Roadmap

1st class
Divide and conquer

Asymptotic Analysis
Recurrences
Randomized Algs

Sorting

Greedy Algs
Dynamic Programming

Graphs!

Data structures

Longest, Shortest, Max and Min...

The Future!

The Future!
Lecture 8
Graphs, BFS and DFS
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
Graphs

Complexity Zoo containment graph
Graphs
Graphs

Immigration flows
Graphs

Potato trade

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

Soybeans

Water
Graphs

Graphical models
Graphs

What eats what in the Atlantic ocean?
Graphs

Neural connections in the brain
Graphs

• There are a lot of graphs.

• We want to answer questions about them.
  • Efficient routing?
  • Community detection/clustering?
    • Computing Bacon numbers
    • Signing up for classes without violating pre-req constraints
    • How to distribute fish in tanks so that none of them will fight.

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

• Have **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

• Have **vertices** and **edges**
  - V is the set of vertices
  - E is the set of **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 2.
• The **out-degree** of vertex 4 is 1.
• Vertex 4’s **incoming neighbors** are 2,3
• 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

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

• Option 2: linked lists.

How would you modify this for directed graphs?

1’s neighbors are 3 and 4

4’s neighbors are 2 and 3

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

• Vertices can store other information
  • Attributes (name, IP address, ...)
  • helper info for algorithms that we will perform on the graph

• Want to be able to do the following operations:
  • **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 $E$?

- $O(1)$
- $O(\text{deg}(v))$ or $O(\text{deg}(w))$

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

- $O(n)$
- $O(\text{deg}(v))$

### Space requirements

- $O(n^2)$
- $O(n + m)$

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

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.

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.
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
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

- Orange circles: Been there, haven’t explored all the paths out.
- Green circles: Been there, have explored all the paths out.
- Blue lines: Connections between rooms.
- Red line: Path taken so far.
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.
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

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

• 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**\((w, \ current\Time)\):
  - \(w.\startTime = \ current\Time\)
  - \(\ current\Time \ +=\)
  - Mark \(w\) as **in progress**.
  - **for** \(v\) in \(w.\neighbors\):
    - **if** \(v\) is **unvisited**:
      - \(\ current\Time\)
      - \(\ current\Time \ +=\)
      - \(w.\finish\Time = \ current\Time\)
      - Mark \(w\) as **all done**
    - **return** \(\ current\Time\)
Depth First Search

currentTime = 0

• **DFS**(w, currentTime):

  • w.startTime = currentTime
  • currentTime ++
  • for v in w.neighbors:
    • currentTime = DFS(v, currentTime)
    • currentTime ++
  • w.finishTime = currentTime
  • return currentTime
Depth First Search

currentTime = 1

• **DFS**(w, currentTime):
  • w.startTime = currentTime
  • currentTime ++
  • Mark w as *in progress*.

Start: 0

- unvisited
- in progress
- all done
Depth First Search

```
currentTime = 1

• **DFS**(w, currentTime):
  • w.startTime = currentTime
  • currentTime ++
  • Mark w as **in progress**.
  • **for** v in w.neighbors:
    • **if** v is **unvisited**:
```
Depth First Search

• **DFS**\( (w, \text{currentTime}) \):
  • \( w.\text{startTime} = \text{currentTime} \)
  • \( \text{currentTime} ++ \)
  • Mark \( w \) as **in progress**.
  • **for** \( v \) in \( w.\text{neighbors} \):
    • **if** \( v \) is **unvisited**:
      • \( \text{currentTime} = \text{DFS}(v, \text{currentTime}) \)
Depth First Search

\[
\text{DFS}(w, \text{currentTime}):
\]
- \( w\text{.startTime} = \text{currentTime} \)
- \( \text{currentTime}++ \)
- Mark \( w \) as \textbf{in progress}.
- \( \text{for } v \text{ in } w\text{.neighbors} : \)
  - \( \text{if } v \text{ is } \textbf{unvisited} : \)
    - \( \text{currentTime} = \text{DFS}(v, \text{currentTime}) \)
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 ++

Takes until currentTime = 20
Depth First Search

 currentTime = 22

• 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 ++

Start:0

Start: 22

Start: 1
End: 21

Takes until
currentTime = 20

unvisited

in progress

all done
Depth First Search

\[ \text{DFS}(w, \text{currentTime}) : \]
- \( w.\text{startTime} = \text{currentTime} \)
- \( \text{currentTime}++ \)
- Mark \( w \) as \textit{in progress}.
- \textbf{for} \( v \) in \( w.\text{neighbors} \):
  - \textbf{if} \( v \) is \textit{unvisited}:
    - \( \text{currentTime} = \text{DFS}(v, \text{currentTime}) \)
    - \( \text{currentTime}++ \)
Depth First Search

• **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
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.
To explore the whole graph

• Do it repeatedly!

code

start

start
Why is it called depth-first?

- We are implicitly building a tree:

  Call this the “DFS tree”

- And first we go as deep as we can.
Running time

- 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)

- Actually, if we want to explore entire graph, it’s $O(m+n)$

Here $m=0$ but it still takes time $O(n)$ to explore the graph.
You check:

DFS works fine on directed graphs too!

Only walk to C, not to B.

Siggi the studious stork
Application: topological sorting

• 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.
Let’s do DFS

... and now what?

Think-Pair-Share!
Finish times seem useful

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.
So to prove this ->

- **Case 1**: B is a descendant of A in the DFS tree.

- Then

  - B.startTime < A.startTime
  - B.finishTime < A.finishTime
  - aka, B.finishTime < A.finishTime.

Suppose the underlying graph has no cycles.
So to prove this ->

If A -> B
Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles

• **Case 2**: B is a **NOT** descendant of A in the DFS tree.
  
  • Then we must have explored B before A.
    
    • Otherwise we would have gotten to B from A, and B would have been a descendant of A in the DFS tree.

• Then

  • aka, B.finishTime < A.finishTime.
Back to this problem

• Question: in what order should I install packages?

Suppose the dependency graph has no cycles: it is a **Directed Acyclic Graph (DAG)**
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.

- dpkg
- coreutils
- tar
- libbz2
- libselinux1
- multiarch_support
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.
Application: Binary search tree -> Sorted array

- 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.
Everything we did before graphs

**Sorting:**
- $\Theta(n \log n)$ time
  - MergeSort (Lecture 2)
  - QuickSort (Lecture 4)

- Can’t do better
  - Comparison model lower bound (Lecture 5)

- $\Theta(n)$ time!!
  - RadixSort (Lecture 5)

**INSERT/DELETE/SEARCH:**
- $\Theta(\log n)$ time
  - Red-Black Trees (Lecture 6)

- Can’t do better
  - Comparison model lower bound (Midterm – bonus Q)

- $\Theta(1)$ time!!
  - Hash functions (Lecture 6)
Application: Binary search tree

Lemma: In the comparisons model, can't INSERT into a binary search tree faster than $\Theta(\log n)$.

Proof: Assume by contradiction that we can. Then we can construct an $n$-node binary search tree faster than $\Theta(n \log n)$. Use DFS to convert to a sorted array (this takes $O(n)$).

$\Rightarrow$ a contradiction to $\Omega(n \log n)$ for sorting!
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 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

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

start
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: explored!
Breadth-First Search
Exploring the world with pseudocode

• Set $L_i = []$ for $i=1,...,n$
• $L_0 = \{w\}$, where $w$ is the start node
• For $i = 0, ..., n-1$:
  • For $u$ in $L_i$:
    • For each $v$ which is a neighbor of $u$:
      • If $v$ isn’t yet visited:
        • mark $v$ 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.
BFS: Running time?

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

$L_i$ is the set of nodes we can reach in $i$ steps from $w$.
Running time
To explore the whole thing

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

\[ O(n + m) \]

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

Verify these!
Why is it called breadth-first?

• We are implicitly building a tree:

• And **first** we go as **broadly** as we can.
Application: Bacon numbers

• What is Samuel L. Jackson’s Bacon number?

(Answer: 2)
Oliver Sacks has Bacon number 3

“Bacon number” = degrees of separation from Kevin Bacon
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 \( w \) and all other vertices \( v \):

- Do a BFS starting at \( w \).
- For all \( v \) in \( L_i \):
  - The shortest path between \( w \) and \( v \) has length \( i \).
  - A shortest path between \( w \) and \( v \) is given by the path in the BFS tree.
- If we never found \( v \), the distance is infinite.

The distance between two vertices is the length of the shortest path between them. Call this the “BFS tree.”

Gauss has no Bacon number.
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: fish in fish tanks

• Some pairs of species will fight if put in the same tank.
• You only have two tanks.
• Connected fish will fight.
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 in tank A
- are in tank B
- if the fish fight

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?
This one?
Solution using BFS

- Color the levels of the BFS tree in alternating colors.
- If you never color two connected nodes the same color, 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
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
Breadth-First Search
For testing bipartite-ness
Breadth-First Search
For testing bipartite-ness

CLEARLY BIPARTITE!
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
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 bipartiteness

- 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.

[Diagram showing a graph with vertices A, B, C, D, E, F, G and edges connecting them.

Red edges indicate an odd cycle.

A hand is pointing to the odd cycle with a note: "This one extra makes it odd."}

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 $O(n + m)$. 
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?
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?

• What is Samuel L. Jackson’s Erdos number?
  • (Or, what if I want everyone’s everyone-else number?)
Next Time

• Strongly Connected Components

**Before** Next Time

• Sections are back – *woohoo!*