CSE 250: Depth- and Breadth-First Search

Lecture 20

Oct 18, 2023

Reminders

- PA2 released yesterday: Implement **Map Routing**
  1. Create an adjacency list (discussed today)
  2. Find a path from A to B with the fewest intersections
  3. Find a path from A to B with the shortest distance
- PA2 test cases due Sun, Oct 22 at 11:59 PM
- PA2 implementation due Sun, Nov 5 at 11:59 PM

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`: $O(M)$

Space Used: $O(N + M)$
(constant space per vertex, edge)

Adjacency List Summary

Starting with an edge list:

- Store a linked list of in/out edges with each vertex
- Store the linked list node for the in/out lists with each edge

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\texttt{deg}(v))$
- `incidentEdges`: $O(1)$ + $O(1)$ per `next()`
- `hasEdgeTo`: $O(\texttt{deg}(v))$

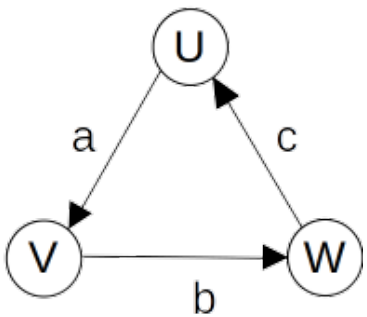
Space Used: $O(N + M)$
(constant space per vertex, edge)

hasEdgeTo

Can we cut `hasEdgeTo` down to $O(1)$?

The Adjacency Matrix Data Structure

Origin \ Destination	U	V	W
U	-	a	-
V	-	-	b
W	c	-	-



Edge List Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Space Used: $O(N^2)$

A few more definitions

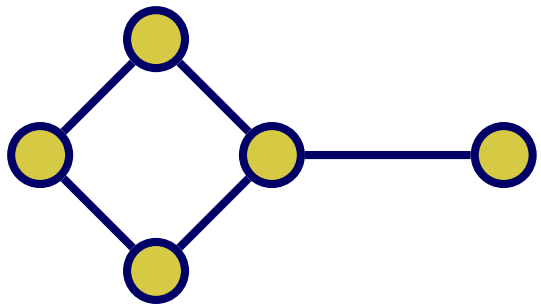
A graph is **connected** if...

- ...there is a path between every pair of vertices.

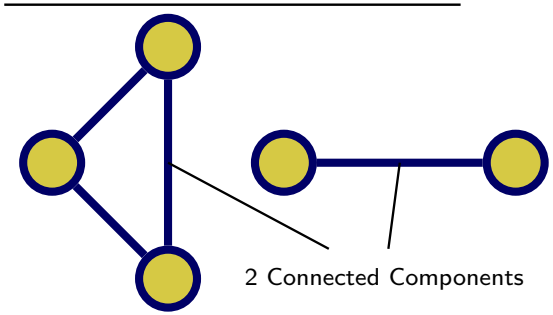
A **connected component** of G is a *maximal*, *connected* subgraph of G

- "*maximal*" means that adding any other vertices from G would break the connected property.
- Any subset of G's edges that makes the subgraph connected is fine.

Connected Graph



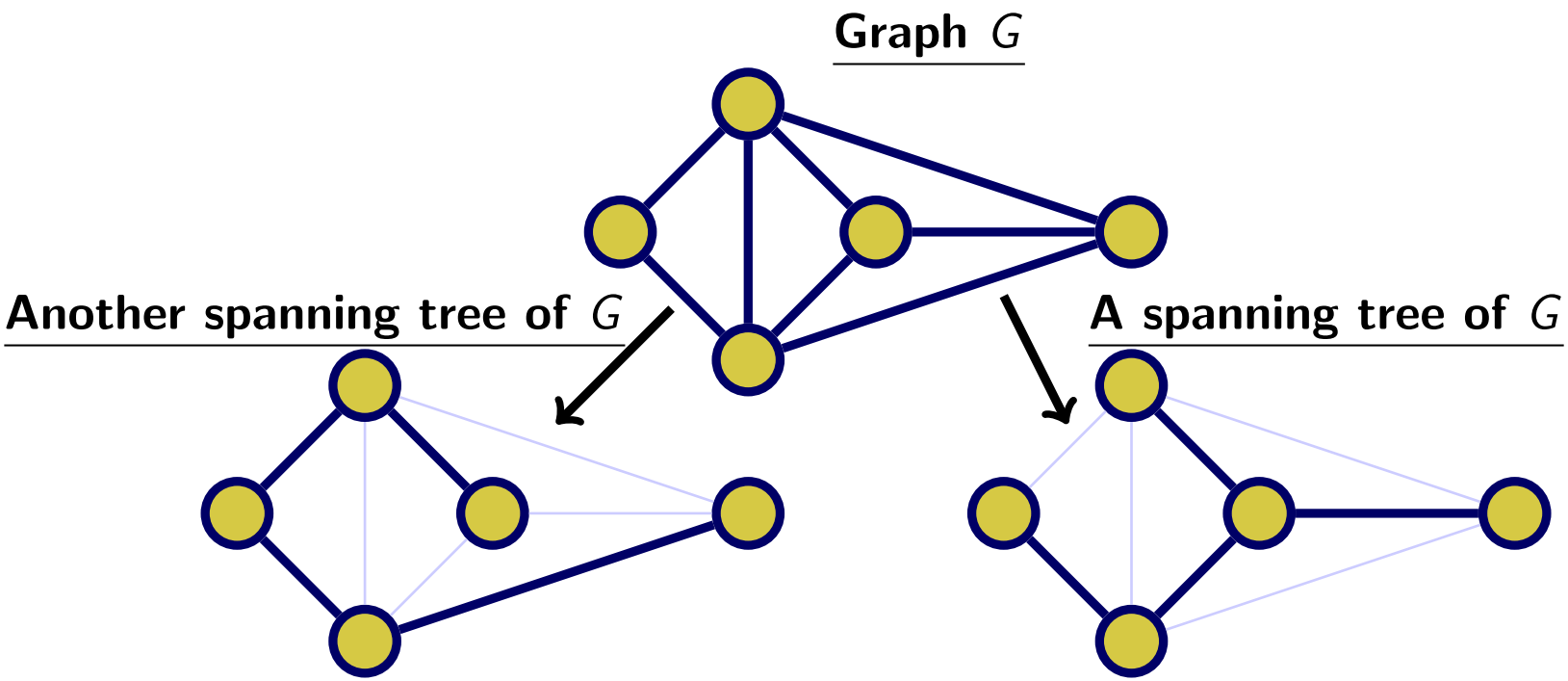
Disconnected Graph



A few more definitions...

A **spanning tree** of a connected graph G is:

- A spanning subgraph of G
- A tree

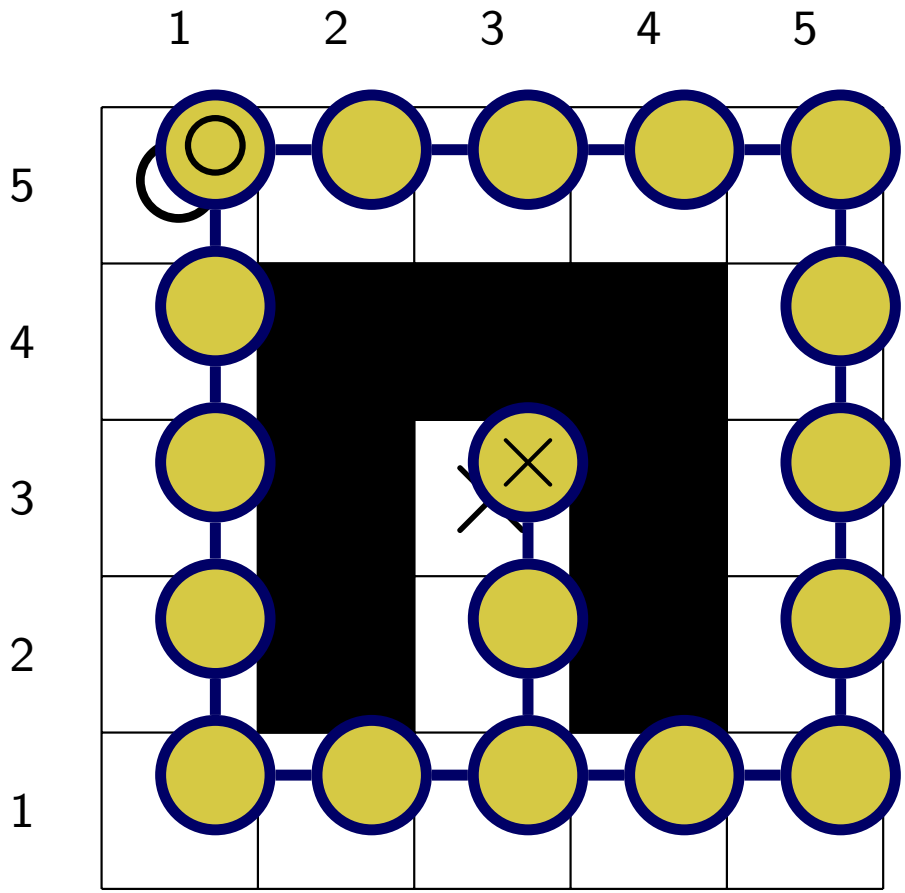


Note: A graph has multiple spanning trees (unless it is already a tree).

Back to connectivity...

So how about connectivity?

... or mazes?



Recall Mazes

- Start at a vertex
- If this is the target vertex, done!
- Push the next adjacent vertex (not filled, or already on the stack) onto the stack
- Explore from that vertex
- When that vertex is explored, pop it and move to the next adjacent vertex
- When everything is explored, return.

This is called Depth-First Search Contrast with Breadth-First Search

Depth First Search (DFS)

Primary Goals

- Visit every vertex in graph $G = (V, E)$.
- Construct a spanning tree for every connected component.
  - **Side Effect**: Compute connected components.
  - **Side Effect**: Compute a path between all connected vertices.
  - **Side Effect**: Determine if the graph is connected.
  - **Side Effect**: Identify any cycles (if they exist).
- Complete in time $O(N + M)$.

Depth First Search (DFS)

`DFS(G)`

Input

- Graph $G = (V, E)$

Output

- Label every edge as a:
  - Spanning Edge: Part of the spanning tree
  - Back Edge: Part of a cycle

Depth First Search (DFS)

`DFSOne(G, v)`

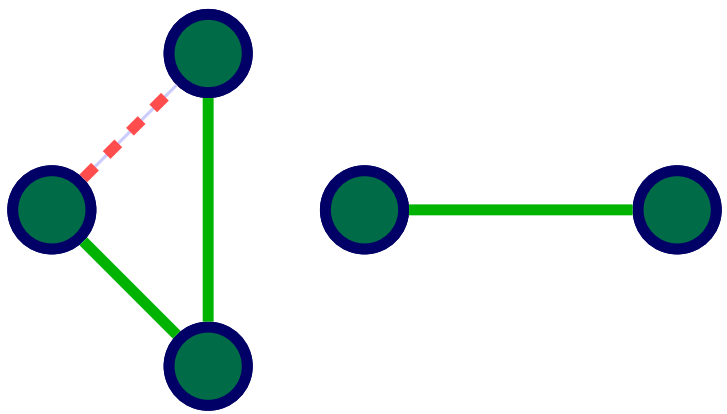
Input

- Graph $G = (V, E)$
- Start vertex $v \in V$

Output

- Label every edge in v's connected component as a:
  - Spanning Edge: Part of the spanning tree
  - Back Edge: Part of a cycle

DFS Example



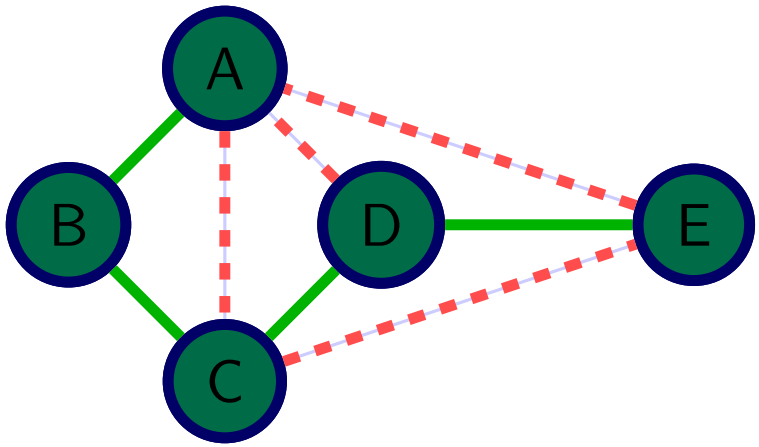
Depth First Search (DFS)

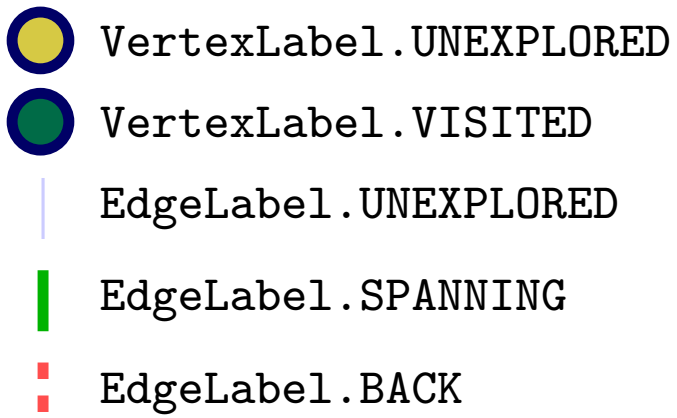
```
1  public enum VertexLabel { UNEXPLORED, VISITED }
2  public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4  public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5  {
6    for(Vertex v : graph.vertices)
7                  { v.setLabel(VertexLabel.UNEXPLORED); }
8    for(Edge e   : graph.edges)
9                  { e.setLabel(EdgeLabel.UNEXPLORED); }
10   for(Vertex v : graph.vertices)
11   {
12     if(v.getLabel() == VertexLabel.UNEXPLORED)
13     {
14       DFSOne(graph, v)
15     }
16   }
17 }
```

DFSOne

```
1  public void DFSOne(Graph<...> graph, Vertex start)
2  {
3    start.setLabel(VertexLabel.VISITED)
4
5    for( edge : start.getIncident() ) {
6      if(edge.getLabel == EdgeLabel.UNEXPLORED){
7        Vertex otherVertex = edge.getOppositeVertex(start);
8
9        if(otherVertex.getLabel == VertexLabel.UNEXPLORED){
10         edge.setLabel(EdgeLabel.SPANNING);
11         DFSOne(graph, otherVertex);
12       } else {
13         edge.setLabel(EdgeLabel.BACK);
14       }
15     }
16   }
17 }
```

Depth First Search (DFS)





Call Stack

`DFS(G)`

`DFSOne(G, A)` ($\rightarrow$ B, C, D, E)

`DFSOne(G, B)` ($\rightarrow$ A, C)

`DFSOne(G, C)` ($\rightarrow$ A, B, D, E)

`DFSOne(G, D)` ($\rightarrow$ A, C, E)

`DFSOne(G, E)` ($\rightarrow$ A, C, D)

DFS on Mazes

The DFS algorithm is like our stack-based maze solver

- Mark each grid square with `VISITED`
- Mark each path with `SPANNING`
- Only visit each vertex once.

DFS won't necessarily find the shortest paths.

DFS

What's the complexity?

Depth First Search (DFS)

```
1  public enum VertexLabel { UNEXPLORED, VISITED }
2  public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4  public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5  {
6    for(Vertex v : graph.vertices)
7                  { v.setLabel(VertexLabel.UNEXPLORED); }
8    for(Edge e   : graph.edges)
9                  { e.setLabel(EdgeLabel.UNEXPLORED); }
10   for(Vertex v : graph.vertices)
11   {
12     if(v.getLabel() == VertexLabel.UNEXPLORED)
13     {
14       DFSOne(graph, v)
15     }
16   }
17 }
```

Depth First Search (DFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     O(N)
7
8     for(Edge e    : graph.edges)
9                    { e.setLabel(EdgeLabel.UNEXPLORED); }
10    for(Vertex v : graph.vertices)
11    {
12      if(v.getLabel() == VertexLabel.UNEXPLORED)
13      {
14        DFSOne(graph, v)
15      }
16    }
17  }
```

Depth First Search (DFS)

```
1  public enum VertexLabel { UNEXPLORED, VISITED }
2  public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4  public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5  {
6    O(N)
7
8    O(M)
9
10   for(Vertex v : graph.vertices)
11   {
12     if(v.getLabel() == VertexLabel.UNEXPLORED)
13     {
14       DFSOne(graph, v)
15     }
16   }
17 }
```

Depth First Search (DFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     O(N)
7
8     O(M)
9
10    O(N) times
11    {
12      if(v.getLabel() == VertexLabel.UNEXPLORED)
13      {
14        DFSOne(graph, v)
15      }
16    }
17  }
```

Depth First Search (DFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void DFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     O(N)
7
8     O(M)
9
10    O(N) times
11    {
12      O(1)
13      {
14        DFSOne(graph, v)
15      } else {
16        O(1)
17      }
18    }
19  }
```

Depth First Search (DFS)

Observation: `DFSOne` is called on each vertex *at most* once.

- If `v`'s label is `VISITED`, both `DFSOne` and `DFS` skip it

$$O(N) \text{ calls to } \texttt{DFSOne}$$

What's the runtime of `DFSOne` **excluding recursive calls**?

DFSOne

```
1  public void DFSOne(Graph<...> graph, Vertex start)
2  {
3    start.setLabel(VertexLabel.VISITED)
4
5    for( edge : start.getIncident() ) {
6      if(edge.getLabel == EdgeLabel.UNEXPLORED){
7        Vertex otherVertex = edge.getOppositeVertex(start);
8
9        if(otherVertex.getLabel == VertexLabel.UNEXPLORED){
10         edge.setLabel(EdgeLabel.SPANNING);
11         DFSOne(graph, otherVertex);
12       } else {
13         edge.setLabel(EdgeLabel.BACK);
14       }
15     }
16   }
17 }
```

DFSOne

```
1   public void DFSOne(Graph<...> graph, Vertex start)
2   {
3     O(1)
4
5     O(deg(start)) times {
6       O(1) {
7         O(1)
8
9         O(1) {
10          O(1)
11          DFSOne(graph, otherVertex);
12        } else {
13          O(1)
14        }
15      }
16    }
17  }
```

Depth First Search (DFS)

What's the runtime of `DFSOne` **excluding recursive calls**?

$$O(\texttt{deg}(\texttt{start}))$$

Depth First Search (DFS)

What's the sum over all calls to `DFSOne`

$$\sum_{v \in V} O(\deg(v))$$

$$= O\left(\sum_{v \in V} \deg(v)\right)$$

$$= O(2M) = O(M)$$

To summarize...

Mark Vertices `UNVISITED`	$O(N)$
Mark Edges `UNVISITED`	$O(M)$
`DFS` Vertex Loop	$O(N)$
All calls to `DFSOne`	$O(M)$
	$O(N+M)$

Breadth First Search (BFS)

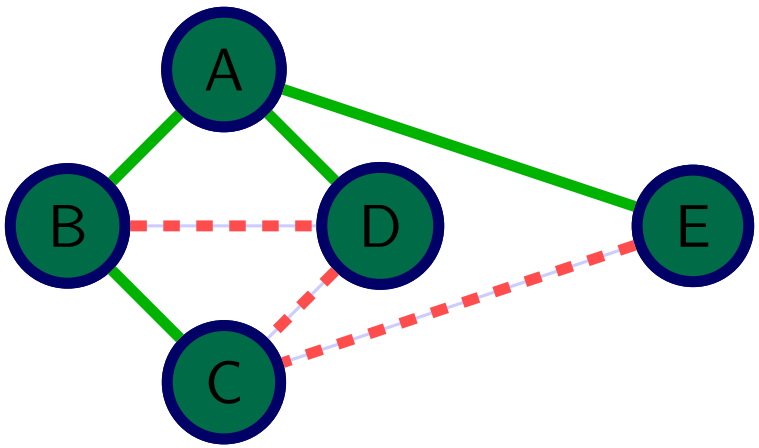
Primary Goals

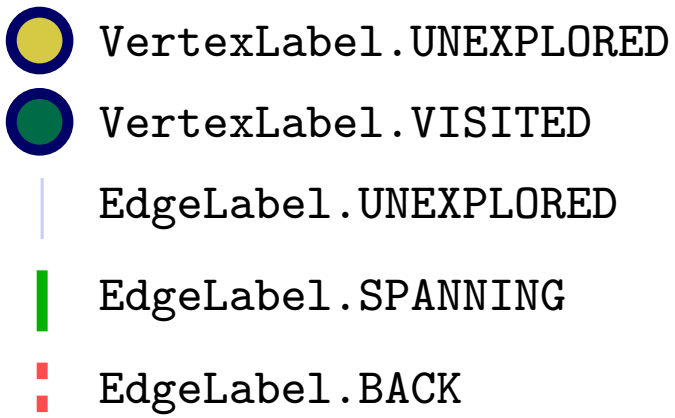
- Visit every vertex in graph $G = (V, E)$.
- Construct a spanning tree for every connected component **in increasing order of distance (steps) from the start vertex**.
  - **Side Effect**: Compute connected components.
  - **Side Effect**: Compute a path between all connected vertices.
  - **Side Effect**: Determine if the graph is connected.
  - **Side Effect**: Identify any cycles (if they exist).
- Complete in time $O(N + M)$ **with memory overhead** $O(|V|)$.

Breadth First Search (BFS)

```
1   public enum VertexLabel { UNEXPLORED, VISITED }
2   public enum EdgeLabel { UNEXPLORED, SPANNING, BACK }
3
4   public void BFS(Graph<VertexLabel,EdgeLabel> graph)
5   {
6     O(N)
7
8     O(M)
9
10    for(Vertex v : graph.vertices)
11    {
12      if(v.getLabel() == VertexLabel.UNEXPLORED)
13      {
14        BFSOne(graph, v)
15      }
16    }
17  }
```

Breadth First Search (BFS)





Call Stack

```
BFS(G)
BFSOne(G, A)
```

Work Queue

```
A
B
D
E
C
```

BFSOne

```
1  public void BFSOne(Graph<V, E> graph, start: Vertex)
2  {
3    Queue<Vertex> todos = new Queue<>();
4    todos.add(start)
5    start.setLabel(VertexLabel.visited)
6    while( !work.isEmpty() ){
7      Vertex current = todos.remove();
8      for(edge : current.getIncident()){
9        if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10         Vertex opposite = edge.getOpposite(current);
11         if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12           work.add(opposite);
13           opposite.setLabel(VertexLabel.VISITED);
14           edge.setLabel(EdgeLabel.SPANNING);
15         } else {
16           edge.setLabel(EdgeLabel.BACK);
17 } } } } }
```

BFSOne

```
1  public void BFSOne(Graph<V, E> graph, start: Vertex)
2  {
3    O(1)
4    O(1)
5    O(1)
6    while( !work.isEmpty() ){
7      Vertex current = todos.remove();
8      for(edge : current.getIncident()){
9        if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10         Vertex opposite = edge.getOpposite(current);
11         if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12           work.add(opposite);
13           opposite.setLabel(VertexLabel.VISITED);
14           edge.setLabel(EdgeLabel.SPANNING);
15         } else {
16           edge.setLabel(EdgeLabel.BACK);
17 } } } } }
```

BFSOne

```
1  public void BFSOne(Graph<V, E> graph, start: Vertex)
2  {
3    O(1)
4    O(1)
5    O(1)
6    while( !work.isEmpty() ){
7      O(1)
8      for(edge : current.getIncident()){
9        if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10         Vertex opposite = edge.getOpposite(current);
11         if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12           work.add(opposite);
13           opposite.setLabel(VertexLabel.VISITED);
14           edge.setLabel(EdgeLabel.SPANNING);
15         } else {
16           edge.setLabel(EdgeLabel.BACK);
17 } } } } }
```

BFSOne

```
1  public void BFSOne(Graph<V, E> graph, start: Vertex)
2  {
3    O(1)
4    O(1)
5    O(1)
6    while( !work.isEmpty() ){
7      O(1)
8      O(deg(v)) times {
9        if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10         Vertex opposite = edge.getOpposite(current);
11         if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12           work.add(opposite);
13           opposite.setLabel(VertexLabel.VISITED);
14           edge.setLabel(EdgeLabel.SPANNING);
15         } else {
16           edge.setLabel(EdgeLabel.BACK);
17 } } } } }
```

BFSOne

```
1  public void BFSOne(Graph<V, E> graph, start: Vertex)
2  {
3    O(1)
4    O(1)
5    O(1)
6    while( !work.isEmpty() ){
7      O(1)
8      O(deg(v)) times {
9        if(O(1)){
10         O(1)
11         if(O(1)){
12           O(1)
13           O(1)
14           O(1)
15         } else {
16           O(1)
17 } } } } }
```

BFSOne

```
1  public void BFSOne(Graph<V, E> graph, start: Vertex)
2  {
3    O(1)
4    while( !work.isEmpty() ){
5      O(deg(v))
6    }
7  }
```

BFSOne

Each node is enqueued at exactly once!
Over all calls to BFSOne:

$$\sum_{v \in V} O(\texttt{deg}(v))$$

$$= O(\sum_{v \in V} \texttt{deg}(v))$$

$$= O(2M)$$

$$= O(M)$$

To summarize...

Mark Vertices `UNVISITED`	$O(N)$
Mark Edges `UNVISITED`	$O(M)$
Add each vertex to the queue	$O(N)$
Process all vertices	$O(M)$
	$O(N+M)$

Other questions

What if we want the path?

Todo

```
1  public class Todo
2  {
3    Vertex vertex;
4    List<Vertex> path;
5  }
```

BFSOne - Pathfinding

```
1   public void BFSOne(Graph<V, E> graph, start: Vertex)
2   {
3     Queue<Path> todos = new Queue<>();                          ←
4     todos.add(new Todo(start, new List()));                     ←
5     start.setLabel(VertexLabel.visited)
6     while( !work.isEmpty() ){
7       Todo curr = todos.remove();                               ←
8       for(edge : curr.vertex.getIncident()){
9         if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10          Vertex opposite = edge.getOpposite(curr.vertex);
11          if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12            work.add(new Todo(opposite,
13              curr.path.clone().add(opposite));                 ←!!!
14            /* ... */
15          } else {
16            edge.setLabel(EdgeLabel.BACK);
17  } } } } }
```

BFSOne - Pathfinding

Problem: "Copying" the path can be $O(N)$

Idea: Store the 'previous hop' for each vertex.

BFSOne - Pathfinding

```
1   public void BFSOne(Graph<V, E> graph, start: Vertex)
2   {
3     Queue<Vertex> todos = new Queue<>();
4     todos.add(start)
5     start.setLabel(VertexLabel.visited)
6     while( !work.isEmpty() ){
7       Vertex current = todos.remove();
8       for(edge : current.getIncident()){
9         if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10          Vertex opposite = edge.getOpposite(current);
11          if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12            /* ... */
13            prevHops.put(opposite, current);
14          } else {
15            edge.setLabel(EdgeLabel.BACK);
16  } } } } }
```

BFSOne - Pathfinding

To compute the path from A to B:

1. B is the last element of the path.
2. `prevHops`$[B]$ is the second-to last element of the path.
3. `prevHops`$[$`prevHops`$[B]]$ is the third-to last element of the path.
4. ...
5. A is the first element of the path.

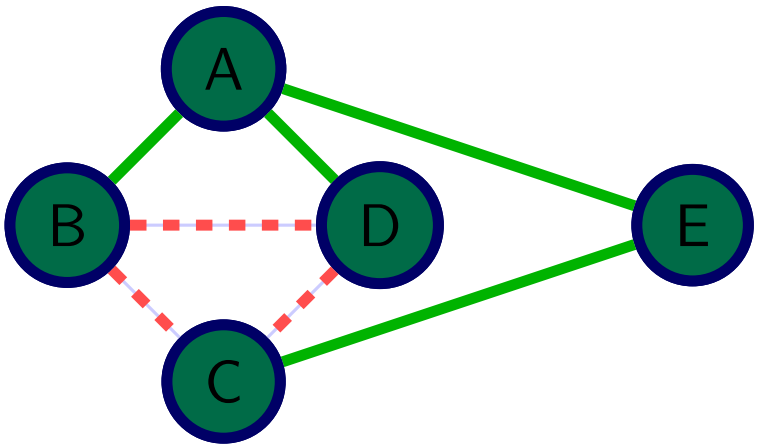
How about

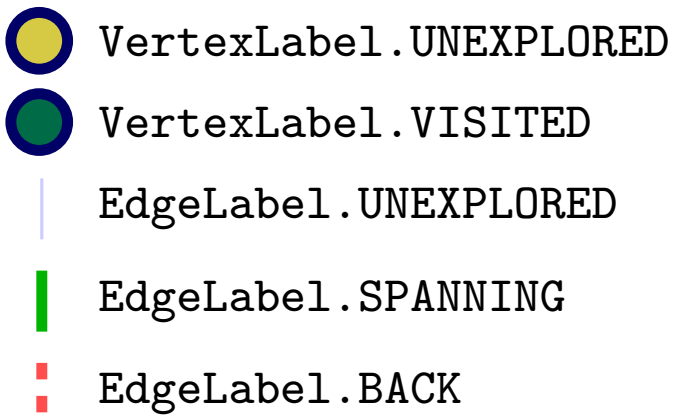
What happens if we use a `Stack` instead of a `Queue`?

Not Quite BFS

```
1  public void NotQuiteBFSOne(Graph<V, E> graph, start: Vertex)
2  {
3    Stack<Vertex> todos = new Stack<>();
4    todos.add(start)
5    start.setLabel(VertexLabel.visited)
6    while( !work.isEmpty() ){
7      Vertex curr = todos.remove();
8      for(edge : curr.getIncident()){
9        if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10         Vertex opposite = edge.getOpposite(curr);
11         if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12           work.add(opposite);
13           opposite.setLabel(VertexLabel.VISITED);
14           edge.setLabel(EdgeLabel.SPANNING);
15         } else {
16           edge.setLabel(EdgeLabel.BACK);
17 } } } } }
```

Not quite BFS





Call Stack

```
NotQuiteBFS(G)
NotQuiteBFSOne(G, A)
```

Work Stack

```
A
B
D
E
C
```

DFS

DFS is BFS with a Stack