



GraphSharp

LACHAUD Samuel / PAZOLA Loïs – Info S6

SOMMAIRE



I) Introduction

- Qu'est-ce que GraphSharp
- Présentation de l'application



II) Fonctionnement de l'application

- Affichage du graphe
- Calcul de dégénérescence
- Calcul du nombre chromatique



III) Fonctionnalités supplémentaires

- Comparatif dégénérescence/nombre chromatique
- Génération d'un fichier PDF
- Algorithme de Matula & Beck

I) Introduction

Qu'est-ce que GraphSharp

Le projet S&R GraphSharp est un projet de graphe codé en C# portant sur le calcul de la dégénérescence de grands graphes réels. Ainsi à partir de deux algorithmes différents, nous allons pouvoir calculer ce nombre de dégénérescence et le comparé avec le nombre chromatique d'une multitude de graphes réels du projet KONNECT (<http://konect.cc/>). Nous proposerons également un export de la dégénérescence de chaque sommet des graphes en PDF sous formes de cercles imbriqués pour une visualisation rapide et démonstrative.

Présentation de l'application

```
GraphSharp x + v
WELCOME TO OUR GRAPH PROJECT, CHOOSE AN OPTION IN THE LIST :
0. Quit
1. Display the graph
2. Calculate degeneration number
3. Calculate chromatic number
4. Generate a PDF with vertex degeneration
5. Compare degeneration and chromatic numbers for a lots of graphs
6. Calculate degeneration number with the Matula & Beck algorithm
```

GraphSharp est une application console proposant par le biais du Windows Terminal, une interface basique demandant à l'utilisateur un choix parmi les différentes fonctionnalités.

Pour les actions autres que la comparaison des nombres chromatiques et de la dégénérescence, plusieurs graphes différents nous sont proposés avec des tailles différentes

```
GraphSharp x + v
CHOOSE A FILE OR 0 TO GO BACK :
1. Exemple 680
2. ExempleLooping 940
3. PoliticalBooks 2,5Ko
4. Crime 10,9Ko
5. ERoadNetworks 11,1Ko
6. JazzMusicians 18,1Ko
7. Facebook 25,6Ko
8. SisterCities 193,8Ko
9. Youtube 3Mo
10. Flickr 92,4Mo
11. UK-Domains 4,2Go [6 Go RAM needed + 504Mo Download]
```

II) Fonctionnement de l'application

Affichage du graphe

Dans un premier temps, nous devons lire le fichier texte, ainsi, on détecte chaque ligne et on récupère les deux valeurs qui sont séparées par une tabulation (\t). Une fois récupéré, nous allons ranger ces valeurs de la manière suivante :

On crée un tableau composé de deux colonnes : la première étant le numéro du sommet et la deuxième une autre liste composée de tout les sommets qui ont une arrête en commun avec le sommet de la première colonne

```
static void ReadFile(string file)
{
    graph = new Dictionary<int, List<int>>();
    lastPercentage = -1; // Some files are immediately 100% loaded
    using (StreamReader streamReader = new StreamReader(@"Resources\" + file))
    {
        long fileLength = streamReader.BaseStream.Length;
        while (!streamReader.EndOfStream)
        {
            ShowProgression(streamReader.BaseStream.Position, fileLength);
            string readLine = streamReader.ReadLine();
            if (!readLine.StartsWith("%") && readLine != "") // read lines except lines who begin with "%"
            {
                string[] splittedLine = readLine.Split('\t'); // get the two ints of the line

                int curNode = int.Parse(splittedLine[0]);
                int nextNode = int.Parse(splittedLine[1]);

                Insert(curNode, nextNode); // insert ints into
                Insert(nextNode, curNode); // insert ints into
            }
        }
    }
}
```

Une fois le fichier chargé, l'affichage du graphe consiste juste à afficher le contenu du tableau nouvellement créé.

Calcul de dégénérescence

Une fois que notre graphe est chargé dans notre tableau, nous pouvons calculer la dégénérescence de celui-ci.

```
static void VertexDegenerationFilling()
{
    vertexDegenerationTable = new Dictionary<int, int>();
    int numberRemovedKeys = 0;
    // copy graph
    int n = graph.Keys.Max() + 1;
    List<int>[] localGraph = new List<int>[n];
    for (int i = 0; i < n; i++)
    {
        if (graph.Keys.Contains(i))
        {
            localGraph[i] = new List<int>(graph[i]);
        }
        else
        {
            localGraph[i] = null;
            numberRemovedKeys++;
        }
    }
    int k = 0;
    while (numberRemovedKeys < n)
    {
        ShowProgression(numberRemovedKeys, n);
        // filling a list with all key we have to delete
        List<int> removeKeys = new List<int>();
        for (int key = 0; key < n; key++) // for each keys
        {
            if (localGraph[key]?.Count <= k) // get all nextNode of the key
            {
                removeKeys.Add(key);
            }
        }
        // Delete keys and delete values in nextnodes values
        foreach (int key in removeKeys) // For each keys to remove
        {
            foreach (int neighborOfKey in localGraph[key]) // For each neighbors of the keys to remove
            {
                if (neighborOfKey != key) // If the node is self-linking it will be removed when setting its neighbors to null
                {
                    localGraph[neighborOfKey].Remove(key);
                }
            }
            localGraph[key] = null;
            vertexDegenerationTable.Add(key, k);

            numberRemovedKeys++;
            ShowProgression(numberRemovedKeys, n);
        }
        if (removeKeys.Count == 0)
        {
            k++;
        }
    }
    Console.WriteLine("\nThe degeneration number is : " + k);
    DegenerationNumber = k;
}
```

Nous stockons le graphe précédemment chargé dans une copie afin de pouvoir localement supprimer les sommets déjà traités par la k-dégénérescence. Cela nous permet de ne pas altérer le graphe chargé et ainsi ne pas avoir à le recharger en mémoire durant une autre opération. En supprimant les nœuds au fur et à mesure, on obtient finalement la dégénérescence du graphe, le k le plus grand.

Sur l'exemple du cours, nous avons donc la dégénérescence $k = 3$ qui est calculée :

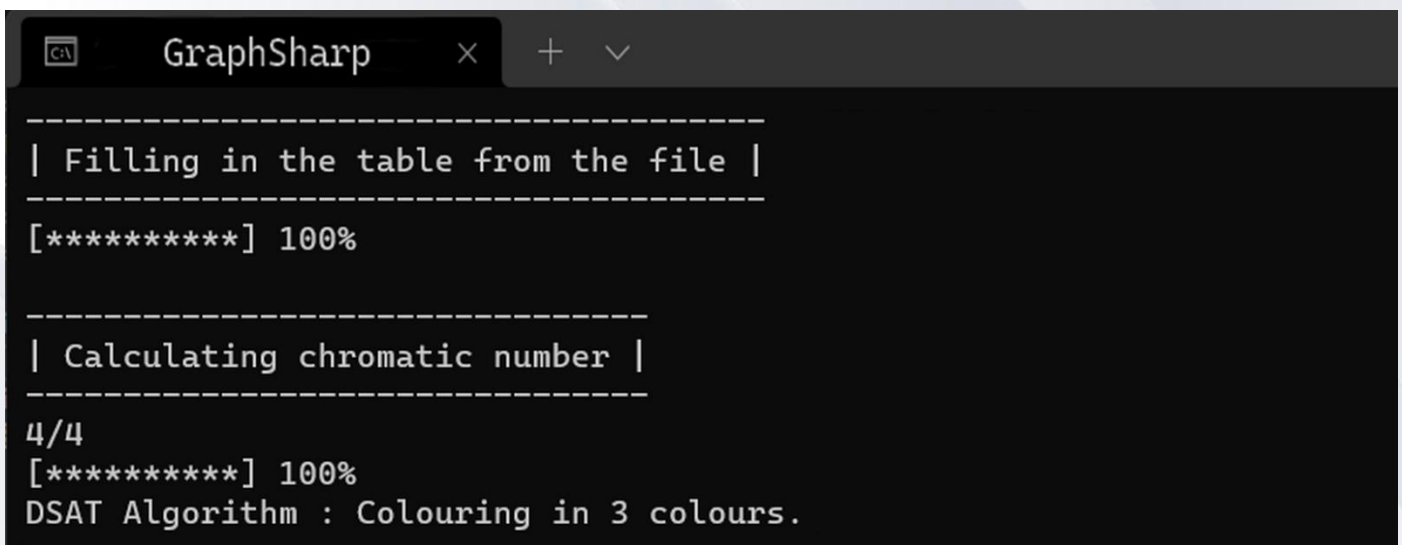
```
GraphSharp x + v
-----
| Filling in the table from the file |
-----
[*****] 100%
-----
| Calculating degeneration number and filling the table |
-----
[*****] 100%
The degeneration number is : 3
```

Calcul du nombre chromatique

Une fois que notre graphe est chargé dans notre tableau, nous pouvons calculer le nombre chromatique de celui-ci. Nous utilisons alors l'algorithme de DSATUR. Celui-ci est une transcription du programme vu en cours développé en C++, mais cette fois il est développé en C#.

Pour cela, nous devons changer notre structure comportant le graphe chargé, en effet pour le bon fonctionnement de l'algorithme DSATUR du cours, nous avons besoin d'une structure qui pour chaque sommet du graphe est associé un tableau où le numéro de ligne du tableau possède la valeur 1 si ce numéro de ligne (le numéro du sommet du graphe) est relié à celui-ci, et 0 si ce n'est pas le cas.

Pour le graphe de l'exemple nous obtenons donc une coloration en 3 couleurs. Pour ce qui est de la comparaison entre les différents graphes et le nombre de dégénérescence, nous en reparlerons dans la partie suivante.



```
-----  
| Filling in the table from the file |  
-----  
[*****] 100%  
  
-----  
| Calculating chromatic number |  
-----  
4/4  
[*****] 100%  
DSAT Algorithm : Colouring in 3 colours.
```


III) Fonctionnalités supplémentaires

Pour aller plus loin dans le projet, 3 améliorations ont été mises en place. Nous allons tout d'abord voir la comparaison entre la dégénérescence et le nombre chromatique sur un ensemble de graphe (**point 3.b du sujet**), puis la génération d'un fichier PDF (**point 3.c du sujet**) et enfin un meilleur algorithme pour le calcul de la dégénérescence qu'est l'algorithme de Matula & Beck (**point 3.a du sujet**).

Dégénérescence/nombre chromatique

Après avoir calculé le nombre chromatique et la dégénérescence d'un graphe, nous pouvons donc maintenant la comparer sur l'ensemble des graphes présents dans le programme. En saisissant l'option 5 sur le programme, ces deux valeurs vont donc être calculées sur l'ensemble des graphes (sauf Flickr car beaucoup trop lourd). Un récapitulatif sera alors montré à l'utilisateur :

```
GraphSharp x + v
file : Crime
degenerationNumber = 3
chromaticNumber = 4

file : Exemple
degenerationNumber = 3
chromaticNumber = 3

file : ExempleLooping
degenerationNumber = 2
chromaticNumber = 3

file : Facebook
degenerationNumber = 3
chromaticNumber = 4

file : Flickr
degenerationNumber = 187
chromaticNumber = 50

file : JazzMusicians
degenerationNumber = 29
chromaticNumber = 30

file : PoliticalBooks
degenerationNumber = 6
chromaticNumber = 6

file : SisterCities
degenerationNumber = 26
chromaticNumber = 27

file : Youtube
degenerationNumber = 22
chromaticNumber = 10
```

A partir de ce rendu, nous pouvons donc déduire que l'affirmation « on peut colorier un graphe k dégénéré en au plus $k + 1$ couleurs » est vrai dans notre cas d'observation. En effet la valeur **chromaticNumber** est toujours inférieure ou égale à la valeur **degenerationNumber** + 1.

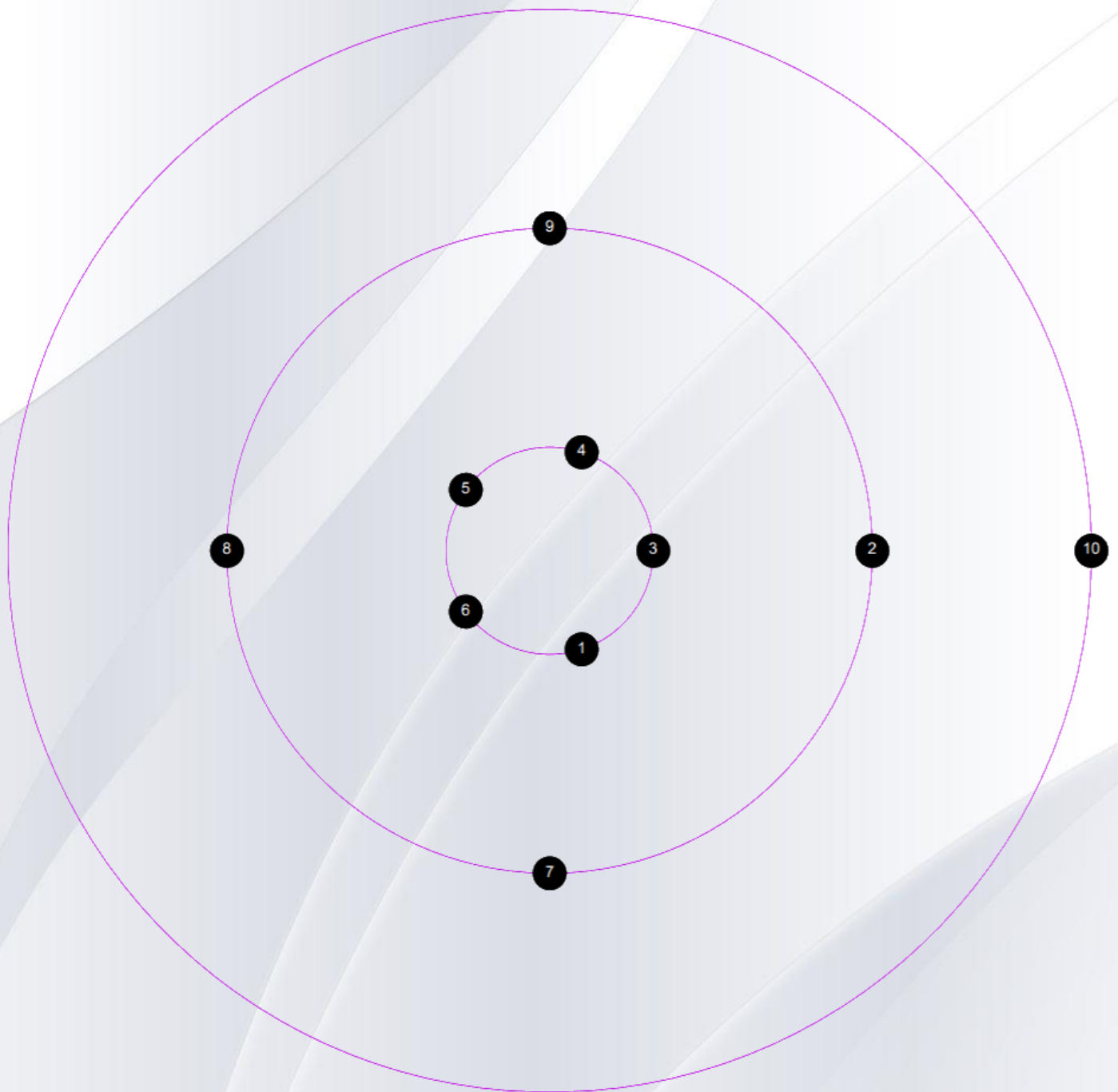
On peut également supposer aux vues de ce rendu que l'écart entre la dégénérescence et le nombre chromatique est plutôt faible, mais que quelques exceptions sont tout de même présentes comme pour le fichier

Génération d'un fichier PDF

La dégénérescence de notre graphe est calculable, mais difficilement visualisable, en effet, on calcule la k-dégénérescence maximale, mais également la dégénérescence de chaque sommet du graphe. Nous pouvons donc visualiser ce graphe avec des cercles imbriqués, un cercle pour chaque k-dégénérescence. Le tout est mis dans un fichier PDF et chaque sommet est placé sur le cercle lui correspondant.

```
static void CreatePDF(string filename){
    string pdfFile = @"Resources\" + filename + ".pdf";
    Document document = new Document(PageSize.A0, 10, 10, 10, 10);
    FileStream fs = new FileStream(pdfFile, FileMode.Create, FileAccess.Write);
    PdfWriter writer = PdfWriter.GetInstance(document, fs);
    document.Open();
    // the pdf content
    PdfContentByte cb = writer.DirectContent;
    // create table k --> number of k-degeneration
    Dictionary<int, int> kNumbers = new Dictionary<int, int>();
    int vertexDegenerationTableKeysMax = vertexDegenerationTable.Keys.Max();
    foreach (int key in vertexDegenerationTable.Keys.OrderBy(key => key)){
        if (!kNumbers.ContainsKey(vertexDegenerationTable[key])){
            kNumbers.Add(vertexDegenerationTable[key], 1);
        }
        else{
            kNumbers[vertexDegenerationTable[key]]++;
        }
        ShowProgression(key, vertexDegenerationTableKeysMax);
    }
    int maxCircleSize = (int)(document.GetRight(-10) / 2) * 80 / 100;
    int minCircleSize = 150;
    int kNumbersKeysMax = kNumbers.Keys.Max();
    foreach (int key in kNumbers.Keys.OrderBy(key => key)){ // circles for loop
        // circles creation
        int circleSize = maxCircleSize / DegenerationNumber * (DegenerationNumber - key) + minCircleSize;
        cb.SetColorStroke(new BaseColor(183, 3, 223));
        cb.Circle(document.GetRight(-10) / 2, document.GetTop(-10) / 2, circleSize);
        cb.Stroke();
        double angle = Math.PI * (360.0 / kNumbers[key]) / 180.0; // angle in rad
        int i = 0;
        foreach (int v in vertexDegenerationTable.Keys){
            if (vertexDegenerationTable[v] == key){
                // calculate the position of the little circles with initial position and angle
                double posX = (double)(document.GetRight(-10) / 2 + circleSize * Math.Cos(angle * i));
                double posY = (double)(document.GetTop(-10) / 2 + circleSize * Math.Sin(angle * i));
                double elemtSize = 75 / DegenerationNumber;
                if (elemtSize < 5){
                    elemtSize = 5;
                }
                cb.Circle(posX, posY, elemtSize);
                cb.SetColorFill(new BaseColor(0, 0, 0));
                cb.Fill();
                cb.SetColorStroke(new BaseColor(0, 0, 0));
                cb.Stroke();
                cb.BeginText();
                cb.SetColorFill(new BaseColor(255, 255, 255));
                cb.SetFontAndSize(BaseFont.CreateFont(BaseFont.HELVETICA, BaseFont.CP1252, BaseFont.NOT_EMBEDDED), (float)elemtSize-2);
                cb.ShowTextAligned(Element.ALIGN_CENTER, v.ToString(), (float)posX, (float)(posY - elemtSize / 5), 0);
                cb.EndText();
                i++;
            }
        }
        ShowProgression(key, kNumbersKeysMax);
    }
    document.Close();
    fs.Close();
    writer.Close();
    //Open the generated pdf
    new Process{
        StartInfo = new ProcessStartInfo(pdfFile){
            UseShellExecute = true
        }
    }.Start();
}
```


Le programme ouvre donc le fichier PDF généré, voici celui correspondant à l'exemple. Ainsi le cercle extérieur représente les sommets de dégénérescence 1 puis à l'intérieur le 2 et enfin le 3 :



Algorithme de Matula & Beck

La dernière amélioration de notre programme est l'ajout de l'algorithme de Matula & Beck. En effet il permet également de calculer la dégénérescence d'un graphe, mais de manière beaucoup plus rapide.

La mise en place de l'algorithme de Matula & Beck s'est fait à l'aide de la thèse suivante : https://schulzchristian.github.io/thesis/thesis_huebner.pdf.

```

static void VertexDegenerationFillingMatulaBeck()
{
    //Initialize an output list L
    vertexDegenerationTableMatulaBeck = new Dictionary<int, List<int>>();
    //Initialize an array D such that D[i] contains a list of the vertices v that are not already in L for which dv = i
    int[] d = new int[graph.Keys.Max() + 1];
    int BiggestI = graph.Max(x => x.Value.Count);
    int DInitialNumberOfElements = 0;
    int dLength = d.Length;
    List<int>[] D = new List<int>[BiggestI + 1];
    for (int j = 0; j < BiggestI + 1; j++){ // Initialise D
        D[j] = new List<int>();
    }
    for (int j = 0; j < dLength; j++){
        d[j] = -1;
        if (graph.ContainsKey(j)){
            d[j] = graph[j].Count;
            DInitialNumberOfElements++;
            D[d[j]].Add(j);
        }
    }
    //Initialize k to 0
    int k = 0;
    int i = 0;
    bool DContainsValues = true;
    int DNumberOfRemovedElements = 0;
    //Repeat n times
    while (DContainsValues){
        DContainsValues = false;
        //Search for the lowest non-empty bucket
        if (D[i].Count == 0){
            for (int j = i; j < D.Length; j++){
                if (D[j].Count > 0){
                    i = j;
                    DContainsValues = true;
                    break;
                }
            }
        }
        else{
            DContainsValues = true;
        }
        k = Math.Max(k, i);
        if (DContainsValues){
            //Remove the first vertex in the lowest non-empty bucket
            int v = D[i][0];
            D[i].Remove(v);
            DNumberOfRemovedElements++;
            if (vertexDegenerationTableMatulaBeck.ContainsKey(k)){
                vertexDegenerationTableMatulaBeck[k].Add(v);
            }
            else{
                vertexDegenerationTableMatulaBeck.Add(k, new List<int> { v });
            }
            d[v] = -1;
            //Update the neighbors of the removed vertex, after checking if they were not removed
            foreach (int u in graph[v]){
                if (d[u] != -1){
                    if (i == d[u]){
                        i--;
                    }
                    D[d[u]].Remove(u);
                    d[u] -= 1;
                    D[d[u]].Add(u);
                }
            }
        }
    }
    Console.WriteLine("\nThe degeneration number is : " + k);
}

```

Pour montrer cela, nous allons comparer la vitesse de calcul de la dégénérescence du graphe **Flickr**.

Ce graphe est composé de plus de **8 500 000 arrêtes** et est stocké dans un fichier pesant près de **100Mo**. Le calcul est pourtant très rapide mais la différence est saisissante

```

GraphSharp
The degeneration number is : 187
Execution Time: 9s 352ms Classic

The degeneration number is : 187
Execution Time: 2s 144ms M&B

9352ms / 2144ms = 4.3x plus rapide

```

```

GraphSharp
The degeneration number is : 943
Execution Time: 30min 12s 598ms Classic

The degeneration number is : 943
Execution Time: 3h 10min 40s 500ms M&B

190min / 30min = 6x plus lent

```

En effectuant des essais sur des plus grands graphes, nous avons trouvé quelque chose d'étrange. Sur le graphe **UK-Domains** composé de **261 787 000 arrêtes** et est stocké dans un fichier de **4Go**. **Matula & Beck semble plus lent pour l'exécution de très grands fichiers !**