GRAPHS

UNDIRECTED GRAPHS

Ruben Acuña

INTRODUCTION

TERMINOLOGY REVIEW

- Graphs
  - Directed vs Undirected
  - Vertices (adjacent)
  - Edges (incident)

- Self-looping nodes
- Parallel edges

TERMINOLOGY REVIEW

- [Simple] Path
- [Simple] Cycle
- Length

- Connected
- Connected Components

GRAPH API OVERVIEW

SOP: graphs are generic data structures, with multiple representations, so define an ADT to enforce the separation of function and implementation.

- Is this API complete enough?

public class	Graph	
	Graph(int V)	*create a V-vertex graph with no edges*
	Graph(In in)	*read a graph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v-w *to this graph*
Iterable<Integer>	adj(int v)	*vertices adjacent to* v
String	toString()	*string representation*

API for an undirected graph

GRAPH API USAGE

- Compute the degree of a vertex:

```
public int degree(Graph G, int v) {
    int degree = 0;
    for (int w : G.adj(v))
        degree++;
    return degree;
}
```

- Count number of self-loops in graph:

```
public int numberOfSelfLoops(Graph G) {
    int count = 0;
    for (int v = 0; v < G.V(); v++)
        for (int w : G.adj(v))
            if (v == w)
                count++;
    return count/2;
}
```

GRAPH DATA STRUCTURES

IMPLEMENTING THE GRAPH ADT

- Ideally we want:
  - Low space usage: linear (e.g., V+E)
  - Fast to add edges: constant (e.g., 1)
  - Fast to find edge on vertex (e.g., V)

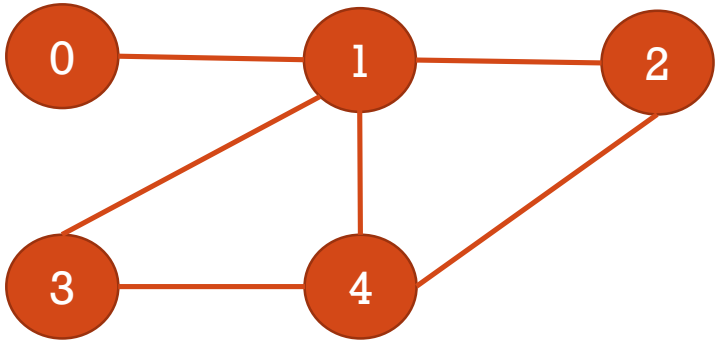
We also want to make sure we support:

- Self-loops
- Parallel edges

Possible ways:

- Adjacency Matrix
- Adjacency Lists

ADJACENCY MATRIX REPRESENTATION



Idea: for a graph of V vertices, make a V by V array, where each index corresponds directly to an edge.

Use a boolean array.

	0	1	2	3	4
0		T			
1	T		T	T	T
2		T			T
3		T			T
4		T	T	T	

Let's evaluate it:

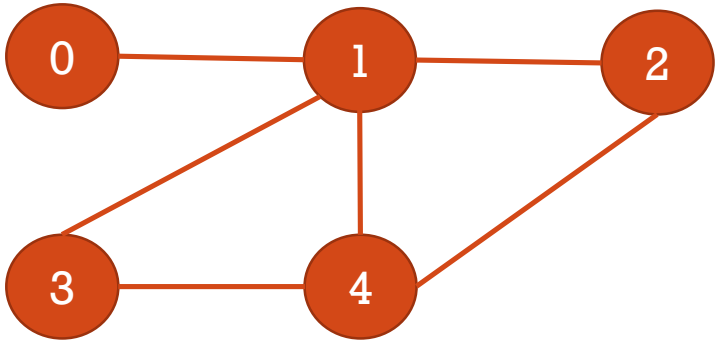
- How much spaces does this require?
- Does it support self-loops?
- Does it support parallel edges?

The book doesn't bother with this approach at all…

- Is there really no place we want to use this idea?

ADJACENCY LIST REPRESENTATION

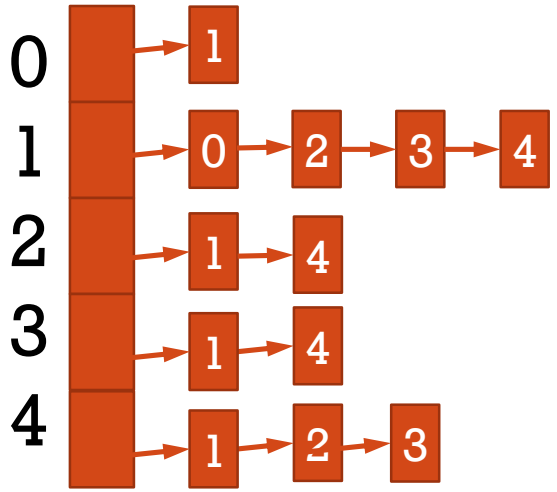
Idea: for a graph of V vertices, make an array of V lists, where each index corresponds to a list of vertices that are connected to that index.



Uses a LinkedList array.

Let's evaluate it:

- How much spaces does this require?
- Does it support self-loops?
- Does it support parallel edges?



(won't be sorted in practice)

CODE

Nothing special.

Below is a performance summary.

```
private final int V;
private int E;
private LinkedList<Integer>[] adj;
public AdjListGraph(int V) {
    this.V = V;
    this.E = 0;
    adj = (LinkedList<Integer>[]) new LinkedList[V];
    for (int v = 0; v < V; v++)
        adj[v] = new LinkedList<>();
}

public int V() { return V; }
public int E() { return E; }

public void addEdge(int v, int w) {
    adj[v].add(w);
    adj[w].add(v);
    E++;
}

public Iterable<Integer> adj(int v) {
    return adj[v];
}
```

Data Structure	Space	Add edge	Check adjacent	Iterate adjacent
Adjacency matrix	V^2	1	1	V
Adjacency list	V+2E	1	degree(V)	degree(V)

THE SEARCH PROBLEM

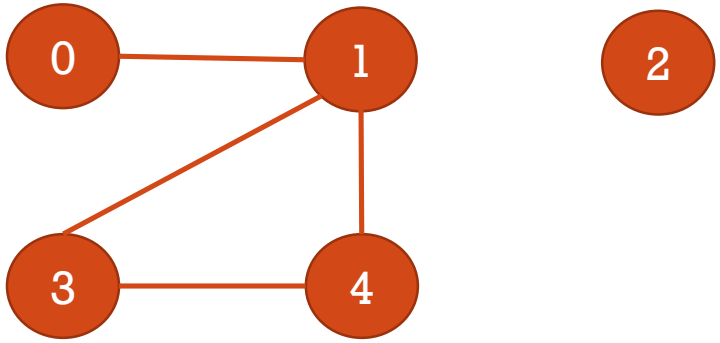
INTRODUCTION

- The basic problem: given an undirected graph, are two vertices connected?

- Our text tries to be a little fancy: it defines an API for checking if vertices are connected.
- A little unnecessary to create an API for one graph function…
- We'll keep it since we'll get a UML diagram at the end that tells us how things are related.

(marked implies visited)

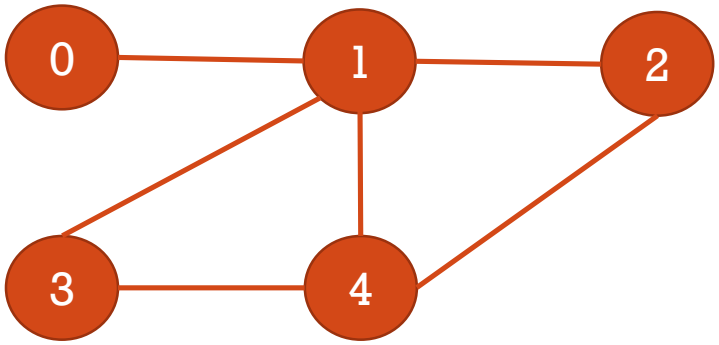
public class	Search	
	Search(Graph G, int s)	*find vertices connected to a source vertex s*
boolean	marked(int v)	*is v connected to s?*
int	count()	*how many vertices are connected to s?*



S = new Search(G, 0)
S.count()→3
S.marked(4)→ T
S.marked(2)→ F

DEPTH-FIRST SEARCH

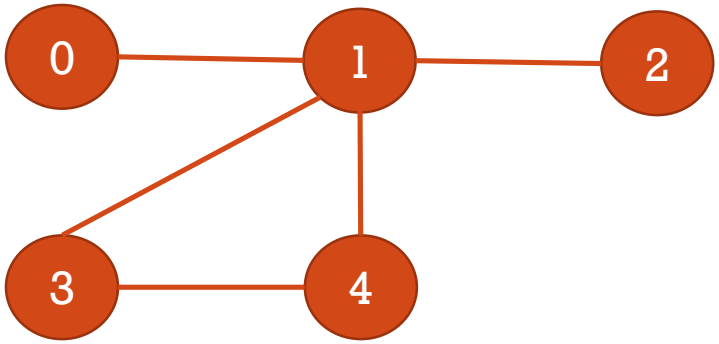
- Basic Idea:
- Start at some node:
  - Mark it as visited.
  - Recursively visit each of its neighbors if they are not marked.



The book has several very detailed images showing the DFS process – check them out.

```
private void dfs(Graph G, int v) {
        marked[v] = true;
        count++;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }
```

ALGORITHM TRACE



- Run the DFS algorithm on this graph to find paths from node 0.
- Your trace should list: each call to DFS, when a neighbor is checked, and when a node is marked. Assume the edges are discovered in the order of the node labels they connect.

IMPLEMENTATION

- Relatively straightforward.
- The real work is being done by the stack (?) that tracks where we have been.

```
private boolean[] marked;
private int count;

public DepthFirstSearch(Graph G, int s) {
    marked = new boolean[G.V()];
    dfs(G, s);
}

private void dfs(Graph G, int v) {
    marked[v] = true;
    count++;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w);
}

public boolean marked(int w) {
    return marked[w];
}

public int count() {
    return count;
}
```

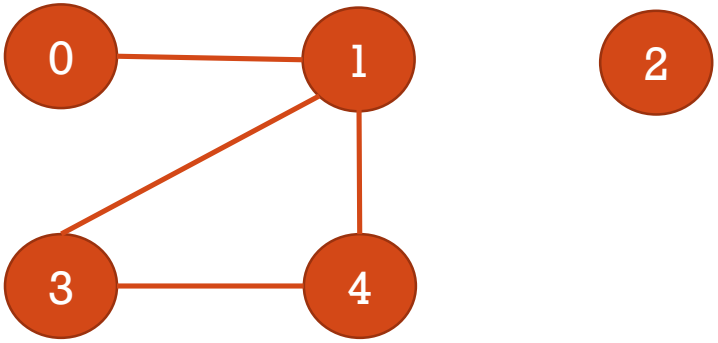
VISUAL TRACE

THE PATHING PROBLEM

INTRODUCTION

- The problem: given an undirected graph, are two vertices connected? If so, what is the path that connects them?
  - Also: is it "the" path or "a" path?
- Can easily update DFS to handle this: keep track of vertex that lead you were you went.

public class	Paths	
	Paths(Graph G, int s)	*find paths in G from source s*
boolean	hasPathTo(int v)	*is there a path from s to v?*
Iterable<Integer>	pathTo(int v)	*path from s to v; null if no such path*



Paths(G, 0)
hasPathTo(2)→ False
hasPathTo(4)→ True
pathTo(4)→(0, 1, 4)

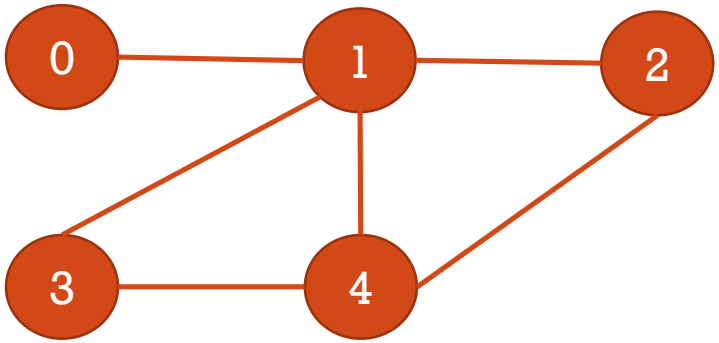
USING DFS

- New variable is edgeTo.
- edgeTo gets updated for each call in dfs().

- Seems to work fine – but what about the length of the path found? Is it the shortest?

```
private boolean[] marked;
private int[] edgeTo;
private final int s;
public DepthFirstPaths(Graph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    this.s = s;
    dfs(G, s);
}
private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v))
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
}
public boolean hasPathTo(int v) {
    return marked[v];
}
public Iterable<Integer> pathTo(int v) {
    if(!hasPathTo(v))
        return null;
    Stack<Integer> path = new Stack<>();
    for(int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}
```

BREADTH-FIRST SEARCH



Basic Idea:

- Starting at some node:
  - Mark it as visited.
  - Add each of its neighbors to a queue if they are not marked.
  - Continue, using the queue to look at nodes in FIFO order.

The result is that we see all the nodes that are 1 edge away from the root, then 2 edges away, and so on.

The book has several very detailed images showing the BFS process – check them out.

```
private void bfs(Graph G, int s) {
        Queue<Integer> queue = new Queue<>();

        marked[s] = true;
        queue.add(s);

        while (!queue.isEmpty()) {
            int v = queue.remove();
            for (int w : G.adj(v))
                if (!marked[w]) {
                    edgeTo[w] = v;
                    marked[w] = true;
                    queue.add(w);
                }
        }
}
```

ALGORITHM TRACE

IMPLEMENTATION

- Very close to the path finding code for DFS.
- However, we have removed recursion (the implicit use of a stack) with the explicit use of a queue.
- Seems to work fine – but what about the length of the path found?

```
private boolean[] marked;
private int[] edgeTo;
private final int s;

public BreadthFirstPaths(Graph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    this.s = s;
    bfs(G, s);
}

private void bfs(Graph G, int s) {
    Queue<Integer> queue = new LinkedList<>();

    marked[s] = true;
    queue.add(s);

    while (!queue.isEmpty()) {
        int v = queue.remove();
        for (int w : G.adj(v))
            if (!marked[w]) {
                edgeTo[w] = v;
                marked[w] = true;
                queue.add(w);
            }
    }
}
//hasPath, and pathTo are same.
```

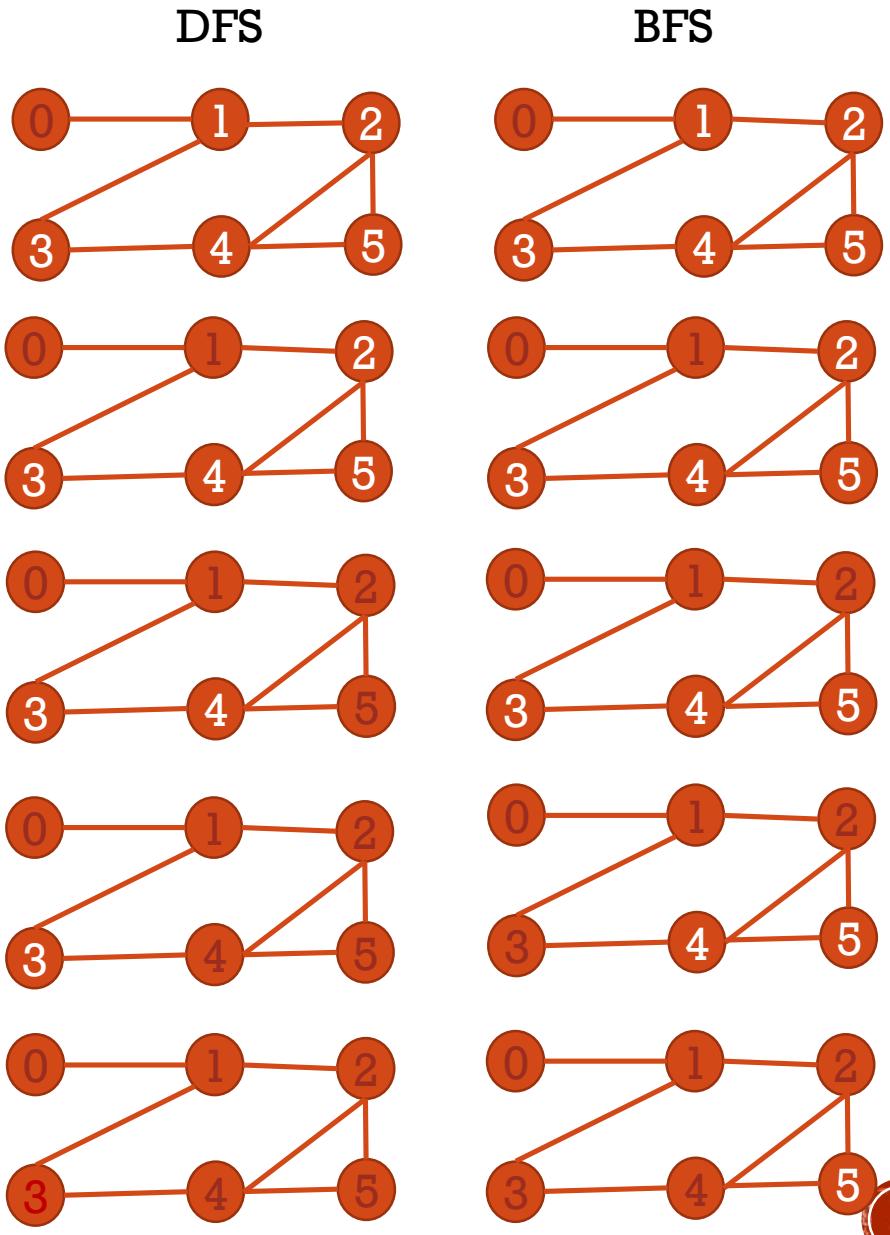
BFS VS DFS

Although DFS and BFS, can solve the same times of problems, how they explore the graph differs.

In DFS, we try to extend the current path by using newest node we've seen.

In BFS, we try to extend the path by looking the oldest unexplored node we've seen.

If we didn't know the size of a graph, which algorithm would be safer to use?



25

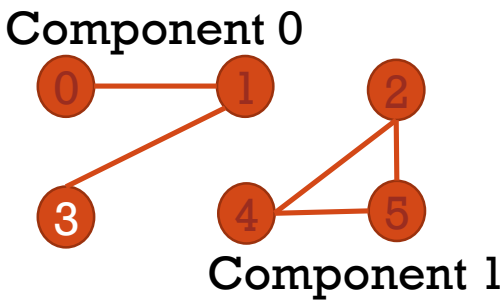
GRAPH THEORETIC APPLICATIONS

CONNECTED COMPONENTS

- Recall that a connected component, is a subset of a graph where every pair of vertices are connected by some path.

public class	CC	
	CC(Graph G)	*preprocessing constructor*
boolean	connected(int v, int w)	*are* v *and* w *connected?*
int	count()	*number of connected components*
int	id(int v)	*component identifier for* v *(between* 0 *and* count()-1*)*

- Assuming we have this functionality available, how it help us solve the connectivity problem?



CC CODE

- No real surprises here.
- The only thing to note is this algorithm is still O(V+E) since it uses marked to make sure it only processes each vertex once.

```
private boolean[] marked;
private int[] id;
private int count;

public CC(Graph G) {
    marked = new boolean[G.V()];
    id = new int[G.V()];
    for (int s = 0; s < G.V(); s++)
        if (!marked[s]) {
            dfs(G, s);
            count++;
        }
}

private void dfs(Graph G, int v) {
    marked[v] = true;
    id[v] = count;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w);
}

public boolean connected(int v, int w) {
    return id[v] == id[w];
}

public int id(int v) { return id[v]; }

public int count() { return count; }
```

DETECTING CYCLES

- In addition to paths, we also mentioned cycles.
- Recall that a cycle is a path where the first and last nodes are the same.

- Could we leverage any of the algorithms we have talked about to check if graph contains a cycle?

2-COLORING

- The k-coloring problem: can the vertices of a graph be colored with k different colors in a way that no adjacent nodes have the same color?

- Could we leverage any of the algorithms we have talked about to check if a graph is 2-colorable?