Subject : Computational Statistics

Date : 2022.12.12

Final Report

A comparison study on the Graph Coloring Algorithms

학번	212STG34	이름	강채리
학번	212STG35	이름	노예림
학번	212STG36	이름	백나림
학번	212STG40	이름	유찬미

목 차

Abstract

NP-Complete 문제 중 하나인 graph coloring 문제는 그래프에서 인접한 꼭짓점이 같은 색을 갖지 않도록 하면서 최소한의 색깔로 칠하는 문제이다. 그래프의 꼭짓점 또는 연결된 선이 증가할수록 복잡성이 증가한다. 본 보고서에서는 graph coloring을 다루는 알고리즘들을 비교하고 그 중 두 가지를 활용해 2022년도 2학기 이화여대 통계학과 시험시간표를 배정할 것이다.

I. Introduction

1. NP-Probelm 이란?

P 문제	NP 문제
다항 시간(polynomial time)에 결정론적(deterministically)으로 해결 가능한 문제들의 집합	다항 시간(polynomial time)에 비결정론적(nondeterministically)으로 해결 가능한 문제들의 집합

Table1. NP problem

NP 문제란, 정확한 해를 찾을 수 있는 방법은 없어도 추측할 수 있는 문제를 의미한다. NP 문제 중에서도 다항 시간내에 풀 수 없는 문제를 NP-hard 문제로 칭한다. 또한, NP-hard 이면서 NP문제인 경우, NP-complete이라고 한다.

NP-complete 문제로는 최적의 경로로 이동하는 영업사원 문제, 모든 꼭짓점을 한 번씩 지나는 해밀턴 경로 문제, 그래프 컬러링 문제가 있다.

2. Graph Coloring 문제란?

그래프 컬러링 문제는 Francis Gurthries가 영국 지도의 모든 국가를 채색하려는 시도에서 시작되었다. 그는 4-color theorem을 만들었는데 이는 4가지 색상이 세계지도를 처리하는데 충분하다는 것이다. 1870년대 후반부터 다양한 수학적 증명이 등장하다 1970년대에 이르러 컴퓨터를 통해 수학적으로 증명되었다.

그래프 컬러링 문제에서는 그래프에서 서로 인접한 꼭짓점이나 가장자리가 같은 색을 갖지 않도록 하면서 최소한의 m개의 다른 색으로 칠하는 모든 방법을 찾는 것이 목표이다. 이 문제는

그래프의 꼭짓점 또는 가장자리 수가 증가함에 따라 문제의 복잡성도 증가한다. 이 때문에 각기 다른 알고리즘은 최적의 색깔 수와 실행 시간이 다를 수 있다.

Definition 1

an undirected graph $G(V,E) is\ a\ function$

$$\emptyset = V \rightarrow \{1, \ldots, k\}$$

that assign distinct values to adjacent verticies

If G has a k-colouring then it is said to be k-colourable.

여기서 V는 vertex(= node, 점)의 개수이고 E는 edge(연결선)의 개수이다.

그래프 컬러링에서 색을 지정하는 것을 레이블링으로 바꿔 생각한다면 실생활의 다양한 분야에 적용이 가능하다. 스도쿠 퍼즐 풀기, 레지스터 할당, 주파수 할당, 패턴일치 및 스케줄링 등에 다양하게 적용되고 있다.

II. Algorithms

알고리즘 종류	정확하게 해를 찾는 알고리즘	최적화하는 근사 알고리즘
solution	해 (solution)	적합한 해(feasible solution)
examples	Backtracking	Greedy, Antcol

Table2. Graph Coloring Algorithm

그래프 컬러링의 대표적인 알고리즘에는 greedy, Antcol, backtracking이 있다. greedy와 Antcol은 최적화 근사 알고리즘에 속하고 backtracking은 모든 경우의 수를 탐색하는 알고리즘이다. backtracking은 노드 개수가 적을 때, 연산이 어렵지 않지만 노드 개수가 증가함에 따라 연산속도가 기하급수적으로 증가한다. 예를 들어, 그래프의 노드 수가 100이고 색상 수가 20이면, 시간 복잡성이 최대 $O(20^{100})$으로 긴 시간을 필요로 한다. 따라서 이 프로젝트에서는 처음으로 해가 나오면 해당 해를 사용하는 방식으로 변형하여 사용했다.

1. Greedy Algorithm

Step 1. 모든 노드에 번호를 매긴다.

Step 2. 1번 노드에 대해 랜덤으로 색상을 할당한다.

Step 3. 순서대로 해당 번호의 노드에 색을 할당하는데, 인접 노드에는 다른 색을 칠하도록 한다.

임의로 생성되는 초기 순열을 사용한다. 각 노드에 색상이 지정되어 충돌을 일으키지 않는지를 인스턴스에서 비교한다. 노드가 색칠되기 전에 각 색상을 확인한다고 가정하면 시간 복잡도는 $O(N^2)$이다.

Algorithm 1 Greedy Algorithm

Require: S = Class Set, V =Colorless vertices in random order.
Ensure: $S = \emptyset$
  for v in V **do**
    for i to $S.length()$ **do**
      if $NonConflitiveEdges(v \cup S_i)$ **then**
        $AssignClass(v, S_i)$
        Next Vertex (break)
      end if
    end for
    if $NotColored(v)$ **then**
      $S_{i+1} = NewClass$
      $AssignClass(v, S_{i+1})$
      Next Vertex
    end if
  end for

Figure 1. Sudo code for greedy algorithm[1]

2. Ant colony optimization(ACO)

조합 최적화 문제를 해결하기 위한 메타 휴리스틱 접근 방식이다. 실제 개미의 채집 행동에서 영감을 얻은 진화적 방법으로, 먹이와 둥지 사이 최단경로를 찾을 수 있다. 각 단계에서 이전 반복에서 얻은 정보와 각 요소의 적합성을 고려한다.

Step 1. 개미들은 무작정 개미집 주변을 돌아다닌다.

Step 2. 만약 음식을 찾아내면 개미는 페로몬을 뿌리며 집으로 돌아온다.

Step 3. 페로몬은 매우 매력적이어서 개미가 따라가고 싶게 만든다.

Step 4. 개미가 집으로 돌아오는 횟수가 많을수록 그 경로는 더 견고해진다.

Step 5. 긴 경로와 짧은 경로가 있으면 같은 시간에 짧은 경로로 이동할 수 있는 횟수가 많다.

Step 6. 짧은 경로는 갈수록 더 많은 페로몬이 뿌려지면서 더욱 견고해진다.

[1] J. Postigo, J. Soto-Begazo, V. R. Fiorela, G. M. Picha, R. Flores-Quispe and Y. Velazco-Paredes, "Comparative Analysis of the main Graph Coloring Algorithms," 2021 IEEE Colombian Conference on Communications and Computing (COLCOM), 2021, pp. 1-6

Antcol 알고리즘에서 두 가지 추가적인 조건이 있다. 첫번째는 페로몬이 시간에 따라 증발한다는 "페로몬 증발"이라는 인공적인 조건으로 망각방법으로 사용된다.
그리고 두번째는 특정 솔루션에 대해 개미의 빈도를 높이기 위해 일반적으로 가장 효과적이라고 생각되는 경우 개미의 빈도를 높이기 위해 인공적으로 페로몬이 추가된다.

일반적으로 이 방법은 local optimizer인 TABU search 알고리즘과 함께 작용한다. TABU search 알고리즘은 local minimum에 대한 정보를 tabu, 즉 금지조건으로 저장하여, 이를 회피하여 global minimum을 찾는 알고리즘 이다. Antcol은 tabu 조건을 활용하여 개미의 움직임에 금지제한을 주어, 최상의 솔루션으로 이동하게 만든다.

```
Begin ACS1

Initialization; (Pheromone and parameters) While (stop
criterion not satisfied) do
Position ants on starting vertices Repeat
 For (each ant) do
       Choose a vertex to color by applying the
         transition rule (*); Tabu
         list Update;

         Update Online stage by stage of pheromone trails

     End for

   Until (each ant construct a solution)
  Select the best solution ;

  Pheromone Offline update of the best solution
  Evaporation if necessary

 End While End
ACS1
```

Figure2. sudo code for Antcol[1]

AntCol은 "다중 집합" 연산자를 이용한다. 색상 클래스의 구성 프로세스는 확률적이기 때문에, 연산자는 순차 검색을 활용하여 각 색상 클래스를 구성하기 위해, $O(N^3)$의 복잡성을 갖는 시도를 한다.

3. Backtracking

Step 1. level 1에서 노드에 가능한 모든 색을 각각 시도한다.

Step 2. level 2에서 노드에 가능한 모든 색을 각각 시도한 후 가지치기한다.

*가지치기: 인접노드에 같은 색이 칠해진 경로는 더 이상 고려하지 않고 걸러내는 것.

Step 3. 전체 level에 대해 반복한 후 해답을 결정한다.

```
def m_coloring(i):
    global count # 현재 발견한 solution 개수
    global m # 현재 사용 색깔 개수
    # 탐색
    if promising(i):
        if i == n: # 끝까지 탐색 -> 정답
            count+=1
        else:
            for color in range(1, m + 1): # 모든 가능 색깔 탐색
                vcolor[i + 1] = color
                m_coloring(i + 1) # 다음 재귀로 넘어가고 불가능하다면 backTracking



def promising(i):
    j = 1 #SST를 탐색
    flag = True

    while j < i and flag:
        if W[i][j] and vcolor[i] == vcolor[j]: #W[i][j]가 연결되어 있으며, 두 노드의 색깔이 같다면
            flag = False
        j += 1
    return flag
```

Figure 3. code for Backtracking algorithm

본 프로젝트에서는 backtracking을 원래 작동 방식대로 이용하지는 않았고 처음으로 해가 나오면 해당 해를 사용하는 방식으로 변형하였다. sudo 코드에서 promising 부분이 가지치기를 하는 코드이다. 한 노드에 대해 모든 색(m개)를 시도해야 하므로 시간 복잡도는 $O(M^N)$이다. 하지만 가지치기를 통해 상태노드의 수를 줄일 수 있으므로 알고리즘의 효율을 더 좋을 수 있다.

III. Simulation

Simulation 1. 간단한 그래프

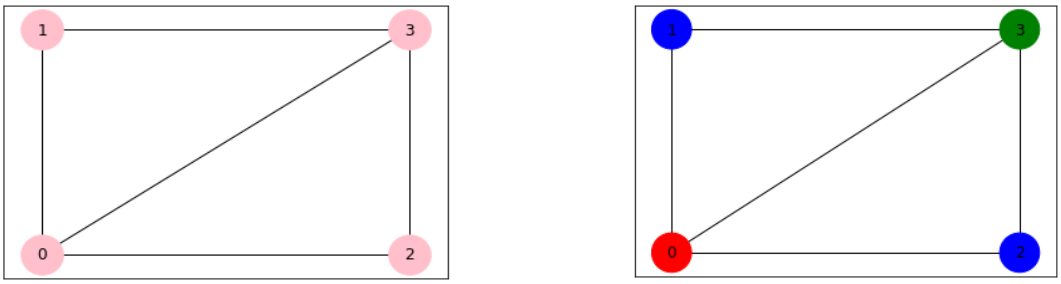

Figure4. n=4 graph

n=4	greedy	Antcol	backtracking
Number of Colors	3	3	3
Iterations	4	1	1
Computation time	0.0000 sec	0.0002 sec	0.0008 sec

Table3. n= 4 graph comparison results

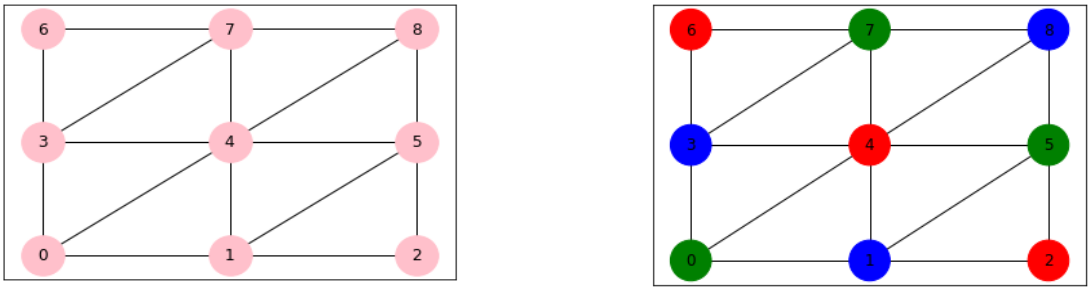

Figure5. n=9 graph

n=9	greedy	Antcol	backtracking
Number of Colors	3	3	3
Iterations	9	1	1
Computation time	0.0000 sec	0.0303 sec	0.0011 sec

Table4. n=9 comparison results

첫번째 시뮬레이션은 가장 간단한 그래프, 즉 노드의 수가 각각 4와 9인 경우에 대해 비교했다. 각 시뮬레이션의 경우 5번 반복측정한 후, 평균값을 제시했다.

노드가 4인 경우 두 가지 색으로 문제를 푸는 건 불가능하다. 세가지 색을 사용한다면 총 6개의 해답을 얻을 수 있다. 그 중 하나의 답안이 오른쪽 그래프이다. 각 알고리즘 모두 정답을 찾아 냈으며, greedy알고리즘이 가장 빠른 속도를 보여준다.

노드가 9인 경우 세 가지 색으로 모든 노드를 색칠할 수 있다.

Simulation 2. 연결선 증가(edge 개수)

두번째 시뮬레이션에서는 노드를 연결하는 edge, 즉 a를 증가시키면서 복잡한 그래프에서 각 알고리즘이 어떻게 작동하는지를 비교한다.

각 그래프는 인접행렬의 상삼각부분에서 0,1을 랜덤하게 생성한 경우로, 0 과 1의 비율이 2:1,1:1, 1:2, 1:3이 되도록 4가지 케이스를 만들었다. 이때 a는 약 1600개, 2500개, 3700개, 4000개가 생성된다.

*backtracking에서 컬러 개수를 Antcol 결과보다 1개 줄여 13개로 지정해서 알고리즘을 돌린 결과, 시간은 25초정도 소요되었다. 즉, backtracking이 Antcol 보다 최적에 가까운 값을 찾을 수는 있지만, 연산시간이 2배 이상 걸린다. 그래프가 복잡해지면, backtracking처럼 전체 경우의 수를 고려하는 알고리즘은 연산시간이 너무 커서 실제 상황에 적용하기 어렵다는 것을 확인할 수 있다.

n = 100, a = 1638	greedy	Antcol	backtracking
Number of Colors	15	14	16
Computation time	0.0118	12.4948	0.0229

n = 100, a = 2467	greedy	Antcol	backtracking
Number of Colors	19	18	21
Computation time	0.0128	13.3168	0.014

n = 100, a = 3747	greedy	Antcol	backtracking
Number of Colors	33	29	34
Computation time	0.0146	14.7023	0.0275

n = 100, a = 4124	greedy	Antcol	backtracking
Number of Colors	38	34	42
Computation time	0.0158	15.0701	0.0402

Figure5. comparison results depending on a(number of edges)

a가 증가할수록 세 알고리즘 모두 시간은 오래걸리고, 이용한 색상의 수도 증가한다. 네가지 경우 모두 Antcol이 가장 적은 개수를 찾아내지만 오랜 시간이 걸린다. greedy는 비교적 짧은 시간내에 적절한 해를 찾아낸다.

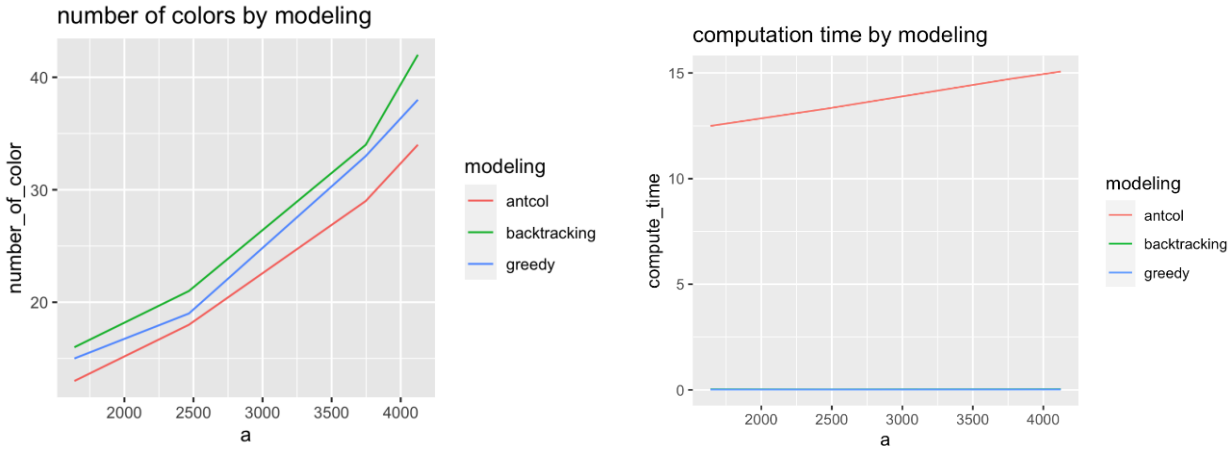

Figure6. number of color and computation time depending on a(number of edges)

왼쪽 그래프에 따르면 연결선(edge, a)가 증가할 수록, 세가지 알고리즘 모두 사용하는 색깔 수가 급격하게 증가한다. 이때, Antcol의 색상 수가 가장 적었다.

오른쪽 그래프에 따르면, Antcol은 다른 두 알고리즘보다 훨씬 많은 시간이 소요됨을 확인할 수 있다. backtracking과 greedy는 연결선이 늘어나도 연산시간이 크게 늘어나지 않는다.

Simulation 3. 그래프 크기 증가(node, a 개수)

세 번째 시뮬레이션에서는 노드와 연결선(a, 즉 edge)가 모두 증가하는 경우 알고리즘을 비교하였다. 노드 개수를 100, 700, 900, 1000인 경우를 각 알고리즘으로 테스트를 수행하고 비교해본 결과를 표에 정리했다.

n=100, a=1638	greedy	Antcol	backtracking
Number of Colors	15	14	16
Computation time	0.0118	12.4948	0.0229

n=500, a=41959	greedy	Antcol	backtracking
Number of Colors	48	43	49
Computation of times	0.8166 sec	1456.0349	0.4818

n=800, a=107163	greedy	Antcol	backtracking
Number of Colors	68	63	69
Computation of times	3.1208 sec	6214.8627	1.5928 sec

n=1000, a=167232	greedy	Antcol	backtracking
Number of Colors	83	74	83
Computation of times	6.2304 sec	12200.6978sec	3.0254 sec

Table6. comparison results depending on the number of nodes and edges

노드와 연결선이 많아질수록 컬러 수와, 계산 시간이 빠르게 증가한다. 전반적으로 Antcol이 가장 오랜 시간이 걸리지만, 최적의 컬러 수를 찾는다. 반면에, backtracking과 greedy는 매우 빠른 시간안에 답을 찾는 대신, 사용한 색상의 수가 많다.

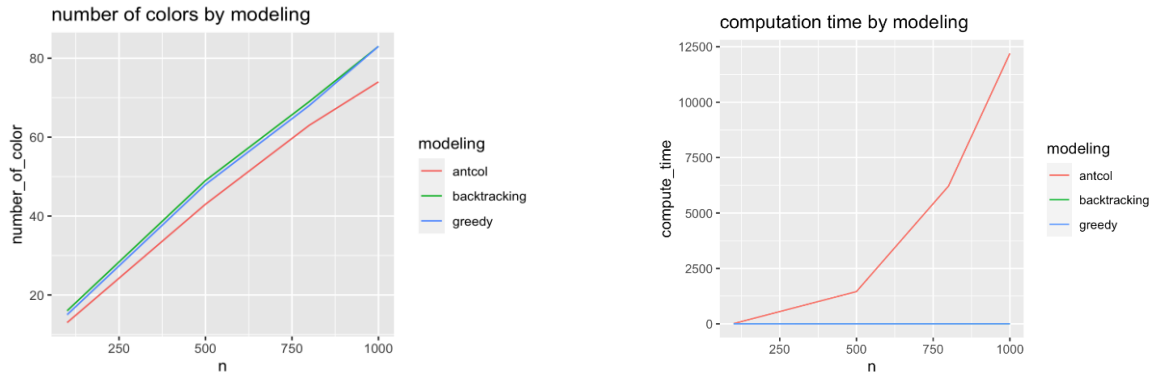

Figure7. number of color and computation time depending on nodes and a(number of edges)

Figure7의 왼쪽은 그래프의 크기(노드 개수)에 따라 각 알고리즘에서 사용한 최소 색상의 수를 나타낸 그래프이고, 오른쪽 그래프는 그래프가 크기에 따른 알고리즘의 계산시간이다.
왼쪽 그래프를 보면, 그래프 크기가 증가할수록 3가지 알고리즘 모두 사용하는 색깔의 수는 증가한다.

연산 시간 그래프에서, Antcol의 경우 그래프 크기가 증가할 수록 연산시간이 급격하게 증가함을 확인할 수 있다. 다른 두 알고리즘의 경우 그래프 크기에 따라 연산시간이 크게 변하지 않았고, 가장 복잡한 경우에도 소요시간이 10초 이내이다.

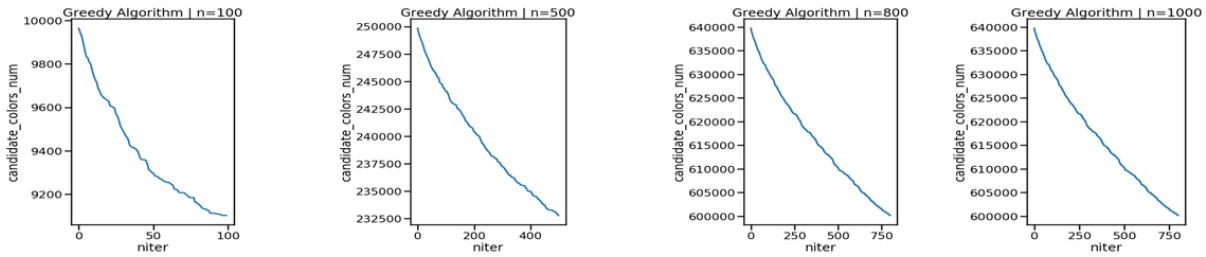

Figure8. converge graph for greedy algorithm

Figure8에 따르면, 그래프 크기가 클 때, Greedy 와 Antcol 모두 수렴했다고 보기 어려우므로 개선의 여지가 있다.

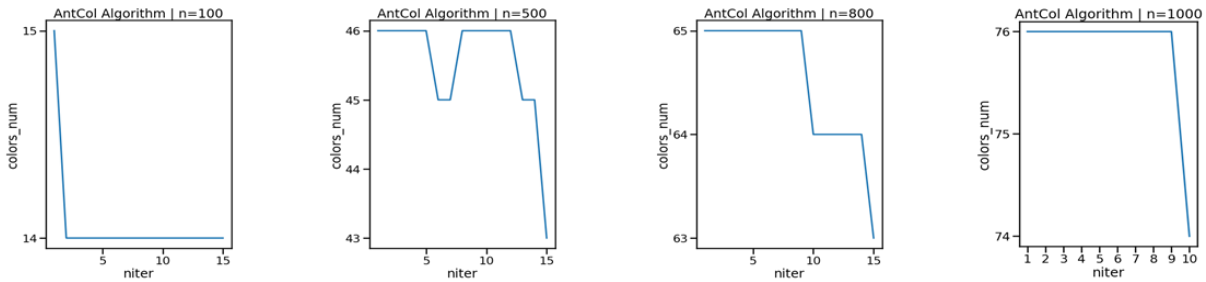

Figure9. converge graph for Antcol algorithm

Figure9에 따르면 Antcol 이 greedy 보다 안정적으로 수렴하는 경향이 있다.

Simulation results

시뮬레이션 결과, 최적의 optimal한 값을 찾고 싶은 경우엔 Antcol, 빠른 시간내에 적당한 값을 찾고 싶은 경우에는 greedy 알고리즘이 적절함을 발견했다.

Antcol 알고리즘은 최적의 값, 즉, 최소의 색깔 수를 찾는다. 하지만 다른 알고리즘에 비해 까다로워서 연산시간이 긴 편이다. 반면에 greedy 알고리즘의 장점은 연산시간이 짧다는 것이다. 하지만 greedy는 최적의 답을 찾지는 않았다.

IV. Application

1. Timetable problem

graph coloring을 적용할 수 있는 문제 중에는 timetabling 문제가 있다. timetable 문제는 제한 조건하에서 가능한 time slot에 수업 등을 배분하는 것이다. 간호사의 shift 시간표를 배분하거나 대학교 시험시간표를 작성하는 문제 등이 있다. 그 중 저희는 시험시간표 작성하는 문제에 그래프 컬러링을 적용해보았다.

노드를 수업, 엣지를 수업 간의 연결된 선으로 보고 graph coloring를 적용할 수 있다. 동시간에 시험을 치를 수 없는 수업을 선으로 연결해서 인접한 두 노드의 색을 겹치지 않게 칠하도록 조건을 추가한다. 시간표 문제에서는 제한조건으로 하드와 소프트 두가지 종류가 있는데
hard 제한은 무조건 만족해야 하는 조건이고 soft 조건은 무조건 만족할 필요는 없지만 만족한다면 좋은 조건이다.

2. Constraints

본 프로젝트를 통해 이화여대 통계학과 시험시간표를 배정해보았다.
통계학과 시험은 보통 1시간 반 안에 끝나는 경우가 적어서 수업시간 내에 치루지 못하고 따로 저녁이나 주말에 시험을 치르는 경우가 있다. 이때 모든 학생들의 시간을 맞추기가 어렵다는 문제가 있어서, 종종 어려움을 겪는다. 2022년도 2학기 통계학과 총 25개 수업을 최소 색깔로, 그러니까, 최소한의 time slot으로 시간표를 배정하는 것이 목표이다.

제한 조건은 다음과 같다.
먼저 hard 조건은 한 명의 교수님이 동시간에 여러 시험장에 계실 수 없다. 즉, 교수님이 동일한 수업은 시험시간이 겹치면 안 된다.
두번째, 데이터 과학자 트랙을 수강하는 학생이 있다. 즉, 기초확률론, 회귀분석, 수리통계학2, 학부 데이터마이닝, 통계프로그래밍을 필수로 수강하는 학생이 있다. 그러므로 해당하는 5가지 과목은 동시간에 시험을 치를 수 없다.

soft 조건은 다음과 같다. 첫번째, 점심시간에는 쉬어야 하기 때문에, 시험을 치지 않는다.
두번째, 보통 한 학년의 수업을 한 번에 듣는 경우가 많으니 각 학년의 수업은 시험시간이 겹치지 않는 것이 좋다.
세번째, 원래 수업 요일에 시험을 치르면 좋다.

이러한 조건들을 반영하여 시험 시간표를 배정해보았다.

3. 이화여대 통계학과 시험시간표 배정

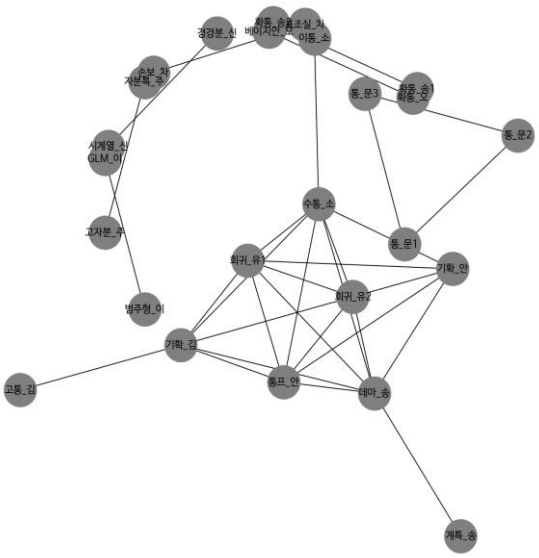

Figure10. network graph

Figure10은 통계학과 25개 수업을 노드로, 동시간에 치를 수 없는 경우 선으로 연결한 network graph이다.

Algorithm	System time	색 개수	결과 (25개 수업명을 알파벳으로 간략하게 표시)
greedy	0.028	6	색0: [a,d,e,h,j,l,n,p,v,y] 색1: [b,f,g,i,k,m,o,w,x] 색2: [c,q,r] 색3: [s] 색4:[t] 색5:[u]

Antcol	0.3251	6	색0: [a,d,f,h,j,l,n,p,v,y] 색1: [b,e,g,i,k,m,o,q,r,w] 색2: [c,t] 색3: [x] 색4:[u] 색5:[s]
backtracking	0.0168	7	색0: [a,d,e,h,j,l,n,p,r,w,y] 색1: [b,f,g,i,k,m,o,q] 색2: [c,s] 색3: [t] 색4:[u] 색5: [v] 색6: [x]

Table7. graph coloring comparison results

greedy와 Antcol은 모두 6개의 색으로 25개의 수업을 색칠했고, backtracking은 7개의 색으로 칠했다.
시간은 n이 크지 않아서 비슷하나, Antcol이 가장 오래 걸렸다.

이 결과를 토대로 greedy와 Antcol을 이용해 이화여대 통계학과 시험시간표를 배정해보았다.

1) greedy로 시간표 배정한 결과

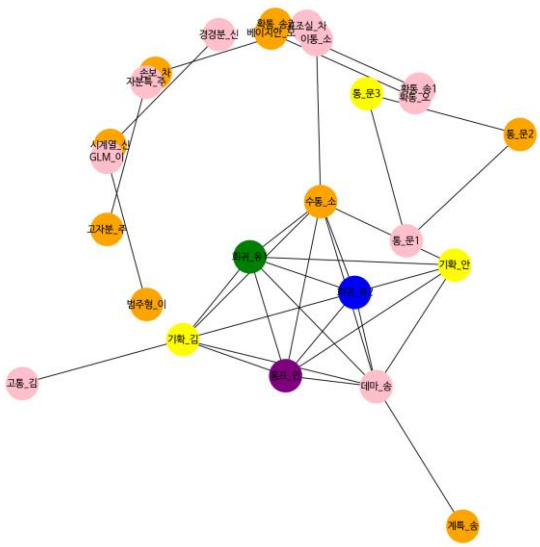

Figure11. network graph colored by greedy algorithm

color 0	1	2	3	4	5
a 통2 1분반 (문온교수님)	b 통2 2분반 (문온교수님)	c 통2 3분반 (문온교수님)	s 회귀분석1분반 (유재근교수님)	t 회귀분석 2분반 (유재근교수님)	u 통계프로그래밍 (안재윤교수님)
d 확통1분반 (오만숙교수님)	F 확통3분반 (송수민교수님)	q 기초확률론 (김경원교수님)			
E 확통2분반 (송수민교수님)	g 베이지안 (오만숙 교수님)	r 기초확률론 (안재윤교수님)			
h 표본조사실습 (차지환교수님)	i 손해보험론 (차지환교수님)				
J 경자분 (신동완교수님)	k 시계열 (신동완교수님)				
l 자료분석특론 (주원영교수님)	m 고급자료분석 (주원영교수님)				
n GLM (이동환교수님)	o 범주형 (이동환교수님)				
p 고급통계 (주원영교수님)	w 통계계산특론 (송종우교수님)				
v 데이터마이닝 (송종우교수님)	x 수리통계2 (소병수교수님)				
Y 이론통계2 (소병수교수님)					

Table8. greedy algorithm results

다른 색이 칠해진 수업은 동시간에 시험을 치를 수 없다. 같은 색이 칠해진 수업들은 동시간에 치를 수 있다.

다음 단계에서는 세가지 soft 조건을 고려한다.

첫번째 soft 조건, 점심시간에는 시험을 치르지 않는다. 11시부터 12시까지를 점심시간으로 두고 해당시간에는 시험을 치르지 않는다.

두번째 soft 조건, 보통 한 학년의 수업을 동시에 듣는다는 것을 반영해서 학년별로 시간표를 만든 것이 Table9다. 1학년의 경우, 오전에 3과목, 오후에 2과목, 저녁에 1과목이 배정됐다.

그리고 원래의 수업 요일에 시험을 치르는 것이 좋다는 세번째 soft 조건까지 모두 만족한 최종 결과는 아래 Table10와 같다.

1st year	수업명
8:00 a.m~11:00a.m	확통 1분반(오만숙교수님) 확통 2분반(송수민교수님) 통2 1분반 (문온교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	통2 2분반 (문온교수님) 확통 3분반(송수민교수님)
15:00p.m~18:00p.m	통2 3분반 (문온교수님)

2nd year	수업명
8:00 a.m~11:00a.m	회귀분석 1분반(유재근교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	회귀분석 2분반(유재근교수님)
15:00p.m~18:00p.m	기초확률론(안재윤교수님) 기초확률론(김경원교수님)

3rd year	수업명
8:00 a.m~11:00a.m	데이터마이닝(송종우교수님) 표본조사실습(차지환교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	수리통계학2(소병수교수님)
15:00p.m~18:00p.m	통계프로그래밍 (안재윤교수님)

4th year	수업명
8:00 a.m~11:00a.m	경영경제자료분석 (신동완교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	범주형(이동환교수님) 고급자료분석 (주원영교수님)
15:00p.m~18:00p.m	

Grad school	수업명
8:00 a.m~11:00a.m	고급통계(김경원교수님) 이론통계2 (소병수교수님) GLM(이동환교수님) 자료분석특론(주원영교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	계산특론(송종우교수님) 시계열(신동완교수님) 베이지안(오만숙 교수님) 손해보험론(차지환교수님)
15:00p.m~18:00p.m	

Table9. Exam Schedule by year

soft 조건을 고려하였을 때, 1학년 시간표는 아래와 같다.

1st year	월	화	수	목	금
8:00 a.m~11:00a.m		확통 1분반 (오만숙교수님)	통계학2 1분반 (문온교수님)	확통 2분반 (송수민교수님)	
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	통계학2 2분반 (문온교수님)	확통 3분반 (송수민교수님)			
15:00p.m~18:00p.m	통계학2 3분반 (문온교수님)				

2학년 시간표는 아래와 같다.

2nd year	월	화	수	목	금
8:00 a.m~11:00a.m					회귀분석 1분반 (유재근교수님)
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m					회귀분석 2분반 (유재근교수님)
15:00p.m~18:00p.m	기초확률론 (안재윤교수님)	기초확률론 (김경원교수님)			

3학년 시간표는 아래와 같다.

3rd year	월	화	수	목	금
8:00 a.m~11:00a.m	표본조사실습 (차지환교수님)			데이터마이닝 (송종우교수님)	
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m		수리통계학2 (소병수교수님)			
15:00p.m~18:00p.m					통계프로그래밍 (안재윤교수님)

4학년 시간표는 아래와 같다.

4th year	월	화	수	목	금
8:00 a.m~11:00a.m		경영경제자료분석 (신동완교수님)			
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	고급자료분석 (주원영교수님)		범주형 (이동환교수님)		
15:00p.m~18:00p.m					

대학원생 시간표는 아래와 같다.

Grad school	월	화	수	목	금
8:00 a.m~11:00a.m	고급통계 (김경원교수님)	GLM (이동환교수님)	이론통계2 (소병수교수님)	자료분석특론 (주원영교수님)	
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	통계계산특론 (송종우교수님)	시계열 (신동완교수님)	손해보험론 (차지환교수님)	베이지안 (오만숙 교수님)	
15:00p.m~18:00p.m					

Table10. Exam schedule for a week

예를 들어, 데이터 과학자 트랙을 신청한 학부생의 경우 이번학기 시간표는 다음과 같다.

	19일 (월)	20일 (화)	21일 (수)	22일 (목)	23일 (금)
8:00 a.m~11:00a.m				데이터마이닝 (송종우교수님)	회귀분석 1분반 (유재근교수님)
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m		수리통계학2 (소병수교수님)			
15:00p.m~18:00p.m	기초확률론 (안재윤교수님)				통계프로그래밍 (안재윤교수님)

Table11. Exam scehdule for a student who applied for 'Data Scientist Track'

최종적으로 전체 통계학과 시간표는 다음과 같다.

	월 19일	화 20일	수 21일	목 22일	금 23일
8:00 a.m~11:00a.m	표본조사실습 (차지환교수님) 고급통계 (김경원교수님)	확통 1분반 (오만숙교수님) 경영경제자료분석 (신동완교수님) GLM (이동환교수님)	통계학2 1분반 (문온교수님) 이론통계2 (소병수교수님)	확통 2분반 (송수민교수님) 데이터마이닝 (송종우교수님) 자료분석특론 (주원영교수님)	회귀분석 1분반 (유재근교수님)
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	통계학2 2분반 (문온교수님) 고급자료분석 (주원영교수님) 통계계산특론 (송종우교수님)	확통 3분반(송수민교수님) 수리통계학2 (소병수교수님) 범주형 (이동환교수님) 시계열 (신동완교수님)	손해보험론 (차지환교수님)	베이지안 (오만숙 교수님)	회귀분석 2분반 (유재근교수님)
15:00p.m~18:00p.m	통계학2 3분반 (문온교수님) 기초확률론 (안재윤교수님)	기초확률론 (김경원교수님)			통계프로그래밍 (안재윤교수님)

Table12. Exam schedule for the department of Statistics

같은 과정을 Antcol 알고리즘을 통해서도 진행하였다.

2) Antcol로 시간표 배정한 결과

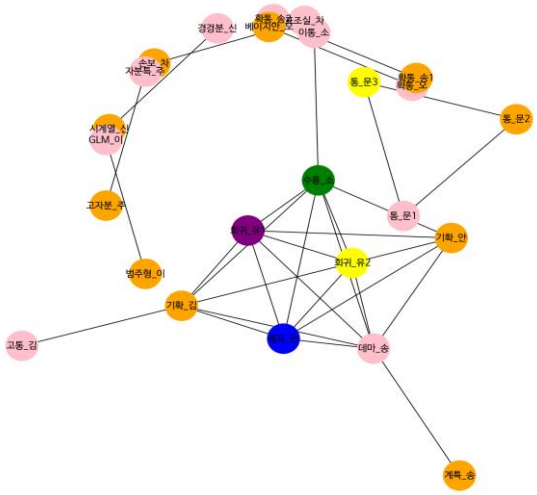

Figure12. network graph colored by Antcol algorithm

0	1	2	3	4	5
a 통계학 1분반 (문온교수님)	b 통계학 2분반 (문온교수님)	c 통계학 3분반 (문온교수님)	x 수리통계학2 (소병수교수님)	u 통계프로그래밍 (안재윤교수님)	s 회귀분석 1분반 (유재근교수님)
d 확통 1분반 (오만숙교수님)	e 확통 2분반 (송수민교수님)	t 회귀분석 2분반 (유재근교수님)			
f 확통3분반 (오만숙교수님)	g 베이지안 (오만숙교수님)				
h 표본조사실습 (차지환교수님)	i 손해보험론 (차지환교수님)				
j 경자분 (신동완교수님)	k 시계열 (신동완교수님)				
l 자료분석특론 (주원영교수님)	m 고급자료분석 (주원영교수님)				

n GLM (이동환교수님)	o 범주형 (이동환교수님)				
p 고급통계 (김경원교수님)	q 기초확률론 (김경원교수님)				
v 데이터마이닝 (송종우교수님)	r 기초확률론 (안재윤교수님)				
y 이론통계2 (소병수교수님)	w 통계계산특론 (송종우교수님)				

Table13. Antcol algorithm results

1st year	수업명
8:00 a.m~11:00a.m	통계학 1분반(문온교수님) 확통 1분반 (오만숙교수님) 확통 3분반(오만숙교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	통계학 2분반(문온교수님) 확통 2분반(송수민교수님)
15:00p.m~18:00p.m	통계학 3분반(문온교수님)

2nd year	수업명
8:00 a.m~11:00a.m	회귀분석 1분반(유재근교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	기초확률론(김경원교수님) 기초확률론(안재윤교수님)
15:00p.m~18:00p.m	회귀분석 2분반(유재근교수님)

3rd year	수업명
8:00 a.m~11:00a.m	표본조사실습(차지환교수님) 데이터마이닝(송종우교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	수리통계학2(소병수교수님)
15:00p.m~18:00p.m	통계프로그래밍(안재윤교수님)

4th year	수업명
8:00 a.m~11:00a.m	경영경제자료분석(신동완교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	고급자료분석(주원영교수님) 범주형(이동환교수님)
15:00p.m~18:00p.m	

Grad school	수업명
8:00 a.m~11:00a.m	자료분석특론(주원영교수님) GLM(이동환교수님) 고급통계(김경원교수님) 이론통계2(소병수교수님)
11:00a.m~12:00p.m	Lunch
12:00p.m~15:00p.m	베이지안(오만숙교수님) 손해보험론(차지환교수님) 시계열(신동완교수님) 통계계산특론(송종우교수님)
15:00p.m~18:00p.m	

Table14. Exam Schedule by year

1st year	월	화	수	목	금
8:00 a.m~11:00a.m	확통 1분반 (오만숙교수님)	확통 3분반 (오만숙교수님)	통계학 1분반 (문온교수님)		
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	통계학 2분반 (문온교수님)	확통 2분반 (송수민교수님)			
15:00p.m~18:00p.m	통계학 3분반 (문온교수님)				

2nd year	월	화	수	목	금
8:00 a.m~11:00a.m					회귀분석 1분반 (유재근교수님)
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	기초확률론 (안재윤교수님)	기초확률론 (김경원교수님)			
15:00p.m~18:00p.m			회귀분석 2분반 (유재근교수님)		

3rd year	월	화	수	목	금
8:00 a.m~11:00a.m	표본조사실습 (차지환교수님)	데이터마이닝 (송종우교수님)			
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m					수리통계학2 (소병수교수님)
15:00p.m~18:00p.m					통계프로그래밍 (안재윤교수님)

4th year	월	화	수	목	금
8:00 a.m~11:00a.m		경영경제자료분석 (신동완교수님)			
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	고급자료분석 (주원영교수님)		범주형 (이동환교수님)		
15:00p.m~18:00p.m					

Grad school	월	화	수	목	금
8:00 a.m~11:00a.m	고급통계학 (김경원교수님)	GLM (이동환교수님)	이론통계학2 (소병수교수님)	자료분석특론 (주원영교수님)	
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	통계계산특론 (송종우교수님)	시계열 (신동완교수님)	손해보험론 (차지환교수님)	베이지안 (오만숙교수님)	
15:00p.m~18:00p.m					

Table15. Exam schedule for a week

마찬가지로, 데이터 과학자 트랙을 신청한 학부생의 이번학기 시간표는 다음과 같다.

	월 19일	화 20일	수 21일	목 22일	금 23일
8:00 a.m~11:00a.m		데이터마이닝 (송종우교수님)			
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	기초확률론 (안재윤교수님)				수리통계학2 (소병수교수님)
15:00p.m~18:00p.m			회귀분석 2분반 (유재근교수님)		통계프로그래밍 (안재윤교수님)

Table16. Exam scehdule for a student who applied for 'Data Scientist Track'

전체 통계학과 시간표는 다음과 같다.

	월 19일	화 20일	수 21일	목 22일	금 23일
8:00 a.m~11:00a.m	확통 1분반 (오만숙교수님) 표본조사실습 (차지환교수님) 고급통계학 (김경원교수님)	확통 3분반 (오만숙교수님) 데이터마이닝 (송종우교수님) 경영경제자료분석 (신동완교수님) GLM (이동환교수님)	통계학 1분반 (문온교수님) 이론통계학2 (소병수교수님)	자료분석특론 (주원영교수님)	회귀분석 1분반 (유재근교수님)
11:00a.m~12:00p.m	Lunch				
12:00p.m~15:00p.m	통계학 2분반 (문온교수님) 기초확률론 (안재윤교수님) 고급자료분석 (주원영교수님) 통계계산특론 (송종우교수님)	확통 2분반 (송수민교수님) 기초확률론 (김경원교수님) 시계열 (신동완교수님)	범주형 (이동환교수님) 손해보험론 (차지환교수님)	베이지안 (오만숙교수님)	수리통계학2 (소병수교수님)
15:00p.m~18:00p.m	통계학 3분반 (문온교수님)		회귀분석 2분반 (유재근교수님)		통계프로그래밍 (안재윤교수님)

Table17. Exam schedule for the department of Statistics

Reference

Donatello Conte, Giuliano Grossi, Raffaella Lanzarotti, Jianyi Lin, Alessandro Petrini, Analysis of a parallel MCMC algorithm for graph coloring with nearly uniform balancing, Pattern Recognition Letters,Volume 149,2021,Pages 30-36

J. Postigo, J. Soto-Begazo, V. R. Fiorela, G. M. Picha, R. Flores-Quispe and Y. Velazco-Paredes, "Comparative Analysis of the main Graph Coloring Algorithms," 2021 IEEE Colombian Conference on Communications and Computing (COLCOM), 2021, pp. 1-6

Nandhini, V.. (2019). A Study on Course Timetable Scheduling and Exam Timetable Scheduling using Graph Coloring Approach. International Journal for Research in Applied Science and Engineering Technology. 7. 1999-2006.

Leighton FT. A Graph Coloring Algorithm for Large Scheduling Problems. J Res Natl Bur Stand (1977). 1979 Nov-Dec;84(6):489-506.

Appendix

시뮬레이션 코드

Graph Coloring Algorithms Codes

We are going to apply three algorithms to three types of simulations.

1. Greedy

2. Ant Col

3. Backtracking

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import networkx as nx
import random
import time
import itertools
import random

import warnings
warnings.filterwarnings( 'ignore' )

# plot font_size 조정
sns.set_context("poster")

# 색상 팔레트 불러오기
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
color_palette = list(mcolors.BASE_COLORS) + list(mcolors.TABLEAU_COLORS) + list(mcolors.CSS4_COLORS)
# len(color_palette)  # 총 166개의 색상이 존재
# print(color_palette[:15])  # 첫 15개만 확인

## Define Functions

def addEdge(adj, v, w):  # adj: adj. matrix, v: 연결 출발점 vertex, w: 연결 종료점 vertex
```

```
    adj[v].append(w)
    adj[w].append(v)  # Note: the graph is undirected | if the graph is NOT undirected, delete this code.

    return adj

# Assigns colors (starting from 0) to all
# vertices and prints the assignment of colors
def greedyColoring(adj, V):  # adj: adj. matrix, V: # of vertices

    # Make empty list to store the 'result'
    ## why filling all values '-1' : the color starts from 0.
    result = [-1] * V

    # Assign the first color to first vertex
    result[0] = 0

    # A temporary array to store the available colors.
    ## True value of available[cr] would mean that the color(cr) is assigned to one of its adjacent vertices
    available = [False] * V

    # Assign colors to remaining V-1 vertices
    for u in range(1, V):

        # Process all adjacent vertices and
        # flag their colors as unavailable
        for i in adj[u]:
            if (result[i] != -1):
                available[result[i]] = True

        # Find the first available color
        cr = 0
        while cr < V:
            if (available[cr] == False):
                break

            cr += 1

        # Assign the found color
        result[u] = cr

        # Reset the values back to false
        # for the next iteration
```

```
        for i in adj[u]:
            if (result[i] != -1):
                available[result[i]] = False

    # Print the result
    for u in range(V):
        print("Vertex", u, " --->  Color", result[u])

### 0. Define the Function Counting Number Of Items in Dictionary

def CountNumOfItems(d):  # d: dictionary

    values_list = [item for item in d.values()]
    notunique_values_list = list(itertools.chain(*values_list))
#     unique_values_list = set(notunique_values_list)

    return len(notunique_values_list)

### 1. Define Greedy Function

def greedy(G, n): # G: adj. graph, n: # of nodes

    start=time.time()

    # inisiate the name of node
    node = list(range(0,n))
    t_={}

    for i in range(len(G)):
        t_[node[i]] = i

    # count degree of all node.
    degree =[]
    for i in range(len(G)):
        degree.append(sum(G[i]))

    # inisiate the posible color
    colorDict = {}
    for i in range(len(G)):
        colorDict[node[i]]=list(range(0,n))



    # sort the node depends on the degree
```

```
    sortedNode = []
    indeks = []
    colorDict_flow = []

    # use selection sort
    for i in range(len(degree)):
        _max = 0
        j = 0
        for j in range(len(degree)):
            if j not in indeks:
                if degree[j] > _max:
                    _max = degree[j]
                    idx = j
        indeks.append(idx)
        sortedNode.append(node[idx])

    # The main process
    theSolution={}
    iter_num = 0

    for n in sortedNode:
        setTheColor = colorDict[n]
        theSolution[n] = setTheColor[0]
        adjacentNode = G[t_[n]]
        for j in range(len(adjacentNode)):
            if adjacentNode[j]==1 and (setTheColor[0] in colorDict[node[j]]):
                colorDict[node[j]].remove(setTheColor[0])
        # count the iter_num
        iter_num += 1

        # for steps
#       print(colorDict)   ################## greedy 이해하기 위해서 이 부분 확인
        colorDict_flow.append( CountNumOfItems(colorDict) )



    final_color_num = max(theSolution.values())+1
    color_num_flow = colorDict_flow
    iter_time = time.time()-start

    # Print the results
    print("Number of Colors used: ", max(theSolution.values())+1)
    print("Number of Iterations: ", iter_num)
    print("Computation time: ", f"{time.time()-start:.4f} sec")
```

```
#     print("Colors_Flow: ", colorDict_flow)

    # Final Return: 함수의 output
    return_dict ={'theSolution': theSolution, 'final_color_num': final_color_num,
                  'color_num_flow': color_num_flow, 'iter_num': iter_num, 'iter_time': iter_time}

    return return_dict

### 2. Define Antcol Function

class Ant:
    # create new ant
    # alpha: the relative importance of pheromone (si_ij)
    # beta: the relative importance of heuristic value (n_ij)
    def __init__(self, alpha=1, beta=3):
        self.graph = None
        self.colors = {}
        self.start = None
        self.visited = []
        self.unvisited = []
        self.alpha = alpha
        self.beta = beta
        self.distance = 0 # number of used colors on a valid solution
        self.number_colisions = 0 # only for consistency check, should be always 0
        self.colors_available = []
        self.colors_assigned = {}

    # reset everything for a new solution
    # start: starting node in g (random by default)
    # return: Ant
    def initialize(self, g, colors, start=None):
        self.colors_available = sorted(colors.copy())

        # init assigned colors with None
        keys = [n for n in g_nodes_int]
        self.colors_assigned = {key: None for key in keys}

        # start node
        if start is None:
            self.start = random.choice(g_nodes_int)
        else:
            self.start = start
```

```
    self.visited = []
    self.unvisited = g_nodes_int.copy()

    # assign min. color number to the start node
    if (len(self.visited)==0):
        self.assign_color(self.start, self.colors_available[0])
    return self

# assign color to node and update the node lists
def assign_color(self, node, color):
    self.colors_assigned[node] = color
    self.visited.append(node)
    self.unvisited.remove(node)

# assign a color to each node in the graph
def colorize(self):
    len_unvisited = len(self.unvisited)
    tabu_colors = []
    # assign color to each unvisited node
    for i in range(len_unvisited):
        next = self.next_candidate()
        tabu_colors = []
        # add colors of neighbours to tabu list
        for j in range(number_nodes):
            if (adj_matrix[next,j]==1):
                tabu_colors.append(self.colors_assigned[j])
        # assign color with the smallest number that is not tabu
        for k in self.colors_available:
            if (k not in tabu_colors):
                self.assign_color(next,k)
                break
    # save distance of the current solution
    self.distance = len(set(self.colors_assigned.values()))
    # consitency check
    ##self.number_colisions = self.colisions()
    ##print('colisions: ' + str(self.number_colisions))

# return the number of different colors among the neighbours of node
def dsat(self, node=None):
    if node is None:
        node = self.start
    col_neighbors = []
    for j in range(number_nodes):
```

```
        if (adj_matrix[node, j]==1):
            col_neighbors.append(self.colors_assigned[j])
    return len(set(col_neighbors))

# return the pheromone trail of the pair (node,adj_node)
def si(self, node, adj_node):
    return phero_matrix[node, adj_node]

# select next candidate node according to the transition rule
def next_candidate(self):
    if (len(self.unvisited)==0):
       candidate = None
    elif (len(self.unvisited)==1):
        candidate = self.unvisited[0]
    else:
        max_value = 0
        heuristic_values = []
        candidates = []
        candidates_available = []
        for j in self.unvisited:
            heuristic_values.append((self.si(self.start, j)**self.alpha)*(self.dsat(j)**self.beta))
            candidates.append(j)
        max_value = max(heuristic_values)
        for i in range(len(candidates)):
            if (heuristic_values[i] >= max_value):
              candidates_available.append(candidates[i])
        candidate = random.choice(candidates_available)
    self.start = candidate
    return candidate

# return your own pheromone trail
def pheromone_trail(self):
    phero_trail = np.zeros((number_nodes, number_nodes), float)
    for i in g_nodes_int:
        for j in g_nodes_int:
            if (self.colors_assigned[i]==self.colors_assigned[j]):
                phero_trail[i,j] = 1
    return phero_trail

# consistency check --> should always return 0
def colisions(self):
    colisions = 0
    for key in self.colors_assigned:
```

```
            node = key
            col = self.colors_assigned[key]
            # check colors of neighbours
            for j in range(number_nodes):
                if (adj_matrix[node, j]==1 and self.colors_assigned[j]==col):
                    colisions = colisions+1
        return colisions

# take input from the txt.file and create an undirected graph
def create_graph(path):
    global number_nodes
    g = nx.Graph()
    f = open(path)
    n = int(f.readline())
    for i in range(n):
        graph_edge_list = f.readline().split()
        # convert to int
        graph_edge_list[0] = int(graph_edge_list[0])
        graph_edge_list[1] = int(graph_edge_list[1])
        # build graph
        g.add_edge(graph_edge_list[0], graph_edge_list[1])
    return g


#draw the graph and display the weights on the edges
def draw_graph(g, col_val):
            pos = nx.spring_layout(g)
            values = [col_val.get(node, 'blue') for node in g.nodes()]
            nx.draw(g, pos, with_labels = True, node_color = values, edge_color = 'black' ,width = 1, alpha = 0.7)
#with_labels=true is to show the node number in the output graph

# initiate a selection of colors for the coloring and compute the min. number of colors needed for a proper coloring
def init_colors(g):
    # grundy (max degree+1)
    colors = []
    grundy = len(nx.degree_histogram(g))
    for c in range(grundy):
      colors.append(c)
    return colors

# create a pheromone matrix with init pheromone values: 1 if nodes not adjacent, 0 if adjacent
def init_pheromones(g):
    phero_matrix = np.ones((number_nodes, number_nodes), float)
```

```
    for node in g:
        for adj_node in g.neighbors(node):
            phero_matrix[node, adj_node] = 0
    return phero_matrix

# calculate the adjacency matrix of the graph
def adjacency_matrix(g):
    adj_matrix = np.zeros((number_nodes, number_nodes), int)
    for node in g_nodes_int:
        for adj_node in g.neighbors(node):
            adj_matrix[node, adj_node] = 1
    return adj_matrix

# create new colony
def create_colony():
    ants = []
    ants.extend([Ant().initialize(g, colors) for i in range(number_ants)])
    return ants

# apply decay rate to the phero_matrix
def apply_decay():
    for node in g_nodes_int:
        for adj_node in g_nodes_int:
            phero_matrix[node, adj_node] = phero_matrix[node, adj_node]*(1-phero_decay)



# select colony's best solution
# update pheromone_matrix according to the elite solution
# return elite solution (coloring) with its distance (number of used colors)
def update_elite():
    global phero_matrix
    # select elite
    best_dist = 0
    elite_ant = None
    ants_num = 0
    allants_dist = []

    for ant in ants:
        if (best_dist==0):
            best_dist = ant.distance
            elite_ant = ant
        elif (ant.distance < best_dist):
            best_dist = ant.distance
```

```
            elite_ant = ant

        ants_num += 1
        allants_dist.append(ant.distance)  # 동일 iteration에서 모든 ant들의 거리(사용한 색깔 수) 저장
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

    # update global phero_matrix
    elite_phero_matrix = elite_ant.pheromone_trail()
    phero_matrix = phero_matrix + elite_phero_matrix
    return elite_ant.distance, elite_ant.colors_assigned, ants_num, allants_dist #
@@@@@@@@@@@@@@@@@@@@@@@@@@@
    #iter_num랑 allants_dist 추가 !!!!!!


# ------------- entry point -------------
# param input_graph - a networkx graph to be colored (node coloring)
# param num_ants - number of ants in the colony
# param iter - number of iterations to be performed
# param a - relative importance of elite pheromones
# param b - relative importance of heuristic value (DSAT)
# param decay - evaporation of pheromones after each iteration
def solve(input_graph, num_ants=10, iter=10, a=1, b=3, decay=0.8):
    global g # graph to be colored (a networkx graph)
    global number_nodes
    global g_nodes_int
    global number_ants
    global alpha
    global beta
    global phero_decay
    global adj_matrix
    global phero_matrix
    global colors
    global ants


    start=time.time()

    # params
    g = input_graph
    number_ants=num_ants
    number_iterations=iter
    alpha = a # relative importance of pheromone (si_ij)
    beta = b # relative importance of heuristic value (n_ij)
```

```
phero_decay=decay # rate of pheromone decay

# results
final_solution = {} # coloring of the graph
final_costs = 0 # number of colors in the solution

# init
number_nodes = nx.number_of_nodes(g)
g_nodes_int = []
for node in g.nodes():  #nodes_iter -> nodes
    g_nodes_int.append(node)
g_nodes_int = list(map(int, sorted(g_nodes_int)))
adj_matrix = adjacency_matrix(g)
colors = init_colors(g)
phero_matrix = init_pheromones(g)

allants_med_dist_flow = []

# ACO_GCP daemon
for i in range(number_iterations):

    # create colony
    ants = []
    ants = create_colony()

    # let colony find solutions
    for ant in ants:
        ant.colorize()
    # apply decay rate
    apply_decay()
    # select elite and update si_matrix
    elite_dist, elite_sol, elite_ants_num, allants_dist = update_elite()
    # estimate global solution so far
    if (final_costs==0):
        final_costs = elite_dist
        final_solution = elite_sol
    elif (elite_dist<final_costs):
        final_costs = elite_dist
        final_solution = elite_sol

    allants_med_dist = np.mean(allants_dist).round()  # 동일 iteration에서 개미들의 dist 중간값
    allants_med_dist_flow.append(allants_med_dist)  # 중간값 append한 리스트
```

```
#         print('niter_num: ', i)  # for steps: Antcol 과정 알려면 이 부분 주석풀기
#         print('allants_dist')
        print(allants_dist)
#         print('allants_med_dist')
#         print(allants_med_dist_flow)
        print('iteration ', i, 'done')


    iter_time = time.time()-start


    # Print the results
    print("Number of Colors used: ", max(final_solution.values())+1)
    print("Number of Iterations: ", number_iterations)
    print("Computation time: ", f"{iter_time:.4f} sec")
#     print("Colors_Flow: ", elite_dist_flow)

    return_dict ={'theSolution': final_solution, 'final_costs': final_costs, 'final_color_num': max(final_solution.values())+1,
                  'color_num_flow': allants_med_dist_flow, 'iter_num': number_iterations, 'iter_time': iter_time,
                 'ants_num': elite_ants_num}
    return return_dict

# global vars
g = None # graph to be colored
number_nodes = 0
g_nodes_int = []
number_ants = 0
alpha = 0
beta = 0
phero_decay = 0
adj_matrix = np.zeros((number_nodes, number_nodes), int)
phero_matrix = np.ones((number_nodes, number_nodes), float)
colors = []
ants = []

### 3. Define Backtracking Algorithm

# A utility function to check
# if the current color assignment
# is safe for vertex v
def isSafe(graph, v, colour, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and colour[i] == c:
```

```
            return False
    return True

# A recursive utility function to solve m
# coloring  problem
def graphColourUtil(graph, m, colour, v):
    if v == len(graph):
        return True

    for c in range(1, m + 1):
        if isSafe(graph, v, colour, c) == True:
            colour[v] = c
            if graphColourUtil(graph, m, colour, v + 1) == True:
                return True
            colour[v] = 0


def Backtracking(graph, m):

    start=time.time()

    colour = [0] * len(graph)
    if graphColourUtil(graph, m, colour, 0) == None:
        return False

    # Print the solution
    print("Solution exist and Following are the assigned colours:")
    iter_num=0
    for c in colour:
        print(c, end=' ')

    print("Number of Colors used : \n", len(set(colour)))
    print("Number of Iterations : \n", iter_num)
    print("Computation time : \n", f"{time.time()-start:.4f} sec")
    #return True, len(set(colour))

## Simulation 1. 간단한 예시

### [Case 1] n=4 (simple graph)

import random
random.seed(1886)
start=time.time()
```

```
# (n = 100, a = 1240)
n=4; G=np.zeros((n,n))

for i in range(0,n):
    for j in range(i+1,n):
        G[i,j]=random.choices(range(0,2), weights=[1,1])[0]



G = np.triu(G)
G += G.T - np.diag(G.diagonal())
print(G.sum()/2)
G
# network plot: 실제 그래프 시각화
G = nx.Graph()
G.add_edges_from([('0','1'),('0','2'),('0','3'),('1','3'),('2','3')])
pos = {'0':[1,1],  '1':[1,5],  '2':[5,1],  '3':[5,5]}
nx.draw_networkx(G,pos,node_size = 1000, node_color = 'pink')
plt.show();

# 0 : color2 / 1,2 : color0 / 3 : color1 /
color_map = ['red','blue','blue','green']
nx.draw_networkx(G,pos,node_size = 1000, node_color = color_map)
plt.show();
```

래프(아래)를 살펴보면, 인접 node는 색이 겹치지 않도록 색칠되었다는 것을 알 수 있다.

1) Greedy Algorithm

```
# Adjacent Matrix 정의
G = [[ 0, 1, 1, 1],
    [ 1, 0, 0, 1],
    [ 1, 0, 0, 1],
    [ 1, 1, 1, 0]]

# greedy 알고리즘 적용
solutions = greedy(G, n=len(G))



# 시각화

# 1) greedy 알고리즘으로 구한 Solution으로 그린 그래프

## 색 id을 색상으로 변환 (ex. 1 -> red)
solution_gd = solutions['theSolution']
sorted_solution_gd = dict(sorted(solution_gd.items()))
```

```
color_values = list(sorted_solution_gd.values())
colors_greedy = [color_palette[color_id] for color_id in color_values]
colors_greedy

# 그리기
plt.figure(figsize=(6,4))
ax = plt.gca()

G = nx.Graph()
G.add_edges_from([('0','1'),('0','2'),('0','3'),('1','3'),('2','3')])  #### 시간허락되면 이 부분 자동화하기
nx.draw_networkx(G,pos,node_size = 1000, node_color = colors_greedy, ax=ax)
ax.set_title('Graph Coloring using Greedy | n=4')
_ = ax.axis('off')


# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num']),
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | n=4',
    xticks=color_num_flow_df['niter'])
fig.tight_layout();
plt.show();
Antcol Algorithm
# Adjacent Matrix 정의
G = nx.Graph()
G.add_edges_from([(0,1),(0,2),(0,3),(1,3),(2,3)])
pos = {0:[1,1],  1:[1,5],  2:[5,1],  3:[5,5]}

# Antcol 적용
solutions = solve(G)

# 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num'])+1,
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='colors_num', title='Antcol Algorithm | n=4',
```

```
    xticks=color_num_flow_df['niter'], yticks=[3,4])
fig.tight_layout();
plt.show();

#### 3) Backtracking

start = time.time()
Backtracking(G,100)
print(f"{time.time()-start:.4f} sec")

ase 2] n=9 (medium graph)
# network plot: 실제 그래프 시각화
G = nx.Graph()
G.add_edges_from([('0','1'),('0','3'),('0','4'),('1','2'),('1','4'),('1','5'),('2','5'),
                  ('3','4'),('3','6'),('3','7'),('4','5'),('4','7'),('4','8'),('5','8'),('6','7'),('7','8')])
pos = {'0':[1,1],  '1':[3,1],  '2':[5,1],
       '3':[1,3],  '4':[3,3],  '5':[5,3],
       '6':[1,5],  '7':[3,5], '8':[5,5]}
nx.draw_networkx(G,pos,node_size = 1000, node_color = 'pink')
plt.show();
# 모범답안

if __name__ == '__main__':

    g1 = [[] for i in range(9)]
    g1 = addEdge(g1, 0, 1)
    g1 = addEdge(g1, 0, 3)
    g1 = addEdge(g1, 0, 4)
    g1 = addEdge(g1, 1, 2)
    g1 = addEdge(g1, 1, 4)
    g1 = addEdge(g1, 1, 5)
    g1 = addEdge(g1, 2, 5)
    g1 = addEdge(g1, 3, 4)
    g1 = addEdge(g1, 3, 6)
    g1 = addEdge(g1, 3, 7)
    g1 = addEdge(g1, 4, 5)
    g1 = addEdge(g1, 4, 7)
    g1 = addEdge(g1, 4, 8)
    g1 = addEdge(g1, 5, 8)
    g1 = addEdge(g1, 6, 7)
    g1 = addEdge(g1, 7, 8)

    print("Coloring of graph 2 ")
```

```
    greedyColoring(g1, 9)

# 색칠한 network plot
# 0,2,6,8 : color1(red) / 1,3 : color2(blue) / 4 : color3(green_) /5,7 : color4(yellow)
G = nx.Graph()
G.add_edges_from([('0','1'),('0','3'),('0','4'),('1','4'),('1','2'),('1','5'),('4','5'),('2','5'),
                  ('3','4'),('3','6'),('3','7'),('6','7'),('4','7'),('4','8'),('7','8'),('5','8')])
pos = {'0':[1,1],  '1':[3,1],  '2':[5,1],
       '3':[1,3],  '4':[3,3],  '5':[5,3],
       '6':[1,5],  '7':[3,5], '8':[5,5]}
color_map = ['red','blue','blue','green','red','yellow','red','yellow','red']
nx.draw_networkx(G,pos,node_size = 1000, node_color = color_map)
plt.show();
1) Greedy Algorithm
# Adjacent Matrix 정의
G = [[ 0, 1, 0, 1, 1, 0, 0, 0, 0],
     [ 1, 0, 1, 0, 1, 1, 0, 0, 0],
     [ 0, 1, 0, 0, 0, 1, 0, 0, 0],
     [ 1, 0, 0, 0, 1, 0, 1, 1, 0],
     [ 1, 1, 0, 1, 0, 1, 0, 1, 1],
     [ 0, 1, 1, 0, 1, 0, 0, 0, 1],
     [ 0, 0, 0, 1, 0, 0, 0, 1, 0],
     [ 0, 0, 0, 1, 1, 0, 1, 0, 1],
     [ 0, 0, 0, 0, 0, 1, 0, 1, 0]]


# greedy 알고리즘 적용
solutions = greedy(G, n=len(G))


# 시각화

# 1) greedy 알고리즘으로 구한 Solution으로 그린 그래프

## 색 id을 색상으로 변환 (ex. 1 -> red)
solution_gd = solutions['theSolution']
sorted_solution_gd = dict(sorted(solution_gd.items()))
color_values = list(sorted_solution_gd.values())
colors_greedy = [color_palette[color_id] for color_id in color_values]
colors_greedy

# 그리기
plt.figure(figsize=(6,4))
```

```
ax = plt.gca()

G = nx.Graph()
G.add_edges_from([('0','1'),('0','3'),('0','4'),('1','4'),('1','2'),('1','5'),('4','5'),('2','5'),
                  ('3','4'),('3','6'),('3','7'),('6','7'),('4','7'),('4','8'),('7','8'),('5','8')])  #### 시간허락되면 이 부분 자동화하기
nx.draw_networkx(G,pos,node_size = 1000, node_color = colors_greedy, ax=ax)
ax.set_title('Graph Coloring using Greedy | n=9')
_ = ax.axis('off')



# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num']),
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | n=9',
    xticks=color_num_flow_df['niter'])
fig.tight_layout();

2) Antcol Algorithm
# Adjacent Matrix 정의
G = nx.Graph()
G.add_edges_from([(0,1),(0,3),(0,4),(1,2),(1,4),(1,5),(2,5),
                  (3,4),(3,6),(3,7),(4,5),(4,7),(4,8),(5,8),(6,7),(7,8)])

pos = {0:[1,1],  1:[3,1],  2:[5,1],
       3:[1,3],  4:[3,3],  5:[5,3],
       6:[1,5],  7:[3,5],  8:[5,5]}

# Antcol 적용
solutions = solve(G)

# 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num'])+1,
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='colors_num', title='Antcol Algorithm | n=9',
    xticks=color_num_flow_df['niter'])
```

```
ax.yaxis.set_major_locator(MaxNLocator(integer=True))  # y축 숫자 int만 출력
fig.tight_layout();
plt.show();
```

Backtracking Algorithm

```
start=time.time()
Backtracking(G,3)
print(f"{time.time()-start:.4f} sec")
```

이렇게 생성된 graph들에 대해 각각의 알고리즘은 어떠한 강점 혹은 약점이 있는지 알아보자.

[Case 1] n=100, a=1638

```
# Adjacent Matrix 정의
import random
random.seed(1886)

n=100
G=np.zeros((n,n))

for i in range(0,n):  # 무작위로 연결선(a) 생성
    for j in range(i+1,n):
        G[i,j]=random.choices(range(0,2), weights=[2,1])[0]

G = np.triu(G)
G += G.T - np.diag(G.diagonal())
print('a = ', G.sum()/2)
```

1) Greedy Algorithm

```
# 1) greedy 알고리즘 적용
solutions = greedy(G, n=len(G))


# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num']),
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | a=1638',
      )
fig.tight_layout();
```

2) Antcol Algorithm

```
# Antcol 적용
G = nx.Graph(G)
solutions1 = solve(G, num_ants=8, iter=15)
```

```
# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions1['iter_num']),
                                  'colors_used': solutions1['color_num_flow']})
sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Antcol Algorithm | a=1638',
    )
fig.tight_layout();
```

2) Backtracking Algorithm

```
start=time.time()
Backtracking(G,100)
print(f"{time.time()-start:.4f} sec")

Case 2] n=100, a=2467
# Adjacent Matrix 정의
import random
random.seed(1886)

n=100
G=np.zeros((n,n))

for i in range(0,n):  # 무작위로 연결선(a) 생성
   for j in range(i+1,n):
      G[i,j]=random.choices(range(0,2), weights=[1,1])[0]

G = np.triu(G)
G += G.T - np.diag(G.diagonal())
print('a = ', G.sum()/2)
1) Greedy Algorithm
# 1) greedy 알고리즘 적용
solutions = greedy(G, n=len(G))


# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num']),
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
```

```
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | a=2467',
    )
fig.tight_layout();
```

3) Antcol Algorithm

```
# Antcol 적용
G = nx.Graph(G)
solutions2 = solve(G, num_ants=8, iter=15)

# 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions2['iter_num']),
                                  'colors_used': solutions2['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Antcol Algorithm | a=2467',
    )
fig.tight_layout();
) Backtracking Algorithm
star
start=time.time()
Backtracking(G,100)
print(f"{time.time()-start:.4f} sec")
[Case 3] n=100, a=3751
) Backtracking Algorithm
start=time.time()
Backtracking(G,100)
print(f"{time.time()-start:.4f} sec")
### [Case 4] n=100, a=4179

# Adjacent Matrix 정의
import random
random.seed(1886)

n=100
G=np.zeros((n,n))

for i in range(0,n):  # 무작위로 연결선(a) 생성
   for j in range(i+1,n):
      G[i,j]=random.choices(range(0,2), weights=[1,5])[0]

G = np.triu(G)
```

```
G += G.T - np.diag(G.diagonal())
print('a = ', G.sum()/2)

#### 1) Greedy Algorithm

# 1) greedy 알고리즘 적용
solutions4 = greedy(G, n=len(G))



# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions4['iter_num']),
                                  'colors_used': solutions4['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | a=4179',
    )
fig.tight_layout();

#### 2) Antcol Algorithm

# Antcol 적용
G = nx.Graph(G)
solutions4 = solve(G, num_ants=8, iter=15)

# 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions4['iter_num']),
                                  'colors_used': solutions4['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Antcol Algorithm | a=4179',
    )
fig.tight_layout();

#### 3) Backtracking Algorithm

start=time.time()
Backtracking(G,100)
print(f"{time.time()-start:.4f} sec")
## Simulation 3. n과 a가 점차적으로 증가하는 그래프
```

- n: node 개수, a: edge 개수 이다.
- seed를 고정하여 무작위로 n과 a 값을 뽑아 graph를 만든다.
- simulation 파트에서는 노드 개수가 최소 100개, 최대 1000개이다. 따라서 그래프 시각화는 컴퓨팅 파워 부족으로 어렵기 때문에 생략하였다.

- 이렇게 생성된 graph들에 대해 각각의 알고리즘은 어떠한 강점 혹은 약점이 있는지 알아보자.

[Case 1] n=100, a = 1638

```
# Adjacent Matrix 정의
import random
random.seed(1886)

n=100
G=np.zeros((n,n))

for i in range(0,n):  # 무작위로 연결선(a) 생성
    for j in range(i+1,n):
        G[i,j]=random.choices(range(0,2), weights=[2,1])[0]

G = np.triu(G)
G += G.T - np.diag(G.diagonal())
print('a = ', G.sum()/2)
```

1) Greedy Algorithm

```
# 1) greedy 알고리즘 적용
solutions = greedy(G, n=len(G))


# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num']),
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | n=100',
      )
fig.tight_layout();
```

2) Antcol Algorithm

```
# Antcol 적용
G = nx.Graph(G)
solutions1 = solve(G, num_ants=8, iter=15)

color_num_flow2 = [
    [14, 15, 14, 14, 14, 14, 15, 14],
    [14, 14, 14, 14, 15, 14, 14, 13],
    [13, 15, 13, 14, 14, 15, 14, 13],
    [13, 13, 13, 14, 15, 13, 13, 13],
    [13, 13, 13, 13, 13, 13, 13, 13],
    [15, 13, 14, 14, 13, 13, 14, 13],
    [14, 14, 13, 13, 14, 13, 13, 13],
    [13, 13, 13, 14, 13, 13, 13, 13],
    [13, 13, 14, 13, 13, 13, 13, 13],
    [15, 13, 13, 14, 15, 14, 13, 14],
    [13, 13, 13, 13, 13, 13, 13, 13],
    [13, 13, 13, 13, 13, 13, 13, 13],
    [13, 15, 13, 14, 13, 13, 14, 14],
    [13, 13, 13, 13, 13, 13, 13, 13],
    [13, 14, 14, 13, 13, 13, 14, 14]
]

# 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(15)+1,
                                  'colors_used': np.min(color_num_flow2, axis=1)+1})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='colors_num', title='Antcol Algorithm | n=100',
    )
ax.yaxis.set_major_locator(MaxNLocator(integer=True))  # y축 숫자 int만 출력
fig.tight_layout();
plt.show();

#### 3) Backtracking Algorithm

Backtracking(G,100)

### [Case 2] n=500, a = 41959

# Adjacent Matrix 정의
```

```
import random
random.seed(1886)

n=500
G=np.zeros((n,n))

for i in range(0,n):  # 무작위로 연결선(a) 생성
    for j in range(i+1,n):
        G[i,j]=random.choices(range(0,2), weights=[2,1])[0]

G = np.triu(G)
G += G.T - np.diag(G.diagonal())
print('a = ', G.sum()/2)

#### 1) Greedy Algorithm

# 1) greedy 알고리즘 적용
solutions = greedy(G, n=len(G))



# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num']),
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | n=500',
      )
fig.tight_layout();

#### 2) Antcol Algorithm

# Antcol 적용
G = nx.Graph(G)
solutions = solve(G, num_ants=5, iter=15)

color_num_flow2 = [
    [47, 47, 46, 47, 46],
    [46, 46, 47, 46, 46],
    [46, 47, 48, 46, 47],
    [47, 46, 46, 46, 46],
    [49, 46, 47, 46, 48],
```

```
    [49, 48, 46, 45, 47],
    [45, 45, 46, 47, 45],
    [47, 46, 47, 47, 46],
    [46, 46, 46, 47, 46],
    [46, 49, 49, 47, 46],
    [48, 46, 48, 47, 46],
    [47, 46, 46, 48, 48],
    [46, 46, 45, 46, 46],
    [45, 46, 46, 45, 46],
    [44, 44, 44, 43, 44]]

# 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(15)+1,
                                  'colors_used': np.min(color_num_flow2, axis=1)})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='colors_num', title='Antcol Algorithm | n=500',
      )
ax.yaxis.set_major_locator(MaxNLocator(integer=True))  # y축 숫자 int만 출력
fig.tight_layout();
plt.show();

#### 3) Backtracking Algorithm

Backtracking(G,1000) ### [Case 3] n=800, a = 107163

# Adjacent Matrix 정의
import random
random.seed(1886)

n=800
G=np.zeros((n,n))

for i in range(0,n):  # 무작위로 연결선(a) 생성
    for j in range(i+1,n):
        G[i,j]=random.choices(range(0,2), weights=[2,1])[0]

G = np.triu(G)
G += G.T - np.diag(G.diagonal())
print('a = ', G.sum()/2)
```

1) Greedy Algorithm

```
# 1) greedy 알고리즘 적용
solutions = greedy(G, n=len(G))



# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num']),
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | n=800',
      )
fig.tight_layout();
```

2) Antcol Algorithm

```
# Antcol 적용
G = nx.Graph(G)
solutions = solve(G, num_ants=5, iter=16)

color_num_flow2 = [
    [66, 66, 65, 66, 66],
    [65, 65, 65, 67, 65],
    [65, 66, 65, 65, 65],
    [68, 66, 65, 67, 66],
    [65, 65, 66, 65, 65],
    [66, 65, 65, 65, 65],
    [68, 70, 65, 65, 67],
    [66, 65, 65, 65, 66],
    [65, 65, 65, 66, 65],
    [65, 64, 66, 65, 65],
    [64, 66, 65, 65, 64],
    [64, 65, 64, 65, 69],
    [64, 66, 64, 64, 64],
    [64, 66, 64, 64, 64],
    [63, 63, 63, 63, 63]]

# 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))
```

```
color_num_flow_df = pd.DataFrame({'niter': np.arange(15)+1,
                                  'colors_used': np.min(color_num_flow2, axis=1)})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='colors_num', title='Antcol Algorithm | n=800',
    )
ax.yaxis.set_major_locator(MaxNLocator(integer=True))  # y축 숫자 int만 출력
fig.tight_layout();
plt.show();

#### 3) Backtracking Algorithm

Backtracking(G,1000)
### [Case 4] n=1000, a = 167232

# Adjacent Matrix 정의
import random
random.seed(1886)

n=1000
G=np.zeros((n,n))

for i in range(0,n):  # 무작위로 연결선(a) 생성
   for j in range(i+1,n):
      G[i,j]=random.choices(range(0,2), weights=[2,1])[0]

G = np.triu(G)
G += G.T - np.diag(G.diagonal())
print('a = ', G.sum()/2)

#### 1) Greedy Algorithm

# 1) greedy 알고리즘 적용
solutions = greedy(G, n=len(G))



# 2) 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': np.arange(solutions['iter_num']),
                                  'colors_used': solutions['color_num_flow']})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
```

```
ax.set(ylabel='candidate_colors_num', title='Greedy Algorithm | n=1000',
    )
fig.tight_layout();

#### 2) Antcol Algorithm

# Antcol 적용
G = nx.Graph(G)
solutions4 = solve(G, num_ants=8, iter=10)

color_num_flow2 = [
 [76, 77, 77, 76, 77, 77, 77, 76],
 [76, 77, 80, 76, 77, 76, 76, 76],
 [80, 76, 77, 77, 76, 76, 77, 76],
 [77, 79, 77, 77, 76, 76, 76, 76],
 [77, 79, 76, 76, 76, 77, 76, 78],
 [79, 78, 77, 76, 76, 76, 79, 77],
 [77, 76, 76, 76, 76, 76, 77, 76],
 [76, 76, 79, 76, 77, 76, 77, 77],
 [77, 76, 76, 76, 76, 76, 76, 77],
 [76, 74, 77, 76, 76, 76, 76, 77]
 ]

# 수렴 그래프
fig, ax = plt.subplots(figsize=(6,8))

color_num_flow_df = pd.DataFrame({'niter': (np.arange(10)+1).round(0),
                          'colors_used': np.min(color_num_flow2, axis=1)})

sns.lineplot(x='niter', y='colors_used', data=color_num_flow_df, ax=ax)
ax.set(ylabel='colors_num', title='Antcol Algorithm | n=1000',
    )
ax.xaxis.set_major_locator(MaxNLocator(integer=True))  # x축 숫자 int만 출력
ax.yaxis.set_major_locator(MaxNLocator(integer=True))  # y축 숫자 int만 출력
fig.tight_layout();
plt.show();

#### 3) Backtracking Algorithm

Backtracking(G,1000)
```

시뮬레이션 시각화

```
library(tidyverse)

numcol=data.frame(number_of_color=c(15,19,33,38,13,18,29,34,16,21,34,42),
                  compute_time=c(0.0118,0.0128,0.0146,0.0158,
                                 12.4948,13.3168,14.7023,15.0701,
                                 0.0229,0.0140,0.0275,0.0402),
                  modeling=c("greedy","greedy","greedy","greedy",
                             "Antcol","Antcol","Antcol","Antcol",
                             "backtracking","backtracking","backtracking","backtracking"),
                  a=c(1638,2467,3747,4124,
                      1638,2467,3747,4124,
                      1638,2467,3747,4124))

ggplot(numcol)+geom_line(aes(x=a, y=number_of_color, group=modeling, colour=modeling))+
  ggtitle("number of colors by modeling")


ggplot(numcol)+geom_line(aes(x=a, y=compute_time, group=modeling, colour=modeling))+
  ggtitle("computation time by modeling")

numcol%>% filter(modeling != "Antcol") %>%ggplot()+geom_line(aes(x=a, y=compute_time, group=modeling,
colour=modeling))+
  ggtitle("computation time by modeling")




numcol=data.frame(number_of_color=c(15,48,68,83,
                                    13,43,63,74,
                                    16,49,69,83),
                  compute_time=c(0.0141,0.8166,3.1208,6.2304,
                                 12.4630,1456.0349,6214.8627,12200.6978,
                                 0.0162,0.4818,1.5928,3.0254),
                  modeling=c("greedy","greedy","greedy","greedy",
                             "Antcol","Antcol","Antcol","Antcol",
                             "backtracking","backtracking","backtracking","backtracking"),
                  n=c(100,500,800,1000,
                      100,500,800,1000,
                      100,500,800,1000)
)
```

```
ggplot(numcol)+geom_line(aes(x=n, y=number_of_color, group=modeling, colour=modeling))+
  ggtitle("number of colors by modeling")


ggplot(numcol)+geom_line(aes(x=n, y=compute_time, group=modeling, colour=modeling))+
  ggtitle("computation time by modeling")

numcol%>% filter(modeling != "Antcol") %>%ggplot()+geom_line(aes(x=n, y=compute_time, group=modeling,
colour=modeling))+
  ggtitle("computation time by modeling")
```

시간표 적용

```
  import numpy as np
  import pandas as pd
  import seaborn as sns
  import matplotlib.pyplot as plt
  import networkx as nx
  import random

  import time # time 라이브러리 import

  ## * 이 셀이 제대로 작동하지 않고 길게 warning 메시지가 나올 경우, 상단의 메뉴에서 <런타임>-<런타임 다시 시작>을 클릭한
뒤 처음부터 코드를 실행하세요
  # 그래프 한글 폰트 설정

  import matplotlib as mpl
  import matplotlib.pyplot as plt
  import matplotlib.font_manager as fm

  %config InlineBackend.figure_format = 'retina'

  #폰트 설치
  !apt -qq -y install fonts-nanum

  #기본 폰트로 지정
  fontpath = '/usr/share/fonts/truetype/nanum/NanumBarunGothic.ttf'
  font = fm.FontProperties(fname=fontpath, size=9)
  plt.rc('font', family='NanumBarunGothic')
  mpl.font_manager._rebuild()

  from IPython.display import set_matplotlib_formats
```

```
#폰트 선명하게 보이게 하기 위해
set_matplotlib_formats('retina')

## Test
plt.figure(figsize=(2,2))
plt.text(0.3, 0.3, '한글 폰트 테스트', size=15)
plt.show()

# 시간표 적용

# Adjacent Matrix: 노드랑 선이 연결되면 1 아니면 0
G = [
    [0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #통계학_문1 a
    [1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #통계학_문2 b
    [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #통계학_문3 c
    [0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #확통_오 d
    [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #확통_송1 e
    [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #확통_송2 f
    [0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #베이지안_오 g
    [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #표조실_차 h
    [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #손보_차(9) i
    [0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #경경분_신(10) j
    [0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #시계열_신(11) k
    [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #자분특_주(12) l
    [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],        #고자분_주(13)m
    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],        #GLM_이(14) n
    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],        #범주형_이(15) o
    [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0],        #고통_김(16) p
    [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,1,1,0,1,0],              #기확_김(17) q
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,1,0],              #기확_안(18) r
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,1,0,1,0],          #회귀_유1(19) s
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,1,0,1,0],          #회귀_유2(20) t
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,1,0,1,0],          #통프_안(21) u
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,1,1,0],          #데마_송(22) v
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0],          #계특_송(23) w
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,1],          #수통_소(24) x
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],          #이통_소(25) y

]

G = nx.Graph()
G.add_nodes_from(['통_문1','통_문2','통_문3','확통_오','확통_송1','확통_송2','베이지안_오','표조실_차','손보_차','경경분_신','시계열_신','자분특_주',
```

```
'고자분_주','GLM_이','범주형_이','고통_김','기확_김','기확_안','회귀_유1','회귀_유2','통프_안','데마_송','계특_송','수통_소','이통_
소'])


  G.add_edges_from([('통_문1','통_문2'), ('통_문1','통_문3'), ('통_문2','통_문3'),
                    ('확통_오','베이지안_오'),('확통_송1','확통_송2'),
                    ('표조실_차','손보_차'),('경경분_신','시계열_신'),
                    ('자분특_주','고자분_주'),
                    ('GLM_이','범주형_이'),
                    ('고통_김','기확_김'),
                    ('기확_김','회귀_유1'),('기확_김','회귀_유2'),('기확_김','수통_소'),('기확_김','데마_송'),('기확_김','통프_안'),
                    ('기확_안','회귀_유1'),('기확_안','회귀_유2'),('기확_안','수통_소'),('기확_안','데마_송'),('기확_안','통프_안'),

('통프_안','기확_안'),('통프_안','기확_김'),('통프_안','회귀_유1'),('통프_안','회귀_유2'),('통프_안','수통_소'),('통프_안','데마_송'),

('데마_송','기확_김'),('데마_송','회귀_유1'),('데마_송','회귀_유2'),('데마_송','수통_소'),('데마_송','통프_안'),('데마_송','기확_안'),('
데마_송','계특_송'),

('수통_소','기확_김'),('수통_소','기확_안'),('수통_소','통프_안'),('수통_소','데마_송'),('수통_소','회귀_유1'),('수통_소','회귀_유2'),('
수통_소','이통_소'),

('회귀_유1','기확_김'),('회귀_유1','기확_안'),('회귀_유1','통프_안'),('회귀_유1','데마_송'),('회귀_유1','수통_소'),('회귀_유1','회귀_
유2'),
                    ('회귀_유2','기확_김'),('회귀_유2','기확_안'),('회귀_유2','통프_안'),('회귀_유2','데마_송'),('회귀_유2','수통_소')])

  pos1 = nx.kamada_kawai_layout(G)
  plt.figure(figsize=(10,10))
  nx.draw(G, pos1, with_labels=True,font_family='NanumBarunGothic', node_color = 'gray',node_size = 1500)
  plt.show()

  len(G.edges())

  ### Greedy(6개 색)

  start = time.time() # 시작

  # initiate the name of node.
  node = "abcdefghijklmnopqrstuvwxy"
  t_={}
  for i in range(len(G)):
    t_[node[i]] = i
```

```
# count degree of all node. 연결된 엣지 갯수
degree =[]
for i in range(len(G)):
  degree.append(sum(G[i]))

# inisiate the posible color
colorDict = {}
for i in range(len(G)):
  colorDict[node[i]]=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]



# sort the node depends on the degree. 내림차순 정렬
sortedNode=[]
indeks = []

# use selection sort
for i in range(len(degree)):
  _max = 0
  j = 0
  for j in range(len(degree)):
    if j not in indeks:
      if degree[j] > _max:
        _max = degree[j]
        idx = j
  indeks.append(idx)
  sortedNode.append(node[idx])

# The main process
theSolution={}
for n in sortedNode:
  setTheColor = colorDict[n]
  theSolution[n] = setTheColor[0]
  adjacentNode = G[t_[n]]
  for j in range(len(adjacentNode)):
    if adjacentNode[j]==1 and (setTheColor[0] in colorDict[node[j]]):
      colorDict[node[j]].remove(setTheColor[0])

# Print the solution
for t,w in sorted(theSolution.items()):
  print("Node",t," = ",w)

print(f"{time.time()-start:.4f} sec")
```

```
#색칠하기

G = nx.Graph()
G.add_nodes_from(['통_문1','통_문2','통_문3','확통_오','확통_송1','확통_송2','베이지안_오','표조실_차','손보_차','경경분_신','시계열_신','자분특_주',

'고자분_주','GLM_이','범주형_이','고통_김','기확_김','기확_안','회귀_유1','회귀_유2','통프_안','데마_송','계특_송','수통_소','이통_소'])


G.add_edges_from([('통_문1','통_문2'), ('통_문1','통_문3'), ('통_문2','통_문3'),
                  ('확통_오','베이지안_오'),('확통_송1','확통_송2'),
                  ('표조실_차','손보_차'),('경경분_신','시계열_신'),
                  ('자분특_주','고자분_주'),
                  ('GLM_이','범주형_이'),
                  ('고통_김','기확_김'),
                  ('기확_김','회귀_유1'),('기확_김','회귀_유2'),('기확_김','수통_소'),('기확_김','데마_송'),('기확_김','통프_안'),
                  ('기확_안','회귀_유1'),('기확_안','회귀_유2'),('기확_안','수통_소'),('기확_안','데마_송'),('기확_안','통프_안'),

('통프_안','기확_안'),('통프_안','기확_김'),('통프_안','회귀_유1'),('통프_안','회귀_유2'),('통프_안','수통_소'),('통프_안','데마_송'),

('데마_송','기확_김'),('데마_송','회귀_유1'),('데마_송','회귀_유2'),('데마_송','수통_소'),('데마_송','통프_안'),('데마_송','기확_안'),('데마_송','계특_송'),

('수통_소','기확_김'),('수통_소','기확_안'),('수통_소','통프_안'),('수통_소','데마_송'),('수통_소','회귀_유1'),('수통_소','회귀_유2'),('수통_소','이통_소'),

('회귀_유1','기확_김'),('회귀_유1','기확_안'),('회귀_유1','통프_안'),('회귀_유1','데마_송'),('회귀_유1','수통_소'),('회귀_유1','회귀_유2'),
                  ('회귀_유2','기확_김'),('회귀_유2','기확_안'),('회귀_유2','통프_안'),('회귀_유2','데마_송'),('회귀_유2','수통_소')])

pos1 = nx.kamada_kawai_layout(G)
plt.figure(figsize=(10,10))

color_map = ['pink','orange','yellow','pink','pink',
             'orange','orange','pink','orange','pink',
             'orange','pink','orange','pink','orange',
             'pink','yellow','yellow','green','blue',
             'purple','pink','orange','orange','pink']
nx.draw(G, pos1, with_labels=True,font_family='NanumBarunGothic', node_color = color_map ,node_size = 1500)
plt.show()
```

```
# pink 색0: [a,d,e,h,j,l,n,p,v,y]
# orange 색1: [b,f,g,i,k,m,o,w,x]
# yellow 색2: [c,q,r]
# green 색3: [s]
# blue 색4:[t]
# purple 색5:[u]


### Antcol

class Ant:
    # create new ant
    # alpha: the relative importance of pheromone (si_ij)
    # beta: the relative importance of heuristic value (n_ij)
    def __init__(self, alpha=1, beta=3):
        self.graph = None
        self.colors = {}
        self.start = None
        self.visited = []
        self.unvisited = []
        self.alpha = alpha
        self.beta = beta
        self.distance = 0 # number of used colors on a valid solution
        self.number_colisions = 0 # only for consistency check, should be always 0
        self.colors_available = []
        self.colors_assigned = {}

    # reset everything for a new solution
    # start: starting node in g (random by default)
    # return: Ant
    def initialize(self, g, colors, start=None):
        self.colors_available = sorted(colors.copy())

        # init assigned colors with None
        keys = [n for n in g_nodes_int]
        self.colors_assigned = {key: None for key in keys}

        # start node
        if start is None:
            self.start = random.choice(g_nodes_int)
        else:
            self.start = start
```

```
    self.visited = []
    self.unvisited = g_nodes_int.copy()

    # assign min. color number to the start node
    if (len(self.visited)==0):
        self.assign_color(self.start, self.colors_available[0])
    return self

# assign color to node and update the node lists
def assign_color(self, node, color):
    self.colors_assigned[node] = color
    self.visited.append(node)
    self.unvisited.remove(node)

# assign a color to each node in the graph
def colorize(self):
    len_unvisited = len(self.unvisited)
    tabu_colors = []
    # assign color to each unvisited node
    for i in range(len_unvisited):
        next = self.next_candidate()
        tabu_colors = []
        # add colors of neighbours to tabu list
        for j in range(number_nodes):
            if (adj_matrix[next,j]==1):
                tabu_colors.append(self.colors_assigned[j])
        # assign color with the smallest number that is not tabu
        for k in self.colors_available:
            if (k not in tabu_colors):
                self.assign_color(next,k)
                break
    # save distance of the current solution
    self.distance = len(set(self.colors_assigned.values()))
    # consitency check
    ##self.number_colisions = self.colisions()
    ##print('colisions: ' + str(self.number_colisions))

# return the number of different colors among the neighbours of node
def dsat(self, node=None):
    if node is None:
        node = self.start
    col_neighbors = []
    for j in range(number_nodes):
```

```
        if (adj_matrix[node, j]==1):
            col_neighbors.append(self.colors_assigned[j])
    return len(set(col_neighbors))

# return the pheromone trail of the pair (node,adj_node)
def si(self, node, adj_node):
    return phero_matrix[node, adj_node]

# select next candidate node according to the transition rule
def next_candidate(self):
    if (len(self.unvisited)==0):
       candidate = None
    elif (len(self.unvisited)==1):
        candidate = self.unvisited[0]
    else:
        max_value = 0
        heuristic_values = []
        candidates = []
        candidates_available = []
        for j in self.unvisited:
            heuristic_values.append((self.si(self.start, j)**self.alpha)*(self.dsat(j)**self.beta))
            candidates.append(j)
        max_value = max(heuristic_values)
        for i in range(len(candidates)):
            if (heuristic_values[i] >= max_value):
               candidates_available.append(candidates[i])
        candidate = random.choice(candidates_available)
    self.start = candidate
    return candidate

# return your own pheromone trail
def pheromone_trail(self):
    phero_trail = np.zeros((number_nodes, number_nodes), float)
    for i in g_nodes_int:
        for j in g_nodes_int:
            if (self.colors_assigned[i]==self.colors_assigned[j]):
                phero_trail[i,j] = 1
    return phero_trail

# consistency check --> should always return 0
def colisions(self):
    colisions = 0
    for key in self.colors_assigned:
```

```
            node = key
            col = self.colors_assigned[key]
            # check colors of neighbours
            for j in range(number_nodes):
                if (adj_matrix[node, j]==1 and self.colors_assigned[j]==col):
                    colisions = colisions+1
        return colisions

# take input from the txt.file and create an undirected graph
def create_graph(path):
    global number_nodes
    g = nx.Graph()
    f = open(path)
    n = int(f.readline())
    for i in range(n):
        graph_edge_list = f.readline().split()
        # convert to int
        graph_edge_list[0] = int(graph_edge_list[0])
        graph_edge_list[1] = int(graph_edge_list[1])
        # build graph
        g.add_edge(graph_edge_list[0], graph_edge_list[1])
    return g


#draw the graph and display the weights on the edges
def draw_graph(g, col_val):
            pos = nx.spring_layout(g)
            values = [col_val.get(node, 'blue') for node in g.nodes()]
            nx.draw(g, pos, with_labels = True, node_color = values, edge_color = 'black' ,width = 1, alpha = 0.7)
#with_labels=true is to show the node number in the output graph

# initiate a selection of colors for the coloring and compute the min. number of colors needed for a proper coloring
def init_colors(g):
    # grundy (max degree+1)
    colors = []
    grundy = len(nx.degree_histogram(g))
    for c in range(grundy):
      colors.append(c)
    return colors

# create a pheromone matrix with init pheromone values: 1 if nodes not adjacent, 0 if adjacent
def init_pheromones(g):
    phero_matrix = np.ones((number_nodes, number_nodes), float)
```

```
    for node in g:
        for adj_node in g.neighbors(node):
            phero_matrix[node, adj_node] = 0
    return phero_matrix

# calculate the adjacency matrix of the graph
def adjacency_matrix(g):
    adj_matrix = np.zeros((number_nodes, number_nodes), int)
    for node in g_nodes_int:
        for adj_node in g.neighbors(node):
            adj_matrix[node, adj_node] = 1
    return adj_matrix

# create new colony
def create_colony():
    ants = []
    ants.extend([Ant().initialize(g, colors) for i in range(number_ants)])
    return ants

# apply decay rate to the phero_matrix
def apply_decay():
    for node in g_nodes_int:
        for adj_node in g_nodes_int:
            phero_matrix[node, adj_node] = phero_matrix[node, adj_node]*(1-phero_decay)



# select colony's best solution
# update pheromone_matrix according to the elite solution
# return elite solution (coloring) with its distance (number of used colors)
def update_elite():
    global phero_matrix
    # select elite
    best_dist = 0
    elite_ant = None
    for ant in ants:
        if (best_dist==0):
            best_dist = ant.distance
            elite_ant = ant
        elif (ant.distance < best_dist):
            best_dist = ant.distance
            elite_ant = ant
    # update global phero_matrix
    elite_phero_matrix = elite_ant.pheromone_trail()
```

```
    phero_matrix = phero_matrix + elite_phero_matrix
    return elite_ant.distance, elite_ant.colors_assigned


# ------------ entry point ------------
# param input_graph - a networkx graph to be colored (node coloring)
# param num_ants - number of ants in the colony
# param iter - number of iterations to be performed
# param a - relative importance of elite pheromones
# param b - relative importance of heuristic value (DSAT)
# param decay - evaporation of pheromones after each iteration
def solve(input_graph, num_ants=10, iter=10, a=1, b=3, decay=0.8):
    global g # graph to be colored (a networkx graph)
    global number_nodes
    global g_nodes_int
    global number_ants
    global alpha
    global beta
    global phero_decay
    global adj_matrix
    global phero_matrix
    global colors
    global ants

    # params
    g = input_graph
    number_ants=num_ants
    number_iterations=iter
    alpha = a # relative importance of pheromone (si_ij)
    beta = b # relative importance of heuristic value (n_ij)
    phero_decay=decay # rate of pheromone decay

    # results
    final_solution = {} # coloring of the graph
    final_costs = 0 # number of colors in the solution
    iterations_needed = 0

    # init
    number_nodes = nx.number_of_nodes(g)
    g_nodes_int = []
    for node in g.nodes():  #nodes_iter -> nodes
        g_nodes_int.append(node)
    g_nodes_int = list(map(int, sorted(g_nodes_int)))
```

```
    adj_matrix = adjacency_matrix(g)
    colors = init_colors(g)
    phero_matrix = init_pheromones(g)

    # ACO_GCP daemon
    for i in range(number_iterations):
        # create colony
        ants = []
        ants = create_colony()
        # let colony find solutions
        for ant in ants:
            ant.colorize()
        # apply decay rate
        apply_decay()
        # select elite and update si_matrix
        elite_dist, elite_sol = update_elite()
        # estimate global solution so far
        if (final_costs==0):
            final_costs = elite_dist
            final_solution = elite_sol
            iterations_needed = i+1
        elif (elite_dist<final_costs):
            final_costs = elite_dist
            final_solution = elite_sol
            iterations_needed = i+1
    return final_costs, final_solution, iterations_needed


# global vars
g = None # graph to be colored
number_nodes = 0
g_nodes_int = []
number_ants = 0
alpha = 0
beta = 0
phero_decay = 0
adj_matrix = np.zeros((number_nodes, number_nodes), int)
phero_matrix = np.ones((number_nodes, number_nodes), float)
colors = []
ants = []


# Adjacent Matrix: 노드랑 선이 연결되면 1 아니면 0
```

```
G_a = np.array([
    [0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #통계학_문1
    [1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #통계학_문2
    [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #통계학_문3
    [0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #확통_오 d
    [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #확통_송1 e
    [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #확통_송2 f
    [0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #베이지안_오
    [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #표조실_차 h
    [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #손보_차(9) i
    [0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #경경분_신(10)
    [0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #시계열_신(11) k
    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #자분특_주(12) l
    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],      #고자분_주(13)m
    [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],      #GLM_이(14)n
    [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],      #범주형_이(15)
    [0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0],      #고통_김(16) p
    [0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,1,1,0,1,0],            #기확_김(17) q
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,1,0],      #기확_안(18) r
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,1,0,1,0],    #회귀_유1(19) s
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,1,0,1,0],    #회귀_유2(20) t
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,1,0,1,0],    #통프_안(21) u
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,1,1,0],    #데마_송(22) v
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0],    #계특_송(23) w
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,1],    #수통_소(24) x
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0],    #이통_소(25) y

])

#Ant cols
G1 = nx.from_numpy_array(G_a)
start = time.time() # 시작

solve(G1)
print(f"{time.time()-start:.4f} sec")

print(solve(G1))

# a: 0,
# b: 1,
# c: 2,
# d: 0,
# e: 1,
```

```
  # f: 0,
  # g: 1,
  # h: 0,
  # i: 1,
  # j: 0,
  # k: 1,
  # l: 0,
  # m: 1,
  # n: 0,
  # o: 1,
  # p: 0,
  # q: 1,
  # r: 1,
  # s: 5,
  # t: 2,
  # u: 4,
  # v: 0,
  # w: 1,
  # x: 3,
  # y: 0

  #색칠하기

  G = nx.Graph()
  G.add_nodes_from(['통_문1','통_문2','통_문3','확통_오','확통_송1','확통_송2','베이지안_오','표조실_차','손보_차','경경분_신','시계열_신','자분특_주',

'고자분_주','GLM_이','범주형_이','고통_김','기확_김','기확_안','회귀_유1','회귀_유2','통프_안','데마_송','계특_송','수통_소','이통_소'])


  G.add_edges_from([('통_문1','통_문2'), ('통_문1','통_문3'), ('통_문2','통_문3'),
                    ('확통_오','베이지안_오'),('확통_송1','확통_송2'),
                    ('표조실_차','손보_차'),('경경분_신','시계열_신'),
                    ('자분특_주','고자분_주'),
                    ('GLM_이','범주형_이'),
                    ('고통_김','기확_김'),
                    ('기확_김','회귀_유1'),('기확_김','회귀_유2'),('기확_김','수통_소'),('기확_김','데마_송'),('기확_김','통프_안'),
                    ('기확_안','회귀_유1'),('기확_안','회귀_유2'),('기확_안','수통_소'),('기확_안','데마_송'),('기확_안','통프_안'),

('통프_안','기확_안'),('통프_안','기확_김'),('통프_안','회귀_유1'),('통프_안','회귀_유2'),('통프_안','수통_소'),('통프_안','데마_송'),

('데마_송','기확_김'),('데마_송','회귀_유1'),('데마_송','회귀_유2'),('데마_송','수통_소'),('데마_송','통프_안'),('데마_송','기확_안'),('
```

```
데마_송','계특_송'),

('수통_소','기확_김'),('수통_소','기확_안'),('수통_소','통프_안'),('수통_소','데마_송'),('수통_소','회귀_유1'),('수통_소','회귀_유2'),('
수통_소','이통_소'),

('회귀_유1','기확_김'),('회귀_유1','기확_안'),('회귀_유1','통프_안'),('회귀_유1','데마_송'),('회귀_유1','수통_소'),('회귀_유1','회귀_
유2'),
                ('회귀_유2','기확_김'),('회귀_유2','기확_안'),('회귀_유2','통프_안'),('회귀_유2','데마_송'),('회귀_유2','수통_소')])

  pos1 = nx.kamada_kawai_layout(G)
  plt.figure(figsize=(10,10))

  color_map = ['pink','orange','yellow','pink','orange',
               'pink','orange','pink','orange','pink',
               'orange','pink','orange','pink','orange',
               'pink','orange','orange','purple','yellow',
               'blue','pink','orange','green','pink']

  nx.draw(G, pos1, with_labels=True,font_family='NanumBarunGothic', node_color = color_map ,node_size = 1500)
  plt.show()

  # pink 색0: [a,d,f,h,j,l,n,p,v,y]
  # orange 색1: [b,e,g,i,k,m,o,q,r,w]
  # yellow 색2: [c,t]
  # green 색3: [x]
  # blue 색4:[u]
  # purple 5:[s]


  ### tabucol (6개 색)

  from collections import deque
  from random import randrange

  def tabucol(graph, number_of_colors, tabu_size=7, reps=100, max_iterations=10000, debug=False):
      # graph is assumed to be the adjacency matrix of an undirected graph with no self-loops
      # nodes are represented with indices, [0, 1, ..., n-1]
      # colors are represented by numbers, [0, 1, ..., k-1]
      colors = list(range(number_of_colors))
      # number of iterations of the tabucol algorithm
      iterations = 0
      # initialize tabu as empty queue
      tabu = deque()
```

```
# solution is a map of nodes to colors
# Generate a random solution:
solution = dict()
for i in range(len(graph)):
    solution[i] = colors[randrange(0, len(colors))]

# Aspiration level A(z), represented by a mapping: f(s) -> best f(s') seen so far
aspiration_level = dict()

while iterations < max_iterations:
    # Count node pairs (i,j) which are adjacent and have the same color.
    move_candidates = set()  # use a set to avoid duplicates
    conflict_count = 0
    for i in range(len(graph)):
        for j in range(i+1, len(graph)):  # assume undirected graph, ignoring self-loops
            if graph[i][j] > 0: # adjacent
                if solution[i] == solution[j]:  # same color
                    move_candidates.add(i)
                    move_candidates.add(j)
                    conflict_count += 1
    move_candidates = list(move_candidates)  # convert to list for array indexing

    if conflict_count == 0:
        # Found a valid coloring.
        break

    # Generate neighbor solutions.
    new_solution = None
    for r in range(reps):
        # Choose a node to move.
        node = move_candidates[randrange(0, len(move_candidates))]

        # Choose color other than current.
        new_color = colors[randrange(0, len(colors) - 1)]
        if solution[node] == new_color:
            # essentially swapping last color with current color for this calculation
            new_color = colors[-1]

        # Create a neighbor solution
        new_solution = solution.copy()
        new_solution[node] = new_color
        # Count adjacent pairs with the same color in the new solution.
```

```
            new_conflicts = 0
            for i in range(len(graph)):
                for j in range(i+1, len(graph)):
                    if graph[i][j] > 0 and new_solution[i] == new_solution[j]:
                        new_conflicts += 1
            if new_conflicts < conflict_count:  # found an improved solution
                # if f(s') <= A(f(s)) [where A(z) defaults to z - 1]
                if new_conflicts <= aspiration_level.setdefault(conflict_count, conflict_count - 1):
                    # set A(f(s) = f(s') - 1
                    aspiration_level[conflict_count] = new_conflicts - 1

                    if (node, new_color) in tabu: # permit tabu move if it is better any prior
                        tabu.remove((node, new_color))
                        if debug:
                            print("tabu permitted;", conflict_count, "->", new_conflicts)
                        break
                else:
                    if (node, new_color) in tabu:
                        # tabu move isn't good enough
                        continue
                if debug:
                    print (conflict_count, "->", new_conflicts)
                break

        # At this point, either found a better solution,
        # or ran out of reps, using the last solution generated

        # The current node color will become tabu.
        # add to the end of the tabu queue
        tabu.append((node, solution[node]))
        if len(tabu) > tabu_size:  # queue full
            tabu.popleft()  # remove the oldest move

        # Move to next iteration of tabucol with new solution
        solution = new_solution
        iterations += 1
        if debug and iterations % 500 == 0:
            print("iteration:", iterations)

# At this point, either conflict_count is 0 and a coloring was found,
# or ran out of iterations with no valid coloring.
    if conflict_count != 0:
        print("No coloring found with {} colors.".format(number_of_colors))
```

```
        return None
    else:
        print("Found coloring:\n", solution)
        print("itertaions :\n", iterations)
        print("number of colors :\n", len(set(solution.values())))


start = time.time()

tabucol(G, 100)
print(f"{time.time()-start:.4f} sec")

# a: 1,
# b: 5,
# c: 4,
# d: 5,
# e: 5
# f: 3
# g: 2,
# h: 1,
# i: 0,
# j: 0,
# k: 1,
# l: 0,
# m: 1,
# n: 3,
# o: 5,
# p: 5,
# q: 1,
# r: 1,
# s: 0,
# t: 4,
# u: 5,
# v: 3,
# w: 1,
# x: 2,
# y: 4

start = time.time()

tabucol(G, 11)
print(f"{time.time()-start:.4f} sec")
```

backtracking(7개 색)

```
# A utility function to check
# if the current color assignment
# is safe for vertex v
def isSafe(graph, v, colour, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and colour[i] == c:
            return False
    return True

# A recursive utility function to solve m
# coloring  problem
def graphColourUtil(graph, m, colour, v):
    if v == len(graph):
        return True

    for c in range(1, m + 1):
        if isSafe(graph, v, colour, c) == True:
            colour[v] = c
            if graphColourUtil(graph, m, colour, v + 1) == True:
                return True
            colour[v] = 0


def Backtracking(graph, m):
    colour = [0] * len(graph)
    if graphColourUtil(graph, m, colour, 0) == None:
        return False

    # Print the solution
    print("Solution exist and Following are the assigned colours:")
    num=0
    for c in colour:
        print(c, end=' ')
    return True, len(set(colour))

start = time.time()
print(Backtracking(G,100))
print(f"{time.time()-start:.4f} sec")

# a:1
# b 2
```

c 3
d 1
e 1
f 2
g 2
h 1
i 2
j 1
k 2
l 1
m 2
n 1
o 2
p 1
q 2
r 1
s 3
t 4
u 5
v 6
w 1
x 7
y 1