Graph Theory Video Series

Graph Theory Intro & Overview

William Fiset

Brief introduction

Graph theory is the mathematical theory of the properties and applications of graphs (networks).

The goal of this series is to gain an understanding of how to apply graph theory to real world applications.

Brief introduction

A graph theory problem might be:
Given the constraints above, how many different sets of clothing can I make by choosing an article from each category?

Brief introduction

The canonical graph theory example is
a social network of friends.

This enables interesting questions such as: how many friends does person X have? Or how many degrees of separation are there between person X and person Y?

Types of Graphs

Undirected Graph

An **undirected graph** is a graph in which edges have no orientation. The edge (u, v) is identical to the edge (v, u). – Wiki

Undirected Graph

An **undirected graph** is a graph in which edges have no orientation. The edge (*u*, *v*) is identical to the edge (*v*, *u*). – Wiki

In the graph above, the nodes could represent cities and an edge could represent a bidirectional road.

Directed Graph (Digraph)

A ***directed graph*** or ***digraph*** is a graph in which edges have orientations. For example, the edge (u, v) is the edge *from* node u *to* node v.

Directed Graph (Digraph)

A ***directed graph*** or ***digraph*** is a graph in which edges have orientations. For example, the edge (u, v) is the edge *from* node u *to* node v.

In the graph above, the nodes could represent people and an edge (u, v) could represent that person u bought person v a gift.

Weighted Graphs

Many graphs can have edges that contain a certain weight to represent an arbitrary value such as cost, distance, quantity, etc…

NOTE: I will usually denote an edge of such a graph as a triplet (u, v, w) and specify whether the graph is directed or undirected.

Special Graphs

Trees!

A **tree** is an **undirected connected graph with no cycles**. Equivalently, it is a connected graph with N nodes and N–1 edges.

Rooted Trees!

A **rooted tree** is a tree with **a designated root node** where every edge either points away from or towards the root node. When edges point away from the root the graph is called an **arborescence (out-tree)** and anti-arborescence (in-tree) otherwise.

In-tree

Out-tree

Out-tree

Directed Acyclic Graphs (DAGs)

DAGs are **directed graphs with no cycles**. These graphs play an important role in representing structures with dependencies. Several efficient algorithms exist to operates on DAGs.

Cool fact: All out-trees are DAGs but **not all DAGs are out-trees.**

Bipartite Graph

A **bipartite graph** is one whose *vertices* can be split into two independent groups U, V such that every edge connects betweens U and V.

Other definitions exist such as: The graph is two colourable or there is no odd length cycle.

Complete Graphs

A **complete graph** is one where there is a unique edge between every pair of nodes. A complete graph with n vertices is denoted as the graph K_n.

Representing Graphs

Adjacency Matrix

A **adjacency matrix** *m* is a very simple way to represent a graph. The idea is that the cell m[i][j] represents the edge weight of going from node i to node j.

$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & \mathbf{0} & \mathbf{4} & \mathbf{1} & \mathbf{9} \\
B & \mathbf{3} & \mathbf{0} & \mathbf{6} & \mathbf{11} \\
C & \mathbf{4} & \mathbf{1} & \mathbf{0} & \mathbf{2} \\
D & \mathbf{6} & \mathbf{5} & \mathbf{-4} & \mathbf{0}
\end{array}
$$

NOTE: It is often assumed that the edge of going from a node to itself has a cost of zero.

Adjacency Matrix

Pros	Cons
Space efficient for representing dense graphs	Requires $O(V^2)$ space
Edge weight lookup is $O(1)$	Iterating over all edges takes $O(V^2)$ time
Simplest graph representation	

Adjacency List

An **adjacency list** is a way to represent a graph as a map from nodes to lists of edges.

```
A -> [(B,4),(C,1)]
B -> [(C,6)]
C -> [(A,4),(B,1),(D,2)]
D -> []
```

Adjacency List

An **adjacency list** is a way to represent a graph as a map from nodes to lists of edges.

```
A -> [(B,4),(C,1)]
B -> [(C,6)]
C -> [(A,4),(B,1),(D,2)]
D -> []
```

Node C can reach
- Node A with cost 4
- Node B with cost 1
- Node D with cost 2

Adjacency List

Pros	Cons
Space efficient for representing sparse graphs	Less space efficient for denser graphs.
Iterating over all edges is efficient	Edge weight lookup is **O(E)**
	Slightly more complex graph representation

Edge List

An **edge list** is a way to represent a graph simply as an unordered list of edges. Assume the notation for any triplet (u,v,w) means: "the cost from node u to node v is w"

[(C,A,4), (A,C,1), (B,C,6), (A,B,4), (C,B,1), (C,D,2)]

This representation is seldomly used because of its lack of structure. However, it is conceptually simple and practical in a handful of algorithms.

Edge List

Pros	Cons
Space efficient for representing sparse graphs	Less space efficient for denser graphs.
Iterating over all edges is efficient	Edge weight lookup is **O(E)**
Very simple structure	

Next Video: Graph Theory Problems

Graph Theory Series

William Fiset

Graph Theory Video Series

Problems in Graph Theory

William Fiset

Common Graph Theory Problems

For the upcoming problems ask yourself:

Is the graph directed or undirected?

Are the edges of the graph weighted?

Is the graph I will encounter likely to be sparse or dense with edges?

Should I use an adjacency matrix, adjacency list, an edge list or other structure to represent the graph efficiently?

Shortest path problem

Given a weighted graph, find the shortest path of edges from node A to node B.

Algorithms: BFS (unweighted graph), Dijkstra's, Bellman-Ford, Floyd-Warshall, A*, and many more.

Shortest path problem

Given a weighted graph, find the shortest path of edges from node A to node B.

Algorithms: BFS (unweighted graph), Dijkstra's, Bellman-Ford, Floyd-Warshall, A*, and many more.

Connectivity

Does there exist a path between node A and node B?

Typical solution: Use union find data structure or any search algorithm (e.g DFS).

Connectivity

Does there exist a path between node A and node B?

Typical solution: Use union find data structure or any search algorithm (e.g DFS).

Negative cycles

Does my weighted digraph have any negative cycles? If so, where?

Algorithms: Bellman-Ford and Floyd-Warshall

Negative cycles

Does my weighted digraph have any negative cycles? If so, where?

Algorithms: Bellman-Ford and Floyd-Warshall

Strongly Connected Components

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.

Algorithms: Tarjan’s and Kosaraju's algorithm

Strongly Connected Components

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.

Algorithms: Tarjan’s and Kosaraju's algorithm

Traveling Salesman Problem

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” – Wiki

Algorithms: Held-Karp, branch and bound and many approximation algorithms

Traveling Salesman Problem

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” – Wiki

Algorithms: Held-Karp, branch and bound and many approximation algorithms

Traveling Salesman Problem

The TSP problem is NP-Hard meaning it’s a very computationally challenging problem. This is unfortunate because the TSP has several very important applications.

Algorithms: Held-Karp, branch and bound and many approximation algorithms

Bridges

A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.

Bridges

A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.

Bridges are important in graph theory because they often hint at weak points, bottlenecks or vulnerabilities in a graph.

Articulation points

An **articulation point / cut vertex** is any node in a graph whose removal increases the number of connected components.

Articulation points

An **articulation point / cut vertex** is any node in a graph whose removal increases the number of connected components.

Articulation points are important in graph theory because they often hint at weak points, bottlenecks or vulnerabilities in a graph.

Minimum Spanning Tree (MST)

A **minimum spanning tree (MST)** is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. – Wiki

Algorithms: Kruskal’s, Prim’s & Borůvka's algorithm

Minimum Spanning Tree (MST)

A **minimum spanning tree (MST)** is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. – Wiki

This MST has a total weight of 14. Note that MSTs on a graph are not always unique.

Algorithms: Kruskal’s, Prim’s & Borůvka's algorithm

Minimum Spanning Tree (MST)

MSTs are seen in many applications including: Designing a least cost network, circuit design, transportation networks, and etc…

This MST has a total weight of 14. Note that MSTs on a graph are not always unique.

Algorithms: Kruskal’s, Prim’s & Borůvka's algorithm

Network flow: max flow

Q: With an infinite input source how much “flow” can we push through the network?

Suppose the edges are roads with cars, pipes with water or hallways with packed with people. Flow represents the volume of water allowed to flow through the pipes, the number of cars the roads can sustain in traffic and the maximum amount of people that can navigate through the hallways.

Algorithms: Ford-Fulkerson, Edmonds-Karp & Dinic’s algorithm

Next Video: Depth First Search

Problems in Graph Theory

Graph Theory Video Series

Graph Theory: Depth First Search

William Fiset

DFS overview

The **Depth First Search (DFS)** is the most fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of **O(V+E)** and is often used as a building block in other algorithms.

By itself the DFS isn’t all that useful, but **when augmented** to perform other tasks such as count connected components, determine connectivity, or find bridges/articulation points then **DFS really shines**.

Basic DFS

As the name suggests, a DFS plunges depth first into a graph without regard for which edge it takes next until it cannot go any further at which point it backtracks and continues.

Basic DFS

Start DFS at node 0

Basic DFS

Pick an edge outwards from node 0

Basic DFS

Once at 9 pick an edge outwards from node 9

Basic DFS

Go to node 8

Basic DFS

Pick an edge outwards from 8…

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Make sure you don’t re-visit visited nodes!

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

We haven’t finished visiting all the neighbours of 7 so continue DFS in another direction

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Backtrack when a dead end is reached.

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

Basic DFS

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
     dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
     dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
     dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
     dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
     dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
     dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
     dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
     dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

Connected Components

Connected Components

Sometimes a graph is split into multiple components. It’s useful to be able to identify and count these components.

Connected Components

Sometimes a graph is split into multiple components. It’s useful to be able to identify and count these components.

Connected Components

Assign an integer value to each group to be able to tell them apart.

We can use a DFS to identify components. First, make sure all the nodes are labeled from [0, n) where n is the number of nodes.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it’s already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

Algorithm: Start a DFS at every node (except if it's already been visited) and mark all reachable nodes as being part of the same component.

… and so on for the other components

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

What else can DFS do?

We can augment the DFS algorithm to:

- Compute a graph’s minimum spanning tree.
- Detect and find cycles in a graph.
- Check if a graph is bipartite.
- Find strongly connected components.
- Topologically sort the nodes of a graph.
- Find bridges and articulation points.
- Find augmenting paths in a flow network.
- Generate mazes.

Next Video: Breadth First Search

Depth First Search Algorithm

Graph Theory Video Series

Graph Theory: Breadth First Search

William Fiset

BFS overview

The **Breadth First Search (BFS)** is another fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of **O(V+E)** and is often used as a building block in other algorithms.

The BFS algorithm is particularly useful for one thing: finding the **shortest path on unweighted graphs**.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

Using a Queue

The BFS algorithm uses a queue data structure to track which node to visit next. Upon reaching a new node the algorithm adds it to the queue to visit it later. The queue data structure works just like a real world queue such as a waiting line at a restaurant. People can either enter the waiting line (**enqueue**) or get seated (**dequeue**).

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and 0 ≤ e,s < n
function bfs(s, e):

  # Do a BFS starting at node s
  prev = solve(s)

  # Return reconstructed path from s -> e
  return reconstructPath(s, e, prev)
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and 0 ≤ e,s < n
function bfs(s, e):

  # Do a BFS starting at node s
  prev = solve(s)

  # Return reconstructed path from s -> e
  return reconstructPath(s, e, prev)
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and 0 ≤ e,s < n
function bfs(s, e):

  # Do a BFS starting at node s
  prev = solve(s)

  # Return reconstructed path from s -> e
  return reconstructPath(s, e, prev)
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and 0 ≤ e,s < n
function bfs(s, e):

  # Do a BFS starting at node s
  prev = solve(s)

  # Return reconstructed path from s -> e
  return reconstructPath(s, e, prev)
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
  return prev
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
  return prev
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
  return prev
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
return prev
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
  return prev
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
return prev
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
return prev
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
return prev
```

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
return prev
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and 0 ≤ e,s < n
function bfs(s, e):

  # Do a BFS starting at node s
  prev = solve(s)

  # Return reconstructed path from s -> e
  return reconstructPath(s, e, prev)
```

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and 0 ≤ e,s < n
function bfs(s, e):

  # Do a BFS starting at node s
  prev = solve(s)

  # Return reconstructed path from s -> e
  return reconstructPath(s, e, prev)
```

```
function reconstructPath(s, e, prev):

  # Reconstruct path going backwards from e
  path = []
  for(at = e; at != null; at = prev[at]):
    path.add(at)

  path.reverse()

  # If s and e are connected return the path
  if path[0] == s:
    return path
  return []
```

```
function reconstructPath(s, e, prev):

  # Reconstruct path going backwards from e
  path = []
  for(at = e; at != null; at = prev[at]):
    path.add(at)

  path.reverse()

  # If s and e are connected return the path
  if path[0] == s:
    return path
  return []
```

```
function reconstructPath(s, e, prev):

  # Reconstruct path going backwards from e
  path = []
  for(at = e; at != null; at = prev[at]):
    path.add(at)

  path.reverse()

  # If s and e are connected return the path
  if path[0] == s:
    return path
  return []
```

```
function reconstructPath(s, e, prev):

  # Reconstruct path going backwards from e
  path = []
  for(at = e; at != null; at = prev[at]):
    path.add(at)

  path.reverse()

  # If s and e are connected return the path
  if path[0] == s:
    return path
  return []
```

Breadth First Search Algorithm

Graph Theory Video Series

BFS Shortest Path on a Grid

William Fiset

BFS Video

Please watch **Breadth First Search Algorithm** to understand the basics of a BFS before proceeding.

Link should be in the description below.

Motivation

Many problems in graph theory can be represented using a grid. Grids are a form of **implicit graph** because we can determine a node’s neighbours based on our location within the grid.

Motivation

Many problems in graph theory can be represented using a grid. Grids are a form of **implicit graph** because we can determine a node's neighbours based on our location within the grid.

A type of problem that involves finding a path through a grid is solving a maze:

Motivation

Many problems in graph theory can be represented using a grid. Grids are a form of **implicit graph** because we can determine a node's neighbours based on our location within the grid.

Another example could be routing through obstacles (trees, rivers, rocks, etc) to get to a location:

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

0	**1**
2	**3**
4	**5**

First label the cells in the grid with numbers [0, n) where n = #rows x #columns

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

0	**1**
2	**3**
4	**5**

Adjacency List:

Adjacency Matrix:

	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

Adjacency List:

0 -> [1, 2]

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1						
2						
3						
4						
5						

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

Adjacency List:

```
0 -> [1, 2]
1 -> [0, 3]
```

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2						
3						
4						
5						

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

Adjacency List:

```
0 -> [1, 2]
1 -> [0, 3]
2 -> [0, 3, 4]
```

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3						
4						
5						

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

Adjacency List:

```
0 -> [1, 2]
1 -> [0, 3]
2 -> [0, 3, 4]
3 -> [1, 2, 5]
```

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4						
5						

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

Adjacency List:

```
0 -> [1, 2]
1 -> [0, 3]
2 -> [0, 3, 4]
3 -> [1, 2, 5]
4 -> [2, 5]
```

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5						

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

Adjacency List:

```
0 -> [1, 2]
1 -> [0, 3]
2 -> [0, 3, 4]
3 -> [1, 2, 5]
4 -> [2, 5]
5 -> [3, 4]
```

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an **adjacency list/matrix**.

Empty Grid

0	1
2	3
4	5

Adjacency List:

```
0 -> [1, 2]
1 -> [0, 3]
2 -> [0, 3, 4]
3 -> [1, 2, 5]
4 -> [2, 5]
5 -> [3, 4]
```

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

Once we have an adjacency list/matrix we can run whatever specialized graph algorithm to solve our problem such as: shortest path, connected components, etc…

However, **transformations between graph representations can usually be avoided** due to the structure of a grid.

Direction Vectors

Due to the structure of a grid, if we are at the **red ball** in the middle we know we can move left, right, up and down to reach adjacent cells:

Direction Vectors

Mathematically, if the **red ball** is at the row-column coordinate (r, c) we can add the row vectors **[-1, 0]**, **[1, 0]**, **[0, 1]**, and **[0, -1]** to reach adjacent cells.

Direction Vectors

If the problem you are trying to solve allows moving diagonally then you can also include the row vectors: **[-1, -1]**, **[-1, 1]**, **[1, 1]**, **[1, -1]**

Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for
# north, south, east and west.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]

for(i = 0; i < 4; i++):
  rr = r + dr[i]
  cc = c + dc[i]
  # Skip invalid cells. Assume R and
  # C for the number of rows and columns
  if rr < 0 or cc < 0: continue
  if rr >= R or cc >= C: continue
  #(rr, cc) is a neighbouring cell of (r, c)
```

Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for
# north, south, east and west.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
for(i = 0; i < 4; i++):
  rr = r + dr[i]
  cc = c + dc[i]
  # Skip invalid cells. Assume R and
  # C for the number of rows and columns
  if rr < 0 or cc < 0: continue
  if rr >= R or cc >= C: continue
  #(rr, cc) is a neighbouring cell of (r, c)
```

Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for
# north, south, east and west.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]

for(i = 0; i < 4; i++):
   rr = r + dr[i]
   cc = c + dc[i]
   # Skip invalid cells. Assume R and
   # C for the number of rows and columns
   if rr < 0 or cc < 0: continue
   if rr >= R or cc >= C: continue
   #(rr, cc) is a neighbouring cell of (r, c)
```

Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for
# north, south, east and west.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]

for(i = 0; i < 4; i++):
  rr = r + dr[i]
  cc = c + dc[i]
  # Skip invalid cells. Assume R and
  # C for the number of rows and columns
  if rr < 0 or cc < 0: continue
  if rr >= R or cc >= C: continue
  #(rr, cc) is a neighbouring cell of (r, c)
```

Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for
# north, south, east and west.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]

for(i = 0; i < 4; i++):
  rr = r + dr[i]
  cc = c + dc[i]
  # Skip invalid cells. Assume R and
  # C for the number of rows and columns
  if rr < 0 or cc < 0: continue
  if rr >= R or cc >= C: continue
  #(rr, cc) is a neighbouring cell of (r, c)
```

Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for
# north, south, east and west.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]

for(i = 0; i < 4; i++):
  rr = r + dr[i]
  cc = c + dc[i]
  # Skip invalid cells. Assume R and
  # C for the number of rows and columns
  if rr < 0 or cc < 0: continue
  if rr >= R or cc >= C: continue
  #(rr, cc) is a neighbouring cell of (r, c)
```

Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for
# north, south, east and west.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]

for(i = 0; i < 4; i++):
  rr = r + dr[i]
  cc = c + dc[i]
  # Skip invalid cells. Assume R and
  # C for the number of rows and columns
  if rr < 0 or cc < 0: continue
  if rr >= R or cc >= C: continue
  #(rr, cc) is a neighbouring cell of (r, c)
```

Dungeon Problem Statement

You are trapped in a 2D dungeon and need to find the quickest way out! The dungeon is composed of unit cubes which may or may not be filled with rock. It takes one minute to move one unit north, south, east, west. You cannot move diagonally and the maze is surrounded by solid rock on all sides.

Is an escape possible?
If yes, how long will it take?

This is an easier version of the “Dungeon Master” problem on Kattis: open.kattis.com/problems/dungeon. Link in description.

Dungeon Problem Statement

The dungeon has a size of **R** x **C** and you start at cell ’S’ and there’s an exit at cell ‘E’. A cell full of rock is indicated by a ’#’ and empty cells are represented by a ’.’

C

R

S	.	.	#	.	.	.
.	#	.	.	.	#	.
.	#	.	.	.	.	.
.	.	#	#	.	.	.
#	.	#	E	.	#	.

Dungeon Problem Statement

The dungeon has a size of **R** x **C** and you start at cell ’S’ and there’s an exit at cell ‘E’. A cell full of rock is indicated by a ’#’ and empty cells are represented by a ’.’

	0	1	2	3	4	5	6
0	S	.	.	#	.	.	.
1	.	#	.	.	.	#	.
2	.	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

Start at the start node coordinate by adding (sr, sc) to the queue.

	0	1	2	3	4	5	6
0	(0,0)	.	.	#	.	.	.
1	.	#	.	.	.	#	.
2	.	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	.	#	.	.	.
1	(1,0)	#	.	.	.	#	.
2	.	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

(1, 0)

(0, 1)

(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	.	.	.	#	.
2	(2,0)	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	(1,2)	.	.	#	.
2	(2,0)	#	.	.	.	.	.
3	(3,0)	.	#	#	.	.	.
4	#	.	#	E	.	#	.

(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	(1,2)	(1,3)	.	#	.
2	(2,0)	#	(2,2)	.	.	.	.
3	(3,0)	(3,1)	#	#	.	.	.
4	#	.	#	E	.	#	.

(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	.	.	.
3	(3,0)	(3,1)	#	#	.	.	.
4	#	(4,1)	#	E	.	#	.

(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	.	.
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	(2,4)	.	.
3	(3,0)	(3,1)	#	#	.	.	.
4	#	(4,1)	#	E	.	#	.

(0, 5)
(3, 4)
2, 5)
(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	.
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	.
3	(3,0)	(3,1)	#	#	(3,4)	.	.
4	#	(4,1)	#	E	.	#	.

(0, 5)
(3, 4)
2, 5)
(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

(0, 6)
(4, 4)
(3, 5)
(2, 6)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	(0,6)
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	#	#	(3,4)	(3,5)	.
4	#	(4,1)	#	E	(4,4)	#	.

(0, 5)
(3, 4)
2, 5)
(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

(4, 3)
(1, 6)
(3,6)
(0, 6)
(4, 4)
(3, 5)
(2, 6)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	(0,6)
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	(1,6)
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	#	#	(3,4)	(3,5)	(3,6)
4	#	(4,1)	#	(4,3)	(4,4)	#	.

We have reached the end, and if we had a 2D prev matrix we could regenerate the path by retracing our steps.

(0, 5)
(3, 4)
2, 5)
(0, 4)
(2, 4)
(4, 1)
(2, 3)
(1, 4)
(3, 1)
(2, 2)
(1, 3)
(3, 0)
(1, 2)
(2, 0)
(0, 2)
(1, 0)
(0, 1)
(0, 0)

(4, 3)
(1, 6)
(3,6)
(0, 6)
(4, 4)
(3, 5)
(2, 6)

	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	(0,6)
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	(1,6)
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	#	#	(3,4)	(3,5)	(3,6)
4	#	(4,1)	#	(4,3)	(4,4)	#	.

Alternative State representation

So far we have been storing the next x–y position in the queue as an (x, y) pair. This works well but requires either an array or an object wrapper to store the coordinate values. In practice, this requires a lot of packing and unpacking of values to and from the queue.

Let’s take a look at an alternative approach which also scales well in higher dimensions and (IMHO) requires less setup effort…

Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

Alternative State representation

An alternative approach is to **use one queue for each dimension**, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
if reached_end:
  return move_count
return -1
```

Summary

Representing a grid as an adjacency list and adjacency matrix.

Empty Grid

0	1
2	3
4	5

Adjacency List:

```
0 -> [1, 2]
1 -> [0, 3]
2 -> [0, 3, 4]
3 -> [1, 2, 5]
4 -> [2, 5]
5 -> [3, 4]
```

Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

Summary

Using direction vectors to visit neighbouring cells.

Summary

Explored an alternative way to represent multi dimensional coordinates using multiple queues.

Summary

How to use BFS on a grid to find the shortest path between two cells.

Breadth First Search
Shortest Path on a Grid

S	.	.	#	.	.	.
.	#	.	.	.	#	.
.	#	.	.	.	.	.
.	.	#	#	.	.	.
#	.	#	E	.	#	.

Graph Theory Video Series

Topological Sort

William Fiset

Many real world situations can be modelled as a graph with directed edges where some events must occur before others.

- School class prerequisites
- Program dependencies
- Event scheduling
- Assembly instructions
- Etc…

Suppose you're a student at university X and you want to take Class H, then you must take classes A, B, D and E as prerequisites.
In this sense there is an **ordering** on the nodes of the graph.

Suppose you're a student at university X and you want to take Class H, then you must take classes A, B, D and E as prerequisites. In this sense there is an **ordering** on the nodes of the graph.

Suppose you're a student at university X and you want to take Class H, then you must take classes A, B, D and E as prerequisites. In this sense there is an **ordering** on the nodes of the graph.

Suppose you're a student at university X and you want to take Class H, then you must take classes A, B, D and E as prerequisites. In this sense there is an **ordering** on the nodes of the graph.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

A **topological ordering** is an ordering of the nodes in a directed graph where for each directed edge from node A to node B, node A appears before node B in the ordering.

The **topological sort** algorithm can find a topological ordering in **O(V+E)** time!

NOTE: Topological orderings are NOT unique.

Directed Acyclic Graphs(DAG)

Not every graph can have a topological ordering. A graph which contains a **cycle** cannot have a valid ordering:

Directed Acyclic Graphs(DAG)

Not every graph can have a topological ordering. A graph which contains a **cycle** cannot have a valid ordering:

Directed Acyclic Graphs(DAG)

The only type of graph which has a valid topological ordering is a **Directed Acyclic Graph (DAG)**. These are graphs with directed edges and no cycles.

Directed Acyclic Graphs(DAG)

The only type of graph which has a valid topological ordering is a **Directed Acyclic Graph (DAG)**. These are graphs with directed edges and no cycles.

Q: How do I verify that my graph does not contain a directed cycle?

A: One method is to use Tarjan’s strongly connected component algorithm which can be used to find these cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.

Topological ordering from left to right:

Topological Sort Algorithm

Pick an unvisited node

Beginning with the selected node, do a Depth First Search (DFS) exploring only unvisited nodes.

On the recursive callback of the DFS, add the current node to the topological ordering in reverse order.

Topological Sort Algorithm

DFS recursion
call stack:

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion
call stack:

Node H
Node J
Node M

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J
Node L

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ M

Topological Sort Algorithm

DFS recursion
call stack:

Node H
Node J

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ L M

Topological Sort Algorithm

DFS recursion call stack:

Node H

Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H

Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node I

Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node I

Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node I

Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node H

Topological ordering:

_ _ _ _ _ _ _ _ _ _ I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node E

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node E

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node E
Node A

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node E
Node A

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D
Node G

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D
Node G

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D
Node G

Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D

Topological ordering:

_ _ _ _ _ _ _ G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A

Topological ordering:

_ _ _ _ _ _ D G H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node E

Topological ordering:

_ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node E

Topological ordering:

_ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node E

Topological ordering:

_ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F

Topological ordering:

_ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F

Topological ordering:

_ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F
Node K

Topological ordering:

_ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F
Node K

Topological ordering:

_ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F
Node K

Topological ordering:

_ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F

Topological ordering:

_ _ _ _ K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F

Topological ordering:

_ _ _ _ K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F

Topological ordering:

_ _ _ _ K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E

Topological ordering:

_ _ _ F K A D G H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C

Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C

Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C
Node B

Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C
Node B

Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C
Node B

Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node C

Topological ordering:

_ B E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node C

Topological ordering:

_ B E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Node C

Topological ordering:

_ B E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion
call stack:

Topological ordering:

C B E F K A D G H I J L M

Topsort pseudocode

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      visitedNodes = []
      dfs(at, V, visitedNodes, graph)
      for nodeId in visitedNodes:
        ordering[i] = nodeId
        i = i - 1
  return ordering
```

Topsort pseudocode

```
# Execute Depth First Search (DFS)
function dfs(at, V, visitedNodes, graph):

  V[at] = true

  edges = graph.getEdgesOutFromNode(at)
  for edge in edges:
    if V[edge.to] == false:
      dfs(edge.to, V, visitedNodes, graph)

  visitedNodes.add(at)
```

Topsort pseudocode

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      visitedNodes = []
      dfs(at, V, visitedNodes, graph)
      for nodeId in visitedNodes:
        ordering[i] = nodeId
        i = i - 1
  return ordering
```

Topsort pseudocode

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      visitedNodes = []
      dfs(at, V, visitedNodes, graph)
      for nodeId in visitedNodes:
        ordering[i] = nodeId
        i = i - 1
  return ordering
```

Topsort Optimization

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      i = dfs(i, at, V, ordering, graph)

  return ordering
```

Topsort Optimization

```
# Execute Depth First Search (DFS)
function dfs(i, at, V, ordering, graph):

  V[at] = true

  edges = graph.getEdgesOutFromNode(at)
  for edge in edges:
    if V[edge.to] == false:
      i = dfs(i, edge.to, V, ordering, graph)

  ordering[i] = at
  return i - 1
```

Source Code Link

Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description:

Topological Sort Algorithm

Graph Theory Video Series

Kahn's Algorithm

An intuitive topological sort algorithm

William Fiset

How to get dressed

One (of many) possible Topological Orderings:

How to get dressed

One (of many) possible Topological Orderings:

How to get dressed

One (of many) possible Topological Orderings:

How to get dressed

One (of many) possible Topological Orderings:

How to get dressed

One (of many) possible Topological Orderings:

How to get dressed

One (of many) possible Topological Orderings:

How to get dressed

One (of many) possible Topological Orderings:

How to get dressed

One (of many) possible Topological Orderings:

Motivation

Many real world situations can be modeled as a graph with directed edges where some events must occur before others, for instance:

- Program build dependencies
- College class prerequisites
- Event scheduling
- Assembly instructions
- Etc…

A **topological ordering** is an ordering of the nodes in a directed graph where for each directed edge from node A to node B, node A appears before node B in the ordering.

Kahn's algorithm is a simple **topological sort** algorithm can find a topological ordering in **O(V+E)** time!

NOTE: Topological orderings are **NOT** unique.

Which graphs have topological sorts?

Only certain types of graphs have a topological orderings. These are **Directed Acyclic Graphs (DAGs)**.

"A directed acyclic graph is a finite **directed graph** with **no directed cycles**."
– Wiki

Which of these are Directed Acyclic Graphs (DAGs)?

(1)

(2)

(3)

(4)

(5)

(6)

Which of these are Directed Acyclic Graphs (DAGs)?

(1)

(2)

(3)

(4)

(5)

(6)

Which graphs have topological sorts?

Try and imagine what happens if you try and find the topological ordering of a graph that contains a cycle:

Which graphs have topological sorts?

First you would pick node 4, the only node without any dependencies

4,

Which graphs have topological sorts?

Then, you could process node 1, which no longer depends on node 4

4, 1

Which graphs have topological sorts?

Followed by node 0, which no longer depends on node 1

4, 1, 0

Which graphs have topological sorts?

Then you get stuck in a cycle. Every node depends on another node 😱

4, 1, 0, ???

Kahn’s Algorithm Intuition

The intuition behind Kahn’s algorithm is to repeatedly remove nodes without any dependencies from the graph and add them to the topological ordering.

As nodes without dependencies (and their outgoing edges) are removed from the graph, new nodes without dependencies should become free.

We repeat removing nodes without dependencies from the graph until all nodes are processed, or a cycle is discovered.

Topological Ordering:

_ _ _ _ _ _

Node 2 is the only node without dependencies

Topological Ordering:

_ _ _ _ _ _

Add node 2 to the topological ordering, and remove it from the graph.

Topological Ordering:

2 _ _ _ _ _

Node 0 and node 4 don’t have any dependencies. Select either to be added to the topological ordering.

Topological Ordering:

2 _ _ _ _ _

Add node 0 to the topological ordering, and remove it from the graph.

Topological Ordering:

2 0 _ _ _ _

Node 4 is the only node without dependencies

Topological Ordering:

2 0 _ _ _ _

Add node 4 to the topological ordering, and remove it from the graph.

Topological Ordering:

2 0 4 _ _ _

Node 3 and node 5 don’t have any dependencies. Select either to be added to the topological ordering.

Topological Ordering:

2 0 4 _ _ _

Add node 3 to the topological ordering, and remove it from the graph.

Topological Ordering:
2 0 4 3 _ _

Node 5 is the only node without dependencies

Topological Ordering:
2 0 4 3 _ _

Add node 5 to the topological ordering, and remove it from the graph.

Topological Ordering:
2 0 4 3 5 _

Finally, only node 1 remains

Topological Ordering:
2 0 4 3 5 _

Topological Ordering:
2 0 4 3 5 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13

Topological Ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _ _

Begin by counting the incoming degree of each node

0	1	2	3	4	5	6	7	8	9	10	11	12	13

<u>Topological Ordering:</u>

_ _ _ _ _ _ _ _ _ _ _ _ _ _

For example, node 6 has an incoming degree of 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13
						3							

<u>Topological Ordering:</u>

_ _ _ _ _ _ _ _ _ _ _ _ _ _

And node 7 has an incoming degree of 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13
						3	1						

<u>Topological Ordering:</u>

_ _ _ _ _ _ _ _ _ _ _ _ _ _

Nodes by incoming degree count:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	2	1	3	1	3	1	2	0	1	1	2	0

<u>Topological Ordering:</u>

_ _ _ _ _ _ _ _ _ _ _ _ _ _

Queue

Maintain a queue of all nodes with no incoming edges

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	2	1	3	1	3	1	2	0	1	1	2	0

Topological Ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _ _

Add all nodes with a degree of 0 (no incoming edges) to the queue. These are all the nodes in the graph with no dependencies.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	2	1	3	1	3	1	2	0	1	1	2	0

Topological Ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _ _

Remove node 0 from the front of the queue, and add it to the topological ordering.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	2	1	3	1	3	1	2	0	1	1	2	0

Remove node 0 from the graph and decrease the degree of all affected nodes.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	1	0	3	1	2	1	2	0	1	1	2	0

Topological Ordering:

0 _ _ _ _ _ _ _ _ _ _ _ _ _

Queue

node 3

node 13

node 9

Add any new nodes with an incoming degree of 0 to the queue.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	1	0	3	1	2	1	2	0	1	1	2	0

Topological Ordering:

0 _ _ _ _ _ _ _ _ _ _ _ _ _

Remove node 9 from the front of the queue, and add it to the topological ordering.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	1	0	3	1	2	1	2	0	1	1	2	0

Topological Ordering:

0 9 _ _ _ _ _ _ _ _ _ _ _ _

Queue

node 3

node 13

Remove node 9 from the graph and decrease the degree of all affected nodes.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	0	3	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 _ _ _ _ _ _ _ _ _ _ _ _ _

Add any new nodes with an incoming degree of 0 to the queue.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	0	3	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 _ _ _ _ _ _ _ _ _ _ _ _ _

Remove node 13 from the front of the queue, and add it to the topological ordering.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	0	3	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 _ _ _ _ _ _ _ _ _ _ _

Remove node 13 from the graph and decrease the degree of all affected nodes. Node 13 has no outgoing edges, so no node degrees are updated (and hence no new nodes are added to the queue either).

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	0	3	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 _ _ _ _ _ _ _ _ _ _ _

Remove node 3 from the front of the queue, and add it to the topological ordering.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	0	3	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 _ _ _ _ _ _ _ _ _ _

Remove node 3 from the graph and decrease the degree of all affected nodes.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	2	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 _ _ _ _ _ _ _ _ _ _

Add any new nodes with an incoming degree of 0 to the queue.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	2	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 _ _ _ _ _ _ _ _ _ _

Let the animation play…

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	2	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 _ _ _ _ _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	2	1	2	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 2 _ _ _ _ _ _ _ _ _

Queue
node 1
node 10
13
0
1
2
3
4
5
6
7
8
9
10
11
12
0 1 2 3 4 5 6 7 8 9 10 11 12 13
0 0 0 0 2 1 1 1 2 0 0 1 2 0
Topological Ordering:
0 9 13 3 2 _ _ _ _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	2	1	1	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 2 10 _ _ _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	2	1	0	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 2 10 _ _ _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	2	1	0	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 2 10 _ _ _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	2	1	0	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 2 10 1 _ _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	1	1	0	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 2 10 1 _ _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	1	1	0	1	2	0	0	1	2	0

Topological Ordering:

0 9 13 3 2 10 1 6 _ _ _ _ _ _

Queue

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	1	1	0	0	2	0	0	0	2	0

Topological Ordering:

0 9 13 3 2 10 1 6 _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	1	1	0	0	2	0	0	0	2	0

Topological Ordering:

0 9 13 3 2 10 1 6 _ _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	1	1	0	0	2	0	0	0	2	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 _ _ _ _ _

Queue

node 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	1	0	0	2	0	0	0	1	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 _ _ _ _ _

Queue

node 4

node 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	1	0	0	2	0	0	0	1	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 _ _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	1	0	0	2	0	0	0	1	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 _ _ _ _

Queue

node 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	1	0	0	2	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	1	0	0	2	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 _ _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	1	0	0	2	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 _ _ _

Queue

node 12

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	1	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	1	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 _ _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	1	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 _ _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 _

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Queue

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Queue

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

Topological Ordering:

0 9 13 3 2 10 1 6 7 11 4 12 5 8

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

```
# `g` is a directed acyclic graph represented as an adjacency list.
function FindTopologicalOrdering(g):
  n = g.size()
  in_degree = [0,0,…,0,0] # size n
  for (i = 0; i < n; i++):
    for (to in g[i]):
      in_degree[to] = in_degree[to] + 1

  # `q` always contains the set nodes with no incoming edges.
  q = … # empty integer queue data structure
  for (i = 0; i < n; i++):
    if (in_degree[i] == 0):
      q.enqueue(i)

  index = 0
  order = [0,0,…0,0] # size n
  while (!q.isEmpty()):
    at = q.dequeue()
    order[index++] = at
    for (to in g[at]):
      in_degree[to] = in_degree[to] - 1
      if in_degree[to] == 0:
        q.enqueue(to)
  if index != n:
    return null # Oops, graph contains a cycle
  return order
```

Source Code Link

Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description

Topological Sort
Kahn's Algorithm

Graph Theory Video Series

Shortest and longest paths on DAGs

William Fiset

Directed Acyclic Graph (DAG)

Recall that a **Directed Acyclic Graph (DAG)** is a graph with directed edges and no cycles. By definition this means all **trees** are automatically DAGs since they do not contain cycles.

Directed Acyclic Graph (DAG)

Recall that a **Directed Acyclic Graph (DAG)** is a graph with directed edges and no cycles. By definition this means all **trees** are automatically DAGs since they do not contain cycles.

Q: Is this graph a DAG?

Directed Acyclic Graph (DAG)

Recall that a **Directed Acyclic Graph (DAG)** is a graph with directed edges and no cycles. By definition this means all **trees** are automatically DAGs since they do not contain cycles.

Q: Is this graph a DAG?

A: Yes!

Directed Acyclic Graph (DAG)

Recall that a **Directed Acyclic Graph (DAG)** is a graph with directed edges and no cycles. By definition this means all **trees** are automatically DAGs since they do not contain cycles.

Q: Is this graph a DAG?

Directed Acyclic Graph (DAG)

Recall that a **Directed Acyclic Graph (DAG)** is a graph with directed edges and no cycles. By definition this means all **trees** are automatically DAGs since they do not contain cycles.

Q: Is this graph a DAG?

A: No, the structure may be a tree, but it does not have directed edges.

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

∞	∞	∞	∞	∞	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	∞	∞	∞	∞	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	∞	∞	∞	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	∞	∞	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	∞	∞	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	∞	14	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	14	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	14	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	14	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	14	∞	∞	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	14	∞	17	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	14	∞	17	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	14	12	17	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	14	12	9	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	∞
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	11
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	11
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	11
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	11
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	11
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	11
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	11
A	B	C	D	E	F	G	H

SSSP on DAG

The **Single Source Shortest Path (SSSP)** problem can be solved efficiently on a DAG in **O(V+E)** time. This is due to the fact that the nodes can be ordered in a **topological ordering** via topsort and processed sequentially.

Arbitrary topological order: A, B, C, D, G, E, F, H

0	3	6	7	3	12	9	11
A	B	C	D	E	F	G	H

Longest path on DAG

What about the longest path? On a general graph this problem is **NP-Hard**, but on a DAG this problem is solvable in **O(V+E)**!

Longest path on DAG

What about the longest path? On a general graph this problem is **NP-Hard**, but on a DAG this problem is solvable in **O(V+E)**!

The trick is to multiply all edge values by -1 then find the shortest path and then multiply the edge values by -1 again!

Longest path on DAG

What about the longest path? On a general graph this problem is **NP-Hard**, but on a DAG this problem is solvable in **O(V+E)**!

The trick is to multiply all edge values by -1 then find the shortest path and then multiply the edge values by -1 again!

Longest path on DAG

What about the longest path? On a general graph this problem is **NP-Hard**, but on a DAG this problem is solvable in **O(V+E)**!

The trick is to multiply all edge values by −1 then find the shortest path and then multiply the edge values by −1 again!

Longest path on DAG

What about the longest path? On a general graph this problem is **NP-Hard**, but on a DAG this problem is solvable in **O(V+E)**!

The trick is to multiply all edge values by -1 then find the shortest path and then multiply the edge values by -1 again!

$(-3 + -11 + -9) * -1 = 23$

Source Code Link

Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description:

Shortest & Longest path on a DAG

Graph Theory Video Series

Dijkstra’s Shortest Path Algorithm

William Fiset

What is Dijkstra’s algorithm?

Dijkstra’s algorithm is a Single Source Shortest Path (SSSP) algorithm for graphs with **non-negative edge weights**.

Depending on how the algorithm is implemented and what data structures are used the time complexity is typically **O(E*log(V))** which is competitive against other shortest path algorithms.

Algorithm prerequisites

One constraint for Dijkstra’s algorithm is that the graph must only contain **non-negative edge weights**. This constraint is imposed to ensure that once a node has been visited its optimal distance cannot be improved.

This is property is especially important because it enables Dijkstra’s algorithm to act in a greedy manner by always selecting the next most promising node.

Outline

The goal of this slide deck is for you to understand how to implement Dijkstra's algorithm and implement it efficiently.

- Lazy Dijkstra's animation
- Lazy Dijkstra's pseudo-code
- Finding SP + stopping early optimization
- Using indexed priority queue + decreaseKey to reduce space and increase performance.
- Eager Dijkstra's animation
- Eager Dijkstra's pseudo-code
- Heap optimization with D-ary heap

Quick Algorithm Overview

Maintain a ‘dist’ array where the distance to every node is positive infinity. Mark the distance to the start node ’s' to be 0.

Maintain a PQ of key-value pairs of (node index, distance) pairs which tell you which node to visit next based on sorted min value.

Insert (s, 0) into the PQ and loop while PQ is not empty pulling out the next most promising (node index, distance) pair.

Iterate over all edges outwards from the current node and relax each edge appending a new (node index, distance) key-value pair to the PQ for every relaxation.

Lazy Dijkstra's

Lazy Dijkstra's

dist

0	1	2	3	4
∞	∞	∞	∞	∞

Lazy Dijkstra's

dist

0	1	2	3	4
∞	∞	∞	∞	∞

(index, dist)
key-value pairs

Lazy Dijkstra's

(index, dist)
key-value pairs

(0, 0)

dist

0	1	2	3	4
0	∞	∞	∞	∞

Lazy Dijkstra's

(index, dist)
key-value pairs

→ (0, 0)

dist

0	1	2	3	4
0	∞	∞	∞	∞

Lazy Dijkstra's

(index, dist)
key-value pairs

→ (0, 0)
(1, 4)

dist

0	1	2	3	4
0	4	∞	∞	∞

Best distance from node 0 to node 1 is:
dist[0] + edge.cost = 0 + 4 = 4

Lazy Dijkstra's

(index, dist)
key-value pairs

→ (0, 0)
(1, 4)
(2, 1)

dist

0	1	2	3	4
0	4	1	∞	∞

Best distance from node 0 to node 2 is:
dist[0] + edge.cost = 0 + 1 = 1

Lazy Dijkstra's

(index, dist)
key-value pairs

→ ~~(0, 0)~~
(1, 4)
(2, 1)

dist

0	1	2	3	4
0	4	1	∞	∞

Lazy Dijkstra's

dist

0	1	2	3	4
0	4	1	∞	∞

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
→ (2, 1)

Lazy Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
→ (2, 1)
(1, 3)

dist

0	1	2	3	4
0	3	1	∞	∞

Best distance from node 2 to node 1 is:
dist[2] + edge.cost = 1 + 2 = 3

Lazy Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
→ (2, 1)
(1, 3)
(3, 6)

dist

0	1	2	3	4
0	3	1	6	∞

Best distance from node 2 to node 3 is:
dist[2] + edge.cost = 1 + 5 = 6

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	6	∞

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
→ ~~(2, 1)~~
(1, 3)
(3, 6)

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	6	∞

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
~~(2, 1)~~
→ (1, 3)
(3, 6)

Lazy Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
~~(2, 1)~~
→ (1, 3)
(3, 6)
(3, 4)

dist

0	1	2	3	4
0	3	1	4	∞

Best distance from node 1 to node 3 is:
dist[1] + edge.cost = 3 + 1 = 4

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	∞

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
~~(2, 1)~~
→ ~~(1, 3)~~
(3, 6)
(3, 4)

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	∞

(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
(3, 4)

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	∞

(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
(3, 4)

We have already found a better route to get to node 1 (since dist[1] has value 3) so we can ignore this entry in the PQ.

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	∞

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
→ (3, 4)

Lazy Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
→ (3, 4)
(4, 7)

dist

0	1	2	3	4
0	3	1	4	7

Best distance from node 3 to node 4 is:
dist[3] + edge.cost = 4 + 3 = 7

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	7

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
(3, 6)
→ ~~(3, 4)~~
(4, 7)

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	7

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
→ (3, 6)
~~(3, 4)~~
(4, 7)

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	7

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
→ (3, 6)
~~(3, 4)~~
(4, 7)

We have already found a better route to get to node 3 (since dist[3] has value 4) so we can ignore this entry in the PQ.

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	7

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
~~(3, 6)~~
~~(3, 4)~~
→ (4, 7)

Lazy Dijkstra's

dist

0	1	2	3	4
0	3	1	4	7

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(1, 3)~~
~~(3, 6)~~
~~(3, 4)~~
~~(4, 7)~~

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

Assume PQ stores (node index, best distance) pairs sorted by minimum distance.

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

In practice most standard libraries do not support the decrease key operation for PQs. A way to get around this is to add a new (node index, best distance) pair every time we update the distance to a node.

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

As a result, it is possible to have duplicate node indices in the PQ. This is not ideal, but inserting a new key-value pair in **O(log(n))** is much faster than searching for the key in the PQ which takes **O(n)**

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true

    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return dist
```

A neat optimization we can do which ignores stale (index, dist) pairs in our PQ is to skip nodes where we already found a better path routing through others nodes before we got to processing this node.

Finding the optimal path

If you wish to not only find the optimal distance to a particular node but also **what sequence of nodes were taken** to get there you need to track some additional information.

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s and
# the prev array to reconstruct the shortest path itself
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  prev = [null, null, …, null] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        prev[edge.to] = index
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
  return (dist, prev)
```

```
# Finds the shortest path between two nodes.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
# e - the index of the end node (0 ≤ e < n)
function findShortestPath(g, n, s, e):
  dist, prev = dijkstra(g, n, s)
  path = []
  if (dist[e] == ∞) return path
  for (at = e; at != null; at = prev[at])
    path.add(at)
  path.reverse()
  return path
```

Stopping Early

Q: Suppose you know the destination node you're trying to reach is 'e' and you start at node 's' do you still have to visit every node in the graph?

Stopping Early

A: Yes, in the worst case. However, it is possible to stop early once you have finished visiting the destination node.

Stopping Early

The main idea for stopping early is that Dijkstra's algorithm processes each next most promising node in order. So if the destination node has been visited, its shortest distance will not change as more future nodes are visited.

```
# Runs Dijkstra's algorithm and returns the shortest distance
# between nodes 's' and 'e'. If there is no path, ∞ is returned.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
# e - the index of the end node (0 ≤ e < n)
function dijkstra(g, n, s, e):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  pq = empty priority queue
  pq.insert((s, 0))
  while pq.size() != 0:
    index, minValue = pq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        pq.insert((edge.to, newDist))
    if index == e:
      return dist[e]
  return ∞
```

Eager Dijkstra's using an Indexed Priority Queue

Our current lazy implementation of Dijkstra's inserts **duplicate key-value pairs** (keys being the node index and the value being the shortest distance to get to that node) in our PQ because it's more efficient to insert a new key-value pair in **O(log(n))** than it is to update an existing key's value in **O(n)**.

This approach is inefficient for dense graphs because we end up with **several stale outdated key-value pairs** in our PQ. The eager version of Dijkstra's avoids duplicate key-value pairs and supports efficient value updates in **O(log(n))** by using an **Indexed Priority Queue (IPQ)**

Indexed Priority Queue DS Video

<insert video clip>

github.com/williamfiset/data-structures

Eager Dijkstra's

(index, dist)
key-value pairs

dist =

0	1	2	3	4	5
∞	∞	∞	∞	∞	∞

Eager Dijkstra's

(index, dist)
key-value pairs

(0, 0)

dist =

0	1	2	3	4	5
0	∞	∞	∞	∞	∞

Eager Dijkstra's

(index, dist)
key-value pairs

→ (0, 0)

dist =

0	1	2	3	4	5
0	∞	∞	∞	∞	∞

Eager Dijkstra's

(index, dist)
key-value pairs

→ (0, 0)
(1, 5)

dist =

0	1	2	3	4	5
0	5	∞	∞	∞	∞

Best distance from node 0 to node 1:
dist[0] + edge.cost = 0 + 5 = 5

Eager Dijkstra's

(index, dist)
key-value pairs

→ (0, 0)
(1, 5)
(2, 1)

dist =

0	1	2	3	4	5
0	5	1	∞	∞	∞

Best distance from node 0 to node 2:
dist[0] + edge.cost = 0 + 1 = 1

Eager Dijkstra's

(index, dist)
key-value pairs

→ ~~(0, 0)~~
(1, 5)
(2, 1)

dist =

0	1	2	3	4	5
0	5	1	∞	∞	∞

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 5)
→ (2, 1)

dist =

0	1	2	3	4	5
0	5	1	∞	∞	∞

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 5)
→ (2, 1)
(4, 13)

dist =

0	1	2	3	4	5
0	5	1	∞	13	∞

Best distance from node 2 to node 4:
dist[2] + edge.cost = 1 + 12 = 13

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
→ (2, 1)
(4, 13)

dist =

0	1	2	3	4	5
0	4	1	∞	13	∞

Best distance from node 2 to node 1:
dist[2] + edge.cost = 1 + 3 = 4

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
(1, 4)
→ ~~(2, 1)~~
(4, 13)

dist =

0	1	2	3	4	5
0	4	1	∞	13	∞

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
(4, 13)

dist =

0	1	2	3	4	5
0	4	1	∞	13	∞

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
(4, 13)

	0	1	2	3	4	5
dist =	**0**	**4**	**1**	**∞**	**13**	**∞**

Node 2 has already been visited so we cannot improve it's already best distance

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
(4, 13)

	0	1	2	3	4	5
dist =	**0**	**4**	**1**	**∞**	**13**	**∞**

dist[1] + edge.cost = 4 + 20 = 24 > dist[4] = 13
so we cannot update best distance.

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
→ (1, 4)
~~(2, 1)~~
(4, 13)
(3, 7)

dist =

0	1	2	3	4	5
0	4	1	7	13	∞

Best distance from node 1 to node 3:
dist[1] + edge.cost = 4 + 3 = 7

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
→ ~~(1, 4)~~
~~(2, 1)~~
(4, 13)
(3, 7)

dist =

0	1	2	3	4	5
0	4	1	7	13	∞

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 13)
→ (3, 7)

dist =

0	1	2	3	4	5
0	4	1	7	13	∞

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 13)
→ (3, 7)

	0	1	2	3	4	5
dist =	**0**	**4**	**1**	**7**	**13**	**∞**

Node 2 is already visited, therefore we cannot improve on its best distance

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 9)
→ (3, 7)

dist =

0	1	2	3	4	5
0	4	1	7	9	∞

Best distance from node 3 to node 4:
dist[3] + edge.cost = 7 + 2 = 9

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 9)
→ (3, 7)
(5, 13)

dist =

0	1	2	3	4	5
0	4	1	7	9	13

Best distance from node 3 to node 5:
dist[3] + edge.cost = 7 + 6 = 13

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
(4, 9)
→ ~~(3, 7)~~
(5, 13)

dist =

0	1	2	3	4	5
0	4	1	7	9	13

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
→ (4, 9)
~~(3, 7)~~
(5, 13)

dist =

0	1	2	3	4	5
0	4	1	7	9	13

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
→ (4, 9)
~~(3, 7)~~
(5, **10**)

dist =

0	1	2	3	4	5
0	**4**	**1**	**7**	**9**	**10**

Best distance from node 4 to node 5:
dist[3] + edge.cost = 9 + 1 = 10

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
→ ~~(4, 9)~~
~~(3, 7)~~
(5, 10)

dist =

0	1	2	3	4	5
0	4	1	7	9	10

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(4, 9)~~
~~(3, 7)~~
→ (5, 10)

dist =

0	1	2	3	4	5
0	4	1	7	9	10

Eager Dijkstra's

(index, dist)
key-value pairs

~~(0, 0)~~
~~(1, 4)~~
~~(2, 1)~~
~~(4, 9)~~
~~(3, 7)~~
~~(5, 10)~~

dist =

0	1	2	3	4	5
0	4	1	7	9	10

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  ipq = empty indexed priority queue
  ipq.insert(s, 0)
  while ipq.size() != 0:
    index, minValue = ipq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        if !ipq.contains(edge.to):
          ipq.insert(edge.to, newDist)
        else:
          ipq.decreaseKey(edge.to, newDist)
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  ipq = empty indexed priority queue
  ipq.insert(s, 0)
  while ipq.size() != 0:
    index, minValue = ipq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        if !ipq.contains(edge.to):
          ipq.insert(edge.to, newDist)
        else:
          ipq.decreaseKey(edge.to, newDist)
  return dist
```

```
# Runs Dijkstra's algorithm and returns an array that contains
# the shortest distance to every node from the start node s.
# g - adjacency list of weighted graph
# n - the number of nodes in the graph
# s - the index of the starting node (0 ≤ s < n)
function dijkstra(g, n, s):
  vis = [false, false, … , false] # size n
  dist = [∞, ∞, … ∞, ∞] # size n
  dist[s] = 0
  ipq = empty indexed priority queue
  ipq.insert(s, 0)
  while ipq.size() != 0:
    index, minValue = ipq.poll()
    vis[index] = true
    if dist[index] < minValue: continue
    for (edge : g[index]):
      if vis[edge.to]: continue
      newDist = dist[index] + edge.cost
      if newDist < dist[edge.to]:
        dist[edge.to] = newDist
        if !ipq.contains(edge.to):
          ipq.insert(edge.to, newDist)
        else:
          ipq.decreaseKey(edge.to, newDist)
  return dist
```

The main advantage to using decreaseKey is to prevent duplicate node indexes to be present in the PQ.

D-ary Heap optimization

When executing Dijkstra's algorithm, especially on dense graphs, there are a lot more updates (i.e decreaseKey operations) to key-value pairs than there are dequeue (poll) operations.

A **D-ary heap** is a heap variant in which each node has D children. This speeds up decrease key operations at the expense of more costly removals.

D-ary Heap (with D = 4)

D-ary Heap (with D = 4)

Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

D-ary Heap (with D = 4)

Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

Assuming we have an Indexed D-ary Heap we can update the key's value in **O(1)** but we still need to adjust its position.

D-ary Heap (with D = 4)

Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

D-ary Heap (with D = 4)

Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

D-ary Heap (with D = 4)

Suppose we want to update the node with index 6 to have a new shortest distance of 1 (a.k.a **decreaseKey**(6, 1))

The whole update took only two operations because the heap was very flat.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

In contrast suppose we want to remove the root node.

In a D-ary heap we have to search D children of the current node to find the minimum (k, v) pair to swap downwards.

D-ary Heap (with D = 4)

Removals are clearly much more expensive, but they are also a lot less common in Dijkstra's than decreaseKey operations.

Optimal D-ary Heap degree

Q: What is the optimal D-ary heap degree to maximize performance of Dijkstra's algorithm?

A: In general **D = E/V** is the best degree to use to balance removals against decreaseKey operations improving Dijkstra's time complexity to **$O(E*\log_{E/V}(V))$** which is much better especially for dense graphs which have lots of decreaseKey operations.

The state of the art

The current state of the art as of now is the **Fibonacci heap** which gives Dijkstra's algorithm a time complexity of **O(E + Vlog(V))**

However, in practice, Fibonacci heaps are very **difficult to implement** and have a **large enough constant amortized overhead** to make them impractical unless your graph is quite large.

Source Code and Slides

Implementation **source code** and **slides** can be found at the following link:

github.com/williamfiset/algorithms

Link in the description:

Next Video: Dijkstra source code

Dijkstra's Algorithm source code

William Fiset

Dijkstra's Shortest Path Algorithm

Graph Theory Video Series

Bellman-Ford Algorithm

William Fiset

BF algorithm overview

In graph theory, the **Bellman-Ford (BF)** algorithm is a **Single Source Shortest Path (SSSP)** algorithm. This means it can find the shortest path from one node to any other node.

However, BF is not ideal for most SSSP problems because it has a time complexity of **O(EV)**. It is better to use Dijkstra's algorithm which is much faster. It is on the order of **0((E+V)log(V))** when using a binary heap priority queue.

BF algorithm overview

However, Dijkstra's algorithm can fail when the graph has negative edge weights. This is when BF becomes really handy because it can be used to detect **negative cycles** and **determine where they occur**.

Finding negative cycles can be useful in many types of applications. One particularly neat application arises in finance when performing an **arbitrage** between two or more markets.

Negative Cycles

Negative cycles can manifest themselves in many ways…

Negative Cycles

Negative cycles can manifest themselves in many ways…

Unaffected node

Directly in negative cycle

Reachable by negative cycle

Negative Cycles

Negative cycles can manifest themselves in many ways…

Negative Cycles

Negative cycles can manifest themselves in many ways…

Unaffected node

Directly in negative cycle

Reachable by negative cycle

Negative Cycles

Negative cycles can manifest themselves in many ways…

Unaffected node

Directly in negative cycle

Reachable by negative cycle

Negative Cycles

Negative cycles can manifest themselves in many ways…

Unaffected node

Directly in negative cycle

Reachable by negative cycle

Negative Cycles

Negative cycles can manifest themselves in many ways…

BF Algorithm Steps

Let’s define a few variables…

Let **E** be the number of edges.

Let **V** be the number of vertices.

Let **S** be the id of the starting node.

Let **D** be an array of size **V** that tracks the best distance from **S** to each node.

BF Algorithm Steps

1) Set every entry in **D** to +∞
2) Set **D**[**S**] = 0
3) Relax each edge V−1 times:

BF Algorithm Steps

1) Set every entry in D to +∞
2) Set D[S] = 0
3) Relax each edge V−1 times:

```
for (i = 0; i < V−1; i = i + 1):
  for edge in graph.edges:
    // Relax edge (update D with shorter path)
    if (D[edge.from] + edge.cost < D[edge.to])
      D[edge.to] = D[edge.from] + edge.cost
```

BF Algorithm Steps

1) Set every entry in D to +∞
2) Set D[S] = 0
3) Relax each edge V−1 times:

```
for (i = 0; i < V−1; i = i + 1):
  for edge in graph.edges:
    // Relax edge (update D with shorter path)
    if (D[edge.from] + edge.cost < D[edge.to])
      D[edge.to] = D[edge.from] + edge.cost

// Repeat to find nodes caught in a negative cycle
for (i = 0; i < V−1; i = i + 1):
  for edge in graph.edges:
    if (D[edge.from] + edge.cost < D[edge.to])
      D[edge.to] = −∞
```

0	∞
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	25
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	25
3	∞
4	∞
5	35
6	∞
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	25
3	∞
4	∞
5	35
6	65
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	25
3	35
4	∞
5	35
6	65
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	25
3	35
4	100
5	35
6	65
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	100
5	35
6	65
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	100
5	35
6	65
7	∞
8	∞
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	65
7	∞
8	∞
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	∞
8	∞
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	∞
8	85
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	85
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	-20
9	200

Iteration 1 complete, 8 more to go…

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	30
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	20
3	30
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	160

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	160

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	160

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	160

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	160

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	160

NOTE: The edges do not need to be chosen in any specific order.

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	160

Iteration 2 complete, 7 more to go…
Let’s fast-forward to the end…

0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160

We're finished with the SSSP part. Now let's detect those negative cycles. If we can relax an edge then there's a negative cycle.

0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-20
3	-∞
4	60
5	35
6	40
7	-10
8	-20
9	160

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-20
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	160

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	160

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞

Unaffected node

Directly in negative cycle

Reachable by negative cycle

0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞

Repeat this for another 8 iterations in order to ensure the cycles fully propagate. In this example, we happened to detect all cycles on the first iteration, but this was a coincidence.

Bellman-Ford Shortest Path Algorithm

Graph Theory Video Series

Floyd-Warshall Algorithm

All Pairs Shortest Path (APSP)

William Fiset

FW algorithm overview

In graph theory, the **Floyd-Warshall (FW)** algorithm is an **All-Pairs Shortest Path (APSP)** algorithm. This means it can find the shortest path between all pairs of nodes.

The time complexity to run FW is **$O(V^3)$** which is **ideal for graphs no larger than a couple hundred nodes**.

Shortest Path (SP) Algorithms

	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	O(V+E)	O((V+E)logV)	O(VE)	$O(V^3)$
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

Reference: Competitive Programming 3, P. 161, Steven & Felix Halim

Shortest Path (SP) Algorithms

	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	O(V+E)	O((V+E)logV)	O(VE)	$O(V^3)$
Recommended graph size	Large	Large/ Medium	Medium/ Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

Reference: Competitive Programming 3, P. 161, Steven & Felix Halim

Graph setup

With FW, the optimal way to **represent our graph is with a 2D adjacency matrix *m*** where cell m[i][j] represents the edge weight of going from node i to node j.

$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & \mathbf{0} & \mathbf{4} & \mathbf{1} & \mathbf{9} \\
B & \mathbf{3} & \mathbf{0} & \mathbf{6} & \mathbf{11} \\
C & \mathbf{4} & \mathbf{1} & \mathbf{0} & \mathbf{2} \\
D & \mathbf{6} & \mathbf{5} & \mathbf{-4} & \mathbf{0}
\end{array}
$$

Graph setup

With FW, the optimal way to **represent our graph is with a 2D adjacency matrix *m*** where cell m[i][j] represents the edge weight of going from node i to node j.

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 4 & 1 & 9 \\ B & 3 & 0 & 6 & 11 \\ C & 4 & 1 & 0 & 2 \\ D & 6 & 5 & -4 & 0 \end{array}$$

NOTE: In the graph above, it is assumed that **the distance from a node to itself is zero**. This is why the diagonal is all zeros.

Graph setup

If there is no edge from node i to node j then set the edge value for m[i][j] to be positive infinity.

$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & 0 & 4 & 1 & \infty \\
B & \infty & 0 & 6 & \infty \\
C & 4 & 1 & 0 & 2 \\
D & \infty & \infty & \infty & 0
\end{array}
$$

Graph setup

If there is no edge from node i to node j then set the edge value for m[i][j] to be positive infinity.

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 4 & 1 & \infty \\ B & \infty & 0 & 6 & \infty \\ C & 4 & 1 & 0 & 2 \\ D & \infty & \infty & \infty & 0 \end{array}$$

IMPORTANT: If your programming language does not support a special constant for +∞ such that ∞ + ∞ = ∞ and x + ∞ = ∞ then **avoid using $2^{31}-1$ as infinity**! This will cause integer overflow; prefer to use a large constant such as 10^7 instead.

The main idea behind the Floyd-Warshall algorithm is to gradually **build up all intermediate routes between nodes i and j** to find the optimal path.

The main idea behind the Floyd-Warshall algorithm is to gradually **build up all intermediate routes between nodes i and j** to find the optimal path.

Suppose our adjacency matrix tells us that the distance from **a** to **b** is: **m[a][b] = 11**

The main idea behind the Floyd-Warshall algorithm is to gradually **build up all intermediate routes between nodes i and j** to find the optimal path.

Suppose there exists a third node, **c**. If **m[a][c] + m[c][b] < m[a][b]** then it's better to route through **c**!

The goal of Floyd-Warshall is to eventually consider going through all possible intermediate nodes on paths of different lengths.

The goal of Floyd-Warshall is to eventually consider going through all possible intermediate nodes on paths of different lengths.

The goal of Floyd-Warshall is to eventually consider going through all possible intermediate nodes on paths of different lengths.

The goal of Floyd-Warshall is to eventually consider going through all possible intermediate nodes on paths of different lengths.

The Memo Table

Let ‘dp’ (short for Dynamic Programming) be a 3D matrix of size n x n x n that acts as a memo table.

dp[k][i][j] = shortest path from i to j routing through nodes {0,1,…,k-1,k}

Start with k = 0, then k = 1, then k = 2, … This gradually builds up the optimal solution routing through 0, then all optimal solutions routing through 0 and 1, then all optimal solutions routing through 0, 1, 2, etc… up until n-1 which stores to APSP solution.

Specifically dp[n-1] is the 2D matrix solution we’re after.

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

$$dp[k][i][j] = m[i][j] \text{ if } k = 0$$

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

```
dp[k][i][j] = m[i][j] if k = 0
```

otherwise:

```
dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])
```

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

```
dp[k][i][j] = m[i][j] if k = 0
```

otherwise:

```
dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])
```

Reuse the best distance from i to j with values routing through nodes {0,1,…,k-1}

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

```
dp[k][i][j] = m[i][j] if k = 0
```

otherwise:

```
dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])
```

Find the best distance from i to j through node k reusing best solutions from {0,1,…,k-1}

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

```
dp[k][i][j] = m[i][j] if k = 0
```

otherwise:

```
dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])
```

The right side of the min function in english essentially says: “go from i to k” and then “go from k to j”

In the beginning the optimal solution from i to j is simply the distance in the adjacency matrix.

```
dp[k][i][j] = m[i][j] if k = 0
```

otherwise:

```
dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k]+dp[k-1][k][j])
```

Visually this looks like:

Currently we're using **$O(V^3)$** memory since our memo table 'dp' has one dimension for each of k, i and j.

Notice that we will be looping over k starting from 0, then 1, 2… and so fourth. The important thing to note here is that previous result builds off the last since we need state k-1 to compute state k. With that being said, it is possible to **compute the solution for k in-place** saving us a dimension of memory and reducing the space complexity to **$O(V^2)$**!

Currently we're using **$O(V^3)$** memory since our memo table 'dp' has one dimension for each of k, i and j.

Notice that we will be looping over k starting from 0, then 1, 2… and so fourth. The important thing to note here is that previous result builds off the last since we need state k-1 to compute state k. With that being said, it is possible to **compute the solution for k in-place** saving us a dimension of memory and reducing the space complexity to **$O(V^2)$**!

The new recurrence relation is:

```
dp[i][j] = m[i][j] if k = 0

otherwise:
dp[i][j] = min(dp[i][j], dp[i][k]+dp[k][j])
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
function setup(m):
  dp = empty matrix of size n x n

  # Should contain null values by default
  next = empty integer matrix of size n x n

  # Do a deep copy of the input matrix and setup
  # the 'next' matrix for path reconstruction.
  for(i := 0; i < n; i++):
    for(j := 0; j < n; j++):
      dp[i][j] = m[i][j]
      if m[i][j] != +∞:
        next[i][j] = j
```

```
function setup(m):
  dp = empty matrix of size n x n

  # Should contain null values by default
  next = empty integer matrix of size n x n

  # Do a deep copy of the input matrix and setup
  # the 'next' matrix for path reconstruction.
  for(i := 0; i < n; i++):
    for(j := 0; j < n; j++):
      dp[i][j] = m[i][j]
      if m[i][j] != +∞:
        next[i][j] = j
```

```
function setup(m):
  dp = empty matrix of size n x n

  # Should contain null values by default
  next = empty integer matrix of size n x n

  # Do a deep copy of the input matrix and setup
  # the 'next' matrix for path reconstruction.
  for(i := 0; i < n; i++):
    for(j := 0; j < n; j++):
      dp[i][j] = m[i][j]
      if m[i][j] != +∞:
        next[i][j] = j
```

```
function setup(m):
  dp = empty matrix of size n x n

  # Should contain null values by default
  next = empty integer matrix of size n x n

  # Do a deep copy of the input matrix and setup
  # the 'next' matrix for path reconstruction.
  for(i := 0; i < n; i++):
    for(j := 0; j < n; j++):
      dp[i][j] = m[i][j]
      if m[i][j] != +∞:
        next[i][j] = j
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

Negative Cycles

What do we mean by a negative cycle?

Negative Cycles

Negative cycles can manifest themselves in many ways…

Directly in negative cycle

Unaffected node

Negative Cycles

Negative cycles can manifest themselves in many ways…

Directly in negative cycle

Unaffected node

Negative Cycles

Negative cycles can manifest themselves in many ways…

Directly in negative cycle

Unaffected node

Negative Cycles

Negative cycles can manifest themselves in many ways…

Directly in negative cycle

Unaffected node

Negative Cycles

Negative cycles can manifest themselves in many ways…

Negative Cycles

The important thing to ask ourselves is does the optimal path from node i to node j go through a **red** node? If so the path is affected by the negative cycle and is compromised.

```
function propagateNegativeCycles(dp, n):

  # Execute FW APSP algorithm a second time but
  # this time if the distance can be improved
  # set the optimal distance to be −∞.
  # Every edge (i, j) marked with −∞ is either
  # part of or reaches into a negative cycle.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = −∞
          next[i][j] = -1
```

```
# Global/class scope variables
n = size of the adjacency matrix
dp = the memo table that will contain APSP soln
next = matrix used to reconstruct shortest paths

function floydWarshall(m):
  setup(m)

  # Execute FW all pairs shortest path algorithm.
  for(k := 0; k < n; k++):
    for(i := 0; i < n; i++):
      for(j := 0; j < n; j++):
        if(dp[i][k] + dp[k][j] < dp[i][j]:
          dp[i][j] = dp[i][k] + dp[k][j]
          next[i][j] = next[i][k]
  # Detect and propagate negative cycles.
  propagateNegativeCycles(dp, n)

  # Return APSP matrix
  return dp
```

```
# Reconstructs the shortest path between nodes
# 'start' and 'end'. You must run the
# floydWarshall solver before calling this method.
# Returns null if path if affected by negative cycle.
function reconstructPath(start, end):
  path = []
  # Check if there exists a path between
  # the start and the end node.
  if dp[start][end] == +∞: return path

  at := start
  # Reconstruct path from next matrix
  for(;at != end; at = next[at][end]):
    if at == -1: return null
    path.add(at)

  if next[at][end] == -1: return null
  path.add(end)
  return path
```

Source Code Link

Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description

Next Video: Floyd–Warshall source code

Floyd-Warshall All Pairs Shortest Path

Graph Theory Video Series

Algorithm to Find Bridges and Articulation Points

William Fiset

What are bridges & articulation points?

A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.

What are bridges & articulation points?

A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.

What are bridges & articulation points?

An **articulation point / cut vertex** is any node in a graph whose removal increases the number of connected components.

What are bridges & articulation points?

An **articulation point / cut vertex** is any node in a graph whose removal increases the number of connected components.

What are bridges & articulation points?

Bridges and **articulation points** are important in graph theory because they often hint at **weak points, bottlenecks or vulnerabilities in a graph**. Therefore, it's important to be able to quickly find/detect when and where these occur.

Both problems are related so we will develop an algorithm to find bridges and then modify it slightly to find articulation points.

Bridges algorithm

Start at any node and do a Depth First Search (DFS) traversal labeling nodes with an increasing id value as you go. Keep **track the id of each node** and the **smallest low-link** value. During the DFS, bridges will be found where the id of the node your edge is coming from is less than the low link value of the node your edge is going to.

NOTE: The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node when doing a DFS (including itself).

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

DFS traversal

Undirected edge

Directed edge

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Initially all low-link values can be initialized to the node ids.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 1 is 0 since node 0 is reachable from node 1.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

The low-link value of node 2 is 0 since node 0 is reachable from node 2.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Cannot update low-link values for nodes 3, 4 and 5.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Node 6’s low-link value can be updated to 5 since node 5 is reachable from node 6.

The **low-link** value of a node is defined as the smallest [lowest] id reachable from that node using forward and backward edges.

Undirected edge

Directed edge

Now notice that the condition for a directed edge ‘e’ to have nodes that belong to a bridge is when the **id(e.from) < lowlink(e.to)***

* Where **e.from** is the node the directed edge starts at and **e.to** is the node the directed edge ends at.

Undirected edge

Directed edge

Now notice that the condition for a directed edge ‘e’ to have nodes that belong to a bridge is when the **id(e.from) < lowlink(e.to)***

* Where **e.from** is the node the directed edge starts at and **e.to** is the node the directed edge ends at.

Undirected edge

Directed edge

Complexity

What’s the runtime of our algorithm to find bridges? Right now we’re doing one DFS to label all the nodes plus V more DFSs to find all the low-link values, giving us roughly:

O(V(V+E))

Fortunately, we are able do better by updating the low-link values in one pass for

O(V+E)

```
id = 0
g = adjacency list with undirected edges
n = size of the graph

# In these arrays index i represents node i
ids = [0, 0, … 0, 0]          # Length n
low = [0, 0, … 0, 0]          # Length n
visited = [false, …, false] # Length n

function findBridges():
  bridges = []
  # Finds all bridges in the graph across
  # various connected components.
  for (i = 0; i < n; i = i + 1):
    if (!visited[i]):
      dfs(i, -1, bridges)
  return bridges
```

```
id = 0
g = adjacency list with undirected edges
n = size of the graph

# In these arrays index i represents node i
ids = [0, 0, … 0, 0]          # Length n
low = [0, 0, … 0, 0]          # Length n
visited = [false, …, false] # Length n

function findBridges():
  bridges = []
  # Finds all bridges in the graph across
  # various connected components.
  for (i = 0; i < n; i = i + 1):
    if (!visited[i]):
      dfs(i, -1, bridges)
  return bridges
```

```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
  visited[at] = true
  id = id + 1
  low[at] = ids[at] = id

  # For each edge from node ‘at’ to node ‘to’
  for (to : g[at]):
    if to == parent: continue
    if (!visited[to]):
      dfs(to, at, bridges)
      low[at] = min(low[at], low[to])
      if (ids[at] < low[to]):
        bridges.add(at)
        bridges.add(to)
    else:
      low[at] = min(low[at], ids[to])
```

```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
  visited[at] = true
  id = id + 1
  low[at] = ids[at] = id

  # For each edge from node 'at' to node 'to'
  for (to : g[at]):
    if to == parent: continue
    if (!visited[to]):
      dfs(to, at, bridges)
      low[at] = min(low[at], low[to])
      if (ids[at] < low[to]):
        bridges.add(at)
        bridges.add(to)
    else:
      low[at] = min(low[at], ids[to])
```

```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
  visited[at] = true
  id = id + 1
  low[at] = ids[at] = id

  # For each edge from node ‘at’ to node ‘to’
  for (to : g[at]):
    if to == parent: continue
    if (!visited[to]):
      dfs(to, at, bridges)
      low[at] = min(low[at], low[to])
      if (ids[at] < low[to]):
        bridges.add(at)
        bridges.add(to)
    else:
      low[at] = min(low[at], ids[to])
```

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
  visited[at] = true
  id = id + 1
  low[at] = ids[at] = id

  # For each edge from node 'at' to node 'to'
  for (to : g[at]):
    if to == parent: continue
    if (!visited[to]):
      dfs(to, at, bridges)
      low[at] = min(low[at], low[to])
      if (ids[at] < low[to]):
        bridges.add(at)
        bridges.add(to)
    else:
      low[at] = min(low[at], ids[to])
```

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Undirected edge

Directed edge

Undirected edge

Directed edge

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current
Visited
Unvisited
0
0
0
1
1
2
5
5
3
3
4
4

Undirected edge
Directed edge

Current
Visited
Unvisited
0
0
0
1
1
2
5
5
3
3
4
4

Undirected edge
Directed edge

Current
Visited
Unvisited
0
0
0
1
1
2
5
5
6
6
3
3
4
4
Undirected edge
Directed edge

Current
Visited
Unvisited
0
0
0
1
1
2
5
5
6
6
3
3
4
4
Undirected edge
Directed edge

Current
Visited
Unvisited
0
0
0
1
1
2
3
3
4
4
5
5
6
6
7
7

Undirected edge
Directed edge

Current
Visited
Unvisited
0
0
0
1
1
2
3
3
4
4
5
5
6
6
7
7

Undirected edge
Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current Visited Unvisited

Undirected edge

Directed edge

Current
Visited
Unvisited
0
0
0
1
1
2
3
3
4
4
5
5
6
6
7
5
8
5
Undirected edge
Directed edge

```
# Perform Depth First Search (DFS) to find bridges.
# at = current node, parent = previous node. The
# bridges list is always of even length and indexes
# (2*i, 2*i+1) form a bridge. For example, nodes at
# indexes (0, 1) are a bridge, (2, 3) is another etc...
function dfs(at, parent, bridges):
  visited[at] = true
  id = id + 1
  low[at] = ids[at] = id

  # For each edge from node 'at' to node 'to'
  for (to : g[at]):
    if to == parent: continue
    if (!visited[to]):
      dfs(to, at, bridges)
      low[at] = min(low[at], low[to])
      if (ids[at] < low[to]):
        bridges.add(at)
        bridges.add(to)
    else:
      low[at] = min(low[at], ids[to])
```

Current
Visited
Unvisited
0
0
0
1
1
2
3
3
4
4
5
5
5
5
6
7
8
Undirected edge
Directed edge

Current
Visited
Unvisited
0
0
0
1
1
2
3
3
4
4
5
5
5
5
6
7
8
Undirected edge
Directed edge

Undirected edge

Directed edge

Current
Visited
Unvisited
0
0
0
1
2
0
3
3
4
4
5
5
5
6
7
5
8
5
Undirected edge
Directed edge

Current
Visited
Unvisited
0
0
0
0
1
2
3
4
5
5
5
5
6
7
8
Undirected edge
Directed edge

Current
Visited
Unvisited
0
0
0
0
1
2
3
4
5
5
5
5
6
7
8
Undirected edge
Directed edge

The condition for a directed edge ‘e’ to have nodes that belong to a bridge is when the **id(e.from) < lowlink(e.to)**

Undirected edge

Directed edge

The condition for a directed edge ‘e’ to have nodes that belong to a bridge is when the **id(e.from) < lowlink(e.to)**

Undirected edge

Directed edge

The condition for a directed edge ‘e’ to have nodes that belong to a bridge is when the **id(e.from) < lowlink(e.to)**

Undirected edge

Directed edge

The condition for a directed edge ‘e’ to have nodes that belong to a bridge is when the **id(e.from) < lowlink(e.to)**

Undirected edge

Directed edge

Articulation points

Articulation points are related very closely to bridges. It won’t take much modification to the finding bridges algorithm to find articulation points.

Articulation points

Simple observation about articulation points:

On a connected component with three or more vertices if an edge (u, v) is a bridge then either u or v is an articulation point.

Articulation points

Simple observation about articulation points:

On a connected component with three or more vertices if an edge (u, v) is a bridge then either u or v is an articulation point.

Articulation points

However, this condition alone is not sufficient to capture all articulation points. There exist cases where there is an articulation point without a bridge:

There are no bridges but node 2 is an articulation point since its removal would cause the graph to split into two components.

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

Articulation points

On the callback, if **id(e.from) == lowlink(e.to)** then there was a **cycle**.

The indication of a cycle back to the original node implies an articulation point.

Articulation points

The only time **id(e.from) == lowlink(e.to)** fails is when the **starting node** has 0 or 1 outgoing directed edges. This is because either the node is a singleton (0 case) or the node in trapped in a **cycle** (1 case).

Here the condition is met, but the starting node only has 1 outgoing edge. Therefore, the start node is <u>not</u> an articulation point.

Start node

Articulation points

The only time **id(e.from) == lowlink(e.to)** fails is when the **starting node** has 0 or 1 outgoing directed edges. This is because either the node is a singleton (0 case) or the node in trapped in a **cycle** (1 case).

However, when there are more than 1 outgoing edges the starting node can escape the cycle and thus becomes an articulation point!

Start node

```
id = 0
g = adjacency list with undirected edges
n = size of the graph
outEdgeCount = 0

# In these arrays index i represents node i
low = [0, 0, … 0, 0]          # Length n
ids = [0, 0, … 0, 0]          # Length n
visited = [false, …, false] # Length n
isArt   = [false, …, false] # Length n

function findArtPoints():
  for (i = 0; i < n; i = i + 1):
    if (!visited[i]):
      outEdgeCount = 0 # Reset edge count
      dfs(i, i, -1)
      isArt[i] = (outEdgeCount > 1)
  return isArt
```

```
# Perform DFS to find articulation points.
function dfs(root, at, parent):
  if (parent == root): outEdgeCount++
  visited[at] = true
  id = id + 1
  low[at] = ids[at] = id

  # For each edge from node 'at' to node 'to'
  for (to : g[at]):
    if to == parent: continue
    if (!visited[to]):
      dfs(root, to, at)
      low[at] = min(low[at], low[to])
      # Articulation point found via bridge
      if (ids[at] < low[to]):
        isArt[at] = true
      # Articulation point found via cycle
      if (ids[at] == low[to]):
        isArt[at] = true
    else:
      low[at] = min(low[at], ids[to])
```

Being explicit here.
However, this could just
be a <= clause.

Source Code Link

Slides/source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description:

Bridges and Articulation Points

Bridges and Articulation Points Source Code

Graph Theory Video Series

Tarjan’s Algorithm for Finding Strongly Connected Components

William Fiset

What are SCCs?

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.

What are SCCs?

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

The low-link of node 1 is 0, since node 0 is node with the lowest id reachable from node 1

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

The low-link of node 3 is 2, since node 2 is node with the lowest id reachable from node 3

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Started DFS on the rightmost node, resumed DFS on node 2, and then resumed DFS on node 4 to finish labelling all nodes in the graph.

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

Low-Link Values

The **low-link** value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

All low link values are the same, but there are multiple SCCs!

IMPORTANT: Depending on where the DFS starts, and the order in which nodes/edges are visited, the low-link values for identifying SCCs could be wrong. In the context of Tarjan's SCC algorithm, we maintain an invariant that prevents SCCs to interfere with the low link value of other SCCs.

The Stack Invariant

To cope with the random traversal order of the DFS, Tarjan’s algorithm maintains a set (often as a stack) of valid nodes from which to update low-link values from.

Nodes are added to the stack [set] of valid nodes as they’re explored for the first time.

Nodes are removed from the stack [set] each time a complete SCC is found.

New low-link update condition

If u and v are nodes in a graph and we're currently exploring u then our new low-link update condition is that:

To update node u's low-link value to node v's low-link value there has to be a path of edges from u to v and **node v must be on the stack**.

Time Complexity

Another difference we're going to make to finding all low-link values is that instead of finding low-link values after the fact we're going to update them "on the fly" during the DFS so we can get a linear **O(V+E)** time complexity :)

Tarjan’s Algorithm Overview

Mark the id of each node as unvisited.

Start DFS. Upon visiting a node assign it an id and a low-link value. Also mark current nodes as visited and add them to a seen stack.

On DFS callback, if the previous node is on the stack then min the current node’s low-link value with the last node’s low-link value*.

After visiting all neighbors, if the current node started a connected component** then pop nodes off stack until current node is reached.

*This allows low-link values to propagate throughout cycles.

As we will see, a node started a connected component if its **id equals its low link value

Unvisited

Visiting neighbours

Visited all neighbours

Stack

If a node’s colour is **grey** or **orange** then it is on the stack and we can update its low-link value.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Start DFS anywhere.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Unvisited
Visiting neighbours
Visited all neighbours
Stack
2
1
0
2
2
0
1
0
1

Unvisited
Visiting neighbours
Visited all neighbours
Stack
2
1
0
2
2
0
1
0
1

```
lowlink[2] = min(lowlink[2], lowlink[0])
           = 0
```

Unvisited
Visiting neighbours
Visited all neighbours
Stack
2
1
0
0
2
0
1
0
1

```
lowlink[1] = min(lowlink[1], lowlink[2])
           = 0
```

Unvisited
Visiting neighbours
Visited all neighbours
Stack
2
1
0
0
2
0
1
0
0

```
lowlink[0] = min(lowlink[0], lowlink[1])
           = 0
```

Unvisited

Visiting neighbours

Visited all neighbours

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

We're not done exploring the graph so pick another starting node at random.

Unvisited
Visiting neighbours
Visited all neighbours
Stack
3
3
0
2
0
1
0
0
3

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Unvisited
Visiting neighbours
Visited all neighbours
Stack
3
4
0
3
4
2
0
1
0
0
4
3

Unvisited
Visiting neighbours
Visited all neighbours
Stack
3
4
0
3
4
2
0
1
0
0
4
3

Unvisited
Visiting neighbours
Visited all neighbours
Stack
5
4
3
3
4
0
3
4
2
5
0
1
5
0
0

Unvisited
Visiting neighbours
Visited all neighbours
Stack
5
4
3
3
4
0
3
4
2
5
0
1
5
0
0

Node 0 is not on stack so don’t min with its low-link value.

Unvisited
Visiting neighbours
Visited all neighbours
Stack
5
4
3
3
4
0
3
4
2
5
0
1
5
0
0

Unvisited
Visiting neighbours
Visited all neighbours
Stack
6
5
4
3
3
4
6
0
3
4
6
2
5
0
1
5
0
0

Unvisited
Visiting neighbours
Visited all neighbours
Stack
6
5
4
3
3
4
6
0
3
4
6
2
5
0
1
5
0
0

Node 2 is not on stack so don’t min with its low-link value.

Unvisited
Visiting neighbours
Visited all neighbours
Stack
6
5
4
3
3
4
6
0
3
4
6
2
5
0
1
5
0
0

```
lowlink[6] = min(lowlink[6], lowlink[4])
           = 4
```

Unvisited
Visiting neighbours
Visited all neighbours
Stack
6
5
4
3
3
4
4
0
3
4
6
2
5
0
1
5
0
0

Node 0 is not on stack so don’t min with its low-link value.

Unvisited
Visiting neighbours
Visited all neighbours
Stack
6
5
4
3
3
4
4
0
3
4
6
2
5
0
1
5
0
0

```
lowlink[5] = min(lowlink[5], lowlink[6])
           = 4
```

Unvisited
Visiting neighbours
Visited all neighbours
Stack
6
5
4
3
3
4
4
0
3
4
6
2
5
0
1
4
0
0

```
lowlink[4] = min(lowlink[4], lowlink[5])
           = 4
```

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

3

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Node 4 is not on stack so don’t min with its low-link value.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

3

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Unvisited

Visiting neighbours

Visited all neighbours

Stack

Node 5 is not on stack so don’t min with its low-link value.

Unvisited

Visiting neighbours

Visited all neighbours

Stack

```
lowlink[6] = min(lowlink[6], lowlink[3])
           = 3
```

Unvisited
Visiting neighbours
Visited all neighbours
Stack
3
4
4
0
3
4
6
2
7
5
0
1
3
4
0
0
7
3

```
lowlink[3] = min(lowlink[3], lowlink[6])
           = 3
```

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

When a completed SCC is found (current node has visited all its neighbours and its lowlink value equals its id) pop off all associated nodes off the stack.

Unvisited
Visiting neighbours
Visited all neighbours
3
4
4
0
3
4
6
2
7
5
0
1
3
4
0
0

```
UNVISITED = -1
n = number of nodes in graph
g = adjacency list with directed edges

id = 0          # Used to give each node an id
sccCount = 0 # Used to count number of SCCs found

# Index i in these arrays represents node i
ids = [0, 0, … 0, 0]                    # Length n
low = [0, 0, … 0, 0]                    # Length n
onStack = [false, false, …, false] # Length n
stack = an empty stack data structure

function findSccs():
  for(i = 0; i < n; i++): ids[i] = UNVISITED
  for(i = 0; i < n; i++):
    if(ids[i] == UNVISITED):
      dfs(i)
  return low
```

```
UNVISITED = -1
n = number of nodes in graph
g = adjacency list with directed edges

id = 0          # Used to give each node an id
sccCount = 0 # Used to count number of SCCs found

# Index i in these arrays represents node i
ids = [0, 0, … 0, 0]                    # Length n
low = [0, 0, … 0, 0]                    # Length n
onStack = [false, false, …, false] # Length n
stack = an empty stack data structure

function findSccs():
  for(i = 0; i < n; i++): ids[i] = UNVISITED
  for(i = 0; i < n; i++):
    if(ids[i] == UNVISITED):
      dfs(i)
  return low
```

```
UNVISITED = -1
n = number of nodes in graph
g = adjacency list with directed edges

id = 0          # Used to give each node an id
sccCount = 0 # Used to count number of SCCs found

# Index i in these arrays represents node i
ids = [0, 0, … 0, 0]                    # Length n
low = [0, 0, … 0, 0]                    # Length n
onStack = [false, false, …, false] # Length n
stack = an empty stack data structure

function findSccs():
  for(i = 0; i < n; i++): ids[i] = UNVISITED
  for(i = 0; i < n; i++):
    if(ids[i] == UNVISITED):
      dfs(i)
  return low
```

```
UNVISITED = -1
n = number of nodes in graph
g = adjacency list with directed edges

id = 0          # Used to give each node an id
sccCount = 0 # Used to count number of SCCs found

# Index i in these arrays represents node i
ids = [0, 0, … 0, 0]                    # Length n
low = [0, 0, … 0, 0]                    # Length n
onStack = [false, false, …, false] # Length n
stack = an empty stack data structure

function findSccs():
  for(i = 0; i < n; i++): ids[i] = UNVISITED
  for(i = 0; i < n; i++):
    if(ids[i] == UNVISITED):
      dfs(i)
  return low
```

```
UNVISITED = -1
n = number of nodes in graph
g = adjacency list with directed edges

id = 0          # Used to give each node an id
sccCount = 0 # Used to count number of SCCs found

# Index i in these arrays represents node i
ids = [0, 0, … 0, 0]                    # Length n
low = [0, 0, … 0, 0]                    # Length n
onStack = [false, false, …, false] # Length n
stack = an empty stack data structure

function findSccs():
  for(i = 0; i < n; i++): ids[i] = UNVISITED
  for(i = 0; i < n; i++):
    if(ids[i] == UNVISITED):
      dfs(i)
  return low
```

```
UNVISITED = -1
n = number of nodes in graph
g = adjacency list with directed edges

id = 0          # Used to give each node an id
sccCount = 0 # Used to count number of SCCs found

# Index i in these arrays represents node i
ids = [0, 0, … 0, 0]                    # Length n
low = [0, 0, … 0, 0]                    # Length n
onStack = [false, false, …, false] # Length n
stack = an empty stack data structure

function findSccs():
  for(i = 0; i < n; i++): ids[i] = UNVISITED
  for(i = 0; i < n; i++):
    if(ids[i] == UNVISITED):
      dfs(i)
  return low
```

```
UNVISITED = -1
n = number of nodes in graph
g = adjacency list with directed edges

id = 0          # Used to give each node an id
sccCount = 0 # Used to count number of SCCs found

# Index i in these arrays represents node i
ids = [0, 0, … 0, 0]                    # Length n
low = [0, 0, … 0, 0]                    # Length n
onStack = [false, false, …, false] # Length n
stack = an empty stack data structure

function findSccs():
  for(i = 0; i < n; i++): ids[i] = UNVISITED
  for(i = 0; i < n; i++):
    if(ids[i] == UNVISITED):
      dfs(i)
  return low
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of ‘at’
  # if we're at the start of a SCC empty the seen
  # stack until we’re back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of ‘at’
  # if we're at the start of a SCC empty the seen
  # stack until we’re back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of 'at'
  # if we're at the start of a SCC empty the seen
  # stack until we're back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of 'at'
  # if we're at the start of a SCC empty the seen
  # stack until we're back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of 'at'
  # if we're at the start of a SCC empty the seen
  # stack until we're back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of 'at'
  # if we're at the start of a SCC empty the seen
  # stack until we're back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of 'at'
  # if we're at the start of a SCC empty the seen
  # stack until we're back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of 'at'
  # if we're at the start of a SCC empty the seen
  # stack until we're back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

```
function dfs(at):
  stack.push(at)
  onStack[at] = true
  ids[at] = low[at] = id++

  # Visit all neighbours & min low-link on callback
  for(to : g[at]):
    if(ids[to] == UNVISITED): dfs(to)
    if(onStack[to]): low[at] = min(low[at],low[to])

  # After having visited all the neighbours of ‘at’
  # if we're at the start of a SCC empty the seen
  # stack until we’re back to the start of the SCC.
  if(ids[at] == low[at]):
    for(node = stack.pop();;node = stack.pop()):
      onStack[node] = false
      low[node] = ids[at]
      if(node == at): break
    sccCount++
```

Tarjan's Strongly Connected Component Algorithm (UPDATED)

Tarjan's SCC Algorithm Source Code

Graph Theory Video Series

Strongly Connected Components (SCCs)

Kosaraju’s algorithm

William Fiset

What are SCCs?

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.

What are SCCs?

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.

Kosaraju’s algorithm intuition

We are able to find a graph traversal ordering which enables us to visit SCCs

Kosaraju’s algorithm intuition

Kosaraju's algorithm intuition

[0, 1, 6, 2, 3, 7, 4, 5, 9, 8]

Kosaraju's algorithm intuition

Nodes than can reach

[1, 3, 0, 5, 2, 4]

Kosaraju's algorithm intuition

[0, 3, 2, 1, 7, 6, 5, 4]

The reverse post order ordering presents the nodes in a
order in which the node(s) which leak the most appear
first. In G^T, these nodes will be those which are part
the "smallest SCC", that is the SCC which leak the least

Kosaraju’s algorithm intuition

[0, 3, 2, 1, 7, 6, 5, 4]

Graph Theory Video Series

Traveling Salesman Problem (TSP) with Dynamic Programming

William Fiset

What is the TSP?

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” – Wiki

What is the TSP?

In other words, the problem is: given a **complete graph** with weighted edges (as an adjacency matrix) what is the **Hamiltonian cycle** (path that visits every node once) of minimum cost?

	A	B	C	D
A	0	4	1	9
B	3	0	6	11
C	4	1	0	2
D	6	5	-4	0

What is the TSP?

In other words, the problem is: given a **complete graph** with weighted edges (as an adjacency matrix) what is the **Hamiltonian cycle** (path that visits every node once) of minimum cost?

	A	B	C	D
A	0	4	1	9
B	3	0	6	11
C	4	1	0	2
D	6	5	-4	0

Full tour: A -> D -> C -> B -> A

Tour cost: 9 + −4 + 1 + 3 = 9

What is the TSP?

Finding the optimal solution to the TSP problem is **very hard**; in fact, the problem is known to be **NP-Complete**.

Brute force solution

The brute force way to solve the TSP is to compute the cost of every possible tour. This means we have to try all possible permutations of node orderings which takes **O(n!)** time.

	A	B	C	D
A	0	4	1	9
B	3	0	6	11
C	4	1	0	2
D	6	5	-4	0

Tour	Cost	Tour	Cost
A B C D	18	C A B D	15
A B D C	15	C A D B	24
A C B D	19	C B A D	9
A C D B	11	C B D A	19
A D B C	24	C D A B	18
A D C B	9	C D B A	11
B A C D	11	D A B C	18
B A D C	9	D A C B	19
B C A D	24	D B A C	11
B C D A	18	D B C A	24
B D A C	19	D C A B	15
B D C A	15	D C B A	9

TSP with DP

The dynamic programming solution to the TSP problem significantly improves on the time complexity, taking it from **$O(n!)$** to **$O(n^2 2^n)$**.

At first glance, this may not seem like a substantial improvement, however, it now makes solving this problem feasible on graphs with up to roughly 23 nodes on a typical computer.

TSP with DP

n	n!	$n^2 2^n$
1	1	2
2	2	16
3	6	72
4	24	256
5	120	800
6	720	2304
...	...	...
15	1307674368000	7372800
16	20922789888000	16777216
17	355687428096000	37879808

TSP with DP

The main idea will be to compute the optimal solution for all the subpaths of length **N** while using information from the already known optimal partial tours of length **N-1**.

TSP with DP

Before starting, make sure to **select a node $0 \leq S < N$ to be the designated starting node for the tour.**

TSP with DP

Before starting, make sure to **select a node $0 \leq S < N$ to be the designated starting node for the tour**.

For this example let **S** = node 0

TSP with DP

Next, compute and store the optimal value from **S** to each node **X** (≠ **S**). This will solve TSP problem for all paths of length n = 2.

TSP with DP

Next, compute and store the optimal value from **S** to each node **X** (≠ **S**). This will solve TSP problem for all paths of length n = 2.

TSP with DP

To compute the optimal solution for paths of length 3, we need to remember (store) two things from each of the n = 2 cases:

1) The **set of visited nodes** in the subpath

2) The **index of the last visited node** in the path

Together these two things form our dynamic programming state. There are N possible nodes that we could have visited last and 2^N possible subsets of visited nodes. Therefore the space needed to store the answer to each subproblem is bounded by **$O(N2^N)$**.

Visited Nodes as a Bit Field

The best way to represent the set of visited nodes is to use **a single 32-bit integer**. A 32-bit int is compact, quick and allows for easy caching in a memo table.

Visited Nodes as a Bit Field

The best way to represent the set of visited nodes is to use **a single 32-bit integer**. A 32-bit int is compact, quick and allows for easy caching in a memo table.

State
Binary rep: 0011_2 = 3
Last node: 1

State
Binary rep: 1001_2 = 9
Last node: 3

State
Binary rep: 0101_2 = 5
Last node: 2

TSP with DP

To solve $3 \leq n \leq N$, we're going to take the solved subpaths from n-1 and add another edge extending to a node which has not already been visited from the last visited node (which has been saved).

State

Binary rep: $1001_2 = 9$

Last node: 3

TSP with DP

To solve $3 \leq n \leq \mathbf{N}$, we're going to take the solved subpaths from n-1 and add another edge extending to a node which has not already been visited from the last visited node (which has been saved).

State
Binary rep: 1011_2 = 11
Last node: 1

State
Binary rep: 1001_2 = 9
Last node: 3

State
Binary rep: 1101_2 = 13
Last node: 2

TSP with DP

To complete the TSP tour, we need to connect our tour back to the start node **S.**

Loop over the end state* in the memo table for every possible end position and minimize the lookup value plus the cost of going back to **S.**

* *The end state is the state where the binary representation is composed of **N** 1's*

TSP Pseudocode

In the next few slides we'll look at some pseudocode for the TSP problem.

WARNING: The following slides make use of advanced bit manipulation techniques. Make sure you're very comfortable with the binary operators <<, &, | and ^

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
# Initializes the memo table by caching
# the optimal solution from the start
# node to every other node.
function setup(m, memo, S, N):

  for (i = 0; i < N; i = i + 1):

    if i == S: continue

    # Store the optimal value from node S
    # to each node i (this is given as input
    # in the adjacency matrix m).
    memo[i][1 << S | 1 << i] = m[S][i]
```

Node S

Node i

State
Binary rep: 0011_2 = 3
Last node: 1

State
Binary rep: 1001_2 = 9
Last node: 3

State
Binary rep: 0101_2 = 5
Last node: 2

```
# Initializes the memo table by caching
# the optimal solution from the start
# node to every other node.
function setup(m, memo, S, N):

  for (i = 0; i < N; i = i + 1):

    if i == S: continue

    # Store the optimal value from node S
    # to each node i (this is given as input
    # in the adjacency matrix m).
    memo[i][1 << S | 1 << i] = m[S][i]
```

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
function solve(m, memo, S, N):
 for (r = 3; r <= N; r++):
  # The combinations function generates all bit sets
  # of size N with r bits set to 1. For example,
  # combinations(3, 4) = {0111₂, 1011₂, 1101₂, 1110₂}
  for subset in combinations(r, N):
   if notIn(S, subset): continue
   for (next = 0; next < N; next = next + 1):
    if next == S || notIn(next, subset): continue
    # The subset state without the next node
    state = subset ^ (1 << next)
    minDist = +∞
    # 'e' is short for end node.
    for (e = 0; e < N; e = e + 1):
     if e == S || e == next || notIn(e, subset)):
      continue
     newDistance = memo[e][state] + m[e][next]
     if (newDistance < minDist): minDist = newDistance
    memo[next][subset] = minDist
# Returns true if the iᵗʰ bit in 'subset' is not set
function notIn(i, subset):
 return ((1 << i) & subset) == 0
```

```
function solve(m, memo, S, N):
 for (r = 3; r <= N; r++):
  # The combinations function generates all bit sets
  # of size N with r bits set to 1. For example,
  # combinations(3, 4) = {0111₂, 1011₂, 1101₂, 1110₂}
  for subset in combinations(r, N):
   if notIn(S, subset): continue
   for (next = 0; next < N; next = next + 1):
    if next == S || notIn(next, subset): continue
    # The subset state without the next node
    state = subset ^ (1 << next)
    minDist = +∞
    # 'e' is short for end node.
    for (e = 0; e < N; e = e + 1):
     if e == S || e == next || notIn(e, subset)):
      continue
     newDistance = memo[e][state] + m[e][next]
     if (newDistance < minDist): minDist = newDistance
    memo[next][subset] = minDist
# Returns true if the iᵗʰ bit in 'subset' is not set
function notIn(i, subset):
 return ((1 << i) & subset) == 0
```

```
# Generate all bit sets of size n with r bits set to 1.
function combinations(r, n):
  subsets = []
  combinations(0, 0, r, n, subsets)
  return subsets

# Recursive method to generate bit sets.
function combinations(set, at, r, n, subsets):
  if r == 0:
    subsets.add(set)
  else:
    for (i = at; i < n; i = i + 1):
      # Flip on ith bit
      set = set | (1 << i)

      combinations(set, i + 1, r - 1, n, subsets)

      # Backtrack and flip off ith bit
      set = set & ~(1 << i)
```

NOTE: For a more detailed explanation on generating combinations see video “Backtracking tutorial: power set”

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
function findMinCost(m, memo, S, N):

  # The end state is the bit mask with
  # N bits set to 1 (equivalently 2^N - 1)
  END_STATE = (1 << N) - 1

  minTourCost = +∞

  for (e = 0; e < N; e = e + 1):

    if e == S: continue

    tourCost = memo[e][END_STATE] + m[e][S]
    if tourCost < minTourCost:
      minTourCost = tourCost

  return minTourCost
```

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
function findOptimalTour(m, memo, S, N):
  lastIndex = S
  state = (1 << N) - 1; # End state
  tour = array of size N+1

  for (i = N-1; i >= 1; i--):
    index = -1
    for (j = 0; j < N; j++):
      if j == S || notIn(j, state): continue
      if (index == -1) index = j
      prevDist = memo[index][state] + m[index]lastIndex]
      newDist  = memo[j][state] + m[j][lastIndex];
      if (newDist < prevDist) index = j

    tour[i] = index
    state = state ^ (1 << index)
    lastIndex = index

  tour[0] = tour[N] = S
  return tour
```

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

Next video: Source Code!

Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description:

Traveling Salesman Problem Dynamic Programming

$$\begin{bmatrix} 0 & 4 & 1 & 9 \\ 3 & 0 & 6 & 11 \\ 4 & 1 & 0 & 2 \\ 6 & 5 & -4 & 0 \end{bmatrix}$$

TSP Dynamic Programming Source code

Graph Theory Video Series

Existence of Eulerian Paths and Circuits

William Fiset

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

We can find an Eulerian path on the following graph, but only if we start at specific nodes.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

We can find an Eulerian path on the following graph, but only if we start at specific nodes.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

We can find an Eulerian path on the following graph, but only if we start at specific nodes.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

We can find an Eulerian path on the following graph, but only if we start at specific nodes.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

We can find an Eulerian path on the following graph, but only if we start at specific nodes.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

We can find an Eulerian path on the following graph, but only if we start at specific nodes.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

Suppose we start another path but this time at a different node.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

Suppose we start another path but this time at a different node.

What is an Eulerian Path?

An **Eulerian Path** (or Eulerian Trail) is a path of edges that visits all the edges in a graph exactly once.

Choosing the wrong starting node can lead to having unreachable edges.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

If you know the graph contains an Eulerian cycle then you can start anywhere.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

If your graph does not contain an Eulerian cycle then you may not be able to return to the start node or you will not be able to visit all edges of the graph.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

If your graph does not contain an Eulerian cycle then you may not be able to return to the start node or you will not be able to visit all edges of the graph.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

If your graph does not contain an Eulerian cycle then you may not be able to return to the start node or you will not be able to visit all edges of the graph.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

If your graph does not contain an Eulerian cycle then you may not be able to return to the start node or you will not be able to visit all edges of the graph.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

If your graph does not contain an Eulerian cycle then you may not be able to return to the start node or you will not be able to visit all edges of the graph.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

If your graph does not contain an Eulerian cycle then you may not be able to return to the start node or you will not be able to visit all edges of the graph.

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

Oops, we're stuck and can't make it back to start node

What is an Eulerian circuit?

Similarly, an **Eulerian circuit** (or Eulerian cycle) is an Eulerian path which starts and ends on the same vertex.

There are also unvisited edges remaining

Oops, we're stuck and can't make it back to start node

What conditions are required for a valid Eulerian Path/Circuit?

That depends on what kind of graph you're dealing with. Altogether there are four flavors of the Euler path/circuit problem we care about:

	Eulerian Circuit	**Eulerian Path**
Undirected Graph	Every vertex has an even degree.	Either every vertex has even degree or exactly two vertices have odd degree.
Directed Graph	Every vertex has equal indegree and outdegree	At most one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

Node Degrees

Node Degrees

Undirected graph

Node degree = 3

The degree of a node is how many edges are attached to it.

Node Degrees

Undirected graph

Node degree = 3

The degree of a node is how many edges are attached to it.

Directed graph

In degree = 2
Out degree = 1

The indegree is the number of incoming edges and outdegree is number of outgoing edges.

What conditions are required for a valid Eulerian Path/Circuit?

That depends on what kind of graph you're dealing with. Altogether there are four flavors of Euler path/circuit problem we care about:

	Eulerian Circuit	**Eulerian Path**
Undirected Graph	Every vertex has an even degree.	Either every vertex has even degree or exactly two vertices have odd degree.
Directed Graph	Every vertex has equal indegree and outdegree	At most one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

What conditions are required for a valid Eulerian Path/Circuit?

That depends on what kind of graph you're dealing with. Altogether there are four flavors of Euler path/circuit problem we care about:

	Eulerian Circuit	Eulerian Path
Undirected Graph	Every vertex has an even degree.	Either every vertex has even degree or exactly two vertices have odd degree.
Directed Graph	Every vertex has equal indegree and outdegree	At most one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

What conditions are required for a valid Eulerian Path/Circuit?

That depends on what kind of graph you're dealing with. Altogether there are four flavors of Euler path/circuit problem we care about:

	Eulerian Circuit	**Eulerian Path**
Undirected Graph	Every vertex has an even degree.	Either every vertex has even degree or exactly two vertices have odd degree.
Directed Graph	Every vertex has equal indegree and outdegree	At most one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

What conditions are required for a valid Eulerian Path/Circuit?

That depends on what kind of graph you're dealing with. Altogether there are four flavors of Euler path/circuit problem we care about:

	Eulerian Circuit	**Eulerian Path**
Undirected Graph	Every vertex has an even degree.	Either every vertex has even degree or exactly two vertices have odd degree.
Directed Graph	Every vertex has equal indegree and outdegree	At most one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

What conditions are required for a valid Eulerian Path/Circuit?

That depends on what kind of graph you're dealing with. Altogether there are four flavors of Euler path/circuit problem we care about:

	Eulerian Circuit	Eulerian Path
Undirected Graph	Every vertex has an even degree.	Either every vertex has even degree or exactly two vertices have odd degree.
Directed Graph	Every vertex has equal indegree and outdegree	At most one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

Does this undirected graph have an Eulerian path/circuit?

Does this undirected graph have an Eulerian path/circuit?

No Eulerian path or circuit.
There are too many nodes that have an odd degree.

Does this undirected graph have an Eulerian path/circuit?

Does this undirected graph have an Eulerian path/circuit?

Only Eulerian path.

Does this undirected graph have an Eulerian path/circuit?

Does this undirected graph have an Eulerian path/circuit?

Yes! It has both an Eulerian path and circuit.

Does this undirected graph have an Eulerian path/circuit?

True or false: if a graph has an Eulerian circuit, it also has an Eulerian path.

Does this undirected graph have an Eulerian path/circuit?

True or false: if a graph has an Eulerian circuit, it also has an Eulerian path.

Does this undirected graph have an Eulerian path/circuit?

Does this undirected graph have an Eulerian path/circuit?

There are no Eulerian paths/circuits. An additional requirement when finding paths/circuits is that all vertices with nonzero degree need to belong to a single connected component.

Does this directed graph have an Eulerian path/circuit?

Does this directed graph have an Eulerian path/circuit?

Yes, it has both an Eulerian path and an Eulerian circuit because all in/out degrees are equal.

Does this directed graph have an Eulerian path/circuit?

Does this directed graph have an Eulerian path/circuit?

No path or circuit. The red nodes have either too many in coming or outgoing edges.

Does this directed graph have an Eulerian path/circuit?

Does this directed graph have an Eulerian path/circuit?

This graph has an Eulerian path, but no Eulerian circuit. It also has a unique start/end node for the path.

Does this directed graph have an Eulerian path/circuit?

Note that the singleton node has no incoming/ outgoing edges, so it doesn’t impact whether or not we have an Eulerian path.

Next Video: Eulerian path algorithm

Existence of Eulerian Paths and Circuits

Graph Theory Video Series

Finding Eulerian Paths and Circuits

William Fiset

Previous video:

Finding an Eulerian path (directed graph)

Finding an Eulerian path (directed graph)

Step 1 to finding an Eulerian path is determining if there even exists an Eulerian path.

Finding an Eulerian path (directed graph)

Step 1 to finding an Eulerian path is determining if there even exists an Eulerian path.

Recall that for an Eulerian path to exist at most one one vertex has (outdegree) – (indegree) = 1 and at most one vertex has (indegree) – (outdegree) = 1 and all other vertices have equal in and out degrees.

Finding an Eulerian path (directed graph)

Node	In	Out
0		
1		
2		
3		
4		
5		
6		

Count the in/out degrees of each node by looping through all the edges.

Finding an Eulerian path (directed graph)

Node	In	Out
0		
1		1
2	1	
3		
4		
5		
6		

Finding an Eulerian path (directed graph)

Node	In	Out
0		
1		2
2	1	
3	1	
4		
5		
6		

Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	1	
3	1	1
4		
5		
6		

Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	2	
3	1	2
4		
5		
6		

Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	2	1
3	1	2
4	1	
5		
6		

Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	3	2
3	1	2
4	1	
5		
6		

Finding an Eulerian path (directed graph)

Node	In	Out
0		
1	1	2
2	3	2
3	1	2
4	1	
5		
6		

And so on for all other edges…

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Once we've verified that no node has too many outgoing edges (out[i] – in[i] > 1) or incoming edges (in[i] – out[i] > 1) and there are just the right amount of start/end nodes we can be certain that an Eulerian path exists.

The next step is to find a valid starting node.

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Node 1 is the only node with exactly one extra outgoing edge, so it's our only valid start node. Similarly, node 6 is the only node with exactly one extra incoming edge, so it will end up being the end node.

out[1] - in[1] = 2 - 1 = 1

in[6] - out[6] = 2 - 1 = 1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

NOTE: If all in and out degrees are equal (Eulerian circuit case) then any node with non-zero degree would serve as a suitable starting node.

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Now that we know the starting node, let’s find an Eulerian path!

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Let’s see what happens if we do a naive DFS, trying to traverse as many edges as possible until we get stuck.

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

By randomly selecting edges during the DFS we made it from the start node to the end node.

However, we did not find an Eulerian path because we didn’t traverse all the edges in our graph!

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

The good news is we can modify our DFS to handle forcing the traversal of all edges :)

Finding an Eulerian path (directed graph)

To illustrate this, consider starting at node 0 and trying to find an Eulerian path.

Finding an Eulerian path (directed graph)

To illustrate this, consider starting at node 0 and trying to find an Eulerian path.

Finding an Eulerian path (directed graph)

To illustrate this, consider starting at node 0 and trying to find an Eulerian path.

Finding an Eulerian path (directed graph)

Whoops… we skipped the edges going to node 2 and back which need to be part of the solution.

Finding an Eulerian path (directed graph)

Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Finding an Eulerian path (directed graph)

Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [4]

Finding an Eulerian path (directed graph)

When backtracking, if the current node has any remaining unvisited edges (white edges) we follow any of them calling our DFS method recursively to extend the Eulerian path.

Solution: [3, 4]

Finding an Eulerian path (directed graph)

Solution: [3, 4]

Finding an Eulerian path (directed graph)

Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [3, 4]

Finding an Eulerian path (directed graph)

Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [1, 3, 4]

Finding an Eulerian path (directed graph)

Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [2, 1, 3, 4]

Finding an Eulerian path (directed graph)

Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [1, 2, 1, 3, 4]

Finding an Eulerian path (directed graph)

Once we get stuck (meaning the current node has no unvisited outgoing edges), we backtrack and add the current node to the solution.

Solution: [0, 1, 2, 1, 3, 4]

Finding an Eulerian path (directed graph)

Node	In	Out
0	0	0
1	1	2
2	3	3
3	3	3
4	2	2
5	1	1
6	2	1

Coming back to the previous example, let’s restart the algorithm, but this time track the number of unvisited edges we have left to take for each node.

Finding an Eulerian path (directed graph)

Node	[redacted]	Out
0	[redacted]	0
1	[redacted]	2
2	[redacted]	3
3	[redacted]	3
4	[redacted]	2
5	[redacted]	1
6	[redacted]	1

In fact, we have already computed the number of outgoing edges for each edge in the “out” array which we can reuse.

We won’t be needing the “in” array after we’ve validated that an Eulerian path exists.

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	2
2	3
3	3
4	2
5	1
6	1

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	2
2	3
3	3
4	2
5	1
6	1

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	3
3	3
4	2
5	1
6	1

Every time an edge is taken, reduce the outgoing edge count in the out array.

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	3
3	2
4	2
5	1
6	1

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	3
3	2
4	2
5	0
6	1

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	3
3	2
4	2
5	0
6	0

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	3
3	1
4	2
5	0
6	0

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	2
3	1
4	2
5	0
6	0

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	2
3	1
4	1
5	0
6	0

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	2
3	1
4	1
5	0
6	0

When the DFS is stuck, meaning there are no more outgoing edges (i.e out[i] = 0), then we know to backtrack and add the current node to the solution.

Solution = []

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	2
3	1
4	1
5	0
6	0

When backtracking, if the current node has any remaining unvisited edges (white edges), we follow any of them, calling our DFS method recursively to extend the Eulerian path. We can verify there still are outgoing edges by checking if out[i] != 0.

Solution = [6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	2
3	1
4	0
5	0
6	0

Solution = [6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	1
2	2
3	0
4	0
5	0
6	0

Solution = [6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	2
3	0
4	0
5	0
6	0

Solution = [6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	1
3	0
4	0
5	0
6	0

Solution = [6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	1
3	0
4	0
5	0
6	0

When the DFS is stuck, meaning there are no more outgoing edges (i.e out[i] = 0), then we know to backtrack and add the current node to the solution.

Solution = [6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	1
3	0
4	0
5	0
6	0

Node 2 still has an unvisited edge (since out[i] != 0) so we need to follow that edge.

Solution = [4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

When the DFS is stuck, meaning there are no more outgoing edges (i.e out[i] = 0), then we know to backtrack and add the current node to the solution.

Solution = [4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [6,3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [5,6,3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [3,5,6,3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Solution = [1,3,5,6,3,2,4,3,1,2,2,4,6]

Finding an Eulerian path (directed graph)

Node	Out
0	0
1	0
2	0
3	0
4	0
5	0
6	0

The time complexity to find an eulerian path with this algorithm is **O(E)**. The two calculations we're doing: computing in/out degrees + DFS are both linear in the number of edges.

Solution = [1,3,5,6,3,2,4,3,1,2,2,4,6]

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
    for edges in g:
      for edge in edges:
        out[edge.from]++
        in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
    for edges in g:
      for edge in edges:
        out[edge.from]++
        in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
function countInOutDegrees():
   for edges in g:
     for edge in edges:
       out[edge.from]++
       in[edge.to]++

function graphHasEulerianPath():
  start_nodes, end_nodes = 0, 0
  for (i = 0; i < n; i++):
    if (out[i] - in[i]) > 1 or (in[i] - out[i]) > 1:
      return false
    else if out[i] - in[i] == 1:
      start_nodes++
    else if in[i] - out[i] == 1:
      end_nodes++
  return (end_nodes == 0 and start_nodes == 0) or
         (end_nodes == 1 and start_nodes == 1)
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

Avoids starting DFS at a singleton

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

The out array is currently serving two purposes. One purpose is to track whether or not there are still outgoing edges, and the other is to index into the adjacency list to select the next outgoing edge.

This assumes the adjacency list stores edges in a data structure that is indexable in **O(1)** (e.g stored in an array). If not (e.g a linked-list/stack/etc…), you can use an iterator to iterate over the edges.

```
function dfs(at):
    # While the current node still has outgoing edges
    while (out[at] != 0):

        # Select the next unvisited outgoing edge
        next_edge = g[at].get(--out[at])
        dfs(next_edge.to)

    # Add current node to solution
    path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
function findStartNode():
  start = 0
  for (i = 0; i < n; i = i + 1):
    # Unique starting node
    if out[i] - in[i] == 1: return i

    # Start at any node with an outgoing edge
    if out[i] > 0: start = i
  return start

function dfs(at):
  # While the current node still has outgoing edges
  while (out[at] != 0):

    # Select the next unvisited outgoing edge
    next_edge = g[at].get(--out[at])
    dfs(next_edge.to)

  # Add current node to solution
  path.insertFirst(at)
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

```
# Global/class scope variables
n = number of vertices in the graph
m = number of edges in the graph
g = adjacency list representing directed graph

in  = [0, 0, …, 0, 0] # Length n
out = [0, 0, …, 0, 0] # Length n

path = empty integer linked list data structure

function findEulerianPath():

    countInOutDegrees()
    if not graphHasEulerianPath(): return null

    dfs(findStartNode())

    # Return eulerian path if we traversed all the
    # edges. The graph might be disconnected, in which
    # case it’s impossible to have an euler path.
    if path.size() == m+1: return path
    return null
```

Source Code Link

Implementation source code can be found at the following link:

github.com/williamfiset/algorithms

Link in the description:

Eulerian Paths and Circuits source code

William Fiset

Finding Eulerian Paths and Circuits

Graph Theory Video Series

Prim’s Minimum Spanning Tree Algorithm

(lazy version)

William Fiset

What is a Minimum Spanning Tree?

Given an **undirected graph** with weighted edges, a **Minimum Spanning Tree** (MST) is a subset of the edges in the graph which connects all vertices together (without creating any cycles) while minimizing the total edge cost.

What is a Minimum Spanning Tree?

Given an **undirected graph** with weighted edges, a **Minimum Spanning Tree** (MST) is a subset of the edges in the graph which connects all vertices together (without creating any cycles) while minimizing the total edge cost.

What is a Minimum Spanning Tree?

This particular graph has a unique MST highlighted in green. However, it is common for a graph to have multiple valid MSTs of equal costs.

MST cost: 0 + 5 + −2 + 2 + 1 + 3 = 9

Find any MST

Find any MST

MST cost: 3 + 1 + 2 + 0 + 8 = 14

Find any MST

Find any MST

MST cost: 3 + 1 + 2 + 4 + 7 + 8 + 9 + 5 = 39

Find any MST

Find any MST

This graph has no MST!

All nodes must be connected to form a spanning tree.

Prim’s MST Algorithm

Prim’s is a greedy MST algorithm that works well on **dense graphs**. On these graphs, Prim’s meets or improves on the time bounds of its popular rivals (Kruskal’s & Borůvka’s).

However, when it comes to finding the **minimum spanning forest** on a disconnected graph, Prim’s cannot do this as easily (the algorithm must be run on each connected component individually).

The **lazy version** of Prim’s has a runtime of **O(E*log(E))**, but the **eager version** (which we will also look at) has a better runtime of **O(E*log(V))**.

Lazy Prim’s MST Overview

Maintain a min Priority Queue (PQ) that sorts edges based on min edge cost. This will be used to determine the next node to visit and the edge used to get there.

Start the algorithm on any node **s**. Mark **s** as visited and iterate over all edges of **s**, adding them to the PQ.

While the PQ is not empty and a MST has not been formed, dequeue the next cheapest edge from the PQ. If the dequeued edge is outdated (meaning the node it points to has already been visited) then skip it and poll again. Otherwise, mark the current node as visited and add the selected edge to the MST.

Iterate over the new current node’s edges and add all its edges to the PQ. Do not add edges to the PQ which point to already visited nodes.

Lazy Prim's

Lazy Prim's

One thing to bear in mind is that although the graph above represents an **undirected graph**, the internal adjacency list representation typically has each undirected edge stored as **two directed edges**.

Lazy Prim's

The actual internal representation typically looks like this.

Lazy Prim's

Edges in PQ
(start, end, cost)

On the right we will be keeping track of the PQ containing the edge objects as triplets: (start node, end node, edge cost)

Lazy Prim's

Edges in PQ
(start, end, cost)

Unvisited Visiting Visited

Lazy Prim's

Edges in PQ
(start, end, cost)

Let's begin Prim's at node 0.

Lazy Prim's

Edges in PQ
(start, end, cost)

Iterate over all outgoing edges
and add them to the PQ.

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
(0, 2, 1)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
(0, 2, 1)
(0, 3, 4)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
(0, 2, 1)
(0, 3, 4)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
(0, 2, 1)
(0, 3, 4)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
(0, 2, 1)
(0, 3, 4)

Edge (0, 2, 1) has the lowest value in the PQ so it gets added to the MST. This also means that the next node we process is node 2.

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
→ ~~(0, 2, 1)~~
(0, 3, 4)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)

Next, iterate through all the edges of node 2 and add them to the PQ.

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
→ ~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
→ ~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
(2, 3, 2)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
(2, 3, 2)

Node 0 is already visited so we skip adding the edge (2, 0, 1) to the PQ.

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
(2, 3, 2)

The reason we don't include edges which point to already visited nodes is that either they overlap with an edge already part of the MST (as is the case with the edge on this slide) or it would introduce a **cycle** in the MST, if included.

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
→ ~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
(2, 3, 2)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
→ (2, 3, 2)

Edge (2, 3, 2) has the lowest value in the PQ so it gets added to the MST. This also means that the next node we process is node 3.

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
→ ~~(2, 3, 2)~~

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
→ ~~(2, 3, 2)~~

The same process of adding edges to the PQ and polling the smallest edge continues until the MST is complete.

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
→ ~~(2, 3, 2)~~
(3, 5, 2)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
→ ~~(2, 3, 2)~~
(3, 5, 2)
(3, 6, 7)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
→ ~~(2, 3, 2)~~
(3, 5, 2)
(3, 6, 7)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
→ ~~(2, 3, 2)~~
(3, 5, 2)
(3, 6, 7)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
→ (3, 5, 2)
(3, 6, 7)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
→ ~~(3, 5, 2)~~
(3, 6, 7)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
→ ~~(3, 5, 2)~~
(3, 6, 7)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
→ ~~(3, 5, 2)~~
(3, 6, 7)
(5, 4, 1)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
→ ~~(3, 5, 2)~~
(3, 6, 7)
(5, 4, 1)
(5, 7, 9)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
→ ~~(3, 5, 2)~~
(3, 6, 7)
(5, 4, 1)
(5, 7, 9)
(5, 6, 6)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
→ ~~(3, 5, 2)~~
(3, 6, 7)
(5, 4, 1)
(5, 7, 9)
(5, 6, 6)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
→ ~~(3, 5, 2)~~
(3, 6, 7)
(5, 4, 1)
(5, 7, 9)
(5, 6, 6)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
→ (5, 4, 1)
(5, 7, 9)
(5, 6, 6)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
→ ~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
→ ~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
(4, 1, 0)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
→ ~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
(4, 1, 0)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
→ ~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
(4, 1, 0)
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
→ ~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
(4, 1, 0)
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
→ ~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
(4, 1, 0)
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
→ (4, 1, 0)
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
→ ~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
→ ~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
→ ~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
→ ~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
(2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
→ ~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
(0, 3, 4)
→ (2, 1, 3)
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
~~(4, 1, 0)~~
(4, 7, 8)

The edge (2, 1, 3) is stale, poll again.

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
→ (0, 3, 4)
~~(2, 1, 3)~~
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
(5, 6, 6)
~~(4, 1, 0)~~
(4, 7, 8)

The edge (0, 3, 4) is also stale, poll again.

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
→ (5, 6, 6)
~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
→ ~~(5, 6, 6)~~
~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
→ ~~(5, 6, 6)~~
~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
→ ~~(5, 6, 6)~~
~~(4, 1, 0)~~
(4, 7, 8)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
→ ~~(5, 6, 6)~~
~~(4, 1, 0)~~
(4, 7, 8)
(6, 7, 12)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
→ ~~(5, 6, 6)~~
~~(4, 1, 0)~~
(4, 7, 8)
(6, 7, 12)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
(2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
(3, 6, 7)
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
(4, 7, 8)
(6, 7, 12)

The edge (3, 6, 7) is also stale, poll again.

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
→ (2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
(4, 7, 8)
(6, 7, 12)

There's a tie between (2, 5, 8) and (4, 7, 8) for which edge gets polled next. Since (2, 5, 8) was added first, let's assume it gets priority (it doesn't matter in practice).

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
→ (2, 5, 8)
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
(4, 7, 8)
(6, 7, 12)

The edge (2, 5, 8) is stale, poll again.

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
~~(2, 5, 8)~~
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
→ (4, 7, 8)
(6, 7, 12)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
~~(2, 5, 8)~~
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
→ ~~(4, 7, 8)~~
(6, 7, 12)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
~~(2, 5, 8)~~
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
→ ~~(4, 7, 8)~~
(6, 7, 12)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
~~(2, 5, 8)~~
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
→ ~~(4, 7, 8)~~
(6, 7, 12)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
~~(2, 5, 8)~~
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
→ ~~(4, 7, 8)~~
(6, 7, 12)

Lazy Prim's

Edges in PQ

(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
~~(2, 5, 8)~~
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
→ ~~(4, 7, 8)~~
(6, 7, 12)

Lazy Prim's

Edges in PQ
(start, end, cost)

(0, 1, 10)
~~(0, 2, 1)~~
~~(0, 3, 4)~~
~~(2, 1, 3)~~
~~(2, 5, 8)~~
~~(2, 3, 2)~~
~~(3, 5, 2)~~
~~(3, 6, 7)~~
~~(5, 4, 1)~~
(5, 7, 9)
~~(5, 6, 6)~~
~~(4, 1, 0)~~
→ ~~(4, 7, 8)~~
(6, 7, 12)

We can now stop Prim's since the MST is complete. We know the MST is complete because the number of edges in the tree is one less than the number of nodes in the graph (i.e. the definition of a tree).

Lazy Prim's

If we collapse the graph back into the undirected edge view it becomes clear which edges are included in the MST.

Lazy Prim's

Lazy Prim's

The MST cost is:

$1 + 2 + 2 + 6 + 1 + 0 + 8 = 20$

Lazy Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

pq = … # PQ data structure; stores edge objects consisting of
       # {start node, end node, edge cost} tuples. The PQ sorts
       # edges based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Lazy Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

pq = … # PQ data structure; stores edge objects consisting of
       # {start node, end node, edge cost} tuples. The PQ sorts
       # edges based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Lazy Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

pq = … # PQ data structure; stores edge objects consisting of
       # {start node, end node, edge cost} tuples. The PQ sorts
       # edges based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Lazy Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

pq = … # PQ data structure; stores edge objects consisting of
       # {start node, end node, edge cost} tuples. The PQ sorts
       # edges based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Lazy Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

pq = … # PQ data structure; stores edge objects consisting of
       # {start node, end node, edge cost} tuples. The PQ sorts
       # edges based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

Helper method to iterate over the edges of a node and add edges to the PQ:

```
function addEdges(nodeIndex):
  # Mark the current node as visited.
  visited[nodeIndex] = true

  # Iterate over all edges going outwards from the current node.
  # Add edges to the PQ which point to unvisited nodes.
  edges = g[nodeIndex]
  for (edge : edges):
    if !visited[edge.to]:
      pq.enqueue(edge)
```

Helper method to iterate over the edges of a node and add edges to the PQ:

```
function addEdges(nodeIndex):
  # Mark the current node as visited.
  visited[nodeIndex] = true

  # Iterate over all edges going outwards from the current node.
  # Add edges to the PQ which point to unvisited nodes.
  edges = g[nodeIndex]
  for (edge : edges):
    if !visited[edge.to]:
      pq.enqueue(edge)
```

Helper method to iterate over the edges of a node and add edges to the PQ:

```
function addEdges(nodeIndex):
  # Mark the current node as visited.
  visited[nodeIndex] = true

  # Iterate over all edges going outwards from the current node.
  # Add edges to the PQ which point to unvisited nodes.
  edges = g[nodeIndex]
  for (edge : edges):
    if !visited[edge.to]:
      pq.enqueue(edge)
```

Helper method to iterate over the edges of a node and add edges to the PQ:

```
function addEdges(nodeIndex):
  # Mark the current node as visited.
  visited[nodeIndex] = true

  # Iterate over all edges going outwards from the current node.
  # Add edges to the PQ which point to unvisited nodes.
  edges = g[nodeIndex]
  for (edge : edges):
    if !visited[edge.to]:
      pq.enqueue(edge)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

Skip edge that points to a visited node.

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function lazyPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m
  addEdges(s)

  while (!pq.isEmpty() and edgeCount != m):
    edge = pq.dequeue()
    nodeIndex = edge.to

    if visited[nodeIndex]:
      continue

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    addEdges(nodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

Next Video: Eager Prim’s

Prim’s Minimum Spanning Tree Algorithm (lazy version)

Graph Theory Video Series

Prim’s Minimum Spanning Tree Algorithm

(eager version)

William Fiset

Previous Video: Lazy Prim’s MST

<insert video clip>

Link to lazy Prim’s in the description

Eager Prim's

The lazy implementation of Prim's inserts up to E edges into the PQ. Therefore, each poll operation on the PQ is **O(log(E))**.

Instead of blindly inserting edges into a PQ which could later become stale, the eager version of Prim's tracks **(node, edge) key-value pairs** that can easily be **updated** and **polled** to determine the next best edge to add to the MST.

Eager Prim's

Key realization: for any MST with directed edges, each node is paired with **exactly one** of its incoming edges (except for the start node).

This can easily be seen on a directed MST where you can have multiple edges leaving a node, but at most one edge entering a node.

Let's take a closer look…

Eager Prim's

Original undirected graph.

Eager Prim's

Equivalent directed version.

Eager Prim's

MST starting from node 0.

Eager Prim's

Looking at the directed MST, you can see that each node is paired with an incoming edge except for the starting node.

Eager Prim's

Looking at the directed MST, you can see that each node is paired with an incoming edge except for the starting node.

Eager Prim’s

In the eager version of Prim’s we are trying to determine which of a node’s incoming edges we should select to include in the MST.

A slight difference from the lazy version is that instead of adding edges to the PQ as we iterate over the edges of node, we’re going to **relax** (update) the destination node’s most promising incoming edge.

Eager Prim's

A natural question to ask at this point is how are we going to efficiently update and retrieve these (node, edge) pairs?

One possible solution is to use an **Indexed Priority Queue (IPQ)** which can efficiently update and poll key-value pairs. This reduces the overall time complexity from **O(E*logE)** to **O(E*logV)** since there can only be V (node, edge) pairs in the IPQ, making the update and poll operations **O(logV)**.

Indexed Priority Queue DS Video

<insert video clip>

Link to IPQ video the description

Eager Prim's Algorithm

Maintain a **min Indexed Priority Queue (IPQ)** of size V that sorts vertex–edge pairs (v, e) based on the min edge cost of e. By default, all vertices v have a best value of (v, ∅) in the IPQ.

Start the algorithm on any node 's'. Mark **s** as visited and **relax** all edges of **s**.

While the IPQ is not empty and a MST has not been formed, dequeue the next best (v, e) pair from the IPQ. Mark node v as visited and add edge e to the MST.

Next, relax all edges of v while making sure not to relax any edge pointing to a node which has already been visited.

---

relaxing in this context refers to updating the entry for node v in the IPQ from (v, e_{old}) to (v, e_{new}) if the new edge e_{new} from u -> v has a lower cost than e_{old}.

9
0
1
0
5
−2
3
2
7
3
4
4
2
3
6
6
5
6
1

One thing to bear in mind is that while the graph above represents an **undirected graph**, the internal adjacency list representation typically has each undirected edge stored as **two directed edges**.

(node, edge) key-value pairs in IPQ

(node, edge) key-value pairs in IPQ

(node, edge) key-value pairs in IPQ

2 -> (0, 2, 0)

node index -> (start node, end node, edge cost)

(node, edge) key-value pairs in IPQ

2 -> (0, 2, 0)
5 -> (0, 5, 7)

(node, edge) key-value pairs in IPQ

```
2 -> (0, 2, 0)
5 -> (0, 5, 7)
3 -> (0, 3, 5)
```

(node, edge) key-value pairs in IPQ

```
2 -> (0, 2, 0)
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)
```

(node, edge) key-value pairs in IPQ

```
2 -> (0, 2, 0)
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)
```

(node, edge) key-value pairs in IPQ

2 -> (0, 2, 0)
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

The next cheapest (node, edge) pair is 2 -> (0, 2, 0).

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

Ignore edges pointing to already-visited nodes.

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (0, 5, 7)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

Edge (2, 5, 6) has a lower cost going to node 5 so update the IPQ with the new edge.

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

Edge (2, 5, 6) has a lower cost going to node 5 so update the IPQ with the new edge.

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
3 -> (0, 3, 5)
1 -> (0, 1, 9)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
~~3 -> (0, 3, 5)~~
1 -> (0, 1, 9)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
~~3 -> (0, 3, 5)~~
1 -> (0, 1, 9)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, -2)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (2, 5, 6)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)
6 -> (3, 6, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)
6 -> (3, 6, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)
6 -> (3, 6, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
1 -> (3, 1, −2)
6 -> (3, 6, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
5 -> (3, 5, 2)
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (3, 6, 3)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (5, 6, 1)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
6 -> (5, 6, 1)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, -2)~~
6 -> (5, 6, 1)
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, -2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, -2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, -2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, -2)~~
~~6 -> (5, 6, 1)~~
4 -> (1, 4, 3)

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
~~4 -> (1, 4, 3)~~

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
~~4 -> (1, 4, 3)~~

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, −2)~~
~~6 -> (5, 6, 1)~~
~~4 -> (1, 4, 3)~~

(node, edge) key-value pairs in IPQ

~~2 -> (0, 2, 0)~~
~~5 -> (3, 5, 2)~~
~~3 -> (0, 3, 5)~~
~~1 -> (3, 1, -2)~~
~~6 -> (5, 6, 1)~~
~~4 -> (1, 4, 3)~~

If we collapse the graph back into the undirected edge view it becomes clear which edges are included in the MST.

The MST cost is:

0 + 5 + 2 + 1 + −2 + 3 = 9

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

Graph edge density analysis

X-axis: Edge density percentage on graph with 5000 nodes.

Y-axis: time required to find MST in milliseconds.

Eager Prim's Pseudo Code

Let's define a few variables we'll need:

```
n = … # Number of nodes in the graph.

ipq = … # IPQ data structure; stores (node index, edge object)
        # pairs. The edge objects consist of {start node, end
        # node, edge cost} tuples. The IPQ sorts (node index,
        # edge object) pairs based on min edge cost.

g = … # Graph representing an adjacency list of weighted edges.
      # Each undirected edge is represented as two directed
      # edges in g. For especially dense graphs, prefer using
      # an adjacency matrix instead of an adjacency list to
      # improve performance.

visited = [false, …, false] # visited[i] tracks whether node i
                            # has been visited; size n
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

Helper method

```
function relaxEdgesAtNode(currentNodeIndex):
  # Mark the current node as visited.
  visited[currentNodeIndex] = true

  # Get all the edges going outwards from the current node.
  edges = g[currentNodeIndex]

  for (edge : edges):
    destNodeIndex = edge.to

    # Skip edges pointing to already visited nodes.
    if visited[destNodeIndex]:
      continue

    if !ipq.contains(destNodeIndex):
      # Insert edge for the first time.
      ipq.insert(destNodeIndex, edge)
    else:
      # Try and improve the cheapest edge at destNodeIndex with
      # the current edge in the IPQ.
      ipq.decreaseKey(destNodeIndex, edge)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

```
# s - the index of the starting node (0 ≤ s < n)
function eagerPrims(s = 0):
  m = n - 1 # number of edges in MST
  edgeCount, mstCost = 0, 0
  mstEdges = [null, …, null] # size m

  relaxEdgesAtNode(s)

  while (!ipq.isEmpty() and edgeCount != m):
    # Extract the next best (node index, edge object)
    # pair from the IPQ
    destNodeIndex, edge = ipq.dequeue()

    mstEdges[edgeCount++] = edge
    mstCost += edge.cost

    relaxEdgesAtNode(destNodeIndex)

  if edgeCount != m:
    return (null, null) # No MST exists!

  return (mstCost, mstEdges)
```

Next Video: eager Prim’s source code

Prim's Minimum Spanning Tree

Looking at the directed MST, you can see that each node is paired with an incoming edge except for the starting node.

Prim's MST Source Code

William Fiset

Graph Theory Video Series

BRONX
MANHATTAN
Central Park
QUEENS
BROOKLYN
The Narrows
HUDSON RIVER
East River
Harlem River
Jamaican Bay
Rockaway Inlet
ATLANTIC OCEAN
Red Express
Grey Local
Blue No stop
Orange Local
Orange Express
Grey
Red
Blue
Local
Local stop
Transfer
Express and local stop
Local stop
Rush hour stop
WASHINGTON HEIGHTS 207 St
Dyckman 200 St
190 St
181 St
175 St
168 St BROADWAY
157 St
145 St
City College 137 St
125 St
Columbia U. 116 St
Cathedral Pkwy 110 St
103 St
96 St
86 St
79 St
72 St
86 St - Lincoln Center
59 St - Columbus Circle
50 St
42 St
34 St - Penn Station
23 St
14 St - 8 av
PORT AUTHORITY BUS TERMINAL
WOODLAWN
242 ST VAN CORTLANDT PARK
Bedford Park Blvd - Lehman College
Kingsbridge Rd
Fordham Rd
Burnside
183 St
176 St Mt Eden
170 St
167 St
161 St Yankee Stadium
148 St LENOX TERMINAL
145 St
135 St
125 St
116 St Cathedral Pkwy - 110 St
103 St
96 St
86 St
81 St - Museum of Natural History
72 St
68 St
57 St
59 St
50 St
49 St
42 St - TIMES SQ
34 St - Penn Sta
28 St
23 St
18 St
14 St
8 St
Christopher St Sheridan Sq
W 4 St Washington Sq
Houston
Canal
Franklin
Chambers
Park Place
WORLD TRADE CENTER
Cortlandt
Rector
SOUTH FERRY
BOWLING GREEN
Wall St
Broadway - Nassau
Fulton
BROAD ST
BEDFORD PARK BLVD
205 ST
Kingsbridge Rd
Fordham Rd
182 - 183 Sts
Tremont
174 - 175
170
167 St
149 St 3 av
138 St
Grand Concourse
E 180 St
Morris Park
Pelham Pkwy
Gun Hill Rd
Baychester
241 ST
DYRE AVENUE
PELHAM BAY PARK
E Tremont - West Farms Sq
174 St
Freeman
Simpson
Intervale - 163 St
Prospect
Jackson
Zerega
Castle Hill
Westchester Sq
Middletown Rd
Buhre
PARKCHESTER
E 177 St
St Lawrence
Morrison - Sound View
Elder
Whitlock
Hunts Pt
Longwood
E 149 St
E 143 St - St Mary's
Cypress
Brook
138 St - 3 av
DITMARS
ASTORIA
Astoria Blvd - Hoyt
30 av Grand
Broadway
36 av Washington
39 av Beebe
Queensboro Plaza
Lexington av
TRAMWAY
Lexington - 3 avs
42 ST GRAND CENTRAL
Steinway
46 St
Northern Blvd
65 St
36 St
Queens Plaza
Court Sq
21 St Van Alst
Greenpoint
Nassau
45 Rd - Court House
Hunters Pt
Vernon - Jackson
Bedford
Bleeker (Down Town)
Broadway
Lafayette
Astor Pl Bleecker
Delancey Essex
Marcy Hewes
Broadway Flushing
Lorimer Flushing
Knickerbocker Central
Myrtle - Willoughby
Bedford - Nostrand
Classon
Clinton - Washington
Fulton
Myrtle Kosciuszko
Gates
Halsey Chauncey
Rockaway
Ralph
Utica
Kingston - Throop
Nostrand
Junius
Rockaway
Saratoga
Sutter
Rutland Rd
Utica
Kingston
Nostrand
Atlantic Sutter
Broadway Eastern Pkwy
Broadway E N.Y.
Van Siclen Pennsylvania
Livonia New Lots E 105 ST
ROCKAWAY PKWY
NEW LOTS
Broad Channel
Beach 25 St
Beach 36 St
Beach 44 St
Beach 60 St
Beach 67 St
Beach 90 St
Beach 98 St
Beach 105 St
ROCKAWAY PARK BEACH 116 ST
FAR ROCKAWAY MOTT AV
Rockaway Blvd
88 St Boyd
80 St Hudson
Grant
EUCLID AV
Shepherd
Van Siclen
Liberty
Aqueduct Racetrack
Aqueduct
North Conduit
JFK AIRPORT
HOWARD BEACH
LA GUARDIA
MAIN ST FLUSHING
82 St - Jackson Hts
90 St Elmhurst
Junction Blvd
103 St Corona Pl
111 St
Willets Pt Shea Stadium
71 - CONTINENTAL AVS FOREST HILLS
179 St JAMAICA
169 St
Parsons Blvd
Sutphin Blvd
Van Wyck Blvd
121 ST
Union Turnpike
Kew Gardens
75 av
Elmhurst
Grand
Newtown
Woodhaven - Queens Mall
63 Dr Rego Pk
67 av
74 St Broadway
Roosevelt - Jackson Hts
69 St Fisk
61 St Woodside
53 St Lincoln
46 St Bliss
40 St Lowery
33 St Rawson
111 St
102 St
Woodhaven Blvd
Forest Pkwy
Elderts Lane
Cypress Hills
Crescent
Norwood
Cleveland
Van Siclen
Alabama
LEFFERTS BLVD
111 St Greenwood
104 St Oxford
METROPOLITAN AV
Fresh Pond Rd
Forest
Seneca
FLATBUSH BKLYN COLLEGE
Parkside
Church
Beverley Rd
Cortelyou Rd
Newkirk
Av H
Av J
Av M
Kings Hwy
Av U
Neck Rd
Sheepshead B
BRIGHTON BEACH
Ocean Pkwy
W 8 St N.Y. Aquarium
Ft Hamilton Pkwy
50 St
55 St
New Utrecht - 62 ST
71 St
79 St
18 av
20 av
BAY PARKWAY
25 av
Bay 50 St
95 STREET FT HAMILTON
STILLWELL AV CONEY ISLAND
Prospect 25 St
36 St
45 St
53 St
59 St
Bay Ridge
77 St
86 St
9 St
4 av
Union
Bergen
Carroll
SMITH - 9 STS
Clark
Borough Hall
Hoyt St - Fulton Mall
Nevins
Atlantic
Court
Lawrence
Dekalb
Pacific
Hoyt - Schermerhorn
Lafayette
Clinton - Washington
7 av
Grand Army Plaza
Franklin
Eastern Pkwy - BKLYN Museum
Park Slope
FRANKLIN
Dean
Park Place
Botanic Garden
PROSPECT PARK
President
Sterling
Winthrop
Church
Beverley Rd
Newkirk
BKLYN BRIDGE - CITY HALL
CHAMBERS
Jay
Borough Hall
GRAND
2 av
E Broadway
Bowery Canal
Prince
Canal
City Hall
Spring
York
Broadway - Nassau
High St Bklyn Bridge
Wall St
Cortlandt
Rector
Whitehall - South Ferry
HOW TO FIND THE CROSS STREET NEAREST AN AVENUE ADDRESS IN MANHATTAN
1. - Cancel the last figure to the address
2. - Divide by 2
3. - Add or substract the code number given here
EXAMPLE: 625 MADISON AVENUE
Cancel the last figure: 5 of 625 — 62. Divide 62 by 2 — 31.
Then use the CODE N° of MADISON (+ 26) 31 (+26) — 57.
Result: 625 MADISON AVENUE is in the vicinty of 57 STREET.
AVENUES | CODE N°
A B C D | (+)3
1 2 4 | (+)3
3 | (+)10
5 up 200 | (+)13
5 up 400 | (+)16
5 up 600 | (+)18
5 up 775 | (+)20
5 from 775 to 1286 cancel last figure and | (−)18
5 up 1500 | (+)45
AVENUES | CODE N°
5 above 2000 OF AMERICAS | (+)24
7 | (−)12
7 above 110 | (−)12
8 | (+)20
9 | (+)10
10 | (+)13
AMSTERDAM | (+)14
AUDUBON | (+)60
BROADWAY (23 to 192) | (+)165
( )30
AVENUES | CODE N°
COLOMBUS | (+)60
CONVENT | (+)127
CENTRAL PARK WEST Divide by 10 the address N° then | (+)60
EDGECOMBE | (+)134
FT. WASHINGTON | (+)158
LENOX | (+)110
LEXINGTON | (+)22
MADISON | (+)26
MANHATTAN | (+)100
AVENUES | CODE N°
PARK | (+)35
PLEASANT | (+)101
RIVERSIDE DRIVE Divide by 10 the address N° then up to 165 St | (+)72
ST NICHOLAS | (+)110
WADSWORTH | (+)173
WEST END | (+)60
In MANHATTAN
Uptown is North
Downtown is South
Crosstown is East or West
Divided by 5 Avenue
Note: The scale of the city has been distorted. This is to give more space.
© COPYRIGHT 1988 THE REPRODUCTION OF THIS MAP IS FORBIDDEN