Project: Computing the chromatic numbers of a graph

Author: Filippa Getzner

ID: 21124839

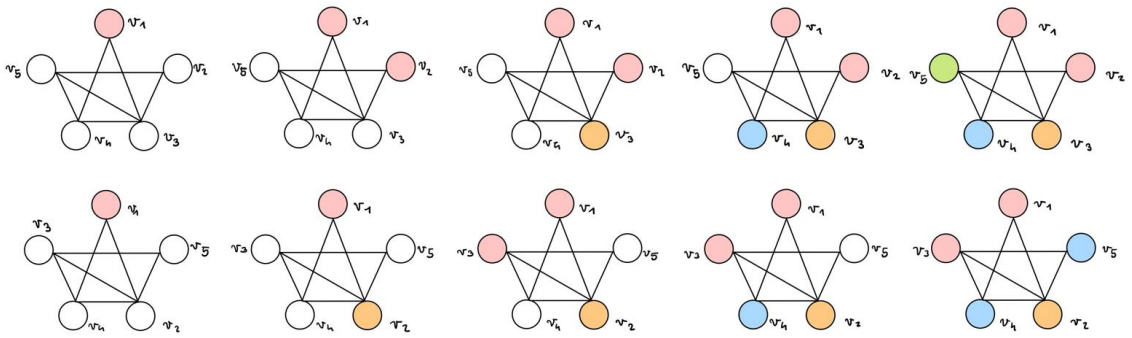
Module: CS319 Scientific computing

Email: f.getzner1@universityofgalway.ie

1) Introduction

The goal of the project was to write a class in C++ that represents graphs as adjacency matrices. It should have several member functions that e.g. add an egde, test if a graph is bipartite and most importantly, compute the chromatic number of a graph and return a colouring.

To recap: The chromatic number of a graph is the smallest number of colours needed to colour every vertex of the graph such that no two adjacent vertices have the same colour. The problem of finding the chromatic number is NP-hard. The greedy algorithm was chosen which only provides the user with an upper bound for the chromatic number. The greedy algorithm always chooses the best local option. It iterates through the vertices of a graph and tries to find the optimal colour for every vertex. The problem is that it can return different colourings for isomorphic graphs. In the following figure we can see that by changing the order of the vertices, greedycolouring outputs a different colouring. The graph in the first row clearly is not coloured optimally, whereas the one on the bottom row is (the figure is based on Guichard: An Introduction to Combinatorics and Graph Theory).



2) Code

As mentioned in the introduction, the goal was to write a class in C++ that represents a graph as an adjacency matrix. A method for testing bipartiteness, adding edges and computing the upper bound for the chromatic number were implemented.

We can split the C++ code into three parts:

1) `Graph.h` :

This is the header file in which the private and public members (methods) of the class are declared.

2) `Graph.cpp` :

In this file we have a constructor, destructor and the definition of all the methods. Here are the most important methods:

`Graph::Graph(unsigned int numVertices): num_Vertices(numVertices) , AdjacencyMatrix(numVertices)` : In the constructor we construct the `Graph` object using an initializer list. Doing it this way was necessary because we are using the `Matrix` object of the `Matrix10` class. The two colouring arrays (one is needed for testing the bipartiteness, the other one for the colouring function) are both initialized to `-1` . The size of the adjacency matrix is set to the number of vertices and the entries are set to `0` .

`Graph::~Graph()` : The destructor deletes the memory which was allocated for the two colouring arrays.

`Matrix Graph::Adj_Matrix_from_edges(double edges[][2], int numEdges)` : This function takes a 2D array (the edges) and an integer (the number of edges) as an input. The function defines the vertex `vi` as the first entry of the array of edges and the vertex `vj` as the second entry. Whenever there is an edge between `vi` and `vj` it calls the function `.setij()` to set the corresponding entry of the adjacency matrix to `1` .

`void Graph::add_edge(unsigned int vi, unsigned vj)` : The input consists of two vertices between which an edge should be added. It then uses `.setij()` to set the entry of the adjacency matrix to `1` .

`bool Graph::BipartiteTest()` and `bool Graph::dfsBipartite(unsigned int vi, int colour)` test if a graph is bipartite or not. The method `bool Graph::BipartiteTest()` loops through each vertex of the graph and whenever it hasn't been coloured, the function `bool Graph::dfsBipartite(unsigned int vi, int colour)` is called. This function uses depth first search to recursively check if the graph can be coloured in a bipartite way. That is, if the number of needed colours exceeds 2, then it returns `False` and as a consequence `bool Graph::BipartiteTest()` returns `False` .

`int Graph::greedyColouring(Matrix AdjacencyMatrix)` : This functions takes an adjacency matrix as input. at the start, an unordered set of used colours is initialised. It iterates through all vertices, checks the colour of the neighbouring vertices and assigns the lowest possible colour to the current vertex (Note: colours in this case are just integers greater or equal to 0). At the end this function finds the largest integer stored in the used colours set and returns this integer as the chromatic number.

3) `main.cpp` :

In the main.cpp file we want to test the functions as well input and output files.

First, a `.csv` file is loaded into the program. The first row consists of only one integer, namely the number of vertices. The remaining rows are edge pairs with the vertices being seperated by a space. The program counts the number of edges and initialises a 2D array to store the edges. Then, a `Graph` object with the given number of vertices is created and the `Matrix Graph::Adj_Matrix_from_edges(double edges[][2], int numEdges)` function computes the adjacency matrix. The program then prompts the user to input two vertices of their choice to add an edge between them using the `void Graph::add_edge(unsigned int vi, unsigned vj)` function. After adding the edge, the bipartiteness (of the original graph) is tested and the chromatic number is computed using `bool Graph::BipartiteTest()` and `int Graph::greedyColouring(Matrix AdjacencyMatrix)` respectively. Finally, using the function `void WriteVertexColouringCSV(Graph graph, std::string FileName)` a csv file is created storing the vertices and their colours.

3) Case use:

In [ ]:
```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import grinpy as gp
import pandas as pd

opts = {
    "with_labels": True,
    "font_weight": 'bold',
    "node_color": 'pink',
    "node_size": 500,
}
```

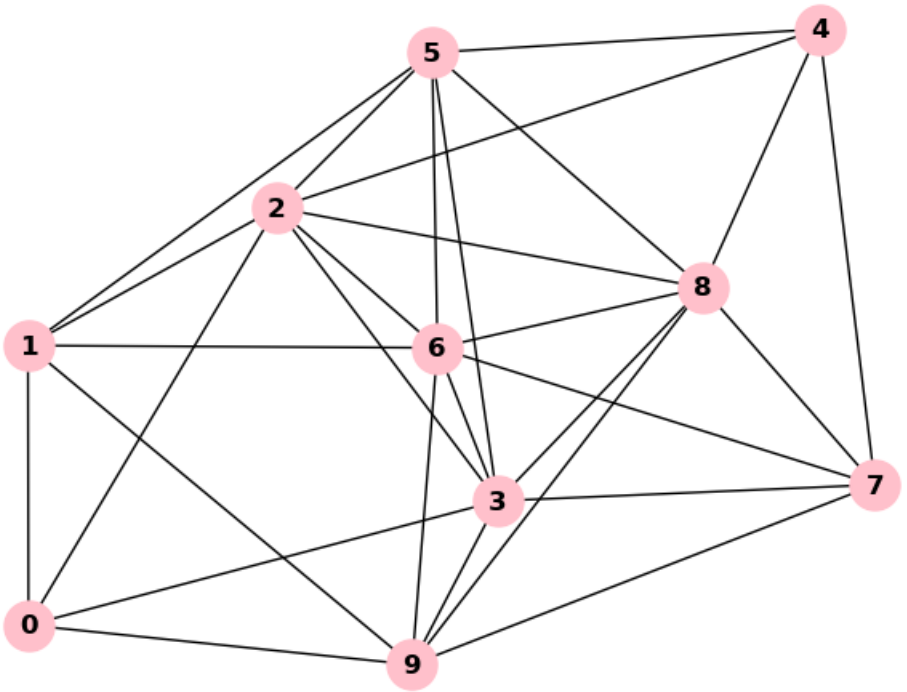
Generate an example graph

The `export_to_csv` function can be used to export (generate) a `.csv` file storing the number of vertices and the edge pairs.

In [ ]:
```
def export_to_csv(graph, file_path):

    with open(file_path, 'w') as file:

        file.write(str(graph.number_of_nodes()) + '\n')

        for edge in graph.edges():
            file.write(f"{edge[0]} {edge[1]}\n")
```

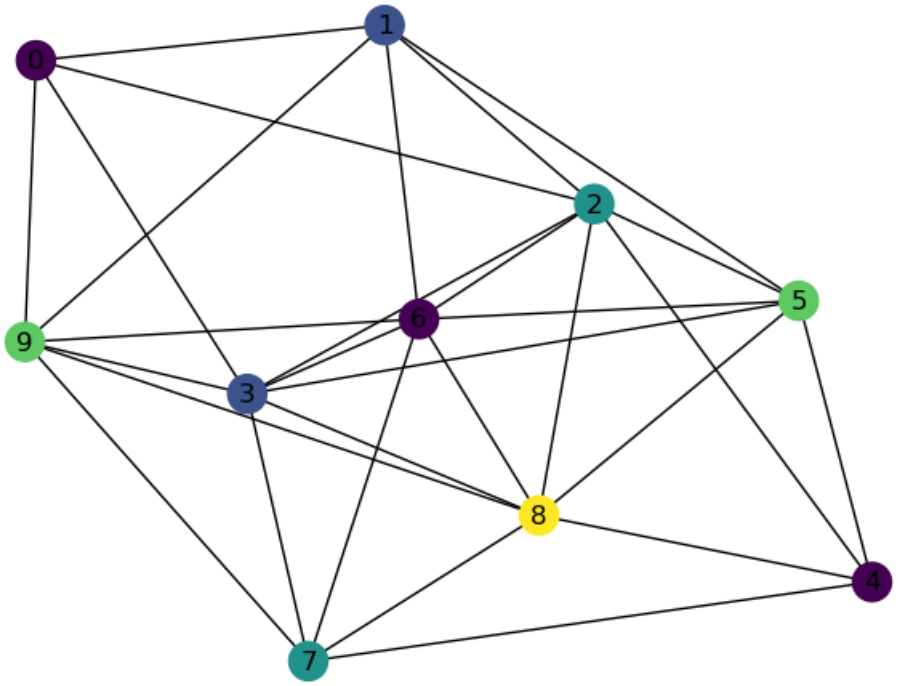
In [ ]:
```
n = 10
p = 0.5
g = nx.gnp_random_graph(n,p)
nx.draw(g,**opts)

InFileName = 'graph_data.csv'

# Export the graph to the CSV file
export_to_csv(g, InFileName)

print(f"Graph exported to {InFileName}")
```

```
Graph exported to graph_data.csv
```



Run the `C++` code now. It will generate the `colouring.csv` file which is needed for the next step: Visualising the coloured graph.

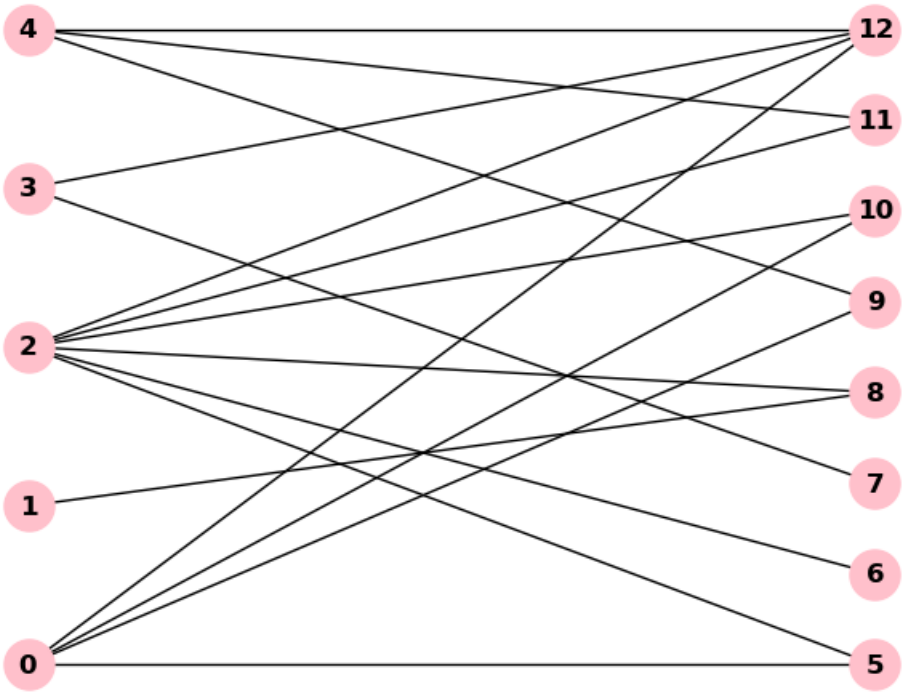
In [ ]:
```
# read the vertex colors from the colouring.csv file
df = pd.read_csv('colouring.csv', header=None, names=['vertex', 'color'])

color_map = df.set_index('vertex')['color'].to_dict()

# apply the colours to the graph
colors = [color_map[node] for node in g.nodes()]

# draw the graph
nx.draw(g, node_color=colors, with_labels=True)
```



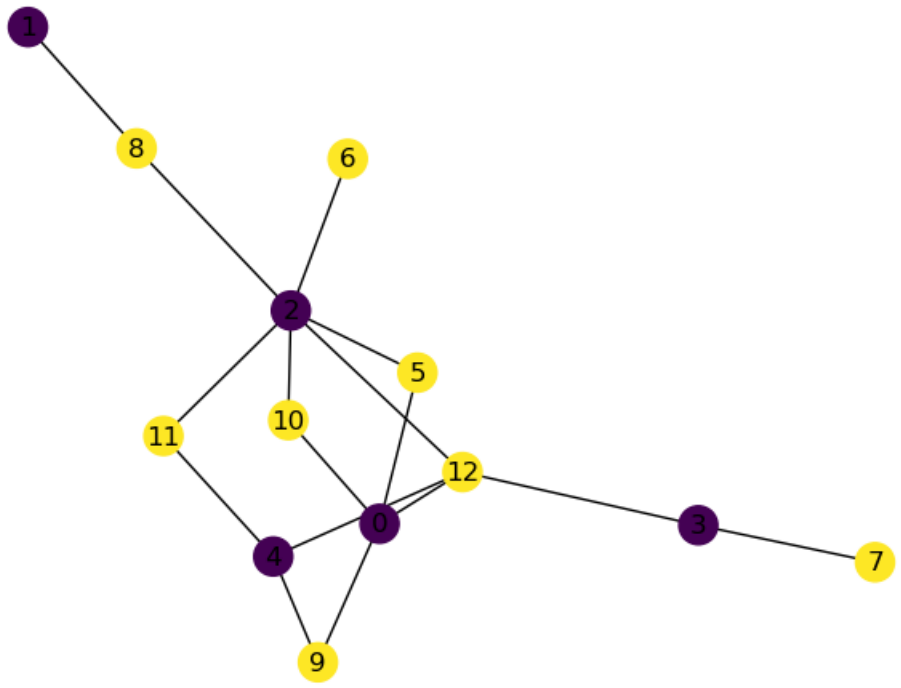
Test a graph that is definitely bipartite

In [ ]:
```
G = nx.from_edgelist([(5,0),(5,2),(6,2),(7,3),(8,1),(8,2),(9,0),(9,4),(10,0),(10,2),(11,2),(11,4),(12,0),

# Assign "colours" (0,1) to the actors and foci
color = nx.bipartite.color(G)

# nodes actors and foci based on their colour
actors = [node for node, color in color.items() if color == 0]
foci = [node for node, color in color.items() if color == 1]

# draw and show the graph
pos = nx.bipartite_layout(G, actors)
nx.draw(G, pos, **opts)

InFileName = 'graph_data_bip.csv'

# Export the graph to the CSV file
export_to_csv(G, InFileName)

print(f"Graph exported to {InFileName}")
```

```
Graph exported to graph_data_bip.csv
```



In [ ]:
```
# read the vertex colors from the colouring.csv file
df = pd.read_csv('colouring_bip.csv', header=None, names=['vertex', 'color'])

color_map = df.set_index('vertex')['color'].to_dict()

# apply the colours to the graph
colors = [color_map[node] for node in G.nodes()]

# draw the graph
nx.draw(G, node_color=colors, with_labels=True)
plt.show()
```



4) Discussion

1) What I have learnt.

The most significant skill I learnt is writing a class in `C++`. Also I have learnt how to handle data and practiced the allocation and deallocation of memory. Another thing I have learnt is how to input and output `csv` files and how to combine `Python` and `C++`. Properly understanding algorithms (dfs and greedy) was tough but I got there in the end (Note: I got the algorithms from geeksforgeeks.com). I think it was quite interesting to see how the greedy algorithm produces different colourings depending on the order of vertices and I also think that is when the greedy algorithm finally made sense to me.

2) Verification.

I verified the correctness of the adjacency matrix simply by comparing the edges from the csv file to the entries of the matrix. By comparing the matrix before and after adding the edge I could see that there was indeed a new edge between the two entered vertices. For the bipartiteness test function I tested graphs that are not bipartite and a graph that is bipartite (seen above) to see if it gave the correct answer.

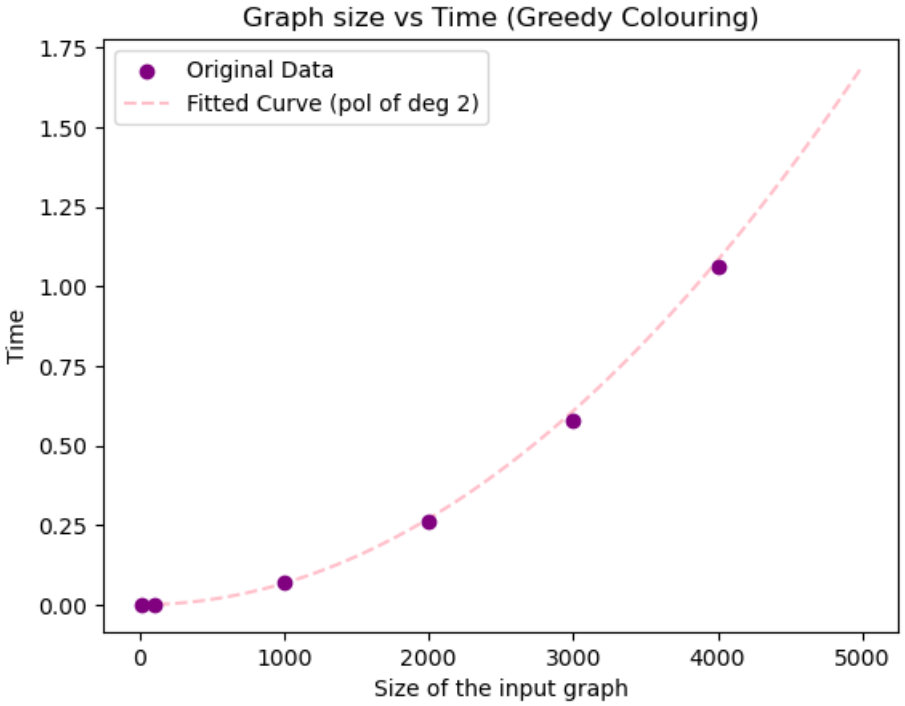
3) Run time analysis.

I determined the run time of the greedy algorithm for differently sized graphs. When I got to 4000 vertices my laptop started making too much noise so I stopped there. We can say from the plot that the run time of the greedy algorithm is polynomial (degree = 2).

In [ ]:
```
''' used this code for the run time analysis
n = 3000
p = 0.5
G = nx.gnp_random_graph(n,p)
InFileName = 'graph_data_3000.csv'
export_to_csv(G, InFileName)
print(f"Graph exported to {InFileName}")
'''
```

Out[ ]:
```
'\nn = 3000\np = 0.5\nG = nx.gnp_random_graph(n,p)\nInFileName = \'graph_data_3000.csv\'\nexport_to_csv
(G, InFileName)\nprint(f"Graph exported to {InFileName}")'
```

In [ ]:
```
N = np.array([10,100,1000,2000,3000,4000])
T = np.array([0, 0, 0.07, 0.262,0.579, 1.062])

poly_coeffs = np.polyfit(N, T, deg=2)
poly_fit = np.polyval(poly_coeffs, N)

Ns = np.linspace(0,5000,100)
fit_Time = []

for i in Ns:
    fit_Time_i = poly_coeffs[0]*i*i+poly_coeffs[1]
    fit_Time.append(fit_Time_i)


plt.scatter(N,T,marker= 'o', color = 'purple', zorder=2, label = 'Original Data')
plt.plot(Ns, fit_Time, '--', label='Fitted Curve (pol of deg 2)', color = 'pink', zorder=1)
plt.title('Graph size vs Time (Greedy Colouring)')
plt.legend()
plt.xlabel('Size of the input graph')
plt.ylabel('Time')
plt.show()
```



5) Highlights

The highlight for me was when I found out why my greedy colouring function was not outputting the correct chromatic number and a colouring which did not make any sense. I used the same colours array for the bipartiteness test function and for the greedy colouring so by the time the program called the greedy colouring function, there were already numbers other than -1 stored in the colours array. I think it took me three hours to find this painfully obvious error. In general, all highlights were about finding an error that was not really an error (as in the program would still run but wouldn't give a sensible output). I am quite proud of how the user (me in this case) can generate graphs which then automatically get saved to the correct folder and then all the user has to do is run the `C++` program and run the `Python` snippet to get the coloured graph.

6) Conclusion

A limitation of my code is that it can not handle large inputs. I think this is due to the way the `.csv` file is read (the program reads through the whole file twice) and how the matrix is created. If I had more time I would try to find a way to do it with sparse matrices. In general, I am quite happy with the project as I achieved everything that I mentioned in the project proposal.

7) References

- Niall Madden: Slides, lab assignments, examples from CS319 Scientific Computing
- GeeksforGeeks: https://www.geeksforgeeks.org/chromatic-number-of-a-graph-graph-colouring/ and https://www.geeksforgeeks.org/bipartite-graph/ (last accessed: 03.04.2024)
- David Guichard: An Introduction to Combinatorics and Graph Theory, https://www.whitman.edu/mathematics/cgt_online/cgt.pdf (last accessed: 03.04.2024)
- Documentation of different functions: https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html (last accessed: 03.04.2024)
- Angela Carnevale: Lecture Notes from CS4423 Networks
- Ceolla Dillon O'Rourke, Filippa Getzner, Deeba Javadpour: NP-hardness of computing the chromatic number (Report/Group project for MA438 Mathematics for Decision Making)
- Explanation of the initializer list: https://www.youtube.com/watch?v=1nfuYMXjZsA (last accessed: 03.04.2024)