Lecture 9

Graphs, BFS and DFS
Announcements!

• HW3 due Friday
  • Follow the Piazza HW3 post for updates – we fixed some typos and also added a hint.

• HW4 released Friday

• I goofed up on the slides for Monday’s lecture – see website for updated slides.
  • Sorry!
Prologue

• Real quick recap of hash functions!
• Go to section this week for a longer recap!
Hash tables

• U is the **universe**. It has size M.
• As **hash table** stores items of U in n buckets.
• A hash table comes with a **hash function** \( h: U \to \{1, \ldots, n\} \) which says what element goes in what bucket.
Hash families

• A hash family is a collection of hash functions.
• For example:
  • $H = \{ \text{all of the functions } h:U \to \{1,\ldots,n\} \}$
  • $H = \{ \text{least-significant-digit, most-significant-digit} \}$
• We can build a hash table by drawing a random function uniformly at random from a hash family and using that hash function.
Example

1. An adversary (who knows H) chooses any $n$ items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

2. You, the algorithm, chooses a random hash function $h: U \rightarrow \{0, \ldots, 9\}$. Choose it randomly from $H$.

3. HASH IT OUT #hashpuns

$I$ picked $h_1$
Example

1. An adversary (who knows H) chooses any $n$ items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

2. You, the algorithm, chooses a **random** hash function $h: U \rightarrow \{0, \ldots, 9\}$. Choose it randomly from $H$.

3. **HASH IT OUT** #hashpuns

$h_0 = \text{Most}\_\text{significant}\_\text{digit}$

$h_1 = \text{Least}\_\text{significant}\_\text{digit}$

$H = \{h_0, h_1\}$

I picked $h_1$
Example

1. An adversary (who knows H) chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

2. You, the algorithm, chooses a random hash function $h: U \to \{0, \ldots, 9\}$. Choose it randomly from H.

3. HASH IT OUT #hashpuns

I picked $h_0$

That’s bad! It takes a long time to search/insert/delete!
Universal hash family

• H is a **universal hash family** if, when h is chosen uniformly at random from H,

\[
P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}
\]

for all \( u_i, u_j \in U \) with \( u_i \neq u_j \),

• In English: For any pair of items in the universe, the probability that they end up in the same bucket is small.

• We came up with this definition because it implied that (in expectation) not too many items land in any given bucket.

• There exist universal hash families of size \( O(M^2) \).
Conclusion

• We can build a hash table that:
  • supports INSERT/DELETE/SEARCH in $O(1)$ expected time
  • requires $O(n \log(M))$ bits of space.
    • $O(n)$ buckets
    • $O(n)$ items with $\log(M)$ bits per item
    • $O(\log(M))$ bits to store the hash function
Questions about hash functions?

• Section this week
• Office hours
• CLRS
• (Optional) lecture notes posted on website
Roadmap

- Sorting
  - Longest, Shortest, Max and Min...
  - Randomized Algs
  - Asymptotic Analysis
  - Recurrences

- Data structures

- Dynamic Programming
- Greedy Algs

- Graphs!

- MIDTERM

- The Future!

1st class: Divide and conquer

5 lectures

2 lectures

9 lectures

1 lecture

More detailed schedule on the website!
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
Graphs

Game of Thrones Character Interaction Network
Graphs

jetblue flights
Graphs

Complexity Zoo
containment graph
Graphs

debian dependency (sub)graph
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?
  • From pre-lecture exercise:
    • 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

- 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 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

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

• Option 1: adjacency matrix

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>3</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>4</td>
<td>0 1 1 0</td>
</tr>
</tbody>
</table>
How do we represent graphs?

• Option 2: linked lists.

1 2 3 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.

<table>
<thead>
<tr>
<th>Edge membership</th>
<th>Neighbor query</th>
<th>Space requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e = {v,w}$ in $E$?</td>
<td>Give me $v$’s neighbors.</td>
<td></td>
</tr>
<tr>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$O(deg(v))$ or $O(deg(w))$</td>
<td>$O(deg(v))$</td>
<td>$O(n + m)$</td>
</tr>
</tbody>
</table>

Generally better for **sparse** graphs.

See Lecture 9 IPython notebook for an actual implementation!
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.

start
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

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
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
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

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 – the bookkeeping will be useful later!
Depth First Search

currentTime = 0

• **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
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`

---

Current Time: 1

- `w` is `unvisited`
- `A` is `in progress`
- `C` is `unvisited`
- `D` is `in progress`
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*:
      • currentTime = DFS(v, currentTime)
      • currentTime ++
    • w.finishTime = currentTime
  • Mark w as *all done*
  • return currentTime
Depth First Search

currentTime = 2

• 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
Depth First Search

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

currentTime = 20

Start: 0

Start: 1

Takes until currentTime = 20

unvisited

in progress

all done

\( \text{currentTime} = 20 \)
Depth First Search

currentTime = 21

Start: 0

A

Start: 1

C

Takes until currentNodeTime = 20

unvisited

in progress

all done

\textbf{DFS}(w, \text{currentTime}):

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

Takes until currentTime = 20

Start: 0
End: 21

Start: 1

A

C

D

w

unvisited
in progress
all done

Start: 0
End: 21
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

currentTime = 22

Takes until currentTime = 20

Start: 0
End: 21
Fun exercise

• Write pseudocode for an iterative version of DFS.
DFS finds all the nodes reachable from the starting point.

One application of DFS: finding connected components.

In an undirected graph, this is called a connected component.
To explore the whole graph

• Do it repeatedly!
Why is it called depth-first?

• We are implicitly building a tree:

• 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 at most twice.
  - Once from each of its endpoints
- And basically we don’t do anything else.
- So...

\[ O(m) \]
Running time

To explore just the connected component we started in

• Assume we are using the linked-list format for G.
• Say $C = (V', E')$ is a connected component.
• We visit each vertex in C exactly once.
  • Here, “visit” means “call DFS on”
• At each vertex $w$, we:
  • Do some book-keeping: $O(1)$
  • Loop over $w$’s neighbors and check if they are visited (and then potentially make a recursive call): $O(1)$ per neighbor or $O(deg(w))$ total.

• Total time:
  • $\sum_{w \in V'} (O(deg(w)) + O(1))$
  • $= O(|E'| + |V'|)$
  • $= O(|E'|)$

In a connected graph, $|V'| \leq |E'| + 1$. 

69
Running time

To explore the whole graph

• Explore the connected components one-by-one.
• This takes time $O(n + m)$
  • Same computation as before:
    \[ \sum_{w \in V} (O(\deg(w)) + O(1)) = O(|E| + |V|) = O(n + m) \]

Here the running time is $O(m)$ like before

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.
Pre-lecture exercise

• How can you sign up for classes so that you never violate the pre-req requirements?

• More practically, how can you install packages without violating dependency requirements?
Application of DFS: topological sorting

• Find an ordering of vertices so that all of the dependency requirements are met.
  • Aka, if v comes before w in the ordering, there is not an edge from w to v.

Suppose the dependency graph has no cycles: it is a Directed Acyclic Graph (DAG)
Can’t always eyeball it.
Let’s do DFS

What do you notice about the finish times? Any ideas for how we should do topological sort?
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.
A more general statement
(this holds even if there are cycles)
This is called the “parentheses theorem” in CLRS

- If \( v \) is a descendant of \( w \) in this tree:
  
  \[
  \begin{align*}
  & w.\text{start} \quad v.\text{start} \quad v.\text{finish} \quad w.\text{finish} \\
  \text{timeline} &
  \end{align*}
  \]

- If \( w \) is a descendant of \( v \) in this tree:
  
  \[
  \begin{align*}
  & v.\text{start} \quad w.\text{start} \quad w.\text{finish} \quad v.\text{finish} \\
  \text{timeline} &
  \end{align*}
  \]

- If neither are descendants of each other:
  
  \[
  \begin{align*}
  & v.\text{start} \quad v.\text{finish} \quad w.\text{start} \quad w.\text{finish} \\
  \text{timeline} &
  \end{align*}
  \]
So to prove this →

If \( A \rightarrow B \)

Then \( B.\text{finishTime} < A.\text{finishTime} \)

Suppose the underlying graph has no cycles

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

• Then

  - A.\text{startTime} ≤ B.\text{startTime} ≤ B.\text{finishTime} ≤ A.\text{finishTime}

• aka, B.\text{finishTime} < A.\text{finishTime}.
So to prove this →

If

Then $B.\text{finishTime} < A.\text{finishTime}$

Suppose the underlying graph has no cycles

- **Case 2**: $B$ is a **NOT** descendant of $A$ in the DFS tree.
  - Notice that $A$ can’t be a descendant of $B$ or else there’d be a cycle; so it looks like this.
  - 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

  - $B.\text{finishTime}$
  - $B.\text{startTime}$
  - $A.\text{finishTime}$
  - $A.\text{startTime}$

  - aka, $B.\text{finishTime} < A.\text{finishTime}$. 
Theorem

• If we run DFS on a directed acyclic graph,

If A → B

Then B.finishTime < A.finishTime
Back to topological sorting

- In what order should I install packages?
- In reverse order of finishing time in DFS!

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

- Do DFS
- When you mark a vertex as **all done**, put it at the **beginning** of the list.

- dpkg
- coreutils
- tar
- libbz2
- libselinux1
- multiarch_support
For implementation, see IPython notebook

```python
In [69]: print(G)
CS161Graph with:
   Vertices:
      dkpg, coreutils, multiarch_support, libselinux1, libbz2, tar,
   Edges:
      (dkpg, multiarch_support) (dkpg, coreutils) (dkpg, tar) (dkpg, libbz2)
      (coreutils, libbz2) (coreutils, libselinux1) (libselinux1, multiarch_support)
      (libbz2, libselinux1)

In [71]: V = topoSort(G)
for v in V:
   print(v)

dkpg
tar
coreutils
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.
Example:

This example skipped in class – here for reference.

Start:0

Unvisited

In progress

All done
Example

This example skipped in class – here for reference.

Unvisited
In progress
All done

Start: 0
Start: 1
This example skipped in class – here for reference.
Example

This example skipped in class – here for reference.
Example

This example skipped in class – here for reference.
This example skipped in class – here for reference.
Example

This example skipped in class – here for reference.
Example

This example skipped in class – here for reference.

Do them in this order:

A
C
D
B
Another use of DFS that we’ve already seen

- In-order enumeration of binary search trees

Do DFS and print a node’s label when you are done with the left child and before you begin the right child.

HINT for Problem 6 on the current homework! You can do this for any BST, even a gooseTree!
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

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

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

World: EXPLORED!
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 $v$ 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.
Running time and extension to directed graphs

- To explore the whole graph, explore the connected components one-by-one.
  - Same argument as DFS: BFS 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:

- First we go as broadly as we can.
Pre-lecture exercise

• What Samuel L. Jackson’s Bacon number?

(Answer: 2)
I wrote the pre-lecture exercise before I realized that I really wanted an example with distance 3.

Kevin Bacon

Oliver Sacks

Tilda Swinton

It is really hard to find people with Bacon number 3!
Application of BFS: shortest path

• How long is the shortest path between w and v?
Application of BFS: 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 number of edges in the shortest path between them.

Call this the "BFS tree"
Proof overview that the BFS tree behaves like it should

• Proof by induction.

• Inductive hypothesis for \( j \):
  • For all \( i < j \) the vertices in \( L_i \) have distance \( i \) from \( v \).

• Base case:
  • \( L_0 = \{ v \} \), so we’re good.

• Inductive step:
  • Let \( w \) be in \( L_j \). Want to show \( \text{dist}(v,w) = j \).
  • \( \text{We know } \text{dist}(v,w) \leq j \), since \( \text{dist}(v, w’s \ parent \ in \ L_{j-1}) = j-1 \) by induction, so that gives a path of length \( j \) from \( v \) to \( w \).
  • On the other hand, \( \text{dist}(v,w) \geq j \), since if \( \text{dist}(v,w) < j \), \( w \) would have shown up in an earlier layer.
  • Thus, \( \text{dist}(v,w) = j \).

• Conclusion:
  • For each vertex \( w \) in \( V \), if \( w \) is in \( L_j \), then \( \text{dist}(v,w) = j \).
What have we learned?

• The BFS tree is useful for computing distances between pairs of vertices.
• We can find the shortest path between u and v in time $O(m)$. 
Another application of BFS

• Testing bipartite-ness
Pre-lecture exercise: fish

• You have a bunch of fish and two fish tanks.
• Some pairs of fish will fight if put in the same tank.
  • Model this as a graph: connected fish will fight.
• Can you put the fish in the two tanks so that there is no fighting?
Bipartite graphs

• A bipartite graph 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?
Application of BFS:
Testing Bipartiteness

• 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

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 of a graph with nodes and edges, illustrating the concept of Breadth-First Search.](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

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

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

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

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

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

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

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...
Some proof required

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

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

• 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 have we learned?

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

• Pre-lecture exercise: Strongly Connected What-Now?