GRAPH: a set of OBJECTS with CONNECTIONS

Interesting and useful abstraction

Study of graph algorithms
  - challenging branch of computer science
Study of mathematical properties of graphs
  - challenging branch of discrete mathematics

Hundreds of interesting graph algorithms known

Important applications abound
  - transportation systems
  - scheduling
  - circuit simulation
  - software systems
  - web search
  - computer vision
  - computational biology

Glossary of terms

Vertex
Edge
Graph
Dense
Sparse
Path
Cycle, Tour
Tree
Spanning tree
Connected
Connected component
Undirected
Digraph
Weighted
Network

CONCRETE models: direct representations
Ex: Transportation network
  - cities connected by roads
Ex: Electric circuit
  - devices connected by wires
Warning: geometric intuition may mislead
Ex: Airline fares (triangle inequality might not hold)

ABSTRACT models: represent other abstractions
Ex: Scheduling
  - tasks connected by precedence constraints
Ex: Programming system
  - functions that call one another
Ex: CFG
  - symbols related by productions
Ex: Game graphs
  - vertices: board positions; edges: moves
Representing graphs

Graphs are abstract mathematical objects. ADT implementations require specific representations.

As usual:
- Many different representations possible.
- Efficiency depends on matching algs to representations.

Standard issues apply:
- Space vs. time
- Array vs. linked list
- Integers vs. reals
- Symbol tables
- Duplicate vertices or edges?
- Mix of ADT operations.

---

Adjacency matrix representation

V-by-V array gives constant-time edge existence test.

**Vertex-indexed array of vertex-indexed arrays**: One entry for each vertex.

**Adjacency matrix**: Vertex-indexed array of vertex-indexed arrays.
- 1 in (i,j) AND (j,i) iff edge i-j in graph.

<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>A</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>B</td>
<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>
</tr>
<tr>
<td>C</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>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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>F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</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>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>1</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>
</tr>
<tr>
<td>H</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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I</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>
<td>0</td>
</tr>
<tr>
<td>J</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>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>K</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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</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>1</td>
<td>1</td>
</tr>
<tr>
<td>M</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>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

Adjacency lists representation

Array of lists takes space proportional to no. of edges.

**Adjacency lists**: Representation.

- Two drawings that represent the same graph.

**Vertex names (A B C D E F G H)**
- Progs use integers between 0 and V-1.
- Convert via implicit or explicit symbol table.

**Set of edges** representation:

---

TWO representations of each edge for undirected graphs.
Graph ADT

Standard mechanism to separate clients from implementations
(plus simple typedef for edges)

GRAPH.h:
- typedef struct { int v; int w; } Edge;
- Edge EDGE(int, int);
- typedef struct graph *Graph;
- Graph GRAPHinit(int);
- void GRAPHinsertE(Graph, Edge);

Typical client program
- calls GRAPHinit to create data structures
- uses Graph handle as arg to graph-processing ADT function
- calls GRAPHinsertE to build graph by adding edges
- calls ADT function to do graph processing

Ex: GRAPHcc computes connected components.

Adjacency lists Graph ADT implementation

```c
#include "GRAPH.h"
typedef struct node *link;
struct node { int v; link next; };
struct graph { int V; int E; link *adj; };
link NEW(int v, link next)
{ link x = malloc(sizeof *x);
  x->v = v; x->next = next;
  return x;
}
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;
}
void GRAPHinsertE(Graph G, Edge e)
{ int v = e.v, w = e.w;
  G->adj[v] = NEW(w, G->adj[v]);
  G->adj[w] = NEW(v, G->adj[w]);
  G->E++;
}
```

Summary of basic costs

E edges, V vertices

Space requirements:
- Adjacency lists: V+E
- Adjacency matrix: V^2
- Set of edges: E (+V)

Choice of representation affects algorithm efficiency
- even for simple primitives

Ex: Is there an edge from A to B?
- lists: O(E)
- matrix: O(1)

Ex: Is there an edge from A to anywhere?
- lists: O(1)
- matrix: O(V)

Basic graph problems (short list)

PATHS
- Is there a path from A to B?
CYCLES
- Does the graph contain a cycle?
CONNECTIVITY (SPANNING TREE)
- Is there a way connect all the vertices?
BICONNECTIVITY
- Is there a vertex whose removal will disconnect the graph?
PLANARITY
- Is there a way to draw the graph without edges crossing?
SHORTEST (LONGEST) PATH
- What is the shortest (longest) way from A to B?
MINIMAL SPANNING TREE
- What is the best way to connect the vertices?
HAMILTON TOUR
- Is there a cycle that uses each vertex exactly once?
ISOMORPHISM
- Do two given adj matrices represent the same graph?
Traversing graphs

Goal: VISIT every vertex in the graph

**Depth-first search (DFS)**
- To VISIT a node \( k \)
  - mark it (recursively) VISIT all unmarked vertices connected to \( k \)
- To TRAVERSE a graph
  - initialize all nodes to be unmarked
  - VISIT each unmarked node

Solves some simple graph problems
- connectivity
- cycles
- basis for solving some difficult graph problems
  - biconnectivity
  - planarity

Traversing a graph's components

Needed for any implementation of VISIT UNLESS graph is known to be connected

IF visit(\( k \)) marks all nodes connected to \( k \)
THEN traverse(G) marks all of G's nodes

```c
int mark[maxV]; int cnt = 0;
traverse(Graph G)
{
    int k;
    for (k = 1; k <= G->V; k++) mark[k] = 0;
    for (k = 1; k <= G->V; k++)
        if (mark[k] == 0) visit(G, k);
}
```

DFS implementation

**Adjacency matrix**

```c
visit(Graph G, int k)
{
    int t;
    mark[k] = ++cnt;
    for (t = 1; t <= V; t++)
        if (G->adj[k][t] != 0)
            if (mark[t] == 0) visit(G, t);
}
```

**Adjacency lists**

```c
visit(Graph G, int k)
{
    link t;
    mark[k] = ++cnt;
    for (t = G->adj[k]; t != z; t = t->next)
        if (mark[t->v] == 0) visit(G, t->v);
}
```

DFS example (adjacency lists)

```
visit A
- visit F (first on A's list)
  - check A on F's list (been there)
  - visit E (second on F's list)
    - visit G (first on E's list)
      - check E on G's list (been there)
      - check A on G's list (been there)
    - check F on E's list (been there)
      - check F on D's list (been there)
    - visit D (third on E's list)
      - check F on D's list (been there)
      - check E on D's list (been there)
      - check D on F's list (done that)
  - visit C (second on A's list)
  - visit B (third on A's list)
  - check G on F's list (done that)
  - ...
```

"been there": currently working on it
"done that": totally finished dealing with it
DFS tree (adjacency lists)

Tree structure captures dynamics of DFS

**TREE links**
- first encounter: recursive call
- second encounter: been there

**BACK links**
- first encounter: been there
- second encounter: done that

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A
H: I
I: H
J: K L M
K: J
L: J M
M: J L

Connected components ADT function

Is there a path from s to t?

**UNION-FIND** (lecture 1)
- query: \(O(\log* V)\)
- preprocessing: \(O(E \log* V)\)
- space: \(O(V)\)

**DFS**
- query: \(O(1)\)
- preprocessing: \(O(E)\)
- space: \(O(V)\)

UF advantage: can intermix query and edge insertion

DFS advantage: can give client the path
- change arg to pass EDGE taken to visit the vertex
- maintain parent-link representation of DFS tree
  - [see text]

Connected-components ADT functions (DFS)

**GRAPHcc**: preprocessing (DFS)
**GRAPHconnect**: query
**cc**: vertex-indexed array in graph representation

```c
void dfsRcc(Graph G, int v, int id)
{ link t;
  G->cc[v] = id;
  for (t = G->adj[v]; t != NULL; t = t->next)
    if (G->cc[t->v] == -1) dfsRcc(G, t->v, id);
}
int GRAPHcc(Graph G)
{ int v, id = 0;
  G->cc = malloc(G->V * sizeof(int));
  for (v = 0; v < G->V; v++)
    G->cc[v] = -1;
  for (v = 0; v < G->V; v++)
    if (G->cc[v] == -1) dfsRcc(G, v, id++);
  return id;
}
int GRAPHconnect(Graph G, int s, int t)
{ return G->cc[s] == G->cc[t]; }
```

Graph-search overview

DFS is one of a family of graph-search functions
- all visit all nodes and edges
- strategy to use dictated by problem at hand

**GENERALIZED GRAPH SEARCH**
To TRAVERSE a graph
- initialize all nodes to be unmarked
- put some vertex on a generalized queue (GQ)
- while the GQ is nonempty
  - remove a vertex and mark it
  - put all unmarked adjacent vertices on the GQ

**ISSUE**: duplicate vertices on queue
- ignore the new one or forget the old one?
Stack-based graph traversal

Use explicit stack instead of recursive calls

```
visit(Graph G, int k)
{ link t;
    STACKpush(k);
    while (!STACKempty())
    {
        k = STACKpop(); mark[k] = ++id;
        for (t = G->adj[k]; t != z; t = t->next)
            if (mark[t->v] == 0)
            { STACKpush(t->v); mark[t->v] = -1; }
    }
}
```

Stack-based traversal example (adjacency lists)

Visit A
- push F, push C, push B, push G
- visit G
  - push E, been to A
  - visit E
    - been to G, been to F, push D
    - visit D
      - been to F, been to E
- visit B
  - been to A
- visit C
  - been to A
- visit F
  - been to A, done with E, done with D

Stack-based search

Not the same as recursive DFS. Why?

Algs differ in treatment of vertices that are adjacent to partially visited vertices
- DFS: visits such a vertex
- stack-based: avoids it
  - (it is on the stack and will get visited later)

Nonrecursive DFS: PUSH next node on adj list
- equivalent to disallowing duplicate vertices on stack

No particular reason to use stack
- other ADTs work as well (stay tuned)

Graph search: generalized-queue-based traversal

Graphs and mazes

Vertices: intersections
Edges: hallways

DFS
- mark ENTRY and EXIT halls at each vertex
- leave by ENTRY when no unmarked halls

Stack-based?
Breadth-first search (BFS)

Put unvisited nodes on a QUEUE, not a stack

```
visit(Graph G, int k)
{
    link t;
    QUEUEput(k);
    while (!QUEUEempty())
    {
        k = QUEUEget(); mark[k] = ++id;
        for (t = G->adj[k]; t != z; t = t->next)
            if (mark[t->v] == 0)
            {
                t->v = QUEUEput(); mark[t->v] = -1;
            }
    }
}
```

DFS example

Search order depends on graph representation

BFS vs DFS example

Depth-first search

Same graph, different order of edges on adj lists

Breadth-first search
Graph search and path problems

Problem: PATHS
- Is there a path from A to B?
Solution: DFS, BFS, any graph search

Problem: SHORTEST PATH
- Find a shortest path (fewest edges) from A to B.
Solution: BFS

Problem: EULER PATH (existence)
- Is there a cycle that uses each EDGE exactly once?
Solution: Yes, if degrees of all vertices are even

Problem: EULER TOUR
- Find a cycle that uses all the graph's edges.
Solution: interesting exercise [see text]

Problem: HAMILTON TOUR
- Is there a cycle that uses each VERTEX exactly once?
Solution: ?? (NP-complete)