# Lab 8 – Grrrraphs

**Instructors:** Lorenzo De Carli & Maan Khedr

Slides by Lorenzo De Carli

# What we are going to focus on today

- Building and processing graph data structures

# Exercise #1: a Graph class [1.5 pts]

- Python lacks a built-in graph abstraction (although there are some nice external modules – but we won't be using those here)
- In this exercise, you will build a simple graph class together with code to enable importing graphs from text files

# Exercise #1 /2

- Define a `Graph` class with the following characteristics:
  - a. Can be used to describe an undirected graph
  - b. Internally stores edges as an **adjacency list** for each node

1. Implements the following methods: **[0.5 pts]**
   - `addNode(data)`: creates a new graph node internally storing the string passed as parameter. Returns a GraphNode object
   - `removeNode(node)`: removes the node
   - `addEdge(n1, n2, weight)`: creates an edge between nodes `n1` and `n2`
   - `removeEdge(n1, n2)`: removes the edge between nodes `n1` and `n2`

# Exercise #1 /3

2. Additionally, implement the following methods: **[1 pts]**
   - `importFromFile(file):` imports a graph description from a GraphViz file. GraphViz files define a simple format for graph description. You will not need to implement all the features of GraphViz, only basic ones described below. The method clears all existing nodes and edges, and replaces them with those listed in the file.

# Graphviz format specifications

- Undirected graphs are represented as in the following example:

```
strict graph G {
    node1 -- node2 [weight=5];
    node2 -- node3;
    node4 -- node3 [weight=6];
}
```

- Where:
  - “strict graph G” signifies that the graph is undirected, and not a multigraph (i.e., there can be at most 1 edge between any given pair of nodes).
  - The content between curly braces consist of a list of edges, one per line, stating the name of a node, the token “--“, and the name of a second node.
  - Edge definitions may be followed by a comma-separated list of attributes between square brackets; the only attribute your loader needs to support is “weight”
  - If edge weight is not specified, it must be implicitly assumed to be 1.
  - **If the file contains semantic errors (e.g., a graph type which is not undirected), or syntactic/lexical errors (e.g., unexpected characters), the method should return null.**

# Exercise #1 - What to deliver

- Submit a file named ex1.py with your answer to questions 1 and 2

# Exercise #2 – Shortest Paths [1.5 pts]

- In the lecture, we have discussed Dijkstra's algorithm for computing shortest paths in a graph.
- One of the core step of the algorithms, at each iteration, is the identification of the node with the shortest path from the source:

```
V <- vertex toBeChecked with smallest currDist // Extract MIN from Q
```

- But… how quickly can we find the node with the smallest distance?
- In other words, how is our queue implemented? The question is important because it affects **efficiency**

# Exercise #2 /2

1. List two possible ways to implement this queue, with different efficiency (a slow one which uses linear search, and something faster) **[0.2 pts]**
2. Implement two version of the algorithm, one with the inefficient node selection logic, and one with the efficient node selection logic. Both should be based on the `Graph` class created in the course of Exercise 1. Implement the two algorithms as two methods named `slowSP(node)` and `fastSP(node)` **[1 pt]**

# Exercise #2 /3

3. Measure the performance of each algorithm on the sample graph provided on the lab's D2L (`random.dot`). **[0.2 pts]**
   - Time the execution of the algorithm, for all nodes
   - Report average, max and min time
4. Plot a histogram of the distribution of execution times across all nodes, and discuss the results **[0.1 pts]**

# Exercise #2 - What to deliver

- Submit a file named ex2.py with your answer to questions 1, 2, 3 and 4
- Your code should also contain comments to the results (question 4) as comments in the code

# Exercise #3 – Good ‘ol Kruskal [1.5 pts]

- In the lectures, we discussed Kruskal’s algorithm for the creation of minimum spanning trees
- In class we have mentioned the role of the UNION-FIND cycle detection algorithm for undirected graphs. We have not seen an implementation, but you can familiarize yourself with that using online resources:
  - https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
  - https://www.programiz.com/dsa/kruskal-algorithm
  - …and others (use whatever you prefer)

# Exercise #3 /2

1. Explain what a minimum spanning tree is, with an example (a simple graph with 5 to 10 nodes is sufficient) **[0.3 pts]**
   - The example must include two figures (full graph; spanning tree)
2. Implement the UNION-FIND cycle detection algorithm for undirected graphs. **[0.6 pts]**
3. Use your implementation of UNION-FIND to implement the full Kruskal algorithm as discussed in class. Implement Kruskal as a method called `mst()` as part of your `Graph` class. The method should return a `Graph` object describing the spanning tree. **[0.6 pts]**

# Exercise #3 - What to deliver

- Submit a file named ex3.pdf with your answers to questions 1 (you can use any editor of your choice – Word, Google Docs, Overleaf/LaTex, etc. as long as the output is PDF)
- Submit a file named ex3.py with your answer to questions 2 and 3

# Exercise #4 – Representations [1.5 pts]

- So far we have used an adjacency list as representation
- What about adjacency matrix?

# Exercise #4 /2

1. Starting from the Graph class created in Exercise 1, create a Graph2 class which uses (you guessed it) an **adjacency matrix** to keep track of the edges **[0.9 pts]**
   - Each location in the matrix must be 0 if an edge is not present, and W (where W is the edge's weight) if an edge is present
2. Extend both `Graph` and `Graph2` to implement DFS traversal. Call the method `dfs()` in both classes. The method should return a list of node in DFS order. **[0.3 pts]**
3. Measure the performance of `dfs()` on the example graph from D2L (`random.dot`) **[0.3 pts]**
   - Repeat the execution of `dfs()` ten times for each implementation, and report maximum, minimum and average time
   - Discuss the results: which implementation is faster? Why?

# Exercise #4 - What to deliver

- Submit a file named ex4.py with your answer to questions 1, 2 and 3
- The file should also contain comments to the results (question 3) as comments in the source code

# Exercise #5 - Topological what? [1 pt]

- In class we discussed the notion of **topological order** – which requires generating a list of nodes in a directed acyclic graph (DAG)
- The goal is to ensure that each node is always listed after its predecessors
  - We can define predecessors only because the graph is acyclic!

# Exercise #5 /2

1. Topological sorting can be implemented using an algorithm seen in class. Which algorithm? Why? **[0.2 pts]**
2. Extend your `Graph` class (exercise 1) with an `isdag()` method that returns true only if a graph does not contain cycles **[0.5 pts]**
3. Extend the same class with a `toposort()` method **[0.3 pts]**
    1. Checks if the graph is a DAG
    2. If yes, returns a list of nodes in topological order. Return None otherwise.

# Exercise #5 - What to deliver

- Submit a file named ex5.py with your answer to questions 1, 2 and 3
- The answer to question 1 should be given as comments in the code

# How to submit

- Upload a zip file to the “Lab 8” dropbox on D2L, containing the required content for every exercise

# Grading rubric

- You get **3 pts** for uploading a **partial solution** by **end of lab**
  - **Must not be an empty file or irrelevant material**
- Then, you'll have until **11:59PM of the day before the next lab** to upload the **complete solution**. That will be graded as follows:
  - Exercises 1, 2, 3, 4: 1.5 pts each
  - Exercises 5: 1 pt
  - **Can upload the complete solution to the same dropbox**

# That’s all folks!