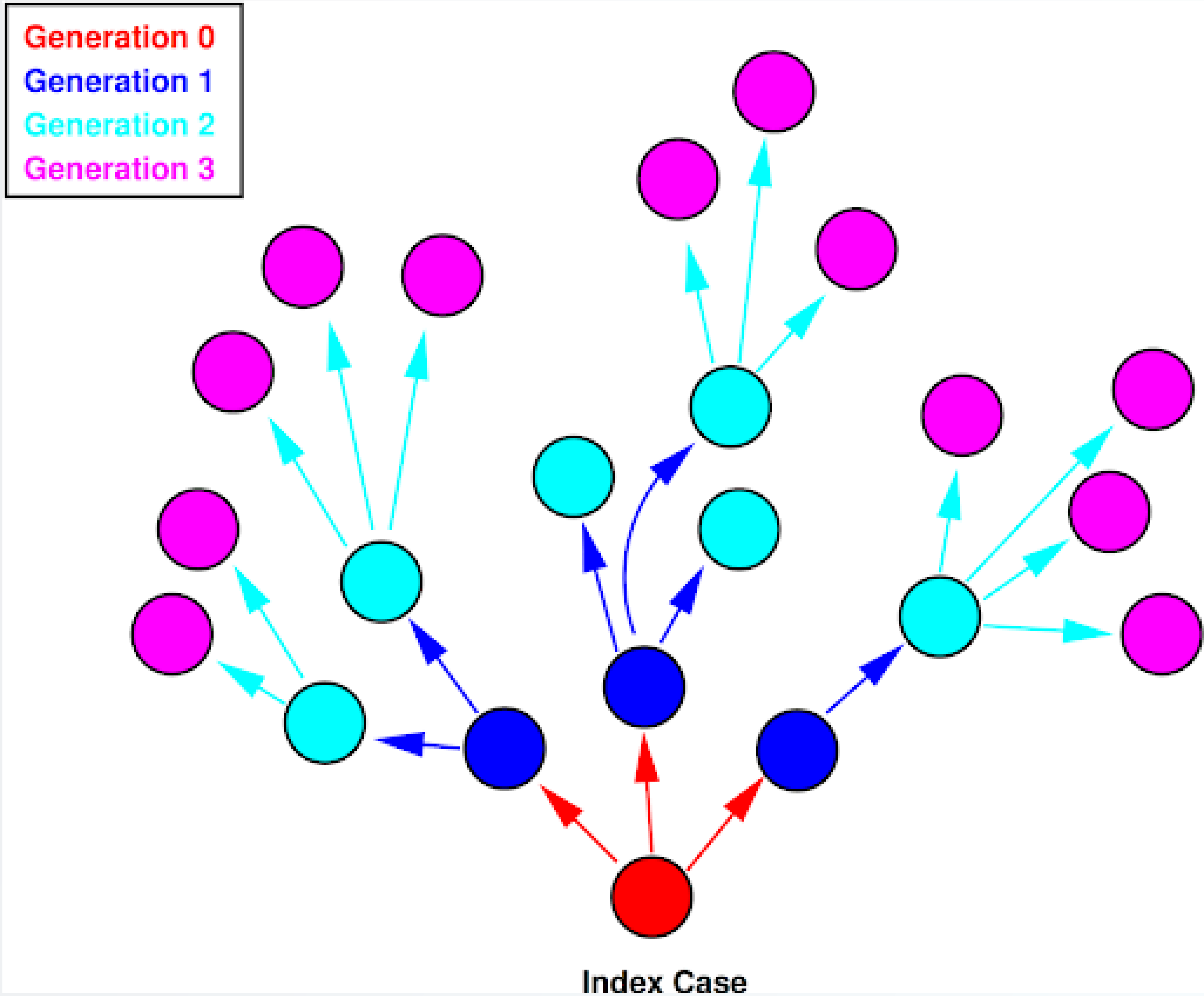
Graphs Applications: Epidemiology (HW6)

Background

- Graphs can be used to model contact between individuals
- Vertices represent individuals
- Edges represent a contact between two individuals
- The probability of infection can be represented as the weight of the edges

Generations

- Waves of secondary infection that flow from each previous infection
- Structure of the graph leads to different knock-on impacts of a single infection
  - Can help assess the pressure on the healthcare system



Calculate Generation of a Node

Just run BFS with a special `visitor` function that marks the generation of the node as 1 + the generation of the parent.

Immunization

Goal: lower the **reproduction number** $R_0 = \tau c d$

- τ is the *transmissibility*, or likelihood of an individual to infect one other on contact.
- c is the average *degree of contact* between susceptible individuals; in other words, the **average degree of the graph.**
- d is the duration of contact—we'll just assume $d = 1$.

If you remember from 2020-2021, we reduce the number of cases of a disease by bringing down R_0.

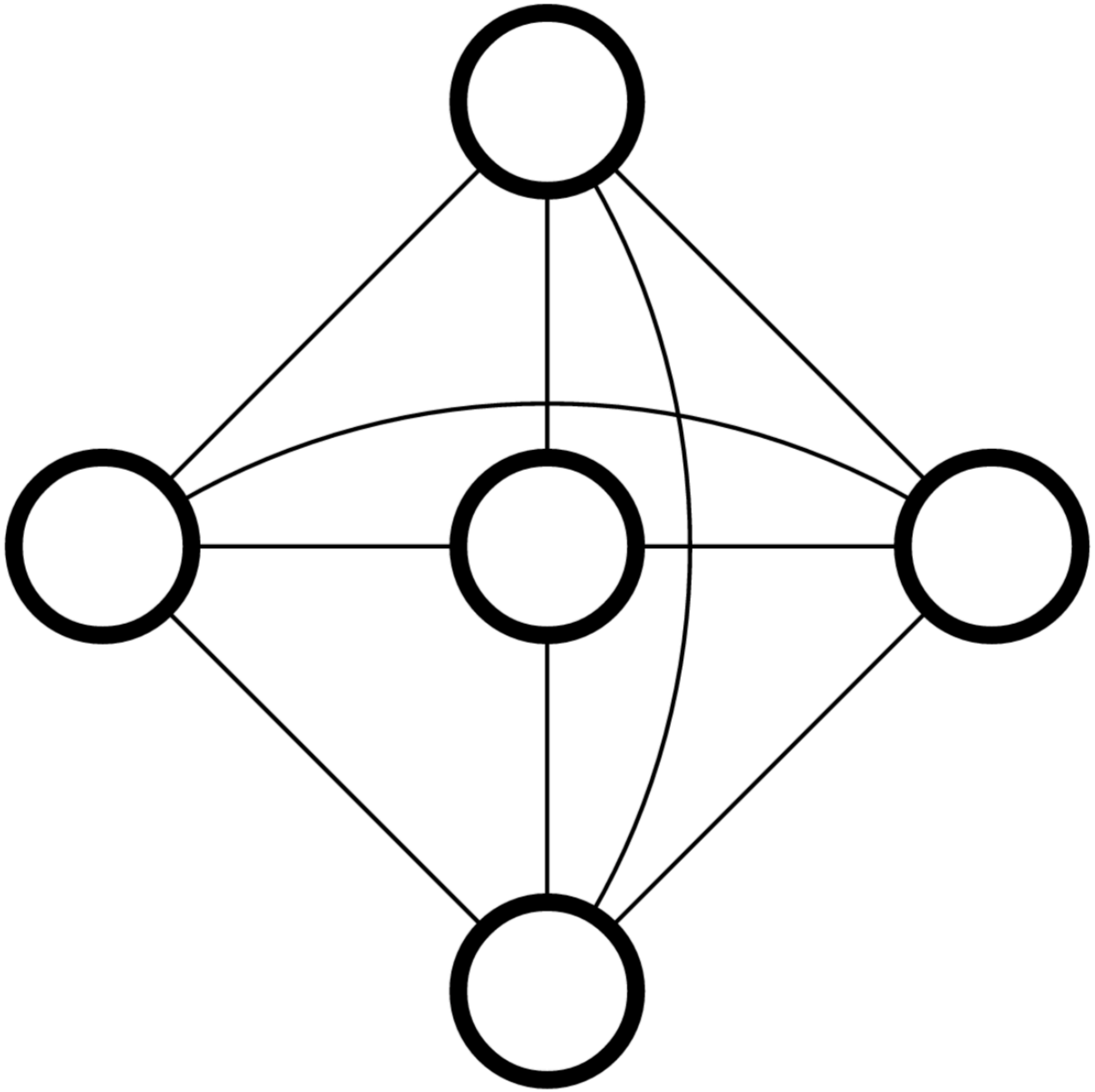
Immunization: Removing Nodes from the Graph

We'll (incorrectly) assume that immunization is perfect, which means that immunization $\rightarrow$ removing a node

1. Remove all nodes with a high degree
    i. Give vaccines to popular people
2. Reduce the number of cliques in the graph (clustering coefficient)
    i. Give vaccines to people whose friends hang out a lot
3. Remove nodes with high degrees that are less likely to be a member of a clique
    i. Give vaccines to people who are important connections between friend groups.

Clique

- A **clique** is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent



Clustering Coefficient

The **clustering coefficient** (between 0 and 1) that measures the degree to which nodes in a graph tend to cluster together

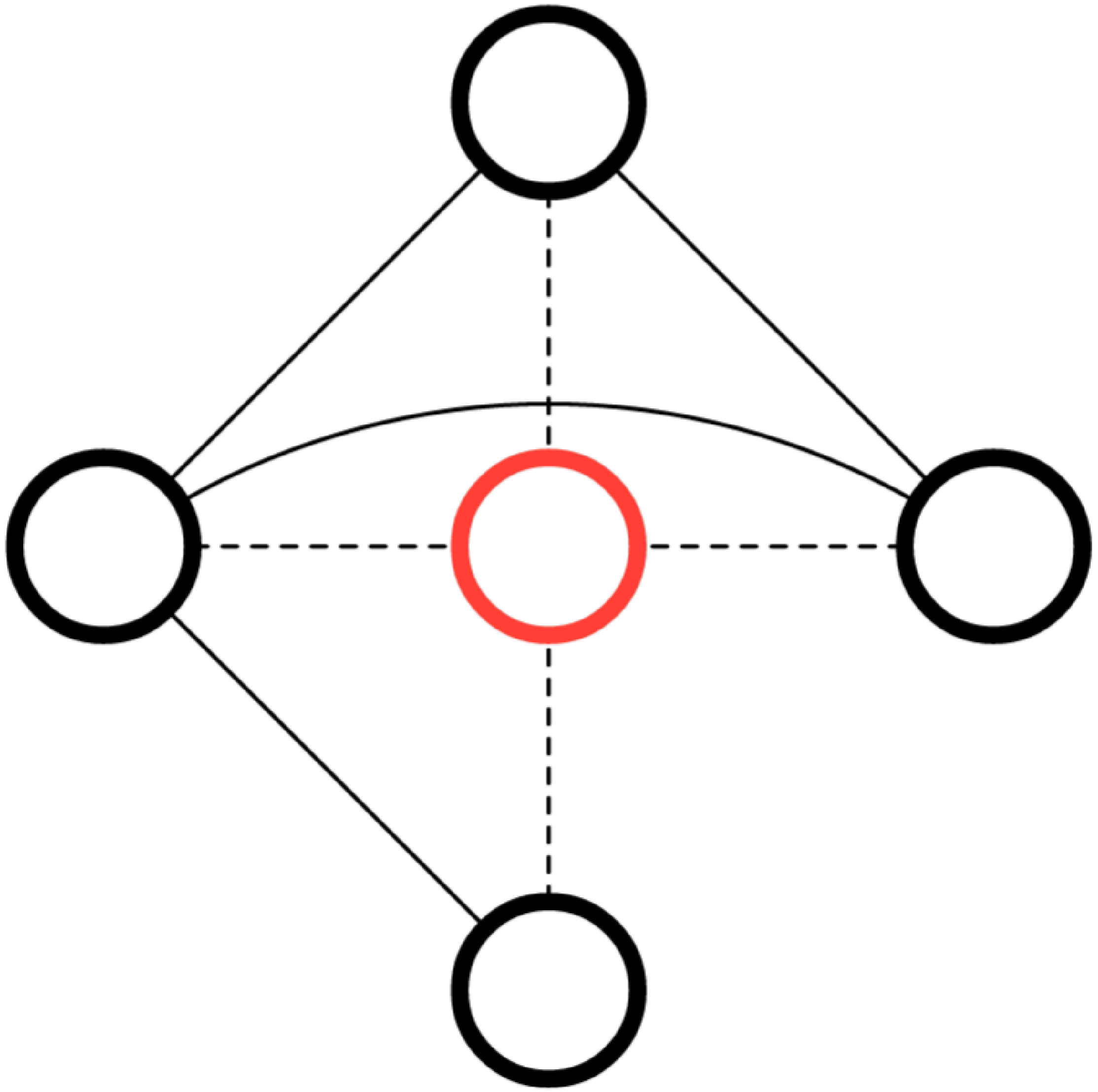
- quantifies how close the neighbors of a node are to being a clique

$$C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{|N_i|(|N_i| - 1)}$$

Clustering Coefficient

The **clustering coefficient** of the red node to the right?

$$C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{|N_i|(|N_i| - 1)}$$

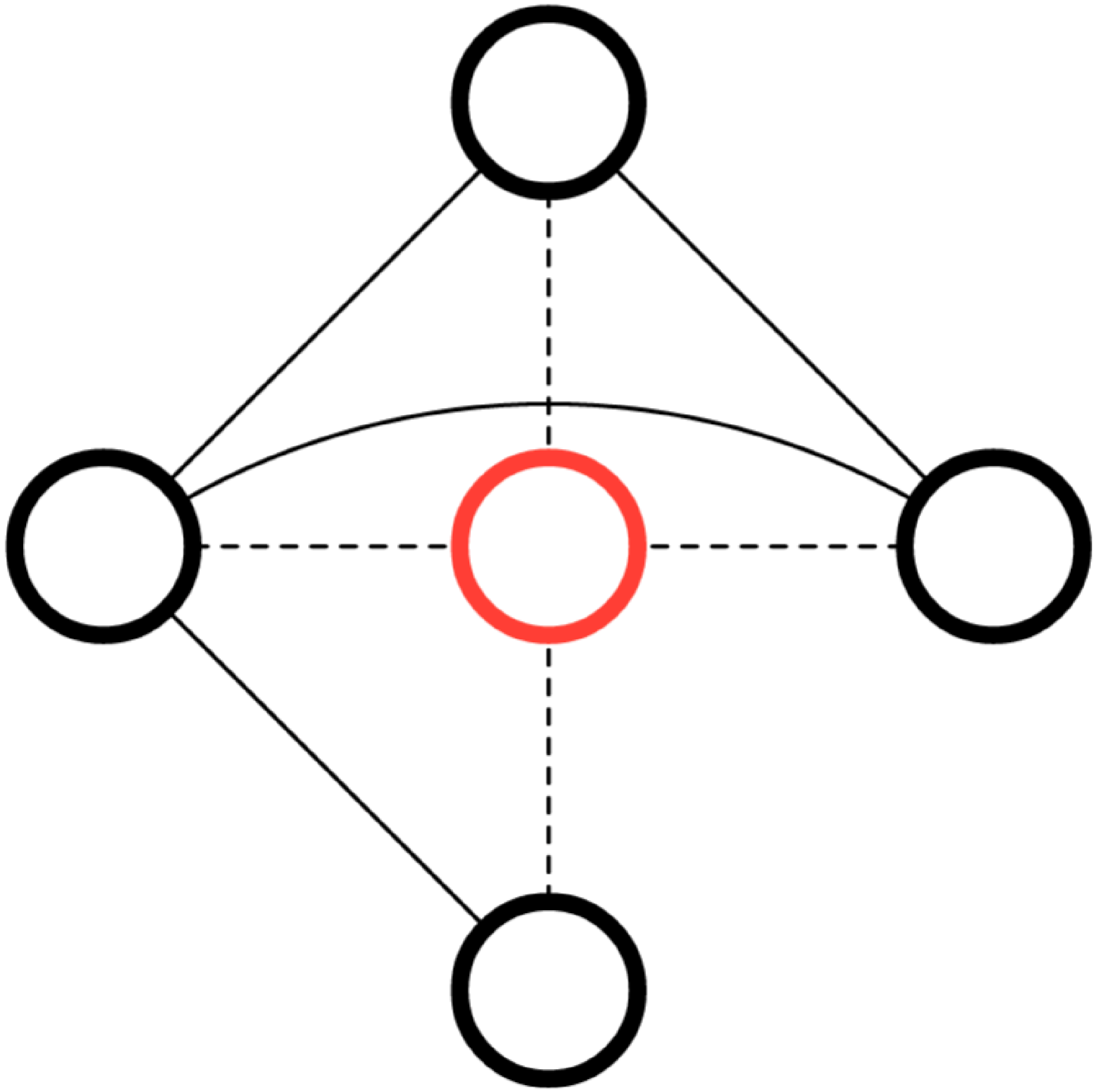


Clustering Coefficient

The **clustering coefficient** of the red node to the right?

- Ignore the dashed edges
- There are four undirected edges (so eight directed...)
- There are $4 * 3 = 12$ possible edges.
- ➡️ $C_i = \frac{8}{12} = 0.666667$

$$C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{|N_i|(|N_i| - 1)}$$



HW7 Goal

- Which strategy will be the most effective?
- References
  - https://web.stanford.edu/class/earthsys214/notes/R0.html
  - https://en.wikipedia.org/wiki/Clustering_coefficient

Which Is Most Effective?

1. Remove all nodes with a high degree
2. Reduce the number of cliques in the graph (clustering coefficient)
3. Remove nodes with high degrees that are less likely to be a member of a clique

… you're going to find out!

Back to Dijkstra's

```java
public static Map<Vertex, PathVertexInfo> dijkstraShortestPath(Graph graph, Vertex startVertex) {
        Map<Vertex, PathVertexInfo> info = new HashMap<>();
        Set<Vertex> visited = new HashSet<>();
        for (Vertex v : graph.getVertices()) {
            info.put(v, new PathVertexInfo(v));
        }

        PathVertexInfo start = info.get(startVertex);
        start.setDistance(0);

        for (int i = 0; i < graph.getVertices().size(); i++) {
            Vertex nextClosest = minVertex(graph, info, visited);
            visited.add(nextClosest); // mark as visited
            PathVertexInfo nextClosestInfo = info.get(nextClosest);
            if (nextClosestInfo.getDistance() == Double.POSITIVE_INFINITY) {
                return info;
            }
            for (Edge e : graph.getEdgesFrom(nextClosest)) {
                Vertex neighbor = e.toVertex;
                double weight = e.weight;
                double oldDistance = info.get(neighbor).getDistance();
                double newDistance = nextClosestInfo.getDistance() + weight;
                if (oldDistance > newDistance) {
                    info.get(neighbor).setDistance(newDistance);
                    info.get(neighbor).setPredecessor(nextClosest);
                }
            }
        }
    }
```

Back to Dijkstra's

```java
// setup: init data structures
Map<Vertex, PathVertexInfo> info = new HashMap<>();
Set<Vertex> visited = new HashSet<>();
for (Vertex v : graph.getVertices()) {
    info.put(v, new PathVertexInfo(v));
}
```

Back to Dijkstra's

```
// prime the search
PathVertexInfo start = info.get(startVertex);
start.setDistance(0);
```

Back to Dijkstra's

```
for (int i = 0; i < graph.getVertices().size(); i++) {
    // pull out the next vertex...
    Vertex nextClosest = minVertex(graph, info, visited);
    visited.add(nextClosest);
    PathVertexInfo nextClosestInfo = info.get(nextClosest);
    // ...and stop if we find an unreachable vertex
    if (nextClosestInfo.getDistance() == Double.POSITIVE_INFINITY) {
        return info;
    }
    ...
}
```

Back to Dijkstra's

```java
// prime the search
for (int i = 0; i < graph.getVertices().size(); i++) {
    ...
    // update all edges with a new path if found
    for (Edge e : graph.getEdgesFrom(nextClosest)) {
        Vertex neighbor = e.toVertex;
        double weight = e.weight;
        double oldDistance = info.get(neighbor).getDistance();
        double newDistance = nextClosestInfo.getDistance() + weight;
        if (oldDistance > newDistance) {
            info.get(neighbor).setDistance(newDistance);
            info.get(neighbor).setPredecessor(nextClosest);
        }
    }
}
```

Processing the Table

How to get from a table of predecessor pointers to an actual path?

Node	Distance	Predecessor
A	0	null
B	8	A
C	4	D
D	3	A
E	6	C
F	10	E
G	11	F

Processing the Table

```
public static String getShortestPath(Vertex endVertex,
                                     Map<Vertex, PathVertexInfo> infoMap) {
    StringBuilder path    = new StringBuilder();
    Vertex        current = endVertex;
    while (current != null) {
        path.append(current);
        current = infoMap.get(current).getPredecessor();
    }
    return path.toString();
}
```

Activity: what's wrong with the approach above?

Processing the Table

```
public static String getShortestPath(Vertex endVertex,
                                     Map<Vertex, PathVertexInfo> infoMap) {
    StringBuilder path    = new StringBuilder();
    Vertex        current = endVertex;
    while (current != null) {
        path.insert(0, current);
        current = infoMap.get(current).getPredecessor();
    }
    return path.toString();
}
```

Now the path is no longer reversed...

Finding the Min Vertex: Runtime?

```
private static Vertex minVertex(Graph graph, Map<Vertex,PathVertexInfo> info,
                                Set<Vertex> visited) {
    Vertex closest = null;
    double minDistance = Double.POSITIVE_INFINITY;
    for (Vertex candidate : info.keySet()) {
        if (visited.contains(candidate)) {
            continue;
        }
        PathVertexInfo data = info.get(candidate);
        if (closest == null || data.getDistance() < minDistance) {
            closest = candidate;
            minDistance = data.getDistance();
        }
    }
    return closest;
}
```

Finding the Min Vertex: Runtime?

```
private static Vertex minVertex(Graph graph, Map<Vertex,PathVertexInfo> info,
                                Set<Vertex> visited) {
    Vertex closest = null;
    double minDistance = Double.POSITIVE_INFINITY;
    for (Vertex candidate : info.keySet()) {       // max O(V) iterations
        if (visited.contains(candidate)) {         // O(1) for HashSet
            continue;
        }
        PathVertexInfo data = info.get(candidate);  // O(1) for HashMap
        if (closest == null || data.getDistance() < minDistance) {
            closest = candidate;
            minDistance = data.getDistance();
        }                                          // all this is O(1)
    }
    return closest;
}
```

Finding the Min Vertex: Runtime?

```
private static Vertex minVertex(Graph graph, Map<Vertex,PathVertexInfo> info,
                                Set<Vertex> visited) {
    ...
}
```

Simple solution gives $O(|V|)$ runtime per call—called V times, so $O(|V|^2)$ indeed works as a real-world upper bound.

Functional Programming & Streams

At a high-level, to implement `minVertex`, we:

- filter the set of all vertices to only include unvisited ones.
- find the minimum vertex based on the shortest distance of its `PathVertexInfo`

Can we express the ideas of these two *highly common* ideas a little more succinctly?

Streams

"A sequence of elements supporting sequential and parallel aggregate operations."

To perform a computation, stream operations are composed into a **stream pipeline:**

- a source (which might be an array, a collection, a generator function, an I/O channel, etc)
- zero or more intermediate operations (which transform a stream into another stream, such as `filter(Predicate)`),
- and a terminal operation (which produces a result or side-effect, such as `count()` or `forEach(Consumer)`).

Streams

```java
private static Vertex minVertex(Map<Vertex,PathVertexInfo> info,
                                 Set<Vertex> visited) {
    return info.keySet().stream()
                    ...
}
```

Filter

A function that *takes in a function* that returns a boolean deciding whether an element should be included.

How to pass in a function as an input to another? **Lambdas** (anonymous functions)

An example predicate lambda:

```
//  input   -> boolean expression using input var
    element -> element < 0 || element % 2 == 1
```

Streams

```
private static Vertex minVertex(Map<Vertex,PathVertexInfo> info,
                                 Set<Vertex> visited) {
    return info.keySet().stream()
               .filter(vertex -> !visited.contains(vertex))
               ...
}
```

Finding the Minimum

`min()` allows you to find the minimum element of a stream using a comparator.

Could write a comparator using a lambda:

```
(e1, e2) -> info.get(e1).compareTo(info.get(e2))
```

Finding the Minimum

`min()` allows you to find the minimum element of a stream using a comparator.

Could write a comparator using a lambda:

```
(e1, e2) -> info.get(e1).compareTo(info.get(e2))
```

Can also pass a named function as a reference:

```
Comparator.comparing(info::get)
```

"Make a comparator by comparing the result of calling `info.get()` with the element as an input."

Streams

```java
private static Vertex minVertex(Map<Vertex,PathVertexInfo> info,
                                 Set<Vertex> visited) {
    return info.keySet().stream()
                .filter(vertex -> !visited.contains(vertex))
                .min(Comparator.comparing(info::get))
                .orElse(null); // min might not find an answer, so return null
}
```