# FINAL PROJECT REPORT 2022

110511059 紀禹豪

# Problem I. Maximal Maximal K-clique

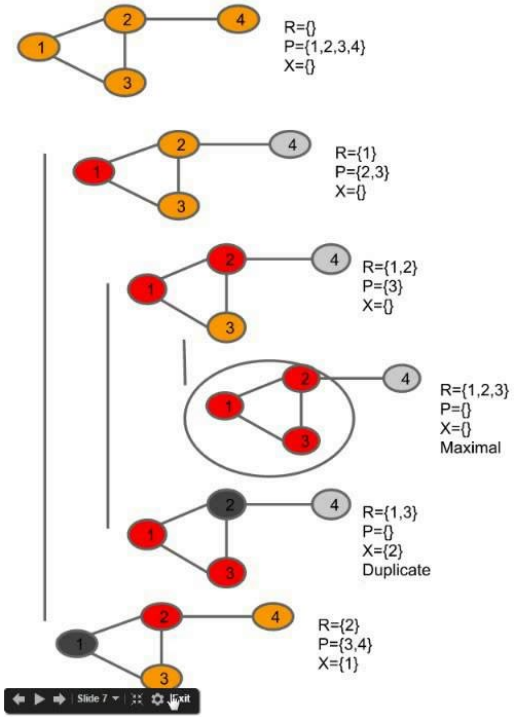
想法：

因為一開始並沒有甚麼頭緒

所以我就上網Google了“Algorithm for finding clique”

找到了一個演算法叫作 Bron–Kerbosch Algorithm

並決定使用它來完成第一題



# Bron–Kerbosch Algorithm

利用回朔法，一層一層紀錄，找出所有解
理論數值：
時間複雜度 $O(n \cdot 3^{n/3})$，選擇適當的 pivot ，讓各階段列舉的點都是最少，時間複雜度加速為 $O(3^{n/3})$ 。
點的列舉順序採用 degeneracy order ，時間複雜度加速為 $O(d \cdot n \cdot 3^{d/3})$ ， d 是原圖的 degeneracy

總共有三個集合用來存放Vertices，分別是
**R(all):** 目前的Clique。
**P(some):** 可以增大目前Clique的點集合。接下來要列舉的點。
（與目前Clique上所有點皆相鄰的點，構成的集合）
**X(none):** 可以增大目前Clique的點集合，但是先前已經列舉過。用來避免重複列舉。

所以一開始P包含所有Vertices，R、X皆為empty
接下來會選用其中一個Vertex作為Pivot，將其放入R，
P則會剩下所有和R(Clique)相鄰的Vertices

接著，繼續再將P中的點放入R，並再次更新P，直到P為空集合、X也為空集合時，我們就能找到一個Maximal Clique

最後會逆推，將P中一個不為Pivot的Vertex丟入X，並更新R、P，直到X數為pivot edge數時完成

完成後，則會繼續使用另一個Vertex作為Pivot，並重複上述步驟，找出所有Maximal Clique

# Code - Algorithm

```
void max_cliq(int d, int an, int sn, int nn)
{
    int u = some[d][0];

    for(size_t i = 0; i < sn; ++i)
    {
        int v = some[d][i];

        if(graph[u][v])
            continue;

        for(size_t j = 0; j < an; ++j)
            all[d+1][j] = all[d][j];
        all[d+1][an] = v;

        int nextsn, nextnn;
        nextsn = nextnn = 0;

        for(size_t j = 0; j < sn; ++j)
            if(graph[v][some[d][j]])
                some[d+1][nextsn++] = some[d][j];
        for(size_t j = 0; j < nn; ++j)
            if(graph[v][none[d][j]])
                none[d+1][nextnn++] = none[d][j];

        if(!nextsn && !nextnn)
            clique.push_back(vector<int>(all[d+1], all[d+1] + an + 1));

        max_cliq(d+1, an+1, nextsn, nextnn);
        some[d][i] = -1, none[d][nn++] = v;
    }
}
```

<-- Pivot

<-- 目前P數量，為0時代表結束

<-- u v相連，直接換下一個vertex

<-- 將取出的vertex加入下一層的R

<-- 下一層P、X及和數量為0，
則把R的vertices丟到vector儲存
(也就是Maximal Clique)

<-- 進入下一層

逆推->

# Code - Sort

目的：方便輸出

<- u v size 一樣，用vector class operator < 比大小
u v size 不一樣，直接比size大小

```
inline bool compare(const vector<int> &u, const vector<int> &v)
{
    return (u.size() == v.size()) ? u < v : u.size() < v.size();
}
void sort_2d_vec()
{
    for(size_t i = 0; i < clique.size(); ++i)
        sort(clique[i].begin(), clique[i].end());
    sort(clique.begin(), clique.end(), compare);
}
```

<- 一維vector由小排到大
使用內建的sort function

二維vector由小排到大
使用內建的sort function

**comp** - comparison function object (i.e. an object that satisfies the requirements of *Compare*) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

While the signature does not need to have `const &`, the function must not modify the objects passed to it and must be able to accept all values of type (possibly const) Type1 and Type2 regardless of value category (thus, `Type1 &` is not allowed, nor is `Type1` unless for Type1 a move is equivalent to a copy (since C++11)).
The types `Type1` and `Type2` must be such that an object of type `RandomIt` can be dereferenced and then implicitly converted to both of them.

std::sort

Defined in header <algorithm>

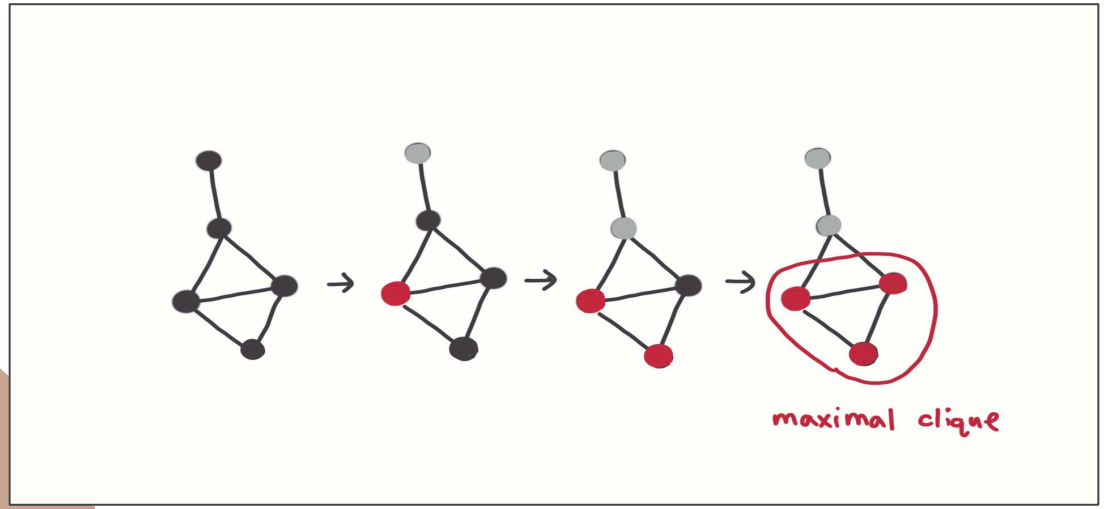
```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );                                   (1)  (until C++20)
template< class RandomIt >
constexpr void sort( RandomIt first, RandomIt last );                              (since C++20)

template< class ExecutionPolicy, class RandomIt >
void sort( ExecutionPolicy&& policy,                                          (2)  (since C++17)
           RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );                     (3)  (until C++20)
template< class RandomIt, class Compare >
constexpr void sort( RandomIt first, RandomIt last, Compare comp );                (since C++20)

template< class ExecutionPolicy, class RandomIt, class Compare >
void sort( ExecutionPolicy&& policy,                                          (4)  (since C++17)
           RandomIt first, RandomIt last, Compare comp );
```

# Other Method - Greedy

任選一Vertex開始不斷延伸，將共同連接點加入，直到沒有共同連接點為止時，
會得到一個Maximal Clique

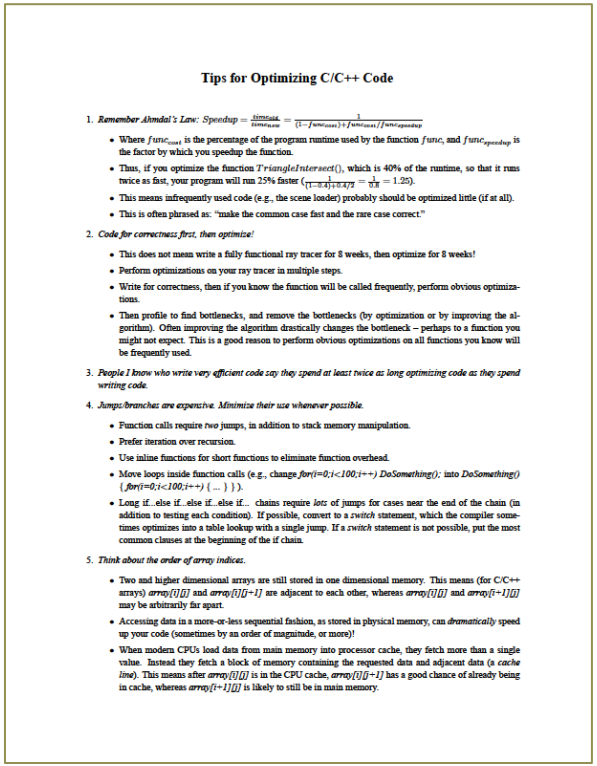
從不同Vertex開始，得到的Clique可能不同；將所有Vertices run過一輪可以得到所有Maximal Clique

時間複雜度：$\Theta(n^2)$



# Speed Up Runtime

参考：optimize.pdf



**Tips for Optimizing C/C++ Code**

1. ***Remember Ahmdal's Law:*** $Speedup = \frac{time_{old}}{time_{new}} = \frac{1}{(1-func_{cost})+\frac{func_{cost}}{func_{speedup}}}$
   - Where $func_{cost}$ is the percentage of the program runtime used by the function $func$, and $func_{speedup}$ is the factor by which you speedup the function.
   - Thus, if you optimize the function $TriangleIntersect()$, which is 40% of the runtime, so that it runs twice as fast, your program will run 25% faster ($\frac{1}{(1-0.4)+\frac{0.4}{2}} = \frac{1}{0.8} = 1.25$).
   - This means infrequently used code (e.g., the scene loader) probably should be optimized little (if at all).
   - This is often phrased as: "make the common case fast and the rare case correct."
2. ***Code for correctness first, then optimize!***
   - This does not mean write a fully functional ray tracer for 8 weeks, then optimize for 8 weeks!
   - Perform optimizations on your ray tracer in multiple steps.
   - Write for correctness, then if you know the function will be called frequently, perform obvious optimizations.
   - Then profile to find bottlenecks, and remove the bottlenecks (by optimization or by improving the algorithm). Often improving the algorithm drastically changes the bottleneck – perhaps to a function you might not expect. This is a good reason to perform obvious optimizations on all functions you know will be frequently used.
3. ***People I know who write very efficient code say they spend at least twice as long optimizing code as they spend writing code.***
4. ***Jumps/branches are expensive. Minimize their use whenever possible.***
   - Function calls require *two* jumps, in addition to stack memory manipulation.
   - Prefer iteration over recursion.
   - Use inline functions for short functions to eliminate function overhead.
   - Move loops inside function calls (e.g., change *for(i=0;i<100;i++) DoSomething();* into *DoSomething() { for(i=0;i<100;i++) { ... } }*).
   - Long if...else if...else if...else if... chains require *lots* of jumps for cases near the end of the chain (in addition to testing each condition). If possible, convert to a *switch* statement, which the compiler sometimes optimizes into a table lookup with a single jump. If a *switch* statement is not possible, put the most common clauses at the beginning of the if chain.
5. ***Think about the order of array indices.***
   - Two and higher dimensional arrays are still stored in one dimensional memory. This means (for C/C++ arrays) *array[i][j]* and *array[i][j+1]* are adjacent to each other, whereas *array[i][j]* and *array[i+1][j]* may be arbitrarily far apart.
   - Accessing data in a more-or-less sequential fashion, as stored in physical memory, can *dramatically* speed up your code (sometimes by an order of magnitude, or more)!
   - When modern CPUs load data from main memory into processor cache, they fetch more than a single value. Instead they fetch a block of memory containing the requested data and adjacent data (a *cache line*). This means after *array[i][j]* is in the CPU cache, *array[i][j+1]* has a good chance of already being in cache, whereas *array[i+1][j]* is likely to still be in main memory.

# Speed Up Runtime

1. **Better Algorithm**
   Greedy -> Bron-Kerbosch Algorithm

2. **Better Sorting Method**
   自定義函數排序 -> 使用內建函數sort()

3. **Loop**
   I. 減少不必要的for迴圈
   II. 使用++i 代替 i++，因為i++會額外儲存value
   III. 使用size_t代替int

4. **If else**
   使用conditional operator(?:)代替if else，可以加快速度

5. **Operator**
   盡量使用++、--、+=、*= ...

6. **Scanf/Printf**
   much more faster than **cin/cout**

7. **Memory usage**
   I. static array分配的空間盡量小一點
   II. 減少不必要的Variable
   III. 使用Global variable避免多次Pass by Value
   (但不用全部都使用Global Variable )

8. **Inline function**
   適合較小型的function，呼叫時間減短

9. **簡化部分code寫法**

10. **Pass by reference/pointer instead of value**

測試Runtime: 360 – 600

# Problem II.
# Chromatic Number & Graph Coloring

**想法：**因為上課有提過，所以想直接使用Greedy暴力解出

**時間複雜度：O(V^2 + E)**

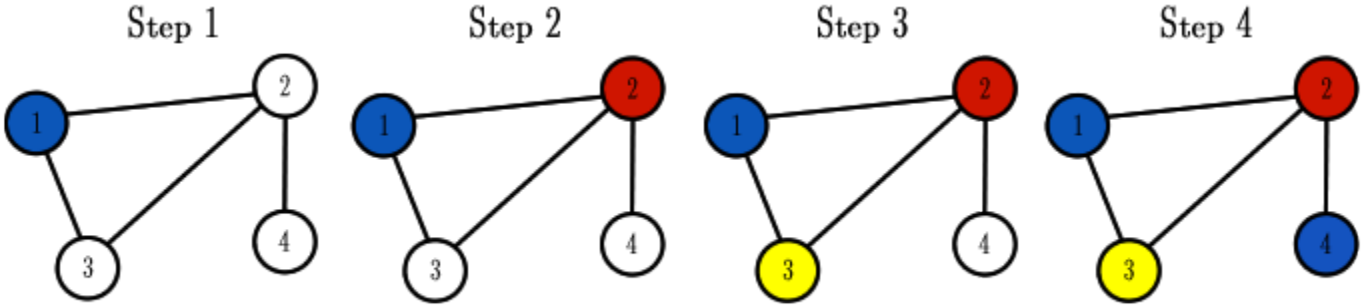



Figure 2: step-by-step process to color the vertices of a graph.

# Code - Greedy

```
void gdc()
{
    bool used[n+1];
    int uncol = n;

    memset(used, false, sizeof used);
    fill(color.begin(), color.end(), -1);

    while(uncol)
    {
        int target, ddeg = 0;
        for(int i = 0; i < degree.size(); i++)
            if(!used[i])
            {
                if(degree[i] > ddeg)
                {
                    ddeg = degree[i];
                    target = i;
                }
                /*else if(degree[i] == ddeg)
                {
                    int a = 0, b = 0;
                    for(int j = 0; j < graph[i].size(); ++j)
                        if(!used[graph[i][j]])
                            a++;
                    for(int j = 0; j < graph[target].size(); ++j)
                        if(!used[graph[target][j]])
                            b++;
                    if(a > b)
                        target = i;
                }*/
            }
```

```
        used[target] = true;
        int cr = 1;
        if(uncol == n)
        {
            color[target] = cr;
            max_color = max(max_color, cr);
            uncol--;
            continue;
        }
        while(1)
        {
            bool next = false;
            for(int i = 0; i < graph[target].size(); ++i)
                if(cr == color[graph[target][i]])
                    next = true;
            if(next)
                cr++;
            else
            {
                color[target] = cr;
                break;
            }
        }
        //if(target == 0)
        //  printf("%d %d\n",color[0], color[36]);
        max_color = max(max_color, cr);
        uncol--;
    }
```

# Code - Greedy

```
void gdc()
{
    bool used[n+1];
    int uncol = n;

    memset(used, false, sizeof used);
    fill(color.begin(), color.end(), -1);

    while(uncol)
    {
        int target, ddeg = 0;
        for(int i = 0; i < degree.size(); i++)
            if(!used[i])
            {
                if(degree[i] > ddeg)
                {
                    ddeg = degree[i];
                    target = i;
                }
                /*else if(degree[i] == ddeg)
                {
                    int a = 0, b = 0;
                    for(int j = 0; j < graph[i].size(); ++j)
                        if(!used[graph[i][j]])
                            a++;
                    for(int j = 0; j < graph[target].size(); ++j)
                        if(!used[graph[target][j]])
                            b++;
                    if(a > b)
                        target = i;
                }*/
            }
```

<-- used紀錄是否填過，uncol紀錄還有幾個vertex沒填

<-- color會存放該點的顏色，-1代表沒填過

<-- 先從Deg最大的點開始填顏色

# Code - Greedy

```
    used[target] = true;
    int cr = 1;
    if(uncol == n)
    {
        color[target] = cr;
        max_color = max(max_color, cr);
        uncol--;
        continue;
    }
    while(1)
    {
        bool next = false;
        for(int i = 0; i < graph[target].size(); ++i)
            if(cr == color[graph[target][i]])
                next = true;
        if(next)
            cr++;
        else
        {
            color[target] = cr;
            break;
        }
    }
    //if(target == 0)
    //  printf("%d %d\n",color[0], color[36]);
    max_color = max(max_color, cr);
    uncol--;
}
```

<-- cr為要填的顏色

<-- 如果是第一個被填的點不用判斷相鄰是否填過

<-- 之後被填的點要判斷相鄰點是否填過該顏色
如果有，cr往上加；反之，填入

*另外如果有好幾個Vertices的deg相同，
如何決定誰先填會影響結果

code是從數字最小開始填

<-- 紀錄Chromatic Number

# Code - Input

```
while(scanf("%d",&u) != EOF)
{
    scanf("%d",&v);

    if(!u || !v)
        with_zero = true;

    edge[u][v] = edge[v][u] = 1;

    n = u > n ? u : n;
    n = v > n ? v : n;
    e++;
}
if(with_zero)
    n++;
```

<-- 跟上一題使用同個code
將edge用2d bool array紀錄
並且判斷節點是否有0

# Code - Input

```
graph.resize(n+1);
degree.resize(n+1);
color.resize(n+1);

fill(degree.begin(), degree.end(), 0);

for(size_t i = 0; i <= n-1 ; ++i)
    for(size_t j = i+1; j <= n; ++j)
        if(edge[i][j])
        {
            graph[i].push_back(j);
            graph[j].push_back(i);
            degree[i]++, degree[j]++;
        }
```

<-- 將graph、degree等vector調整至適當大小

Graph為2d vec，負責記錄和誰連接
Degree為1d vec，負責記錄該點的deg大小

<-- 如果edge[i][j]為true，代表兩點有連接
所以graph[i]要加入j，graph[j]要加入I
同時，i, j 的degree都要增加1

# Code - Output

```
void print()
{
    for(int i = 0; i < color.size(); ++i)
        if(color[i] > 0)
            printf("%2d %2d\n", i, color[i]);
}
```

<- 先前有將各點顏色存入陣列
所以直接定義一個function輸出

```
printf("%d\n", max_color);
```

<- Chromatic Number在greedy過程紀錄
跑完gdc()後，直接輸出

# NP-Complete Problem

有和幾位朋友一起測試及討論，我們發現在填色時，遇到**degree相同的點時，決定誰先填(Order)**會影響答案結果

以助教提供的input 2測資為例
**Case I: 由小到大**
Chromatic Number 為 27

**Case II: 由大到小**
Chromatic Number 為 28

**Case III: 由相鄰最多未填格點的vertex開始填**
Chromatic Number 為 27

另外，**同個方法對應不同的graph並不一定能找到最佳解**
在input 3測資中
使用Case I的解為29，而使用Case III的解為32
但在input 2測資中
Case I和Case III兩者卻能找到同樣的答案27

```
/*else if(degree[i] == ddeg)
{
    int a = 0, b = 0;
    for(int j = 0; j < graph[i].size(); ++j)
        if(!used[graph[i][j]])
            a++;
    for(int j = 0; j < graph[target].size(); ++j)
        if(!used[graph[target][j]])
            b++;
    if(a > b)
        target = i;
}*/
```