Programing Abstractions
Spring 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 18.6
At this point in the quarter...

https://i.redd.it/e5uylwsqzizx.jpg
Today's Topics

• Logistics
  • Trailblazer: Final assignment! Out tomorrow.

• More on Graphs (and a bit on Trees)
  • Depth First Search
  • Breadth First Search
You create Google Maps!

You need to implement four different (but related) types of searches:

- Breadth First Search (today)
- Dijkstra (Friday)
- A* (Friday)
- Alternate (you must determine algorithm)
When you hover over an XKCD comic, you get an extra joke:

Wikipedia trivia: if you take any article, click on the first link in the article text not in parentheses or italics, and then repeat, you will eventually end up at "Philosophy".

Wikipedia trivia: if you take any article, click on the first link in the article text not in parentheses or italics, and then repeat, you will eventually end up at "Philosophy".

Is this true??


As of February 2016, 97% of all articles in Wikipedia eventually lead to the article Philosophy.

How can we find out? We shall see!
Recall that a graph is the "wild west of trees" — graphs relate vertices (nodes) to each other by way of edges, and they can be directed or undirected. Take the following directed graph:

A search on this graph starts at one vertex and attempts to find another vertex. If it is successful, we say there is a path from the start to the finish vertices.

What paths are there from 0 to 6?

- 0 → 4 → 6
- 0 → 3 → 1 → 6
- 0 → 3 → 7 → 5 → 6
Graph Searching

Recall that a *graph* is the "wild west of trees" — graphs relate *vertices* (nodes) to each other by way of *edges*, and they can be directed or undirected. Take the following directed graph:

A search on this graph starts at one vertex and attempts to find another vertex. If it is successful, we say there is a path from the start to the finish vertices.

What paths are there from 0 to 6?

0 → 3 → 7 → 5 → 7 → 5 → 6
0 → 3 → 1 → 0 → 3 → 1 → 0 → 4 → 6

We have to watch out for cycles!
Graph Searching

What paths are there from 3 to 2?

3 - 1 - 6 - 2
3 - 7 - 5 - 6 - 2
3 - 1 - 0 - 4 - 6 - 2
Graph Searching

What paths are there from 4 to 1?

None! :(
We have different ways to search graphs:

- **Depth First Search**: From the start vertex, explore as far as possible along each branch before backtracking.

- **Breadth First Search**: From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors.

Both methods have pros and cons — let's explore the algorithms.
Depth First Search (DFS)

From the start vertex, explore as far as possible along each branch before backtracking.

This is often implemented recursively. For a graph, you must mark visited vertices, or you might traverse forever (e.g., c->e->f->c->e...)

DFS from a to h (assuming a-z order) visits:

a
b
e
f
c
i (dead end — back to c,f,e,b,a)
d
g
h

path found: a→d→g→h

Notice: not the shortest!
Depth First Search (DFS): Recursive pseudocode

**dfs** from $v_1$ to $v_2$:
- base case: if at $v_2$, found!
- mark $v_1$ as visited.
- for all edges from $v_1$ to its neighbors:
  - if neighbor $n$ is unvisited, recursively call **dfs**($n$, $v_2$).
Depth First Search (DFS): Recursive pseudocode

**dfs** from \(v_1\) to \(v_2\):
- mark \(v_1\) as visited.
- for all edges from \(v_1\) to its neighbors:
  - if neighbor \(n\) is unvisited, recursively call **dfs**\((n, \ v_2)\).

Let's look at **dfs** from \(h\) to \(c\):

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>false</td>
</tr>
<tr>
<td>b</td>
<td>false</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>false</td>
</tr>
<tr>
<td>e</td>
<td>false</td>
</tr>
<tr>
<td>f</td>
<td>false</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>false</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>
**Depth First Search (DFS): Recursive pseudocode**

**dfs** from $v_1$ to $v_2$:
- mark $v_1$ as visited.
- for all edges from $v_1$ to its neighbors:
  - if neighbor $n$ is unvisited, recursively call **dfs**($n$, $v_2$).

Let's look at **dfs** from $h$ to $c$:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>false</td>
</tr>
<tr>
<td>b</td>
<td>false</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>false</td>
</tr>
<tr>
<td>e</td>
<td>false</td>
</tr>
<tr>
<td>f</td>
<td>false</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>

Call stack:

- **dfs**(h,c)
**Depth First Search (DFS): Recursive pseudocode**

**dfs** from $v_1$ to $v_2$:
- mark $v_1$ as visited.
- for all edges from $v_1$ to its neighbors:
  - if neighbor $n$ is unvisited, recursively call dfs($n$, $v_2$).

Let's look at dfs from $h$ to $c$:

<table>
<thead>
<tr>
<th>Vertex Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>

**call stack:**

- dfs(e,c)
- dfs(h,c)
DFS from $v_1$ to $v_2$:
mark $v_1$ as visited.
for all edges from $v_1$ to its neighbors:
if neighbor $n$ is unvisited, recursively call $\text{dfs}(n, v_2)$.

Let's look at $\text{dfs}$ from $h$ to $c$:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>false</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>false</td>
</tr>
<tr>
<td>e</td>
<td>true</td>
</tr>
<tr>
<td>f</td>
<td>false</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>

Call stack:
\[
dfs(a, c) \\
dfs(e, c) \\
dfs(h, c)
\]
**Depth First Search (DFS): Recursive pseudocode**

**dfs from** $v_1$ **to** $v_2$:
- mark $v_1$ as visited.
- for all edges from $v_1$ to its neighbors:
  - if neighbor $n$ is unvisited, recursively call $\text{dfs}(n, v_2)$.

Let's look at $\text{dfs}$ from $h$ to $c$:

**Call stack:**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>true</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>false</td>
</tr>
<tr>
<td>e</td>
<td>true</td>
</tr>
<tr>
<td>f</td>
<td>false</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>

**Vertex Map**

- $a$: true
- $b$: true
- $c$: false
- $d$: false
- $e$: true
- $f$: false
- $g$: false
- $h$: true
- $i$: false
Depth First Search (DFS): Recursive pseudocode

**dfs** from \( v_1 \) to \( v_2 \):
mark \( v_1 \) as visited.
for all edges from \( v_1 \) to its neighbors:
if neighbor \( n \) is unvisited, recursively call \( \text{dfs}(n, v_2) \).

Let's look at **dfs** from \( h \) to \( c \):

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>true</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>true</td>
</tr>
<tr>
<td>e</td>
<td>true</td>
</tr>
<tr>
<td>f</td>
<td>false</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>

**call stack:**

dfs(d,c)
dfs(a,c)
dfs(e,c)
dfs(h,c)
Depth First Search (DFS): Recursive pseudocode

**dfs** from \( v_1 \) to \( v_2 \):
- mark \( v_1 \) as visited.
- for all edges from \( v_1 \) to its neighbors:
  - if neighbor \( n \) is unvisited, recursively call **dfs**(\( n \), \( v_2 \)).

Let's look at **dfs** from \( h \) to \( c \):

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>true</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>true</td>
</tr>
<tr>
<td>e</td>
<td>true</td>
</tr>
<tr>
<td>f</td>
<td>false</td>
</tr>
<tr>
<td>g</td>
<td>true</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>

**call stack:**
- **dfs**(g,c)
- **dfs**(d,c)
- **dfs**(a,c)
- **dfs**(e,c)
- **dfs**(h,c)
Depth First Search (DFS): Recursive pseudocode

dfs from $v_1$ to $v_2$:
mark $v_1$ as visited.
for all edges from $v_1$ to its neighbors:
    if neighbor $n$ is unvisited, recursively call dfs($n$, $v_2$).

Let's look at dfs from $h$ to $c$:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>true</td>
</tr>
<tr>
<td>$b$</td>
<td>true</td>
</tr>
<tr>
<td>$c$</td>
<td>false</td>
</tr>
<tr>
<td>$d$</td>
<td>true</td>
</tr>
<tr>
<td>$e$</td>
<td>true</td>
</tr>
<tr>
<td>$f$</td>
<td>true</td>
</tr>
<tr>
<td>$g$</td>
<td>true</td>
</tr>
<tr>
<td>$h$</td>
<td>true</td>
</tr>
<tr>
<td>$i$</td>
<td>false</td>
</tr>
</tbody>
</table>

call stack:

```
dfs(h,c)
dfs(e,c)
dfs(a,c)
dfs(g,c)
dfs(d,c)
```
**Depth First Search (DFS): Recursive pseudocode**

**dfs** from $v_1$ to $v_2$:
- mark $v_1$ as visited.
- for all edges from $v_1$ to its neighbors:
  - if neighbor $n$ is unvisited, recursively call **dfs**($n$, $v_2$).

Let's look at **dfs** from $h$ to $c$:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>true</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>true</td>
</tr>
<tr>
<td>e</td>
<td>true</td>
</tr>
<tr>
<td>f</td>
<td>true</td>
</tr>
<tr>
<td>g</td>
<td>true</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>

**call stack:**

```
dfs(f, c)  
dfs(e, c)  
dfs(h, c)  
```
Depth First Search (DFS): Recursive pseudocode

**dfs** from $v_1$ to $v_2$:
- mark $v_1$ as visited.
- for all edges from $v_1$ to its neighbors:
  - if neighbor $n$ is unvisited, recursively call $dfs(n, v_2)$.

Let's look at **dfs** from $h$ to $c$:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>true</td>
</tr>
<tr>
<td>c</td>
<td>true</td>
</tr>
<tr>
<td>d</td>
<td>true</td>
</tr>
<tr>
<td>e</td>
<td>true</td>
</tr>
<tr>
<td>f</td>
<td>true</td>
</tr>
<tr>
<td>g</td>
<td>true</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>

**call stack:**

- $dfs(c, c)$
- $dfs(f, c)$
- $dfs(e, c)$
- $dfs(h, c)$

**found!**
**Depth First Search (DFS): Iterative pseudocode**

\textbf{dfs} from $v_1$ to $v_2$:
- create a stack, $s$
- $s$.push($v_1$)
- while $s$ is not empty:
  - $v = s$.pop()
  - if $v$ has not been visited:
    - mark $v$ as visited
    - push all neighbors of $v$ onto the stack
**Depth First Search (DFS): Iterative pseudocode**

**dfs** from \( v_1 \) to \( v_2 \):
- create a stack, \( s \)
- \( s.push(v_1) \)
- while \( s \) is not empty:
  - \( v = s.pop() \)
  - if \( v \) has not been visited:
    - mark \( v \) as visited
    - push all neighbors of \( v \) onto the stack

Let's look at **dfs** from \( h \) to \( c \):
- push \( h \)
Depth First Search (DFS): Iterative pseudocode

**dfs** from $v_1$ to $v_2$:
- create a stack, $s$
- $s$.push($v_1$)
- while $s$ is not empty:
  - $v = s$.pop()
  - if $v$ has not been visited:
    - mark $v$ as visited
    - push all neighbors of $v$ onto the stack

Let's look at **dfs** from $h$ to $c$:
- in while loop:
  - $v = s$.pop()
  - $v$: $h$

Vertex Map:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>false</td>
</tr>
<tr>
<td>b</td>
<td>false</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>false</td>
</tr>
<tr>
<td>e</td>
<td>false</td>
</tr>
<tr>
<td>f</td>
<td>false</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>
**Depth First Search (DFS): Iterative pseudocode**

**dfs** from $v_1$ to $v_2$:
- create a stack, $s$
- $s.push(v_1)$
- while $s$ is not empty:
  - $v = s.pop()$
  - if $v$ has not been visited:
    - mark $v$ as visited
    - push all neighbors of $v$ onto the stack

Let's look at **dfs** from $h$ to $c$:

- in while loop:
  - push all neighbors of $h$
### Depth First Search (DFS): Iterative pseudocode

**dfs** from \( v_1 \) to \( v_2 \):
- create a stack, \( s \)
- \( s.push(v_1) \)
- while \( s \) is not empty:
  - \( v = s.pop() \)
  - if \( v \) has not been visited:
    - mark \( v \) as visited
    - push all neighbors of \( v \) onto the stack

Let's look at **dfs** from \( h \) to \( c \):
- in while loop:
  - \( v = s.pop() \)
  - \( v: f \)

---

**Vertex Map**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>false</td>
</tr>
<tr>
<td>b</td>
<td>false</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>false</td>
</tr>
<tr>
<td>e</td>
<td>false</td>
</tr>
<tr>
<td>f</td>
<td>true</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>
Depth First Search (DFS): Iterative pseudocode

**dfs** from $v_1$ to $v_2$:
- create a stack, $s$
- $s$.push($v_1$)
- while $s$ is not empty:
  - $v = s$.pop()
  - if $v$ has not been visited:
    - mark $v$ as visited
    - push all neighbors of $v$ onto the stack

Let’s look at **dfs** from $h$ to $c$:

in while loop:
- push all
  - neighbors of $f$

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>false</td>
</tr>
<tr>
<td>b</td>
<td>false</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>false</td>
</tr>
<tr>
<td>e</td>
<td>false</td>
</tr>
<tr>
<td>f</td>
<td>true</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>
Depth First Search (DFS): Iterative pseudocode

**dfs** from $v_1$ to $v_2$:
- create a stack, $s$
- $s$.push($v_1$)
- while $s$ is not empty:
  - $v = s$.pop()
  - if $v$ has not been visited:
    - mark $v$ as visited
    - push all neighbors of $v$ onto the stack

Let's look at **dfs** from $h$ to $c$:
- in while loop:
  - $v = s$.pop()
- $v$: c
- found — stop!

---

**Vertex Map**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>false</td>
</tr>
<tr>
<td>b</td>
<td>false</td>
</tr>
<tr>
<td>c</td>
<td>false</td>
</tr>
<tr>
<td>d</td>
<td>false</td>
</tr>
<tr>
<td>e</td>
<td>false</td>
</tr>
<tr>
<td>f</td>
<td>true</td>
</tr>
<tr>
<td>g</td>
<td>false</td>
</tr>
<tr>
<td>h</td>
<td>true</td>
</tr>
<tr>
<td>i</td>
<td>false</td>
</tr>
</tbody>
</table>
Depth First Search (DFS)

Both the recursive and iterative solutions to DFS were correct, but because of the subtle differences in recursion versus using a stack, they traverse the nodes in a different order.

For the h to c example, the iterative solution happened to be faster, but for different graphs the recursive solution may have been faster.

To retrieve the DFS path found, pass a collection parameter to each cell (if recursive) and choose-explore-unchoose (our old friend, recursive backtracking!)
Depth First Search (DFS)

DFS is guaranteed to find a path if one exists.

It is not guaranteed to find the best or shortest path! (i.e., it is not optimal)
Breadth First Search (BFS)

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors.

This *isn't easy to implement* recursively. The iterative algorithm is very similar to the DFS iterative, except that we use a queue.

BFS from a to i (assuming a-z order) visits:

\[
\begin{align*}
\text{a} & \rightarrow \text{b} \\
\text{a} & \rightarrow \text{d} \\
\text{a} & \rightarrow \text{e} \\
\text{a} & \rightarrow \text{d} \rightarrow \text{g} \\
\text{a} & \rightarrow \text{d} \rightarrow \text{h} \\
\text{a} & \rightarrow \text{e} \rightarrow \text{f} \\
\text{a} & \rightarrow \text{d} \rightarrow \text{h} \rightarrow \text{i}
\end{align*}
\]

\{neighbors of a\}
\{neighbors of d\}

Notice: the shortest!
Breadth First Search (BFS): Iterative pseudocode

bfs from $v_1$ to $v_2$:
create a queue of paths (a vector), $q$
$q$.enqueue($v_1$ path)
while $q$ is not empty and $v_2$ is not yet visited:
    path = $q$.dequeue()
    $v$ = last element in path
    if $v$ is not visited:
        mark $v$ as visited
    if $v$ is the end vertex, we can stop.
    for each unvisited neighbor of $v$:
        make new path with $v$'s neighbor as last element
        enqueue new path onto $q$
Breadth First Search (BFS): Iterative pseudocode

Let's look at `bfs` from `a` to `i`:

```
queue:    front
          a
```

- Vector`<Vertex *>` `startPath`
- `startPath.add(a)`
- `q.enqueue(startPath)`

`bfs` from `v1` to `v2`:
- create a queue of paths (a vector), `q`
- `q.enqueue(v1 path)`
- while `q` is not empty and `v2` is not yet visited:
  - `path = q.dequeue()`
  - `v = last element in path`
  - if `v` is not visited:
    - mark `v` as visited
    - if `v` is the end vertex, we can stop.
    - for each unvisited neighbor of `v`:
      - make new path with `v`'s neighbor as last element
      - enqueue new path onto `q`
Breadth First Search (BFS): Iterative pseudocode

Let's look at **bfs** from a to i:

- **queue:**
  - `ae`  `ad`  `ab`

  *in while loop:*
  - `curPath = q.dequeue() (path is a)`
  - `v = last element in curPath (v is a)`
  - `mark v as visited`
  - `enqueue all unvisited neighbor paths onto q`

**Visited Set:**
- `a`
Breadth First Search (BFS): Iterative pseudocode

- **bfs** from v₁ to v₂:
  - create a queue of paths (a vector), q
  - q.enqueue(v₁ path)
  - while q is not empty and v₂ is not yet visited:
    - path = q.dequeue()
    - v = last element in path
    - if v is not visited:
      - mark v as visited
      - if v is the end vertex, we can stop.
      - for each unvisited neighbor of v:
        - make new path with v's neighbor as last element
        - enqueue new path onto q

Let's look at **bfs** from a to i:

```
queue:    front
         a
         b
         c
         d
         e
         f
         g
         h
         i
```

in while loop:
- curPath = q.dequeue() (path is ab)
- v = last element in curPath (v is b)
- mark v as visited
- enqueue all unvisited neighbor paths onto q

- **Visited Set:** a b
Breadth First Search (BFS): Iterative pseudocode

**bfs** from $v_1$ to $v_2$:
- create a queue of paths (a vector), $q$
- $q$.enqueue($v_1$ path)
- while $q$ is not empty and $v_2$ is not yet visited:
  - path = $q$.dequeue()
  - $v$ = last element in path
  - if $v$ is not visited:
    - mark $v$ as visited
    - if $v$ is the end vertex, we can stop.
    - for each unvisited neighbor of $v$:
      - make new path with $v$'s neighbor as last element
      - enqueue new path onto $q$

Let's look at **bfs** from a to i:

<table>
<thead>
<tr>
<th>queue:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adh</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>abe</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ae</td>
</tr>
</tbody>
</table>

in while loop:
- curPath = $q$.dequeue() (path is ad)
- $v$ = last element in curPath (v is d)
- mark $v$ as visited
- enqueue all unvisited neighbor paths onto $q$

Visited Set:
- a
- b
- d
Breadth First Search (BFS): Iterative pseudocode

**bfs** from \( v_1 \) to \( v_2 \):
- create a queue of paths (a vector), \( q \)
- \( q.enqueue(v_1 \text{ path}) \)
- while \( q \) is not empty and \( v_2 \) is not yet visited:
  - \( \text{path} = q.dequeue() \)
  - \( v = \text{last element in path} \)
  - if \( v \) is not visited:
    - mark \( v \) as visited
    - if \( v \) is the end vertex, we can stop.
    - for each unvisited neighbor of \( v \):
      - make new path with \( v \)'s neighbor as last element
      - enqueue new path onto \( q \)

Let's look at **bfs** from \( a \) to \( i \):

<table>
<thead>
<tr>
<th>queue:</th>
<th></th>
<th></th>
<th></th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>aef</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adh</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>abe</td>
</tr>
</tbody>
</table>

in while loop:
- \( \text{curPath} = q.dequeue() \) (path is \( ae \))
- \( v = \text{last element in curPath (v is e)} \)
- mark \( v \) as visited
- enqueue all unvisited neighbor paths onto \( q \)
Breadth First Search (BFS): Iterative pseudocode

Let's look at **bfs** from a to i:

```
queue:     front
          aef  adh  adg
```

in while loop:
- `curPath = q.dequeue()` (path is abe)
- `v = last element in curPath (v is e)`
- mark v as visited (already been marked, no need to enqueue neighbors)
Breadth First Search (BFS): Iterative pseudocode

**bfs from** \( v_1 \) to \( v_2 \):
- create a queue of paths (a vector), \( q \)
- \( q.enqueue(v_1 \text{ path}) \)
- while \( q \) is not empty and \( v_2 \) is not yet visited:
  - path = q.dequeue()
  - \( v = \) last element in path
  - if \( v \) is not visited:
    - mark \( v \) as visited
    - if \( v \) is the end vertex, we can stop.
    - for each unvisited neighbor of \( v \):
      - make new path with \( v \)'s neighbor as last element
      - enqueue new path onto \( q \)

Let's look at **bfs** from \( a \) to \( i \):

**queue:**

\[
\begin{array}{c|c|c|c|c}
& & & & \text{front} \\
\hline
\text{adgh} & \text{aef} & \text{adh} & \text{adgh} & \text{aef} & \text{adh} \\
\end{array}
\]

in while loop:
- curPath = q.dequeue() (path is adg)
- \( v = \) last element in curPath (\( v \) is g)
- mark \( v \) as visited
- enqueue all unvisited neighbor paths onto \( q \)
**Breadth First Search (BFS): Iterative pseudocode**

**bfs** from $v_1$ to $v_2$:
- create a queue of paths (a vector), $q$
- $q.enqueue(v_1\text{ path})$
- while $q$ is not empty and $v_2$ is not yet visited:
  - path = $q.dequeue()$
  - $v$ = last element in path
  - if $v$ is not visited:
    - mark $v$ as visited
    - if $v$ is the end vertex, we can stop.
    - for each unvisited neighbor of $v$:
      - make new path with $v$’s neighbor as last element
      - enqueue new path onto $q$

Let's look at **bfs** from $a$ to $i$:

- **queue:**
  - front
    - adhi adhf adgh aef

in while loop:
- curPath = $q.dequeue()$ (path is adh)
- $v$ = last element in curPath ($v$ is h)
- mark $v$ as visited
- enqueue all unvisited neighbor paths onto $q$
Let's look at BFS from a to i:

queue: 

in while loop:

curPath = q.dequeue() (path is aef)
v = last element in curPath (v is f)
mark v as visited
enqueue all unvisited neighbor paths onto q

Visited Set:

bfs from v₁ to v₂:
create a queue of paths (a vector), q
q.enqueue(v₁ path)  
while q is not empty and v₂ is not yet visited:
  path = q.dequeue()
v = last element in path
if v is not visited:
  mark v as visited
  if v is the end vertex, we can stop.
  for each unvisited neighbor of v:
    make new path with v's neighbor as last element
    enqueue new path onto q
Breadth First Search (BFS): Iterative pseudocode

Let's look at **bfs** from a to i:

**queue:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>aefc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adhi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adhf</td>
</tr>
</tbody>
</table>

in while loop:

curPath = q.dequeue() (path is adgh)

v = last element in curPath (v is h)

mark v as visited (already been marked, no need to enqueue neighbors)

**Visited Set:**

- a
- b
- d
- e
- f
- h
Breadth First Search (BFS): Iterative pseudocode

**bfs from v₁ to v₂:**
create a queue of paths (a vector), q
q.enqueue(v₁ path)
while q is not empty and v₂ is not yet visited:
  path = q.dequeue()
  v = last element in path
  if v is not visited:
    mark v as visited
    if v is the end vertex, we can stop.
    for each unvisited neighbor of v:
      make new path with v’s neighbor as last element
      enqueue new path onto q

Let’s look at **bfs** from a to i:

queue:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>

in while loop:
curPath = q.dequeue() (path is adhf)
v = last element in curPath (v is f)
mark v as visited (already been marked, no need to enqueue neighbors)
Let's look at **bfs** from a to i:

**queue:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>aefc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adhi</td>
</tr>
</tbody>
</table>

In while loop:
- `curPath = q.dequeue()` (path is adhi)
- `v = last element in curPath (v is i)`
- **found!**

**Visited Set:**

- a
- b
- d
- e
- f
- h
- i
So I downloaded Wikipedia…

It turns out that you *can* download Wikipedia, but it is > 10 Terabytes (!) uncompressed. The reason Wikipedia asks you for money every so often is because they have lots of fast computers with lots of memory, and this is expensive (so donate!)

But, the Internet is just a graph...so, Wikipedia pages are just a graph...let's just do the searching by taking advantage of this: download pages as we need them.
What kind of search is the "getting to philosophy" algorithm?
"Clicking on the first lowercase link in the main text of a Wikipedia article, and then repeating the process for subsequent articles, usually eventually gets one to the Philosophy article."

This is a depth-first search! To determine if a Wikipedia article will get to Philosophy, we just select the first link each time. If we ever have to select a second link (or if a first-link refers to a visited vertex), then that article doesn't get to Philosophy.
We can also perform a Breadth First Search, as well. How would this change our search?

A BFS would look at all links on a page, then all links for each link on the page, etc. This has the potential of taking a long time, but it will find a shortest path.
References and Advanced Reading

• References:

• Advanced Reading:
  • Visualizations:
    • https://www.cs.usfca.edu/~galles/visualization/DFS.html
    • https://www.cs.usfca.edu/~galles/visualization/BFS.html
A Breadth First Search on a tree will produce a "level order traversal":

Breadth First Search: a b c d e g h f i

This is necessary if we want to print the tree to the screen in a pretty way, such that it retains its tree-like structure.