Lecture 9

Graphs, BFS and DFS
Announcements!

- HW4 due **MONDAY** (Notice the change)
  - But you might want to get it in on Friday anyway, so that you can study for the….
- **MIDTERM** Wednesday May 10.
  - In this room, during class (3-4:20)
  - Please show up
  - Any material through Wednesday 5/3 is fair game
  - You may bring one double-sided letter-size page of notes, that you have prepared yourself.
Thanks for filling out feedback 2!

• **Sorry** about typos on homework!
  • We have changed our review process to be more stringent.

• I still need to work on the pacing of my lectures
  • **Less review at the beginning, go slower over technical stuff.**
  • I **do** want to take time to cover the important basics carefully, since there are many different backgrounds in this class.
  • I view the stuff at the end of the lecture as less important.
  • But I will try to figure out a more satisfying way to present things.

• The point of lecture? (In my view)
  • High-level overview of the big conceptual ideas.
  • When it comes to proofs:
    • My intent is to emphasize the important structure/outline of the proofs, rather than technical details.
    • (But obviously this is not an excuse for typos!)

• To get all the details, do the suggested reading
  • Before or after class, depending on how you learn best.
Outline

• Part 0: Graphs and terminology

• Part 1: Depth-first search
  • Application: topological sorting
  • Application: in-order traversal of BSTs

• Part 2: Breadth-first search
  • Application: shortest paths
  • Application (if time): is a graph bipartite?
Part 0: Graphs
Graphs

Graph of the internet
(circa 1999...it’s a lot bigger now...)
Graphs

Citation graph of literary theory academic papers
Graphs

Theoretical Computer Science academic communities

Example from DBLP:
Communities within the co-authors of Christos H. Papadimitriou
Graphs

Game of Thrones Character Interaction Network
Graphs

jetblue flights
Graphs

Complexity Zoo containment graph
Graphs

The bilateral flows between 196 countries are estimated from sequential stock tables (see overleaf for details). They are comparable across countries and capture the number of people who changed their country of residence between mid-2005 and mid-2010.

The circular plot shows the estimates of directional flows between the 50 countries that send and/or receive at least 0.5% of the world’s migrants in 2005-10. Tick marks indicate gross migration (in + out) in 100,000s.

Immigration flows
World trade in fresh potatoes, flows over 0.1 m US$ average 2005-2009
Graphs

Soybeans

Water
Graphs

Graphical models
What eats what in the Atlantic ocean?
Graphs

Neural connections in the brain
There are a lot of graphs.

We want to answer questions about them.
  • Efficient routing?
  • Community detection/clustering?
  • An ordering that respects dependencies?

This is what we’ll do for the next several lectures.
Undirected Graphs

• Has vertices and edges
  • $V$ is the set of vertices
  • $E$ is the set of edges
  • Formally, a graph is $G = (V,E)$

• Example
  • $V = \{1,2,3,4\}$
  • $E = \{\{1,3\}, \{2,4\}, \{3,4\}, \{2,3\}\}$

• The degree of vertex 4 is 2.
  • There are 2 edges coming out
  • Vertex 4’s neighbors are 2 and 3
Directed Graphs

• Has vert\(\text{ices}\) and edges
  • \(V\) is the set of vertices
  • \(E\) is the set of \textbf{DIRECTED} edges
  • Formally, a graph is \(G = (V,E)\)

• Example
  • \(V = \{1,2,3,4\}\)
  • \(E = \{(1,3), (2,4), (3,4), (4,3), (3,2)\}\)

  • The in-degree of vertex 4 is 1.
  • The out-degree of vertex 4 is 1.
  • Vertex 4’s incoming neighbors are 2
  • Vertex 4’s outgoing neighbor is 3.
How do we represent graphs?

- Option 1: adjacency matrix

\[
\begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix}
\]
How do we represent graphs?

- Option 1: adjacency matrix

\[
\begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix}
\]
How do we represent graphs?

• Option 1: adjacency matrix

\[
\begin{array}{cccc}
\text{Source} & 1 & 2 & 3 & 4 \\
1 & 0 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 1 \\
3 & 0 & 1 & 0 & 1 \\
4 & 0 & 0 & 1 & 0 \\
\end{array}
\]
How do we represent graphs?

• Option 2: linked lists.

4’s neighbors are 2 and 3

How would you modify this for directed graphs?
In either case

- May think of vertices storing other information
  - Attributes (name, IP address, ...)
  - helper info for algorithms that we will perform on the graph

- We will want to be able to do the following ops:
  - Edge Membership: Is edge \( e \) in \( E \)?
  - Neighbor Query: What are the neighbors of vertex \( v \)?
## Trade-offs

Say there are $n$ vertices and $m$ edges.

### Edge membership
Is $e = \{v,w\}$ in $E$?

<table>
<thead>
<tr>
<th></th>
<th>$O(1)$</th>
<th>$O(\deg(v))$ or $O(\deg(w))$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(n^2)$</td>
<td></td>
</tr>
</tbody>
</table>

### Neighbor query
Give me $v$’s neighbors.

<table>
<thead>
<tr>
<th></th>
<th>$O(n)$</th>
<th>$O(\deg(v))$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n + m)$</td>
</tr>
</tbody>
</table>

We'll assume this representation for the rest of the class.

Generally better for **sparse** graphs.
Part 1: Depth-first search
How do we explore a graph?

At each node, you can get a list of neighbors, and choose to go there if you want.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

start

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet
Been there, haven’t explored all the paths out.
Been there, have explored all the paths out.

start
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet
Been there, haven’t explored all the paths out.
Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

- Not been there yet
- Been there, haven’t explored all the paths out.
- Been there, have explored all the paths out.
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.

start
Depth First Search
Exploring a labyrinth with chalk and a piece of string

Not been there yet
Been there, haven’t explored all the paths out.
Been there, have explored all the paths out.

start
Depth First Search

Exploring a labyrinth with chalk and a piece of string

Not been there yet

Been there, haven’t explored all the paths out.

Been there, have explored all the paths out.

Labyrinth: EXPLORED!
Depth First Search
Exploring a labyrinth with pseudocode

• **DFS**(w, currentTime):
  • w.entryTime = currentTime
  • currentTime ++
  • Mark w as **in progress**.
  • **for** v in w.neighbors:
    • **if** v is **unvisited**:
      • currentTime = **DFS**(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
    • Mark w as **all done**
  • **return** currentTime

Each node is going to keep track of whether it’s unvisited, in progress, or all done.

We’ll also keep track of the time at which we started and finished with that node.
Depth First Search
Exploring a labyrinth with pseudocode

• Each vertex keeps track of whether it is:
  • Unvisited
  • In progress
  • All done

• Each vertex will also keep track of:
  • The time we first enter it.
  • The time we finish with it and mark it all done.

You might have seen other ways to implement DFS than what we are about to go through. This way has more bookkeeping, but more intuition – also, the bookkeeping will be useful later!
Depth First Search

**DFS**\( (w, \text{currentTime}) \):
- \( w.\text{entryTime} = \text{currentTime} \)
- \( \text{currentTime} \) ++
- Mark \( w \) as **in progress**.
- **for** \( v \) in \( w.\text{neighbors} \):
  - **if** \( v \) is **unvisited**:
    - \( \text{currentTime} = \text{DFS}(v, \text{currentTime}) \)
    - \( \text{currentTime} \) ++
  - \( w.\text{finishTime} = \text{currentTime} \)
- Mark \( w \) as **all done**
- **return** \( \text{currentTime} \)

current\( \text{Time} = 0 \)
Depth First Search

currentTime = 1

• DFS(w, currentTime):
  • w.entryTime = currentTime
  • currentTime ++
  • Mark w as in progress.
  • for v in w.neighbors:
    • if v is unvisited:
      • currentTime = DFS(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
  • Mark w as all done
  • return currentTime
Depth First Search

currentTime = 1

\[ \text{DFS}(w, \text{currentTime}) : \]
\[ \text{w.entryTime} = \text{currentTime} \]
\[ \text{currentTime} \text{ ++} \]
\[ \text{Mark w as in progress.} \]
\[ \text{for v in w.neighbors:} \]
\[ \quad \text{if v is unvisited:} \]
\[ \quad \quad \text{currentTime} = \text{DFS}(v, \text{currentTime}) \]
\[ \quad \quad \text{currentTime} \text{ ++} \]
\[ \quad \text{w.finishTime} = \text{currentTime} \]
\[ \quad \text{Mark w as all done} \]
\[ \text{return currentTime} \]
Depth First Search

currentTime = 2

- DFS(w, currentTime):
  - w.entryTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - for v in w.neighbors:
    - if v is unvisited:
      - currentTime = DFS(v, currentTime)
      - currentTime ++
    - w.finishTime = currentTime
  - Mark w as all done
  - return currentTime
**Depth First Search**

\[ \text{currentTime} = 20 \]

- **DFS** \((w, \text{currentTime})\):
  - \(w.\text{entryTime} = \text{currentTime}\)
  - \(\text{currentTime}++\)
  - Mark \(w\) as **in progress**.
  - **for** \(v\) in \(w.\text{neighbors}\):
    - **if** \(v\) is **unvisited**:
      - \(\text{currentTime} = \text{DFS}(v, \text{currentTime})\)
      - \(\text{currentTime}++\)
  - \(w.\text{finishTime} = \text{currentTime}\)
  - Mark \(w\) as **all done**
  - **return** \(\text{currentTime}\)
Depth First Search

currentTime = 21

• **DFS**(*w, currentTime*):
  • *w*.startTime = currentTime
  • currentTime ++
  • Mark *w* as **in progress**.
  • for *v* in *w*.neighbors:
    • if *v* is **unvisited**:
      • currentTime = **DFS**(v, currentTime)
      • currentTime ++
    • *w*.finishTime = currentTime
  • Mark *w* as **all done**
  • return currentTime

Start: 0

End: 21

Takes until currentTime = 20

unvisited

in progress

all done
Depth First Search

currentTime = 21

• **DFS**(w, currentTime):
  • w.startTime = currentTime
  • currentTime ++
  • Mark w as *in progress*.
  • for v in w.neighbors:
    • if v is *unvisited*:
      • currentTime = DFS(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
  • Mark w as *all done*
  • return currentTime

Start: 0
End: 21

Takes until currentTime = 20
DFS finds all the nodes reachable from the starting point.

One application: finding connected components.

In an undirected graph, this is called a **connected component**.
Why is it called depth-first?

- We are implicitly building a tree:

- And first we go as deep as we can.

Call this the “DFS tree”
Running time
To explore just the connected component we started in

• We look at each edge only once.
• And basically don’t do anything else.
• So...

\[ O(m) \]

• (Assuming we are using the linked-list representation)
Running time
To explore the whole thing

• Explore the connected components one-by-one.
• This takes time

$O(n + m)$
You check:

DFS works fine on directed graphs too!

Only walk to C, not to B.

Ollie the over-achieving ostrich
Application: topological sorting

- Example: package dependency graph
- Question: in what order should I install packages?

Suppose the dependency graph has no cycles: it is a Directed Acyclic Graph (DAG)
Can’t always eyeball it.
Application: topological sorting

- Example: package dependency graph
- Question: in what order should I install packages?
Let’s do DFS

Observations:
• The start times don’t seem that useful.
• But the packages we should include earlier have larger finish times.
This is not an accident

**Claim:** In general, we’ll always have:

Suppose the underlying graph has no cycles.

To understand why, let’s go back to that DFS tree.
A more general statement
(this holds even if there are cycles)
This is called the “parentheses theorem” in CLRS

• If \( v \) is a descendent of \( w \) in this tree:

\[
\begin{align*}
\text{w.start} & \quad \text{v.start} & \quad \text{v.finish} & \quad \text{w.finish} \\
\text{timeline} & \quad \quad & \quad & \\
\end{align*}
\]

• If \( w \) is a descendent of \( v \) in this tree:

\[
\begin{align*}
\text{v.start} & \quad \text{w.start} & \quad \text{w.finish} & \quad \text{v.finish} \\
\text{timeline} & \quad \quad & \quad & \\
\end{align*}
\]

• If neither are descendents of each other:

\[
\begin{align*}
\text{v.start} & \quad \text{v.finish} & \quad \text{w.start} & \quad \text{w.finish} \\
\text{timeline} & \quad \quad & \quad & \\
\end{align*}
\]

(or the other way around)

(check this statement carefully!)
So to prove this ->

- Since the graph has no cycles, B must be a descendant of A in that tree.
  - All edges go down the tree.

- Then

  - aka, $B.\text{finishTime} < A.\text{finishTime}$.
Back to this problem

- Example: package dependency graph
- Question: in what order should I install packages?
In reverse order of finishing time

• Do DFS
• Maintain a list of packages, in the order you want to install them.
• When you mark a vertex as all done, put it at the beginning of the list.
What did we just learn?

- DFS can help you solve the **TOPOLOGICAL SORTING PROBLEM**
  - That’s the fancy name for the problem of finding an ordering that respects all the dependencies

- Thinking about the DFS tree is helpful.
Example:

```
Start: 0
```

Diagram:

- Node A: Start point
- Node B: Connected to A and C
- Node C: Connected to B and D
- Node D: Connected to C

Colors:
- Unvisited
- In progress
- All done
Example
Example
Example

A
Start:0

B
Start:3

C
Start:1

D
Start:2

Unvisited

In progress

All done
Example

![Diagram with nodes A, B, C, D and arrows indicating start and leave times.]

- Node A: Start:0
- Node B: Start:3, Leave:4
- Node C: Start:1
- Node D: Start:2

Legend:
- Unvisited
- In progress
- All done
Example

- A: Start:0, Leave:5
- B: Start:3, Leave:4
- C: Start:1, Leave:6
- D: Start:2, Leave:5

- Unvisited
- In progress
- All done
Example

Do them in this order:

A  C  D  B
Another use of DFS

• In-order enumeration of binary search trees

Given a binary search tree, output all the nodes in order.

Instead of outputting a node when you are done with it, output it when you are done with the left child and before you begin the right child.
Part 2: breadth-first search
How do we explore a graph?

If we can fly
Breadth-First Search
Exploring the world with a bird’s eye view

start

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
Exploring the world with a bird’s-eye view

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
Exploring the world with a bird’s-eye view

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
Exploring the world with a bird’s-eye view

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
Exploring the world with a bird’s-eye view

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps

World: EXPLORERD!
Breadth-First Search
Exploring the world with pseudocode

• Set $L_i = \{\}$ for $i=1,\ldots,n$
• $L_0 = \{w\}$, where $w$ is the start node
• For $i = 0, \ldots, n-1$:
  • For $u$ in $L_i$:
    • For each $v$ which is a neighbor of $u$:
      • If $u$ isn’t yet visited:
        • mark $u$ as visited, and put it in $L_{i+1}$

$L_i$ is the set of nodes we can reach in $i$ steps from $w$

Go through all the nodes in $L_i$ and add their unvisited neighbors to $L_{i+1}$
BFS also finds all the nodes reachable from the starting point.

It is also a good way to find all the connected components.
Running time
To explore the whole thing

• Explore the connected components one-by-one.
• Same argument as DFS: running time is

\[ O(n + m) \]

Verify these!

• Like DFS, BFS also works fine on directed graphs.

Ollie the over-achieving ostrich
Why is it called breadth-first?

• We are implicitly building a tree:

YOINK!

• And first we go as broadly as we can.

Call this the “BFS tree”
Application: shortest path

• How long is the shortest path between w and v?
Application: shortest path

• How long is the shortest path between w and v?

It’s three!
To find the distance between \textit{w} and all other vertices \textit{v}

- **Do a DFS starting at \textit{w}**

- **For all \textit{v} in \textit{L}_i (the i’th level of the BFS tree)**
  - The shortest path between \textit{w} and \textit{v} has length \textit{i}
  - A shortest path between \textit{w} and \textit{v} is given by the path in the BFS tree.

- **If we never found \textit{v}, the distance is infinite.**
Proof idea

• Suppose by **induction** it’s true for vertices in $L_0, L_1, L_2$
  • For all $i < 3$, the vertices in $L_i$ have distance $i$ from $v$.

• **Want to show**: it’s true for vertices of distance 3 also.
  • aka, the shortest path between $w$ and $v$ has length 3.

• Well, it has distance at most 3
  • Since we just found a path of length 3

• And it has distance at least 3
  • Since if it had distance $i < 3$, it would have been in $L_i$. 

Just the idea...see CLRS for details!
What did we just learn?

• The BFS tree is useful for computing distances between pairs of vertices.

• We can find the shortest path between $u$ and $v$ in time $O(m)$.

The BFS tree is also helpful for:

• Testing if a graph is bipartite or not.
Application: testing if a graph is bipartite

- Bipartite means it looks like this:

Can color the vertices red and orange so that there are no edges between any same-colored vertices

**Example:**
- are students
- are classes
- if the student is enrolled in the class
Is this graph bipartite?
How about this one?
How about this one?
Solution using BFS

• Color the levels of the BFS tree in alternating colors.

• If you ever color a node so that you never color two connected nodes the same, then it is bipartite.

• Otherwise, it’s not.
Breadth-First Search
For testing bipartite-ness

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps

start
Breadth-First Search
For testing bipartite-ness

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
For testing bipartite-ness

- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

start

Not been there yet
Breadth-First Search
For testing bipartite-ness

start

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
For testing bipartite-ness

![Diagram](image)

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

**CLEARLY BIPARTITE!**
Breadth-First Search
For testing bipartite-ness

start

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps
Breadth-First Search
For testing bipartite-ness

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
For testing bipartite-ness

- Not been there yet
- Can reach there in zero steps
- Can reach there in one step
- Can reach there in two steps
- Can reach there in three steps

start
Breadth-First Search
For testing bipartite-ness

Not been there yet
Can reach there in zero steps
Can reach there in one step
Can reach there in two steps
Can reach there in three steps
Breadth-First Search
For testing bipartite-ness

WHOA NOT BIPARTITE!
Hang on now.

• Just because **this** coloring doesn’t work, why does that mean that there is no coloring that works?

I can come up with plenty of bad colorings on this legitimately bipartite graph...

Plucky the pedantic penguin
Some proof required

- If BFS colors two neighbors the same color, then it’s found an **cycle of odd length** in the graph.

There must be an even number of these edges

This one extra makes it odd

Ollie the over-achieving ostrich

Make this proof sketch formal!
Some proof required

• If BFS colors two neighbors the same color, then it’s found an **cycle of odd length** in the graph.
• So the graph has an **odd cycle** as a **subgraph**.
• But you can **never** color an odd cycle with two colors so that no two neighbors have the same color.
  • [Fun exercise!]

• So you can’t legitimately color the whole graph either.
• **Thus it’s not bipartite.**
What did we just learn?

BFS can be used to detect bipartite-ness in time $O(n + m)$. 
Recap

- **Depth-first search**
  - Useful for topological sorting
  - Also in-order traversals of BSTs

- **Breadth-first search**
  - Useful for finding shortest paths
  - Also for testing bipartiteness

- **Both DFS, BFS:**
  - Useful for exploring graphs, finding connected components, etc
Still open (next few classes)

• We can now find components in undirected graphs...
  • What if we want to find strongly connected components in directed graphs?

• How can we find shortest paths in weighted graphs?

To be continued...