Announcements

Reminder:

- Office hours for a given assignment end 3 days after the deadline.
    - That means today is the last day for project 2A.
    - Office hours will no longer cover project 2A starting 3/21.

Pseudowalkthrough for video Project 2B exists.

CS61B, 2019

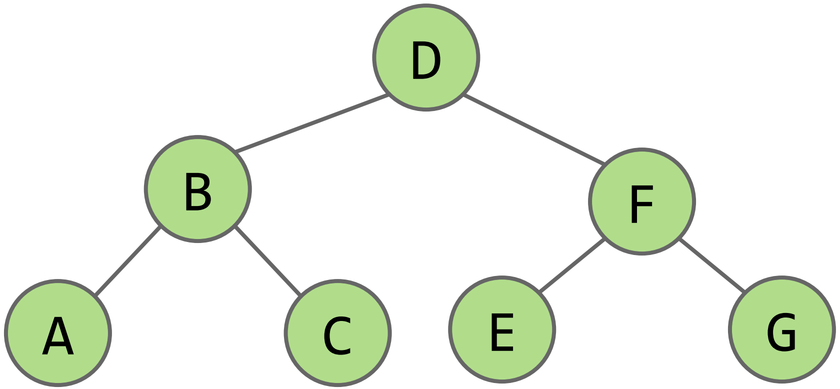
Lecture 24: Graphs II: Graph Traversal Implementations

- BreadthFirstPaths
- Graph API
- Graph Representations and Graph Algorithm Runtimes
- Graph Traversal Runtimes
- Layers of Abstraction

Tree and Graph Traversals

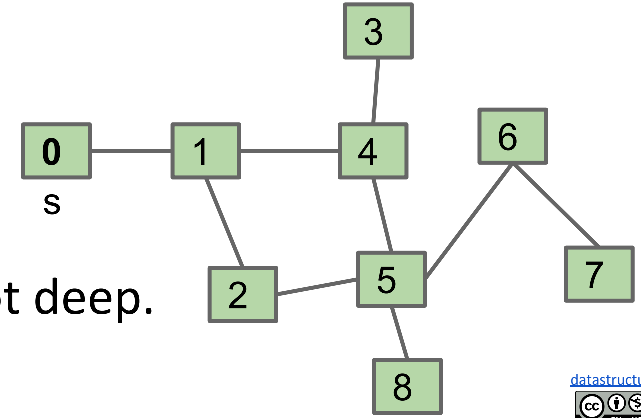
Just as there are many tree traversals:

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

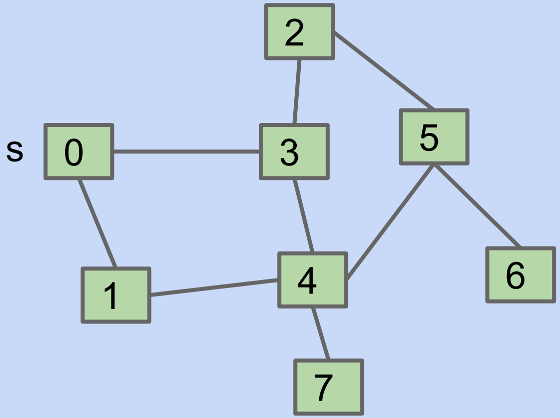


So too are there many graph traversals, given some source:

- 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



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.

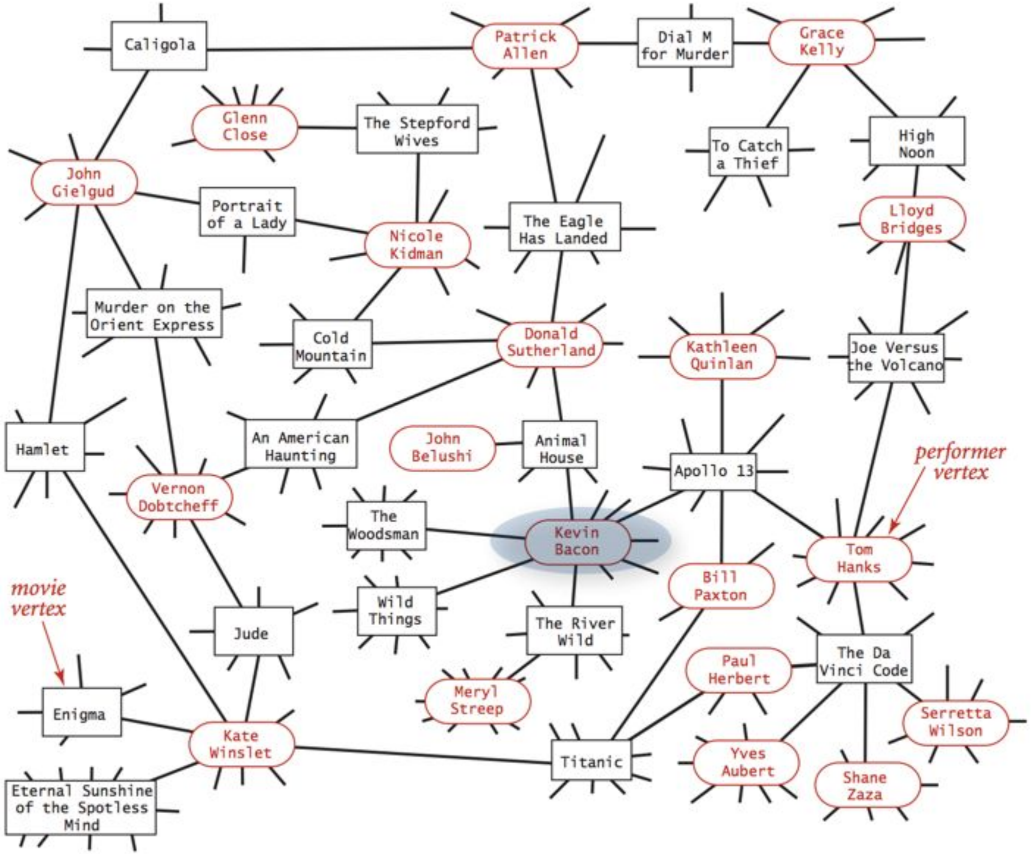
Demo: [Breadth First Paths]()

One use of BFS: Kevin Bacon

Graph with two types of vertices:

- Movies
- Actors

Perform BFS from s=Kevin Bacon.



BreadthFirstSearch for Google Maps

Would breadth first search be a good algorithm for a navigation tool (e.g. Google Maps)?

- Assume vertices are intersection and edges are roads connecting intersections.

BreadthFirstSearch for Google Maps

Would breadth first search be a good algorithm for a navigation tool (e.g. Google Maps)?

- Assume vertices are intersection and edges are roads connecting intersections.

Some roads are longer than others.

- BAD!

Will discuss how to deal with this in the next lecture.

- First, we should talk about how graphs and graph algorithms are actually implemented in a programming language.

Graph API

Graph Representations

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

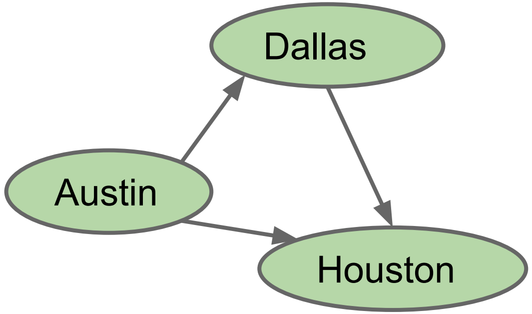
- **An API** (Application Programming Interface) for graphs.
  - For our purposes today, these are our Graph methods, including their signatures and behaviors.
  - Defines how Graph client programmers must think.
- An underlying data structure to represent our graphs.

Our choices can have profound implications on:

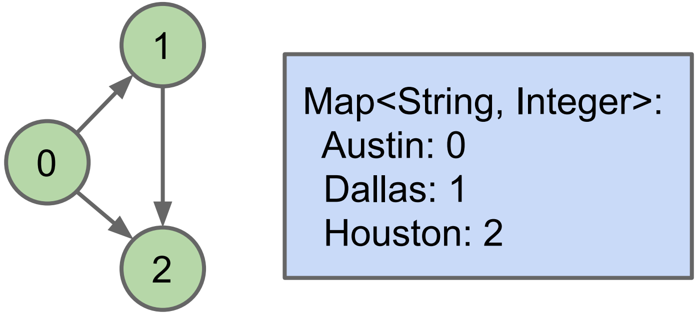
- **Runtime.**
- **Memory usage.**
- **Difficulty of implementing various graph algorithms.**

Graph 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.



How you'd build it.

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
...
```

Some features:

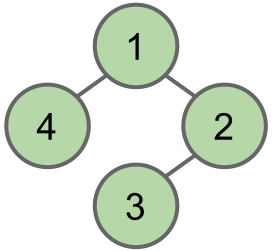
- Number of vertices must be specified in advance.
- Does not support weights (labels) on nodes or edges.
- Has no method for getting the number of edges for a vertex (i.e. its degree).

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, 2) = 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; }
```

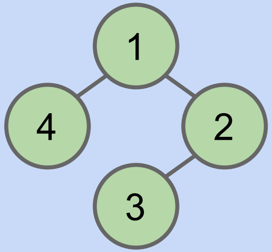
(degree = # edges)

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) {
    ???
}
```

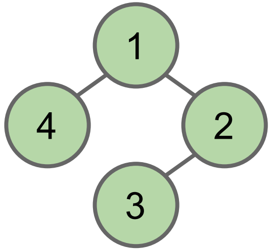
```
$ java printDemo
1 - 2
1 - 4
2 - 1
2 - 3
3 - 2
4 - 1
```

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
1 - 2
1 - 4
2 - 1
2 - 3
3 - 2
4 - 1
```

Graph API and DepthFirstPaths

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

- Will come back to this in more depth, but first...

Graph Representation and Graph Algorithm Runtimes

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.

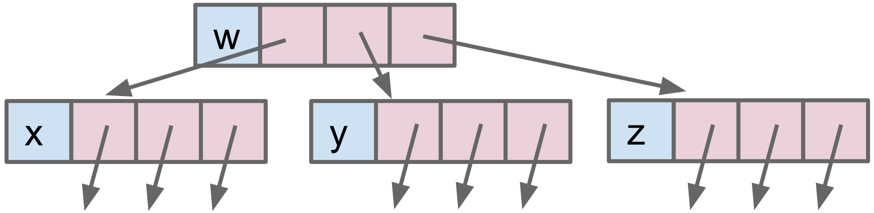
Graph Representations

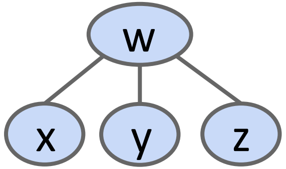
Just as we saw with trees, there are many possible implementations we could choose for our graphs.

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

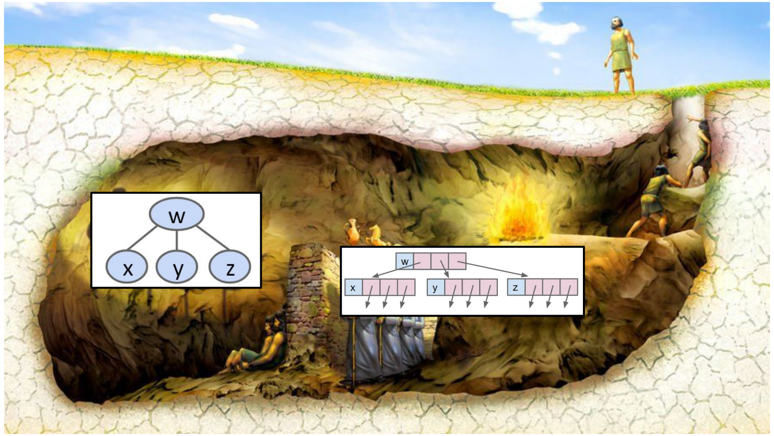
Tree Representations

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



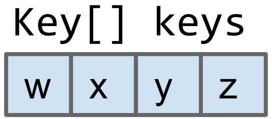


1a: Fixed Number of Links (One Per Child)

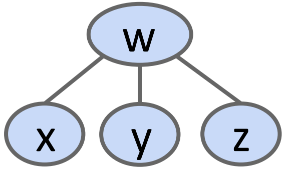


Tree Representations

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

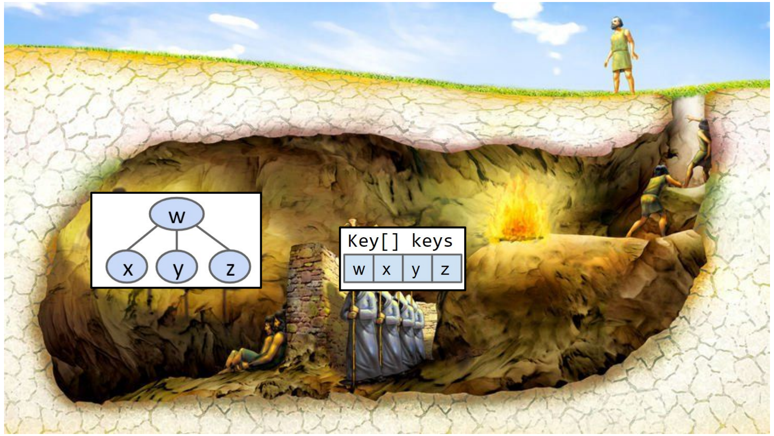


3: Array of Keys



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

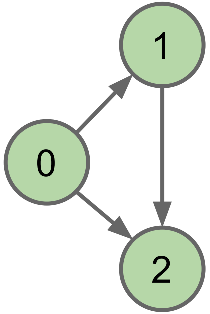
… but only works for complete trees.



Graph Representations

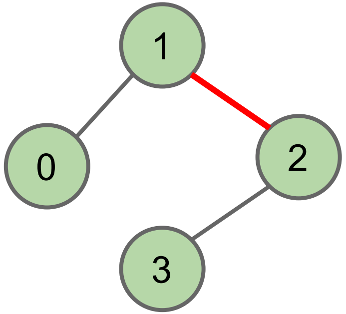
Graph Representation 1: Adjacency Matrix.

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



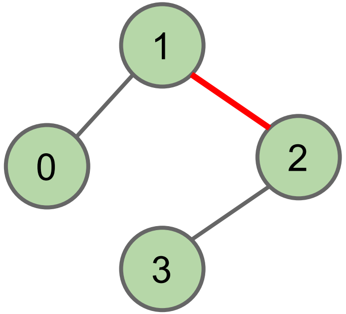
Graph Representations

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/allow

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);
    }
}
```

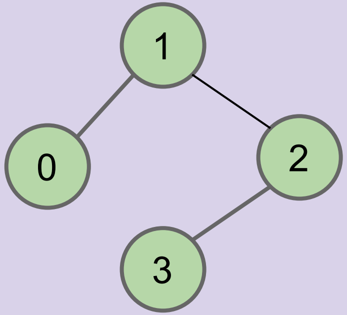
Runtime to iterate over v's neighbors?

-

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: http://yellkey.com/allow

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. Θ(V)
B. Θ(V + E)
C. Θ(V^2)
D. Θ(V*E)

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

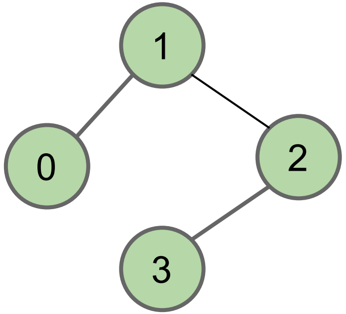
Runtime to iterate over v’s neighbors?

- Θ(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

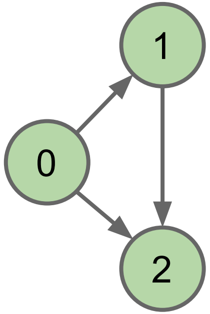


More Graph Representations

Representation 2: Edge Sets: Collection of all edges.

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

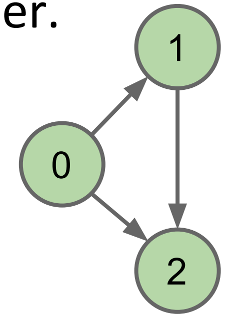
```
{(0, 1), (0, 2), (1, 2)}
```

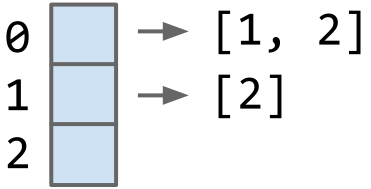


More Graph Representations

Representation 3: Adjacency lists.

- Common approach: Maintain array of lists indexed by vertex number.
- Most popular approach for representing graphs.





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

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. $\Theta(V*E)$

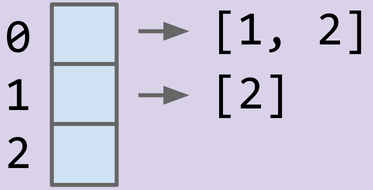
```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + “-” + w);
    }
}
```

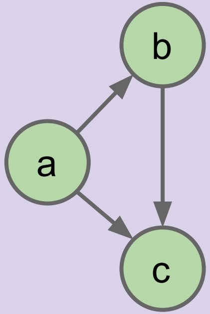
Runtime to iterate over v’s neighbors?

-

How many vertices do we consider?

-





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

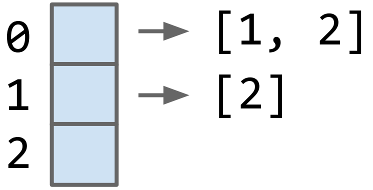
What is the order of growth of the running time of the print client if the graph uses an ***adjacency-list*** representation, where V is the number of vertices, and E is the total number of edges? **Best case: Θ(V)** **Worst case: $\Theta(V^2)$**

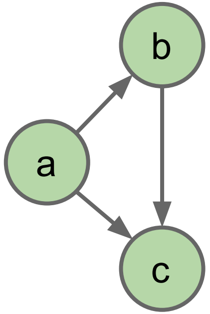
A. Θ(V)
B. Θ(V + E)
C. $\Theta(V^2)$
D. Θ(V*E)

```
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? List can be between 1 and V items.

- Ω(1), O(V).





How many vertices do we consider?

- V.

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

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? **Best case: Θ(V) Worst case: Θ(V²)**

A. Θ(V)
B. Θ(V + E)
C. Θ(V²)
D. Θ(V*E)

? ? ? ? ?

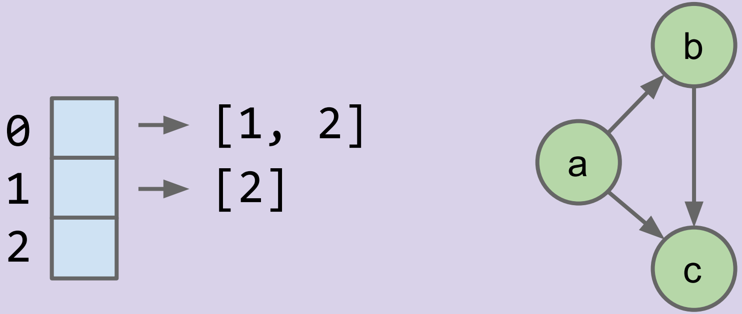
```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

Runtime to iterate over v's neighbors? List can be between 0 and V items.

- Ω(1), O(V).

How many vertices do we consider?

- V.



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

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?

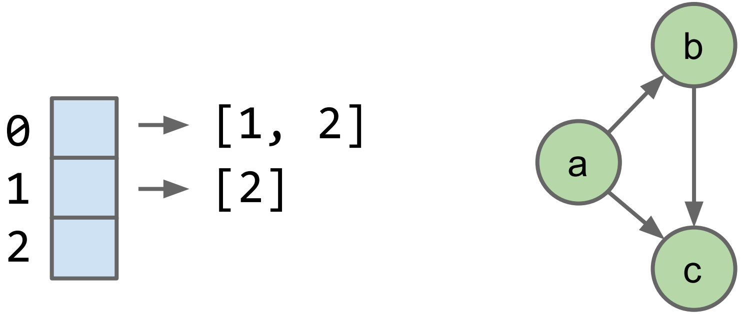
A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(V*E)$

```
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: $\Theta(V^2)$

All cases: $\Theta(V + E)$

- Create V iterators.
- Print E times.



Graph Printing Runtime

Runtime: Θ(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 "shape" of increasingly complex graphs we generate, as V and E grow, the runtime will always grow exactly as Θ(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 Θ(sqrt(V)). Runtime is Θ(V + sqrt(V)), which is just Θ(V).
- Example shape 2: Very dense graph where E grows very quickly, e.g. every vertex connected to every other.
  - E is Θ(V^2). Runtime is Θ(V + V^2), which is just Θ(V^2).

Graph Representations

Runtime of some basic operations for each representation:

idea	addEdge(s, t)	for(w : adj(v))	print()	hasEdge(s, t)	space used
adjacency matrix	$\Theta(1)$	$\Theta(V)$	$\Theta(V^2)$	$\Theta(1)$	$\Theta(V^2)$
list of edges	$\Theta(1)$	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$
adjacency list	$\Theta(1)$	$\Theta(1)$ to $\Theta(V)$	$\Theta(V+E)$	$\Theta(\text{degree}(v))$	$\Theta(E+V)$

Note: These operations are not part of the `Graph` class's API.

In practice, adjacency lists are most common.

- Many graph algorithms rely heavily on `adj(s)`.
- Most graphs are sparse (not many edges in each bucket).

Bare-Bones Undirected Graph Implementation

```
public class Graph {
    private final int V;  private List<Integer>[] adj;

    public Graph(int V) {
        this.V = V;
        adj = (List<Integer>[]) new ArrayList[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new ArrayList<Integer>();
        }
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);   adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```

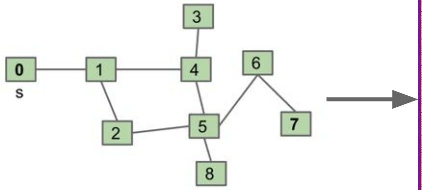
Cannot create array of anything involving generics, so have to use weird cast as with project 1A.

Graph Traversal Implementations and Runtime

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.



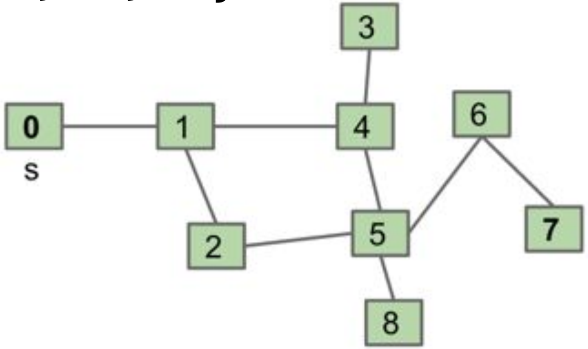
Paths.java

```
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 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

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 list!

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 list!

O(V + E)

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

vertex visits (w takes on < V values)

edge considerations (fewer than 2E calls)

Cost model is the sum of:

- next() calls to G.adj(v) iterator.
- marked[w] checks.

Graph Problems

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

Runtime is $O(V+E)$

- Based on cost model: $O(V)$ .next() calls and $O(E)$ marked[w] checks.
- Note, can't say $\Theta(V+E)$, example: Graph with no edges touching source.

Space is $\Theta(V)$.

- Need arrays of length V to store information.

BreadthFirstPaths Implementation

```java
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.

Layers of Abstraction

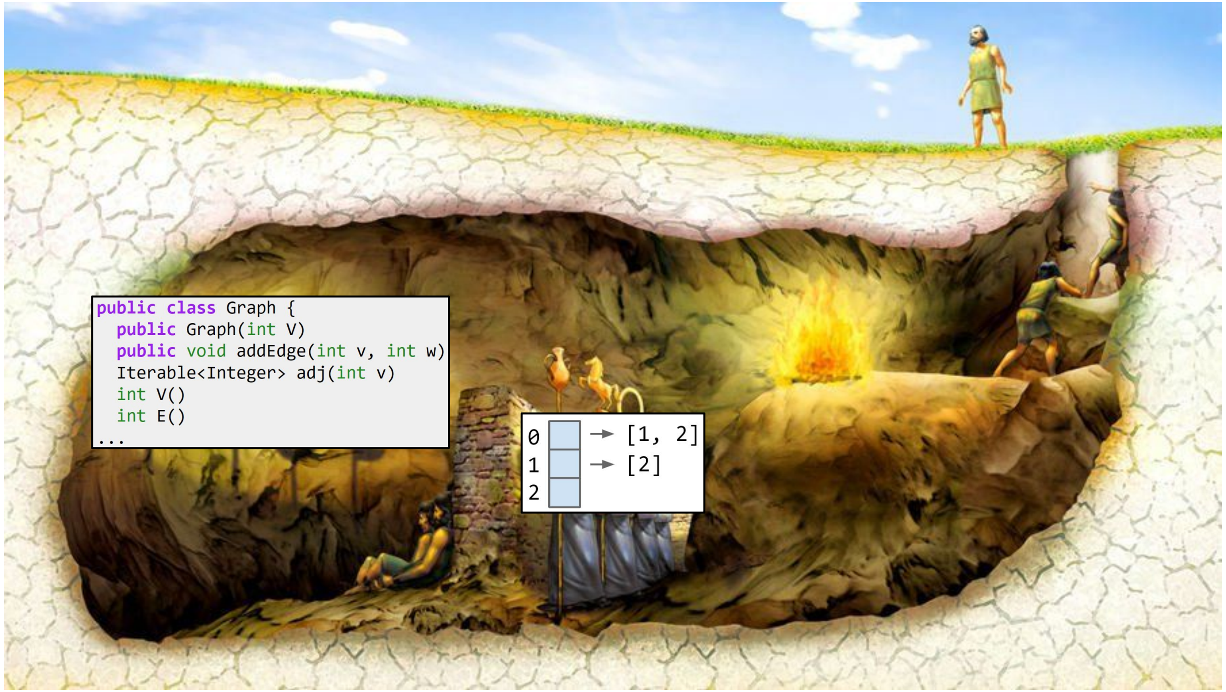
Clients and Our Graph API

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

Our Graph API and Implementation

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

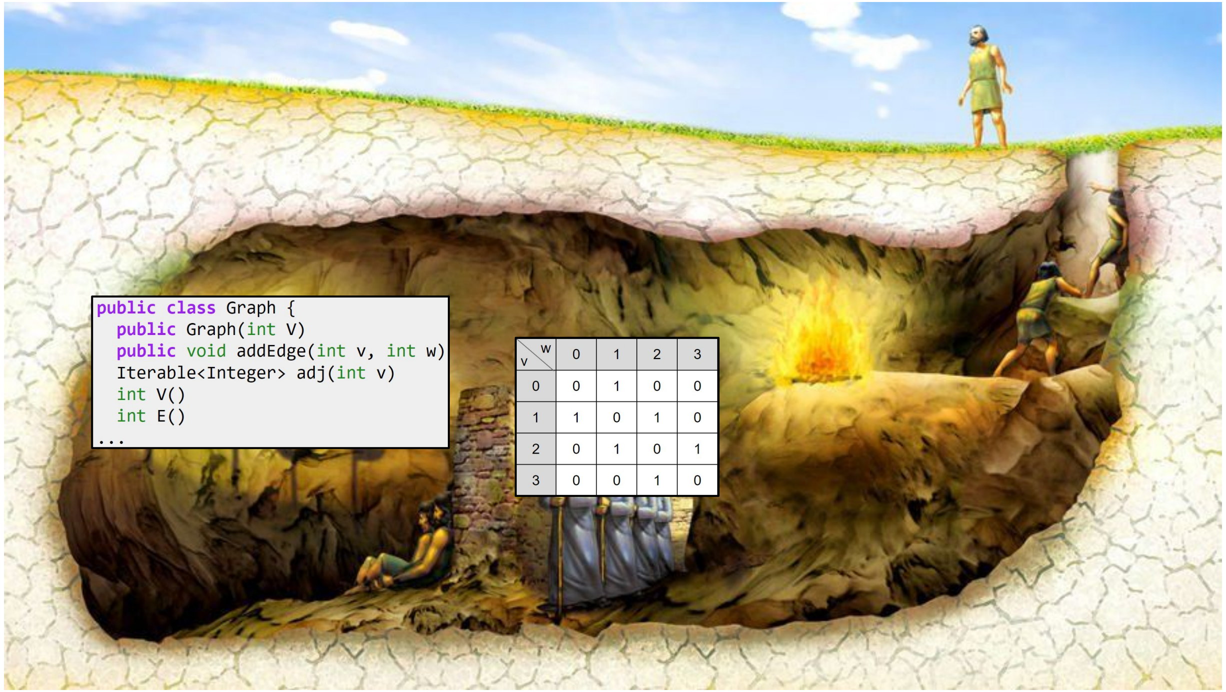
- Example: Saw that DepthFirstPaths on Adjacency Lists was O(V + E).



Our Graph API and Implementation

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

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



```
public class Graph {
  public Graph(int V)
  public void addEdge(int v, int w)
  Iterable<Integer> adj(int v)
  int V()
  int E()
...
```

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

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!

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!

$O(V^2)$

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

We create ≤ V iterators.

- Each one takes a total of Θ (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.

Summary

Summary

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

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

- This is just one possible API. We'll see other APIs in this class.
- 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).

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: $O(V^2)$

Citations

http://www.gosidemount.com/Guided_Diving/images/guided_cavern.jpg