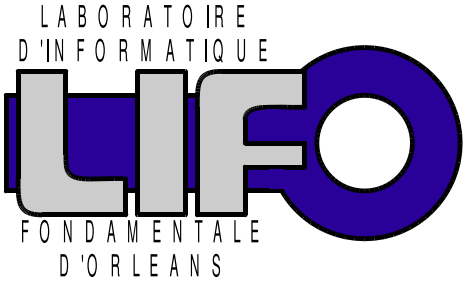


4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

ANR AGAPE - Implémentation d'algorithmes exacts pour le problème du stable maximum

Vincent Levorato
LIFO, Université d'Orléans

Rapport n^o **RR-2011-15**

Table des matières

1 Contexte et travail effectué

Les travaux que j'ai effectués jusqu'à présent concernent l'implémentation d'algorithmes exacts connus sur le problème de la recherche d'un *ensemble stable maximum* dans les graphes, ainsi que l'étude et l'utilisation de la librairie MASCOPT pour l'implémentation des algorithmes. Ce rapport détaille les algorithmes, explique l'implémentation de ceux-ci, et quels choix ont été retenus pour la représentation des structures de données. Il y sera également abordé des comparaisons d'implémentation avec d'autres librairies et/ou langage de programmation (C++ notamment), ainsi que des tests de performance. Les algorithmes sont détaillés pour faire apparaître certaines opérations non visibles dans les algorithmes de base (facteur polynomial) pour ainsi avoir une vision claire de la version implémentée. Une étude des résultats est également proposée en dernière partie.

2 Algorithmes exacts pour le problème du Stable Maximum

2.1 Algorithme basé sur le travail de Moon and Moser (1965)

Cet algorithme (voir Alg. 1) énumère tous les ensembles stables maximaux d'un graphe et permet donc d'en déduire un plus grand (maximum). Sa com-

plexité en temps d'exécution est bornée en $O^*(1,4423^n)$. Un appel de **MaxIS-MM**(G,$\emptyset$) nous renvoie un stable maximum. Le facteur polynomial le plus élevé que l'on peut trouver dans l'algorithme est celui de la copie de graphe qui se fait en $O(n+m)$.

Algorithme 1 Maximum Independent Set - [MM65]

```
Method : MaxIS-MM(Graph G(V,E) , Set S)
Begin
  if |V| > 0 then
    Vertex v ← MinDegVertex(G)
    Set Smax ← ∅
    for all Vertex u ∈ N[v] do
      Graph G'(V', E') ← copy(G(V, E))
      V' ← V' − N[u]
      Set Stemp ← MaxIS-MM(G', S ∪ {u})
      if |Stemp| > |Smax| then
        Smax ← Stemp
      end if
    end for
    return Smax
  else
    return S
  end if
End
```

2.2 Algorithme Degré Maximum

Cet algorithme (voir Alg. 2) utilise une règle de branchement qui, pendant les appels récursifs, vérifie s'il ne reste plus que des sommets de degré 1 ou 2 (cycle ou chemin) auquel cas on sait trouver le stable maximal en temps polynomial (approche gloutonne par exemple). La méthode **MaximalIndependentSet-Greedy**(G) renvoie un ensemble stable maximal de G. Comme cette méthode n'est appliquée qu'à des cycles ou des chemins, il suffit de commencer par le sommet de plus petit degré et de prendre un sommet sur 2 comme appartenant au stable. Le facteur polynomial le plus élevé que l'on peut trouver dans l'algorithme est également celui de la copie de graphe ($O(n+m)$). La légère différence avec l'algorithme de [MM65] est que l'on réalise 2 fois cette copie pour un appel récursif.

2.3 Algorithme Fomin, Grandoni and Kratsch (2009)

Cet algorithme utilise la méthode *Mesurer pour conquérir* [FGK09, FK10] pour établir le temps d'exécution au pire cas et permet de trouver un ensemble stable maximum dans un graphe avec une complexité en temps d'exécution en $O^*(1,2201^n)$. Celui-ci fait appel à certaines opérations sur les graphes comme la *domination*, le *pliage* de noeud (folding), ou les noeuds *mirroirs* (mirroring). Ces concepts sont adaptés dans l'article de Fomin et al. à partir d'autres méthodes dont les références se trouvent dans [FGK09].
Note : l'algorithme original renvoie seulement la taille d'un stable maximum, une version renvoyant le stable sera proposée par la suite.

Algorithme 2 Maximum Independent Set - MaxDegree

Method : **MaxIS-MaxDeg**(Graph G(V,E))
Begin
  Integer degmax ← $MaxDeg(G)$
  Set S ← $\emptyset$
  if $degmax \geq 3$ **then**
    Vertex v ← $MaxDegVertex(G)$
    Graph $G'(V', E')$ ← $copy(G(V, E))$
    V' ← $V' - N[v]$
    Set S ← **MaxIS-MaxDeg**(G') ∪{v}
    Graph G"(V", E") ← $copy(G(V, E))$
    Set S' ← **MaxIS-MaxDeg**(G" − {v})
    if $|S'| > |S|$ **then**
      $S \leftarrow S'$
    end if
  else
    $S \leftarrow$ **MaximalIndependentSetGreedy**(G)
  end if
  return S
End

Algorithme 3 Maximum Independent Set - [FGK09] (original version)

Method : **mis**(Graph G(V,E))
Begin
  if $|V| \leq 1$ **then**
    return $|V|$
  end if
  if ∃ *component* $C \subset G$ **then**
    return mis(C) + mis($G - C$)
  end if
  if ∃ vertices v and w / $N[w] \subseteq N[v]$ **then**
    return mis($G - \{v\}$)
  end if
  if ∃ vertex v with $deg(v) = 2$ **then**
    return 1 + mis($\tilde{G}$(v)) {folding G on v}
  end if
  select a vertex v of maximum degree, which minimizes $|E(N(v))|$
  return max{ mis($G - \{v\} - M(v)$) , 1 + mis($G - N[v]$) } {$M(v)$ corresponds to mirrors of v}
End

L'algorithme de Fomin et al., bien que simple dans sa vision globale (voir Alg. 3), se découpe en sous-problèmes parfois « coûteux » (en terme polynomial) avec dans l'ordre :

1. Recherche de composantes connexes dans un graphe
2. Recherche de noeuds dominés
3. Pliage de noeuds (et dépliage le cas échéant)
4. Recherche de noeuds mirroirs

2.3.1 Recherche de composantes connexes

Cette opération correspond à un parcours en profondeur (DFS). Dans l'algorithme, il suffit de renvoyer la première composante connexe trouvée. Je ne détaille ici pas les opérations d'un DFS (trivial). Il faut juste noter que le sommet source est pris au hasard. La complexité d'un DFS est en $O(n + m)$.

2.3.2 Recherche de noeuds dominés

La recherche de noeuds dominés est un peu plus coûteuse en temps que l'opération précédente. Néanmoins, elle ne sera appelée qu'une seule fois par appel récursif et n'a besoin que de renvoyer un seul sommet dominé (ou on peut renvoyer l'ensemble des sommets dominés pour nettoyer le graphe de tous ses sommets dominés à un moment précis de l'algorithme, test effectué qui n'a pas changé les temps d'exécution). Afin de trouver des sommets v et w tels que $N[w] \subseteq N[v]$, on est obligé de visiter tous les voisinages du graphe dans le pire des cas. Ensuite, le problème de trouver un noeud dominé revient à chercher un recouvrement total d'un ensemble (de sommets) par un autre. Une stratégie simple consiste à trier par taille les ensembles de voisinages de G, de commencer par le plus grand, vérifier s'il ne recouvre pas un ensemble de voisinage (en commençant par le plus petit), et recommencer l'opération tant que ce n'est pas le cas (Alg. 4). On considère que la relation $N[x] \mapsto x$ est représentée par une table de hachage et que déterminer à quel sommet correspond un voisinage se fait en temps constant. Si l'on considère la complexité dans le pire des cas, la méthode contient un QuickSort qui se fait en $O(n^2)$ (il est néanmoins connu que le QuickSort se comporte plutôt bien en moyenne ($\Theta(n \cdot log(n))$)), et la recherche de recouvrement qui se fait en $O(n^3)$. Cependant, même s'il faudrait le démontrer, on est loin d'atteindre cette borne et on s'approche plutôt d'un $O(n^2/2)$ (constaté en pratique).

2.3.3 Pliage de noeuds (folding)

Avant de plier un noeud, il faut tout d'abord en rechercher un de degré 2 (en $O(3n) \simeq O(n)$). Plier un noeud v de G pour obtenir $\tilde{\mathrm{G}}$(v) se fait en 4 étapes :

1. ajouter un sommet u_{ij} pour chaque anti-arête $u_i u_j$ de $N(v)$
2. ajouter une arête entre u_{ij} et chaque sommet de $N(u_i) \cup N(u_j)$
3. ajouter une arête entre chaque paire de nouveaux sommets

Algorithme 4 Dominated Vertex

```
Method : getDominatedVertex(Graph G(V,E))
Begin
  Vertex x ← null
  Array NList ← empty
  for all v ∈ V do
    NList.add(N[v])
  end for
  QuickSortAsc(NList)
  vfound ← false
  Integer i ← |NList| − 1
  while i > 0 and ¬vfound do
    Integer j ← 0
    while j < i and ¬vfound do
      if NList[j] ⊆ NList[i] then
        x ← NList[i].v
        vfound ← true
      end if
      j ← j + 1
    end while
    i ← i − 1
  end while
  return x
End
```

4. supprimer les sommets de $N[v]$

La méthode proposée (Alg. 5) est générique (applicable à des sommets de degré supérieur à 2). Dans le cas de l'algorithme FGK09, on se limite aux noeuds de degré 2. La complexité du pliage du noeud est donc bornée par la copie du graphe ($O(n+m)$) puisque l'autre opération coûteuse ne dépassera pas $O(m)$ (quand on doit relié le nouveau noeud u_{ij} à tous les voisins de u_i et de u_j).

2.3.4 Noeuds mirroirs (mirroring)

Avant de rechercher les noeuds mirroirs d'un noeud v, il est nécessaire d'extraire le noeud avec un degré maximum et dont les voisins soient les moins connectés possible. Dans le pire cas, on obtiendra une complexité de l'ordre de $O(n.m)$. Ensuite, on aura besoin de copier deux fois le graphe : une fois pour rechercher les noeuds mirroirs de v et une fois pour supprimer $N[v]$ (l'algorithme branche des deux côtés).
Un mirroir $u \in N^2(v)$ de v est un mirroir de v si $N(v) - N(u)$ est une clique (vide ou non). Les mirroirs de v sont notés $M(v)$. Dans cette définition, il y a besoin de savoir si un ensemble de sommets est une clique. Pour cela, on utilise la méthode **isClique(.)** dont l'algorithme est donné Alg. 6. Soit K l'ensemble de noeuds à tester, la complexité de ce test est en $O(|K|^2)$ dans le pire des cas (qui est le fait de trouver une clique). Dans le cas où K n'est pas une clique, on sera tout de même en $O((|K|-2) \times |K|)$ (cas où un noeud est connecté à tous les noeuds sauf un). Le meilleur cas, outre le fait que la clique soit vide ou de taille égale à 1 est en $\Omega(1)$.

La méthode **getMirrors** renvoie un ensemble de mirroirs d'un noeud v donné en paramètre (Alg. 7). La clique C testée peut être de taille $n-1$ au

Algorithme 5 Folding Vertex

Method : **getFoldingGraph**(Graph G(V,E), Vertex v)
HashTable $hmFold$ is used to keep track of merged vertices : $x_{12} \rightarrow \{x_1, x_2\}$
Begin
  Graph $G'(V', E') \leftarrow copy(G(V, E))$
  Array $NList \leftarrow N[v]$
  Set $newV \leftarrow \emptyset$
  for Integer $i \leftarrow 0$ to $|NList|$ **do**
    for Integer $j \leftarrow i + 1$ to $|NList|$ **do**
      Vertex $u_i \leftarrow NList[i],\ u_j \leftarrow NList[j]$
      if ¬**isEdge**(u_i, u_j) **then**
        Vertex $u_{ij} \leftarrow G'.newNode()$
        hmFold.put$(u_{ij}, \{u_i, u_j\})$ {used only for unfolding operation}
        for all Vertex $x \in N[u_i] \cup N[u_j]$ **do**
          G'.addEdge(u_{ij}, x)
        end for
        $newV \leftarrow newV \cup \{u_{ij}\}$
      end if
    end for
  end for
  for all pair(x,y) of $newV$ **do**
    G'.addEdge(x, y)
  end for
  $G' \leftarrow G' - N[v]$
  return G'
End

Algorithme 6 Clique test

Method : **isClique**(Graph G(V,E), Set K)
Begin
  Bool $b \leftarrow true$
  if $|K| \leq 1$ **then**
    return true
  end if
  Vertex $x \leftarrow K[0]$
  Set $A \leftarrow N[x]$
  Integer $i \leftarrow 1$
  while ¬b and $i < |K|$ **do**
    Vertex $y \leftarrow K[i]$
    $A \leftarrow A \cap N[y]$
    if $K \neq A$ **then**
      $b \leftarrow false$
    end if
    $i \leftarrow i + 1$
  end while
  return b
End

maximum. La complexité de cette méthode devrait être autour de $O(c.n^2)$, avec c une constante, ce qui correspond en partie à la complexité pour tester si un ensemble est une clique ou non, car deux cas extrêmes sont possibles : soit $|N^2[v]|$ tend vers n et la complexité du test de clique chute, soit $|N^2[v]|$ tend vers 1 et donc la boucle sur $u \in N^2[v]$ sera petite (d'où la constante c) mais les ensembles testés par **isClique** seront plus grands et pourront augmenter le temps d'exécution.

Algorithme 7 Mirroring

```
Method : getMirrors(Graph G(V,E), Vertex v)
Begin
  Set Mv ← ∅
  for all u ∈ N²[v] do
    Set C ← N[v] − N[u]
    if isClique(G, C) then
      Mv ← Mv ∪ {u}
    end if
  end for
  return Mv
End
```

2.3.5 Algorithme FGK09 renvoyant l'ensemble stable

Les modifications par rapport à l'algorithme de base se font au niveau du *folding* et du *mirroring*.

Concernant le Folding : pendant un folding, on garde en mémoire les noeuds pliés dans un tableau associatif (noeud fusionné $\mapsto$ noeuds originaux) grâce à une table de hachage. Au moment où l'algorithme fait remonter le résultat du stable S, on vérifie si des noeuds fusionnés se trouvent dans S. Si c'est le cas, on déplie ces fameux noeuds dans S grâce au tableau, et on renvoie S. Sinon, cela veut dire qu'aucun voisin de v, le noeud plié, n'a été gardé dans le stable. On renvoie donc $S \cup \{v\}$.

Concernant le Mirroring : Dans l'algorithme de FGK09, après le retour des stables que l'on note S_1 sans v et $M(v)$, et S_2 sans $N[v]$, il faut ajouter v à S_2 car on a déjà enlevé ses voisins, et il ne sera pas compris dans S_1. Finalement, si on arrive à cette branche de l'algorithme, cela revient à faire comme dans Alg. 2 (on branche soit avec v, soit sans v).

3 Résultats

Dans cette partie sont présentés les résultats obtenus sur le problème de l'ensemble stable maximum en JAVA avec MASCOPT et JUNG, et en C++ avec LEMON, et deux implémentations « homemade » de structures de graphes. Les tests ont été réalisés sur des graphes aléatoires $G_{n,p}$ d'Erdös et Rényi [ER59]. La collection de graphes générés comprend des graphes $G_{n,p}$ avec $n \in$

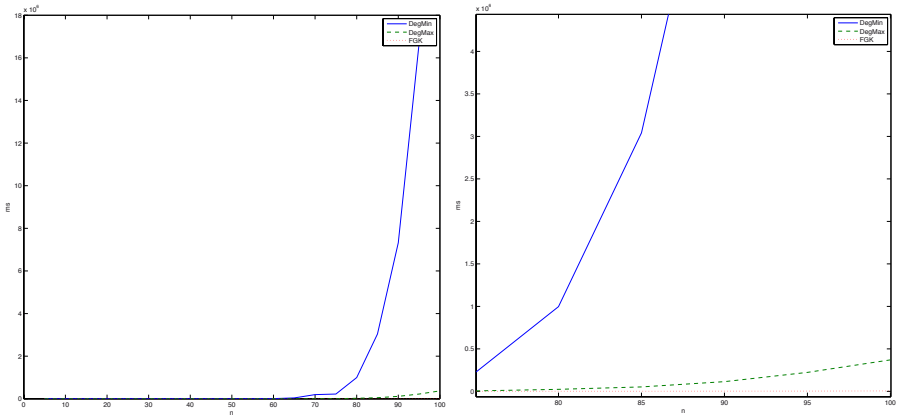
$\{5, 10, 15, ..., 95, 100\}$ et $p \in \{0.0, 0.1, 0.2, ..., 0.9, 1.0\}$. Une fois générés, les graphes utilisés pour les tests ont été les mêmes pour toutes les implémentations. Tout d'abord il sera présenté les résultats des comparaisons entre algorithmes puis entre les différentes implémentations. Pour finir, une étude des résultats obtenus est proposée.

3.1 Algorithmes

Les résultats en terme de temps d'exécution des trois algorithmes présentés plus haut ne sont présentés, par souci de lisibilité, que pour les graphes $G_{n,p}$ avec n variant de 5 à 100 et $p = 0.1$ (cas où les algorithmes de Moon et Moser, et DegMax ont le plus de difficulté, légèrement moins pour FGK qui semble avoir plus de problème à $p = 0.2$). Nous nommerons DegMin l'algorithme de Moon et Moser, et FGK l'algorithme de Fomin et al. (qui renvoie le stable et non pas sa taille uniquement). La figure 2 montre la différence de résultats (ceux-ci sont disponibles en annexe) avec n le nombre de sommets du graphe en abscisse et le temps en ms en ordonnée. Pour donner un ordre d'idée, pour $n = 95$ et $p = 0.1$:

- t(DegMin) = 4h 38min
- t(DegMax) = 3min 43s
- t(FGK) = 3,1s

Etant donné la grande différence de résultat en terme d'amplitude, la figure 2 montre les même données avec une échelle logarithmique sur l'axe temporel. Un test sur la gestion de la pile a été effectué en dé-recursivant l'algorithme DegMin et en utilisant la pile de manière explicite à l'aide de l'objet `Stack` : sans surprise, la version récursive où la pile est gérée par JAVA est plus rapide que la version non-récursive où la pile est gérée dans le programme.



(a) Vue globale (b) Zoom sur la partie n={75 :100}

FIGURE 1 – Temps d'exécution des algorithmes *stable max* (échelle linéaire)

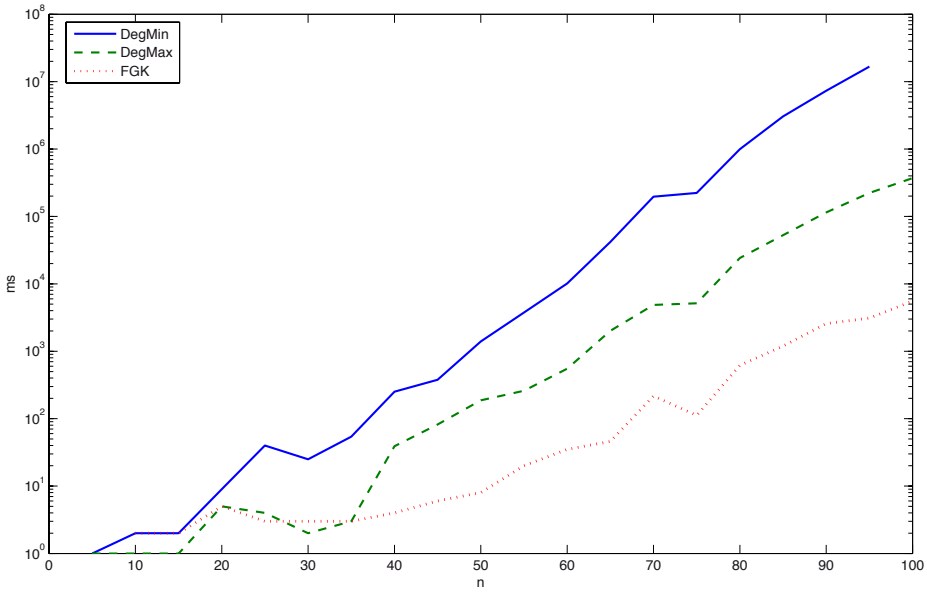


FIGURE 2 – Temps d'exécution des algorithmes *stable max* (échelle log)

3.2 Implémentations

Dans cette partie, on montre tout d'abord la différence de performances qu'il y a entre Mascopt (JAVA), JUNG (JAVA), Lemon (C++), et MyGraph (C++). JUNG [1] est la librairie de graphes la plus connue et utilisée dans le monde JAVA. En effet, elle fournit un aspect algorithmique de base (algorithmes les plus connus et génération de graphes) d'une part, et d'autre un aspect visualisation assez avancé. En C++, ayant tenté d'utiliser Boost et LEDA (Free Edition), celles-ci se sont révélées inadaptées à l'utilisation d'algorithmes avec appels récursifs et suppression de sommets en l'état (la version Professional Edition ou Research Edition fournit des algorithmes sur les graphes). En C++, Lemon [2] est la librairie qui s'en sort le mieux, et MyGraph est une implémentation que j'ai réalisée en choisissant une liste d'adjacence pour représenter le graphe, le tout basé sur des structures de tables de hachage implémentés par la librairie Google Sparse Hash [3] qui est une des meilleures implémentation de HashMap en C++ (voir `http://msinilo.pl/blog/?p=668` pour les benchs), l'autre version de mon implémentation étant basée sur des arbres binaires et listes doublement chaînées. En comparant les implémentations JAVA Mascopt et JUNG (voir Fig. 3 pour DegMin), on constate que JUNG est beaucoup plus rapide que Mascopt pour les algorithmes de recherche d'un stable maximum, JUNG étant entre 50

1. http ://jung.sourceforge.net/
2. http ://lemon.cs.elte.hu/trac/lemon
3. http ://goog-sparsehash.sourceforge.net/

et 100 fois plus rapide selon les graphes.

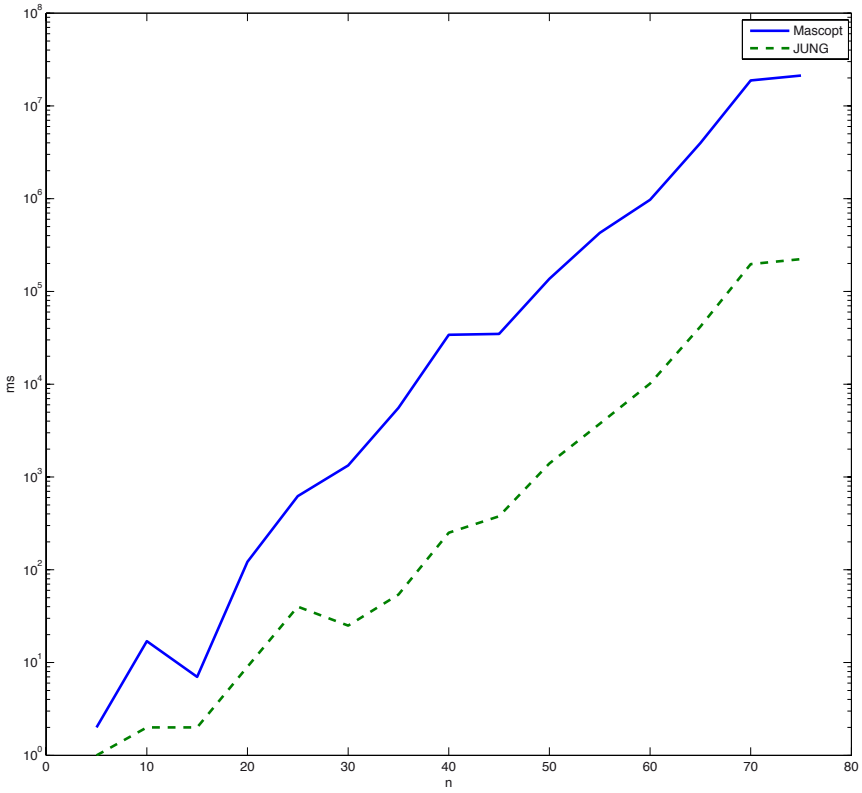


FIGURE 3 – Temps d'exécution de l'algo DegMin pour Mascopt et JUNG (échelle log)

Concernant le C++, Lemon est un peu plus rapide que mon implémentation (voir Fig. 4), expliqué par le fait que pour Lemon, je ne fais pas de copie de graphe, mais on enlève sommets et arêtes qu'on réinsère une fois le résultat de la récurrence obtenu (la version arbre/liste n'est pas présentée car elle présente peu de différence dans les résultats obtenus).

Un des résultats intéressant qui ressort est que Lemon n'est pas vraiment plus rapide que JUNG (voir Fig. 5). Cela peut s'expliquer par le fait que les algorithmes de recherche de stable max ne font principalement que des accès mémoire et que, dans ce cas, (où les structures données sont sensiblement les mêmes), les performances entre C++ et JAVA sont quasiment identiques. Le fait que JAVA dépasse légèrement C++ par moment peut être le fait que le Garbage Collector « gère mieux » la mémoire qu'une implémentation lambda faite en C++.

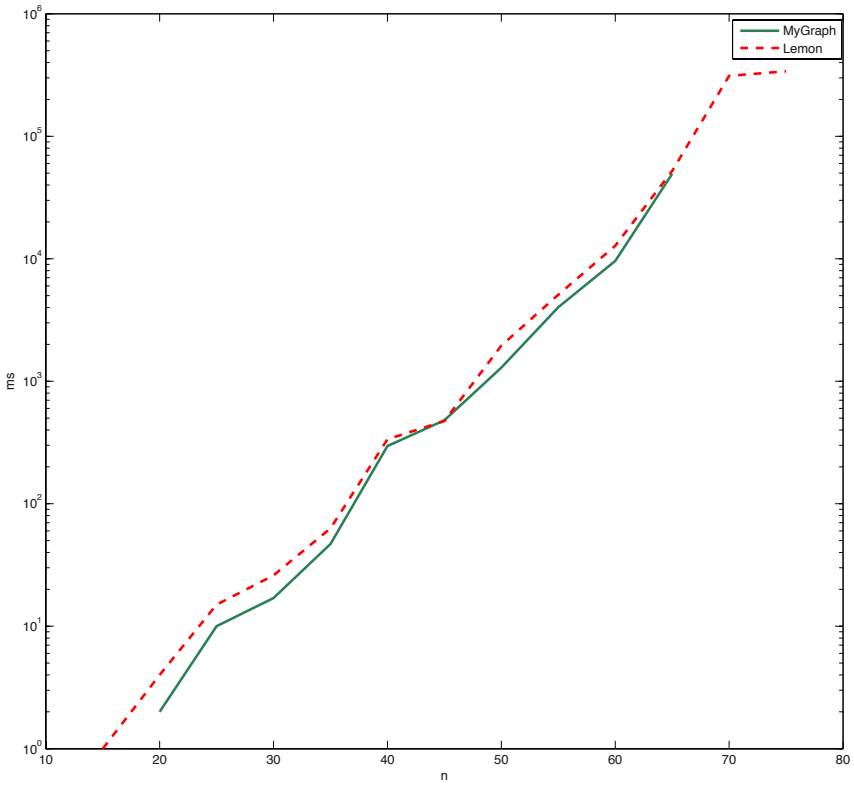


FIGURE 4 – Temps d'exécution de l'algo DegMin pour Lemon et MyGraph (échelle log)

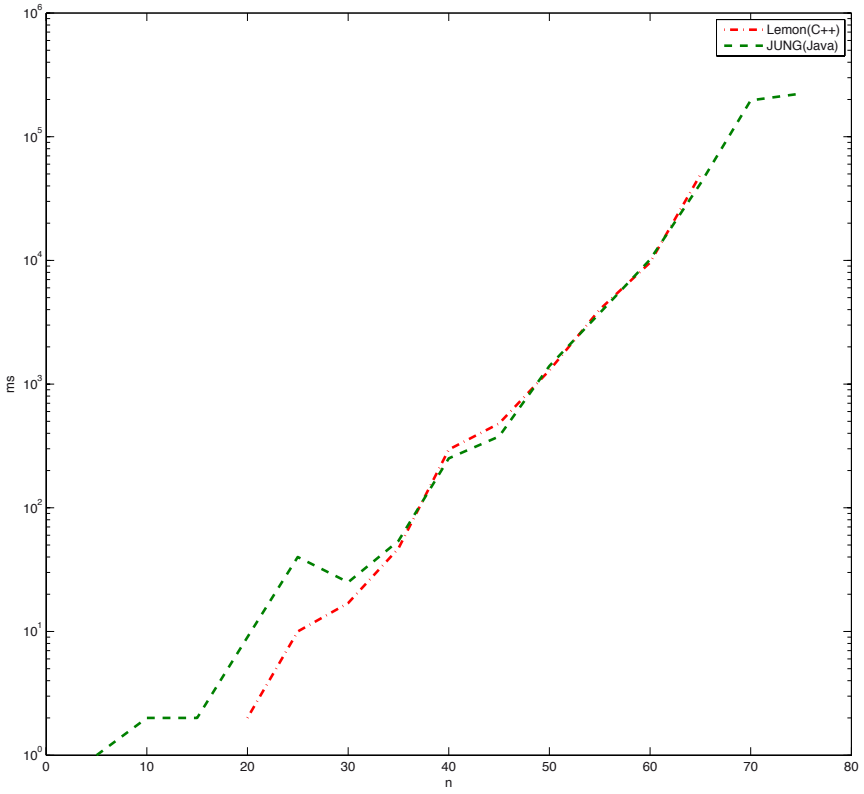


FIGURE 5 – Temps d'exécution de l'algo DegMin pour Lemon et JUNG (échelle log)

3.3 Etude des appels récursifs pour FGK09

Il peut être intéressant pour l'algorithme FGK d'observer comment celui-ci ≪ branche ≫ les sous-problèmes. Dans la figure 6, on a étudié les appels des méthodes de FGK sur quelques cas pour $n = 100$ et $p = \{0.1, 0.2, 0.5, 0.6, 0.9\}$. On distingue l'appel récursif de la méthode elle-même (FGK), la recherche de composantes connexes (CComponent), la recherche de noeud dominé (DomVertex), le pliage de noeud (Folding) et la recherche de noeuds mirroirs (Mirroring). De manière empirique, on note que l'algorithme est plus lent quand celui-ci branche par la recherche de mirroirs sans savoir si celle-ci est effective ou pas, et qu'une recherche infructueuse peut faire brancher l'algorithme vers le cas le pire pour un appel récursif.

3.4 Interprétation des résultats

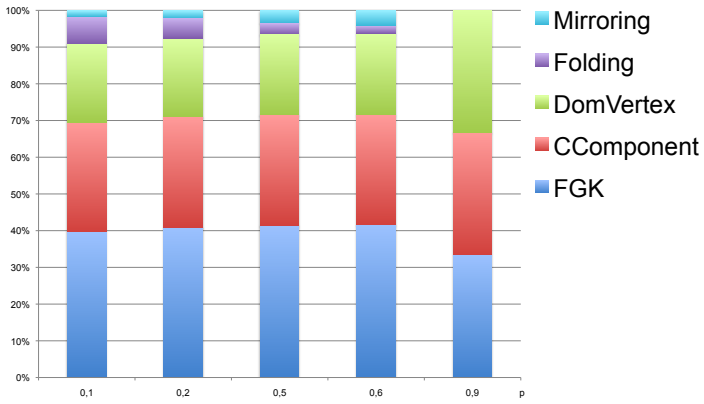
Il faut rester prudent quant aux résultats présentés dans ce rapport pour la simple et bonne raison que les graphes aléatoires $G_{n,p}$ d'Erdös et Rényi sont des graphes aux propriétés particulières [New02], et qu'il faudra réfléchir à d'autres algorithmes de génération de graphes par la suite. En effet, le degré moyen z d'un $G_{n,p}$ s'exprime en fonction de n et p : $z \simeq np$. Il y a également une phase de transition avec l'apparition d'une composante géante pour $z = 1$. Néanmoins, on peut établir quelques remarques sur le comportement des algorithmes. Les résultats de la figure 7 donne une idée sur le type de graphe qui peut poser problème (p entre 0,1 (pour Degmin et DegMax) et 0,2 (pour FGK)).

On peut également s'intéresser aux informations structurelles des graphes. En se basant sur la méthodes des *B-matrices* [BBSBA08], on obtient une ≪ empreinte ≫ visuelle du graphe (voir Figure 8 qui représente des B-matrices cases rectangulaires)). Une B-Matrice est une matrice qui contient à la cellule $B(l, k)$ le nombre de noeuds ayant k voisins à la distance géodésique l (une *l-shell* correspond, pour un noeud x, à $N^l(x) - N^{l-1}(x)$). Pour $l = 1$, on a la distribution des degrés du graphe (ligne du bas sur la figure). La couleur représente le nombre de noeuds (0 pour blanc) : de bleu sombre (faible nombre) à rouge (grand nombre). Sans aller plus loin, cela permet d'avoir une première intuition sur le type de graphes qui peut mettre à l'épreuve les algorithmes stable max.

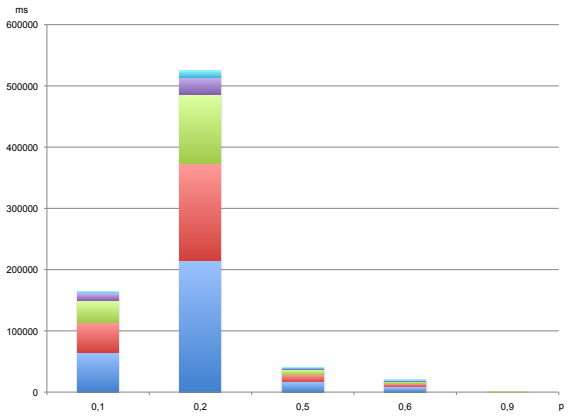
En ce qui concerne la complexité ≪ pratique ≫ des algorithmes par rapport à leur borne théorique, il suffit de faire une régression non-linéaire sur les données que l'on a obtenues durant nos tests pour savoir si en effet, cette fameuse borne est atteinte. En se basant sur la méthode de Gauss-Newton, on cherche une fonction de la forme :

$$f(x, \theta) = \theta_1 e^{-\theta_2 . x}$$

Considérons un ensemble de variables explicatives sans erreur x et un ensemble de variables dépendantes observées y, alors chaque variable y peut être modélisée comme une variable aléatoire dont la moyenne est donnée par $f(x, \theta)$. On a finalement un problème de moindres carrés non-linéaires que l'on résout par l'algorithme de Gauss-Newton, et que nous ne détaillerons pas ici.
La figure 9 représente les courbes de la borne théorique et de résultats obtenus



(a) Proportion des appels

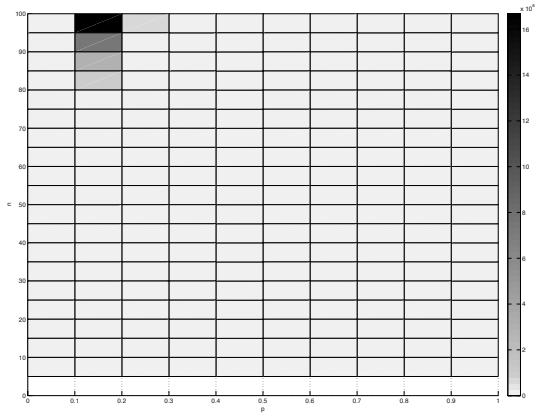


(b) Nombre d'appels

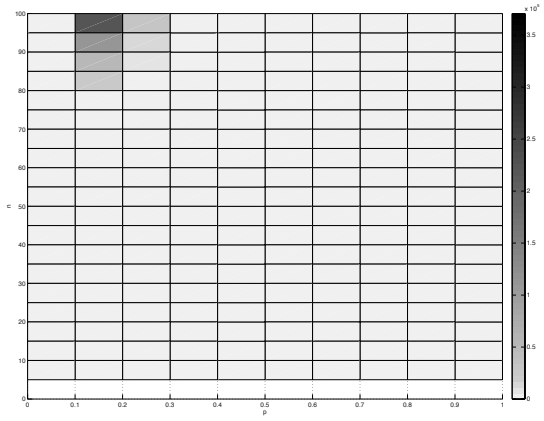
FGK	Composante connexe	Nœud dominé	Folding	Mirroring		Temps (ms)	n=100, p=
65430	48984	35230	12297	2691		5551	0,1
215186	158280	112210	28710	10835		14578	0,2
16740	12130	8906	1196	1385		1658	0,5
8378	6010	4467	422	824		1159	0,6
100	99	99	0	0		405	0,9

(c) Données numériques (nombre d'appels)

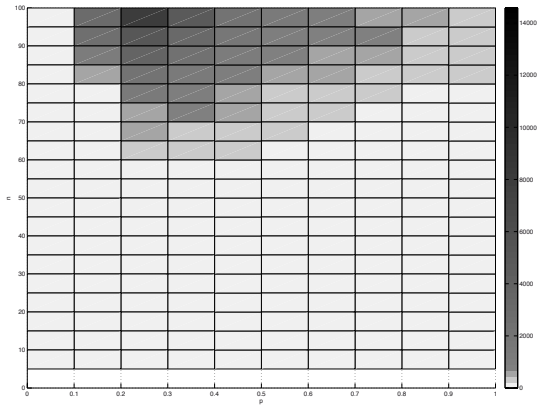
FIGURE 6 – Appels des méthodes de l'algorithme FGK



(a) DegMin

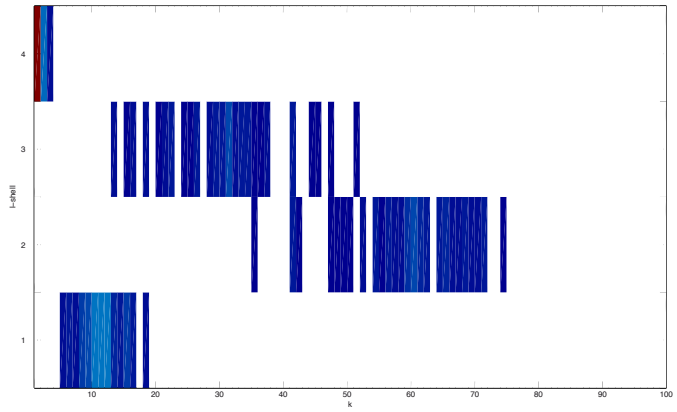


(b) DegMax

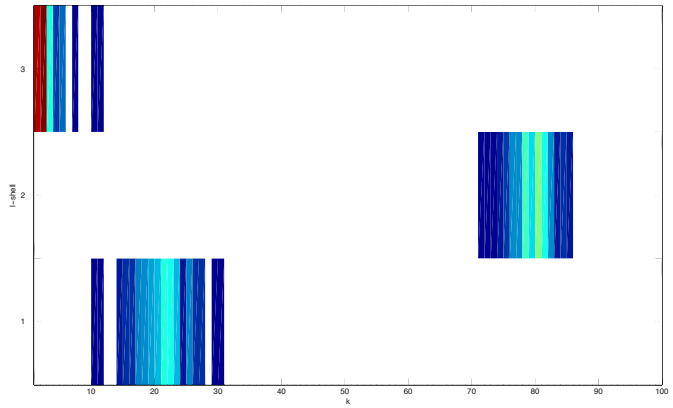


(c) FGK

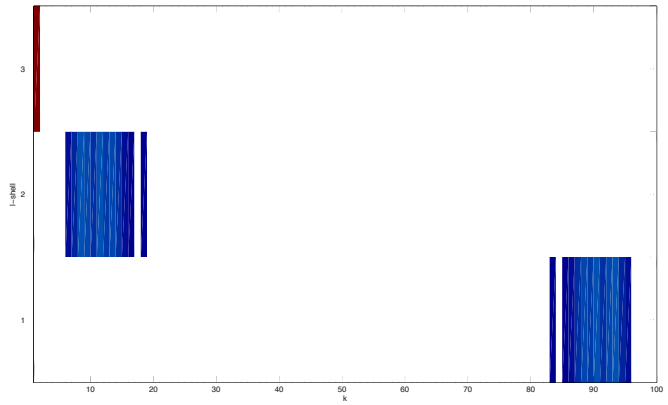
FIGURE 7 – Temps d'exécution en ms sur l'ensemble des $G_{n,p}$ pour n=[5 :100] par pas de 5 et p=[0.0 :1.0] par pas de 0.1



(a) $G_{100,0.1}$



(b) $G_{100,0.2}$



(c) $G_{100,0.8}$

FIGURE 8 – B-Matrices des graphes ER pour $n = 100$ et $p = 0.1, 0.2, 0.8$

avec FGK où celui-ci met le plus de temps à trouver le stable max. On constate qu'on est assez éloigné de la borne théorique (ce qui conforte l'idée que les graphes $G_{n,p}$ ne sont pas forcément adaptés à notre étude).

4 Perspectives

Jusqu'à maintenant, on ne s'est intéressé qu'aux algorithmes stable max, et également à des aspects techniques. Cette étape passée, nous pourrons nous pencher sur les prochains algorithmes à implémenter et étudier :

- Vertex cover
- Kernelisation
- Séparateurs minimaux
- Cliques maximales potentielles

Un travail sur Mascopt va être mené afin d'adapter une classe spécialisée aux problèmes étudiés, la librairie n'étant de base pas dédiée à ces problématiques.

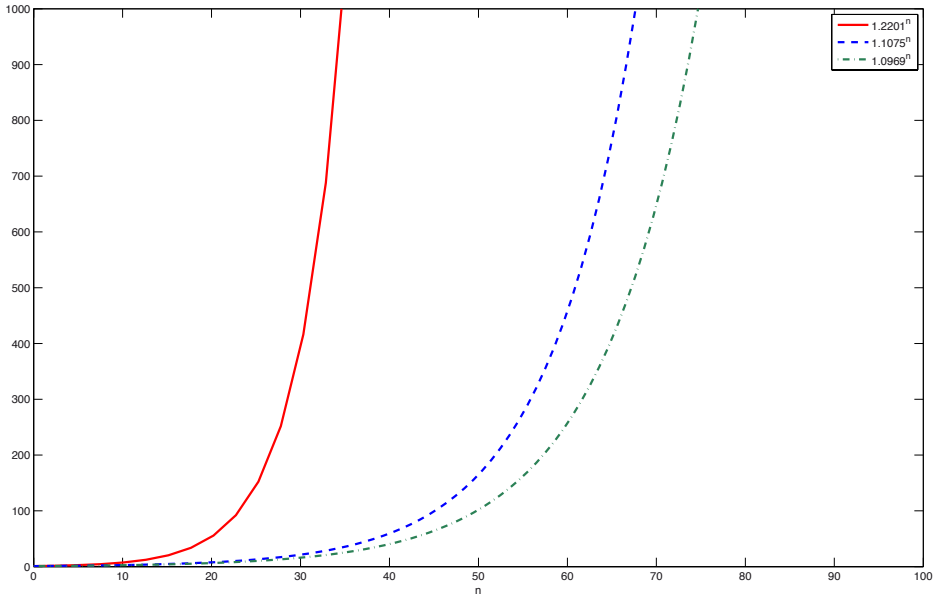


FIGURE 9 – Visualisation pour FGK de la borne théorique $O^*(1.2201^n)$ et des bornes pratiques obtenues pour $p = 0.1$ $(1,0969^n)$ et $p = 0.2$ $(1,1075^n$ pire cas pratique rencontré)

Références

[BBSBA08] J. P. Bagrow, E. M. Bollt, J. D. Skufca, and D. Ben-Avraham. Portraits of complex networks. *EPL*, 81, 2008.

[ER59] Paul Erdös and Alfréd Rényi. On random graphs. *Publicationes Mathematicae*, 6 :290–297, 1959.

[FGK09] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM*, 56(5) :1–32, 2009.

[FK10] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 2010.

[Lie07] Mathieu Liedloff. *Algorithmes exacts et exponentiels pour les problèmes NP-difficiles : domination, variantes et généralisations*. PhD thesis, Université Paul Verlaine, 2007.

[MM65] J.W. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3 :23–28, 1965.

[New02] M. E. J. Newman. Random graphs as models of networks. *Handbook of Graphs and Networks*, 2002.