CSE 373: More on graphs; DFS and BFS

Michael Lee
Wednesday, Feb 14, 2018

Warmup

Warmup:

Discuss with your neighbor:

- Remind your neighbor: what is a *simple graph*?
  A simple graph is a graph that has no self-loops and no parallel edges.
- Suppose we have a *simple, directed graph* with x nodes. What is the maximum number of edges it can have, in terms of x?
  Each vertex can connect to $x - 1$ other vertices, so $x(x - 1)$.
- Now, suppose we have a different simple, undirected graph with y edges. What is the maximum number of vertices it can have, in terms of y?
  Infinite: just keep adding nodes with no edges attached.

Warmup:

Some follow-up questions:

- Suppose we have a *simple, undirected graph* with x nodes. What is the maximum number of edges it can have? What if the graph is not simple?
  If the graph is simple, the max number of edges is exactly half of what it would be if the graph were directed. So, $\dfrac{x(x-1)}{2}$.
  If the graph is not simple, it's infinite: assuming $x > 0$, we can just keep adding more and more self-loops. Note that if $x = 0$, there can't be any edges at all.
- Now, suppose we have a different simple, undirected graph with y edges. What is the maximum number of vertices it can have? What if the graph is not simple?
  Either way, it's still infinite, for the same reasons given previously.

Summary

What did we learn?

- In graphs with no restrictions, number of edges and number of vertices are independent.
- In simple graphs, if we know $|V|$ is some fixed value, we also know $|E| \in \mathcal{O}\left(|V|^2\right)$, for both directed and undirected graphs.

Dense graph

If $|E| \in \Theta\left(|V|^2\right)$, we say the graph is **dense**.

To put it another way, dense graphs have "lots of edges"

Sparse graph

If $|E| \in \mathcal{O}\left(|V|\right)$, we sau the graph is **sparse**.

To put it another way, sparse graphs have "few" edges.

How do we represent graphs in code?

So, how do we actually represent graphs in code?

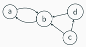
Two common approaches, with different tradeoffs:

- Adjacency matrix
- Adjacency list

Adjacency matrix

Core idea:

- Assign each node a number from 0 to $|V| - 1$
- Create a $|V| \times |V|$ nested array of booleans or ints
- If $(x, y) \in E$, then `nestedArray[x][y] == true`



	a	b	c	d
a				
b				
c				
d				

Adjacency list

What is the worst-case runtime to:

- Get out-edges: $\mathcal{O}(|V|)$
- Get in-edges: $\mathcal{O}(|V|)$
- Decide if an edge exists: $\mathcal{O}(1)$
- Insert an edge: $\mathcal{O}(1)$
- Delete an edge: $\mathcal{O}(1)$

How much space do we use? $\mathcal{O}\left(|V|^2\right)$

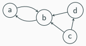
Is this better for sparse or dense graphs? Dense ones

Can we handle self-loops and parallel edges? Self-loops yes, parallel edges, not easily

Adjacency list

Core idea:

- Assign each node a number from 0 to $|V| - 1$
- Create an array of size $|V|$
- Each element in the array stores its out edges in a list or set
- On a higher level: represent as `IDictionary<Vertex, Edges>`.



Adjacency list

We can store edges using either sets or lists. Answer these questions for both.

What is the worst-case runtime to:

- Get out-edges:
- Get in-edges:
- Decide if an edge exists:
- Insert an edge:
- Delete an edge:

How much space do we use?

Is this better for sparse or dense graphs?

Can we handle self-loops and parallel edges?

Which do we pick?

So which do we pick?

Observations:

- Most graphs are sparse
- If we implement adjacency lists using sets, we can get comparable worst-case performance

So by default, pick adjacency lists.

Walks and paths

Walk

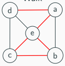
A **walk** is a list of vertices $v_0, v_1, v_2, \dots, v_n$ where if i is some int where $0 \leq i < v_n$, every pair $(v_i, v_{i+1}) \in E$ is true.

More intuitively, a walk is one continous line following the edges.

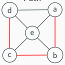
Path

A **path** is a walk that never visits the same vertex twice.

Path or walk?
Walk



Path or walk?
Path



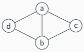
Connected components

Connected graph

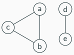
A graph is **connected** if every vertex is connected to every other vertex via some path.

E.g.: if we pick up the graph and shake it, nothing flies off.

Connected or not connected?
Connected



Connected or not connected?
Not connected



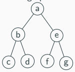
Connected component

A connected component of a graph is any *subgraph* (part of a graph) where all vertices are connected to each other.

Note: A connected graph has only one connected component.

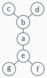
Trees vs graphs

Is this a graph or tree?



Both!

Is this the same thing?



Yes! (If 'a' is the root...)

Tree

A tree is a *connected* and *acyclic* graph.

Rooted tree

A rooted tree is a tree where we call one special node the "root".

Detecting if a graph is connected

Question: How can we tell if a graph is connected or not?

Idea: Let's just find out! Pick a node and see if there's a path to every other node!

Breadth-first search (BFS)

Breadth-first traversal, core idea:

1. Pick some node and "mark" it (or save it in a set, etc...)
2. Examine each neighbor and visit each one (note: save the ones we haven't visited yet in some data structure, like a queue?)
3. Dequeue some node from the data structure. Go to step 1.
4. Keep going until the data structure is empty.

Pseudocode, version 1:

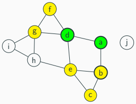
```
search(v):
    visited = empty set
    queue.enqueue(v)

    while (queue is not empty):
        curr = queue.dequeue()
        for (w : v.neighbors()):
            queue.enqueue(w)
```


Breadth-first search (BFS) example

```
search(v):
    queue.enqueue(v)

    while (queue is not empty):
        curr = queue.dequeue()
        for (w : v.neighbors()):
            queue.enqueue(w)
```



Current node: a b d a

Queue: a, b, d, a, b, e, f, g,

What went wrong?

A broken traversal

Problem: We're re-visiting nodes we already visited!

A fix: Keep track of nodes we've already visited in a set!

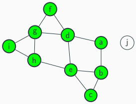
Breadth-first search (BFS) example

```
search(v):
    visited = empty set

    queue.enqueue(v)
    visited.add(curr)

    while (queue is not empty):
        curr = queue.dequeue()

        for (w : v.neighbors()):
            if (w not in visited):
                queue.enqueue(w)
                visited.add(curr)
```



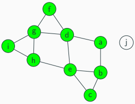
Current node: a b d c e f g h i

Queue: a, b, d, c, e, f, g, h, i,

Visited: a, b, d, c, e, f, g, h, i,

An interesting property...

Note: We visited the nodes in "rings" – maintained a gradually growing "frontier" of nodes.



BFS

Pseudocode

```
search(v):
    visited = empty set

    queue.enqueue(v)
    visited.add(curr)

    while (queue is not empty):
        curr = queue.dequeue()

        for (w : v.neighbors()):
            if (w not in visited):
                queue.enqueue(w)
                visited.add(curr)
```


BFS analysis

Questions:

- What is the worst-case runtime? (Let $|V|$ be the number of vertices, let $|E|$ be the number of edges)
  We visit each vertex once, and each edge once, so $\mathcal{O}(|V| + |E|)$.
- What is the worst-case amount of memory used?
  Whatever the largest "horizon size" is. In the worst case, the horizon will contain $|V| - 1$ nodes, so $\mathcal{O}(|V|)$.

Note: $\mathcal{O}(|V| + |E|)$ is also called "graph linear".

Other applications of BFS

Describe how you would use or modify BFS to solve the following:

- Determine if some graph is also a tree.
- Print out all the elements in a tree level by level.
- Find the shortest path from one node to another.