CSC263 Week 9

Larry Zhang
Announcements

➔ Midterm, class average 62.5% (37.5/60)

➔ PS7 out soon, due next Tuesday

➔ A2 out, due March 31, start early!

➔ Don’t forget to give feedback (especially about the midterm)
   ♦ [http://goo.gl/forms/S9yie3597B](http://goo.gl/forms/S9yie3597B)
Recap

➔ The Graph ADT
- definition and data structures

➔ BFS
- gives us single-source shortest path
- Let $\delta(s, v)$ denote the length of shortest path from $s$ to $v$...
- then after performing a BFS starting from $s$, we have, for all vertices $v$

$$d[v] = \delta(s, v)$$

We can totally prove it.
Idea of the proof

Use contradiction: suppose there exist v s.t. \( d[v] > \delta(s, v) \), let v be the one with the \textbf{minimum} \( \delta(s, v) \).

Then on a shortest path between s and v, pick vertex u which is immediately before v...
then we have \( d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 \)

Must be equal because u is on the shortest path from s to v.

Think about the moment after dequeue u (checking u’s neighbours)
→ if v is white, \( d[v] = d[u] + 1 \) (how BFS works), \textbf{contradiction}!
→ if v is black, \( d[v] \leq d[u] \) (coz v is dequeued before u), \textbf{contradiction}!
→ if v is gray, then it is coloured gray by some other vertex w, then \( d[v] = d[w] + 1 \) and \( d[w] \leq d[u] \), therefore \( d[v] \leq d[u] + 1 \), \textbf{contradiction}!
Depth-First Search
The Depth-First way of learning these subjects

Go towards PhD whenever possible; only start learning physics after finishing everything in CS.
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)

NOT_YET_DFS(root):
    Q ← Stack()
    Push(Q, root)
    while Q not empty:
        x ← Pop(Q)
        print x
        for each child c of x:
            Push(Q, c)

Why they are twins!
DFS in a tree

Output:

```
Output:
```

```
Stack:
```

```
NOT_YET_DFS(root):
```

```
 NOT_YET_DFS(root):
    Q ← Stack()
    Push(Q, root)
    while Q not empty:
        x ← Pop(Q)
        print x
        for each child c of x:
            Push(Q, c)
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
```

```
Stack:
A nicer way to write this code?

The use of stack is basically implementing _recursion_.

```
NOT_YET_DFS(root):
    Q ← Stack()
    Push(Q, root)
    while Q not empty:
        x ← Pop(Q)
        print x
        for each child c of x:
            Push(Q, c)
```

```
NOT_YET_DFS(root):
    print root
    for each child c of x:
        NOT_YET_DFS(c)
```

Exercise: Try this code on the tree in the previous slide.
Avoid visiting a vertex twice, same as BFS

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**
Other values to remember, some are same as BFS

→ **\( \pi[v] \):** the vertex from which \( v \) is encountered
   
   ◆ “I was introduced as **whose** neighbour?”
Other values to remember, different from BFS

➔ There is a clock ticking, incremented whenever someone’s colour is changed

➔ For each vertex v, remember two timestamps

◆ **d[v]**: “discovery time”, when the vertex is first encountered

◆ **f[v]**: “finishing time”, when all the vertex’s neighbours have been visited.

Note: this d[v] is totally different from that distance value d[v] in BFS!
The pseudo-code (incomplete)

DFS_VISIT(G, u):
    colour[u] ← gray
    time ← time + 1
    d[u] ← time
    for each neighbour v of u:
        if colour[v] = white:
            pi[v] ← u
            DFS_VISIT(G, v)
    colour[u] ← black
    time ← time + 1
    f[u] ← time

Why DFS_VISIT instead of DFS?
We will see...

The red part is the same as NOT_YET_DFS
Let’s run an example!  

DFS_VISIT(G, u)
time = 1, encounter the source vertex

\[d=1\]
time = 2, recursive call, level 2
time = 3, recursive call, level 3
time = 4, recursive call, level 4
time = 5, vertex x finished
time = 6, recursion back to level 3, finish y
time = 7, recursive back to level 2, finish v
time = 8, recursion back to level 1, finish u
DFS_VISIT(G, u) done!

What about these two white vertices?

We actually want to visit them (for some reason)
The pseudo-code for visiting everyone

**DFS(G):**

```plaintext
for each v in G.V:
    colour[v] ← white
    f[v] ← d[v] ← ∞
    pi[v] ← NIL

time ← 0
for each v in G.V:
    if colour[v] = white:
        DFS_VISIT(G, v)
```

**Initialization**

**DFS_VISIT(G, u):**

```plaintext
colour[u] ← gray

time ← time + 1
d[u] ← time

for each neighbour v of u:
    if colour[v] = white:
        pi[v] ← u
        DFS_VISIT(G, v)
```

**Make sure NO vertex is left with white colour.**

```plaintext
colour[u] ← black
time ← time + 1
f[u] ← time
```
So, let’s finish this DFS
time = 9, **DFS_VISIT**(G, w)
time = 10
time = 11
time = 12

Diagram:

- Node u with labels 1, 8
- Node v with labels 2, 7
- Node w with labels 9, 12
- Node x with labels 4, 5
- Node y with labels 3, 6
- Node z with labels 10, 11

Connections:
- u to x
- x to y
- y to z
- v to x
- v to y
- w to y
- w to z

Red arrows indicate additional connections or transitions.
DFS(G) done!
Runtime analysis!

The total amount of work (use adjacency list):

- Visit each vertex once
  - constant work per vertex
  - in total: $O(|V|)$
- At each vertex, check all its neighbours (all its incident edges)
  - Each edge is checked once (in a directed graph)
  - in total: $O(|E|)$

Total runtime: $O(|V| + |E|)$

Same as BFS
What do we get from DFS?

➔ Detect whether a graph has a cycle.

◆ That’s why we wanted to visit all vertices -- if you want to be sure whether a graph has a cycle or not, you’d better check everywhere.

◆ Why didn’t we do the similar thing for BFS?

➔ How exactly do we detect a cycle?
CSC263 Week 9

Thursday
Recap: DFS(G) done!

d[v]: discovery time
f[v]: finishing time
How do we use all the info?

We get a DFS forest (a set of disjoint trees)
determine descendant / ancestor relationship in the DFS forest
How to decide whether $y$ is a descendant of $u$ in the DFS forest?

Idea #1: trace back the $\text{pi}[v]$ pointers (the red edges) starting from $y$, see whether you can get to $u$. Worst-case takes $O(n)$ steps.
the “parenthesis structure”

( ( ( ) ) ) ( ) ( ( ) )

➔ Either one pair **contains** the another pair.
➔ Or one pair is **disjoint** from another

( ( ) )

This (overlapping) never happens!
Visualize $d[v]$, $f[v]$ as interval $[d[v], f[v]]$
Now, visualize all the intervals!

What do you see in this?

Parenthesis structure!
The \([d[v], f[v]]\) intervals that we got from DFS follow the parenthesis structure, i.e.,

- Either one interval **contains** another
- Or one is **disjoint** from another

Moreover,

- Iff interval of \(u\) contains interval of \(v\), then \(u\) is an **ancestor** of \(v\) in the DFS forest.
- If interval of \(u\) is disjoint from interval of \(v\), then they are **not** ancestors of each other.
How to decide whether \( y \) is a descendant of \( u \) in the DFS forest?

**Idea #1:** trace back the \( \pi[v] \) pointers (the red edges) starting from \( y \), see whether you can get to \( u \).

Worst-case takes \( O(n) \) steps.

**FORGET ABOUT IT**

**Idea #2:** see if \([d[u], f[u]]\) contains \([d[y], f[y]]\).
Worst-case: 1 step!
We can efficiently check whether a vertex is an ancestor of another vertex in the DFS forest.

so what...
Classifying Edges
4 types of edges in a graph after a DFS

➔ **Tree edge:** an edge in the DFS-forest

➔ **Back edge:** a non-tree edge pointing from a vertex to its **ancestor** in the DFS forest.

➔ **Forward edge:** a non-tree edge pointing from a vertex to its **descendant** in the DFS forest

➔ **Cross edge:** all other edges
Checking edge types

We can efficiently check edge types, because... we can efficiently check whether a vertex is an ancestor / descendant of another vertex using... the parenthesis structure of \([d[v], f[v]]\) intervals!
We can efficiently check edge types after a DFS!

so what...

A graph is **cyclic** if and only if DFS yields a **back edge**.

That’s useful!
A (directed) graph contains a cycle if and only if DFS yields a back edge.
A (directed) graph contains a cycle if and only if DFS yields a back edge

Proof of “if”:
Let the edge be \((u, v)\), then by definition of back edge, \(v\) is an ancestor of \(u\) in the DFS tree, then there is a path from \(v\) to \(u\), i.e., \(v \rightarrow \ldots \rightarrow u\), plus the back edge \(u \rightarrow v\), BOOM! Cycle.

Proof of “only if”:
Let the cycle be..., Let \(v_0\) be the first one that turns gray, when all others in the cycle are white, then \(v_k\) must be a descendant of \(v_0\). (Read “White Path Theorem” in Text)
How about undirected graph?

Should **back** and **forward** edges be the same thing?

→ No, because although the edges are undirected, **neighbour checking** still has a “direction”.

Checking in this direction, so it’s a back edge.
More about undirected graph

After a DFS on an undirected graph, every edge is either a tree edge or a back edge, i.e., no forward edge or cross edge.

If this were a forward edge, it would violate the DFS algorithm (not checking at C but tracing back and check at A).

If this were a cross edge, it violates DFS again (should have checked (A, C) when reached A, but instead wait until C is visited.)
Why do we care about cycles in a graph?

Because cycles have meaningful implication in real applications.
Example: a course prerequisite graph

If the graph has a cycle, all courses in the cycle become impossible to take!
Applications of DFS

➔ Detect cycles in a graph

➔ Topological sort

➔ Strongly connected components
Topological Sort

→ Place the vertices in such an order that all edges are pointing to the right side.

A valid order of getting dressed.
How to do topological sorting

1. Do a **DFS**

2. Order vertices according to their **finishing times** $f[v]$
Strongly connected components

- Subgraphs with strong connectivity (any pair of vertices can reach each other)
Summary of DFS

➔ It’s the twin of BFS (Queue vs Stack)
➔ Keeps two timestamps: $d[v]$ and $f[v]$
➔ Has same runtime as BFS
➔ Does NOT give us shortest-path
➔ Give us cycle detection (back edge)
➔ For real problems, choose BFS and DFS wisely.
Next week

➔ Minimum Spanning Tree

http://goo.gl/forms/S9yie3597B