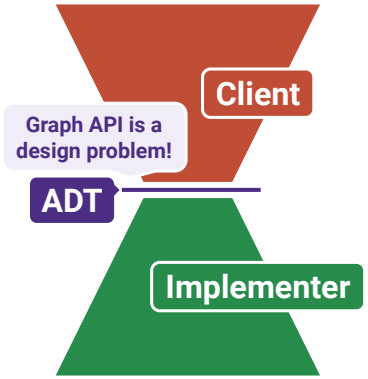
Using vs. Designing Graphs

Client uses graph algorithms like BFS and DFS to solve real-world problems.

ADT specifies the **API** for interacting with graphs, like how to get the neighbors of **v**.

Implementer designs a performant data structure representation for the ADT.



Example Graph API

Create an empty graph for **V** vertices.

Add an edge (**v**, **w**).

Return the neighboring vertices of **v**.

Return the total number of vertices, **V**.

Return the total number of edges, **V**.

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


Design Tradeoffs

Number of vertices (**V**) must be specified in the graph constructor.

Number of neighbors for a vertex **v**: get adj and then return the size of the list.

Unweighted graph only!

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


Q Using the Graph API

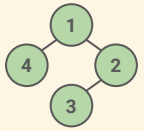
Write a client method to print a graph.

```
void print(Graph G) {



}
```

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

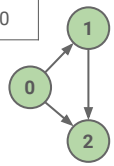



```
$ java printDemo
1 - 2
1 - 4
2 - 1
2 - 3
3 - 2
4 - 1
```

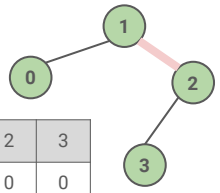
Graph Representation 1: Adjacency Matrix

Directed graph: `a[s][t]`.

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



Undirected graph: `a[v][w], a[w][v]`.

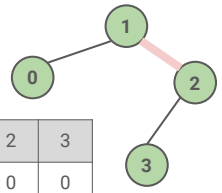


v \ w	0	1	2	3
0	0	**1**	0	0
1	**1**	0	**1**	0
2	0	**1**	0	**1**
3	0	0	**1**	0

Adjacency Matrix Runtime

G.adj(2) returns an iterable of [1, 3].

Runtime to iterate over all neighbors of **v** is **Θ(V)**: adj needs to return a new iterable containing all the indices with value 1.

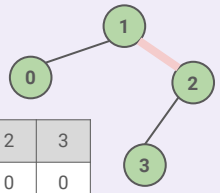


v \ w	0	1	2	3
0	0	**1**	0	0
1	**1**	0	**1**	0
2	0	**1**	0	**1**
3	0	0	**1**	0

Q Adjacency Matrix Runtime

Give the order of growth of the runtime for print if the graph is an **adjacency matrix**, where **V** is the number of vertices and **E** is the number of edges.

```
void print(Graph G) {
  for (int v = 0; v < G.V(); v++) {
    for (int w : G.adj(v)) {
      println(v + "-" + w);
    }
  }
}
```

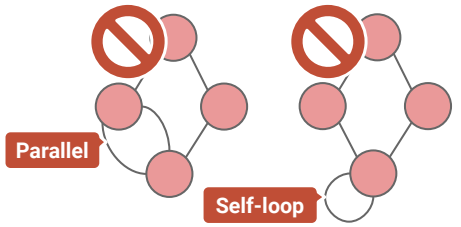


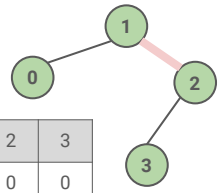
v \ w	0	1	2	3
0	0	**1**	0	0
1	**1**	0	**1**	0
2	0	**1**	0	**1**
3	0	0	**1**	0

Edges vs. Vertices in a Simple Graph

Simple Graph. A graph with no **self-loops** and no **parallel edges**.

$$0 \leq E \leq V(V-1)$$



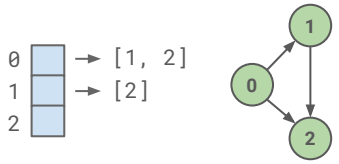


v \ w	0	1	2	3
0	**0**	**1**	0	0
1	**1**	**0**	**1**	0
2	0	**1**	**0**	**1**
3	0	0	**1**	**0**

Graph Representation 2: Adjacency List

Maintain array of lists indexed by vertex number.

Most popular approach for representing graphs.



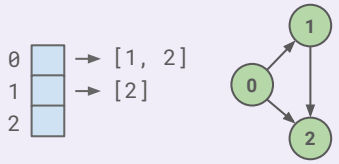
Q Adjacency List Runtime

Give the order of growth of the runtime for print if the graph is an **adjacency list**, where **V** is the number of vertices and **E** is the number of edges.

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

V iterations

???

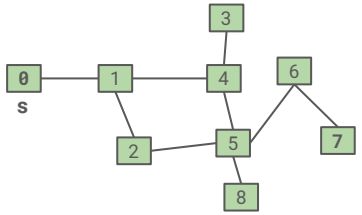


Design Pattern: Graph Solvers

Demo

Design pattern for graph clients: Decouple graph type from processing algorithm.

1. Create a graph instance and populate it with data.
2. Pass the graph instance to the constructor of the **client class**.
3. The **client class runs the algorithm** in its constructor and stores the solutions.
4. **Query the client class** for the stored solutions.




```
class DepthFirstPaths
  DepthFirstPaths(Graph G, int s)
  boolean hasPathTo(int v)
  Iterable<Integer> pathTo(int v)
```


Q Recursive DepthFirstPaths

Instance variables store algorithm data.

marked[**v**] is true iff **v** connected to **s**.
edgeTo[**v**] is vertex visited to get to **v**.

DepthFirstPaths constructor computes the result of the algorithm with the dfs recursive method.

How would we implement pathTo(v) and hasPathTo(v)?

```
private boolean[] marked;
private int[] edgeTo;
private int s;
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);
    }
  }
}
```

Q DepthFirstPaths Runtime

Give a tight big-O runtime bound for the DepthFirstPaths constructor. Assume the **adjacency list** graph representation.

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

```
private boolean[] marked;
private int[] edgeTo;
private int s;
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);
    }
  }
}
```



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

BreadthFirstPaths

Demo

Instance variables store algorithm data.

- marked[**v**] is true iff **v** connected to **s**.
- edgeTo[**v**] is vertex visited to get to **v**.

BreadthFirstPaths constructor computes the result of the algorithm with the bfs iterative method.

Cost model given undirected graph?

Each vertex is visited at most once.
Each edge is checked at most twice.

Graph Problems

Memory usage in addition to graph: O(V) to store the marked and edgeTo arrays.

How does the efficiency compare between **adjacency list and adjacency matrix**?

Problem	Problem Description	Solution	Efficiency (adj. list)
s-t paths	Find a path from s to every reachable vertex.	**Depth-first search**	O(V + E) runtime Θ(V) space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	**Breadth-first search**	O(V + E) runtime Θ(V) space

Graph Problems

If we use an **adjacency matrix**, BFS and DFS become $O(V^2)$. Terrible for **sparse graphs**!

Thus, we'll always use adjacency lists unless otherwise stated.

Problem	Problem Description	Solution	Efficiency (matrix)
s-t paths	Find a path from s to every reachable vertex.	**Depth-first search**	$O(V^2)$ runtime Θ(V) space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	**Breadth-first search**	$O(V^2)$ runtime Θ(V) space