**17.1**

# Graph Colouring

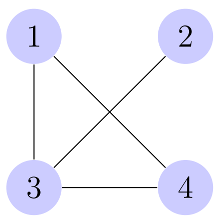
## Contents

## Introduction

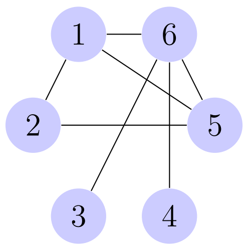
This project is programmed in **Python 3.4**. Consult section A for program documentation, listings and information on the structure of the programming for the project, as appropriate. This report is written in $\LaTeX\,2_\varepsilon$.

# 1 The Greedy Algorithm

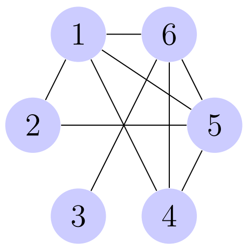
**Q1.** The function `Greedy`$(G, O)$ runs the greedy algorithm on the graph $G$ using vertex ordering $O$.[1] As test data, we'll refer to some example graphs for which manual computations are simple.



$G_1$



$G_2$



$G_3$

Colouring $G_1$ with ordering $(1, 2, 3, 4)$ yields the colouring $(1, 1, 2, 3)$ (as a function whose domain is the vertex set, namely $[4]$). $G_2$ with ordering $(2, 5, 6, 3, 4, 1)$ yields colouring $(3, 1, 2, 2, 2, 1)$. And sure enough, the algorithm yields the following:

```
>>> Greedy(G1,[0,1,2,3,4])
([0, 1, 1, 2, 3], 3)
```

```
>>> Greedy(G2,[0,2,5,6,3,4,1])
([0, 3, 1, 2, 2, 2, 1], 3)
```

The vertex orderings for a graph $G$ described as *(i)*, *(ii)*, *(iii)*, *(iv)* are respectively implemented as the functions `OrdA`$(G)$, `OrdB`$(G)$, `OrdC`$(G)$, `OrdD`$(G)$. Testing on $G_3$, we have, as expected:

```
>>> OrdA(G3)
[0, 3, 2, 4, 1, 5, 6]
>>> OrdB(G3)
[0, 6, 5, 1, 4, 2, 3]
```

```
>>> OrdC(G3)
[0, 6, 5, 4, 1, 2, 3]
>>> OrdD(G3)
[0, 1, 6, 5, 3, 4, 2]
```

We compare the efficacy of the greedy algorithm powered by these ordering algorithms by generating ten random graphs from $\mathcal{G}(70, 0.5)$ and $\mathcal{G}_3(70, 0.75)$, and finding the number of colours used. Note that the function `Graph`$(n, p, k)$ generates a random graph from $\mathcal{G}_k(n, p)$ (where $\mathcal{G}_0 := \mathcal{G}$). In the following, each row represents a particular random graph from these spaces.

| i | ii | iii | iv |
|---|---|---|---|
| 19 | 14 | 16 | 17 |
| 18 | 14 | 14 | 18 |
| 19 | 15 | 16 | 17 |
| 19 | 16 | 16 | 17 |
| 18 | 16 | 15 | 16 |
| 16 | 15 | 14 | 16 |
| 17 | 16 | 14 | 17 |
| 18 | 15 | 15 | 16 |
| 19 | 15 | 15 | 16 |
| 17 | 15 | 14 | 18 |

`Q1a(70,0.5,0)` ($\mathcal{G}$)

| i | ii | iii | iv |
|---|---|---|---|
| 7 | 3 | 3 | 4 |
| 9 | 3 | 3 | 3 |
| 8 | 4 | 3 | 3 |
| 9 | 3 | 3 | 3 |
| 9 | 4 | 3 | 3 |
| 6 | 3 | 3 | 6 |
| 6 | 4 | 3 | 3 |
| 4 | 3 | 3 | 3 |
| 5 | 3 | 3 | 3 |
| 7 | 3 | 3 | 4 |

`Q1a(70,0.75,3)` ($\mathcal{G}_3$)

First, we'll compare general trends that differentiate between the different ordering algorithms. Observe foremost that *(i)* tends to use more colours than *(iv)*, which in turn tends to

[1]Check [] for a description of how data types are implemented in this project.

use more than *(ii)* and *(iii)*. To corroborate these observations, here are the mean number of colours used by each, as well as the proportion of times a given algorithm is optimal among the four, for a sample of size 10,000.

```
G           i    ,  ii   ,  iii  ,  iv
Mean:     17.69,  15.23,  15.54,  16.48
Optimal: 0.020,  0.762,  0.557,  0.167

G3          i    ,  ii   ,  iii  ,  iv
Mean:      5.66,   3.22,   3.00,   4.05
Optimal: 0.100,  0.813,  1.000,  0.406
                                  Q1b
```

These two arguments can be explained intuitively. Firstly, we expect *(ii)* to be better than the average (represented by *(iv)*) because they colour the vertices of higher degree earliest, when few colours have been assigned to any vertex. These vertices thus receive low colours, and later vertices also receive low colours as they have fewer neighbours. More precisely, we have, for an ordering of vertices $(x_i)_{i=1}^n$,

$$\chi(G) \leq \max_{i\in[1,n]}(\min(d(x_i)+1, i))$$

This also suggests why *(i)* performs worse than average; it colours the vertices of highest degree last, when there are both many neighbours for each vertex and many colours in use that they may have already been assigned. Indeed, this ordering typically results in large colours being expended towards the end, as in this example:

```
>>> G=Graph(15,0.5,0); OrdA(G); Greedy(G,OrdA(G))
[0, 12, 13, 15, 4, 1, 6, 9, 5, 7, 10, 11, 14, 8, 3, 2]
([0, 1, 6, 5, 1, 2, 3, 3, 4, 4, 1, 3, 1, 1, 2, 2], 6)
```

It may be expected that ordering *(iii)* is an improvement over *(ii)*, since it greedily optimises for each vertex to have the fewest neighbours among those already coloured, and so, denoting $d'(x_i) := |\Gamma(x_j)_{j=1}^{i-1}| \leq \min(d(x_i), i-1)$,

$$\chi(G) \leq \max_{i\in[1,n]}(d'(x_i)+1) \leq \max_{i\in[1,n]}(\min(d(x_i)+1, i))$$

However, it evidently performs worse in the case of $\mathcal{G}(70, 0.5)$ than *(ii)*. Surprisingly, in the case of $\mathcal{G}_3(70, 0.75)$, it not only performs better but seems to always find a 3-colouring (which is likely optimal, noting that graphs in $\mathcal{G}_3(70, 0.75)$ are tripartite, partitioned into cosets modulo 3, but unlikely to be bipartite, as there is likely a 3-cycle between the parts).

However, it sometimes uses more than 3 colours (though examples are rare[2]); more generally, it can be seen that none of these orderings on either $\mathcal{G}(70, 0.5)$ or $\mathcal{G}_3(70, 0.75)$ always produces an optimal colouring.

**Q2.** The spaces $\mathcal{G}(70, 0.5)$ and $\mathcal{G}_3(70, 0.75)$ are comparable in the sense that the expected number of edges is similar in both. The number of edges in a member is binomially-distributed with mean the product of the number of edges being tested for inclusion with the probability of each being accepted – i.e. $\binom{70}{2} \times 0.5 = 1207.5$ in $\mathcal{G}(70, 0.5)$ and $(\binom{70}{2} - \binom{24}{2} - 2\binom{23}{2}) \times 0.75 = 1224.75$, respectively.[3]

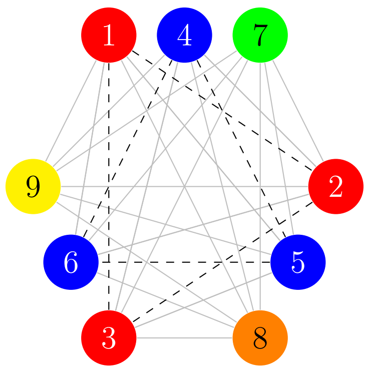
Nonetheless, the members of $\mathcal{G}_3(70, 0.75)$ typically have very different structure to those of $\mathcal{G}(70, 0.5)$. They are all tripartite, partitioned into cosets modulo 3, and can thus be 3-coloured

[2]During my testing, I found it typically took at least 2000 trials before one showed up.

[3]Hence the choice of $p = 0.75$.

by the greedy algorithm with ordering a concatenation of listings for each coset – for example, $(1, 4, ..., 67, 70, 2, 5, ..., 65, 68, 3, 6, ..., 66, 69)$.

Conversely, $\mathcal{G}_3(3n)$ $(n \in \mathbb{N})$ contains graphs for which there exists an ordering that coerces the greedy algorithm into using $n + 2$ colours. This is obtained by completing the tripartition (inserting all edges between vertices in different parts), partitioning the entire vertex set into 3-sets that contain one element from each part, deleting the edges inside each 3-set except one special one, and colouring each 3-set sequentially, making sure to leave the special one until last. Or, in a picture (for $n = 3$):[4]



## 2 The Clique Algorithm

**Q3.** *The greedy-type algorithm $\mathfrak{G}$ on $G$ is unlikely to find a complete subgraph of order 14.*

Consider the event $A_n = [\mathfrak{G}$ returns a complete subgraph of order $\geq n]$. For any $G \in A$, $\mathfrak{G}$ must have already found a complete subgraph $H$ of order 13, and furthermore, some other vertex must have been adjacent to every vertex in $H$ (else $H$ would've been returned – contradiction). Hence,

$$\mathbb{P}(A_{14}) = \mathbb{P}(A_{13})\mathbb{P}(A_{14}|A_{13}) \leq (2000 - 13) \times 0.5^{13} < 0.243 < 0.5$$

*However, the clique number $\omega(G)$ of $G$ is both likely to be at least and at most 17.*

This comes from a standard threshold argument; we quote results from *Imre Leader's 2007 lecture notes*, specifically the section *Structure of a Random Graph*. Let $k \in \mathbb{N}$, $X_k : \mathcal{G}(2000, 0.5) \to \mathbb{N}_0$, $G \mapsto |\{H \leq G : H \textbf{ is complete and } |H| = k\}|$. Let $\mu_k = \mathrm{E}(X)$, $V_k = \mathrm{Var}(X)$. Via Markov's and Chebyshev's inequalities, it follows that

$$1 - \mu_k \leq \mathbb{P}(X = 0) \leq V_k/\mu_k^2$$

By combinatorics,[5]

$$\mu_k = \binom{n}{k} p^{\binom{k}{2}} \qquad\qquad V_k = \mu \sum_{s=2}^{k} \binom{k}{s}\binom{n-k}{k-s} p^{\binom{k}{2}} \left(p^{-\binom{s}{2}} - 1\right)$$

Hence,

$$\begin{aligned} 1 - \mu_{17} &< -2.95 < 0.5 & \qquad V_{17}/\mu_{17}^2 &< 0.44 < 0.5 \\ 1 - \mu_{18} &> 0.997 > 0.5 & \qquad V_{18}/\mu_{18}^2 &> 394 > 0.5 \end{aligned}$$

[4]With edges in grey and absent edges between parts dotted, and numbers representing the vertex ordering used by the greedy algorithm.

[5]I'll defer to the dear Leader on this one.

So,

$$\mathbb{P}(X_{18} = 0) \geq 1 - \mu_{18} > 0.5 \qquad\qquad \mathbb{P}(X_{17} = 0) \leq V_{17}/\mu_{17}^2 < 0.5$$

$k$-complete subgraph containment is a decreasing property (in $k$), so $\forall k \in \mathbb{N}$ $\mathbb{P}(X_k = 0) = \mathbb{P}(\omega(G) \leq k-1)$, and so we have $\mathbb{P}(\omega(G) \leq 17) > 0.5$ and $\mathbb{P}(\omega(G) \geq 17) > 0.5$. In particular, the median clique number equals 17.

**Q4.** The function `KS`$(G, O, k)$ distinctly enumerates the complete subgraphs (if $k = 0$) or independent subsets (if $k = 1$) of the graph $G$ in an order determined by vertex listing $O$. Each time it's called, it greedily finds a maximal complete subgraph/independent subset, having removed and skipped the last vertex it appended last time it was called (if appropriate). For example $G_3$ with ordering $(1, 2, 3, 4, 5, 6)$, it's easy to check that this yields the following listing of independent sets.

```
>>> for x in KS(G3
    ,[0,1,2,3,4,5,6],0)
    : print(x)

[1, 2, 5]
[1, 2]
[1, 4, 5, 6]
[1, 4, 5]
[1, 4, 6]
[1, 4]
[1, 5, 6]
[1, 5]
[1, 6]
[1]
[2, 5]
[2]
[3, 6]
[3]
[4, 5, 6]
[4, 5]
[4, 6]
[4]
[5, 6]
[5]
[6]
[]
```

Likewise, the independent subsets of $G_1$ with ordering $(3, 2, 1, 4)$:

```
for x in KS(G1
    ,[0,3,2,1,4],1):
    print(x)
[3]
[2, 1]
[2, 4]
[2]
[1]
[4]
[]
```

The function `Clique`$(G, O, k)$ uses `KS` to find a clique ($k = 0$) or largest independent set ($k = 1$) in $G$ according to $O$.[6] Let's test it out on the previous random graph spaces.

| i | ii | iii | iv | Clique |
|---|---|---|---|---|
| 17 | 15 | 15 | 16 | 8 |
| 18 | 14 | 15 | 16 | 9 |
| 18 | 16 | 16 | 17 | 9 |
| 17 | 15 | 16 | 17 | 8 |
| 17 | 16 | 14 | 15 | 8 |
| 18 | 15 | 15 | 16 | 9 |
| 18 | 15 | 16 | 16 | 9 |
| 18 | 15 | 16 | 17 | 8 |
| 18 | 15 | 16 | 17 | 8 |
| 17 | 14 | 17 | 18 | 8 |

`Q4(70,0.5,0)` ($\mathcal{G}(70, 0.5)$)

| i | ii | iii | iv | Clique |
|---|---|---|---|---|
| 5 | 4 | 3 | 4 | 3 |
| 6 | 3 | 3 | 6 | 3 |
| 4 | 3 | 3 | 4 | 3 |
| 4 | 3 | 3 | 4 | 3 |
| 6 | 3 | 3 | 5 | 3 |
| 6 | 3 | 3 | 4 | 3 |
| 3 | 3 | 3 | 8 | 3 |
| 12 | 3 | 3 | 4 | 3 |
| 3 | 3 | 3 | 3 | 3 |
| 4 | 3 | 3 | 5 | 3 |

`Q4(70,0.75,3)` ($\mathcal{G}_3(70, 0.75)$)

As expected, the (exact) clique numbers $\omega(G)$ computed don't give much information on the colouring numbers $\chi(G)$. As a sanity check, it is corroborated that $\omega(G) \leq \chi(G)$ in all cases. But this inequality may be strict, and the upper bounds on $\chi(G)$ given by the greedy algorithm may not be tight. For $\mathcal{G}_k(n, p)$ $(k \in \mathbb{N})$, $\omega(G) \leq \chi(G) \leq k$. In the case $k = 3$ (and sufficiently high $p$), there is likely to be a 3-cycle, whence $\omega(G) = 3$.

[6]The size of the subset will obviously be independent of $O$.

## 3 The Independent Set Algorithm

**Q5.** This algorithm is implemented as `IndCol`$(G, O)$, colouring a graph $G$ w.r.t. vertex ordering $O$ (which influences the decomposition into independent subsets). Here is some data. `G` represents the greedy algorithm, `I` represents the independent set algorithm, and the Roman numerals index the ordering algorithms as before – we'll focus on *(ii)* and *(iii)* because they tend to use fewer colours, also when used with `I`.

| Gii | Giii | Cliq. | Iii | Iiii |
|---|---|---|---|---|
| 15 | 15 | 9 | 15 | 14 |
| 15 | 16 | 9 | 14 | 14 |
| 14 | 14 | 8 | 14 | 14 |
| 16 | 15 | 8 | 14 | 14 |
| 15 | 15 | 8 | 14 | 14 |
| 15 | 16 | 8 | 14 | 13 |
| 16 | 16 | 8 | 13 | 14 |
| 15 | 16 | 8 | 13 | 15 |
| 16 | 15 | 8 | 13 | 14 |
| 16 | 16 | 9 | 13 | 13 |

`Q5(70,0.5,0)`

| Gii | Giii | Cliq. | Iii | Iiii |
|---|---|---|---|---|
| 13 | 12 | 7 | 7 | 7 |
| 9 | 13 | 7 | 8 | 8 |
| 13 | 11 | 7 | 7 | 7 |
| 14 | 12 | 7 | 7 | 8 |
| 13 | 12 | 7 | 7 | 7 |
| 12 | 13 | 7 | 8 | 7 |
| 13 | 13 | 7 | 9 | 9 |
| 13 | 13 | 7 | 7 | 7 |
| 11 | 12 | 7 | 8 | 8 |
| 12 | 12 | 7 | 7 | 7 |

`Q5(70,0.5,7)`

For all but one test case, the new algorithm has yielded a more frugal, yet invariably slower, colouring. The improvement is significant in the $\mathcal{G}_l(n, p)$ case. Intuitively, since the parts of the partition into cosets modulo $p$ are independent sets, the algorithm will find sets at least as large at each step, which are likely to comprise significant parts of particular cosets if $p$ is high (and other potential independent sets are thus forbidden). Nevertheless, missing edges between parts lead to imperfections and can inflate the number of colours used. Thus, we would predict that, for $\mathcal{G}_l(n, p)$, the colourings should become more frugal as $p$ increases *despite* the presence of more edges. And sure enough, that is what happens.

| Gii | Giii | Cliq. | Iii | Iiii |
|---|---|---|---|---|
| 11 | 12 | 7 | 10 | 10 |
| 11 | 11 | 6 | 10 | 10 |
| 10 | 11 | 6 | 9 | 7 |
| 12 | 11 | 6 | 8 | 9 |
| 11 | 10 | 6 | 10 | 9 |
| 11 | 11 | 6 | 9 | 10 |
| 11 | 12 | 6 | 8 | 9 |
| 11 | 11 | 7 | 8 | 10 |
| 10 | 11 | 6 | 9 | 8 |
| 10 | 10 | 6 | 9 | 9 |

`Q5(70,0.4,7)`

| Gii | Giii | Cliq. | Iii | Iiii |
|---|---|---|---|---|
| 8 | 7 | 7 | 7 | 7 |
| 13 | 10 | 7 | 7 | 7 |
| 14 | 12 | 7 | 7 | 7 |
| 11 | 13 | 7 | 7 | 7 |
| 13 | 10 | 7 | 7 | 8 |
| 14 | 11 | 7 | 7 | 7 |
| 14 | 15 | 7 | 8 | 7 |
| 14 | 14 | 7 | 7 | 7 |
| 15 | 7 | 7 | 7 | 7 |
| 13 | 12 | 7 | 7 | 7 |

`Q5(70,0.6,7)`

For our $\mathcal{G}(n, p)$ graphs, which generally lack the structure of a partition into large independent sets, the number of colours used is, as usual, positively correlated with (edge) size.

```
Gii    Giii   Cliq.  Iii    Iiii          Gii    Giii   Cliq.  Iii    Iiii
13     13      7     12     12            18     19     10     16     16
13     13      7     11     11            18     19     10     16     17
12     12      7     11     11            18     19     10     16     16
14     13      7     12     12            19     19     11     16     16
13     13      7     12     11            20     18     10     18     16
12     13      7     10     11            18     18     10     17     17
13     13      7     11     12            18     20     10     16     17
13     13      7     12     11            19     19     10     18     16
13     13      7     12     11            19     19     10     17     17
12     13      6     12     11            19     18     10     17     16
         Q5(70,0.4,0)                              Q5(70,0.6,0)
```

**Q6.** The most naive graph colouring procedure would test out every possible function $V(G) \mapsto [1, k]$ ($k \in \mathbb{N}$) to check if it's a valid colouring (using a lexicographical ordering w.r.t. some vertex listing), incrementing $k$ if it has exhausted all functions (starting with $k = 1$). Once a valid colouring is found, it must be optimal, else it would've been discovered earlier.

## A Programs

*All functions are stored in* ***`D.py`***. *Program output printed in the report is generated by functions in* ***`Output.py`***, *as annotated throughout the report.* ***`InitD.py`*** *is a script that, when run in a Python shell, loads all functions into the global memory.*

```
import random as r

def Graph(n,p,k):
    G = [[] for i in range(n+1)]
    for i in range(1,n+1):
        for j in range(i+1,n+1):
            if k == 0 or (i - j) % k != 0:
                if r.random() <= p:
                    G[i].append(j)
                    G[j].append(i)
    return G

def deg(G):
    return [len(g) for g in G]

def OrdA(G):
    D = deg(G)
    O = [i for i in range(len(D))]
    d = 0; l = 1
    while True:
        for i in range(l,len(D)):
            if D[i] != d:
                l = i; break
        else:
            return O
        for i in range(l,len(D)):
            if D[i] == d:
                D[i],D[l] = D[l],D[i]
                O[i],O[l] = O[l],O[i]
                l += 1; break
        else:
            d += 1

def OrdB(G):
    return [0] + OrdA(G)[:0:-1]

def OrdC(G):
    G = list(G); O = [0]
    D = [len(G)] + [0 for i in range(1,len(G))]
    for n in range(1,len(G)):
        E = deg(G)
        D = [D[i] if D[i] == len(G) else E[i] for i in range(len
          (G))]
        m = min(range(len(D)),key=D.__getitem__)
        O.append(m)
        G = [[i for i in g if i != m] for g in G]
        G[m] = []
        D[m] = len(G)
    return [0] + O[:0:-1]
```

```
def OrdD(G):
    O = [i for i in range(1,len(G))]
    r.shuffle(O)
    return [0] + O

def Greedy(G,O):
    C = [0 for i in range(len(G))]; m = 1
    for i in O[1:]:
        N = [C[j] for j in G[i]]
        c = 1
        while True:
            if c in N: c += 1
            else: break
        C[i] = c
        if c > m: m = c
    return C,m

def KS(G,O,k):
    K = []; a = 1
    try:
        while True:
            for i in range(a,len(O)):
                g = G[O[i]]
                for j in K:
                    if (O[j] in g) == k: break
                else: K.append(i)
            yield [O[i] for i in K]
            a = K.pop() + 1
    except IndexError:
        return

def Clique(G,O,k):
    H = []
    for K in KS(G,O,k):
        if len(H) < len(K): H = list(K)
    return H, len(H)

def IndCol(G,O):
    G = list(G); O = list(O); C = [0 for g in G]; c = 0
    while len(O) != 1:
        c += 1
        A = Clique(G,O,1)[0]
        for a in A:
            C[a] = c
            G = [[i for i in g if i != a] for g in G]
            G[a] = []
            O.remove(a)
    return C,c
```