CSC263 Week 8

Larry Zhang

http://goo.gl/forms/S9yie3597B
Announcements (strike related)

➔ Lectures go as normal
➔ Tutorial this week
  ◆ everyone go to BA3012 (T8, F12, F2, F3)
➔ Problem sets / Assignments are submitted as normal.
  ◆ marking may be slower
➔ Midterm: still being marked
➔ Keep a close eye on announcements
This week’s outline

➔ Graph

➔ BFS
A really, really important ADT that is used to model relationships between objects.

Reference: [http://steve-yegge.blogspot.ca/2008/03/get-that-job-at-google.html](http://steve-yegge.blogspot.ca/2008/03/get-that-job-at-google.html)
Things that can be modelled using graphs

- Web
- Facebook
- Task scheduling
- Maps & GPS
- Compiler (garbage collection)
- OCR (computer vision)
- Database
- Rubik’s cube
- .... (many many other things)
Definition

\[ G = (V, E) \]

Set of **vertices**
e.g., \{a, b, c\}

Set of **edges**
e.g., \{(a, b), (c, a)\}
Flavours of graphs
Each edge is an **unordered** pair

\[(u, v) = (v, u)\]

Each edge is an **ordered** pair

\[(u, v) \neq (v, u)\]
Unweighted

Weighted

10  200

-3
Simple

Non-simple

No multiple edge, no self-loop
Connected

Disconnected
Dense

Sparse
Path

Length of path = number of edges

Read Appendix B.4 for more background on graphs.
Operations on a graph

➔ Add a vertex; remove a vertex
➔ Add an edge; remove an edge
➔ Get **neighbours** (undirected graph)
  ◆ Neighbourhood(u): all v ∈ V such that (u, v) ∈ E
➔ Get **in**-neighbours / **out**-neighbours (directed graph)
➔ **Traversal**: visit every vertex in the graph
Data structures for the graph ADT

→ Adjacency matrix
→ Adjacency list
Adjacency matrix

A $|V| \times |V|$ matrix $A$

Let $V = \{v_1, v_2, \ldots, v_n\}$

$$A[i, j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$
### Adjacency matrix

The adjacency matrix represents the connections between nodes in a graph. The matrix is square, with each row and column corresponding to a node in the graph.

#### Graph representation

- Node 1 is connected to nodes 2 and 3.
- Node 2 is connected to nodes 1 and 3.
- Node 3 is connected to nodes 1 and 2.
- Node 4 has no connections.

#### Adjacency matrix

```
   | 1 | 2 | 3 | 4 |
---|---|---|---|---|
 1 | 0 | 1 | 0 | 1 |
 2 | 0 | 0 | 1 | 0 |
 3 | 1 | 0 | 0 | 0 |
 4 | 0 | 0 | 0 | 0 |
```
Adjacency matrix

How much space does it take?

$$|\mathbf{V}|^2$$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</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>0</td>
</tr>
</tbody>
</table>
The adjacency matrix of an undirected graph is **symmetric**.
Adjacency matrix (undirected graph)

How much space does it take?

$$|V|^2$$
Adjacency list
Adjacency list (directed graph)

Each vertex $v_i$ stores a list $A[i]$ of $v_j$ that satisfies $(v_i, v_j) \in E$
Adjacency list (directed graph)

How much space does it take?

$|V| + |E|$
Adjacency list (undirected graph)
Adjacency list (undirected graph)

How much space does it take?

|V| + 2|E|

| 1 |
| 2 |
| 3 |
| 4 |

| 2 | 4 | 3 |
| 1 | 3 |
| 2 | 1 |
| 1 |
Matrix VS List

In term of space complexity

➔ adjacency matrix is $\Theta(|V|^2)$

➔ adjacency list is $\Theta(|V| + |E|)$

Which one is more space-efficient?

Adjacency list, if $|E| \ll |V|^2$, i.e., the graph is not very dense.
Matrix VS List

Anything that **Matrix** does better than **List**?

Check whether edge \((v_i, v_j)\) is in \(E\)

- **Matrix**: just check if \(A[i, j] = 1\), \(O(1)\)
- **List**: go through list \(A[i]\) see if \(j\) is in there, \(O(\text{length of list})\)
Takeaway

Adjacency **matrix** or adjacency **list**?

Choose the more appropriate one depending on the problem.
CSC263 Week 8

Wednesday / Thursday
Announcements

➔ PS6 posted, due next Tuesday as usual

➔ Drop date: March 8th
Recap

➔ ADT: Graph

➔ Data structures
  ◆ Adjacency matrix
  ◆ Adjacency list

➔ Graph operations
  ◆ Add vertex, remove vertex, ..., edge query, ...
  ◆ Traversal
Graph Traversals

BFS and DFS

They are twins!
Graph traversals

Visiting every vertex once, starting from a given vertex.

The visits can follow different orders, we will learn about the following two ways

➔ **Breadth** First Search (**BFS**)  
➔ **Depth** First Search (**DFS**)
Intuitions of BFS and DFS

Consider a special graph -- a tree
“The knowledge learning tree”

High School

College

PhD

Science

CS

Physics

Geology

AI

DB

String Theory

Black hole

Rocks

Sand

Traversing this graph means learning all these subjects.
The **Breadth-First** ways of learning these subjects

Level by level, finish high school, then all subjects at College level, then finish all subjects in PhD level.
The **Depth-First** way of learning these subjects

➔ Go towards PhD whenever possible; only start learning physics after finishing everything in CS.
Now let’s seriously start studying BFS
Special case: BFS in a tree

Review CSC148:
BFS in a tree (starting from root) is a ___________ traversal.

(level-by-level)

(NOT preorder!)

What ADT did we use for implementing the level-by-level traversal?

Queue!
Special case: BFS in a tree

Output:
```
Output: a b c d e f
```

Queue:
```
Queue: a b c d e f
DQ DQ DQ DQ DQ DQ EMPTY!
```

```
NOT_YET_BFS(root):
Q ← Queue()
Enqueue(Q, root)
while Q not empty:
x ← Dequeue(Q)
print x
for each child c of x:
    Enqueue(Q, c)
```
The real deal: BFS in a Graph

If we just run \texttt{NOT\_YET\_BFS(t)} on the above graph. What problem would we have?

It would want to visit some vertex \texttt{twice} (e.g., \texttt{x}), which shall be \texttt{avoided}!

\texttt{NOT\_YET\_BFS(root)}:

\begin{verbatim}
Q ← Queue()
Enqueue(Q, root)
while Q not empty:
x ← Dequeue(Q)
print x
for each \texttt{neighbor} c of x:
    Enqueue(Q, c)
\end{verbatim}
How avoid visiting a vertex twice

Remember you visited it by labelling it using colours.

→ **White**: “unvisited”
→ **Gray**: “encountered”
→ **Black**: “explored”

Initially all vertices are **white**

Colour a vertex **gray** the first time visiting it

Colour a vertex **black** when all its **neighbours** have been encountered

Avoid visiting **gray** or **black** vertices

In the end, all vertices are **black** (sort-of)
Some other values we want to remember during the traversal...

→ **pi[v]**: the vertex from which v is encountered
  ◆ “I was introduced as *whose* neighbour?”

→ **d[v]**: the distance value
  ◆ the distance from v to the source vertex of the BFS

This **d[v]** is going to be really useful!
Pseudocode: the real BFS

BFS(G=(V, E), s):
1   for all v in V:
2      colour[v] ← white
3      d[v] ← ∞  # Initialize vertices
4      pi[v] ← NIL
5   Q ← Queue()
6   colour[s] ← gray  # start BFS by encountering the source vertex
7   d[s] ← 0  # distance from s to s is 0
8   Enqueue(Q, s)
9   while Q not empty:
10      u ← Dequeue(Q)
11      for each neighbour v of u:
12         if colour[v] = white  # only visit unvisited vertices
13            colour[v] ← gray
14            d[v] ← d[u] + 1  # v is “1-level” farther from s than u
15            pi[v] ← u  # v is introduced as u’s neighbour
16            Enqueue(Q, v)
17      colour[u] ← black  # all neighbours of u have been encountered, therefore u is explored
Let’s run an example!

BFS(G, s)
After initialization

All vertices are **white** and have $d = \infty$
Start by “encountering” the source

Colour the source **gray** and set its $d = 0$, and Enqueue it
Dequeue, explore neighbours

Queue: $s \ s \ r \ w$

The red edge indicates the $\text{pi}[v]$ that got remembered.
Colour black after exploring all neighbours

Queue: DQ

s

r

w

1

∞
Deque, explore neighbours (2)

Queue: DQ s r w v
Dequeue, explore neighbours (3)

Queue:

DQ DQ DQ
s r w v t x
after a few more steps...
BFS done!

Queue: s r w v t x u y
What do we get after doing all these?
First of all, we get to visit every vertex once.
Did you know? The official name of the red edges are called “tree edges”.

This is called the **BFS-tree**, it’s a **tree** that connects all vertices, if the graph is **connected**.
These $d[v]$ values, we said they were going to be really useful.

The value indicates the vertex’s **distance** from the source vertex.

Actually more than that, it’s the **shortest-path distance**, we can prove it.

How about finding **short path** itself? Follow the red edges, $pi[v]$ comes in handy for this.
What if G is disconnected?

The infinite distance value of z indicates that it is **unreachable** from the source vertex.
Runtime analysis!

The total amount of work (use adjacency list):

→ Visit each vertex once
  ◆ Enqueue, Dequeue, change colours, assign d[v], ..., constant work per vertex
  ◆ in total: $O(|V|)$

→ At each vertex, check all its neighbours (all its incident edges)
  ◆ Each edge is checked twice (by the two end vertices)
  ◆ in total: $O(|E|)$

Total runtime: $O(|V| + |E|)$
Summary of BFS

➔ Prefer to explore **breadth** rather than depth

➔ Useful for getting **single-source shortest paths** on **unweighted** graphs

➔ Useful for testing reachability

➔ Runtime $O(|V| + |E|)$ with adjacency list (with adjacency **matrix** it’ll be different)
Next week

DFS

http://goo.gl/forms/S9yie3597B