v, w), w(v, w))\)
5. while \( S \) is not empty
6. do \((x, y) \leftarrow \text{Delete-Min}(S)\)
7. if \((x, y)\) is not explored
8. then if \( y \) is not explored
9. then Mark \((x, y)\) as a tree edge
10. Mark \( y \) as explored
11. for every edge \((y, z)\) incident to \( y \)
12. do Insert \((S, (y, z), w(y, z))\)
13. else Mark \((x, y)\) as a non-tree edge
Graph Exploration — Summary

Depth-first search, breadth-first search, Dijkstra’s algorithm, and Prim’s algorithm are variants of the same graph exploration procedure:

Maintain a set of explored vertices. Grow this set by exploring edges incident to these vertices.

What differs is the order in which edges are explored:

- **DFS**: Choose the edge whose source vertex has been discovered most recently.
- **BFS**: Choose the edge whose source vertex has been discovered first.
- **Dijkstra**: Choose the edge whose target is unexplored and has the minimum tentative distance among all unexplored vertices.
- **Prim**: Choose the edge whose target is unexplored and which has minimum weight.
Lemma: Let $s$ be a vertex of $G$, let $T$ be a BFS-tree rooted at $s$, and let $(u, v)$ be an edge of $G$. Then $|\text{dist}_T(s, u) - \text{dist}_T(s, v)| \leq 1$.

In other words, $u$ and $v$ are on the same level or on adjacent levels of $G$. 
Problem: Given a graph $G$, decide whether it is bipartite.
Problem: Given a graph $G$, decide whether it is bipartite.
Problem: Given a graph $G$, decide whether it is bipartite.
**Problem:** Given a graph $G$, decide whether it is bipartite.

**Lemma:** A graph is bipartite if and only if it does not contain an odd cycle.
Lemma: A graph $G$ is bipartite if and only if there are no two adjacent vertices that are on the same level in a BFS-tree of $G$. 
Lemma: A graph $G$ is bipartite if and only if there are no two adjacent vertices that are on the same level in a BFS-tree of $G$. 
Lemma: A graph \( G \) is bipartite if and only if there are no two adjacent vertices that are on the same level in a BFS-tree of \( G \).
Testing Bipartiteness (3)

**TEST-BIPARTITENESS**(*G*):

1. Run BFS on *G* to label all vertices with their distance *d*(v) from some vertex *s*
2. **for** every edge (v, w) of *G*
3. **do if** *d*(v) = *d*(w)
4. **then** Report that *G* is not bipartite and exit
5. \[X \leftarrow \{v \in G \mid *d*(v) \text{ is odd}\}\]
6. \[Y \leftarrow \{v \in G \mid *d*(v) \text{ is even}\}\]
Testing Bipartiteness (3)

\textbf{Test-Bipartiteness}(G)

1. Run BFS on $G$ to label all vertices with their distance $d(v)$ from some vertex $s$
2. for every edge $(v, w)$ of $G$
3. do if $d(v) = d(w)$
4. then Report that $G$ is not bipartite and exit
5. $X \leftarrow \{v \in G \mid d(v) \text{ is odd}\}$
6. $Y \leftarrow \{v \in G \mid d(v) \text{ is even}\}$

\textbf{Lemma:} Given a graph $G$, one can decide in $O(n + m)$ time whether $G$ is bipartite.
A Recursive Depth-First Search Procedure

**DFS**($G$)
1. Mark every vertex and every edge of $G$ as unexplored
2. for every vertex $v$ of $G$
3.   do if $v$ is unexplored
4.       then DFS-from-Vertex($G, v$)

**DFS-from-Vertex**($G, v$)
1. Mark $v$ as explored
2. for every out-edge $(v, w)$ of $v$
3.   do if $w$ is unexplored
4.       then Mark $(v, w)$ as a tree edge
5.       DFS-from-Vertex($G, w$)
6. else Mark $(v, w)$ as a non-tree edge
**Lemma:** For every non-tree edge \((u, v)\) in an undirected graph \(G\) w.r.t. a DFS-tree \(T\) of \(G\), either \(u\) is an ancestor of \(v\) or vice versa; that is, there are no cross edges.
Lemma: Every call to DFS-from-Vertex in Line 4 of Procedure DFS completely explores a connected component of $G$.  

We do not explore more.  

We do not explore less.
**Computing Connected Components**

**CONNECTED-COMPONENTS**\((G')\)

1. \(c \leftarrow 0\)
2. Mark every vertex and every edge of \(G'\) as unexplored
3. **for** every vertex \(v\) of \(G'\)
4. **do** if \(v\) is unexplored
5. **then** \(c \leftarrow c + 1\)
6. Label-Component-from-Vertex\((G, v, c)\)

**LABEL-COMPONENT-FROM-VERTEX**\((G, v, c)\)

1. Mark \(v\) as explored
2. \(\text{component}(v) \leftarrow c\)
3. **for** every out-edge \((v, w)\) of \(v\)
4. **do** if \(w\) is unexplored
5. **then** Mark \((v, w)\) as a tree edge
6. Label-Component-from-Vertex\((G, w, c)\)
7. **else** Mark \((v, w)\) as a non-tree edge
Computing Connected Components

**CONNECTED-COMPONENTS**(*G*)

1. \( c \leftarrow 0 \)
2. Mark every vertex and every edge of *G* as unexplored
3. for every vertex *v* of *G*
4. \( \text{do if } v \text{ is unexplored} \)
5. \( \text{then } c \leftarrow c + 1 \)
6. \( \text{Label-Component-from-Vertex}(G, v, c) \)

**LABEL-COMPONENT-FROM-VERTEX**(*G*, *v*, *c*)

1. Mark *v* as explored
2. component(*v*) \( \leftarrow c \)
3. for every out-edge (*v*, *w*) of *v*
4. \( \text{do if } w \text{ is unexplored} \)
5. \( \text{then Mark } (v, w) \text{ as a tree edge} \)
6. \( \text{Label-Component-from-Vertex}(G, w, c) \)
7. \( \text{else Mark } (v, w) \text{ as a non-tree edge} \)

**Lemma:** The connected components of a graph with *n* vertices and *m* edges can be computed in \( \mathcal{O}(n + m) \) time.
Lemma: When depth-first search backtracks from a vertex $v$, all out-neighbours of $v$ are explored.
**Topological Sorting (2)**

**Top-Sort(G)**

1. \( c \leftarrow n \)
2. Mark every vertex and every edge of \( G \) as unexplored
3. for every vertex \( v \) of \( G \)
4. \hspace{1em} do if \( v \) is unexplored
5. \hspace{2em} then \( c \leftarrow \text{Label-Vertex}(G, v, c) \)

**Label-Vertex(G, v, c)**

1. Mark \( v \) as explored
2. for every out-edge \((v, w)\) of \( v \)
3. \hspace{1em} do if \( w \) is unexplored
4. \hspace{2em} then \( c \leftarrow \text{Label-Vertex}(G, w, c) \)
5. number\((v)\) \(\leftarrow c \)
6. \( c \leftarrow c - 1 \)
7. return \( c \)
Topological Sorting (2)

**Top-Sort**\((G)\)
1. \(c ← n\)
2. Mark every vertex and every edge of \(G\) as unexplored
3. **for** every vertex \(v\) of \(G\)
4. **do if** \(v\) is unexplored
5. **then** \(c ← Label-Vertex(G, v, c)\)

**Label-Vertex**\((G, v, c)\)
1. Mark \(v\) as explored
2. **for** every out-edge \((v, w)\) of \(v\)
3. **do if** \(w\) is unexplored
4. **then** \(c ← Label-Vertex(G, w, c)\)
5. number\((v) ← c\)
6. \(c ← c - 1\)
7. **return** \(c\)

Lemma: A directed acyclic graph can be topologically sorted in \(O(n + m)\) time.
Lemma: If graph $G$ contains a directed cycle $C$, the vertices of $C$ are discovered after the first vertex $v$ in $C$ is discovered and before the recursive call $DFS$-$from$-$Vertex(G, v)$ returns.
Lemma: If graph \( G \) contains a directed cycle \( C \), the vertices of \( C \) are discovered after the first vertex \( v \) in \( C \) is discovered and before the recursive call \( \text{DFS-from-Vertex}(G, v) \) returns.

For procedure Top-Sort, this means that an out-edge of the last vertex \( w \) on the cycle leads to a discovered, but unnumbered vertex, namely \( v \).

The vertices in \( C \) are the vertices on the call stack between \( v \) and \( w \).
Detecting Cycles

\textbf{Detect-Cycle-from-Vertex}(G, v)

1. Mark \(v\) as visited
2. \textbf{for} every out-edge \((v, w)\) of \(v\)
3. \textbf{do if} \(w\) is unvisited
4. \hspace{1em} \textbf{then} \(u \leftarrow \text{Detect-Cycle-from-Vertex}(G, w)\)
5. \hspace{1em} \textbf{if} \(u\) is not nil
6. \hspace{2em} \textbf{then} Output \(v\)
7. \hspace{2em} \textbf{if} \(u = v\)
8. \hspace{3em} \textbf{then} exit
9. \hspace{3em} \textbf{else} return \(u\)
10. \hspace{1em} \textbf{else if} \(w\) is not finished
11. \hspace{2em} \textbf{then} Output \(v\)
12. \hspace{2em} \textbf{return} \(w\)
13. Mark \(v\) as finished
14. return nil
Detecting Cycles (2)

**Detect-Cycle-from-Vertex** \((G, v)\)

1. Mark \(v\) as visited
2. **for** every out-edge \((v, w)\) of \(v\)
   - **do if** \(w\) is unvisited
     - **then** \(u \leftarrow \text{Detect-Cycle-from-Vertex}(G, w)\)
     - **if** \(u\) is not nil
       - **then** Output \(v\)
       - **if** \(u = v\)
         - **then** exit
       - **else** return \(u\)
   - **else if** \(w\) is not finished
     - **then** Output \(v\)
     - **return** \(w\)
3. Mark \(v\) as finished
4. **return** nil

**Lemma:** One can test in \(O(n + m)\) time whether a given directed graph \(G\) contains a directed cycle and, if so, output such a cycle.
Strongly Connected Components

A directed graph $G$ is strongly connected if, for any two vertices $v$ and $w$ in $G$, there exists a path from $v$ to $w$.

The strongly connected components of a directed graph $G$ are the maximal strongly connected subgraphs of $G$.

Alternative formulation: For any two vertices $v$ and $w$ in a strongly connected component, there exists a directed cycle that contains $v$ and $w$. 
A Strong Connectivity Algorithm (1)

Maintain partition of vertices into three sets:

- Finished + live components are strongly connected components of the graph defined by explored edges.
- Finished components are strongly connected components of $G$.
- Live components form a “path” and can merge into larger components as more edges are discovered.
Updating the Partition (1)

- Explore edges out of last live component.
- Three cases, depending on location of target of the edge:
  - Target is finished.
  - Target is unexplored.
  - Target is live.
- When last live component has no unexplored out-edges, mark it as finished and continue processing the previous live component.
Case 1: Edge to finished vertex
Case 1: Edge to finished vertex

Do nothing.
Case 2: Edge to unexplored vertex
**Case 2: Edge to unexplored vertex**

- **Finished**
- **Live**
- **Unexplored**
Case 3: Edge to live vertex
Case 3: Edge to live vertex
Two stacks:

- **Vertex stack** $S$: Contains vertices in order of discovery, numbered in order of discovery
- **Component stack** $C$: Contains one entry per component, the number of the first vertex in this component
A Strong Connectivity Algorithm (2)

**LABEL-COMPONENTS-FROM-VERTEX**\((G, v, c, S, C)\)

1. \(c \leftarrow c + 1\)
2. Label \(v\) as live
3. \(\text{number}(v) \leftarrow c\)
4. Push\((S, v)\)
5. Push\((C, c)\)
6. for every out-edge \((v, w)\) of \(v\) do
   7. if \(w\) is unexplored then
      8. Label-Components-from-Vertex\((G, w, c, S, C)\)
   9. else if \(w\) is live then
      10. repeat \(c' \leftarrow \text{Pop}(C)\)
      11. until \(c' \leq \text{number}(w)\)
      12. Push\((C, c')\)
      13. end if
      14. end if
   15. end for
16. if \(\text{number}(v) = c'\) then
   17. repeat \(w \leftarrow \text{Pop}(S)\)
   18. Mark \(w\) as finished
   19. \(\text{number}(w) \leftarrow c'\)
   20. until \(w = v\)
19. else
   20. Push\((C, c')\)
   21. end if
Lemma: The strongly connected components of a directed graph $G$ can be computed in $O(n + m)$ time.
Summary

*Graphs are fundamental in computer science:*

- Many problems are quite natural to express as graph problems
  - Matching problems
  - Scheduling problems
  - …
- Data structures are graphs whose nodes store useful information

*Graph exploration lets us learn the structure of a graph:*

- Connectivity properties
- Distances between vertices
- Planarity
- …