CEGE0096: Geospatial Programming

Lecture 8: Tree and Graph Representations

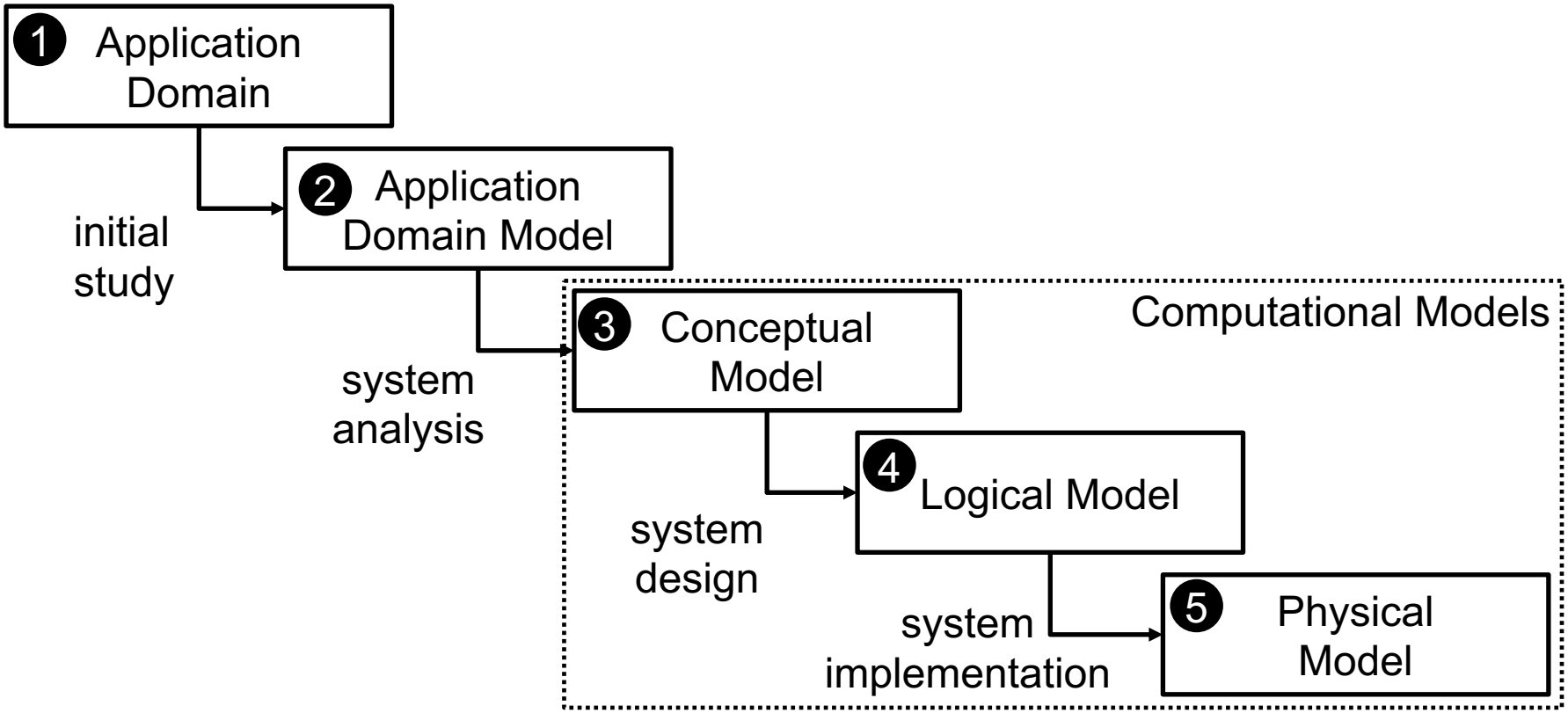
November 25, 2019

Aldo Lipani

Python Packages (So Far)

- NumPy
- Pandas
- PyProj
- Shapely
- GeoPandas
- Rasterio

The Modelling Process – Waterfall Model



Vector and Raster Representations

Geospatial phenomena are modelled as either fields or features.

Fields and Features are modelled in a Geometric Space as Geometric entities.

Geometric entities (e.g. vector polygons or raster pixels) have a permanent spatial relationship.

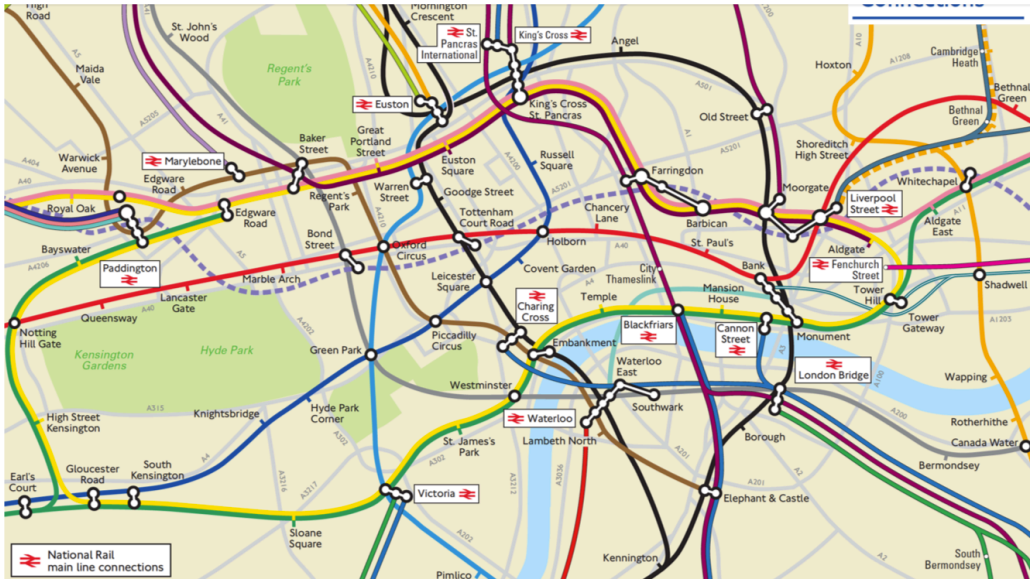
We exploit this spatial relationship using spatial tools such as buffering, intersection, overlay to solve problems.

Vector and Raster Representations (II)

However, these representations do not allow the spatial indexing of entities or to reason about how entities are connected to each other.

Which tube-stations are at 5 minutes walk from my current position?

How do I get from Euston Station to Waterloo Station?



TREE REPRESENTATIONS

Tree Representation

A tree is a widely used data structure that simulates a hierarchical structure, with a root value and subtrees of children with a parent node represented as a set of linked nodes.

The **root** is the top node of the tree (like 1).

A **child** is a node directly connected to another node when moving away from the root (like 2 to 1).

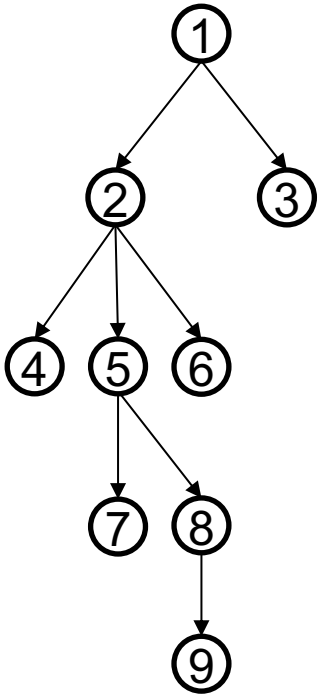
A **parent** is the opposite of a child (like 1 to 2).

Siblings are nodes with the same parent (like 4, 5 and 6).

Descendant is a node reachable by repeated proceeding from parent to child (like 9 to 2).

Ancestor is a node reachable by repeated proceeding form child to parent (like 2 to 9).

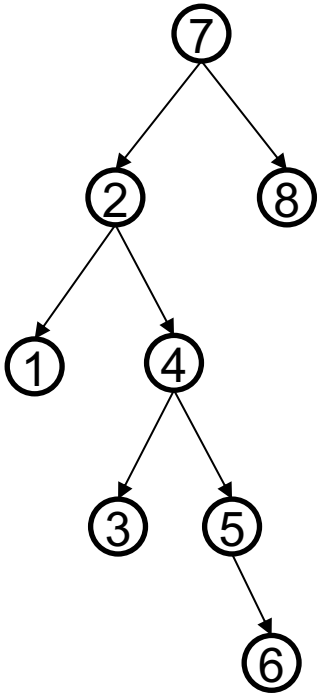
Leaf is a node without children (like 4, 7, 9 and 3).



Binary Trees

A binary tree is a special tree where every node can only have two children.

This is commonly used as a binary search trees (**BST**), where for each node, a node's left sub-tree must have values less than the node value and a node's right sub-tree must have values greater than the node value.



BSTs (when balanced like **B-Trees**) allow a quicker search than storing values in a list. This because at every node (decision point) we only need to check one of the two subtrees.

Basic Operations

Insert an element in a tree or create a tree if empty.

Search an element in a tree.

Pre-order traversal traverse of a tree: the root node is visited first, then the left subtree is traversed, and the right subtree is traversed.

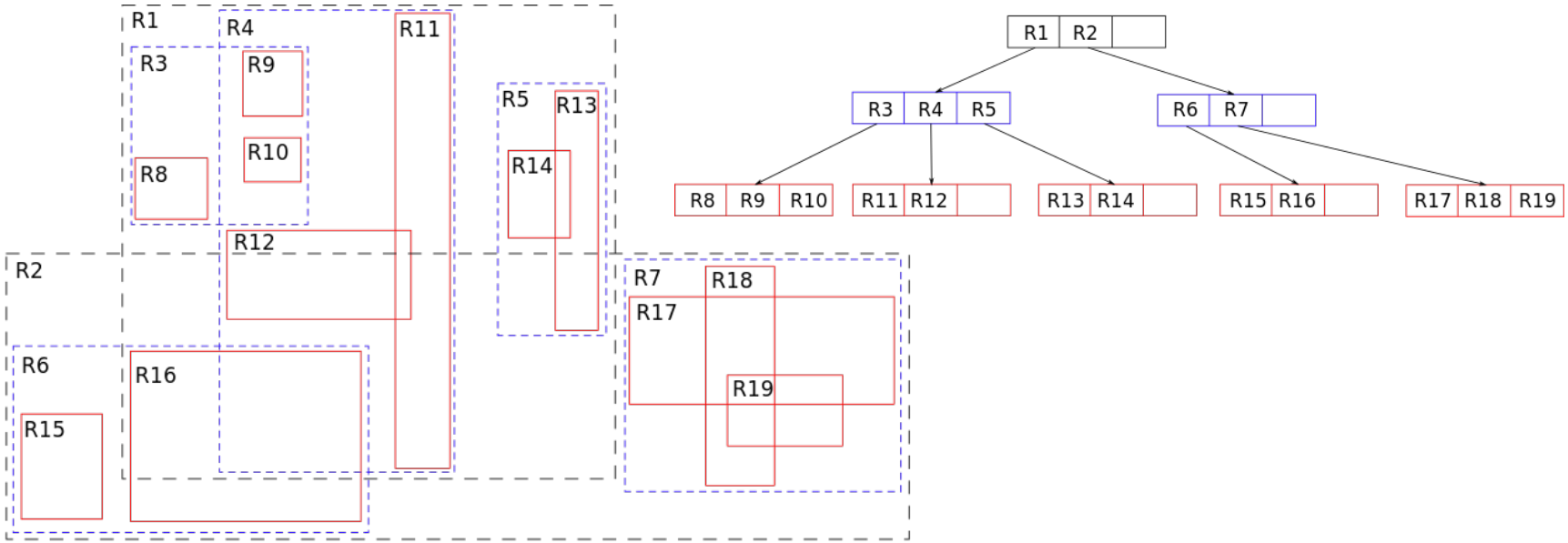
In-order traversal of a tree: the left subtree is traversed, then the root node is visited, and the right subtree is traversed.

Post-order traversal of a tree: the left subtree is traversed, then the right subtree is traversed, and the root node is visited.

R-Trees

R-trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons.

The key idea of the data structure is to group nearby objects and represent them within their minimum bounding rectangle.



Spatial Indexing

Indexing provides a method for quickly and efficiently finding objects in a data structure.

A spatial index provides a way for finding objects in a spatial data structure.

This is commonly done with R-trees.

Spatial Indexing with R-Trees

R-Tree indexing works by storing minimum bounding rectangle information in a spatial index.

A search can then be performed on a spatial index using a query rectangle in the form:

```
(min_x, min_y, max_x, max_y)
```

The index will then return a list of objects in the spatial index that intersect the query rectangle.

RTREE

Rtree

Rtree is a Python package that provides a number of advanced spatial indexing features.

These include:

- Multi-dimensional indexes
- Intersection search
- Nearest neighbour search

To use this library we need to import the index object of rtree:

```
from rtree import index
```

Documentation available here: https://rtree.readthedocs.io/en/latest/

How to Use Rtree?

Let's first create an index:

```
idx = Index.Index()
```

Insert n-rectangles using the insert method of index:

```
idx.insert(1, (a_x_min, a_y_min, b_x_max, b_y_max), "A")
```

We define a query as a recangle:

```
q = (x_min, y_min, x_max, y_max)
```

To query the index we use the intersection method of index:

```
idx.intersection(q, objects = True)
```

Point Spatial Index

As well as rectangles we can add a mix of points and rectangles to the spatial index:

```
points = [(0, 1), (2, 3), (4,-1)]

for n, point in enumerate(points):
   idx.insert(n, point, str(n))
```

Define a query:

```
q = (q_x, q_y)
```

We can then perform a nearest neighbor search:

```
idx.nearest(q, num_results=1, objects=True)
```

GRAPH REPRESENTATIONS

Network Modelling

In some situations we are interested in interconnections.

Can I change from the Victoria Line to the Metropolitan Line at Warren Street?

Geospatial relationships do not communicate the information that we are interested in.

Graphs store connectivity information.

Network Modelling (II)

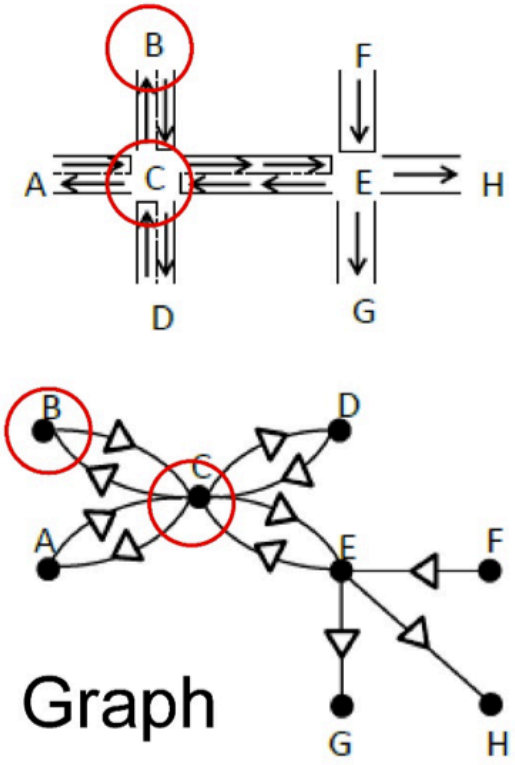
Graph Theory

Graph Theory is a field of mathematics that studies the behaviour of networks.

Graphs consists of nodes and edges.

Graphs can be used to model any kind of real world network.

Graph Representations



	A	B	C	D	E	F	G	H
A	-	-	1	-	-	-	-	-
B	-	-	1	-	-	-	-	-
C	1	1	-	1	1	-	-	-
D	-	-	1	-	-	-	-	-
E	-	-	1	-	-	-	1	1
F	-	-	-	-	1	-	-	-
G	-	-	-	-	-	-	-	-
H	-	-	-	-	-	-	-	-

Adjacency Matrix

Network Modelling (III)

In Transport Networks:

Junctions are modelled by **nodes**.

Roads are modelled by **edges**.

Topological Adjacency

In Vector Modelling, lines and polygons are modelled as stand-alone entities (“spaghetti” representation).

We can use topological networks to model polygons:

- **Vertices** are modelled by **nodes**;
- **Lines** are modelled by **edges**.

Another topological network can be:

- **Polygons** are modelled by **nodes**;
- **Boundaries** are modelled by **edges**.

Examples of Modelling with Networks

Public Transport Network:

- **Underground stations** are modeled by **nodes**;
- **Railway lines** are modelled by **edges**.

Social Networks:

- **People** are modelled by **nodes**;
- **Relationships** are modelled by **edges**.

GRAPH TYPES

Graph

In a simple graph:

- Nodes are unique;
- Edges are undirected;
- Edges are exclusively indexed by an undirected node tuple pair;
- Tuple represents rows/columns in an adjacency matrix.

Only one edge can exist between two nodes:

- Transport Network: traffic is two way;
- Social Network: We must be friends with each other.

Directed Graph (DiGraph)

In a directed graph:

- Nodes are unique;
- Edges are directed;
- Edges are exclusively indexed by a directed node tuple pair;
- Tuple represents rows/columns in adjacency matrix.

One or two edges can exist between two nodes:

- Transport Network: Traffic can either be two-way or one-way only.
- Social Network: I can be your follower but you don't need to follow me.

MultiGraph

In a multigraph:

- Nodes are unique;
- Edges are directed;
- Edges are not exclusively indexed by node pairs, a unique key is required;
- We can no longer use adjacency matrices (we need a bunch).

Transport Network: two nodes could be connected by a motorway and a local road.

Social Network: we can be friends on Facebook and be part of each other family members on Facebook.

MultiDiGraph

In a MultiDiGraph:

- Nodes are unique;
- Edges are directed;
- Edges are not exclusively indexed by node pairs, a unique key is required;
- We can no longer use adjacency matrices (we need a bunch).

Transport Network: Two nodes connected by a motorway and a one-way street.

Social Network: two people can be friends on Facebook and one can be among the closest friends of the other, but not vice versa.

Graph Attributes

Attributes can be linked to **edges** and **nodes**:

- Coordinate of the node;
- Length of the road;
- Geometry of road (e.g. LineString);
- Classification of the road (e.g. Motorway);
- Color and line thickness for visualization.

Weighted Graphs

Each graph edge can have a weight associated to it.

Weight represents impedance through graph:

- E.g. edge length in a transport network

But what happens if we have different speed limits in transport network?

- We can compute the total travel time by weighting each edge by a value equal to edge length divided by speed and then summing them up.

Shortest Path Algorithm

There are many algorithms for calculating shortest distance through a graph network.

Different algorithms are suited for different graphs and have different computational time/memory efficiency.

A shortest path algorithm is designed to minimize the cost of the path from a source node to a destination node.

Dijkstra's Shortest Path Algorithm is one the most commonly-used.

Dijkstra's Algorithm

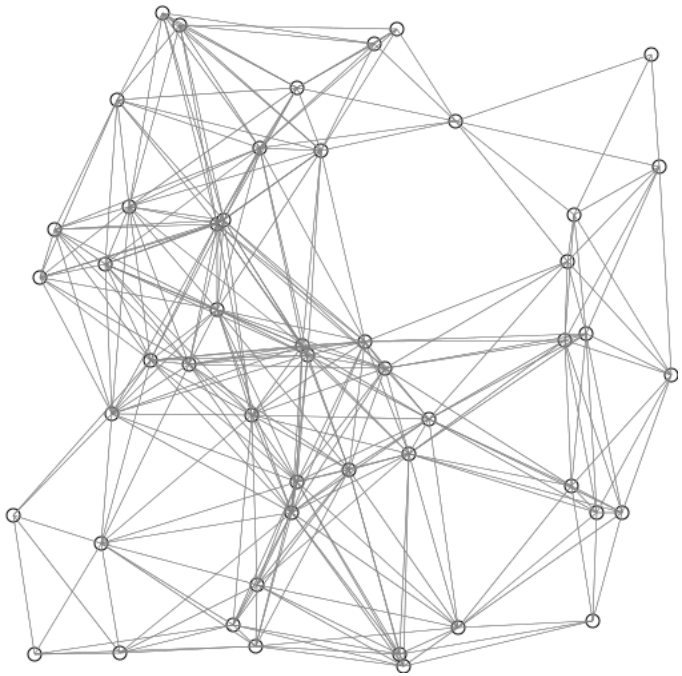
Dijkstra's algorithm is a greedy algorithm for solving single source shortest path that provides us with the shortest path from one particular source node to all other nodes in the given graph.

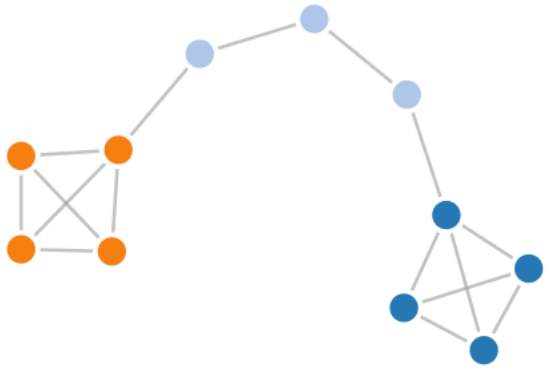
It can be applied on both directed and undirected graph.

It is applied on weighted graph.

For the algorithm to be applicable the source and destination nodes must be connected.

An Animated Example





NETWORKX

NetworkX

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics and functions of complex networks.

This package is usually imported as:

```
import networkx as nx
```

Features:

- Supports graphs, digraphs and multigraphs;
- Many standard graph algorithms (like Dijkstra);
- Edges can hold arbitrary data (e.g. weights, time-series).

Documentation available here: https://networkx.github.io/

Graph

To create a graph object:

```
g = nx.Graph()
```

To add nodes:

```
g.add_node(1, attribute="red")
```

To get the nodes:

```
g.node
```

To add an edge (note that we don't need to add nodes first):

```
g.add_edge(1, 2, attribute="red")
```

To get the edges:

```
graph.edges
```

Example

Let's make a simple 3x3 Manhattan road network:

```
g = nx.Graph()
w, h = 3, 3
```

We label our nodes in accordance with the formula defined by this function:

```
def get_id(r, c):
    return r + c * w
```

We now add the nodes to the graph:

```
for r in range(h):
    for c in range(w):
        g.add_node(get_id(r, c))
        print(get_id(r, c))
```

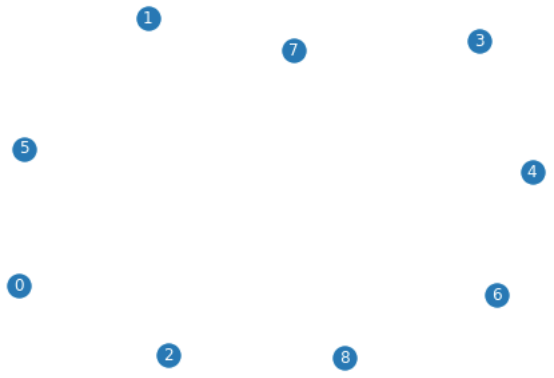
Example (II)

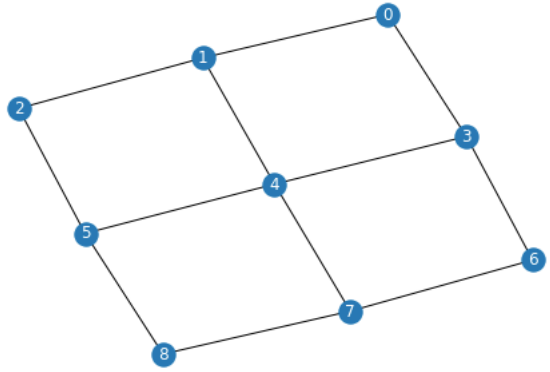
We can draw the graph by:

```
nx.draw(g, font_color='white', with_labels=True)
```

To construct the edges:

```
for r in range(h):
    for c in range(w):
        if c < w - 1:
            g.add_edge(get_id(r, c), get_id(r, c+1), length = 1.0)
        if r < h - 1:
            g.add_edge(get_id(r, c), get_id(r+1, c), length = 1.0)
```





Example (III)

We can use Dijkstra to find the shortest path between to nodes:

```
nx.dijkstra_path(g, source=3, target=8, weight="length")

# [3, 6, 7, 8]
```

We can transform the graph to a DiGraph:

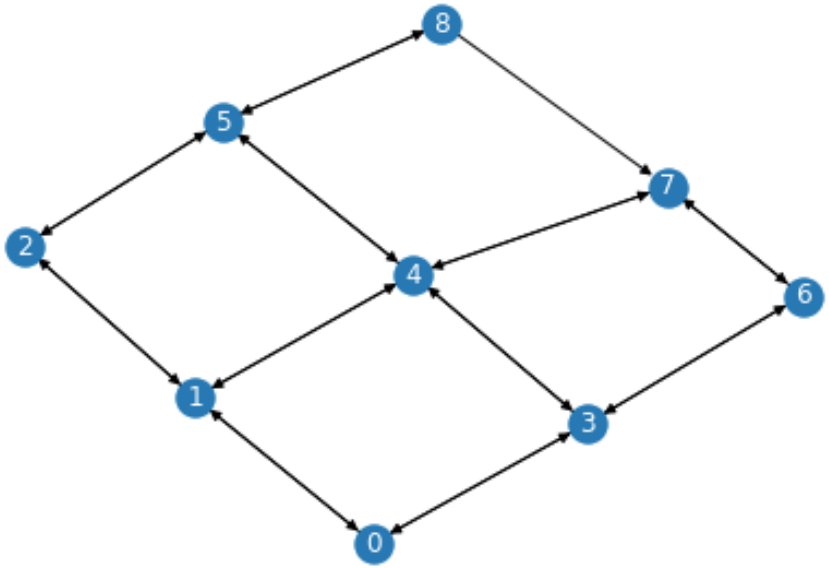
```
g = g.to_directed()
```

Let's now remove one edge:

```
g.remove_edge(7, 8)
```

And, perform Dijkstra again:

```
# [3, 4, 5, 8]
```



See you on Thursday for the Practical

Christopher Ingold Building G20

Do not forget to take your personal laptop with you.