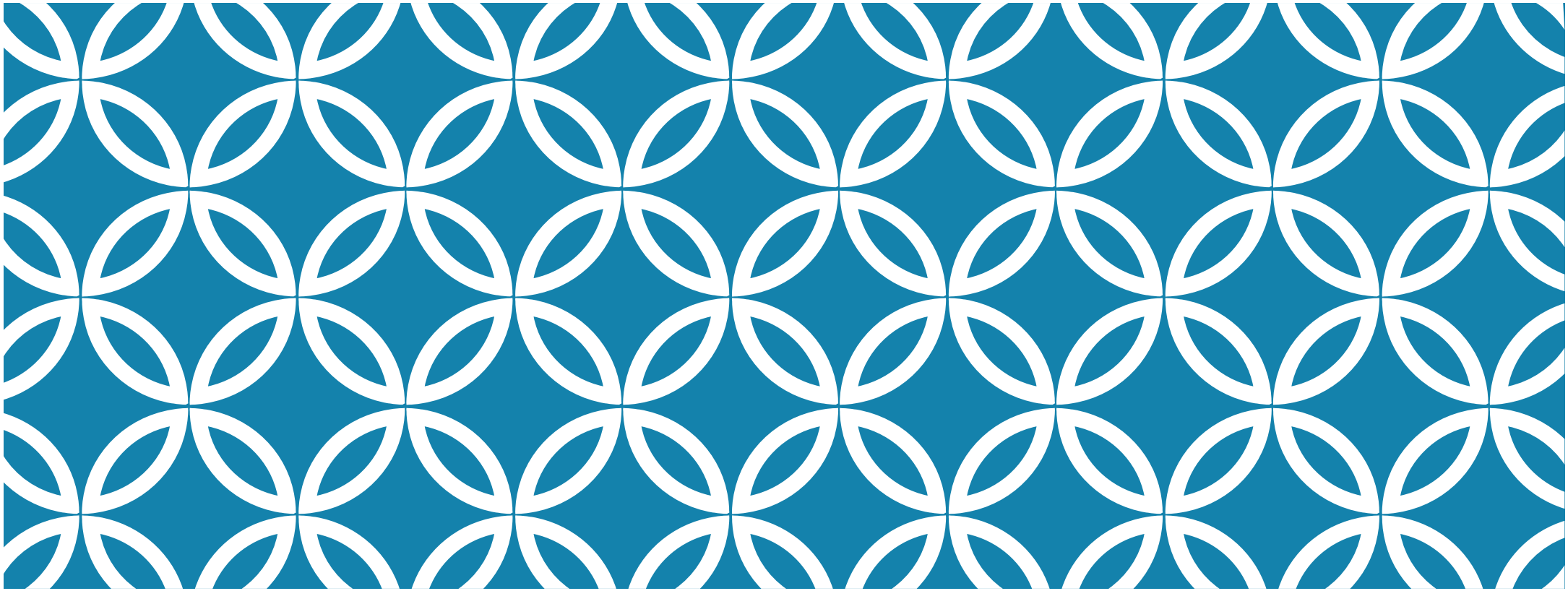


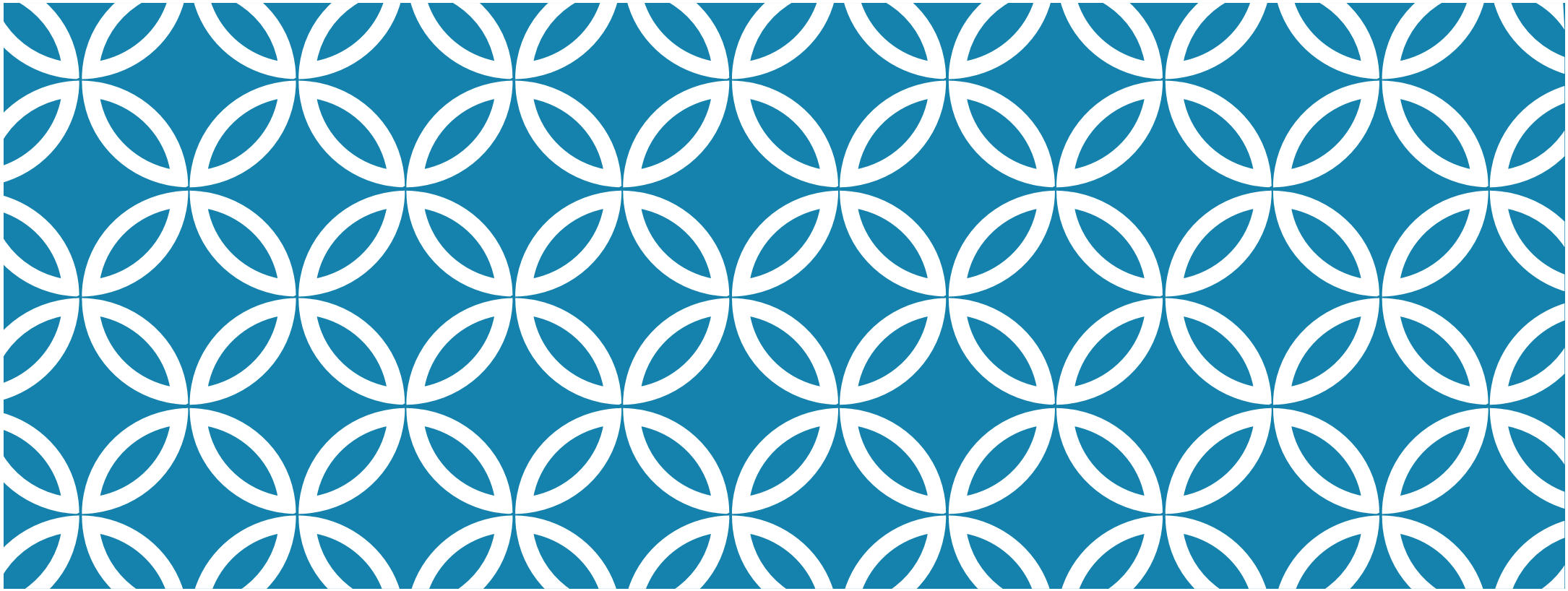
TAKE HOME LAB 4

Solutions

TAKE HOME LAB 4

- 3 Questions to give you practice on Graph Algorithms
  - Topological Ordering
  - Applications of DFS/BFS
  - Dijkstra Algorithm

- Submit by 18th April 2019, 2359h



PANDACHESS

PROBLEM STATEMENT

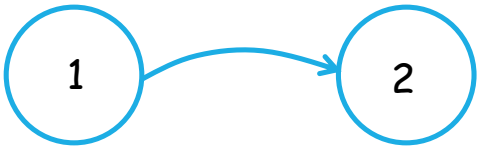
- N players
- M matches between players
- One winner and one loser for each match

- Want to find a ranking such that:

Winner's rank always higher than loser's rank for all matches

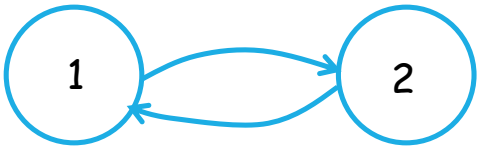
EXAMPLE

- Match1: Player 1 beats Player 2
- Player 1 should have higher rank than Player 2
- Possible ranking: 1, 2



EXAMPLE

- Match1: Player 1 beats Player 2
- Player 1 should have higher rank than Player 2



- Match2: Player 2 beats Player 1
- Player 2 should have higher rank than Player 1

- Impossible!

SOME IDEAS

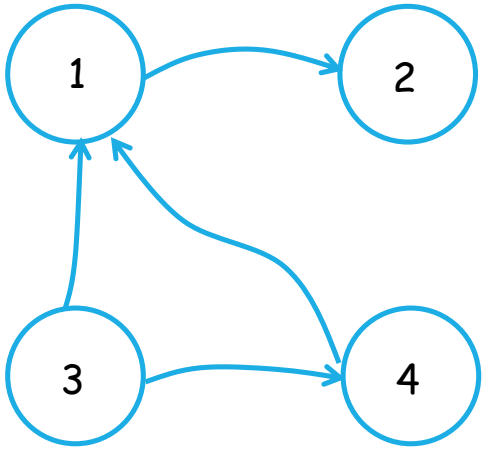
- N players, M matches, each matches between two players

It's a directed graph!

Players as vertices

Matches as directed edges

If player u beats player v, add an (u->v) edge

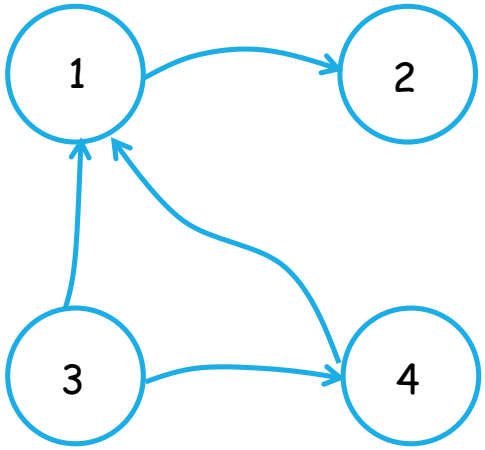


SOME IDEAS

- What is the 'ranking' we want?

If we have edge (u->v), then u must have higher rank than v

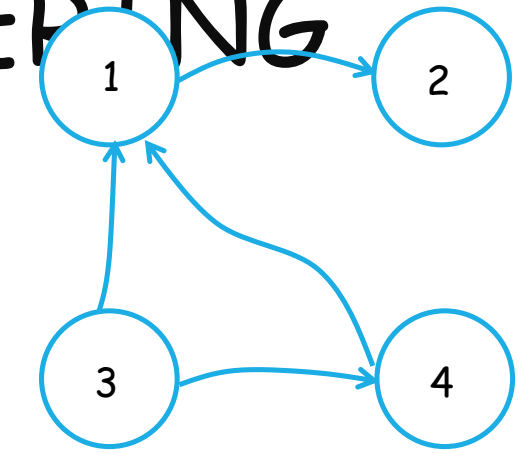
It is the topological order!



TOPOLOGICAL ORDERING

- Definition:

For every directed edge (u->v) in the directed graph, u comes before v in the ordering



Topological ordering: 3, 4, 1, 2

- Algorithm:

Keep removing vertices which have no incoming edge and update the graph.

IMPLEMENTATION

- BFS

Maintaining the number of incoming edges for each vertex

Start with the vertices has no incoming edges

For each vertex during BFS, remove all edges go out from the vertex and update the number of incoming edges for other vertices

After removing, find all vertices which have no incoming edges now and start BFS from them

TIME COMPLEXITY

- BFS implementation:

Store edges with adjacent matrix – $O(N^2)$

Find no incoming edges vertices by maintaining number of incoming edges for each nodes - $O(N)$ for each round, $O(N^2)$ in total

Remove one vertex and update adjacent matrix - $O(N)$ for one vertex, $O(N^2)$ in total

In total $O(N^2)$

- How to improve?

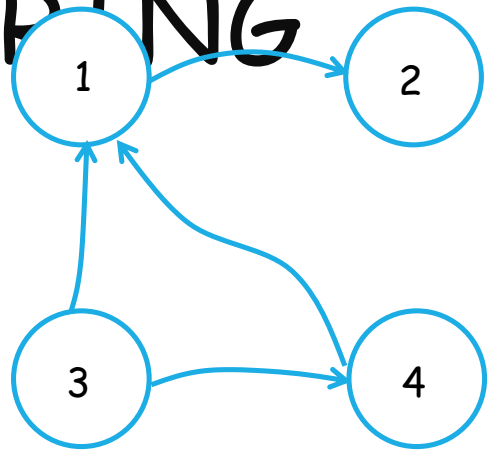
BETTER IMPLEMENTATION

- Still using BFS
- Using adjacent list instead of adjacent matrix
- Maintaining the number of incoming edges for each vertex
- Remove edges and records the vertices **now** has no incoming edges by the meantime

TOPOLOGICAL ORDERING

- Number of incoming edges:

1	2	3	4
2	1	0	1



- Vertices with no incoming edges:

{3}

- Topological ordering:

{}

REMOVING VERTEX 3

- Number of incoming edges:

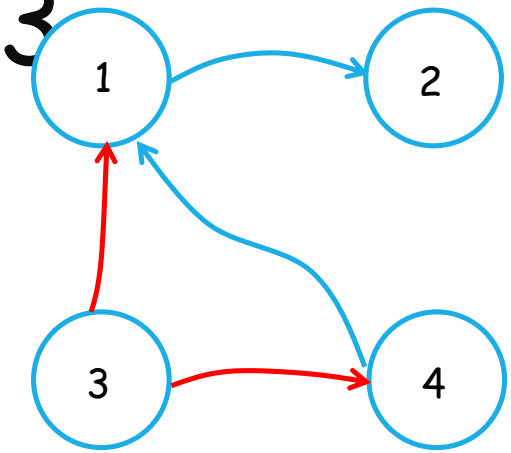
1	2	3	4
1	1	0	0

- Vertices with no incoming edges:

{3, 4}

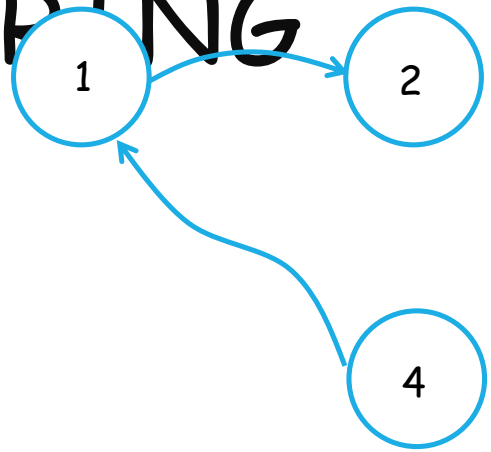
- Topological ordering:

{}



Using adjacent list to find all edges start from vertex 3
Update respective number of incoming edges
Find vertex 4 has no incoming edges now

TOPOLOGICAL ORDERING



- Number of incoming edges:

1	2	3	4
1	1	0	0

- Vertices with no incoming edges:

{4}

- Topological ordering:

{3}

TOPOLOGICAL ORDERING

- Number of incoming edges:

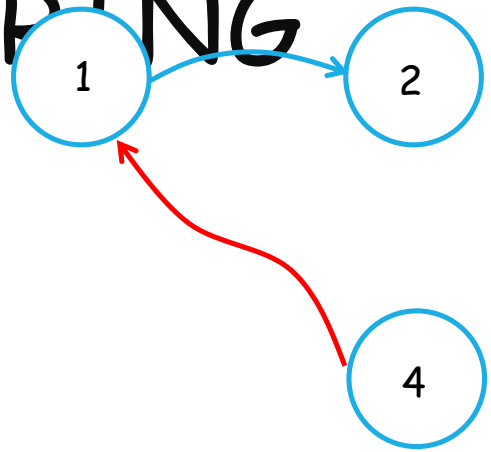
1	2	3	4
0	1	0	0

- Vertices with no incoming edges:

{4, 1}

- Topological ordering:

{3}



Using adjacent list to find all edges start from vertex 4
Update respective number of incoming edges
Find vertex 1 has no incoming edges now

TOPOLOGICAL ORDERING

- Number of incoming edges:

1	2	3	4
0	1	0	0

- Vertices with no incoming edges:

{1}

- Topological ordering:

{3, 4}

TOPOLOGICAL ORDERING

- Number of incoming edges:

1	2	3	4
0	0	0	0

- Vertices with no incoming edges:

{1, 2}

- Topological ordering:

{3, 4}

Using adjacent list to find all edges start from vertex 4
Update respective number of incoming edges
Find vertex 1 has no incoming edges now

TOPOLOGICAL ORDERING

- Number of incoming edges:

1	2	3	4
0	0	0	0

- Vertices with no incoming edges:

{2}

- Topological ordering:

{3, 4, 1}

TOPOLOGICAL ORDERING

- Number of incoming edges:

1	2	3	4
0	0	0	0

- Vertices with no incoming edges:

{}

- Topological ordering:

{3, 4, 1, 2}

TIME COMPLEXITY

- BFS implementation:

Store edges with adjacent list – $O(N+M)$

Maintaining number of incoming edges for each nodes - $O(N)$

Remove one vertex, update adjacent list and number of incoming edges - $O(N+M)$ in total

Find the vertices with no incoming edges when updating adjacent list - $O(N+M)$ in total

In total $O(N+M)$

ISSUES

- Topological sort only works for acyclic graph!

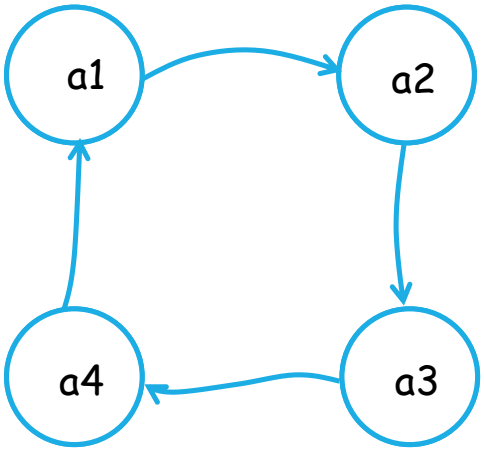
- What happens if the graph has cycle?

There vertices in cycle will not be removed after BFS

- When will the graph has a cycle?

CYCLE

- If we have a cycle with l vertices $a_1, a_2, a_3, \ldots, a_l$



- Have to rank a_1 higher than a_2; a_2 higher than a_3;…; a_{l-1} higher than a_l; a_l higher than a_1

$$a_1 > a_2 > a_3 > \cdots > a_l > a_1$$

- By contradiction, it's impossible to rank them

ISSUES

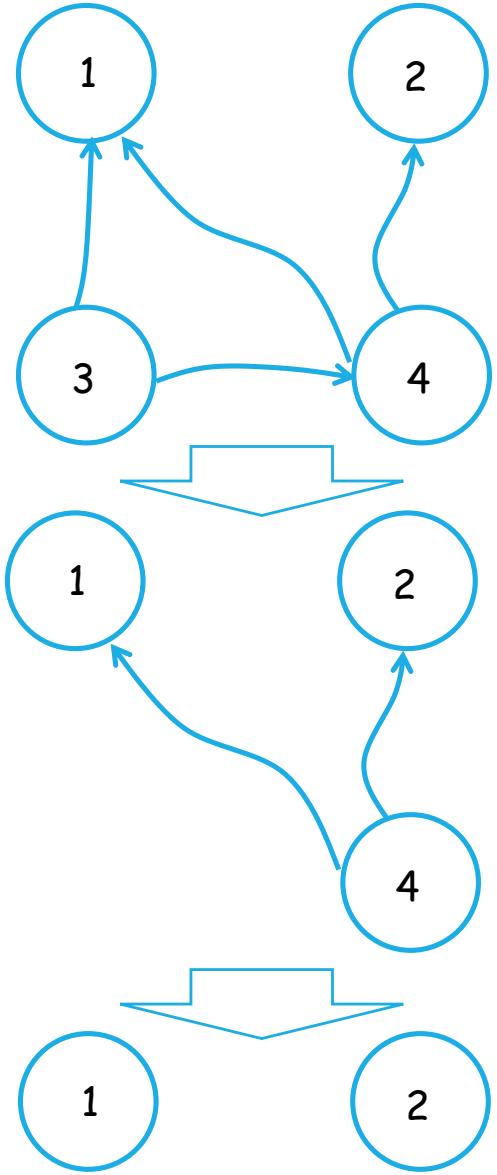
- How to determine whether the ranking is unique or not?

UNIQUE RANKING

- If we have choices of vertices to remove, the solution is not unique!

After removing vertex 4, we can choose vertex 1 or vertex 2

So both 3, 4, 1, 2 and 3, 4, 2, 1 are OK



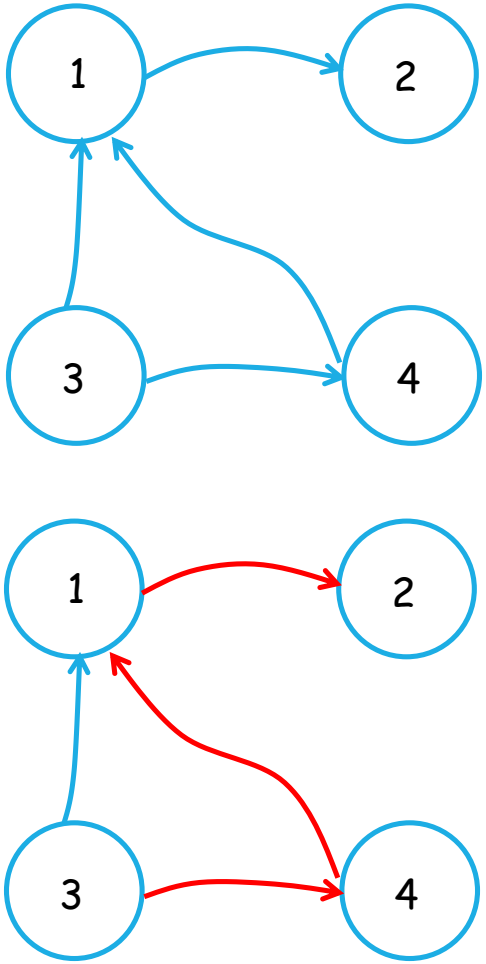
UNIQUE RANKING

- If we don't have choice, the solution is unique

There must be a edge between consecutive vertices in the ranking: (3->4), (4->1), (1->2) (otherwise we will have multiple choices)

These edges forms a chain go through all vertices and we have:

$$a_1 > a_2 > \cdots > a_n$$



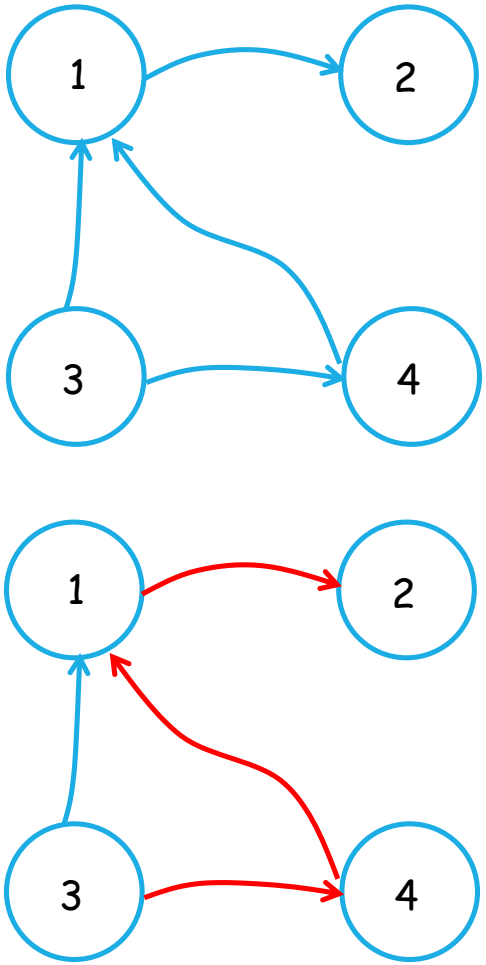
UNIQUE RANKING

- If we don't have choice, the solution is unique

By contradiction, if we have another solution b. We can assume the **first** vertex b_i is not the same as a_i is equal to $a_j (j > i)$. There must be a b_k is the same as a_i and $k > i$:

$$b_1 = a_1, b_2 = a_2 \dots, b_i = a_j, \dots, b_k = a_i \dots$$

From the definition, we put $b_i = a_j$ before $b_k = a_k$.But we also have a_i must be put before $a_j (a_i > a_j)$. Contradiction!

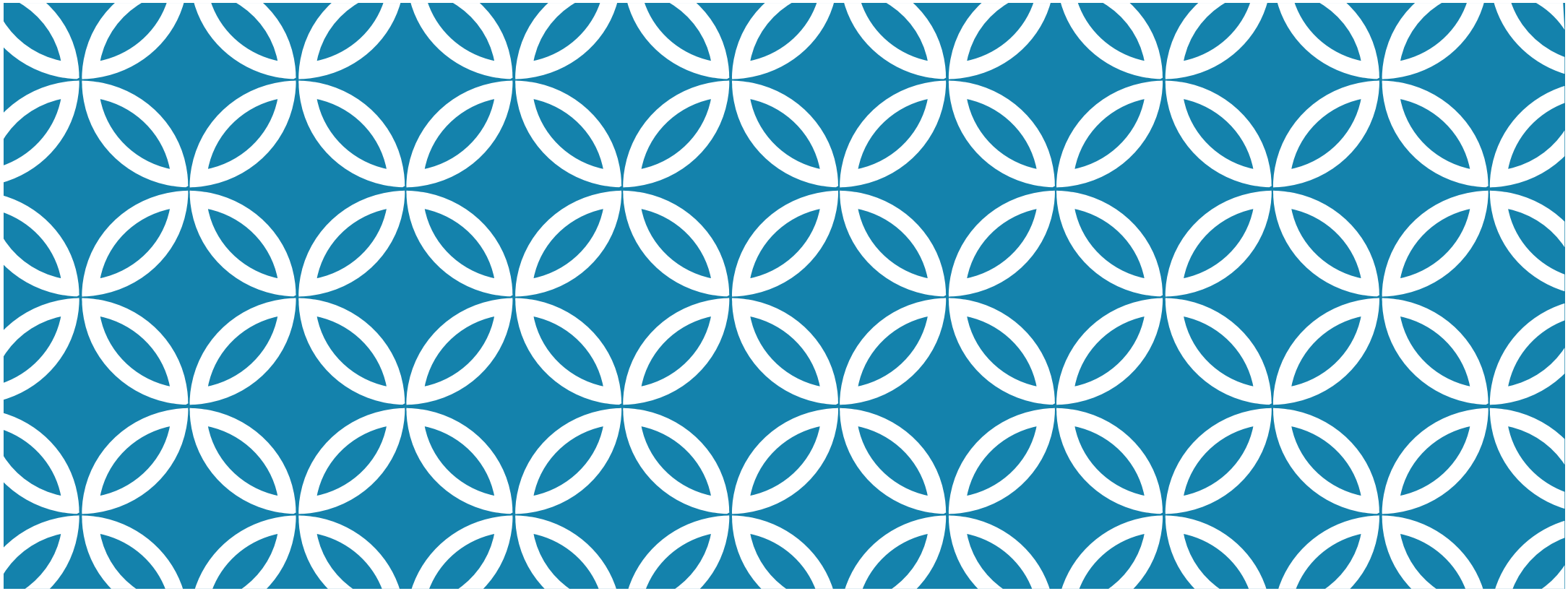


SUMMARY

- Topological sort with adjacent list
- If there is circle (topological sort did not cover all vertices): no solution
- During topological sort (BFS), if there are more than one vertices in queue waiting for process: solution not unique

READING MATERIAL

- https://en.wikipedia.org/wiki/Topological_sorting
- https://en.wikipedia.org/wiki/Partially_ordered_set



PANDA ISLANDS

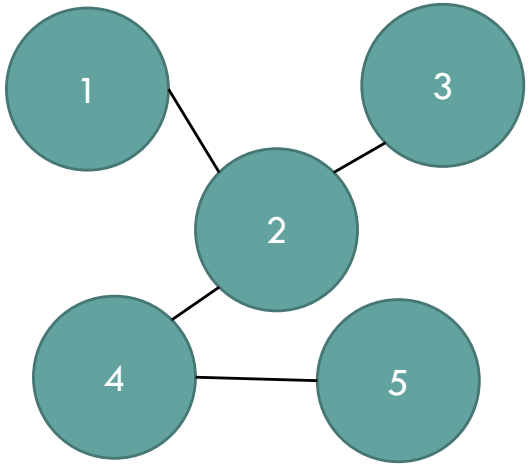
PROBLEM

- **N** islands, 1 <= N <= 50,000
- Each Island has an area **X**
- **E** bridges, 1 <= E <= 300,000
- Any two islands connected by a bridge must be a different color (Black or White)
- Guaranteed that valid coloring exists
- Islands are initially white, find area of Black Paint needed

OBSERVATIONS

- If we pick a island to be black, the surrounding islands connected by a bridge must be white and vice versa
- For this example, area of each island is equal to its index

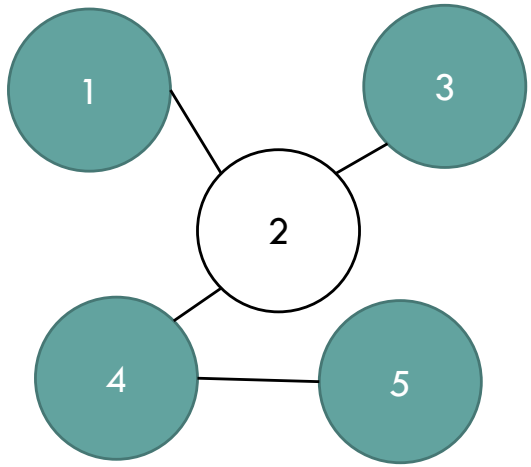
Initially everything is white, but to be clear, lets make un-chosen islands red



OBSERVATIONS

- If we pick a island to be black, the surrounding islands connected by a bridge must be white and vice versa

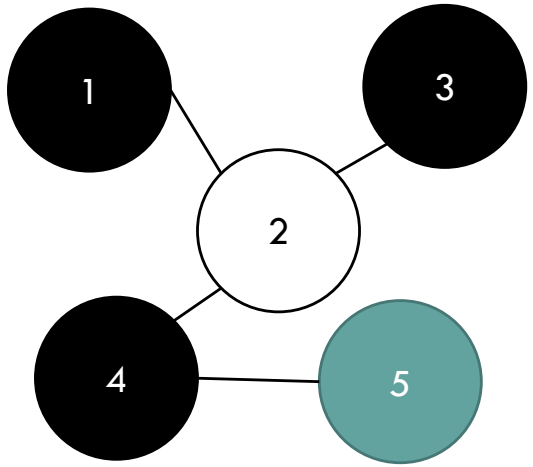
Pick 2 to be white



OBSERVATIONS

- If we pick a island to be black, the surrounding islands connected by a bridge must be white and vice versa

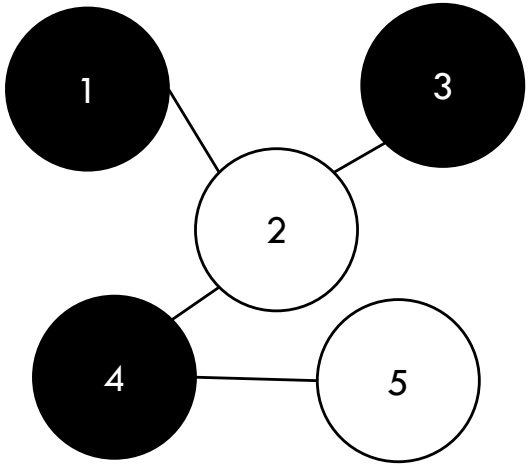
Pick 2 to be white -> 1, 3, 4 must be black



OBSERVATIONS

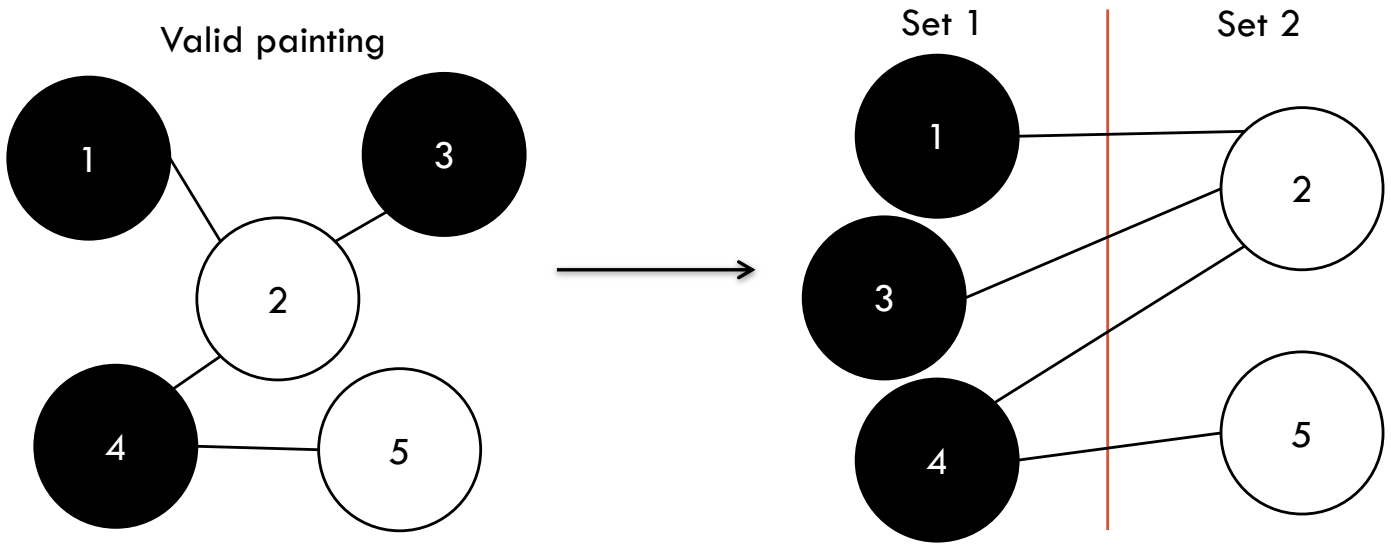
- If we pick a island to be black, the surrounding islands connected by a bridge must be white and vice versa

4 is black -> 5 is white



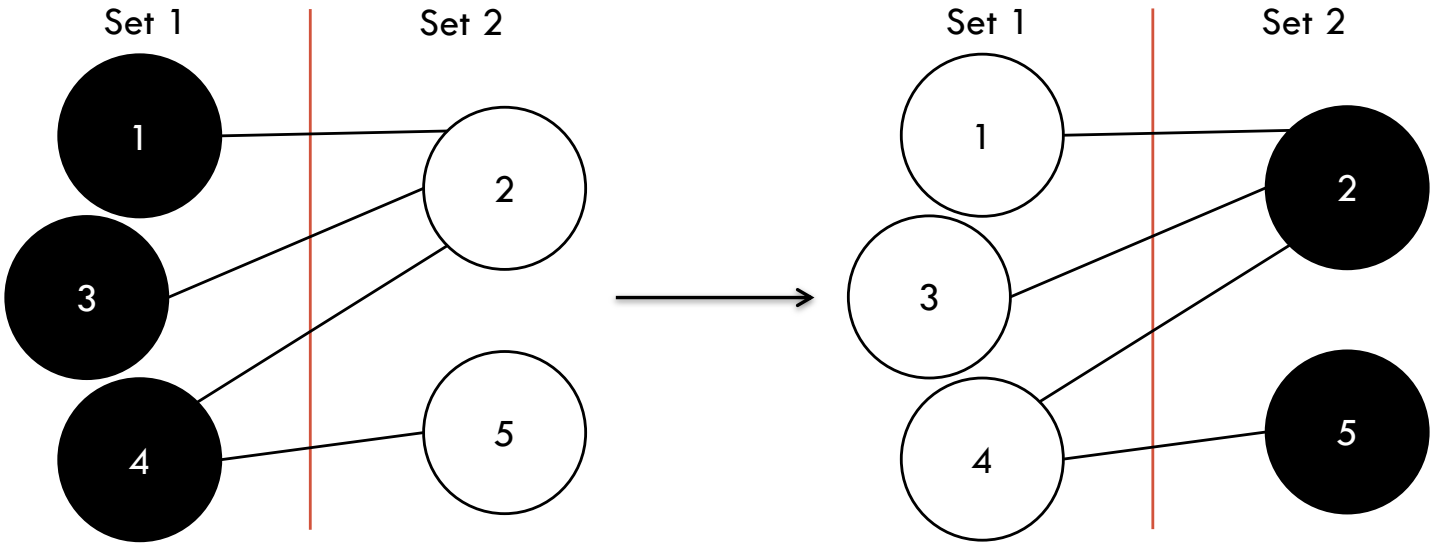
OBSERVATIONS

- Having a valid way to paint the graph means that it is a bipartite graph
- This just means that the nodes of the graph can be split into 2 sets where there only exists edges between the 2 sets.



OBSERVATIONS

- Colors can be switched, so we should color the nodes in the smaller set to be black. Note that either coloring shown here is valid. *No need to actually color, just take the min

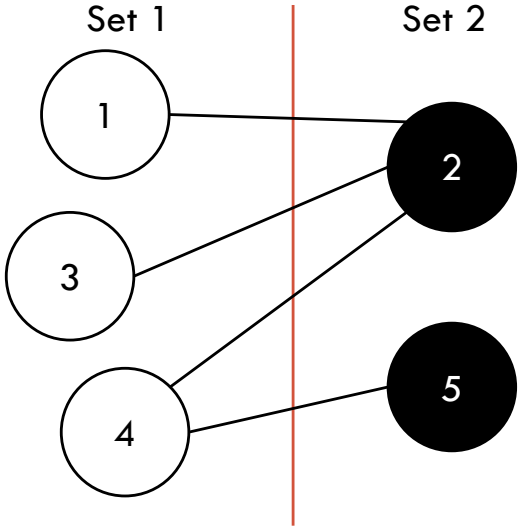


Black area: 8

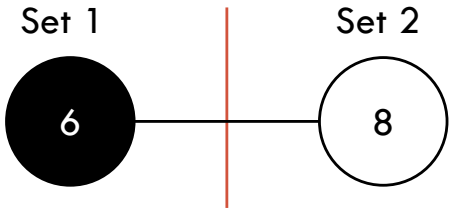
Black area: 7

OBSERVATIONS

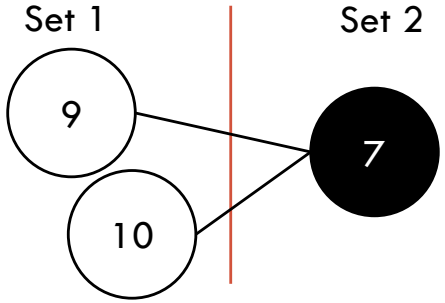
- That only works when the graph is connected. What should we do?
  - Treat each unconnected component as it's own graph and add all the areas together. **Total black area: *7* + 6 + *7* = 20**



Black area: min(8, 7)



Black area: min(6, 8)



Black area: min(19, 7)

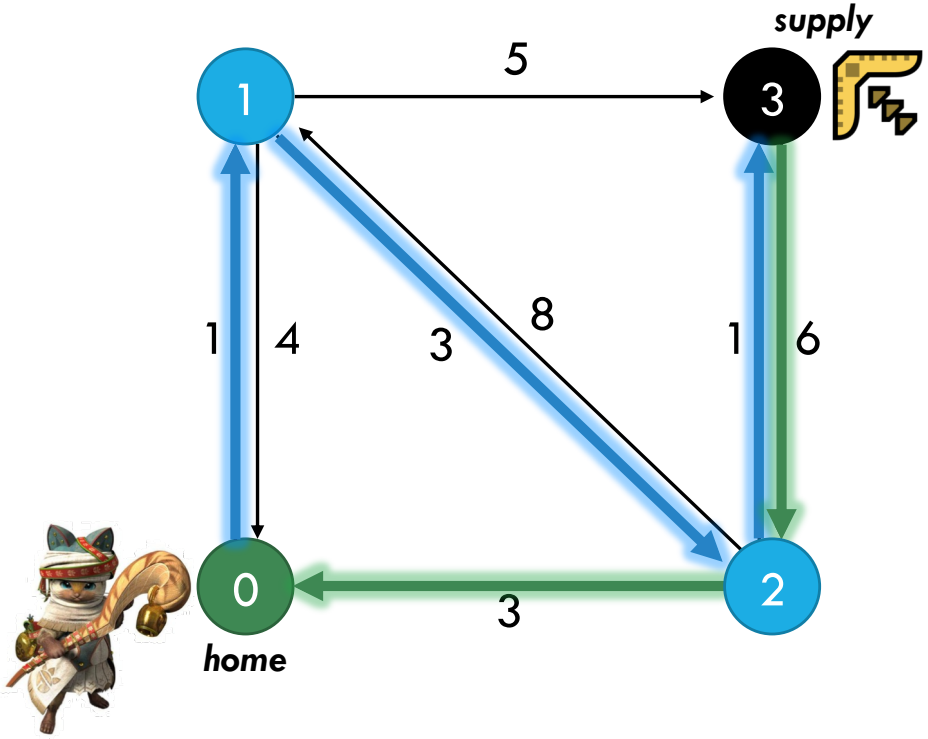
ALGORITHM

- What is the algorithm to paint the graph?
  - DFS/BFS (need to keep track of color and paint respectively)
- How to keep track of area of each set?
  - Store them in a class which keeps track of area of set 1 and area of set 2
- Take note that the graph is unconnected
  - Use a for loop around your initial DFS/BFS call (This will make it so that you go through every component once)
  - Sum the min of the 2 areas you found after every iteration
- The rest is your standard DFS/BFS!

PROBLEM STATEMENT

- You are given a *directed, weighted* graph (with non-negative weights).
- Rar the Cat wishes to start from a ***home*** vertex, go to a specified ***supply*** vertex, and go back to the ***home*** vertex.
- Edges can be used more than once.
- Minimise the total sum of the weights of the edges used to complete the journey.
- $0 < N < 1000$, $0 \leq E \leq N(N - 1)$

Answer = **1** + **3** + **1** + **6** + **3** = 14



OBSERVATION

- The path from ***home*** → ***supply*** does not affect the path from ***supply*** → ***home***.
- The two paths can be calculated independently.
- The answer to the problem is simply the sum of
  - The weight of the shortest path from ***home*** → ***supply***
  - The weight of the shortest path from ***supply*** → ***home***

WHICH **ALGORITHM** TO USE?

Type	Name	Conditions	Time Complexity
APSP	Floyd-Warshall's	None	$O(V^3)$
SSSP	Bellman-Ford's	None	$O(VE)$
SSSP	Dijkstra's	Non-negative Edge Weights	$O((V + E) \log V)$*
SSSP	Process in Topological Order	Directed Acyclic Graph (DAG)	$O(V + E)$
SSSP	BFS	Unweighted Graph (or all edge weights are equal)	$O(V + E)$
SSSP	BFS/DFS	Tree	$O(V + E)$

* (Outside scope of CS2040) The use of a *Fibonacci Heap* can speed this up to $O(E + V \log V)$

ALGORITHM **OUTLINE**

1. Perform Dijkstra's Algorithm starting from the ***home*** vertex to find the shortest distance from ***home*** → ***supply***.
    1.1 If the distance estimate of the ***supply*** is still "Infinite" after the algorithm terminates, the answer is "Impossible". (Alternatively, check if the ***supply*** vertex was ever processed (for certain implementations).)
2. Perform Dijkstra's Algorithm starting from the ***supply*** vertex to find the shortest distance from ***supply*** → ***home***.
    2.1 The same check applies
3. Sum up the values to get the answer.

TIME COMPLEXITY ANALYSIS

- Generation of the Adjacency List takes $O(V + E)$ time.
- We're performing 2 iterations of Dijkstra's Algorithm, each takes $O((V + E) \log V)$.
- Total time complexity is $O((V + E) \log V)$

IMPLEMENTATION DETAILS

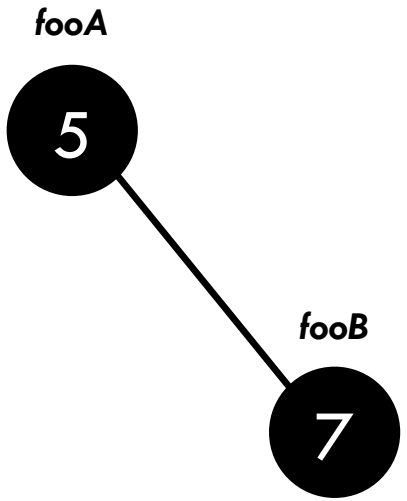
- Dijkstra's Algorithm requires you to support two operations:
  - Decrease the distance estimate of a vertex.
  - Retrieve and remove an unprocessed vertex with the lowest distance estimate.

IMPLEMENTATION **WARNING!**

- The elements in a Priority Queue (or Tree Set, or Tree Map, etc.) are not meant to be modified.
- If you modify the elements stored in your Priority Queue such that the ordering of the elements may change, your data structure will have ***undefined behaviour.***

IMPLEMENTATION **WARNING!**

- Consider the following example, where I have a ***Foo*** class that contains a single integer member called "value".
- ***Foo*** objects are compared by "value" in ascending order.
- The TreeSet on the right represents a possible structure for a TreeSet that contains two ***Foo*** objects, ***fooA*** and ***fooB*** with values 5 and 7 respectively.
- If we were to change ***fooB***'s value to 0...



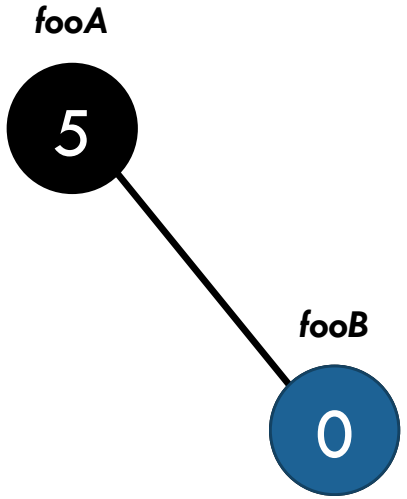
IMPLEMENTATION **WARNING!**

```
Foo fooA = new Foo(5);
Foo fooB = new Foo(7);

TreeSet<Foo> fooSet = ...;

fooSet.add(fooA);
fooSet.add(fooB);

fooB.setValue(0);

return fooSet.contains(fooB);
```

Is this node equal to fooB? No.

fooB is less than this node, so search the left subtree.

Left subtree is empty. Return ***false***.



IMPLEMENTATION DETAILS

- Dijkstra's Algorithm requires you to support two operations:
  - Decrease the distance estimate of a vertex.
  - Retrieve and remove an unprocessed vertex with the lowest distance estimate.
- There are three ways to go about this:
  - Create your own Min Heap that can perform a `decreaseKey(vertex, newKey)` operation.
  - Use a `java.util.PriorityQueue` with Lazy deletion (see Tutorial 8 Q1)
  - Use a `java.util.TreeSet`.
- We'll be using the second (because it's easy to code, and because TreeSet is slow)

IMPLEMENTATION DETAILS

- Your Priority Queue will contain Nodes with the following details:
  - The vertex to be processed.
  - The distance estimate of the vertex.
- We can create a helper class* called DijkNode because I couldn't think of a better name.

```
class DijkNode implements Comparable<DijkNode> {
    private int vertex;
    private int distEstimate;

    public int compareTo(DijkNode other) {
        return this.distEstimate - other.distEstimate;
    }
}
```

* You can also use your Edge class since it's already there.

IMPLEMENTATION DETAILS

- Since the vertices are labelled from 0 to *N* - 1, we'll store the most updated distance estimates in an array of size *N*.

```
final int INF = 1000000000;

int[] distance = new int[N];
for (int i = 0; i < N; ++i) {
    distance[i] = INF;
}
```

IMPLEMENTATION DETAILS

- During initialisation, set the distance of your source as 0 and push the node [source, 0] into your Priority Queue.

```
PriorityQueue<DijkNode> dijkstraPQ = new PriorityQueue<>();

distance[source] = 0;
dijkstraPQ.add(new DijkNode(source, 0));
```

IMPLEMENTATION DETAILS

- While your Priority Queue is not empty, get the next DijkNode.
- If the distance estimate of the DijkNode does not match the most updated value*, ignore it.

```
while (!dijkstraPQ.isEmpty()) {
    DijkNode node = dijkstraPQ.poll();
    int vertex = node.getVertex();
    int distEst = node.getDistEstimate();

    if (distance[vertex] != distEst) {
        continue;
    }
```

* You can also have a boolean array called isProcessed and ignore a DijkNode if the vertex has already been processed.

IMPLEMENTATION DETAILS

- To process a vertex, relax all incident edges and push new DijkNodes into your Priority Queue if a distance estimate is updated.

```
for (Edge edge : adjList.get(vertex)) {
    int nextVertex = edge.getTo();
    int nextDistEst = distEst + edge.getWeight();

    if (nextDistEst < distance[nextVertex]) {
        distance[nextVertex] = nextDistEst;
        dijkstraPQ.add(new DijkNode(nextVertex, nextDistEst));
    }
}
```

IMPLEMENTATION DETAILS

- Finally, return the distance between the source and your specified destination.

```
return distance[destination];
```

QUICK QUESTION!

```
final int INF = 1000000000;

int[] distance = new int[N];
for (int i = 0; i < N; ++i) {
    distance[i] = INF;
}
```

Why is INF set as 1,000,000,000 and not Integer.*MAX_VALUE*?

A. Randomly chosen using a 1,000,000,000-sided die.
B. Oooooooooooooooooooooooverflo
C. Doesn't matter; both are big enough.
D. The zeroes soothe my pain.

QUICK QUESTION!

```
final int INF = 1000000000;

int[] distance = new int[N];
for (int i = 0; i < N; ++i) {
    distance[i] = INF;
}
```

```
for (Edge edge : adjList.get(vertex)) {
    int nextVertex = edge.getTo();
    int nextDistEst = distEst + edge.getWeight();

    if (nextDistEst < distance[nextVertex]) {
```

Why is INF set as 1,000,000,000 and not Integer.*MAX_VALUE*?

A. Randomly chosen using a 1,000,000,000-sided die.
B. Oooooooooooooooooooooooooverf
C. Doesn't matter; both are big enough.
D. The zeroes soothe my pain.

* Disclaimer: I know neither of those values can be INF, but you can see how things may crash and burn in other programs.

WHAT IF I WANT TO USE A TREESET?

- We'll have a TreeSet of DijkNodes, similar to the Priority Queue implementation.
- However, remember that we cannot modify the DijkNodes directly while they're still in the TreeSet.

```
TreeSet<DijkNode> dijkstraTS = new TreeSet<>();
```

WHAT IF I WANT TO USE A TREESET?

- Updating the distance estimate of a vertex in the TreeSet consists of three steps:
  - Remove the relevant DijkNode from the TreeSet.
  - Update the distance estimate.
  - Push the updated DijkNode back into the TreeSet.

```
DijkNode node = new DijkNode(vertex, distance[vertex]);
dijkstraTS.remove(node);

distance[vertex] = newDistEst;
node.setDistEstimate(newDistEst);
dijkstraTS.add(node);
```

WHAT IF I WANT TO USE DECREASE KEY?

- For the sake of completeness, I will also talk about a Dijkstra's implementation that uses DecreaseKey.
- The idea is that you'll create your own Heap data structure, where you have complete control.
- To decrease the distance estimate of a vertex, you can alter the actual node in the Heap and perform bubble-up to re-establish the Heap property.

DECREASE KEY IMPLEMENTATION

- You'll need to create your own Heap data structure. We'll be using an array as the internal container.
- You'll also have to keep track of the index of each vertex in that array.
- Because the vertices are labelled from 0 to *N* - 1, we'll use an int array.

```
class DijkHeap {
    private DijkNode[] heapArray;
    private int[] vertexIndex;

    // Constructor
    public DijkHeap(int numVertices) {
        heapArray = new DijkNode[numVertices + 1];
        vertexIndex = new int[numVertices];

        for (int i = 0; i < numVertices; ++i) {
            heapArray[i + 1] = new DijkNode(i, INF);
            vertexIndex[i] = i + 1;
        }
    }
```

DECREASE KEY IMPLEMENTATION

- To decrease the key (distance estimate):
  - Find the DijkNode in the Heap array
  - Change the distance estimate
  - Re-establish the Heap property using bubble-up.

```
public void decreaseKey(int vertex, int newDist) {
    int index = vertexIndex[vertex];
    DijkNode node = heapArray[index];

    if (newDist < node.getDistEstimate()) {
        node.setDistEstimate(newDist);
        bubbleUp(index);
    }
}
```

DECREASE KEY IMPLEMENTATION

- Ensure that your Heap methods also update the vertex indices as you move the DijkNodes around.

```
private void nodeSwap(int index1, int index2) {
    DijkNode node1 = heapArray[index1];
    DijkNode node2 = heapArray[index2];

    // Update indices
    vertexIndex[node1.getVertex()] = index2;
    vertexIndex[node2.getVertex()] = index1;

    heapArray[index1] = node2;
    heapArray[index2] = node1;
}
```

DECREASE KEY IMPLEMENTATION

- Ensure that your Heap methods also update the vertex indices as you move the DijkNodes around.
- For example, bubble up and bubble down.

```
private void bubbleUp(int index) {
    while (index > 1) {
        int parentIndex = index / 2;
        DijkNode node = heapArray[index];
        DijkNode parent = heapArray[parentIndex];

        if (node.compareTo(parent) < 0) {
            nodeSwap(index, parentIndex);
            index /= 2;
        } else {
            break;
        }
    }
}
```

DECREASE KEY IMPLEMENTATION

- After creating your modified Heap, use it in your main code.

```
DijkHeap dijkHeap = new dijkHeap(numVertices);
dijkHeap.decreaseKey(source, 0);

while (!dijkHeap.isEmpty()) {
    DijkNode node = dijkHeap.poll();
    int vertex = node.getVertex();
    int distEst = node.getDistEstimate();

    if (distEst == INF) {
        // No more reachable vertices
        break;
    }

    for (Edge edge : adjList.get(vertex)) {
        ...
```

DECREASE KEY IMPLEMENTATION

- This implementation is quite specific for vertices labelled from 0 to *N* - 1.
- For more general cases, you can use a HashMap to store the indices of the keys instead.

```
class MinHeap<K,V> {
    ArrayList<Pair<K,V>> heapArray;
    HashMap<K, Integer> keyIndex;

    ...
```