Lecture 23 (Graphs 2)

BFS, DFS and Implementations

CS61B, Spring 2024 @ UC Berkeley

Slides credit: Josh Hug

Princeton Graphs API

Lecture 23, CS61B, Spring 2024

The All Paths Problem

- **Princeton Graphs API**
- DepthFirstPaths Implementation
- The Adjacency List
- DepthFirstPaths Runtime

The All Shortest Paths Problem

- BreadthFirstPaths
- BreadthFirstPaths Implementation

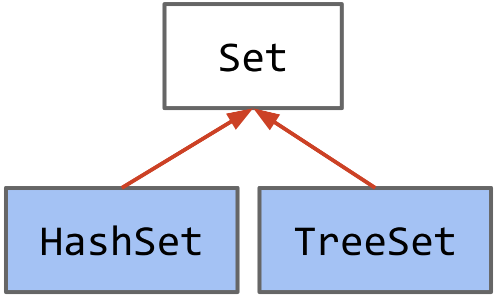
Graph Implementations and Runtime

Project 2B Note

Graph Representations

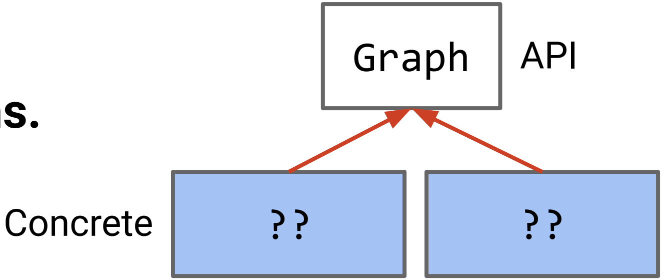
To Implement a graph algorithm like DepthFirstPaths, we need:

- **An API** (Application Programming Interface) for graphs.
    - For our purposes today, these are our Graph methods, including their signatures and behaviors.
    - Defines how Graph client programmers must think.
- A concrete data structure to represent our graphs.



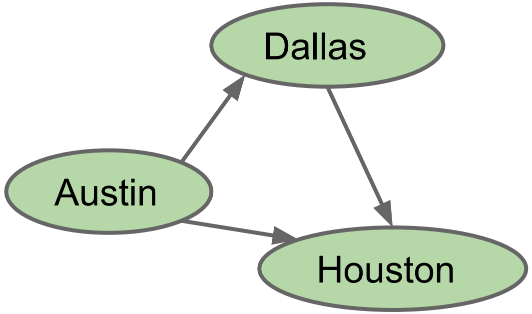
Our choices can have profound implications on:

- **Runtime.**
- **Memory usage.**
- **Difficulty of implementing various graph algorithms.**

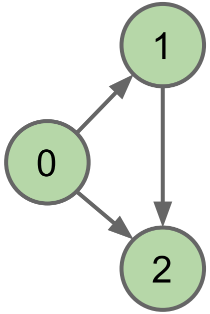


Graph API Decision #1: Integer Vertices

Common convention: Number nodes irrespective of “label”, and use number throughout the graph implementation. To lookup a vertex by label, you’d need to use a Map<Label, Integer>.



Intended graph.



Map<String, Integer>:
Austin: 0
Dallas: 1
Houston: 2

How you’d build it.

Graph API

The Graph API from Princeton's algorithms textbook.

```
public class Graph {
  public Graph(int V):                   Create empty graph with v vertices
  public void addEdge(int v, int w): add an edge v-w
  Iterable<Integer> adj(int v):          vertices adjacent to v
  int V():                               number of vertices
  int E():                               number of edges
...
```

Some features:

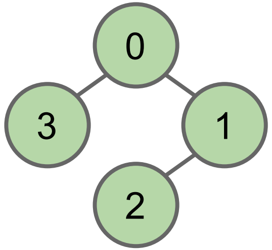
- Number of vertices must be specified in advance.
- Does not support weights (labels) on nodes or edges.
- Has no method for getting the number of edges for a vertex (i.e. its degree).

Graph API

The Graph API from our optional textbook.

```
public class Graph {
  public Graph(int V):                  Create empty graph with v vertices
  public void addEdge(int v, int w): add an edge v-w
  Iterable<Integer> adj(int v):         vertices adjacent to v
  int V():                              number of vertices
  int E():                              number of edges
...
```

Example client:



degree(G, 1) = 2

```
/** degree of vertex v in graph G */
public static int degree(Graph G, int v) {
    int degree = 0;
    for (int w : G.adj(v)) {
        degree += 1;
    }
    return degree; }
```

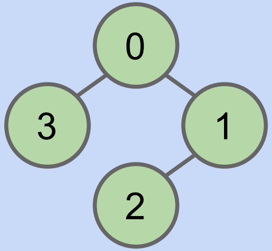
(degree = # edges)

Graph API

The Graph API from our optional textbook.

```
public class Graph {
  public Graph(int V):                 Create empty graph with v vertices
  public void addEdge(int v, int w): add an edge v-w
  Iterable<Integer> adj(int v):        vertices adjacent to v
  int V():                             number of vertices
  int E():                             number of edges
...
```

Challenge: Try to write a client method called print that prints out a graph.




```
public static void print(Graph G) {
    ???
}
```

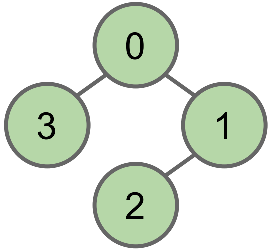
```
$ java printDemo
0 - 1
0 - 3
1 - 0
1 - 2
2 - 1
3 - 0
```

Graph API

The Graph API from our optional textbook.

```
public class Graph {
  public Graph(int V):                 Create empty graph with v vertices
  public void addEdge(int v, int w):   add an edge v-w
  Iterable<Integer> adj(int v):        vertices adjacent to v
  int V():                             number of vertices
  int E():                             number of edges
...
```

Print client:




```
public static void print(Graph G) {
    for (int v = 0; v < G.V(); v += 1) {
        for (int w : G.adj(v)) {
           System.out.println(v + “-” + w);
         }
    }
}
```

```
$ java printDemo
0 - 1
0 - 3
1 - 0
1 - 2
2 - 1
3 - 0
```

Graph API and DepthFirstPaths

Our choice of Graph API has deep implications on the implementation of DepthFirstPaths, BreadthFirstPaths, print, and other graph “clients”.

- Our choice of concrete implementation will affect runtimes and memory usage.

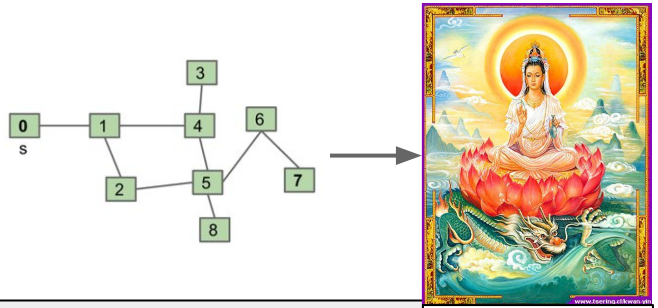
DepthFirstPaths Implementation

Lecture 23, CS61B, Spring 2024

Depth First Search Implementation

Common design pattern in graph algorithms: Decouple type from processing algorithm.

- Create a graph object.
- Pass the graph to a graph-processing method (or constructor) in a client class.
- Query the client class for information.



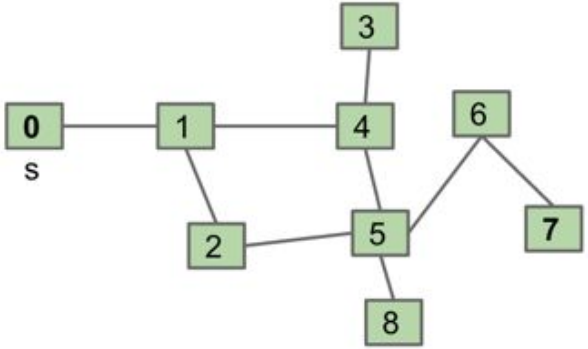
Paths.java

```
public class Paths {
    public Paths(Graph G, int s):    Find all paths from G
    boolean hasPathTo(int v):        is there a path from s to v?
    Iterable<Integer> pathTo(int v): path from s to v (if any)
}
```

Example Usage

Start by calling: `Paths P = new Paths(G, 0);`

- `P.hasPathTo(3); //returns true`
- `P.pathTo(3); //returns {0, 1, 4, 3}`



Paths.java

```
public class Paths {
    public Paths(Graph G, int s):    Find all paths from G
    boolean hasPathTo(int v):        is there a path from s to v?
    Iterable<Integer> pathTo(int v): path from s to v (if any)
}
```

DepthFirstPaths

Let's review DepthFirstPaths by running the demo from last lecture again.

Will then discuss:

- Implementation.
- Runtime.

DepthFirstPaths, Recursive Implementation

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;

  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  }
    ...
}
```

marked[v] is true iff v connected to s

edgeTo[v] is previous vertex on path from s to v

not shown: data structure initialization

find vertices connected to s.

recursive routine does the work and stores results in an easy to query manner!

Question to ponder: How would we write pathTo(v) and hasPathTo(v)?

- Answer on next slide.

DepthFirstPaths, Recursive Implementation

```java
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  ...
  public Iterable<Integer> pathTo(int v) {
    if (!hasPathTo(v)) return null;
    List<Integer> path = new ArrayList<>();
    for (int x = v; x != s; x = edgeTo[x]) {
      path.add(x);
    }
    path.add(s);
    Collections.reverse(path);
    return path;
  }

  public boolean hasPathTo(int v) {
    return marked[v];
  }
}
```

marked[v] is true iff v connected to s

edgeTo[v] is previous vertex on path from s to v

To analyze the runtime, we need to create a concrete Graph Implementation.

The Adjacency List

Lecture 23, CS61B, Spring 2024

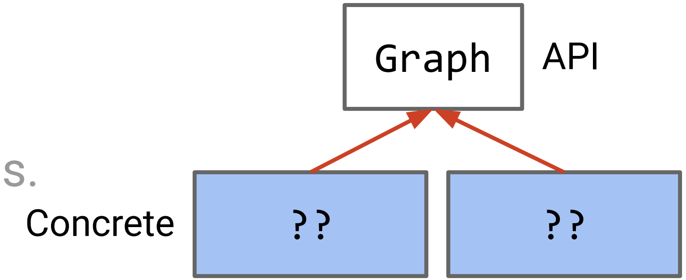
Graph Representations

To Implement our graph algorithms like BreadthFirstPaths and DepthFirstPaths, we need:

- An API (Application Programming Interface) for graphs.
  - For our purposes today, these are our Graph methods, including their signatures and behaviors.
  - Defines how Graph client programmers must think.
- An **underlying data structure to represent our graphs**.

Our choices can have profound implications on:

- **Runtime.**
- **Memory usage.**
- Difficulty of implementing various graph algorithms.



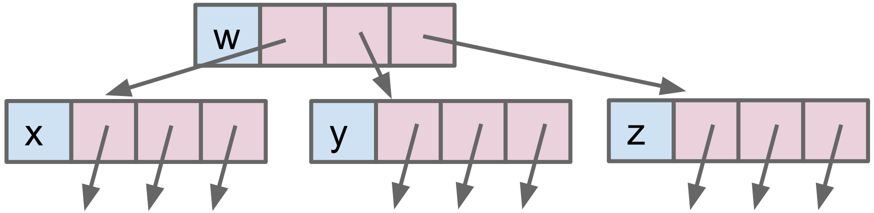
Graph Representations

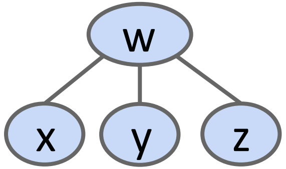
Just as we saw with trees, there are many possible implementations we could choose for our graphs.

Let's review briefly some representations we saw for trees.

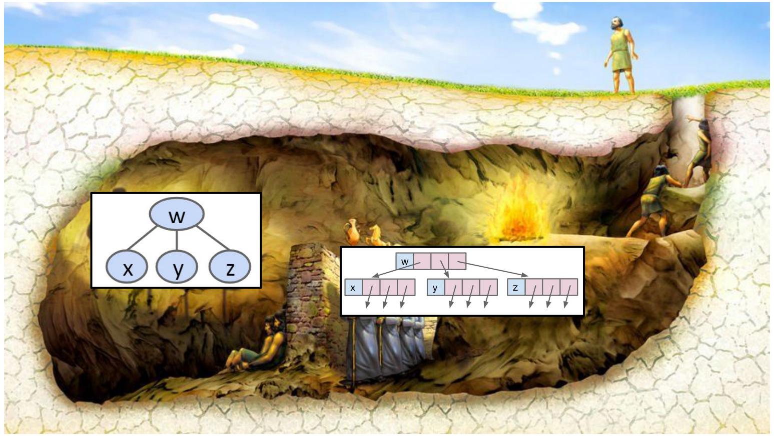
Tree Representations

We've seen many ways to represent the same tree. Example: 1a.





1a: Fixed Number of Links (One Per Child)



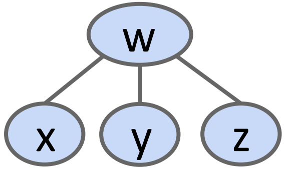
Tree Representations

We’ve seen many ways to represent the same tree. Example: 3.

`Key[] keys`

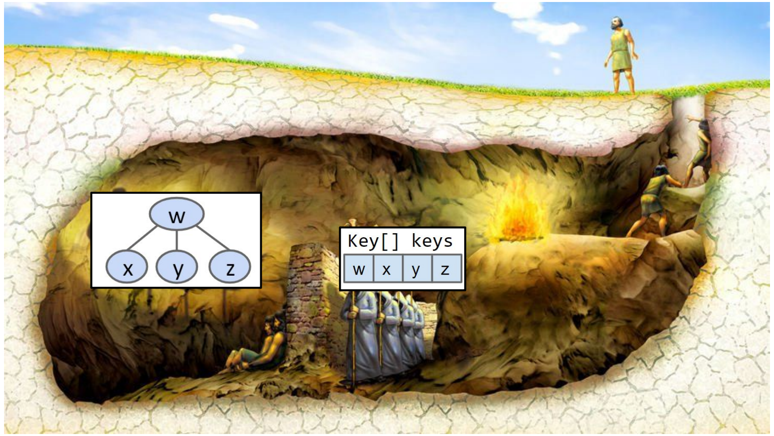
w	x	y	z

3: Array of Keys



Uses much less memory and operations will tend to be faster.

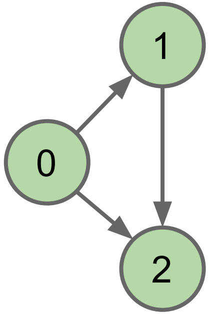
… but only works for complete trees.



Graph Representations

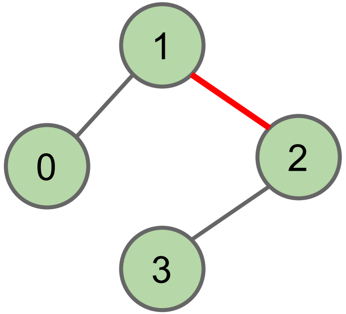
Graph Representation 1: Adjacency Matrix.

s \ t	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0



For undirected graph: Each edge is represented twice in the matrix. Simplicity at the expense of space.

v \ w	0	1	2	3
0	0	1	0	0
1	1	0	**1**	0
2	0	**1**	0	1
3	0	0	1	0

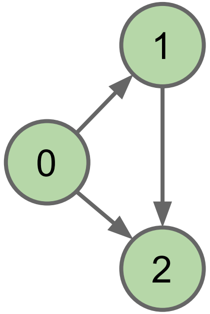


More Graph Representations

Representation 2: Edge Sets: Collection of all edges.

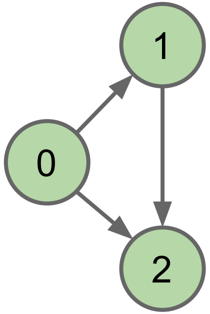
- Example: HashSet<Edge>, where each Edge is a pair of ints.

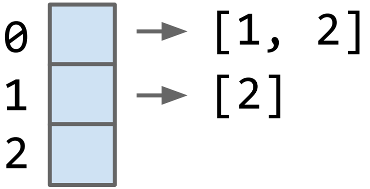
```
{(0, 1), (0, 2), (1, 2)}
```



Representation 3: Adjacency lists.

- Common approach: Maintain array of lists indexed by vertex number.
- Most popular approach for representing graphs.
  - Efficient when graphs are “sparse” (not too many edges).





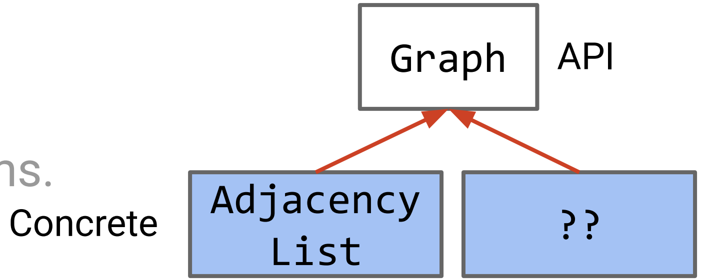
Graph Representations

To Implement our graph algorithms like BreadthFirstPaths and DepthFirstPaths, we need:

- An API (Application Programming Interface) for graphs.
  - For our purposes today, these are our Graph methods, including their signatures and behaviors.
  - Defines how Graph client programmers must think.
- An **underlying data structure to represent our graphs**.

Our choices can have profound implications on:

- **Runtime.**
- **Memory usage.**
- Difficulty of implementing various graph algorithms.



Graph Printing Runtime: http://yellkey.com/second

What is the order of growth of the running time of the print client if the graph uses an ***adjacency-list*** representation, where V is the number of vertices, and E is the total number of edges?

A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(V*E)$

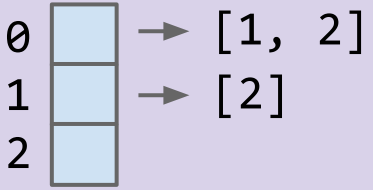
```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

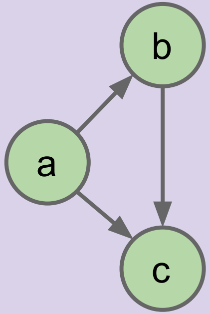
Runtime to iterate over v's neighbors?

-

How many vertices do we consider?

-





Graph Printing Runtime: http://yellkey.com/second

What is the order of growth of the running time of the print client if the graph uses an ***adjacency-list*** representation, where V is the number of vertices, and E is the total number of edges?

A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(V^*E)$

```java
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

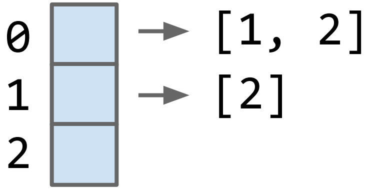
Best case: $\Theta(V)$ Worst case: $\Theta(V^2)$

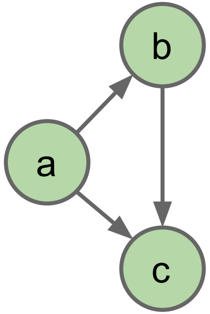
Runtime to iterate over v's neighbors? List can be between 1 and V items.

- $\Omega(1)$, $O(V)$.

How many vertices do we consider?

- V.





Graph Printing Runtime: http://yellkey.com/support

What is the order of growth of the running time of the print client if the graph uses an ***adjacency-list*** representation, where V is the number of vertices, and E is the total number of edges?

A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(V*E)$

? ? ? ? ?

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

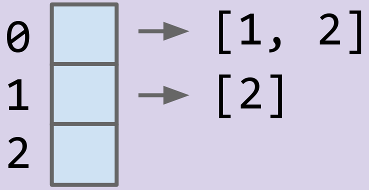
Best case: $\Theta(V)$ **Worst case: $\Theta(V^2)$**

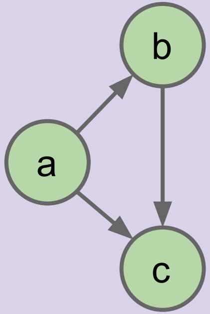
Runtime to iterate over v's neighbors? List can be between 1 and V items.

- $\Omega(1)$, $O(V)$.

How many vertices do we consider?

- V.





Graph Printing Runtime: http://yellkey.com/support

What is the order of growth of the running time of the print client if the graph uses an ***adjacency-list*** representation, where V is the number of vertices, and E is the total number of edges?

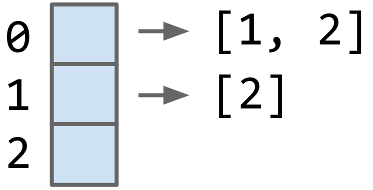
A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(V*E)$

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + “-” + w);
    }
}
```

Best case: $\Theta(V)$ Worst case: $\Theta(V^2)$

Cost model in this analysis is the sum of:
- v +=1 operations
- println calls

All cases: $\Theta(V + E)$
- v+=1 happens V times.
- Print happens 2E times.



Θ(V + E) Interpretation

Runtime: $\Theta(V + E)$

V is total number of vertices.

E is total number of edges in the entire graph.

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

How to interpret: No matter what "family" of increasingly complex graphs we generate, as V and E grow, the runtime will always grow exactly as $\Theta(V + E)$.

- Example shape 1: Very sparse graph where E grows very slowly, e.g. every vertex is connected to its square: 2 - 4, 3 - 9, 4 - 16, 5 - 25, etc.
  - E is $\Theta(\text{sqrt}(V))$. Runtime is $\Theta(V + \text{sqrt}(V))$, which is just $\Theta(V)$.
- Example shape 2: Very dense graph where E grows very quickly, e.g. every vertex connected to every other.
  - E is $\Theta(V^2)$. Runtime is $\Theta(V + V^2)$, which is just $\Theta(V^2)$.

More Θ(V + E) Interpretation

Runtime: $\Theta(V + E)$

V is total number of vertices.

E is total number of edges in the entire graph.

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

$\Theta(V + E)$ is the equivalent of $\Theta(\max(V, E))$.

- Both are technically correct, but the sum is used more often.

Note: We never formally defined asymptotics on multiple variables, and it turns out to be somewhat poorly defined.

- See: https://people.cs.ksu.edu/~rhowell/asymptotic.pdf

DepthFirstPaths Runtime

Lecture 23, CS61B, Spring 2024

The All Paths Problem

- Princeton Graphs API
- DepthFirstPaths Implementation
- The Adjacency List
- **DepthFirstPaths Runtime**

The All Shortest Paths Problem

- BreadthFirstPaths
- BreadthFirstPaths Implementation

Graph Implementations and Runtime

Project 2B Note

DepthFirstPaths, Recursive Implementation

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;

  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  }
    ...
}
```

marked[v] is true iff v connected to s

edgeTo[v] is previous vertex on path from s to v

not shown: data structure initialization

find vertices connected to s.

recursive routine does the work and stores results in an easy to query manner!

Question to ponder: How would we write pathTo(v) and hasPathTo(v)?

- Answer on next slide.

DepthFirstPaths, Recursive Implementation

```java
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  ...
  public Iterable<Integer> pathTo(int v) {
    if (!hasPathTo(v)) return null;
    List<Integer> path = new ArrayList<>();
    for (int x = v; x != s; x = edgeTo[x]) {
      path.add(x);
    }
    path.add(s);
    Collections.reverse(path);
    return path;
  }

  public boolean hasPathTo(int v) {
    return marked[v];
  }
}
```

marked[v] is true iff v connected to s

edgeTo[v] is previous vertex on path from s to v

Runtime for DepthFirstPaths

Give a O bound for the runtime for the DepthFirstPaths constructor.

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  } ...
}
```

Assume graph uses adjacency list!

Runtime for DepthFirstPaths

Give a O bound for the runtime for the DepthFirstPaths constructor.

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  } ...
}
```

Assume graph uses adjacency list!

vertex visits (no more than V calls)

edge considerations, each constant time (no more than 2E calls)

O(V + E)

- Each vertex is visited at most once (O(V)).
- Each edge is considered at most twice (O(E)).

Cost model in analysis above is the sum of:

- Number of dfs calls.
- marked[w] checks.

Runtime for DepthFirstPaths

Very hard question: Could we say the runtime is O(E)?

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  } ...
}
```

Assume graph uses adjacency list!

Argument: Can only visit a vertex if there is an edge to it.

- # of DFS calls is bounded above by E.
- So why not just say O(E)?

Runtime for DepthFirstPaths

Very hard question: Could we say the runtime is O(E)? No.

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  } ...
}
```

Assume graph uses adjacency list!

Can't say O(E)!

- Constructor has to create an all false `marked` array.
- This marking of all vertices as false takes Θ(V) time.

Our cost model earlier (`dfs` calls + `marked` checks) does not provide a tight bound.

Graph Problems

Problem	Problem Description	Solution	Efficiency (adj. list)
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo	O(V+E) time Θ(V) space

Runtime is O(V+E)

- Based on cost model: O(V) dfs calls and O(E) marked[w] checks.
- Can't say O(E) because creating marked array
- Note, can't say Θ(V+E), example: Graph with no edges touching source.

Space is Θ(V).

- Need arrays of length V to store information.

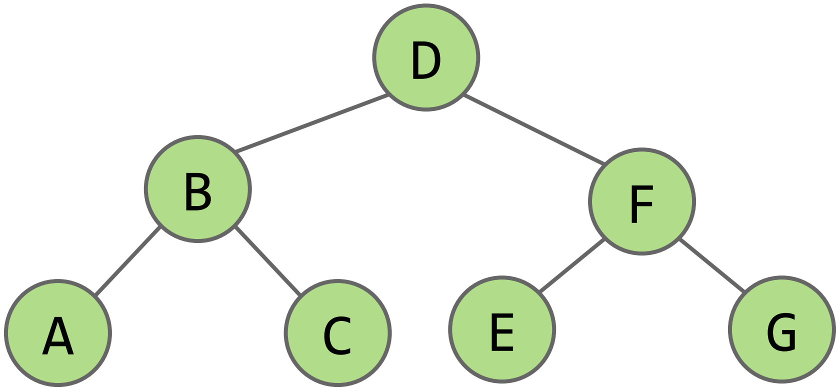
BreadthFirstPaths

Lecture 23, CS61B, Spring 2024

Tree and Graph Traversals

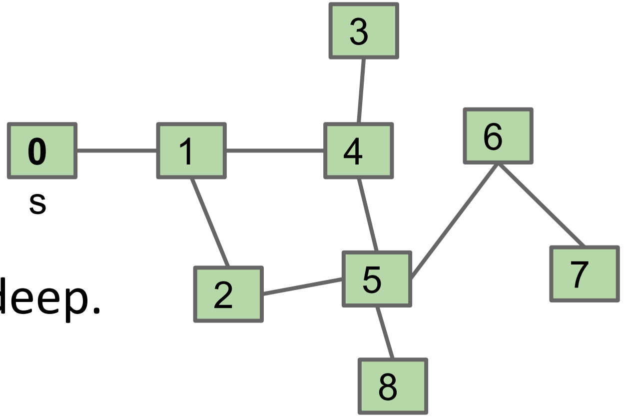
Just as there are many tree traversals:

- Preorder: DBACFEG
- Inorder: ABCDEFG
- Postorder: ACBEGFD
- Level order: DBFACEG

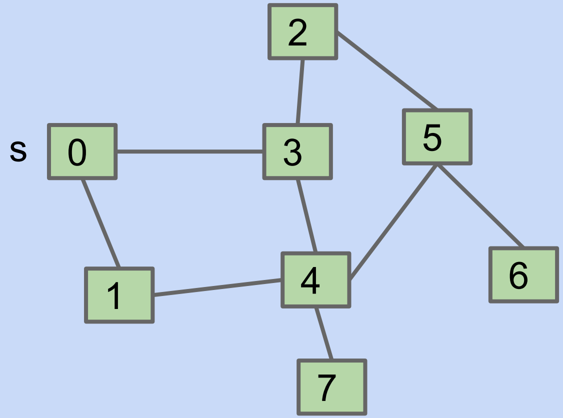


So too are there many graph traversals, given some source:

- DFS Preorder: 012543678 (dfs calls).
- DFS Postorder: 347685210 (dfs returns).
- BFS order: Act in order of distance from s.
  - BFS stands for “breadth first search”.
  - Analogous to “level order”. Search is wide, not deep.
  - 0 1 24 53 68 7



Shortest Paths Challenge (Video Viewers Only)



Goal: Given the graph above, find the shortest path from s to all other vertices.

- Give a general algorithm.
- Hint: You'll need to somehow visit vertices in BFS order.
- Hint #2: You'll need to use some kind of data structure.
- Hint #3: Don't use recursion.

BFS Answer

Breadth First Search.

- Initialize a queue with a starting vertex s and mark that vertex.
  - A queue is a list that has two operations: enqueue (a.k.a. addLast) and dequeue (a.k.a. removeFirst).
  - Let's call this the queue our ***fringe***.
- Repeat until queue is empty:
  - Remove vertex v from the front of the queue.
  - For each unmarked neighbor n of v:
    - Mark n.
    - Set edgeTo[n] = v (and/or distTo[n] = distTo[v] + 1).
    - Add n to end of queue.

A queue is the opposite of a stack. Stack has push (addFirst) and pop (removeFirst).

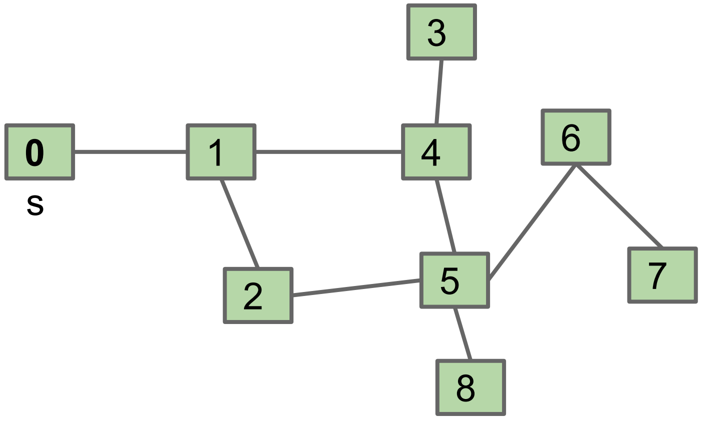
Do this if you want to track distance value.

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	F	-	0
1	F	-	-
2	F	-	-
3	F	-	-
4	F	-	-
5	F	-	-
6	F	-	-
7	F	-	-
8	F	-	-



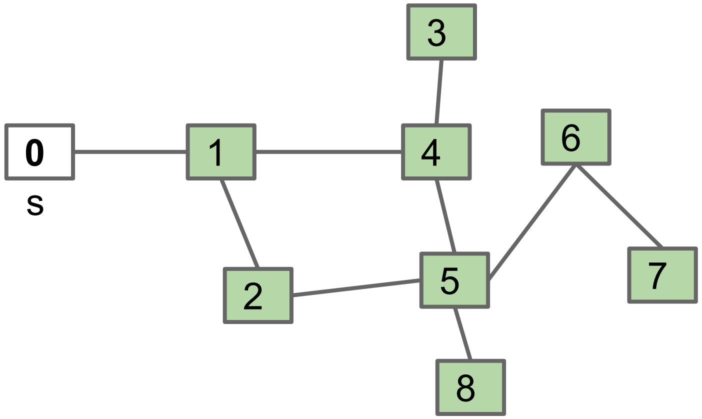
Queue: []

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
    - Remove vertex v from fringe.
    - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	F	-	-
2	F	-	-
3	F	-	-
4	F	-	-
5	F	-	-
6	F	-	-
7	F	-	-
8	F	-	-



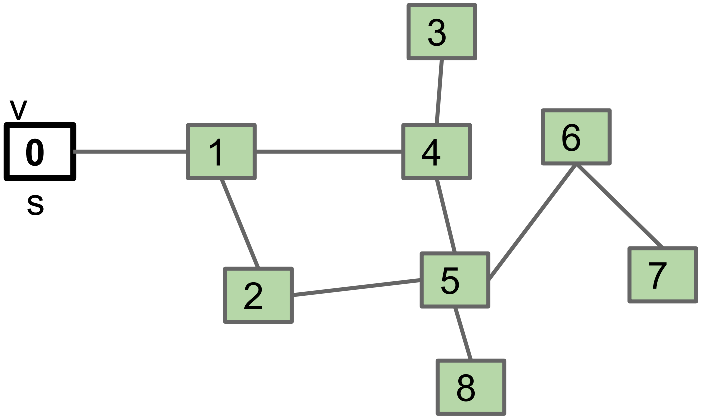
Queue: [0]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	F	-	-
2	F	-	-
3	F	-	-
4	F	-	-
5	F	-	-
6	F	-	-
7	F	-	-
8	F	-	-



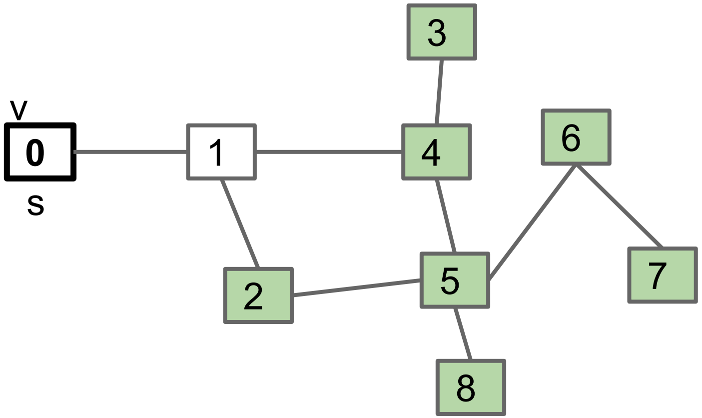
Queue: []

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	F	-	-
3	F	-	-
4	F	-	-
5	F	-	-
6	F	-	-
7	F	-	-
8	F	-	-



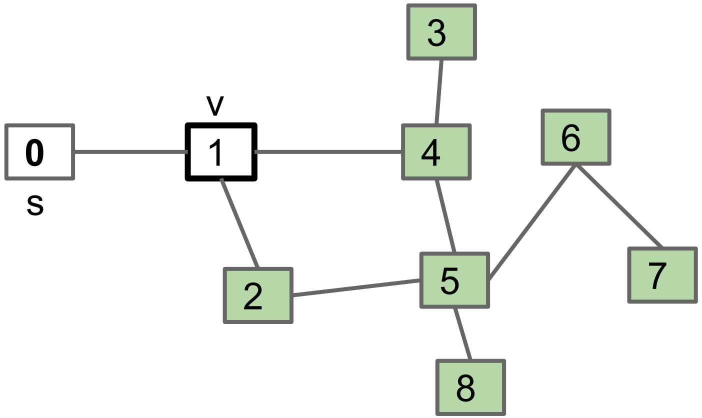
Queue: [1]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	F	-	-
3	F	-	-
4	F	-	-
5	F	-	-
6	F	-	-
7	F	-	-
8	F	-	-



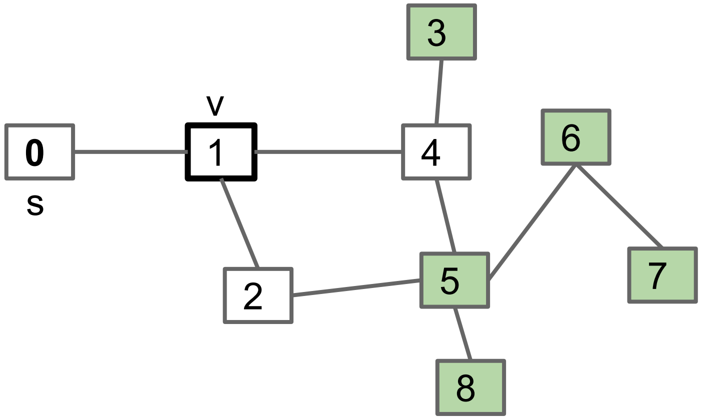
Queue: []

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	F	-	-
4	T	1	2
5	F	-	-
6	F	-	-
7	F	-	-
8	F	-	-



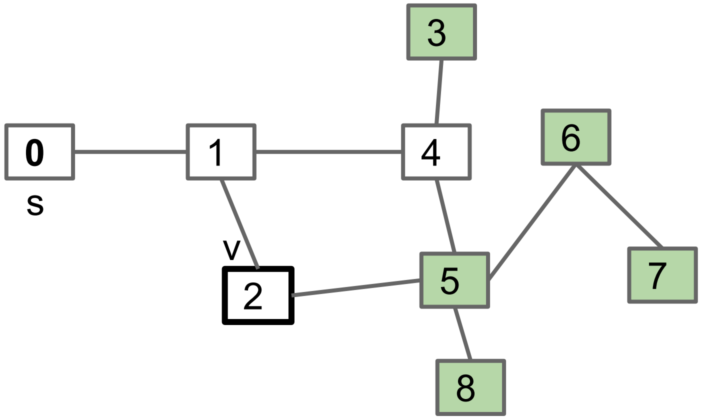
Queue: [2, 4]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	F	-	-
4	T	1	2
5	F	-	-
6	F	-	-
7	F	-	-
8	F	-	-



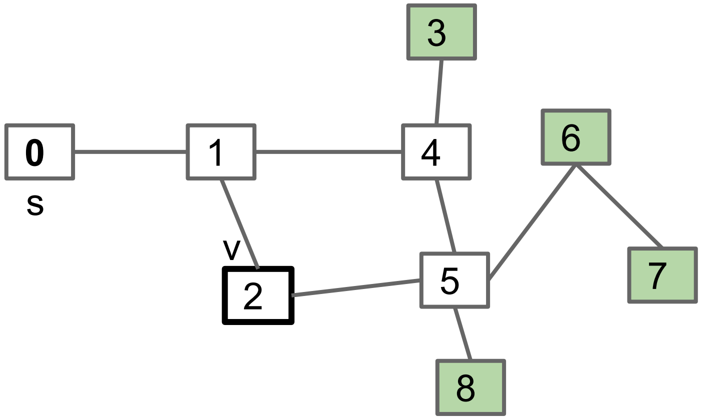
Queue: [4]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	F	-	-
4	T	1	2
5	T	2	3
6	F	-	-
7	F	-	-
8	F	-	-

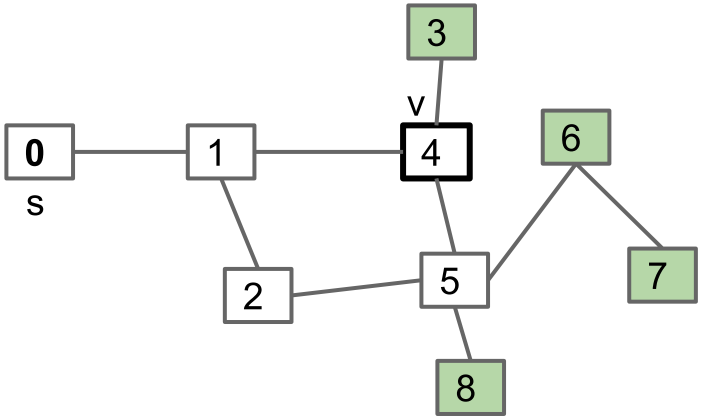


Queue: [4, 5]

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	F	-	-
4	T	1	2
5	T	2	3
6	F	-	-
7	F	-	-
8	F	-	-



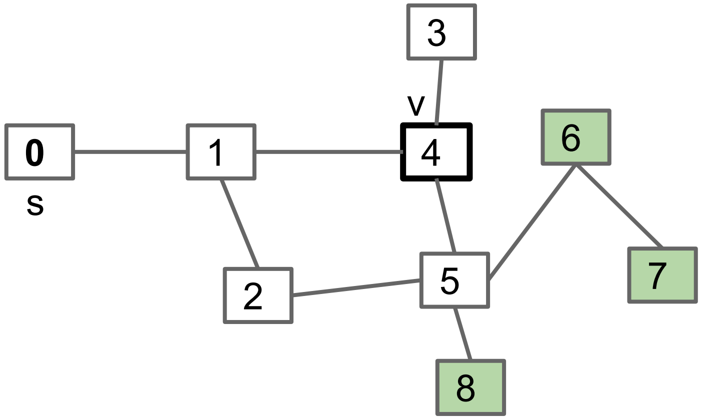
Queue: [5]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
    - Remove vertex v from fringe.
    - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	F	-	-
7	F	-	-
8	F	-	-



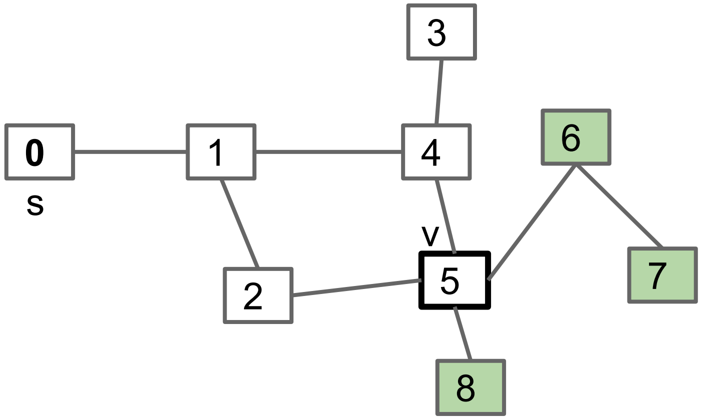
Queue: [5, 3]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	F	-	-
7	F	-	-
8	F	-	-



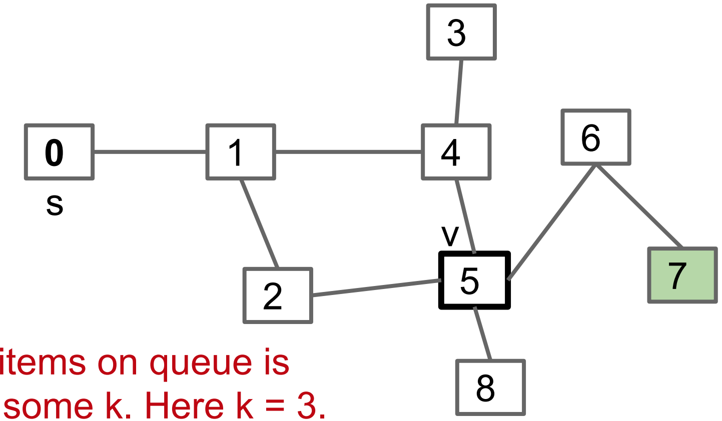
Queue: [3]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	F	-	-
8	T	5	4



Note: distance to all items on queue is always k or k + 1 for some k. Here k = 3.

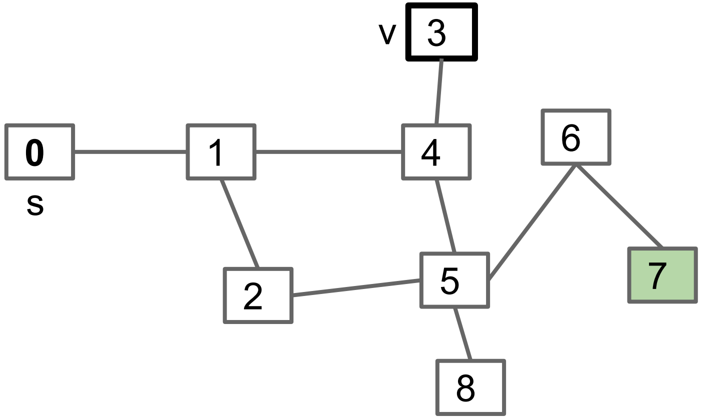
Queue: [3, 6, 8]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	F	-	-
8	T	5	4



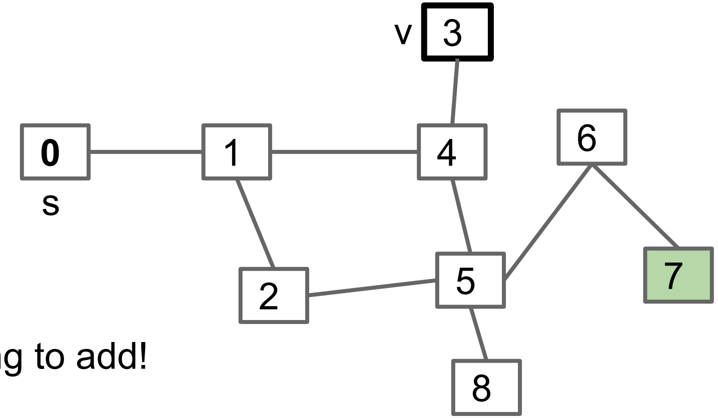
Queue: [6, 8]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	F	-	-
8	T	5	4



Nothing to add!

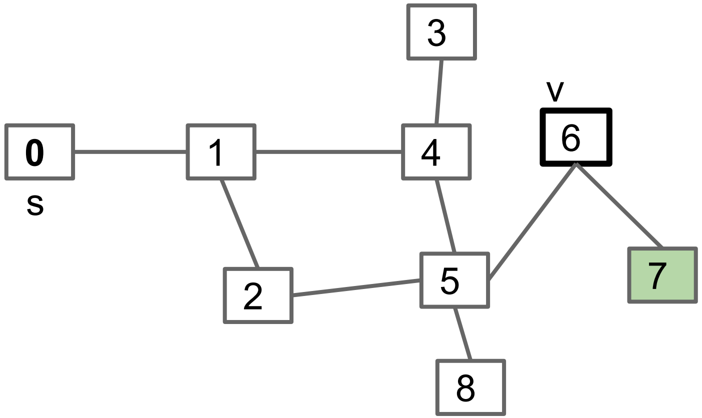
Queue: [6, 8]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	F	-	-
8	T	5	4



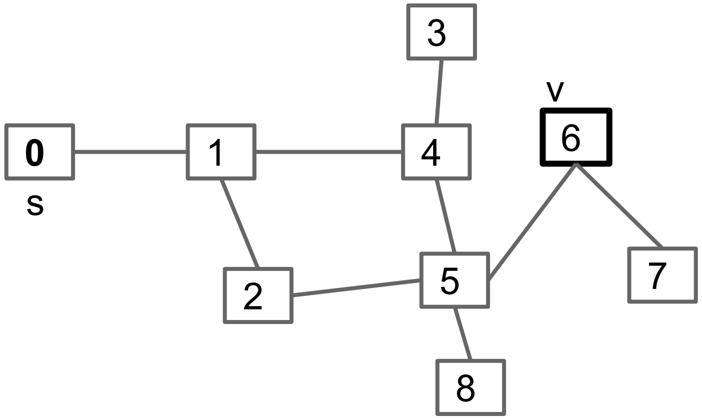
Queue: [8]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	T	6	5
8	T	5	4



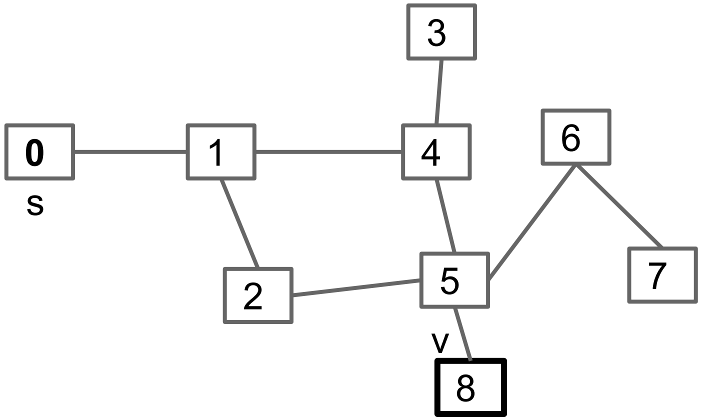
Queue: [8, 7]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
    - Remove vertex v from fringe.
    - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	T	6	5
8	T	5	4



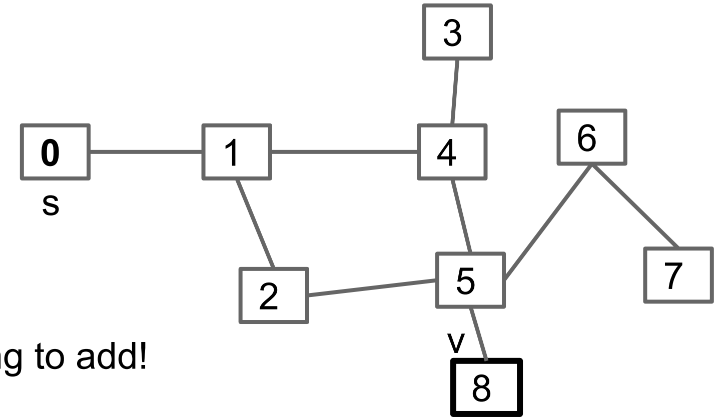
Queue: [7]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	T	6	5
8	T	5	4



Nothing to add!

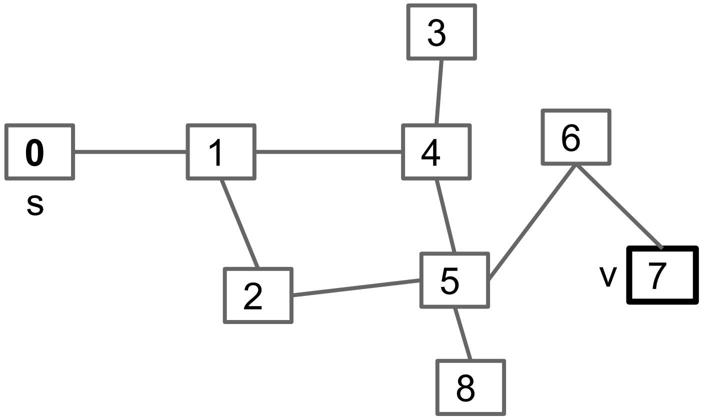
Queue: [7]

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	T	6	5
8	T	5	4



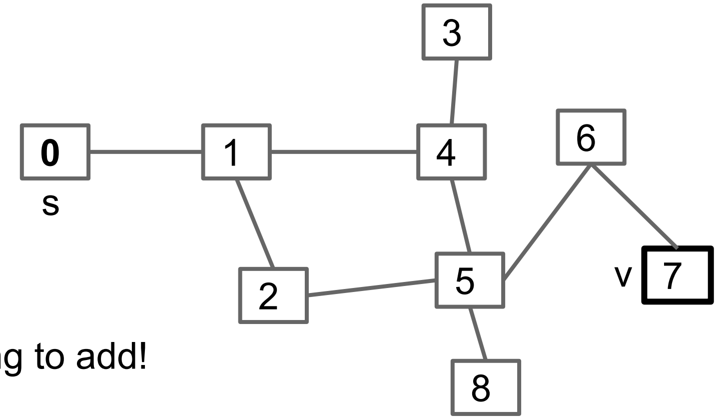
Queue: []

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex.
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	T	6	5
8	T	5	4



Nothing to add!

Queue: []

BreadthFirstPaths Implementation

Lecture 23, CS61B, Spring 2024

BreadthFirstPaths Implementation

```
public class BreadthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  ...

  private void bfs(Graph G, int s) {
  Queue<Integer> fringe =
          new Queue<Integer>();
  fringe.enqueue(s);
  marked[s] = true;
  while (!fringe.isEmpty()) {
    int v = fringe.dequeue();
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        fringe.enqueue(w);
        marked[w] = true;
        edgeTo[w] = v;
      }
    }
  }
}
```

marked[v] is true iff v connected to s

edgeTo[v] is previous vertex on path from s to v

set up starting vertex

for freshly dequeued vertex v, for each neighbor that is unmarked:

- Enqueue that neighbor to the fringe.
- Mark it.
- Set its edgeTo to v.

Graph Problems

Problem	Problem Description	Solution	Efficiency (adj. list)
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo	O(V+E) time Θ(V) space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	O(V+E) time Θ(V) space

Runtime for shortest paths is also O(V+E)

- Based on same cost model: O(V) .next() calls and O(E) marked[w] checks.

Space is Θ(V).

- Need arrays of length V to store information.

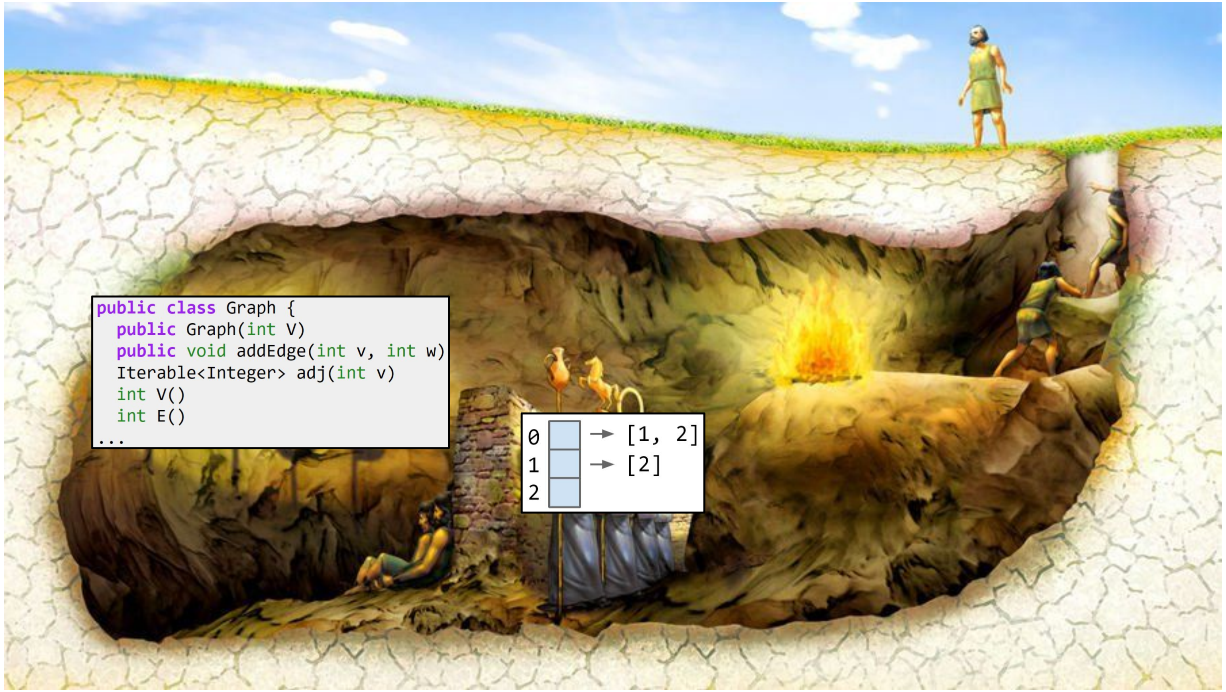
Graph Implementations and Runtime

Lecture 23, CS61B, Spring 2024

Our Graph API and Implementation

Our choice of how to implement the Graph API has profound implications on runtime.

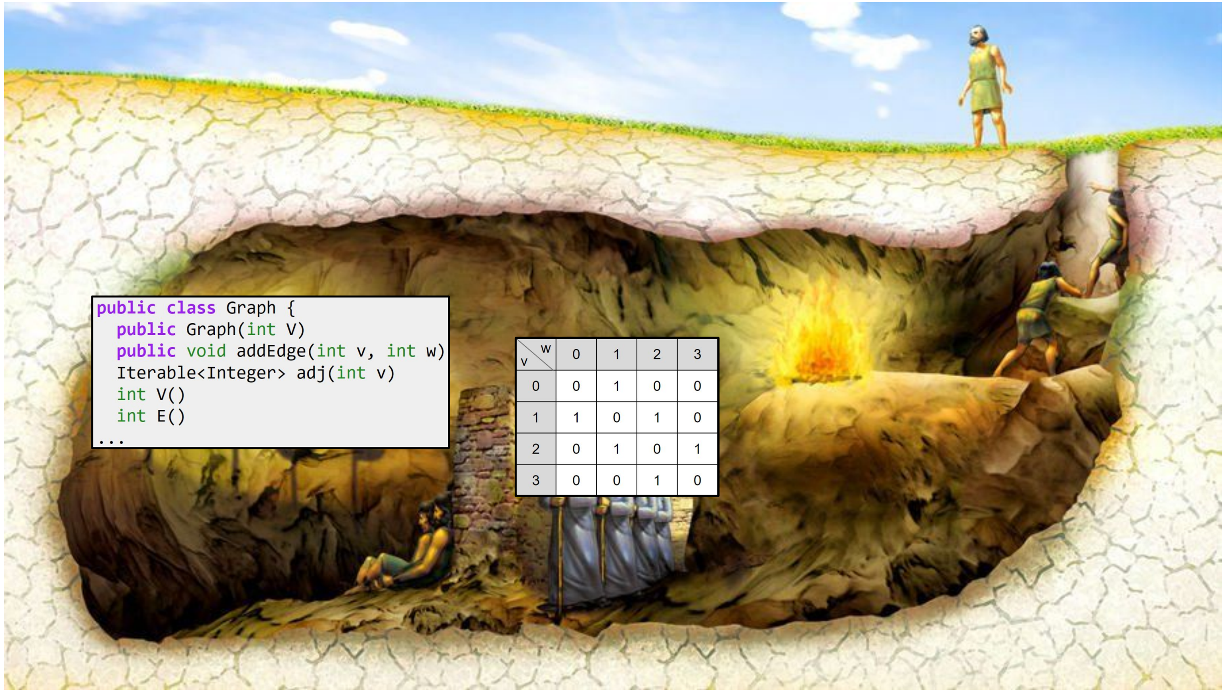
- Example: Saw that print on Adjacency Lists was O(V + E).



Our Graph API and Implementation

Our choice of how to implement the Graph API has profound implications on runtime.

- What happens to print runtime if we use an adjacency matrix?



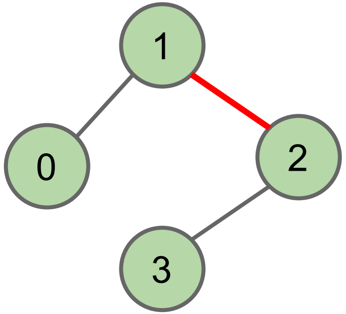
Graph Representations

Graph Representation 1: Adjacency Matrix.

- G.adj(2) would return an iterator where we can call next() up to two times
  - next() returns 1
  - next() returns 3
- Total runtime to iterate over all neighbors of v is Θ(V).
  - Underlying code has to iterate through entire array to handle next() and hasNext() calls.

G.adj(2) returns an iterator that will ultimately provide 1, then 3.

v \ w	0	1	2	3
0	0	1	0	0
1	1	0	**1**	0
2	0	**1**	0	1
3	0	0	1	0



Graph Printing Runtime: http://yellkey.com/pretty

What is the order of growth of the running time of the print client from before if the graph uses an **adjacency-matrix** representation, where V is the number of vertices, and E is the total number of edges?

A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(V*E)$

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

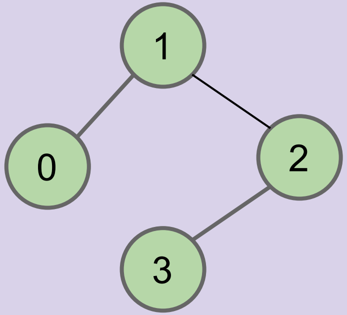
Runtime to iterate over v's neighbors?

-

How many vertices do we consider?

-

	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0



Graph Printing Runtime

What is the order of growth of the running time of the print client from before if the graph uses an **adjacency-matrix** representation, where V is the number of vertices, and E is the total number of edges?

A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\boldsymbol{\Theta(V^2)}$
D. $\Theta(V*E)$

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + “-” + w);
    }
}
```

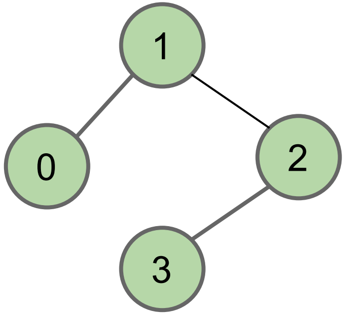
Runtime to iterate over v's neighbors?

- $\Theta(V)$.

How many vertices do we consider?

- V times.

	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0



Runtime for DepthFirstPaths

Give a tight O bound for the runtime for the DepthFirstPaths constructor.

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  } ...
}
```

v \ w	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

Assume graph uses adjacency matrix!

This is a lot to digest!
- If you are feeling lost on this problem, don't feel bad.
- But do work on trying to understand the ideas here!

Runtime for DepthFirstPaths

Give a tight O bound for the runtime for the DepthFirstPaths constructor.

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  } ...
}
```

Assume graph uses adjacency matrix!

v \ w	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

$O(V^2)$

- In the worst case, we iterate over the neighbors of all vertices.

We create ≤ V iterators.

- Each one takes a total of $\Theta(V)$ time to iterate over.

Essentially, iterating over the entire adjacency matrix takes $O(V^2)$ time.

Graph Problems for Adjacency Matrix Based Graphs

Problem	Problem Description	Solution	Efficiency (adj. matrix)
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java [Demo]	$O(V^2)$ time $\Theta(V)$ space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	BreadthFirstPaths.java [Demo]	$O(V^2)$ time $\Theta(V)$ space

If we use an adjacency matrix, BFS and DFS become $O(V^2)$.

- For sparse graphs (number of edges << V for most vertices), this is terrible runtime.
- Thus, we'll always use adjacency-list unless otherwise stated.

Project 2B Note

Lecture 23, CS61B, Spring 2024

Project Note

On project 2B, you cannot import the Princeton Graphs library.

- We want you to design your own graph API that has just the operations you need of the project.

Note: You may not need a separate Graph class. Whether you have one is up to you.

Summary

Graph API: We used the Princeton algorithms book API today.

- This is just one possible graph API. We'll see other graph APIs in this class.
  - You'll decide on your own graph API in project 2B.
- Choice of API determines how client needs to think in order to write code.
  - e.g. Getting the degree of a vertex requires many lines of code with this choice of API.
  - Choice may also affect runtime and memory of client programs.

```
public class Graph {
  public Graph(int V):                 Create empty graph with v vertices
  public void addEdge(int v, int w): add an edge v-w
  Iterable<Integer> adj(int v):        vertices adjacent to v
  int V():                             number of vertices
  int E():                             number of edges ...
```

Summary

Graph Implementations: Saw three ways to implement our graph API.

- Adjacency matrix.
- List of edges.
- Adjacency list (most common in practice).

BFS: Uses a queue instead of recursion to track what work needs to be done.

Choice of implementation has big impact on runtime and memory usage!

- DFS and BFS runtime with adjacency list: $O(V + E)$
- DFS and BFS runtime with adjacency matrix: $O(V^2)$