# CS 106B
## Lecture 25: Depth First and Breadth First Searching

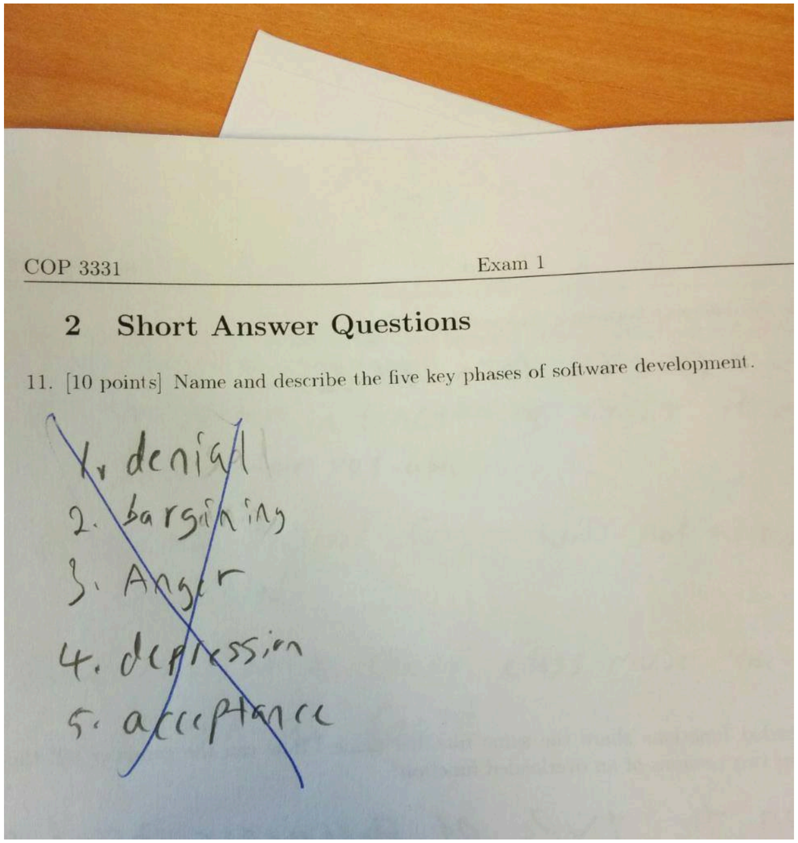
Monday, November 28, 2016

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 18.6

# At this point in the quarter…



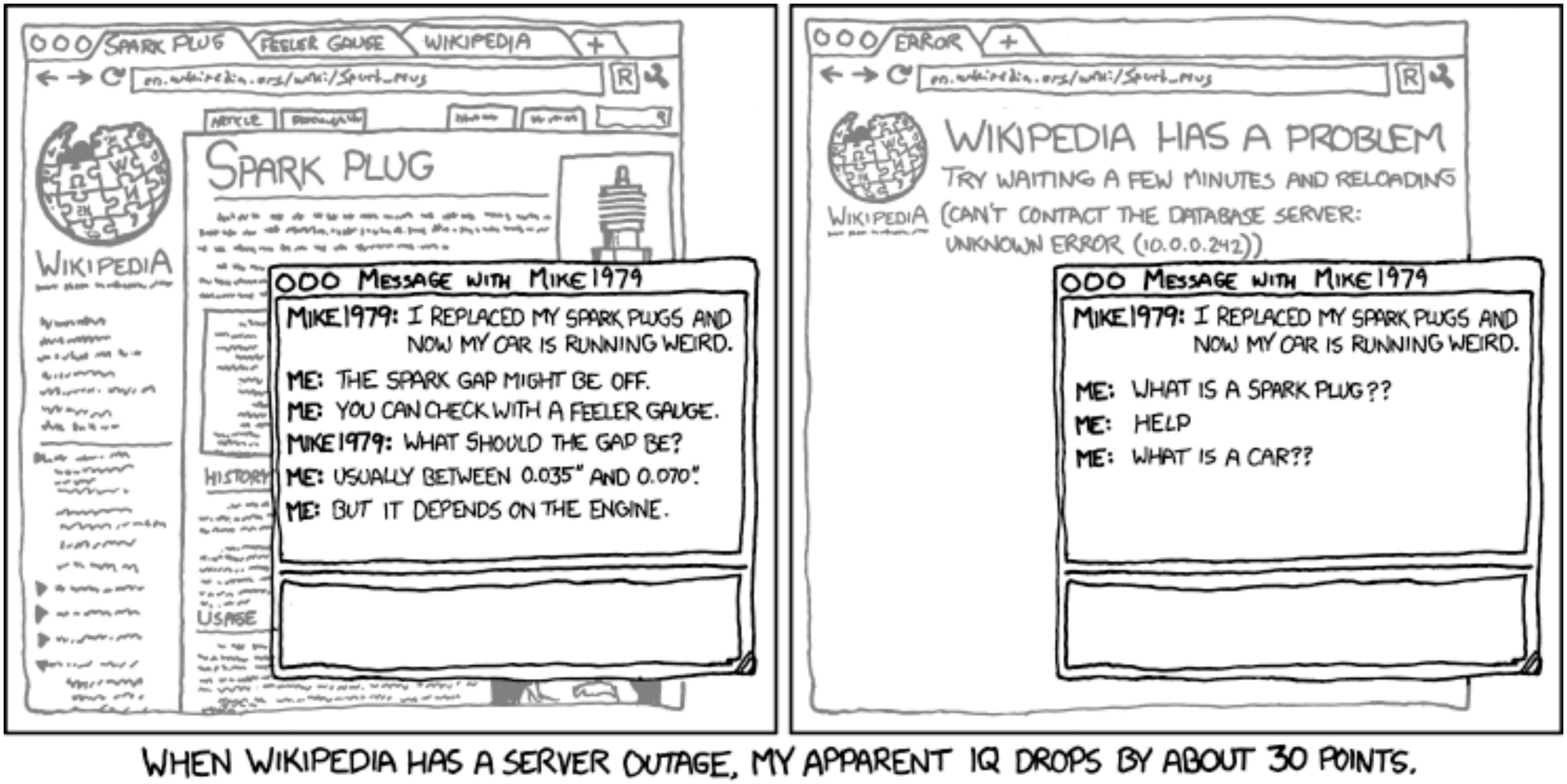

https://i.redd.it/e5uylwsqzizx.jpg

# Today's Topics

- Logistics
  - Chris office hours tomorrow: moved to 10am-11am, and will hold additional hours 3:30-5pm.
  - Candy Graph Challenge: Please email Chris Piech by 5pm today with your solutions!
  - Final Exam: 2 weeks from today! There are no make-up or alternate exams.
  - Assignment 7: Will be due on the last Friday of classes, no late days allowed.

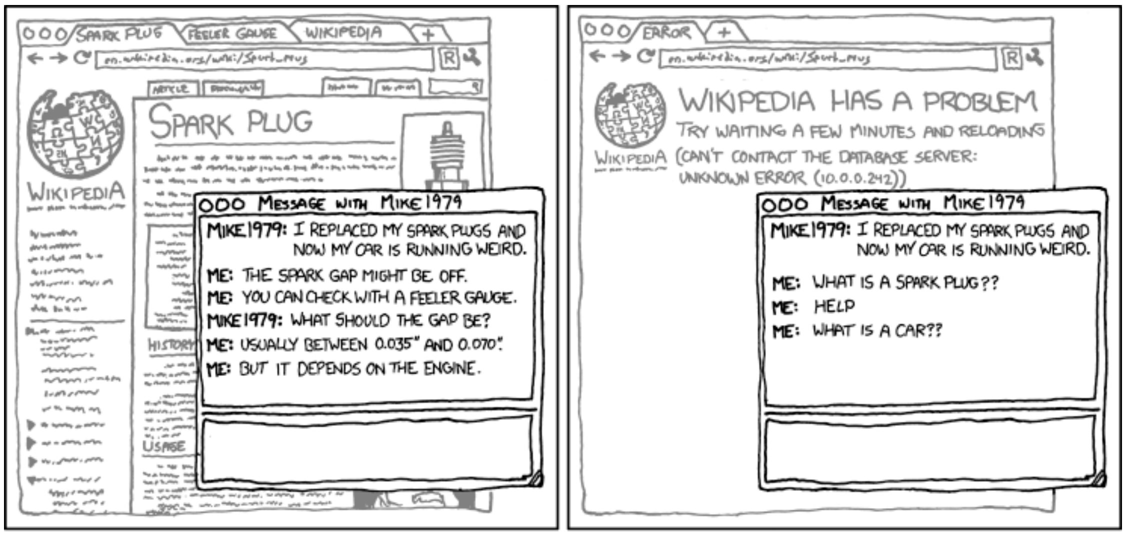
- More on Graphs (and a bit on Trees)
  - Depth First Search
  - Breadth First Search

# Wikipedia




XKCD 903, Extended Mind, http://xkcd.com/903/

# Wikipedia




WHEN WIKIPEDIA HAS A SERVER OUTAGE, MY APPARENT IQ DROPS BY ABOUT 30 POINTS.

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".**

XKCD 903, Extended Mind, http://xkcd.com/903/

# Wikipedia

**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??

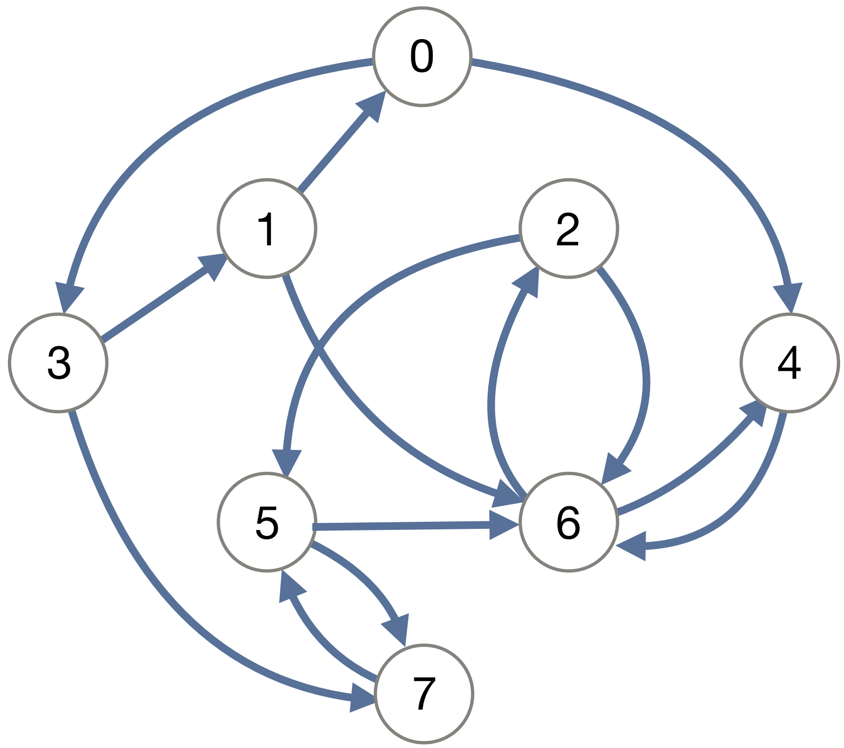
According to the Wikipedia article "Wikipedia:Getting to Philosophy" (so meta), (https://en.wikipedia.org/wiki/Wikipedia:Getting_to_Philosophy):

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

How can we find out? We shall see!

# Graph Searching

Recall from the last couple of lectures 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?

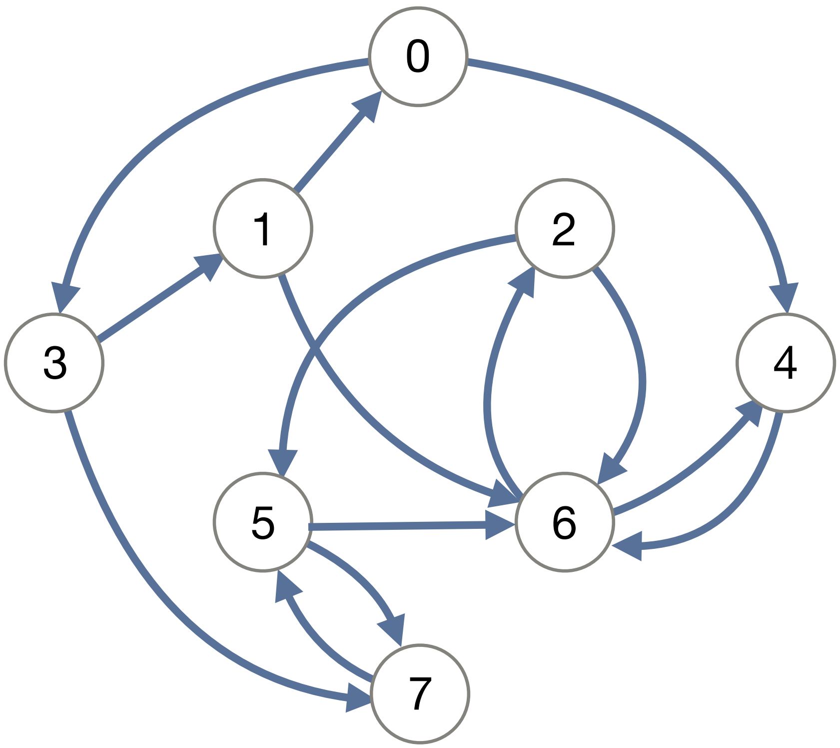
0☞4☞6

0☞3☞1☞6

0☞3☞7☞5☞6

# Graph Searching

What paths are there from 3 to 2?



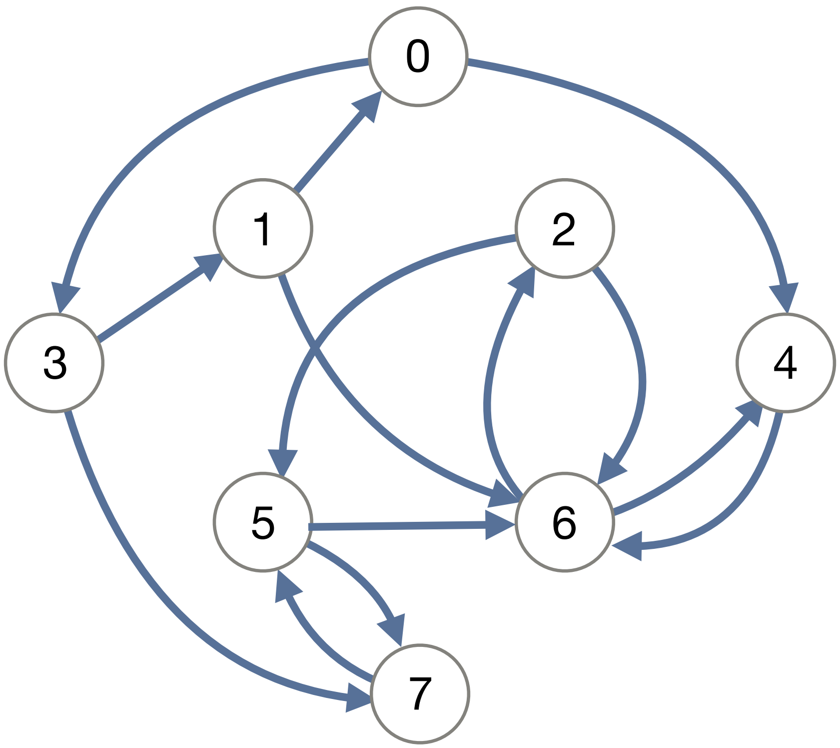

3☞1☞6☞2

3☞7☞5☞6☞2

3☞1☞0☞4☞6☞2

# Graph Searching

What paths are there from 4 to 1?




**None! :(**

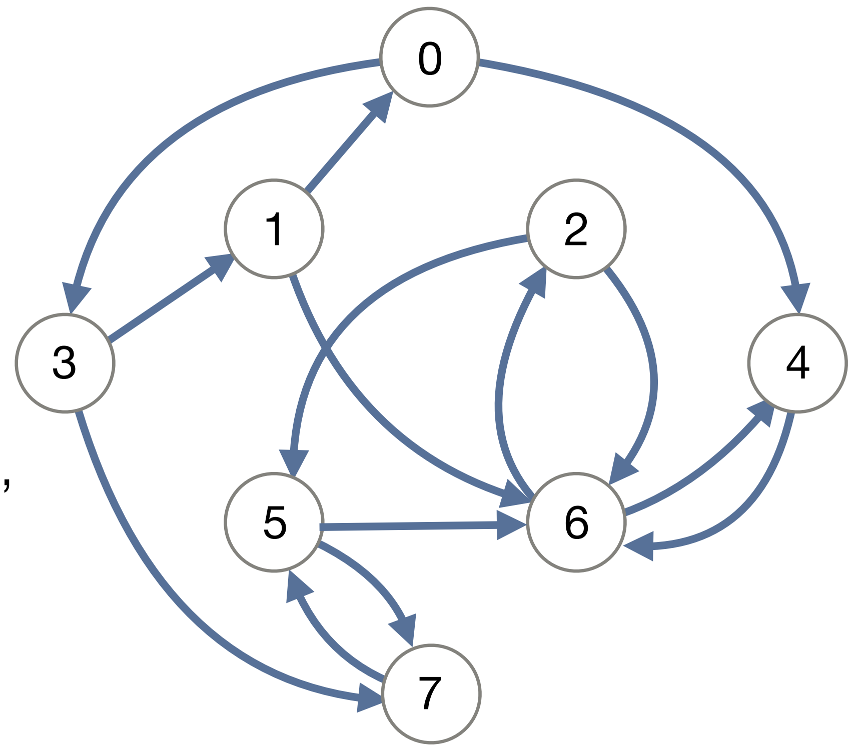
# Graph Searching

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.

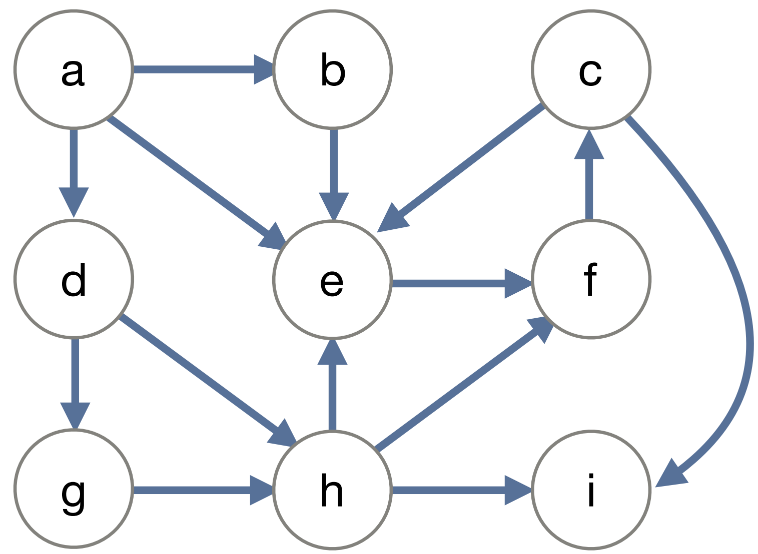
Both methods have pros and cons — let's explore the algorithms.



# 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:

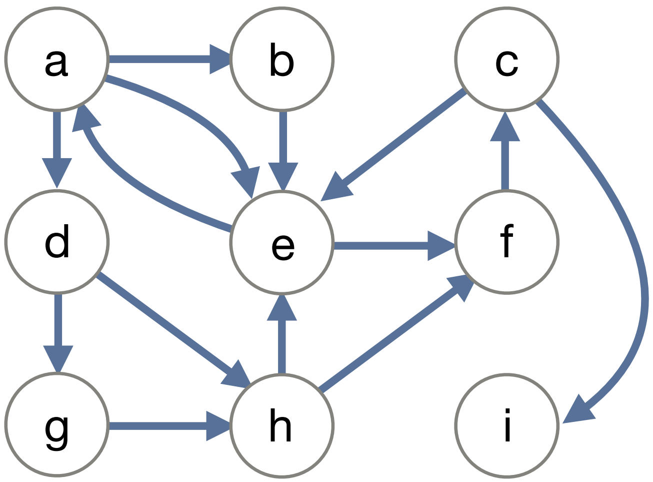
a☞<br>
  b☞<br>
    e☞<br>
     f☞<br>
       c☞<br>
         i (dead end — back to c,f,e,b,a)<br>
  d☞<br>
    g☞<br>
      h

Notice: not the shortest!

path found: a☞d☞g☞h

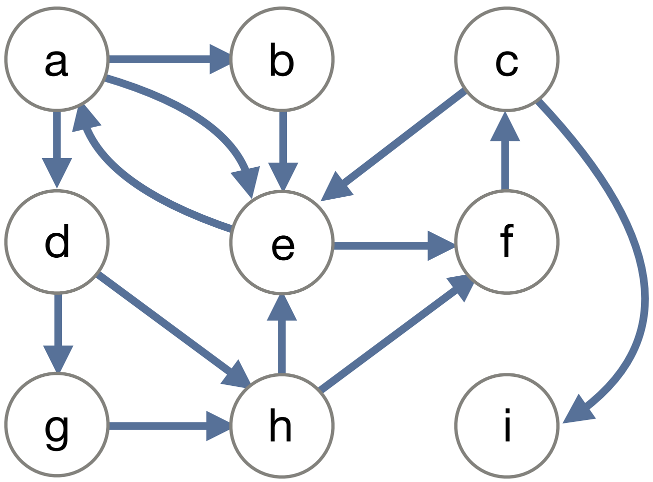
# Depth First Search (DFS): Recursive pseudocode

**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, $v_2$).

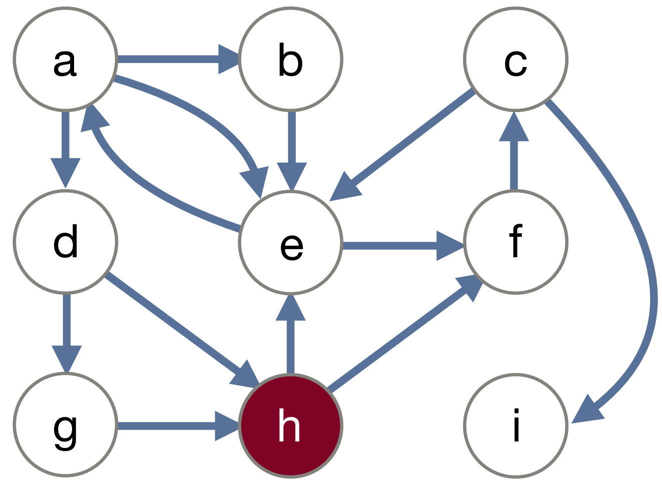



Let's look at **dfs** from h to c:

| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| f | false |
| g | false |
| h | false |
| 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 **dfs**(n, $v_2$).




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

call stack:

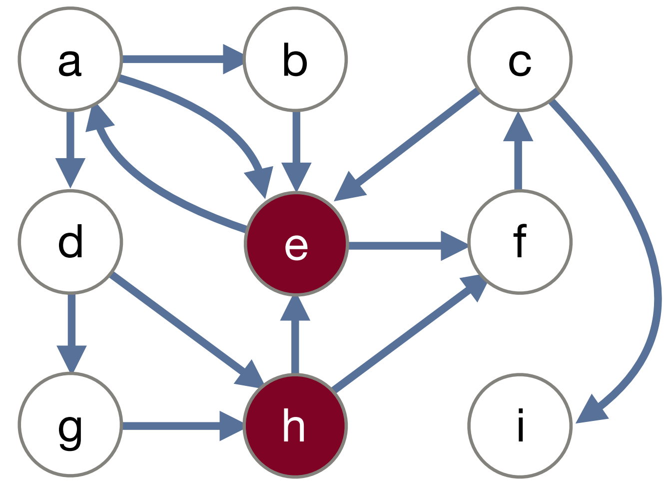
dfs(h,c)

Vertex Map

| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| 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 **dfs**(n, $v_2$).




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

call stack:

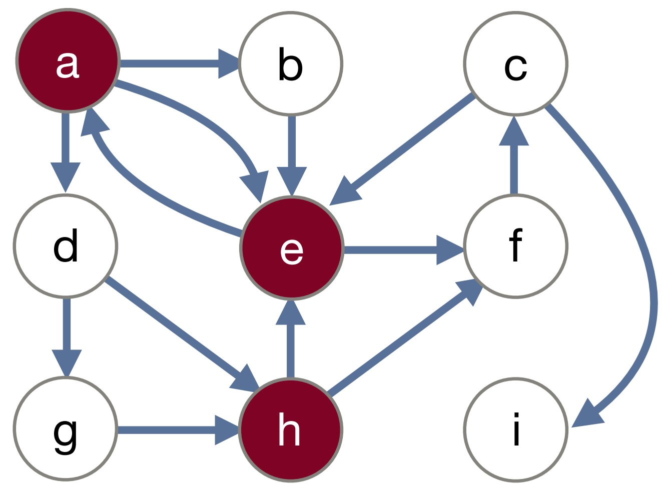
| |
|---|
| |
| |
| |
| |
| |
| dfs(e,c) |
| dfs(h,c) |

Vertex Map

| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| 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 **dfs**(n, $v_2$).




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

call stack:

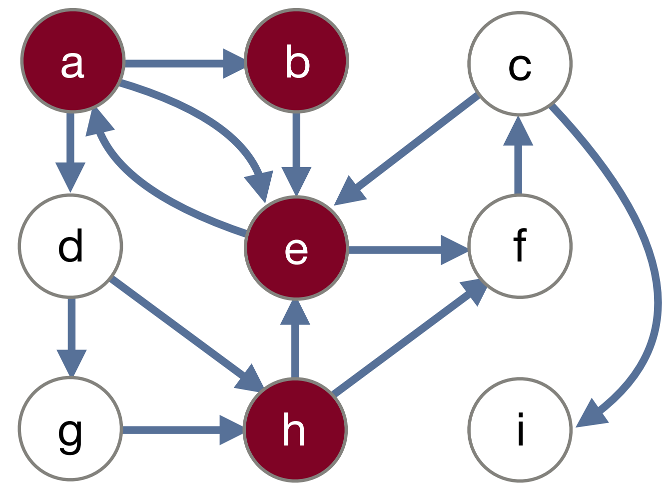
| |
|---|
| |
| |
| |
| |
| dfs(a,c) |
| dfs(e,c) |
| dfs(h,c) |

Vertex Map

| Vertex | Visited? |
|---|---|
| a | true |
| b | false |
| 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 **dfs**(n, $v_2$).




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

call stack:

| |
|---|
| |
| |
| |
| dfs(b,c) |
| dfs(a,c) |
| dfs(e,c) |
| dfs(h,c) |

Vertex Map

| Vertex | Visited? |
|---|---|
| a | true |
| b | true |
| c | false |
| d | false |
| e | true |
| f | false |
| g | false |
| h | true |
| i | false |

# Depth First Search (DFS): Recursive pseudocode

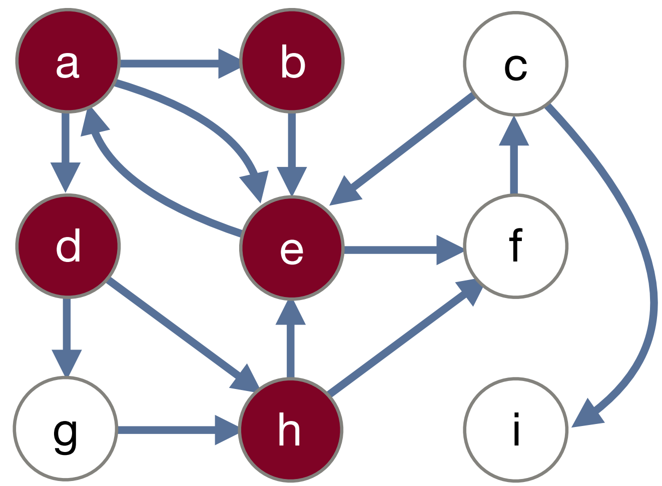
**dfs** from $v_1$ to $v_2$:

  mark $v_1$ as visited.

  for all edges from $v_1$ to its neighbors:

    if neighbor n is unvisited, recursively call **dfs**(n, $v_2$).

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




call stack:

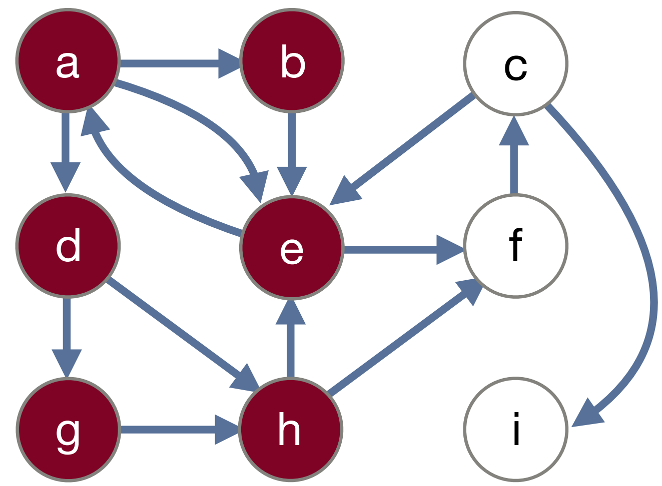
| |
|---|
| |
| |
| |
| dfs(d,c) |
| dfs(a,c) |
| dfs(e,c) |
| dfs(h,c) |

Vertex Map

| Vertex | Visited? |
|---|---|
| a | true |
| b | true |
| c | false |
| d | true |
| 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 **dfs**(n, $v_2$).




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

call stack:

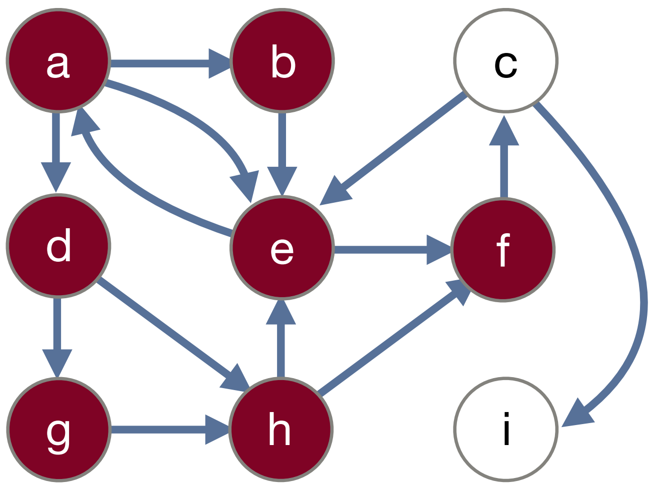
| |
|---|
| |
| |
| dfs(g,c) |
| dfs(d,c) |
| dfs(a,c) |
| dfs(e,c) |
| dfs(h,c) |

Vertex Map

| Vertex | Visited? |
|---|---|
| a | true |
| b | true |
| c | false |
| d | true |
| e | true |
| f | false |
| g | true |
| 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 **dfs**(n, $v_2$).




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

call stack:

| |
|---|
| |
| |
| ~~dfs(g,c)~~ |
| ~~dfs(d,c)~~ |
| ~~dfs(a,c)~~ |
| ~~dfs(e,c)~~ |
| dfs(h,c) |

Vertex Map

| Vertex | Visited? |
|---|---|
| a | true |
| b | true |
| c | false |
| d | true |
| e | true |
| f | true |
| g | true |
| h | true |
| i | false |

# Depth First Search (DFS): Recursive pseudocode

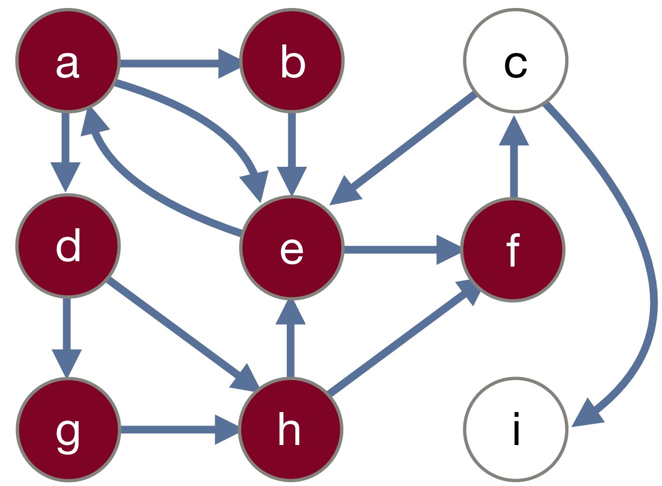
**dfs** from $v_1$ to $v_2$:

mark $v_1$ as visited.

for all edges from $v_1$ to its neighbors:

if neighbor n is unvisited, recursively call **dfs**(n, $v_2$).

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




Vertex Map

| Vertex | Visited? |
|---|---|
| a | true |
| b | true |
| c | false |
| d | true |
| e | true |
| f | true |
| g | true |
| h | true |
| i | false |

call stack:

| |
|---|
| |
| |
| |
| |
| |
| dfs(f,c) |
| dfs(h,c) |

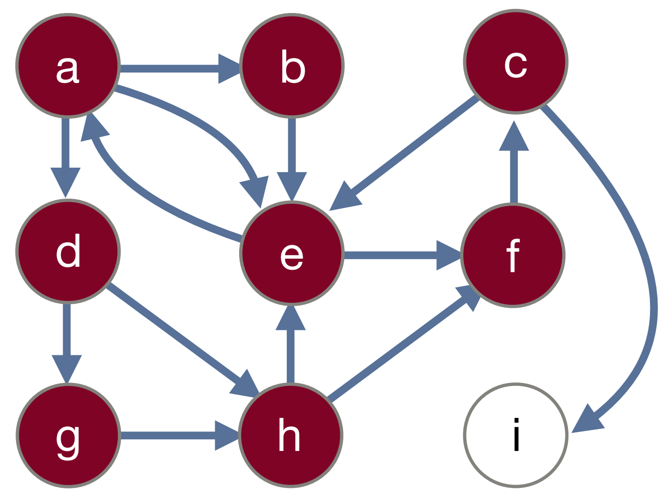
# Depth First Search (DFS): Recursive pseudocode

**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:

Vertex Map

| Vertex | Visited? |
|---|---|
| a | true |
| b | true |
| c | true |
| d | true |
| e | true |
| f | true |
| g | true |
| h | true |
| i | false |

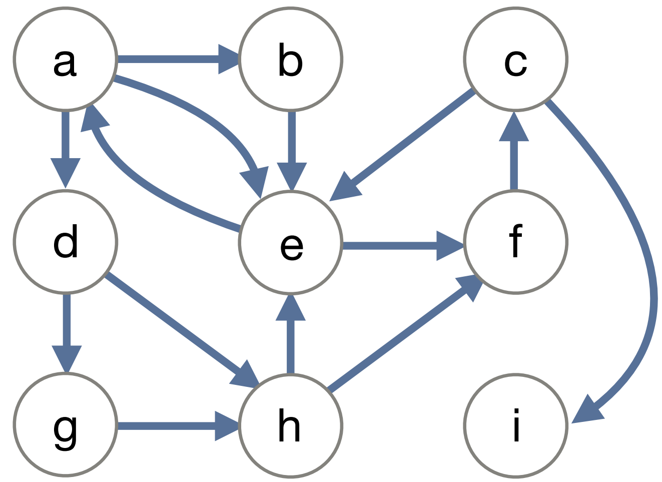
call stack:

| |
|---|
| |
| |
| |
| |
| dfs(c,c) |
| dfs(f,c) |
| dfs(h,c) |

found!

# Depth First Search (DFS): Iterative pseudocode

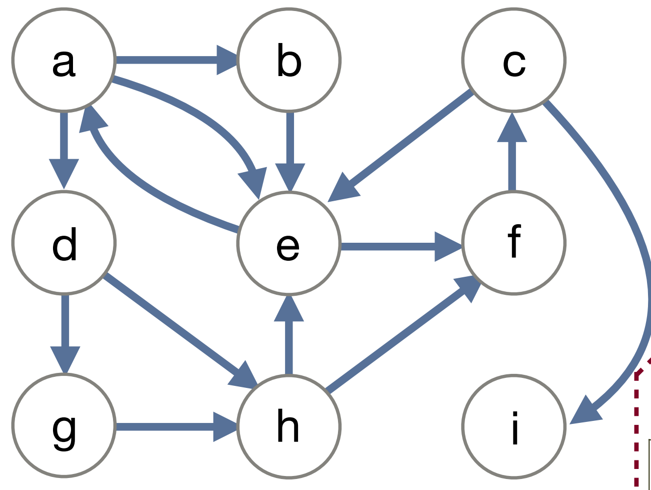
```
dfs from v1 to v2:
    create a stack, s
    s.push(v1)
    while s is not empty:
        v = s.pop()
        if v has not been visited:
            mark v as visited
            push all neighbors of v onto the stack
```



# Depth First Search (DFS): Iterative pseudocode

**dfs** from $v_1$ to $v_2$:

```
create a stack, s
s.push(v1)
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

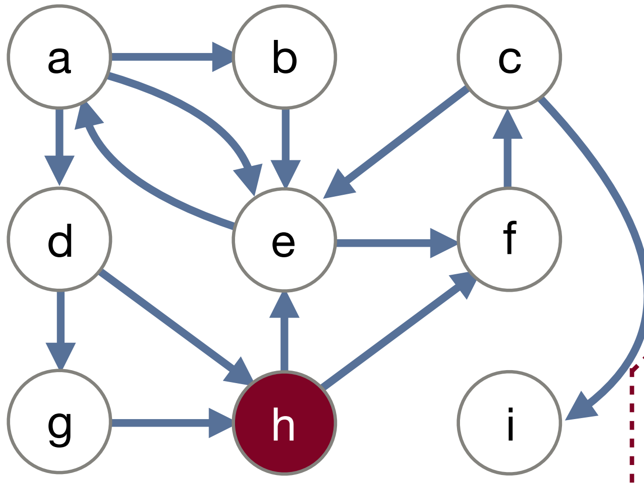
| stack s |
|---|
| |
| |
| |
| |
| |
| |
| h |

**Vertex Map**

| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| f | false |
| g | false |
| h | false |
| i | false |

# 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

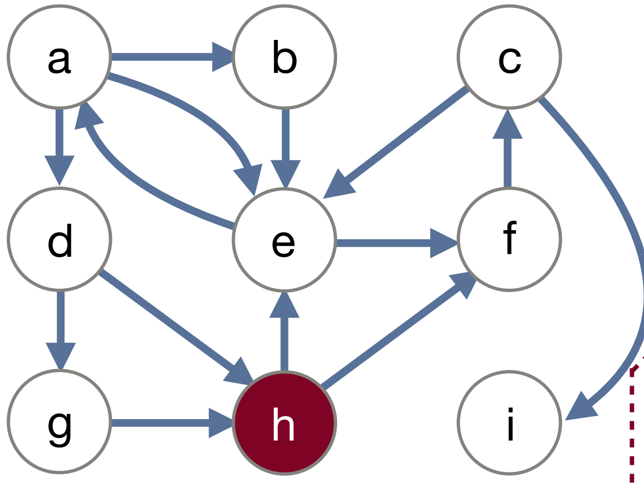
stack s

Vertex Map

| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| f | false |
| g | false |
| h | true |
| i | false |

# 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

stack s

| |
|---|
| |
| |
| |
| |
| |
| f |
| e |

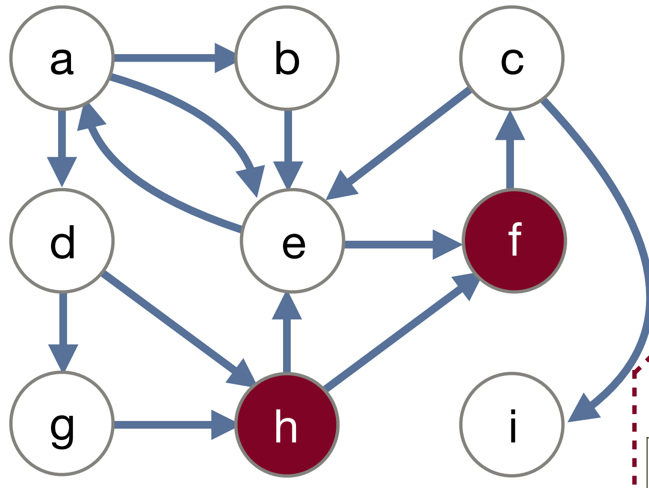
Vertex Map

| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| f | false |
| g | false |
| h | true |
| i | false |

# Depth First Search (DFS): Iterative pseudocode

**dfs** from $v_1$ to $v_2$:

```
create a stack, s
s.push(v1)
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

stack s

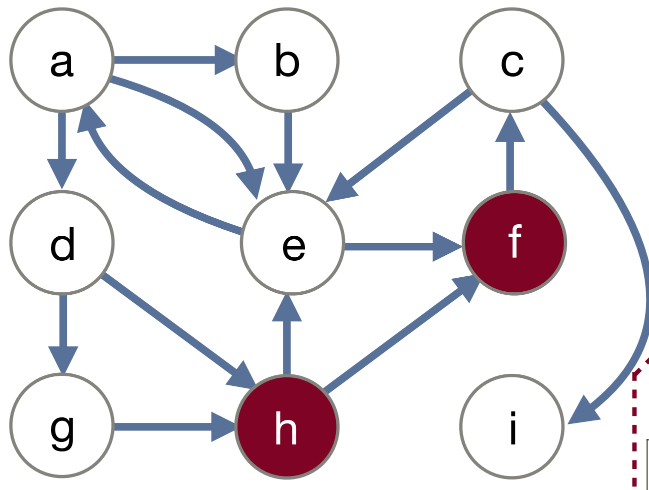
| |
|---|
| |
| |
| |
| |
| |
| |
| e |

Vertex Map

| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| f | true |
| g | false |
| h | true |
| i | false |

# 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

stack s

| |
|---|
| |
| |
| |
| |
| |
| c |
| e |

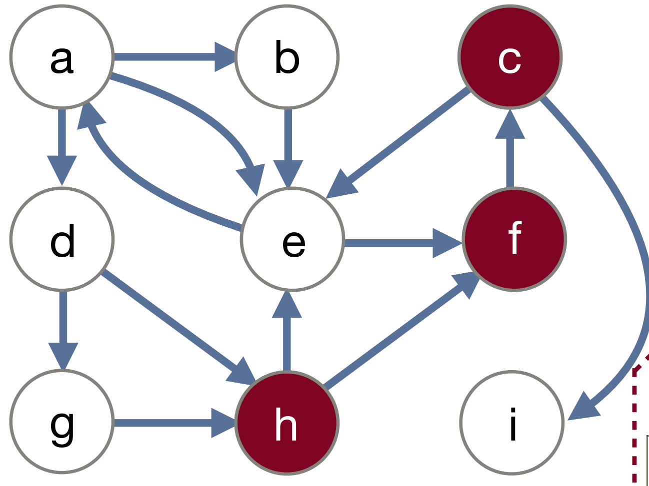
Vertex Map

| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| f | true |
| g | false |
| h | true |
| i | false |

# Depth First Search (DFS): Iterative pseudocode

**dfs** from $v_1$ to $v_2$:

```
create a stack, s
s.push(v1)
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!

| stack s |
|---|
| |
| |
| |
| |
| |
| c |
| e |

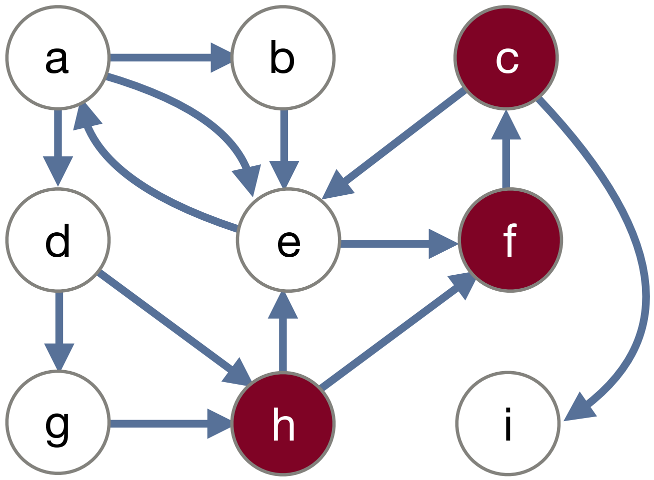
| Vertex Map | |
|---|---|
| Vertex | Visited? |
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| f | true |
| g | false |
| h | true |
| i | false |

# 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.

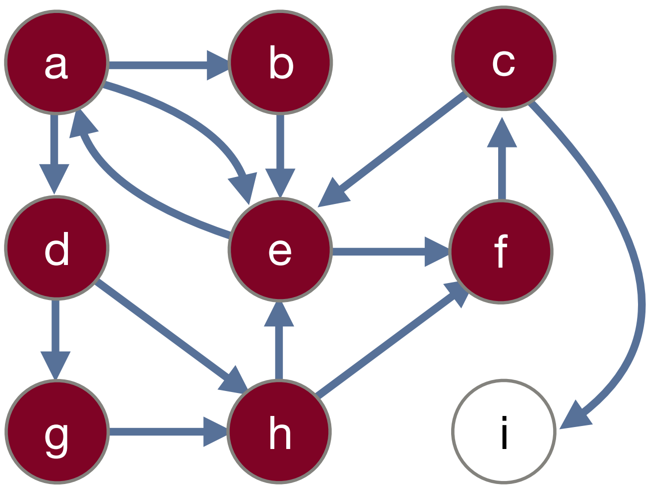
To retrieve the DFS path found, pass a collection parameter to each cell (if recursive) and choose-explore-unchoose (our old friend, recursive backtracking!)



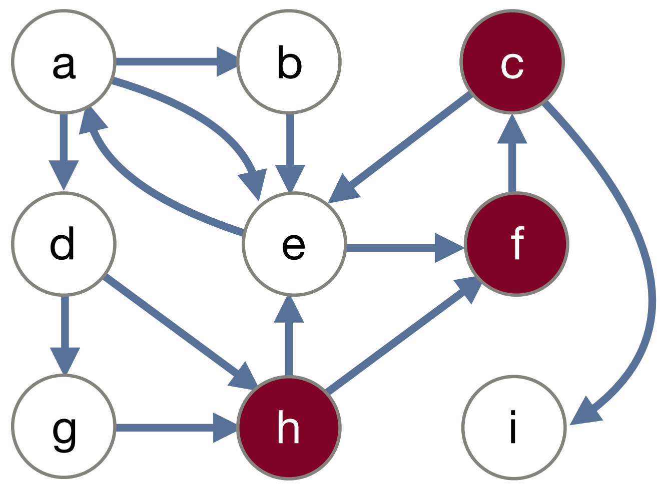
# Depth First Search (DFS)

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)




vs.



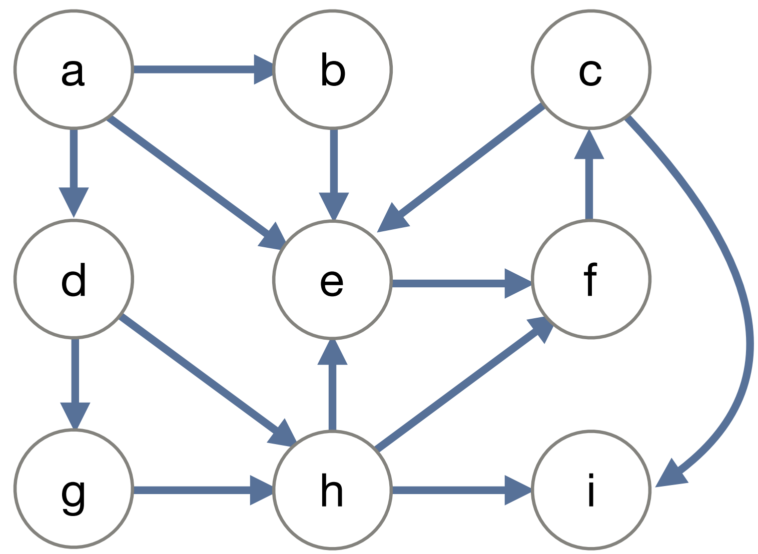
# Breadth First Search (BFS)

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

This *can't be implemented* 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:

a
a☞b
a☞d
a☞e
neighbors of a
a☞d☞g
a☞d☞h
neighbors of d
a☞e☞f
a☞d☞h☞i



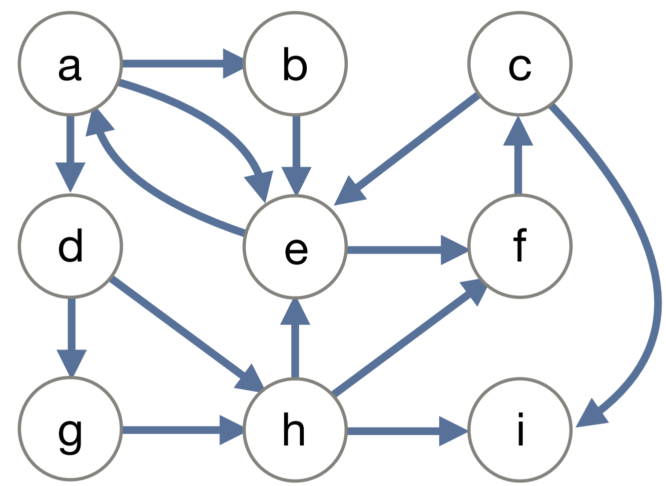

Notice: the shortest!

path: a☞d☞h☞i

# Breadth First Search (BFS): Iterative pseudocode

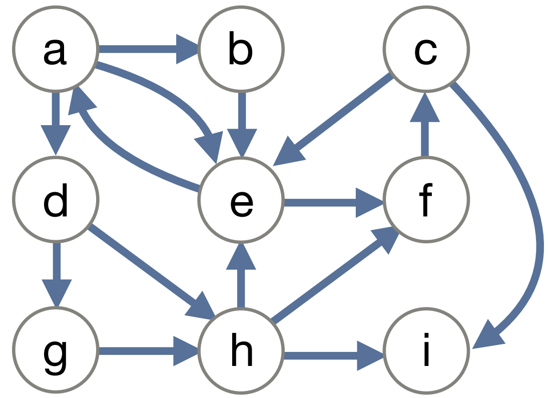
**bfs** from $v_1$ to $v_2$:

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
    mark v as visited
    for each unvisited neighbor of v:
        make new path with v as last element
        enqueue new path 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($v_1$ path)
- while q is not empty and $v_2$ is not yet visited:
  - path = q.dequeue()
  - v = last element in path
  - mark v as visited
  - for each unvisited neighbor of v:
    - make new path with v as last element
    - enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | false |
| c | false |
| d | false |
| e | false |
| f | false |
| g | false |
| h | false |
| i | false |

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

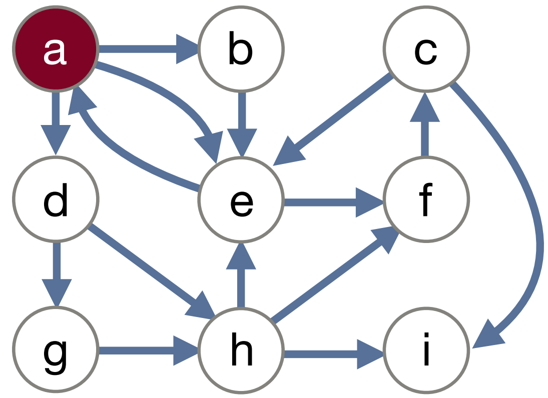
queue:

| | | | | | | | | front |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | a |

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

# 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
    mark v as visited
    for each unvisited neighbor of v:
      make new path with v as last element
      enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | true |
| b | false |
| c | false |
| d | false |
| e | false |
| f | false |
| g | false |
| h | false |
| i | false |

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

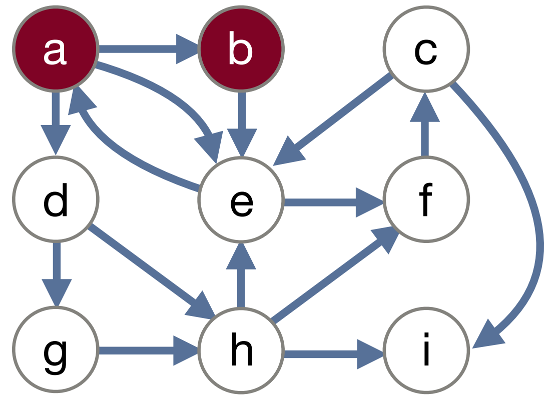
queue:

| | | | | | | | | front |
|---|---|---|---|---|---|---|---|---|
| | | | | | | 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

# 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
    mark v as visited
    for each unvisited neighbor of v:
      make new path with v as last element
      enqueue new path onto q




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

queue:

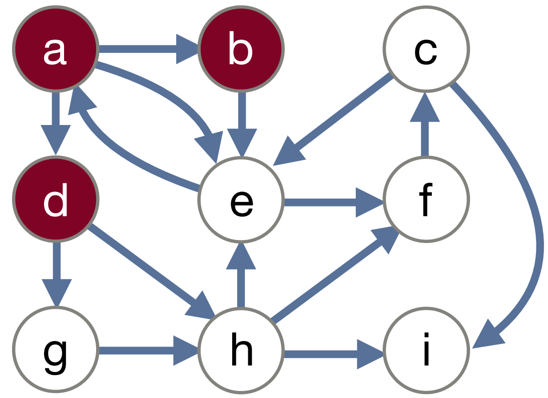
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | | | abe | ae | ad |

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

| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | false |
| e | false |
| f | false |
| g | false |
| h | false |
| i | false |

# 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
        mark v as visited
        for each unvisited neighbor of v:
            make new path with v as last element
            enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | false |
| f | false |
| g | false |
| h | false |
| i | false |

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

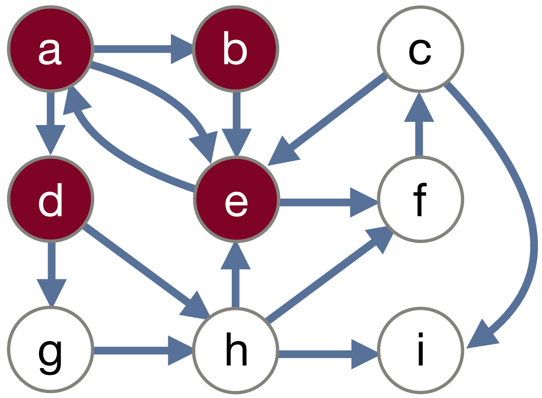
queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | | adh | adg | abe | ae |

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

# 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
    mark v as visited
    for each unvisited neighbor of v:
      make new path with v as last element
      enqueue new path onto q




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

queue:

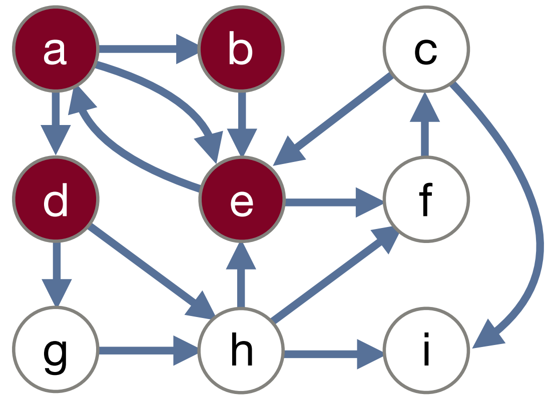
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | | aef | adh | adg | abe |

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

| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | false |
| g | false |
| h | false |
| i | false |

# 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
        mark v as visited
        for each unvisited neighbor of v:
            make new path with v as last element
            enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | false |
| g | false |
| h | false |
| i | false |

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

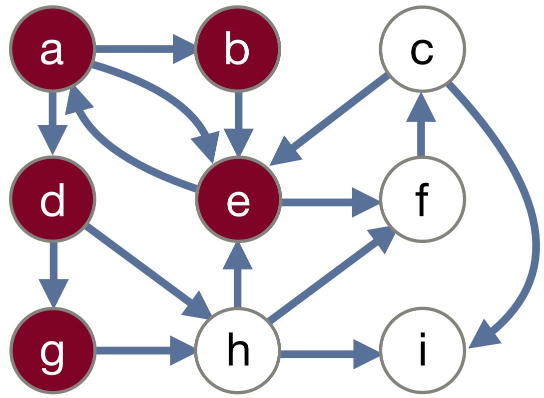
queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | | abef | 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)
  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($v_1$ path)
  while q is not empty and $v_2$ is not yet visited:
    path = q.dequeue()
    v = last element in path
    mark v as visited
    for each unvisited neighbor of v:
      make new path with v as last element
      enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | false |
| g | true |
| h | false |
| i | false |

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

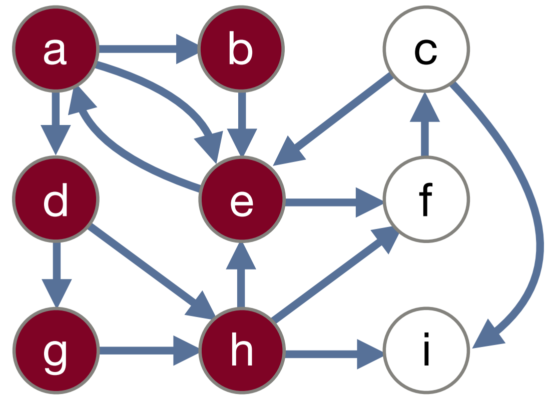
queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | | adgh | abef | aef | adh |

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($v_1$ path)
  while q is not empty and $v_2$ is not yet visited:
    path = q.dequeue()
    v = last element in path
    mark v as visited
    for each unvisited neighbor of v:
      make new path with v as last element
      enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | false |
| g | true |
| h | true |
| i | false |

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

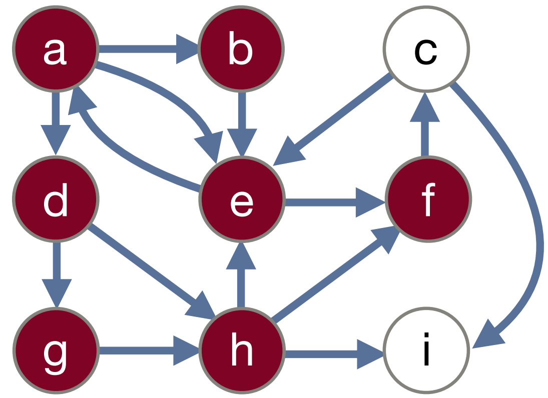
queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | adhi | adhf | adgh | abef | 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

# 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
    mark v as visited
    for each unvisited neighbor of v:
      make new path with v as last element
      enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | true |
| g | true |
| h | true |
| i | false |

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

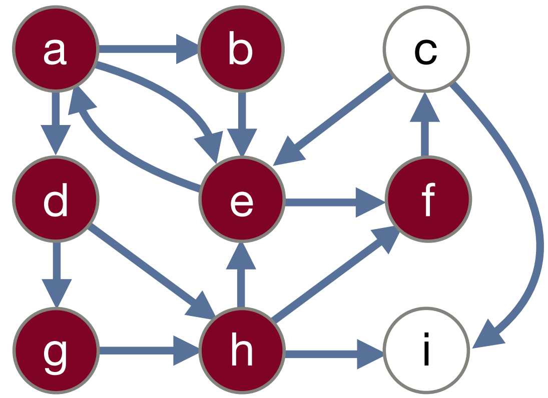
queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | aefc | adhi | adhf | adgh | abef |

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

# 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
    mark v as visited
    for each unvisited neighbor of v:
      make new path with v as last element
      enqueue new path onto q




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

queue:

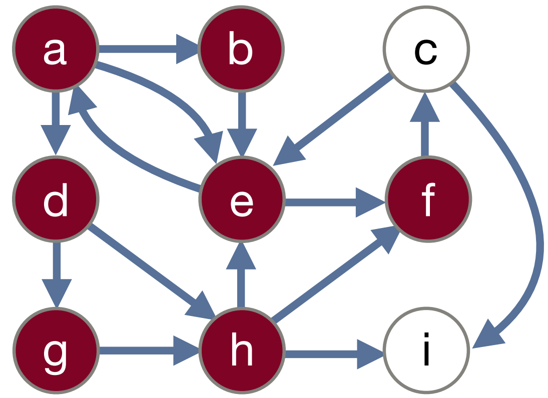
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | abefc | aefc | adhi | adhf | adgh |

in while loop:
  curPath = q.dequeue() (path is abef)
  v = last element in curPath (v is f)
  mark v as visited (already been marked)
  enqueue all unvisited neighbor paths onto q

| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | true |
| g | true |
| h | true |
| i | false |

# 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
  mark v as visited
  for each unvisited neighbor of v:
   make new path with v as last element
   enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | true |
| g | true |
| h | true |
| i | false |

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

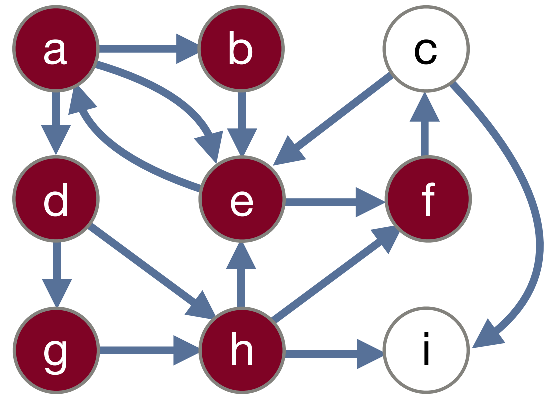
queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | adghi | abefc | aefc | adhi | adhf |

in while loop:
 curPath = q.dequeue() (path is adgh)
 v = last element in curPath (v is h)
 mark v as visited (already been marked)
 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($v_1$ path)
while q is not empty and $v_2$ is not yet visited:
path = q.dequeue()
v = last element in path
mark v as visited
for each unvisited neighbor of v:
make new path with v as last element
enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | true |
| g | true |
| h | true |
| i | false |

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

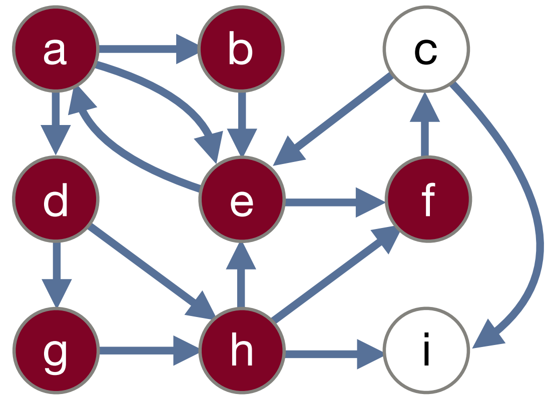
queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | adhfc | adghi | abefc | aefc | adhi |

in while loop:
curPath = q.dequeue() (path is adhf)
v = last element in curPath (v is f)
mark v as visited (already been marked)
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($v_1$ path)
  while q is not empty and $v_2$ is not yet visited:
    path = q.dequeue()
    v = last element in path
    mark v as visited
    for each unvisited neighbor of v:
      make new path with v as last element
      enqueue new path onto q




| Vertex | Visited? |
|---|---|
| a | false |
| b | true |
| c | false |
| d | true |
| e | true |
| f | true |
| g | true |
| h | true |
| i | false |

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

queue:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | front |
| | | | | adhfc | adghi | abefc | aefc | adhi |

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

# Wikipedia: Getting to Philosophy

WIKIPEDIA
The Free Encyclopedia

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.

# Wikipedia: Getting to Philosophy

WIKIPEDIA
The Free Encyclopedia

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.

# Wikipedia: Getting 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

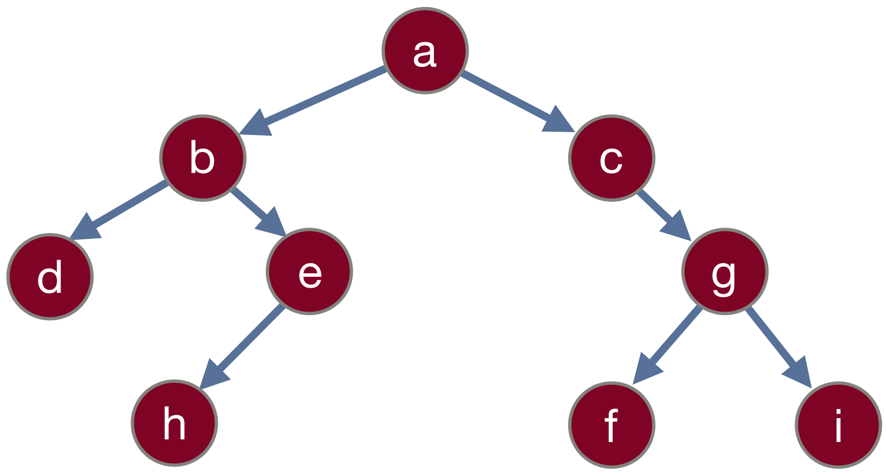
- **References:**
  - Depth First Search, Wikipedia: https://en.wikipedia.org/wiki/Depth-first_search
  - Breadth First Search, Wikipedia: https://en.wikipedia.org/wiki/Breadth-first_search

- **Advanced Reading:**
  - Visualizations:
  - https://www.cs.usfca.edu/~galles/visualization/DFS.html
  - https://www.cs.usfca.edu/~galles/visualization/BFS.html

# Extra Slides

# Breadth First Search (BFS): Tree searching

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.