Graph Algorithms

Thomas Schwarz, SJ
Searching in Graphs

- Exploring a maze
  - You are in the middle of a maze
    - How do you get out
  - Ariadne's solution:
    - Use a thread of glittering jewels in order to avoid using the same edges several times
  - Follow a wall
    - Works for simple mazes
Trémaux's Algorithm

- Trémaux's Algorithm aka Hansel and Gretel's aka Ariadne's
  - Carry bread and leave bread crumbs on each path you follow
  - If you come to an intersection, follow one where there are no bread crumbs, if you can
  - If you come to an intersection and everything has already been marked or you are at a dead-end, turn around if you came at a path that has only one thread of crumbs
- If not, follow a path that has only one trail of crumbs.
Trémaux's Algorithm

• Example
Trémaux's Algorithm
Trémaux's Algorithm
Trémaux's Algorithm
Trémaux's Algorithm
Trémaux's Algorithm

  File:Tremaux_Maze_Solving_Algorithm.gif
Trémaux's Algorithm

- At the end:
  - All paths will be double marked and you will end up at the starting point
  - This means that you walked by the entry
Searching in Graphs

• We can use this idea for defining the first graph exploration algorithm.
  • Goal is to visit all vertices
  • We use a timer:
    • Starts out at 0
    • Incremented every time we do something
  • All nodes get marked with a
    • Discovery time: First time that we see the node
    • Finishing time: When we are done with the node
Breadth First Search

• Color a vertex
  • white: vertex has not yet been discovered
  • gray: vertex has been discovered, but still needs to be a base for exploration
  • black: vertex has been dealt with
Breadth First Search

```python
def bfs(G, s):
    for v in G.vertices:
        v.color = 'white'
        v.dist = float('inf')
        v.pred = None
    s.color = 'gray'
    s.dist = 0
    s.pred = None
    queue = []
    queue.append(s)
    while queue:
        u = queue.pop(0)
        for v in u.adjacency:
            if v.color == 'white':
                v.color = 'gray'
                v.dist = u.dist + 1
                v.pred = u
                queue.append(v)
        u.color = 'black'
```
Breadth First Search

- Example: s=A

```
A
|   |
C   D   E
|   |   |
| F |
|   |
G   H
```

Breadth First Search

- queue = {A}

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)
    u.color = 'black'
Breadth First Search

- queue = {}
- u = A

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist+1
            v.pred = u
            queue.append(v)

u.color = 'black'
Breadth First Search

- queue = {}
- u = A

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist+1
            v.pred = u
            queue.append(v)
    u.color = 'black'

- queue = {B,C,D}
Breadth First Search

- queue = {}
- u = A
- queue = {B,C,D}

```python
while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white':
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)
```

```
    u.color = 'black'
```
Breadth First Search

- queue = \{C, D\}
- u = B

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)
    u.color = 'black'

- queue = \{C, D\}
Breadth First Search

• queue = \{C, D\}

• \(u = B\)

• queue = \{C, D, E\}

while queue:
    \(u = \text{queue.pop(0)}\)
    for \(v\) in \(u\).adjacency:
        if \(v\).color == 'white'
            \(v\).color = 'gray'
            \(v\).dist = \(u\).dist+1
            \(v\).pred = \(u\)
            queue.append(v)
    \(u\).color = 'black'

• queue = \{C, D, E\}
Breadth First Search

- queue = \{C, D, E\}
- u = B

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)

u.color = 'black'

- queue = \{C, D, E\}
Breadth First Search

- queue = \{C, D, E\}
- u = C

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)
    u.color = 'black'

- queue = \{D, E\}
Breadth First Search

- $\text{queue} = \{D, E\}$
- $u = C$

while queue:
  $u = \text{queue.pop}(0)$
  for $v$ in $u$.adjacency:
    if $v$.color == 'white'
      $v$.color = 'gray'
      $v$.dist = $u$.dist+1
      $v$.pred = $u$
      queue.append($v$)
  $u$.color = 'black'

- $\text{queue} = \{D, E, F\}$
Breadth First Search

• queue = {D, E, F}
• u = C

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
        queue.append(v)

u.color = 'black'

• queue = {D, E, F}
Breadth First Search

• queue = {D, E, F}

• u = D

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)

    u.color = 'black'

• queue = {E, F}
Breadth First Search

- queue = \{E,F\}
- u = D

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist+1
            v.pred = u
            queue.append(v)
    u.color = 'black'

- queue = \{E, F, G\}
Breadth First Search

- queue = \{E,F,G\}
- u = D

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist+1
            v.pred = u
        queue.append(v)

u.color = 'black'

- queue = \{E, F, G\}
Breadth First Search

- queue = {E,F,G}
- u = E

```python
while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white':
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
        queue.append(v)
    u.color = 'black'
```

- queue = {F, G}
Breadth First Search

- queue = \{F, G\}
- u = E

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white':
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)
    u.color = 'black'

- queue = \{ F, G, H\}
Breadth First Search

- queue = \{F, G, H\}
- u = E

```
while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white':
            v.color = 'gray'
            v.dist = u.dist+1
            v.pred = u
            queue.append(v)
    u.color = 'black'
```

- queue = \{ F, G, H\}
Breadth First Search

- queue = \{G, H\}
- u = F

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist+1
            v.pred = u
            queue.append(v)
    u.color = 'black'

- queue = \{G, H\}
Breadth First Search

- queue = {G, H}
- u = F

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)

u.color = 'black'

- queue = {G, H}
Breadth First Search

- queue = {G, H}
- u = F
- while queue:
  - u = queue.pop(0)
  - for v in u.adjacency:
    - if v.color == 'white'
      - v.color = 'gray'
      - v.dist = u.dist+1
      - v.pred = u
      - queue.append(v)
  - u.color = 'black'
- queue = {G, H}
Breadth First Search

• queue = \{G, H\}

• u = G

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)

u.color = 'black'

• queue = \{H\}
Breadth First Search

- queue = \{G,H\}
- \( u = G \)

while queue:
  
  \( u = \text{queue.pop}(0) \)

  \( \text{for } v \text{ in } u.\text{adjacency}: \)

  \( \text{if } v.\text{color} == \text{'white'} \)

  \( v.\text{color} = \text{'gray'} \)

  \( v.\text{dist} = u.\text{dist}+1 \)

  \( v.\text{pred} = u \)

  \( \text{queue.append}(v) \)

  \( u.\text{color} = \text{'black'} \)

- queue = \{H\}
Breadth First Search

- queue = \{H\}
- u = G

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)
    u.color = 'black'

- queue = \{H\}
Breadth First Search

• queue = {}
• u = H

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)
    u.color = 'black'

• queue = {}
Breadth First Search

- queue = {}
- u = H

```
while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist+1
            v.pred = u
            queue.append(v)
    u.color = 'black'
```

- queue = {}
Breadth First Search

• queue = {}

• u = H

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist+1
            v.pred = u
            queue.append(v)
    u.color = 'black'

• queue = {}
Breadth First Search

- As you can see, BFS is just a version of Dijkstra's algorithm
- Distance calculates accurately the distance from the starting point
- The pred property allows us to generate a shortest path from the initial node
- We now prove these properties exactly
Breadth First Search

- Lemma: Let $G = (E, V)$ be an undirected or directed graph. Let $s \in V$ be an arbitrary vertex. Then for any edge $(u, v) \in E$
  - $\delta(s, v) \leq \delta(s, u) + 1$

- Recall: $\delta(a, b)$ is the length of a shortest path from $a$ to $b$
Breadth First Search

• Proof:
  • Assume first that $\delta(s, u) = \infty$, i.e. there is no path from $s$ to $u$
  • Then $\delta(s, v) \leq \infty = \delta(s, u) + 1$ regardless whether there is a path from $s$ to $v$. 


Breadth First Search

• Proof:

  • Next assume that $\delta(s, u) < \infty$, i.e. that there is a path from $s$ to $u$.

  • Extend this path to a path from $s$ to $v$.

  • This path has length $\delta(s, u) + 1$.

  • Then $\delta(s, v) = \min(\text{Length of a path from } s \text{ to } v)$

  • $\leq \text{Length of this path}$

  • $= \delta(s, u) + 1$
Breadth First Search

• Lemma: Let $G = (E, V)$ be an undirected or directed graph. Let $s \in V$ be an arbitrary vertex. Run BFS on $G$ and $s$. Then for every vertex $v \in V$, $v \cdot \text{dist} \geq \delta(s, v)$.

• This means that the calculated distance in BFS is at least as large as the actual distance
Breadth First Search

• Proof by induction on the number of enqueue operations

• Notice that v.dist is assigned just when we are about to enqueue it

while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
        -- > v.dist = u.dist+1
        v.pred = u
        queue.append(v)
    u.color = 'black'
Breadth First Search

• Induction Start:
  • When $s$ is enqueued all distance properties are infinity
    • with the exception of $s$ which has dist 0
  • At this point, for every vertex $v \in V$, $v \cdot \text{dist} \geq \delta(s, v)$
### Breadth First Search

- **Induction step:**
  - The value of the distance property only changes when we make the assignment just before enqueuing a white vector.

- **Induction hypothesis implies**
  \[ u \cdot \text{dist} \geq \delta(s, u) \]

- Therefore
  \[ v \cdot \text{dist} = u \cdot \text{dist} + 1 \geq \delta(s, u) + 1 \geq \delta(s, v) \]

```python
while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
        v.dist = u.dist+1
        v.pred = u
        queue.append(v)
    u.color = 'black'
```
Breadth First Search

• Afterwards, the vertex $v$ is no longer white and never changes its distance value
Breadth First Search

• We now need to see more closely how the algorithm works:
  • We can think of the queue as the boundary between black and white vertices that moves slowly away from $s$.
  • Lemma: If the queue has vertices $(v_1, v_2, \ldots, v_n)$ with $v_1$ being the head, then
    • $v_n \cdot \text{dist} \leq v_1 \cdot \text{dist} + 1$
      • and
    • $v_i \cdot \text{dist} \leq v_{i+1} \cdot \text{dist}$ for $i = 1, 2, \ldots, n - 1$
Breadth First Search

• Proof by induction on the number of queue operations

  • Initially, the queue has only $s$ in it, so the property certainly holds

  • The queue changes through enqueuing and dequeuing operations
Breadth First Search

- If the head $v_1$ is dequeued, $v_2$ becomes the new head.
  - (If there is no $v_2$ then the queue is empty, and the assertion holds vacuously)
  - Before dequeuing, $v_1 \cdot \text{dist} \leq v_2 \cdot \text{dist}$, therefore $v_n \cdot \text{dist} \leq v_1 \cdot \text{dist} + 1 \leq v_2 \cdot \text{dist}$
  - Therefore, the first inequality is true
  - The second assertion just loses the first inequality
Breadth First Search

• If a new element \( v_{n+1} \) is enqueued, we just dequeued a vertex \( u \) and are adding all white vertices adjacent to \( u \)

```python
while queue:
    u = queue.pop(0)
    for v in u.adjacency:
        if v.color == 'white'
            v.color = 'gray'
            v.dist = u.dist + 1
            v.pred = u
            queue.append(v)
    u.color = 'black'
```

• Therefore, \( v_{n+1}.dist = u.dist + 1 \).

• By induction hypothesis, \( u.dist \leq v_1.dist \) because \( u \) and \( v_1 \) were just in the same queue
Breadth First Search

- Therefore
  - $v_{r+1}.\text{dist} = u.\text{dist} + 1 \leq v_1.\text{dist} + 1$

- Proving the first assertion
Breadth First Search

- From the induction hypothesis, we also have
  - $v_n \cdot \text{dist} \leq u \cdot \text{dist} + 1$
  - which implies that
    - $v_n \cdot \text{dist} \leq u \cdot \text{dist} + 1 \leq v_{n+1} \cdot \text{dist}$
  - This is the only new part of the second assertion
Depth First Search

- Breadth first search uses a queue.
- In Python, a queue is a list to which you append and from which you pop.
- C++ and Java have libraries that implement queues.

```python
def bfs(G, s):
    for u in G.Vertices:
        u.color = "white"
        u.d = infty
        u.pred = Null
    s.color = "gray"
    s.d = 0
    s.pred = Null
    queue = Queue.queue()
    queue.enqueue(s)
    while queue:
        u = queue.head()
        for v in u.adjacency_list:
            if v.color == "white"
                v.color = "gray"
                v.d = u.d + 1
                v.pred = u
                queue.enqueue(v)
        u.color = "black"
```
Depth First Search

- Depth first search replaces the queue with a stack
  - This changes the behavior of the algorithm considerably
  - Remarkably, the resulting Depth First Search is the more important and interesting algorithm
Depth First Search

• Depth first search

• Version 1
  • Change queue into stack
  • Get rid of the distance

```python
def dfs(G, s):
    for u in G.Vertices:
        u.color = "white"
        u.d = infty
        u.pred = Null
    s.color="gray"
    s.pred = Null
    queue = Stack.Stack()
    stack.push(s)
    while stack:
        u = stack.pop()
        for v in u.adjacency_list:
            if v.color=="white"
                v.color = "gray"
                v.pred = u
                stack.push(v)
        u.color="black"
```
Depth First Search

• We add visiting times to our nodes:
  • Discovered time
    • When a node turns gray
  • Finished time
    • When a node turns black
• Because
  • some derived algorithms use it
  • in order to argue about DFS
• Whenever we change a node color, we increment a clock
Depth First Search

• Unlike BFS, a typical DFS will want to classify all nodes
  • Have a DFS_Visit(start_node) that starts in a node and visit what can be available
  • Have a DFS() function that uses the visit function repeatedly if necessary
Depth First Search

def dfs_visit(u):
    global clock
    clock += 1
    u.d = clock
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            v.pred = u
            dfs_visit(v)
    u.color = 'black'
    clock += 1
    u.f = clock
Depth First Search

defs(G):
    for vertex in G.V:
        vertex.color = 'white'
        vertex.pred = None
    global clock = 0
    for vertex in G.V:
        if vertex.color = 'white':
            dfs_visit(vertex)
Depth First Search

- Understanding the algorithm
  - The stack is hidden in the recursive call
  - We can unroll it
  - But need to be careful as something on the stack can be already found and processed via another route
Depth First Search

def dfs_visit(u):
    stack = [u]
    while stack:
        u = stack.pop()
        if u.color == 'white'
            for v in u.adjacency:
                if v.color == 'white':
                    stack.push(v)
Depth First Search

Start with C

OS stack is
dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black
Depth First Search

Start with C

OS stack
  \texttt{dfs\_visit(C)}

\begin{verbatim}
dfs\_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs\_visit(v)
  u.color = 'black'
\end{verbatim}

We set the clock to 1
We pick arbitrarily E from the adjacency list
Depth First Search

OS stack
  dfs_visit(E)
  dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

We call dfs_visit(E)
Depth First Search

OS stack
  dfs_visit(E)
  dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

u = E, set E's discovery time and set the predecessor link
Depth First Search

OS stack
  dfs_visit(E)
  dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

we pick v = H
Depth First Search

OS stack
- dfs_visit(E)
- dfs_visit(C)

dfs_visit(u):
  - u.color = 'gray'
  - for each v in u.adjacency:
    - if v.color == 'white'
      - dfs_visit(v)
  - u.color = 'black'

we pick v = H and call dfs_visit(H)
Depth First Search

OS stack
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

we pick v = H and call dfs_visit(H)
this colors H gray
Depth First Search

OS stack
dfs_visit(I)
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

we pick v = I and call dfs_visit(I)
this colors I gray
Depth First Search

OS stack
dfs_visit(L)
dfs_visit(I)
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

we pick v = L and call
dfs_visit(L)
this colors L gray
Depth First Search

OS stack
dfs_visit(L)
dfs_visit(I)
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

L has no vertices in the adjacency list
Therefore, we finally go to the last line
Depth First Search

OS stack
  dfs_visit(I)
  dfs_visit(H)
  dfs_visit(E)
  dfs_visit(C)

defs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

We finish dfs_visit(L)
and are back at the
execution of dfs_visit(I)
Depth First Search

OS stack

dfs_visit(J)
dfs_visit(I)
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white':
            dfs_visit(v)
    u.color = 'black'

We pick a white vertex reachable from I: J
Depth First Search

OS stack

dfs_visit(J)
dfs_visit(I)
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

There are no white nodes in the adjacency list of J
Depth First Search

OS stack
dfs_visit(I)
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

We close the call on J and are back to dfs_visit(I)
Depth First Search

OS stack
  dfs_visit(K)
  dfs_visit(I)
  dfs_visit(H)
  dfs_visit(E)
  dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

dfs_visit(I) now goes to K
Depth First Search

OS stack
dfs_visit(I)
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

dfs_visit(K) finishes
Depth First Search

OS stack
  dfs_visit(H)
  dfs_visit(E)
  dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

dfs_visit(I) runs again
but finds no white vertices,
so it finishes
Depth First Search

OS stack
dfs_visit(H)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

dfs_visit(I) runs again but finds no white vertices, so it finishes
Depth First Search

OS stack
  dfs_visit(E)
  dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

dfs_visit(H) runs again but finds no white vertices, so it finishes
Depth First Search

OS stack
  dfs_visit(E)
  dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

dfs_visit(E) runs again
Depth First Search

OS stack
  dfs_visit(F)
  dfs_visit(E)
  dfs_visit(C)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

Finds F
OS stack

dfs_visit(G)
dfs_visit(F)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

Finds G
Depth First Search

OS stack
dfs_visit(F)
dfs_visit(E)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

Nothing left in G
**Depth First Search**

OS stack
- `dfs_visit(E)`
- `dfs_visit(C)`

```
dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'
```

Nothing left in F
Depth First Search

OS stack
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

Finishing E
**Depth First Search**

OS stack

dfs_visit(D)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

Last white vector in the adjacency list of C is D
Depth First Search

OS stack

dfs_visit(D)
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

D has no white vertices in its adjacency list
Depth First Search

OS stack
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

We are back in C
Depth First Search

OS stack
dfs_visit(C)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

Now we close C
Depth First Search

OS stack

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

Now we close C
Depth First Search

- At this point, the original call to dfs_visit(C) is done
- However, since there are still white nodes left, we have to pick one of them and visit again.
- We pick A
Depth First Search

OS stack
dfs_visit(A)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

A is the only node in the stack
Depth First Search

OS stack
  dfs_visit(B)
  dfs_visit(A)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

We discover B from A
Depth First Search

OS stack
  dfs_visit(A)

dfs_visit(u):
  u.color = 'gray'
  for each v in u.adjacency:
    if v.color == 'white'
      dfs_visit(v)
  u.color = 'black'

We can finish B
Depth First Search

OS stack

dfs_visit(A)

dfs_visit(u):
    u.color = 'gray'
    for each v in u.adjacency:
        if v.color == 'white'
            dfs_visit(v)
    u.color = 'black'

We can finish A
Depth First Search

- Now we are done
  - The predecessor relationship has given us a nice set of trees — a "forest"
Depth First Search

- Runtime of algorithm
  - We look at all the elements of the adjacency lists
  - For each, we do constant work
  - But we also need to do some initial work for all vertices
  - Runtime is $\Theta(\max(|V|, |E|))$
Depth First Search

• Properties:
  • Parenthesis Theorem
    • If for two nodes
      • If \([u . d, u . f] \cap [v . d, v . f] = \emptyset\) then neither u and v are descendants in the predecessor forest
      • If \([u . d, u . f] \subset [v . d, v . f]\) then u is a descendant of v
      • If \([u . d, u . f] \supset [v . d, v . f]\) then v is a descendant of u
Depth First Search

• White Path Theorem
  
  • \( v \) is a descendant of \( u \) exactly if
  
  • At the time of discovery of \( u \) there is a path from \( u \) to \( v \) consisting entirely of white vertices
Depth First Search

- Classification of edges:
  - Tree edges are edges in the depth first tree
Depth First Search

- Back edges are edges that go from a descendant to an ancestor.

- Simple example:
  - Start in A, discover B, discover C
  - Edge from C to A is a back edge
Depth First Search

- Forward edges
- Edges connecting an ancestor to a descendant, but that are not in the tree
Depth First Search

- Cross Edges: anything else
  - Can be in the same tree or connecting different trees
Depth First Search

- If we look at an edge \((u,v)\) during depth first search for the first time
  - (In an undirected graph, we look at each edge twice)
    - If \(v\) is white: tree edge
    - If \(v\) is gray: back edge
    - If \(v\) is black: forward or cross edge
Depth First Search

• In a depth first search on an undirected graph, every edge is either a tree edge or a back edge

• Let \((u, v)\) be an edge and assume that \(u\) is discovered first:
  \[ u.d < v.d \]

• The algorithm discovers and finishes \(v\) before \(u\), so
  \[ u.f > v.f \]

• If DFS uses the edge \((u, v)\) from \(u\), then \(v\) is white, and \((u, v)\) becomes a tree edge

• If DFS uses the edge \((u, v)\) from \(v\), then \(u\) is gray at this moment and this becomes a back edge.