**CS5002 Final Project Report:** Optimizing project management using Graph Theory

# **Introduction**

* Context and understanding of the chosen topic
* Clearly defined question inspired by the issue
* Rationale explaining what is being done and why
* Personal investment of Zhiwei and Qiuying in the project
* Importance of the topic beyond personal experience
* (Copy and paste the relevant sections from the Final Project Proposal)

# **Analysis**

* Description of methods used to gather data and/or solve the problem
  + Application of graph theory in project management
  + Focus on Directed Acyclic Graphs (DAGs)
  + Event planning as a general example
* Clear steps throughout the analysis, referencing specific topics/modules covered in the CS 5002 course

Python program development (submit the python file as an Appendix to the .pdf report)

## ***Create Directed Graph***

networkx DiGraph is a directed graph object in the NetworkX Python library. It is used to represent directed graphs, where each node has a directed edge connecting it to another node.

A DiGraph object in networkx can be created using the ‘DiGraph()’function, which initializes an empty directed graph. Nodes can be added to the graph using the ‘add\_node()’ method, and directed edges can be added using the ‘add\_edge()’ method.

For example, the following code creates a DiGraph object with three nodes and two directed edges (1à2, and 2à3):

**import** **networkx** **as** **nx**

G = nx.DiGraph()

G.add\_node(**1**)

G.add\_node(**2**)

G.add\_node(**3**)

G.add\_edge(**1**, **2**)

G.add\_edge(**2**, **3**)

Alternatively, we can use ‘add\_edges\_from()’ method to create directed graph with vertices and edges in one go. NetworkX is smart enough to infer the vertices from a collection of edges.

**import** **networkx** **as** **nx**

G = nx.DiGraph() # DiGraph is short for “directed graph”

G.add\_edges\_from([("root", "a"), ("a", "b"), ("a", "e"), ("b", "c"), ("b", "d"), ("d", "e")])

**print**(nx.shortest\_path(G, 'root', 'e')) # Output: ['root', 'a', 'e']

**print**(nx.dag\_longest\_path(G)) # Output: ['root', 'a', 'b', 'd', 'e']

# **Results**

* Discussion of the findings based on the analysis
* Application of graph theory to optimize project management
  + Scheduling projects
  + Allocating resources to tasks
  + Optimizing project budgets
* Identification of critical paths and activities that can be delayed without affecting the minimum completion time

# **More Examples / Real Life Examples**

* TBU

# **Conclusion**

* Answer the clearly defined question
* Discuss weaknesses and limitations of the project
* Suggest avenues for future research
* Personal reflections on what each group member learned from the project
  + Zhiwei's paragraph
  + Qiuying's paragraph
* Value of the report for future Northeastern courses or other endeavors upon graduation

# **References**

* [Building DAGs / Directed Acyclic Graphs with Python](https://mungingdata.com/python/dag-directed-acyclic-graph-networkx/) from MungingData
* [Notebook 2.2- Weighted and directed graphs](https://transport-systems.imperial.ac.uk/tf/60008_21/n2_2_weighted_and_directed_graphs.html)
* [HiLite.me](http://hilite.me/) to insert code snippet in the word document

# **Appendix**

* Python code for the project (.py file)

import networkx as nx  
import matplotlib.pyplot as plt  
import csv  
  
filename = "event\_planning.csv"  
  
# Initialize an empty dictionary to store the data  
data = []  
  
# Read the CSV file and store the data in the dictionary  
with open(filename, newline='', encoding='UTF-8') as csvfile:  
 reader = csv.DictReader(csvfile)  
 for row in reader:  
 data.append(row)  
  
# Print the data  
# for item in data:  
# print(item)  
  
  
# Global variable: Create an empty Weighted DAG (Directed Acyclic Graph)  
G = nx.DiGraph()  
  
  
# Function to find the longest path (brute-force approach)  
def find\_longest\_path(graph, start, end):  
 longest\_path = []  
 longest\_length = float('-inf')  
  
 for path in nx.all\_simple\_paths(graph, start, end):  
 length = 0  
 for i in range(len(path) - 1):  
 length += graph.edges[path[i], path[i + 1]]['weight']  
 if length > longest\_length:  
 longest\_length = length  
 longest\_path = path  
  
 return longest\_path, longest\_length  
  
  
def save\_graph(filename):  
 filename = filename + '.png'  
 plt.savefig(filename, format="PNG")  
  
  
# Create the edges lists with labels and weights (can set a default weight)  
edge\_list\_ww = [('A', 'B', 3), ('A', 'C', 2), ('B', 'C', 1), ('B', 'E', 3),  
 ('C', 'D', 8), ('E', 'F', 4), ('D', 'F', 2),  
 ('B', 'D', 4), ('E', 'D', 4)]  
  
# Fill the Weighted DAG  
G.add\_weighted\_edges\_from(edge\_list\_ww)  
  
# Graph validation: Returns True if the graph G is a DAG or False if not  
print('is\_directed\_acyclic\_graph: ', nx.is\_directed\_acyclic\_graph(G))  
  
# Define the start and end nodes  
start\_node = 'A'  
end\_node = 'F'  
  
# Find the longest path given the start and end node  
path, length = find\_longest\_path(G, start\_node, end\_node)  
print(f"The longest path is {path} with a length of {length}")  
  
  
# Formatting  
edges\_path = list(zip(path, path[1:]))  
edge\_colors = [  
 'black' if edge not in edges\_path else 'red' for edge in G.edges()]  
  
pos = nx.spring\_layout(G)  
print('pos:', pos.items()) # dict\_items([('A', array([-0.4, 0.76])), ...]  
  
node\_col = ['steelblue' if node not in path else 'red' for node in G.nodes()]  
  
# Draw DAG (w/o weight labels)  
nx.draw(G, pos, with\_labels=True, font\_color='white', edge\_color=edge\_colors,  
 node\_shape='s', node\_color=node\_col)  
  
  
nx.draw(G, pos, with\_labels=True, font\_color='white',  
 edge\_color=edge\_colors, node\_shape='s')  
  
  
# Draw the weight labels  
weight\_labels = nx.get\_edge\_attributes(G, 'weight')  
nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=weight\_labels)  
  
# Plot the graph in various ways  
# nx.draw\_spring(G, with\_labels=True)  
# nx.draw\_spectral(G, with\_labels=True)  
# nx.draw\_planar(G, with\_labels=True)  
# nx.draw\_random(G, with\_labels=True)  
  
# If you want an image file as well as a user interface window,  
# use pyplot.savefig before pyplot.show  
# After show() the figure is closed and thus unregistered from pyplot  
save\_graph('g14')  
plt.show()  
plt.clf()