Undirected Graphs

GRAPH. Set of OBJECTS with pairwise CONNECTIONS.

- Interesting and broadly useful abstraction.

Why study graph algorithms?

- Challenging branch of computer science and discrete math.
- Hundreds of graph algorithms known.
- Thousands of practical applications.

Graphs

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>telephone exchanges, computers,</td>
<td>cables, fiber optics, microwave</td>
</tr>
<tr>
<td></td>
<td>satellites</td>
<td>relays</td>
</tr>
<tr>
<td>circuits</td>
<td>gates, registers, processors</td>
<td>wires</td>
</tr>
<tr>
<td>mechanical</td>
<td>joints</td>
<td>rods, beams, springs</td>
</tr>
<tr>
<td>hydraulic</td>
<td>reservoirs, pumping stations</td>
<td>pipelines</td>
</tr>
<tr>
<td>financial</td>
<td>stocks, currency</td>
<td>transactions</td>
</tr>
<tr>
<td>transportation</td>
<td>street intersections, airports</td>
<td>highways, airway routes</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>software systems</td>
<td>functions</td>
<td>function calls</td>
</tr>
<tr>
<td>internet</td>
<td>web pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>games</td>
<td>board positions</td>
<td>legal moves</td>
</tr>
<tr>
<td>social relationship</td>
<td>people, actors</td>
<td>friendships, movie casts</td>
</tr>
</tbody>
</table>

Graph Jargon

Terminology.

- Vertex: \( v \).
- Edge: \( e = v \rightarrow w \).
- Graph: \( G \).
- \( v \) vertices, \( e \) edges.
- Parallel edge, self loop.
- Directed, undirected.
- Sparse, dense.
- Path.
- Cycle, tour.
- Tree, forest.
- Connected, connected component.
A Few Graph Problems

PATH. Is there a path from s to t?
SHORTEST PATH. What is the shortest path between two vertices?
LONGEST PATH. What is the longest path between two vertices?

CYCLE. Is there a cycle in the graph?
EULER TOUR. Is there a cycle that uses each edge exactly once?
HAMILTON TOUR. Is there a cycle that uses each vertex exactly once?

CONNECTIVITY. Is there a way to connect all of the vertices?
MST. What is the best way to connect all of the vertices?
BI-CONNECTIVITY. Is there a vertex whose removal disconnects graph?

PLANARITY. Can graph be drawn in plane with no crossing edges?
ISOMORPHISM. Do two adjacency matrices represent the same graph?

Graph ADT in C

Standard method to separate clients from implementation.
- Opaque pointer to Graph ADT.
- Plus simple typedef for Edge.

```c
typedef struct graph *Graph;
typedef struct { int v, w; } Edge;
```

GRAPH.h

```c
typedef struct graph *Graph;
typedef struct { int v, w; } Edge;
```

```c
GRAPHinit(int V);
Graph GRAPHinit(int V);
Graph GRAPHrand(int V, int E);
void GRAPHshow(Graph G);
void GRAPHdestroy(Graph G);
void GRAPHinsertE(Graph G, Edge e);
void GRAPHremoveE(Graph G, Edge e);
int GRAPHcc(Graph G);
int GRAPHisplanar(Graph G);
```

```
```

Graph Representation

Vertex names. (A B C D E F G H I J K L M)
- C program uses integers between 0 and V–1.
- Convert via implicit or explicit symbol table.

Two drawing represent same graph.

Set of edges representation.
Adjacency Matrix Representation

- Two-dimensional $V \times V$ array.
- Edge $v\rightarrow w$ in graph: $adj[v][w] = adj[w][v] = 1$.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency Matrix

Graph ADT Implementation: Adjacency Matrix

```c
#include <stdlib.h>
#include "GRAPH.h"

struct graph {
    int V;         // # vertices
    int E;         // # edges
    int **adj;     // V x V adjacency matrix
};

Graph GRAPHinit(int V) {
    Graph G = malloc(sizeof *G);
    G->V = V; G->E = 0;
    G->adj = MATRIXinit(V, V, 0);
    return G;
}

void GRAPHinsertE(Graph G, Edge e) {
    int v = e.v, w = e.w;
    if (G->adj[v][w] == 0) G->E++;
    G->adj[v][w] = G->adj[w][v] = 1;
}
```

Graph ADT Implementation: Adjacency List

```c
#include "GRAPH.h"

typedef struct node *link;

struct node {
    int v;      // current vertex in adjacency list
    link next;  // next node in adjacency list
};

struct graph {
    int V;      // # vertices
    int E;      // # edges
    link *adj;  // array of V adjacency lists
};

link NEWnode(int v, link next) {
    link x = malloc(sizeof *x);
    x->v = v;
    x->next = next;
    return x;
}
```

Adjacency List Representation

- Vertex indexed array of lists.
- Space proportional to number of edges.
- Two representations of each undirected edge.
Adjacency List Graph ADT Implementation

**GRAPH.h**

```c
// initialize a new graph with V vertices
Graph GRAPHinit(int V) {
    int v;
    Graph G = malloc(sizeof *G);
    G->V = V; G->E = 0;
    G->adj = malloc(V * sizeof(link));
    for (v = 0; v < V; v++) G->adj[v] = NULL;
    return G;
}

// insert an edge e = v-w into Graph G
void GRAPHinsertE(Graph G, Edge e) {
    int v = e.v, w = e.w;
    G->adj[v] = NEWnode(w, G->adj[v]);
    G->adj[w] = NEWnode(v, G->adj[w]);
    G->E++;
}
```

Graph Representations

Graphs are abstract mathematical objects.
- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Space</th>
<th>Edge between v and w?</th>
<th>Edge from v to anywhere?</th>
<th>Enumerate all edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency matrix</td>
<td>$O(V^2)$</td>
<td>$O(1)$</td>
<td>$O(V)$</td>
<td>$O(V^2)$</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>$O(E + V)$</td>
<td>$O(E)$</td>
<td>$O(1)$</td>
<td>$O(E + V)$</td>
</tr>
</tbody>
</table>

Most real-world graphs are sparse $\Rightarrow$ adjacency list.

Graph Search

**Goal.** Visit every node and edge in Graph.

**A solution.** Depth-first search.
- To visit a node $v$:
  - mark it as visited
  - recursively visit all unmarked nodes $w$ adjacent to $v$
- To traverse a Graph $G$:
  - initialize all nodes as unmarked
  - visit each unmarked node

Enables direct solution of simple graph problems.
- Connected components.
- Cycles.

Basis for solving difficult graph problems.
- Biconnectivity.
- Planarity.

Depth First Search: Connected Components

```c
#define UNMARKED -1
static int mark[MAXV];

// traverse component of graph
int GRAPHcc(Graph G) {
    int v, id = 0;
    // initialize all nodes as unmarked
    for (v = 0; v < G->V; v++) mark[v] = UNMARKED;
    // visit each unmarked node
    for (v = 0; v < G->V; v++)
        if (mark[v] == UNMARKED) dfsR(G, v, id++);
    return id;
}
```

```c
// return 1 if s and t in same connected component
int GRAPHconnect(int s, int t) {
    return mark[s] == mark[t];
}
```
**Depth First Search: Connected Components**

### Depth First Search: Adjacency Matrix

```c
void dfsR(Graph G, int v, int id) {
    int w;
    mark[v] = id;
    for (w = 0; w < G->V; w++)
        if (G->adj[v][w] != 0 && mark[w] == UNMARKED)
            dfsR(G, w, id);
}
```

### Depth First Search: Adjacency List

```c
void dfsR(Graph G, int v, int id) {
    link t;
    int w;
    mark[v] = id;

    // iterate over all nodes w adjacent to v
    for (t = G->adj[v]; t != NULL; t = t->next) {
        w = t->v;
        if (mark[w] == UNMARKED) dfsR(G, w, id);
    }
}
```

**Connected Components**

**PATHS.** Is there a path from s to t?

<table>
<thead>
<tr>
<th>Method</th>
<th>Preprocess</th>
<th>Query</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union Find</td>
<td>O(E log* V)</td>
<td>O(log* V)</td>
<td>O(V)</td>
</tr>
<tr>
<td>DFS</td>
<td>O(E + V)</td>
<td>O(1)</td>
<td>O(V)</td>
</tr>
</tbody>
</table>

**UF advantage.**
- Dynamic: can intermix query and edge insertion.

**DFS advantage.**
- Can get path itself in same running time.
  - maintain parent-link representation of tree
  - change DFS argument to pass EDGE taken to visit vertex
- Extends to other problems.

**Graphs and Mazes**

**Maze graphs.**
- Vertices = intersections
- Edges = hallways

**DFS.**
- Mark ENTRY and EXIT halls at each vertex.
- Leave by ENTRY when no unmarked halls.

**Breadth First Search**

**Depth-first search.**
- Visit all nodes and edges recursively.
- Put unvisited nodes on a STACK.

**Breadth-first search.**
- Put unvisited nodes on a QUEUE.

**SHORTEST PATH.** What is fewest number of edges to get from s to t?

**Solution.** BFS.
- Initialize mark[s] = 0.
- When considering edge v→w:
  - if w is marked then ignore
  - if w not marked, set mark[w] = mark[v] + 1
### Breadth First Search

**bfs(Graph G, int s)**

```c
link t;
int v, w;
QUEUEput(s);
mark[s] = 0;
while (!QUEUEempty()) {
    v = QUEUEget();
    for (t = G->adj[v]; t != NULL; t = t->next) {
        w = t->v;
        if (mark[w] == UNMARKED) {
            mark[w] = mark[v] + 1;
            QUEUEput(w);
        }
    }
}
```

### Related Graph Search Problems

- **PATHS.** Is there a path from s to t?
  - Solution: DFS, BFS, any graph search.

- **SHORTEST PATH.** Find shortest path (fewest edges) from s to t.
  - Solution: BFS.

- **CYCLE.** Is there a cycle in the graph?
  - Solution: DFS. See textbook.

- **EULER TOUR.** Is there a cycle that uses each edge exactly once?
  - Yes if connected and degrees of all vertices are even.
  - See textbook to find tour.

- **HAMILTON TOUR.** Is there a cycle that uses each vertex exactly once?
  - Solution: ??? (NP-complete)