In [1]:

```
import pandas as pd
import numpy as np
import itertools
from itertools import chain
import networkx as nx
import matplotlib.pyplot as plt
from random import sample
import networkx as nx

from openpyxl.styles import PatternFill
from openpyxl import Workbook
from openpyxl.styles import Border,Side,Alignment,Font


import os
import time


mypath = os.getcwd()

# Available fill color RGB code

"""
colors source: https://color.d777.com/
#00bfff  Deep Sky Blue
#8dd9cc  Middle Blue Green
#cccaa8 Thistle Green
#bcd4e6 Pale Aqua
#ee82ee Lavender Magenta
#ffc0cb Pink
#a899e6 Dull Lavender
#fe4eda Purple Pizzazz
#f8de7e Mellow Yellow
#ffffbf Very Pale Yellow
#d2b48c Tan
#bfafb2 Black Shadows
#e5d7bd Stark White
#3399ff Brilliant Azure
#91a3b0 Cadet Grey
#d0ff14 Arctic Lime
#fde1dc Cinderella
"""
colors = ['#00bfff','#8dd9cc','#cccaa8','#bcd4e6','#ee82ee',
          '#ffc0cb','#a899e6','#fe4eda','#f8de7e','#ffffbf',
          '#d2b48c','#bfafb2','#e5d7bd','#3399ff','#91a3b0',
          '#d0ff14','#fde1dc']

#Course week
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

#Course time period
times = ['08:00-08:45','09:00-09:45','10:00-10:45','11:00-11:45','13:30-14:15','14:30-15:15'

#Initializes the total class schedule
course_table = pd.DataFrame([[""]*5]*len(times),columns=days,index=times)


# In[3]:

```

```python

def get_nodes_edges(data):
    """
    data: (Dataframe)
    Generate bound edges between course nodes and courses according to the student’s course
    """
    all_courses = data.loc[:,1:].stack().groupby(level=1, sort=False).agg(list).tolist()
    nodes = list(set(list(chain(*all_courses))))
    courses = data.fillna(’nan’)

    courses_pair = []
    for i in range(len(courses)):
        comb = list(itertools.combinations(courses.loc[i,1:].values, 2))
        courses_pair.extend(comb)

    edges = []
    for comb_i in list(set(courses_pair)):
        if comb_i[0]==’nan’ or comb_i[1]==’nan’:
            continue
        edges.append(comb_i)
    edges = list(set(edges))
    return nodes, edges

class Backtracking:
    """
    Graph coloring backtracking algorithm
    """
    def __init__(self, nodes, edges):
        G = nx.Graph()
        G.add_nodes_from(nodes)
        G.add_edges_from(edges)
        # Number of vertices
        self.G = G
        self.V = self.G.number_of_nodes()
        # The adjacency matrix of the graph
        self.graph = nx.to_numpy_array(self.G)

    def is_safe(self, v, color, c):
        # Check whether vertex v can be colored to color c
        for i in range(self.V):
            if self.graph[v][i] == 1 and color[i] == c:
                return False
        return True

    def graph_color_util(self, m, color, v):
        if v == self.V:
            return True

        # Try all possible colors
        for c in range(1, m + 1):
            if self.is_safe(v, color, c):
                color[v] = c

                if self.graph_color_util(m, color, v + 1):
                    return True

                # Backtrack, reset the color of vertex v
                color[v] = 0

        return False

    def graph_coloring(self, m):
        color = [0] * self.V
        color_map = {}
        if not self.graph_color_util(m, color, 0):
            #print("No viable solution can be found")
            return color_map

        #print("Feasible solution exists")
        for vertex in range(self.V):
            #print(f"顶点 {list(G.nodes)[vertex]} 的颜色: {colors[color[vertex]]}")
            color_map[list(self.G.nodes)[vertex]] = colors[color[vertex]]
        return color_map

def greedy_graph_coloring(courses, students):
    """
    Greedy algorithm graph coloring
    """
    # Create a dictionary to store the courses adjacent to each course
    adjacency_list = {}

    # Add a course to the dictionary
    for course in courses:
        adjacency_list[course] = set()

    # Build the relationship between adjacent courses
    for student in students:
        for i in range(len(student)):
            for j in range(i + 1, len(student)):
                course1, course2 = student[i], student[j]
                adjacency_list[course1].add(course2)
                adjacency_list[course2].add(course1)

    # Store the color of each course
    color_map = {}

    # Walk through each course and color it
    #参考贪心算法https://blog.csdn.net/nice___amusin/article/details/117090393
    for course in courses:
        # Gets the colors of adjacent courses
        neighbor_colors = {color_map.get(neighbor) for neighbor in adjacency_list[course]}

        # Find an available color to color
        for color in colors:
            if color not in neighbor_colors:
                color_map[course] = color
                break
    return color_map

def smallest_last_graph_coloring(nodes, edges):
    """
    smallest last graph coloring
    """

    G = nx.Graph()
    G.add_nodes_from(nodes)
    G.add_edges_from(edges)

    #Gets a list of neighbors for each vertex
    graph = {}
    for n in G.adj:
        graph[n] = []
        for a in G.adj[n]:
            graph[n].append(a)


    #Calculate the degree of each vertex
    degrees = {vertex: len(adj) for vertex, adj in graph.items()}
    #Sort the vertices in order of degree from smallest to largest
    sorted_vertices = sorted(degrees, key=degrees.get)

    # Do the following for each vertex:
      # (1) Find the first color in the color list that does not conflict with the colors of
      # (2) Assigns the color to the current vertex and saves the result to the result dicti
    result = {}
    for vertex in sorted_vertices:
        neighbor_colors = {result.get(adj) for adj in graph[vertex] if adj in result}
        available_colors = [color for color in colors if color not in neighbor_colors]

        if not available_colors:
            colors.append(len(colors))
            result[vertex] = len(colors) - 1
        else:
            result[vertex] = min(available_colors)

    return result

def graph_coloring(courses, students_courses, alg='greedy'):
    """
    Two graph coloring algorithms are selected
    """
    if alg == 'greedy':
        course_colors = greedy_graph_coloring(courses, students_courses)

    elif alg == 'backtracking':
        g = Backtracking(courses, students_courses)
        # Maximum number of available colors
        max_color = len(courses)
        course_colors = g.graph_coloring(max_color)

    elif alg == 'smallest last':
        course_colors = smallest_last_graph_coloring(courses, students_courses)

    if len(course_colors.items()) != len(courses):
        raise ValueError("Not complete all course coloring")

    return course_colors

def check_conflict(courses_raw, course_colors):
    """
    According to the results of graph coloring,
    the conflict detection of students' course
    selection table is carried out
    """
    students_courses = courses_raw.replace(course_colors)
    students_courses['colors'] = students_courses.iloc[:,1:].apply(lambda row: list(row.dro
    students_courses['is_unconflict'] = students_courses['colors'].apply(lambda x: len(set(
    #students_courses.to_csv('test.csv',index=False)
    if len(students_courses[students_courses['is_unconflict']==False])==0:
        return False
    return True

def courses_slots(course_colors):
```

```python
    """
    Fill slots according to the result of shading generated time
    """

    mark_colors = list(set(course_colors.values()))
    days_times = list(itertools.product(days,times))
    times_slot = sample(days_times, len(mark_colors*2))

    #Assign lessons of the same color to the same time slot and different classrooms
    slot_course_room = []
    for cinx,c in enumerate(sample(mark_colors*2,len(mark_colors*2))):
        course_array = colors_course[colors_course['colors']==c]['course'].values
        #print(days_times[inx],'\t',c,'\t',colors_course[colors_course['colors']==c]['course
        for rinx, i in enumerate(range(len(course_array))):
            room  = '(room_' + str(i+1) + ')'
            #print(f"""{days_times[cinx]}\t{c}\t{course_list[rinx]}\t{room}""")
            slot_course_room.append([times_slot[cinx][0],times_slot[cinx][1],str(course_arra
    slot_course_room = pd.DataFrame(slot_course_room,columns = ['week','time','course','roo

    # Fill all the courses into the class schedule
#     for inx,row in slot_course_room.iterrows():
#         course_table.loc[row['time'],row['week']] += row['course']+row['room']+'|'
    return slot_course_room

def get_students_timetable(sid):
    """
    Generate student schedule based on student id and course summary
    """
    selected_course = students_selected.loc[students_selected['id']==sid]['selected'].value
    students_course_table = pd.DataFrame([[""]*5]*len(times),columns=days,index=times)

    for sc in selected_course:
        course_df = slot_course_room[slot_course_room.loc[:,'course']==sc]
        for i,row in course_df.iterrows():
            students_course_table.loc[row['time'],row['week']] += row['course']+row['room']
    return students_course_table

def fill_timetable(sid, table, alg):
    """
    Use openpyxl to fill cells with values and backgrounds
    """

    # Set font font
    font_title = Font(u'times new roman',size=20)
    font_values = Font(u'times new roman',size=10)

    # Centered style
    align = Alignment(horizontal='center',vertical='center',wrap_text=True)

    # Border style
    border = Border(left=Side(border_style='thin'),
        right=Side(border_style='thin'),
        top=Side(border_style='thin'),
        bottom=Side(border_style='thin'))
    # Create a Workbook object and get its active worksheet
    wb = Workbook()
    ws = wb.active
    ws.cell(1, 1, f'SID: {sid}')
    # Write header
    for i, header in enumerate(days):
        cell = ws.cell(1, i+2, header)
        cell.alignment = align
        cell.border = border
        cell.font = font_title
    for i, time_range in enumerate(times):
        cell = ws.cell(i+2, 1, time_range)
        cell.alignment = align
        cell.border = border
        cell.font = font_title

    #Set column width
    colwidth = 30
    ws.column_dimensions['a'].width = 28
    ws.column_dimensions['b'].width = colwidth
    ws.column_dimensions['c'].width = colwidth
    ws.column_dimensions['d'].width = colwidth
    ws.column_dimensions['e'].width = colwidth
    ws.column_dimensions['f'].width = colwidth
    #原文链接：https://blog.csdn.net/bigfishfish/article/details/123247362


    table.columns = range(len(table.columns))
    table.index = range(len(table.index))

    # Iterate over each cell of the DataFrame,
    #filling in the conditions that satisfy the background coloring
    for index, row in table.iterrows():
        for col_index, value in row.items():
            cell = ws.cell(row=index + 2, column=col_index + 2, value=value)
            cell.alignment = align
            cell.border = border
            cell.font = font_values
            if value:
                cs = value[:2]
                fill_color = colors_course[colors_course['course']==cs]['colors'].values[0]
                fill = PatternFill(start_color=fill_color,end_color=fill_color,fill_type='s
                cell.fill = fill

    # Save Excel file
    wb.save(f"./students_table/{sid}_courses_table_{alg}.xlsx")
```

In [2]:

```
def test(alg):
    #start = time.process_time()
    start = time.process_time_ns()
    path = 'anonymised(3).csv'
    courses_raw = pd.read_csv(path,header=None)
    courses_raw = courses_raw.replace(' ','',regex=True)
    nodes, edges = get_nodes_edges(courses_raw)
    students_courses = courses_raw.iloc[:,1:].apply(lambda row: list(row.dropna()), axis=1)

    #Generate the student course selection table
    courses_raw['selected'] = courses_raw.iloc[:,1:].apply(lambda row: list(row.dropna()),

    global students_selected,slot_course_room,colors_course
    students_selected = courses_raw[[0,'selected']]
    students_selected.columns = ['id','selected']

    #Generate class time classroom slots
    if alg == 'greedy':
        course_colors = graph_coloring(nodes, students_courses, alg)
    elif alg == 'smallest last':
        course_colors = graph_coloring(nodes, edges, alg)
    elif alg == 'backtracking':
        course_colors = graph_coloring(nodes, edges, alg)

    colors_course = pd.DataFrame(zip(course_colors.values(), course_colors.keys()),columns=[
    slot_course_room = courses_slots(course_colors)
    time_process = []
    for sid in range(880):
        sid = int(sid)
        students_course_table = get_students_timetable(sid)
        fill_timetable(sid, students_course_table, alg)
        #print("KMeans Time cost: {0}".format(time.process_time() - start))
        time_process.append((time.process_time_ns() - start)*0.000000001)
    return time_process,len(slot_course_room['room'].unique())
```

In [3]:

```
%%time
res = pd.DataFrame()
for alg in ['greedy', 'backtracking', 'smallest last']:
    res[alg],nums = test(alg)
    print(alg,'Number of classrooms in use :',nums)
```

```
greedy Number of classrooms in use : 6
backtracking Number of classrooms in use : 6
smallest last Number of classrooms in use : 9
CPU times: total: 40.9 s
Wall time: 45.5 s
```

In [4]:

```python
res
```

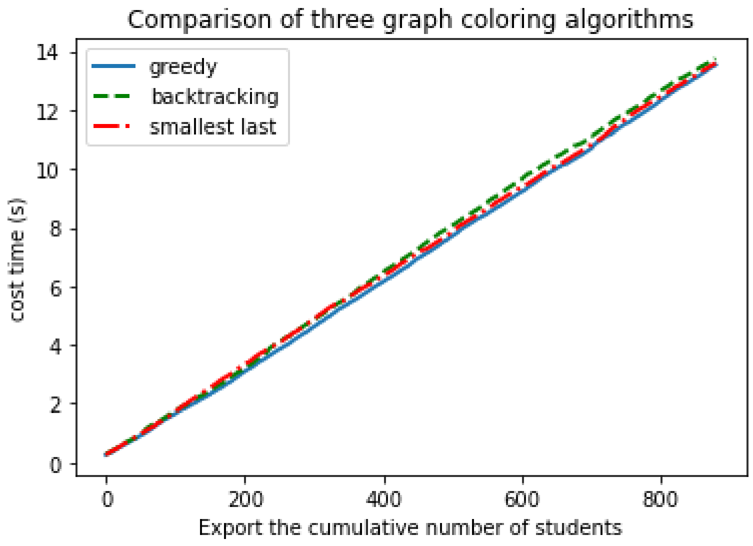
Out[4]:

```
       greedy  backtracking  smallest last
0    0.265625      0.281250       0.265625
1    0.281250      0.296875       0.281250
2    0.296875      0.312500       0.296875
3    0.312500      0.328125       0.312500
4    0.328125      0.343750       0.328125
...       ...           ...            ...
875 13.484375     13.687500      13.531250
876 13.500000     13.703125      13.546875
877 13.500000     13.718750      13.562500
878 13.515625     13.734375      13.578125
879 13.531250     13.750000      13.593750

880 rows × 3 columns
```

In [5]:

```python
plt.plot(res.index,res['greedy'],label='greedy',linewidth = 2)
plt.plot(res.index,res['backtracking'],'g--',label='backtracking',linewidth = 2)
plt.plot(res.index,res['smallest last'],'r-.',label='smallest last',linewidth = 2)
plt.ylabel('cost time (s)')
plt.xlabel('Export the cumulative number of students')
plt.title('Comparison of three graph coloring algorithms')
plt.legend()
plt.show();
```


Comparison of three graph coloring algorithms
greedy
backtracking
smallest last
cost time (s)
Export the cumulative number of students