CodeClub JU : Session 5

Introduction to Graphs

Siddhanth Gupta
BCSE-4
siddhanthgupta@gmail.com

Jadavpur University - Dept. of CSE

2015-2016 Session
15th January 2016

Outline

Introduction

Things we hope to cover today

- Introduction to Graphs: What are graphs?
- Representing Graphs in C++ (or any language for that matter)
- An Introduction to Graph Search (DFS and BFS)
- LOTS of problems!!

We're a little short on time this evening, so I may be a bit fast. Please ask me to slow down if you have any problems whatsoever.

What is a Graph?

A graph G is a **Data Structure** composed of two ingredients:

1. A set of vertices *AKA*, *nodes* represented by the symbol **V**
2. A set of edges represented by the symbol **E**
    - An edge is essentially just a **pair of vertices**
    - Pair can be unordered (resulting in an **undirected graph**) or ordered (**directed graph**)

What is a Graph?

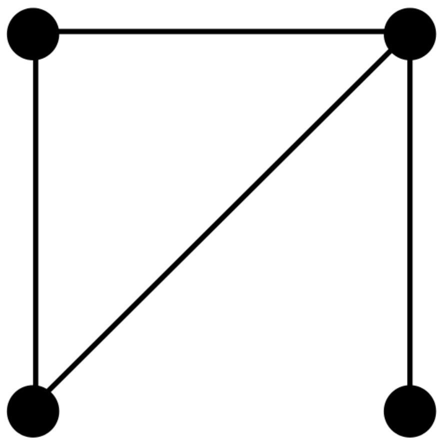


Figure : A simple undirected graph

What is a Graph?

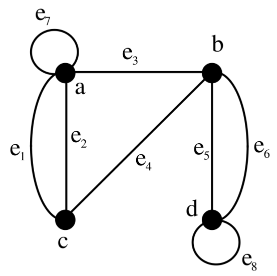


Figure : An undirected graph with parallel edges and loops

For the most part, we will **NOT** consider graphs having loops or parallel edges

Running Time of a Graph Algorithm

- Running time of an algorithm that operates on a graph require discussion about the "input size".
- For arrays, we had the number of elements **n**
- For graphs, we have the number of vertices **n** and the number of edges **m**

Consider an undirected graph with n vertices, no parallel edges, and is connected (i.e., in one piece). What is the minimum and maximum number of edges that the graph could have, respectively?

Sparse Graphs and Dense Graphs

- **Number of vertices = n**
- **Number of edges = m**
- In most (but not all) applications, **m** is $\Omega(n)$ and $O(n^2)$ (when there are no parallel arcs)
- Sparse means closer to lower bound
- Dense means closer to upper bound
- Usually, greater than $n \log n$ is dense

Adjacency Matrix

- Not really used for most of the applications due to large space complexity
- **Graph is represented by a 0-1 matrix A_i where** $A_{ij} = 1 \iff$ G has an i-j edge
- There can be many variants: Value of A_{ij} may represent weight of the graph
- In an undirected graph, $A_{ij} = A_{ji}$
- What is the space complexity? The space complexity turns out to be $O(n^2)$ since the adjacency matrix is a $n \times n$ matrix.

Adjacency Lists

What is an adjacency list made of?

- Array (or list) of vertices
- Array (or list) of edges
- Each edge points to it's endpoints
- Each vertex points to edges incident on it

What is the space complexity?

Adjacency Lists

- Array (or list) of vertices requires $\theta(\mathbf{n})$
- Array (or list) of edges requires $\theta(\mathbf{m})$
- Each edge points to it's endpoints requires $\theta(\mathbf{m})$
- Each vertex points to edges incident on it requires $\theta(\mathbf{m})$

So the complexity is $\theta(\mathbf{m} + \mathbf{n})$

Adjacency Lists

- Array (or list) of vertices requires $\theta(\mathbf{n})$
- Array (or list) of edges requires $\theta(\mathbf{m})$ Since there are n vertices, and m vertices, this is pretty clear.
- Each edge points to it's endpoints requires $\theta(\mathbf{m})$ Since each edge has 2 endpoints, the total storage required for this should be $2 \times m = O(m)$.
- Each vertex points to edges incident on it requires $\theta(\mathbf{m})$ To understand this, realise that for each edge pointing to an endpoint (vertex), there is a corresponding pointer for that vertex pointing to the same edge. There is a one to one correspondence between these two. Hence, this should also require $\theta(m)$ storage.

So the net complexity is $\theta(\mathbf{m} + \mathbf{n})$

What to use?

- Adjacency lists are more space efficient, although adjacency matrices are (slightly) easier to code
- Most real world applications require adjacency lists, because of the massive number of nodes one may deal with in those applications
- Consider the web-graph (graph of the Web pages together with the hypertext links between them) that has approximately 10^{10} nodes (10 billion nodes)
- If we require an 8-byte value for each of the elements in an adjacency matrix representation, that would amount to **80 GB of space just for a single matrix**

What to use in the context of competitive coding?

Adjacency Matrix

- **Pros:** Matrix Representation is easier to implement and follow. Removing an edge takes O(1) time. **Queries like whether there is an edge from vertex u to vertex v are efficient** and can be done O(1).
- **Cons:** Consumes more space $O(n^2)$. Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(n^2)$ time.

Adjacency Lists

- **Pros:** Saves space O(m+n). Useful for sparse graphs.
- **Cons:** Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(n).

So, how do we do it?

The big question here is, how do we represent a graph when we're actually coding stuff?

- In the context of C++, for an **unweighted graph** (without any weights associated with the edges), we usually stick to something simple, like

  ```
  vector<int> graph[MAX_N_VALUE];
  ```
- This is an array of vectors (each vector holds integers).
- Each element in this array corresponds to a node in the graph
- Each element in this array is a vector.
- If we want to add an edge between node 1 and node 2:

  ```
  graph[1].push_back(2);
  graph[2].push_back(1); // only in case of undirected
  ```

Graph Search

Motivation

- Check if a network is connected
- Shortest path : driving directions, routes, etc
- Formulate a plan: A more abstract mentality, where we see a "path" as a sequence of edges results in solutions to problems (Sudoku puzzle, n-queens problem)
- Computing pieces or connected components of a graphs (will be covered later) used in clustering, or figuring out the structure of the web-graph

Generic Graph Search

Goals

- Find everything findable from a given start vertex (refer to whiteboard)
- Do it efficiently: Don't explore anything twice. Expected complexity should then be linear in the number of edges and vertices.

Generic Search on a graph G from start vertex S

We maintain a "visited" boolean array which tells us which vertices have already been explored (works the same for undirected or directed)

```
1: while it is still possible do
2:     Choose and edge (u,v) such that u is explored, and v is
       unexplored
3:     Mark v as explored
4: end while
```

Generic Graph Search

Claim: At the end of the algo, v explored $\iff$ in G, there is a path from S to v

Proof:

Part 1: if v is explored, then there is a path from S to V (Basic Induction on v).

Part 2: if there is a path from S to v, then v is explored.

- Refer to the diagram on the whiteboard

BFS and DFS

What is the ambiguity in our generic method? Why can we have different implementations of it? At any given iteration, you have a set of explored nodes, and a set of unexplored nodes with a set of crossing edges. **Which of these crossing or frontier edges to choose?**

- Refer to the diagram on the whiteboard

BFS and DFS

BFS

- Explore nodes in layers
- Can compute shortest paths (Movie graph)
- Can compute connected components of an undirected graph (not special to breadth first)
- We will do this in linear time O(m+n) using a **FIFO Queue**

DFS

- Explore aggressively, much like exploring a maze
- Does incredible things for directed graphs: topo-sort, connected components
- We will do this in linear time O(m+n) using a **LIFO Stack**

Queue

A queue is a simple extension of the stack data type. Whereas the stack is a LIFO (last-in first-out) data structure the queue is a FIFO (first-in first-out) data structure. What this means is the first thing that you add to a queue will be the first thing that you get when you perform a pop().
There are four main operations on a queue:

- Push Adds an element to the back of the queue
- Pop Removes the front element from the queue
- Front Returns the front element on the queue
- Empty Tests if the queue is empty or not

In C++, this is done with the STL class queue:

```
#include <queue>
std::queue<int> myQueue;
```

BFS

- Explore nodes in layers
- Can compute shortest paths (Movie graph)
- Can compute connected components of an undirected graph (not special to breadth first)
- We will do this in linear time O(m+n) using a **FIFO Queue**

BFS

BFS (Graph G, Start Vertex s)

```
 1: All nodes are initially unexplored.
 2: Mark s as explored
 3: Let Q = queue data structure (FIFO), initialized with s
 4: **while** Q is not empty **do**
 5:     Remove the first node of Q, call it v
 6:     **for** each edge (v,w) **do**
 7:         **if** w is unexplored **then**
 8:             Mark w as explored
 9:             Add w to end of queue
10:         **end if**
11:     **end for**
12:     Mark v as explored
13: **end while**
```

BFS

Running Time Analysis: O($m_s + n_s$) where m_s and n_s are the number of edges reached from s and the number of nodes reached from s respectively

- Inspect the code
- While node works with nodes in the queue. Only nodes reachable from start node are going to be pushed into the queue
- Clearly, nodes get pushed into the queue only once. So that's O(n_s)
- Consider edges: We do constant work PER EDGE. How many times does each edge get looked at?
- Each edge gets looked at twice at most: once from each endpoint. Constant work per edge = O(m_s)

Applications of BFS

Shortest Path: We add some extra code

- Initialize distance(v) $= 0$ for s, ∞ for all other vertices
- When considering edge (v,w): if w is unexplored, we set distance(w) $=$ distance(v) $+ 1$

Let's try it out on the previous diagram.
WE see that distance(v) $=$ i means that v lies in the ith layer.

Applications of BFS

Computing connected components in an undirected graph
Let's consider a sample problem. Consider a n x n 2D Matrix which represents a map of a group of islands. a 1 represents land, whereas a 0 represents water.

Calculate the number of islands in the map ($0 < n < 1000$)
Sample Input:

```
1 1 0 0 0
0 1 0 0 1
1 0 0 1 1
0 0 0 0 0
1 0 1 0 1
```

Output: 5

DFS

- Explore aggressively, much like exploring a maze. Backtrack only when necessary.
- Does incredible things for directed graphs: topo-sort, connected components
- We will do this in linear time O(m+n) using a **LIFO Stack**

Stack

A stack is one of the simplest data structures available. There are four main operations on a stack:

- Push Adds an element to the top of the stack
- Pop Removes the top element from the stack
- Top Returns the top element on the stack
- Empty Tests if the stack is empty or not

In C++, this is done with the STL class stack:

```
#include <stack>
std::stack<int> myStack;
```

DFS

One way to implement DFS would be by simply replacing the Queue in the BFS with a LIFO Stack.

Another more elegant way (and easier way) involves recursion. **All nodes are initially unexplored.**

DFS (Graph G, Start Vertex s

```
1: Mark s as explored
2: for each edge (s, v) do
3:     if v is unexplored then
4:         DFS(G,v)
5:     end if
6: end for
```

DFS

All nodes are initially unexplored.

DFS (Graph G, Start Vertex s

```
1: Mark s as explored
2: for each edge (s, v) do
3:     if v is unexplored then
4:         DFS(G,v)
5:     end if
6: end for
```

Identical running time of O($(m_s + n_s)$) where m_s and n_s are the number of edges reached from s and the number of nodes reached from s respectively. No node is explored more than once. Each edge is looked at at-most twice.

Application of DFS

Topological Ordering of a Directed Acyclic Graph
An ordering of a directed graph so that all the arcs go forward in the ordering.

- We encode the ordering by labels from 1 to n. (One label per vertex). Every directed edge of G goes forward in the ordering.
- For each edge, label of tail is less than label of head.
- Used in scheduling tasks with precedence constraints
- **Graph must be directed and acyclic**

Topological Ordering of a Directed Acyclic Graph

Every directed acyclic graph needs to have atleast some sink vertex. (Suppose this isn't true. If there's no sink vertex, we keep following the outgoing arcs as long as we want. There's a finite number of vertices, and we follow n edges, we find n+1 vertices)

- A topological ordering must end in a sink
- Label that sink node as n. Remove the sink, and recurse on the remainder of the graph.

Why does this work??

Topological Ordering of a Directed Acyclic Graph

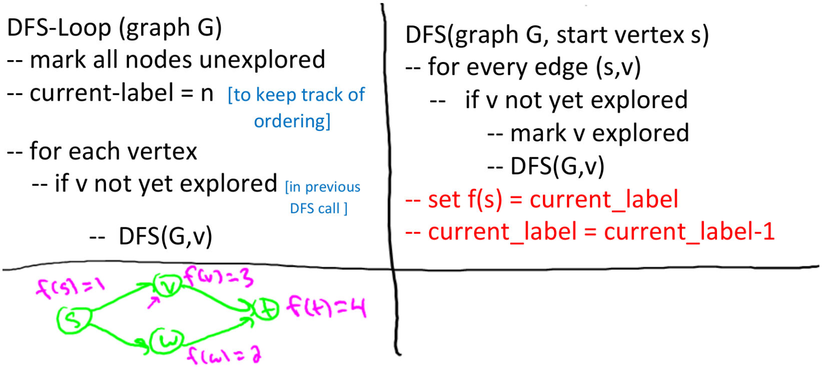
Every directed acyclic graph needs to have atleast some sink vertex. (Suppose this isn't true. If there's no sink vertex, we keep following the outgoing arcs as long as we want. There's a finite number of vertices, and we follow n edges, we find n+1 vertices)

- A topological ordering must end in a sink
- Label that sink node as n. Remove the sink, and recurse on the remainder of the graph.

Why does this work??

We always use sinks. When v is assigned label i, then v is a sink vertex when considering the first i vertices. This means every outgoing edge of v goes to already deleted vertices.

Topological Ordering of a Directed Acyclic Graph



```
DFS-Loop (graph G)
-- mark all nodes unexplored
-- current-label = n  [to keep track of ordering]
-- for each vertex
    -- if v not yet explored [in previous DFS call ]
        -- DFS(G,v)
```

```
DFS(graph G, start vertex s)
-- for every edge (s,v)
    -- if v not yet explored
        -- mark v explored
        -- DFS(G,v)
-- set f(s) = current_label
-- current_label = current_label-1
```

Figure : Toposort using DFS

Topological Ordering of a Directed Acyclic Graph

Linear running time O(m+n)
Constant work per node, each edge is looked at once.

Correctness: We need to show that if u,v is an edge, then $f(u) < f(v)$.

- There is an edge from u to v. Say u is visited by DFS before v.
- Then there will be a recursive call to DFS(G,v) from u. DFS(G,v) finishes before DFS(G,u), so v is assigned a higher label
- Say v is visited by DFS before u.
- There cannot be any path from v to u. DFS invoked from v does not discover u. So, DFS(G,v) is finished even before DFS(G,u) is started. So it's label is larger.

Cycle Detection

Consider a simple undirected graph first.

- An undirected graph has a cycle if and only if a depth-first search (DFS) finds an edge that points to an ancestor vertex (a back edge).
- A back edge is an edge that is from a node to itself (selfloop) or one of its ancestor in the tree produced by DFS.
- For a formal proof, read a textbook like CLRS. The intuition however is simple: when we perform a DFS, and we find a path leading to an ancestor of the current node, then it's clearly a cycle. And if a cycle exists, when we start DFS at a node in the cycle, we will end up with some ancestor node.

Cycle Detection

Consider a simple undirected graph first.

- In the regular DFS function, keep track of the immediate parent
- If any neighbor of the current node leads to an already visited vertex other than the parent, a cycle exists.

Cycle Detection

All nodes are initially marked as unvisited.

DFS (Graph G, Start Vertex s, Parent Node p)

```
1: Mark s as explored
2: for each edge (s, v) where $v \neq p$ do
3:     if v is unexplored then
4:         DFS(G,v,s)
5:     else
6:         Cycle Detected
7:     end if
8: end for
```

Cycle Detection

For a directed graph

- Simply checking visited edges will not do. This is because an edge may be visited earlier, but may not be an ancestor of the current node.
- Only nodes currently in the DFS stack form the ancestors of the current vertex. So, we need to keep track of the nodes in the recursion stack.
- This is also called a three-colour algorithm: Nodes that have been already visited are marked **BLACK**. These edges are also called forward edges or cross edges, since these edges lead to different connected components of the graph.
- Nodes that are currently in the recursion stack (the ancestors) are coloured **GREY**
- Nodes that are unvisited are coloured **WHITE**

Cycle Detection

- We only mark a node visited (BLACK) after we complete DFS from that node (after we've examined all it's neighbors).
- When we start DFS on a node, we mark it as GREY (not visited)
- All other nodes are marked white initially.

DFS (Graph G, Start Vertex s)

```
1: Mark s as GREY
2: for each edge (s, v) do
3:     if v is WHITE then
4:         DFS(G,v)
5:     else if v is GREY then
6:         Cycle Detected
7:     end if
8: end for
9: Mark s as BLACK
```

Important Problems we hope to cover

http://www.spoj.pl/problems/PT07Y/
http://www.spoj.pl/problems/BITMAP/
http://www.spoj.pl/problems/ESCJAILA/
http://www.spoj.pl/problems/PPATH/
http://www.spoj.pl/problems/PT07Z/
http://www.spoj.pl/problems/LABYR1/
http://www.spoj.pl/problems/ONEZERO/
http://www.spoj.pl/problems/BUGLIFE/
http://www.spoj.pl/problems/POUR1/
https://www.codechef.com/problems/DIGJUMP
https://www.codechef.com/problems/TR002/