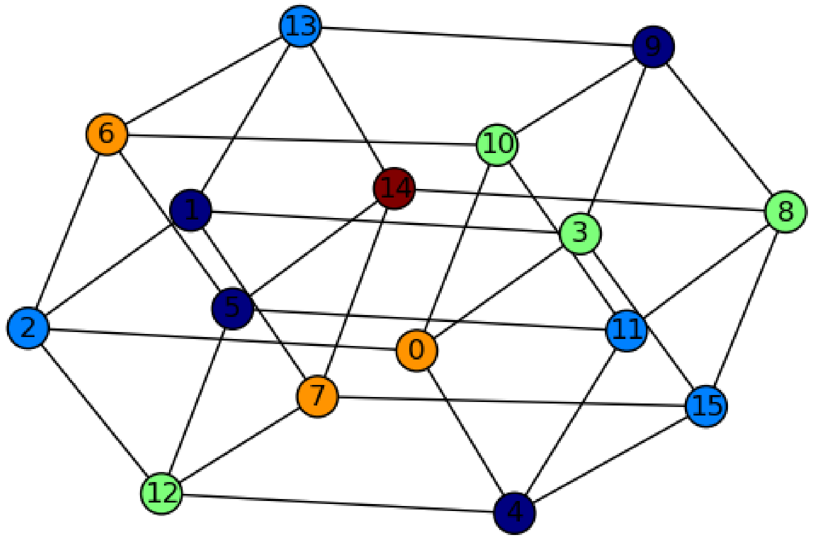
2016 Fall, GIST, Graph Theory

Scheduling problem, chromatic number with Python and Networkx

20145071 Junghoon Seo



Introduction

This report is for the 2016 GIST fall semester graph theory project problem. I have been given four selection topics: 'Graceful Labeling of trees',' Various Algorithms and Questions', 'Bipartite matching', 'Scheduling problem, chromatic number', and I am more comfortable writing code than writing a too long English report so I select the topic 'Scheduling problem, Chromatic number '. I designed, implemented, and optimized the algorithm to meet the problems presented in the project. Python and NetworkX have been used, and NetworkX has been limited to visualization of graphs or automatic generation of graphs to implement core algorithms directly. The project problem is largely divided into three parts: finding the worst case of the greedy algorithm, finding the chromatic number using the Fundamental Reduction Theorem, and solving the scheduling problem using the algorithm we designed in the second. As a result, I succeeded in designing all the algorithms and implementing the code to implement them.

Problems and Solutions

1) Find a bad ordering of the vertices of G that requires (gamma + 1) colors to properly color Q_4 when applying Greedy Algorithm with that order.

In general, the degree-based vertex ordering heuristic is considered when solving the graph coloring problem with the greedy algorithm. I did not take much of the degree-based vertex ordering heuristic to solve this problem. Because the hypercube graph has the same degree of all vertices, it is meaningless to do degree-based vertex ordering. Of course, it is also possible to come up with a heuristic to get a good vertex ordering to get the worst case as the problem demands. However, the 4-hypercube has only 16 vertices, and all of this generates only 16! (= 2.092279e + 13) permutations of vertex ordering. So I decided to start with a really ignorant heuristic: Random ordering heuristic. If the algorithm through this approach terminates too late, another heuristic can be considered. So I started my greedy graph coloring by each randomly-generated vertex order, and I write the code to finish the algorithm if the number of coloring numbers matches the upper bound of chromatic number 5.

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

Q4 = nx.hypercube_graph(4)
Q4 = nx.convert_node_labels_to_integers(Q4)
Q4_nodes = Q4.nodes()

while(1):
    random_permu = np.random.permutation(len(Q4_nodes))
```

```
node_coloring = []
for i in range(len(Q4_nodes)):
    node_coloring.append(None)

for i in random_permu:
    node_neighbors = Q4.neighbors(Q4_nodes[i])
    neighbors_colors = [node_coloring[j] for j in node_neighbors]

    for color in range(5):
        if color in neighbors_colors:
            pass
        else:
            node_coloring[i] = color
            break

if max(node_coloring) == 4:
    nx.draw(Q4 ,pos=nx.spring_layout(Q4), with_labels = True, node_color=node_coloring)
    plt.show()
    print("Vertex Ordering")
    print(random_permu)
    print("Node Coloring")
    print(node_coloring)
    break
```

Unexpectedly, the algorithm quickly got out of the infinite loop, not worth measuring time. The figure drawn by the 'draw' function is the figure in the first chapter of this report. Vertex has numbering from 0 to 15. The 'random_permu' representing the order of coloring the vertex was output as [4 1 11 2 12 15 3 9 0 10 7 13 5 8 14 6]. 'Node_coloring' is a list of colors that are colorized by the algorithm. The color of the ith vertex is node_coloring [i]. We can see that the algorithm works for 4-hypercube case seamlessly with bad performance.

2) Use Fundamental Reduction Theorem to construct an algorithm for determining the chromatic number of graph G. Use your algorithm calculate the chromatic number for various graphs with n >= 20. How good is your algorithm?

First, we need to implement edge deletion and edge contraction in the graph. This is because they are inputs to the recursive equation of the Fundamental Reduction Ther-terem. Edge deletion is not difficult, but edge contraction needs to be more careful because it merges two vertices. Let v and w denote the two vertices that are indicated by the edge to be contracted. Add an edge that connects v with neighbors (not v) of w, and delete w and contracted edge.

```
def merge_first_edge(G):
  v = G.edges()[0][0]
  w = G.edges()[0][1]
  edges = G.edges()

  for edge in edges:
    if edge[0] == v and edge[1] != w:
      G.add_edge(edge[1], w)

  G.remove_edge(G.edges()[0][0], G.edges()[0][1])
  G.remove_node(v)

  return G

def remove_first_edge(G):
  G.remove_edge(G.edges()[0][0], G.edges()[0][1])

  return G
```

The next step is to calculate the coefficients of each polynomial term by the Fundamental Reduction Theorem. We use a numeric array to finally calculate the polynomial array by adding or subtracting the degree of the no edge graph. Note that the dynamic programming technique was used to minimize computation time. For a graph that has already computed a degree, it does not recompute the degree list, but instead loads the array from the list that was just memoized. As a result of comparing the computation time before and after application of the dynamic programming technique, the results were 371 times and 668 times faster for the 20-cycle graph and the 21-cycle graph, respectively. (Before applying DP, 142 seconds at 20-C and 290 seconds at 21-C. After applying DP, 0.12

sec at 20-C and 0.14 sec at 21-C.) To check the isomorphism of graphs and memoized graphs of DP, we first checked the degree check and then confirmed whether it is isomorphic. This also showed a speed increase of about 3 times in both C20 and C21 depending on the degree confirmation. The implemented function returning chromatic polynomial is as follows.

```
import numpy as np

memo_ls = {}
memo_return = {}
memo_index = 0

def graph_chromatic_num(G, init_node_n):
    global memo_index

    for edge in G.edges():
        if edge[0] == edge[1]:
            G.remove_edge(edge[0], edge[1])

    if len(G.edges()) == 0:
        deg = np.array([0]*(init_node_n + 1))
        deg[len(G.nodes())] = 1
        return deg

    for i, memoed_graph in enumerate(memo_ls.values()):
        if nx.faster_could_be_isomorphic(G, memoed_graph):
            if nx.is_isomorphic(G, memoed_graph):
                return memo_return[i]

    memo_return[memo_index] = (graph_chromatic_num(remove_first_edge(G.copy()), init_node_n) -
graph_chromatic_num(merge_first_edge(G.copy()), init_node_n))
    memo_ls[memo_index] = G.copy()
    memo_index += 1

    return memo_return[memo_index - 1]
```

In general, for n> = 20, the coefficient of the polynomial equation becomes very large. Therefore, for numerical stability, it is better to rearrange the rank of the operator in parentheses rather than calculating the polynomial directly. In fact, if you do not use this method, you will not be able to solve the problem due to integer overflow in the latter scheduling problem.

```
def chromatic_calc(x, degree):
  s = degree[len(degree) - 2] + degree[len(degree) - 1] * x
  for i in range(1, len(degree) - 1):
    s = degree[len(degree) - i - 2] + x * s
  return s
```

In the end, the chromatic number is the smallest integer starting from 1 and making sure that the value of this 'chromatic_calc' is not zero.

```
def chromatic_check(G, degree):
  for i in range(1, len(G.nodes())+1):
    chromatic = chromatic_calc(i, degree)
    if chromatic != 0:
      return i
```

For or C20, C21, C22, I tested the algorithm I designed and implemented. You can see that Degree and Chromatic Number are properly returned, and computation time is acceptable. Note that for all cycle graphs, if n is even, the chromatic number is 2, and if n is odd, the chromatic number is 3.

```
G = nx.cycle_graph(20)

memo_ls = {}
memo_return = {}
memo_index = 0

start_time = time.time()
degree = graph_chromatic_num(G, len(G.nodes()))
print(degree)
print(chromatic_check(G, degree))
print("--- %s seconds ---" % (time.time() - start_time))
```

```
[      0     -19     190   -1140    4845  -15504   38760  -77520  125970
 -167960  184756 -167960  125970  -77520   38760  -15504    4845   -1140
     190     -20       1]
2
--- 0.170773029327 seconds ---
```

```
G = nx.cycle_graph(21)

memo_ls = {}
memo_return = {}
memo_index = 0

start_time = time.time()
degree = graph_chromatic_num(G, len(G.nodes()))
print(degree)
print(chromatic_check(G, degree))
print("--- %s seconds ---" % (time.time() - start_time))
```

```
[      0      20    -210    1330   -5985   20349  -54264  116280 -203490
  293930 -352716  352716 -293930  203490 -116280   54264  -20349    5985
   -1330     210     -21       1]
3
--- 0.160291910172 seconds ---
```

```
G = nx.cycle_graph(22)

memo_ls = {}
memo_return = {}
memo_index = 0

start_time = time.time()
degree = graph_chromatic_num(G, len(G.nodes()))
print(degree)
print(chromatic_check(G, degree))
print("--- %s seconds ---" % (time.time() - start_time))
```

```
[      0     -21     231   -1540    7315  -26334   74613 -170544  319770
 -497420  646646 -705432  646646 -497420  319770 -170544   74613  -26334
    7315   -1540     231     -22       1]
2
--- 0.189464092255 seconds ---
```

3) Apply your algorithm to determine the minimum number of seats a train requires for arranging passengers having their travel intervals as follows.

Route of a train : A - B - C - D - E - F - G - H

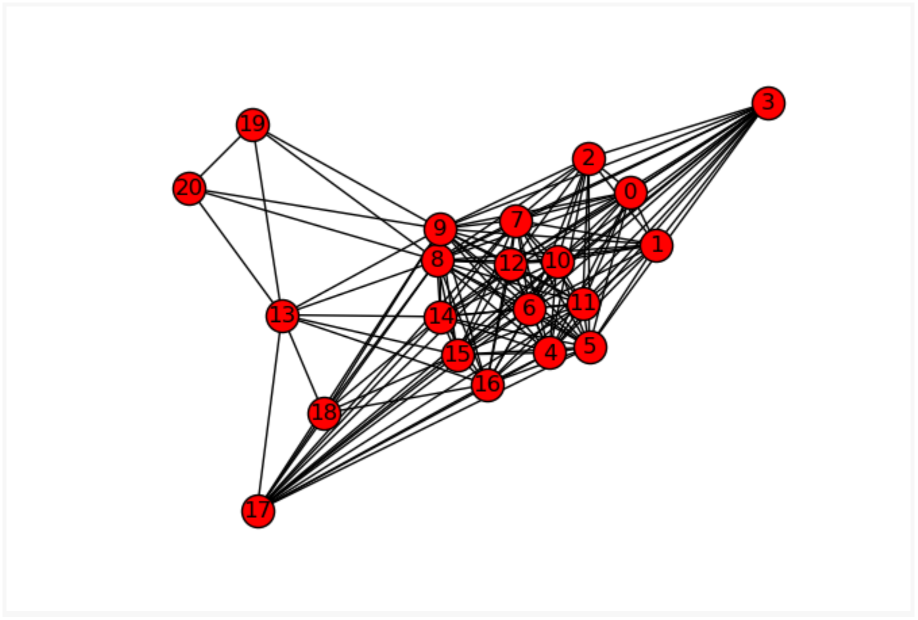
Intervals of travel: A-C : 3 people, A-D : 5 people, A-H : 2 people, B-E : 3 people, E-G : 1 person, C-F : 4 people, E-F : 1 person, F-H : 2 people

To solve this scheduling problem, we first need to model the scheduling problem as a graph problem. Note that passengers on A-F and F-H do not overlap seats. i.e. The latter characters are not considered for group overlapping because they are the arrival points. Because passengers belonging to the same group can not sit in the same seat, the same group of origin and destination is modeled as a complete subgraph. If there is one overlap in the line, even if it is another group, then each passengers must be seated in a different seat in this section. Therefore, in this case, the edges of all pairs of passenger nodes belonging to each group should be added. If the two groups do not overlap in the line, we do not add an edge between them. The modeling function is implemented as follows.

```
from networkx.utils import accumulate
import itertools

def schedule_graph(block_range, *block_sizes):
    G = nx.empty_graph(sum(block_sizes))
    extents = zip([0] + list(accumulate(block_sizes)), accumulate(block_sizes))
    blocks = [range(start, end) for start, end in extents]

    ls = []
    for i in range(len(block_range)):
        ls.append((block_range[i], blocks[i]))

    for (i, block) in enumerate(blocks):
        G.add_nodes_from(block, block=i)

    for block1, block2 in itertools.combinations(ls, 2):
        if len(block1[0].intersection(block2[0])) != 0:
            G.add_edges_from(itertools.product(block1[1], block2[1]))

    for block in ls:
        G.add_edges_from(itertools.product(block[1], block[1]))
    G.name = 'schedule_graph{0}'.format(block_sizes)
    return G
```

Depending on the above function and the condition of the problem, we construct a graph model to solve the scheduling problem.

```
travel_list = []
travel_list.append((set(range(1, 3)), 3))
travel_list.append((set(range(1, 4)), 5))
travel_list.append((set(range(1, 8)), 2))
travel_list.append((set(range(2, 5)), 3))
travel_list.append((set(range(5, 7)), 1))
travel_list.append((set(range(3, 6)), 4))
travel_list.append((set(range(5, 6)), 1))
travel_list.append((set(range(6, 8)), 2))
G = schedule_graph([item[0] for item in travel_list], *[item[1] for item in travel_list])
nx.draw(G, pos=nx.spring_layout(G), with_labels = True)
plt.show()
```



It is somewhat difficult to verify, but it was created to suit the problem condition.

Finally, we obtain the chromatic number of this graph by the function implemented in 2). This chromatic number will be the minimum number of seats required to answer the question. Notice that the answer is 14.

```
memo_ls = {}
memo_return = {}
memo_index = 0

start_time = time.time()
degree = graph_chromatic_num(G, len(G.nodes()))
print(degree)
print(chromatic_check(G, degree))
print("--- %s seconds ---" % (time.time() - start_time))
```

```
[                   0   4142712397824000 -18009337380249600
  35379877853491200 -42097637788904448  34215000544592640
 -20273140331750304   9114943300069584  -3192212706340592
    886236192572184   -197300676211418     35467521191453
     -5163784384616       608507801213       -57781765852
         4381994042         -261536200           12005962
            -408826               9721               -144
                  1]
14
--- 13.3116710186 seconds ---
```

Conclusion

I have appropriately solved all the parts of the project problem that were given in the class, which are in the part 'Scheduling problem, chromatic number'. I could find the worst case of greedy coloring algorithm with random vertex ordering. The chromatic polynomial of the graph was found explicitly using the Fundamental Reduction Theorem and the chromatic number was calculated using this. Some of the processes have been optimized to be reasonably computationally efficient. In addition, we solved one scheduling problem by using graph modeling.

However, there are a lot of problems that can be improved a little more technically. First, if you are doing a reduction and number of the graph component is not one, it would be computationally more efficient to divide the component apart. Second, it seems that the computation is parallelized using OpenMP and MPI. However, due to the nature of the dynamic programming technique, attention should be paid to the mixed use of the shared memory type system and the message passing type system. Third, in the second problem, when n becomes much larger, you will have to deal with overflow because of the huge coefficients.

And all the code I wrote for this project is available here :
https://github.com/mikigom/mikigom_course/blob/master/EC4207/EC4207_project.ipynb

Biography

Junghoon Seo is a B.S student in GIST College, Gwangju Institute of Science and Technology, Republic of Korea. He is now majoring in Electrical Engineering and Computer Science. His current research interests includes deep learning, image processing, recommender system, and natural language processing. Mr. Seo is a student member of KIISE and a member of KIISE Artificial Intelligence Society.