4.2 Directed Graphs

- Introduction
- Digraph API
- Digraph search
- Topological sort
- Strong components

Directed graphs

**Digraph.** Set of vertices connected pairwise by directed edges.

Road network

Vertex = intersection; edge = one-way street.

Political blogosphere graph

Vertex = political blog; edge = link.

Overnight interbank loan graph

Vertex = bank; edge = overnight loan.
Uber taxi graph

Vertex = taxi pickup; edge = taxi ride.

[Image of Uber taxi graph]

http://blog.uber.com/2012/01/09/uberdatsanfranciscosf/ 1

Implication graph

Vertex = variable; edge = logical implication.

[Image of Implication graph]

Vertex = variable; edge = logical implication.

Combinational circuit

Vertex = logical gate; edge = wire.

[Image of Combinational circuit]

WordNet graph

Vertex = synset; edge = hypernym relationship.

[Image of WordNet graph]

http://wordnet.princeton.edu 10

Digraph applications

<table>
<thead>
<tr>
<th>digraph</th>
<th>vertex</th>
<th>directed edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>bank</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>gene</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>

[Image of Digraph applications]

Some digraph problems

<table>
<thead>
<tr>
<th>problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s -&gt; t path</td>
<td>Is there a path from s to t?</td>
</tr>
<tr>
<td>shortest s -&gt; t path</td>
<td>What is the shortest path from s to t?</td>
</tr>
<tr>
<td>directed cycle</td>
<td>Is there a directed cycle in the graph?</td>
</tr>
<tr>
<td>topological sort</td>
<td>Can the digraph be drawn so that all edges point upwards?</td>
</tr>
<tr>
<td>strong connectivity</td>
<td>Is there a directed path between all pairs of vertices?</td>
</tr>
<tr>
<td>transitive closure</td>
<td>For which vertices u and v is there a directed path from u to v?</td>
</tr>
<tr>
<td>PageRank</td>
<td>What is the importance of a web page?</td>
</tr>
</tbody>
</table>
4.2 Directed Graphs

- Digraph API

What about an interface?

Since Graph and Digraph have many of the same methods, we can put them in an interface:

```java
public interface BaseGraph {
    int V();
    int E();
    void addEdge(int v, int w);
    Iterable<Integer> adj(int v);
}
```

public class Graph implements BaseGraph {
    int nV;
    int nE;
    // other methods...
}

public class Digraph implements BaseGraph {
    int nV;
    int nE;
    // other methods...
}
```

- Then we could write generic code across a group of graphs, some of which are Graphs and others Digraphs
- However, this is a rare thing to do, so we’ll adopt S&W’s simpler approach of treating them separately as types.
- Another way to handle a group of Graphs and Digraphs: convert the Graphs to equivalent Digraphs (all edges both ways).

Digraph API

Almost identical to Graph API.

```java
public class Digraph {
    Digraph(int V) { ... }  // create an empty digraph with V vertices
    Digraph(In in) { ... }  // create a digraph from input stream
    void addEdge(int v, int w) { ... }  // add a directed edge v→w
    Iterable<Integer> adj(int v) { ... }  // vertices pointing from v
    int V() { ... }  // number of vertices
    int E() { ... }  // number of edges
    Digraph reverse() { ... }  // reverse of the digraph
    String toString() { ... }  // string representation
}
```

Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.

```java
public static Digraph createDigraphFromFile(String filePath) {
    Scanner in = new Scanner(new File(filePath));
    System.out.println("Creating graph of " + filePath);
    int nV = Integer.parseInt(in.nextLine());
    int nE = Integer.parseInt(in.nextLine());
    System.out.println("Creating graph of " + nV + " nodes + " + nE + " edges");
    Digraph G = new Digraph(nV);  // empty digraph with
    while (in.hasNextLine()) {
        String line = in.nextLine().trim();  // trim off spaces on either end
        String[] tokens = line.split(" ");  // allow multiple spaces between
        int w = Integer.parseInt(tokens[1].trim());
        G.addEdge(v, w);
    }
    in.close();
    return G;
}
```

Basic usage:

```java
Digraph G = createDigraphFromFile("tinyDG.txt");
for (int v : G.V()) {
    StdOut.println(v + "'s neighbors:");
    for (int w : G.adj(v)) {
        StdOut.println(w);
    }
}
```
In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from v.
- Real-world digraphs tend to be sparse.

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from v to w</th>
<th>edge from v to w?</th>
<th>iterate over vertices pointing from w?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>(</td>
<td>)</td>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>( V \times V )</td>
<td>(</td>
<td>)</td>
<td>(</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>(</td>
<td>\times V )</td>
<td>(</td>
<td>)</td>
</tr>
</tbody>
</table>

\( V \) — huge number of vertices, small average vertex degree

- Algorithms based on iterating over vertices pointing from v.
- Real-world digraphs tend to be sparse.

Adjacency-lists Digraph representation: Java implementation

```java
public class Digraph
{
    private final int V;
    private final Bag<Integer>[] adj;

    public Digraph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[])(new Bag[V]);
        for (int v = 0; v < V; ++v)
            adj[v] = new Bag<Integer>();
    }

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

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

Recall Digraph API

Almost identical to Graph API. One additional method: reverse()

```java
public class Digraph
{
    private final int V;
    private final Bag<Integer>[] adj;

    public Digraph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[])(new Bag[V]);
        for (int v = 0; v < V; ++v)
            adj[v] = new Bag<Integer>();
    }

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

    public Iterable<Integer> adj(int v)
    {
        return adj[v];
    }
}
```
Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

**DFS (to visit a vertex v)**

Mark v as visited.
Recursively visit all unmarked vertices w pointing from v.

Depth-first search algorithm

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v.

Depth-first search (in undirected graphs)

Recall code for undirected graphs.

```java
public class DepthFirstSearch
{
    private boolean[] marked;

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

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

        public boolean visited(int v)
        { return marked[v]; }
    }
}
```

Depth-first search (in directed graphs)

Code for directed graphs identical to undirected one. (Substitute Digraph for Graph)

```java
public class DirectedDFS
{
    private boolean[] marked;

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

    private void dfs(Digraph G, int s)
    {
        marked[s] = true;
        for (int w : G.adj(s))
            if (!marked[w]) dfs(G, w);

        public boolean visited(int v)
        { return marked[v]; }
    }
}
```
Reachability application: program control-flow analysis

Every program is a digraph.
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead code elimination.
Find (and remove) unreachable code.

Infinite loop detection.
Determine whether exit is unreachable.

Reachability application: mark-sweep garbage collector

Every data structure is a digraph.
- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).

Mark-sweep algorithm. [McCarthy, 1960]
- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).

Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.
- Reaching.
- Path finding.
- Topological sort.
- Directed cycle detection.

Basis for solving difficult digraph problems.
- 2-satisfiability (in implication graphs, not covered)
- Directed Euler path (not covered)
- Strongly connected components (covered by clever algorithm, not that difficult)

Depth-first search in digraphs algorithm

Repeat until queue is empty:
- Remove vertex + from queue.
- Add to queue all unmarked vertices pointing from + and mark them.

Proposition. BFS computes shortest paths (fewest number of edges) from + to all other vertices in a digraph in time proportional to E + V.
Directed breadth-first search algorithm

Repeat until queue is empty:
  • Remove vertex v from queue.
  • Add to queue all unmarked vertices pointing to v and mark them.

Breadth-first search in digraphs application: web crawler

Goal. Crawl web, starting from some root web page, say web.princeton.edu

Solution. [BFS with implicit digraph]
  • Choose root web page as source v.
  • Maintain a queue of websites to explore.
  • Maintain a set of discovered websites.
  • Dequeue the next website and enqueue websites to which it links (provided you haven’t done so before).

Q. Why not use DFS?

Web crawler output

BFS crawl

http://web.princeton.edu
http://www.ai.org
http://tango.princeton.edu
http://www.gutenberg.org
http://library.princeton.edu
http://esl.princeton.edu
http://www.mitpressjournals.org
http://mitpressjournals.org
http://www.worldcat.org
http://www.library.princeton.edu
http://artmuseum.princeton.edu
http://princetonusg.com
http://vimeo.com
http://chatroom.lenovo.com
http://www.google.com
http://www.deimos.apple.com
http://www.princeton.edu
http://ogp.me
http://www.w3.org

DFS crawl

http://web.princeton.edu
http://www.ai.org
http://tango.princeton.edu
http://www.gutenberg.org
http://library.princeton.edu
http://esl.princeton.edu
http://www.mitpressjournals.org
http://mitpressjournals.org
http://www.worldcat.org
http://www.library.princeton.edu
http://artmuseum.princeton.edu
http://princetonusg.com
http://vimeo.com
http://chatroom.lenovo.com
http://www.google.com
http://www.deimos.apple.com
http://www.princeton.edu
http://ogp.me
http://www.w3.org

Bare-bones web crawler: Java implementation

(SC310: don’t worry about details here)

Queue<String> queue = new Queue<String>();

String root = 
queue.enqueue(root);
String root = 
String regexp = 
Pattern pattern = Pattern.compile(regexp);
Matcher matcher = pattern.matcher(input);
String input = in.readAll();
In in = new 
StdOut.println(v);
String v = queue.dequeue();

while (!queue.isEmpty()) { String w = matcher.group();
if (!marked.contains(w)) {
marked.add(w);
queue.enqueue(w);
Shortest path to
Shortest path from

Multiple-source shortest paths

Multiple-source shortest paths. Given a digraph and a set of source vertices, find shortest path from any vertex in the set to each other vertex.

Ex. S = \{1, 7, 10\}.
  • Shortest path to 4 is 7-\rightarrow-4.
  • Shortest path to 5 is 7-\rightarrow-4-\rightarrow-5.
  • Shortest path to 12 is 10-\rightarrow-12.
  • ...