BERNA Fabrice
PECLIER Thomas
COLIN Estelle

IUP Génie Mathématiques & Informatique

ALGORITHME DISTRIBUE
PROJET : COLORATION DE GRAPHE

Tuteurs de stage : *M. BAGET*
M. KONIG
M. HABIB

1. Sommaire

2. Introduction

Qu'est-ce qu'un algorithme distribué?
Par définition, un algorithme est un ensemble d'instructions qui régit le déroulement d'un programme informatique.
Un algorithme distribué : se dit d'un algorithme s'il est exécuté de manière simultanée sur un ensemble de ressources. Cette exécution, en simultanée sur plusieurs ressources distinctes, permet alors la réalisation d'un seul et même calcul. Le comportement de chaque processus est déterminé par un algorithme local et la communication entre les processus se fait par échange de messages uniquement.

Dans la réalisation d'un calcul, il est possible de rencontrer trois types d'exécution, une exécution linéaire, une exécution répartie ou encore une exécution distribuée.
Lors d'une exécution linéaire, le calcul est réalisé sur une seule ressource à la fois ; alors que lors d'une exécution répartie, le calcul se fait de manière analogue et simultanée sur toutes les ressources. Lors d'une exécution distribuée, le calcul est partagé entre toutes les ressources et s'exécute, aussi, de façon simultanée.
Lors de la réalisation de ce projet pour lequel nous avons utilisé le langage Java, nous tenterons de mesurer les améliorations entre une exécution linéaire et une exécution distribuée, si amélioration il y a.

Le problème que nous avons tenté de réaliser est la coloration d'un réseau en un nombre de couleurs imposé, k, soit un problème de k-coloration, que nous avons assimilé à une coloration de graphe. Le problème de k-coloration sur un graphe, ensemble de sommets dont chaque sommet est relié, ou non, à un ou plusieurs autres sommets, consiste à trouver une couleur pour chaque sommet de sorte qu'elle soit différente de celle de ses voisins. Et ce, bien sur, en ne dépassant pas un nombre k de couleurs demandées.

Tout d'abord, nous définirons la structure que nous avons mise en place pour ce projet. Nous vous expliquerons par la suite, une solution linéaire (circulation d'un jeton) au problème de k-coloration. Et enfin, la solution orientée distribué que nous avons mise en place.

3. Résolution du problème par circulation d'un jeton

3.1. Election à jeton

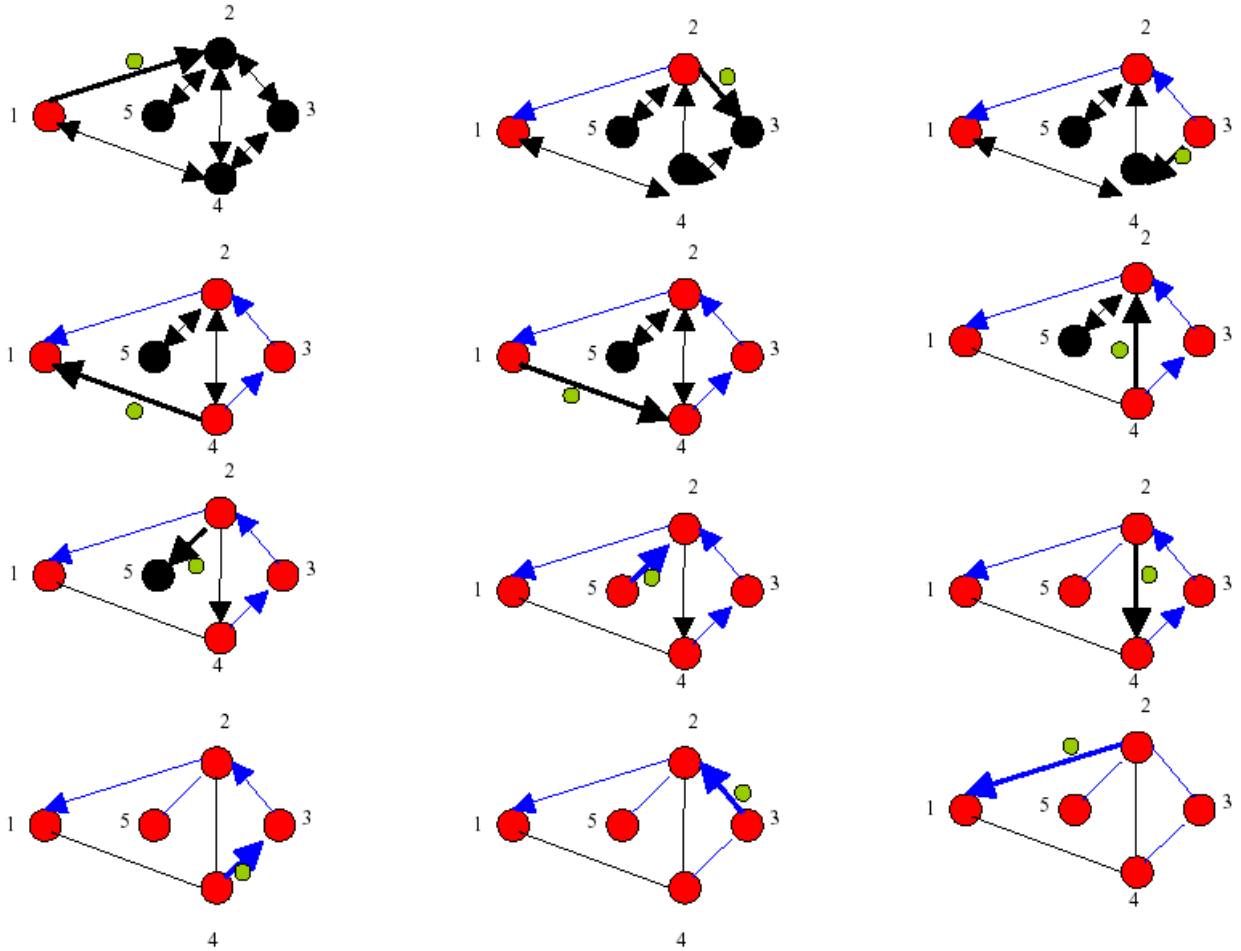
3.1.1. Algorithme de parcours du graphe de communication

3.1.1.1. Présentation de l'algorithme

On généralise l'algorithme de CHANG et ROBERTS (algorithme d'élection dans un graphe en anneau) à un graphe de communication quelconque. La difficulté réside dans le fait que la connaissance d'un graphe par un sommet se limite à ses voisins.
Cette propriété est suffisante pour gérer un parcours du graphe de communication tel que :

1. ce parcours se termine chez l'initiateur
2. lorsqu'il se termine, ce parcours a visité au moins une fois tous les sites et a traversé exactement une fois chaque canal de communication dans les deux directions

Expliquons l'algorithme sur l'exemple suivant :



Chaque canal sortant est initialement considéré ouvert (ce qui est représenté par un flèche en extrémité). Lorsqu'un message est envoyé sur ce canal, celui-ci se ferme, et ne pourra plus être utilisé par le parcours.
Un site déjà visité est représenté en rouge. Dans le cas contraire, il est représenté en noir.
Lors de la première visite d'un sommet par ce parcours, le site mémorise le canal comme étant celui de son père (en bleu sur la figure). Il n'utilisera ce canal qu'après avoir utilisé tous les autres canaux.

Déroulons l'exemple :

1. Le site initie un parcours en envoyant le message au site 2. Par conséquent, il clôt ce canal.
2. À la réception, le site 2 mémorise que le canal de 2 vers 1 est le canal de "retour". Il a le choix entre les trois autres canaux et décide de l'envoyer au site 3 (canal fermé).
3. À la réception, le site 3 mémorise que le canal de 3 vers 2 est le canal de "retour". Il ne dispose que d'un autre canal et renvoie au site 4 (canal fermé).
4. À la réception, le site 4 mémorise que le canal de 4 vers 3 est le canal de "retour". Il a le choix entre les deux autres canaux et décide de renvoyer au site 1 (canal fermé).
5. À la réception, le site 1 ne dispose plus que d'un autre canal et le renvoie au site 4 (canal fermé).
6. À la réception, le site 4 ne dispose plus que d'un canal différent du canal de retour et le renvoie au site 2 (canal fermé).
7. À la réception, le site 2 a le choix entre deux canaux différents du canal de retour et décide de renvoyer au site 5 (canal fermé).
8. À la réception, le site 5 mémorise que le canal de 5 vers 2 est le canal de "retour". Il ne dispose d'aucun autre canal et le renvoie au site 2 (canal fermé).
9. À la réception, le site 2 ne dispose plus que d'un canal différent du canal de retour et le renvoie au site 4 (canal fermé).
10. À la réception, le site 4 ne dispose que du canal de retour et le renvoie au site 3 (canal fermé).
11. Il. A la réception, le site 3 ne dispose que du canal de retour et le renvoie au site 2 (canal fermé)
12. À la réception, le site 2 ne dispose que du canal de retour et le renvoie au site 1 (canal fermé). Lorsque ce message arrive au site 1, celui-ci n'ayant plus de canal ouvert conclut que le parcours est terminé.

3.1.2. Preuve de l'algorithme

Tout d'abord, le parcours se termine nécessairement car un canal n'est emprunté qu'au plus une fois et le nombre de canaux est fini. Supposons qu'il se termine ailleurs que chez initiateurs. Ceci signifie que ce site i n'a plus de canal sortant ouvert, lors d'une visite. Or puisqu'il décrémente le nombre de canaux sortants à chaque visite. On en conclut qu'il a été visité plus de n fois. Mais puisque n est également le nombre de canaux entrants, cela signifie qu'un canal entrant a été emprunté plus d'une fois. Ce qui est impossible. Le parcours se termine donc chez l'initiateur.
Démontrons finalement que tous les sites sont visités. Si ce n'est pas le cas, on partitionne l'ensemble des sites entre ceux qui sont visités et les autres. Puisque le graphe est connexe, il existe au moins une arête reliant un site visité à un site non visité. D'après le paragraphe précédent, ce canal a été emprunté d'où la contradiction.

3.1.3. Variables du site i

Pour gérer un parcours initié par un quelconque des sites, on utilise un tableau T indicé par les sites. Ceci peut sembler contradictoire avec l'hypothèse de minimalité sur la connaissance des sites, mais il est facile de remplacer T par une allocation dynamique au prix d'une programmation plus "lourde".
L'algorithme travaille avec des ensembles de canaux (ou voisins). La fonction extraire(ensemble) renvoie un élément de l'ensemble (s'il est non vide) et le retire de celui-ci. On s'autorise bien entendu à tester si un ensemble est vide.

- Voisins: constante contenant l'ensemble des voisins de i dans le graphe de communication.
- T[1..N] : table de routage pour réorienter les messages.
- prochain : variable contenant l'identité d'un voisin de i.

3.1.4. Algorithmes du site i

Pour initier un parcours du graphe, on initialise le champ voisins aux voisins du site et on extrait un voisin de cet ensemble pour débuter le parcours.

```
initier(type)
       T[i].voisins = i;
       Prochain = extraire(T[i].voisins) ;
       envoyer(type,i) à prochain;
Fin


sur-réception-de(j, (type,k))
       Si (!faire-suivre(j,type,k)) Alors
       afficher("parcours terminé") ;
       Fin
Fin
```

S'il s'agit de la première visite de ce parcours, on initialise les champs père et voisins de la cellule concernée en extrayant le "père" des voisins. S'il reste des voisins, on extrait l'un d'entre eux pour continuer le parcours. Sinon si i n'est pas l'initiateur du parcours, on emprunte le canal de son père. Le parcours étant terminé sur ce site, on réinitialise le champ père pour un éventuel nouveau parcours. Si i est l'initiateur, la fonction renvoie l'indication que le parcours est terminé.

```
faire-suivre(j,type,k)
        Si ((k!=i) && (T[k].père==i)) Alors
               T[k].père = j ;
               T[k].voisins = Voisins\{j};
        Fin
        Si T[k].voisins != 0 Alors
               prochain = extraire(T[k].voisins) ;
               envoyer(type,k) à prochain ;
               renvoyer (VRAI) ;
        Sinon
               Si (k!=i) Alors
                      envoyer(type,k) à T[k].père;
                      T[k].père = i;
                      renvoyer (VRAI) ;
               Sinon
                      renvoyer(FAUX) ;
               Fin
        Fin
 Fin
```

3.2. Description de l'algorithme d'élection

3.2.1. Variables du site i

Aux variables nécessaires au parcours, on ajoute les variables nécessaires à l'élection

- ***état*** : état du service. Cette variable prend les valeurs parmi l'ensemble (repos, encours, terminé). Elle est initialisée à repos
- ***chef*** : identité du site élu

3.2.2. Algorithme du site i

Si le site participe pour la première fois à un processus d'élection, celui-ci initialise un tel processus.

```
Candidature()
        Si (pas de voisins) Alors je suis élu
        Sinon
               Si (Etat = repos) Alors
                      Etat = encours ;
                      Chef = i ;
                      Initier(« Election ») ;
               Fin
        Fin
 Fin
```

A la réception d'une requête « Election », si le site est au repos, alors son état devient « encours ». Dans le cas où il reçoit une meilleure requête, il en tient compte et la fait suivre. Dans le cas contraire, la requête est avortée (disparition de la requête). Par conséquent, seule la meilleure requête revient à son initiateur. Dans le cas où la requête nous revient, on est l'élu. On initie un message de confirmation à tout le graphe pour annoncer l'identité de l'élu.

```
Sur_Reception_Election(initiateur)
        Si ((Etat = repos) || (initiateur>=chef) Alors
               Etat = encours ;
               Chef = initiateur ;
               Si ( ! faire_suivre(« Election »,initiateur)) Alors
                      Etat = terminé ;
                      Initier(« Elu », i) ;
               Fin
        Fin
Fin
```

Le message « Elu » fait un parcours de tout le graphe. Lorsque le message revient à l'élu, celui-ci lance une coloration.

```
Sur_Reception_Elu(initiateur)
       Etat = terminé ;
       Si ( ! Faire_suivre(« Elu »,initiateur)) Alors
              Si (i=chef) Alors
                     Je suis Elu, Je lance une coloration ;
              Fin
       Fin
Fin
```

3.2.3. Complexité de l'algorithme

Lorsque deux requêtes sont envoyées par deux initiateurs, seule celle émise par le site de plus grande identité reviendra à son point de départ.
Une fois élu, l'initiateur réémet un message de confirmation qui parcours tout le graphe pour informer l'identité de l'élu.
Chaque initiateur initie un parcours de graphe et l'élu initie le message de confirmation. En notant E le d'arêtes du graphe de communication, on obtient :

$Pire(n) = (n+1).(2.E) = ?(n.E)$

3.3. Coloration à jeton

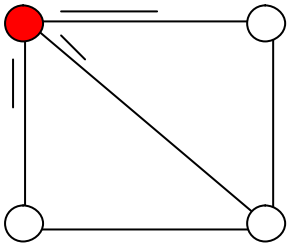
La coloration en linéaire s'effectue par circulation d'un jeton, c'est-à-dire qu'à un instant donné, un seul sommet du graphe est en train de choisir une couleur.

La circulation du jeton se fera sur l'arbre recouvrant exclusivement. Cet arbre se construira au fur et à mesure de la coloration, par l'envoi ou non de la couleur à ses voisins.

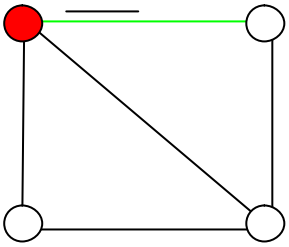
Dans le graphe, un sommet aura des voisins, un père, des aïeuls, et des fils.
Voisins : tous les sommets voisins.
Père : le sommet qui lui a demandé de se colorer.
Aïeuls : tous les voisins colorés (ils se sont colorés avant moi).
Fils : tous mes voisins non colorés.

Un sommet donné ne communiquera sa couleur qu'à ses Fils. Ses Aïeuls, qui sont prioritaires sur lui pour choisir leur couleur n'ont pas besoin de connaître sa couleur.

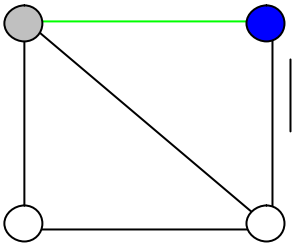
3.3.1. Schéma de principe de coloration par jeton



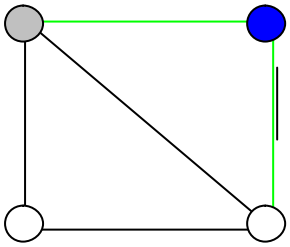
Phase 1 :
L'élu choisit sa couleur et en informe ses voisins



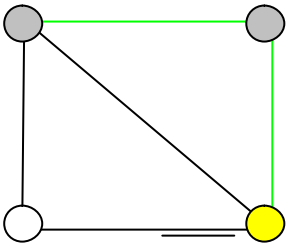
Phase 2 :
L'élu demande à un de ses fils de se colorer.



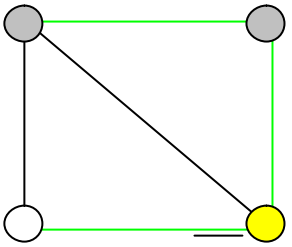
Phase 3 :
Le sommet choisit sa couleur et en informe ses fils



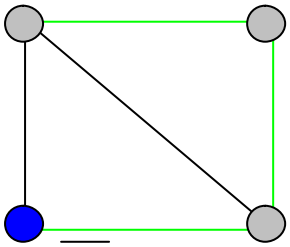
Phase 4 :
Le sommet demande à un de ses fils de se colorer.



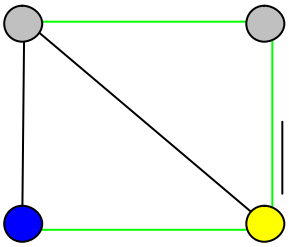
Phase 5 :
Le sommet choisit sa couleur et en informe ses fils



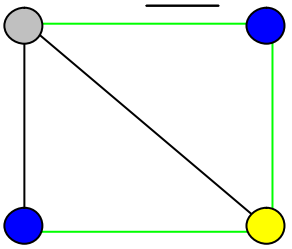
Phase 6 :
Le sommet demande à un de ses fils de se colorer.



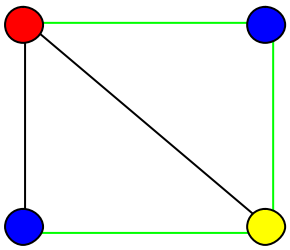
Phase 7 :
Le sommet choisit sa couleur. Il n'a plus de fils : remonte OK



Phase 8 :
Plus de fils non coloré.
Remonte OK



Phase 9 :
Plus de fils non coloré.
Remonte OK



Phase 10 :
Plus de fils non coloré.
→ **Coloration réussie**

3.3.2. Implémentation de l'algorithme de coloration

3.3.2.1. Variables du site i

Les variables nécessaires à la coloration sont les suivantes.

- ***nbCouleur*** : le nombre de couleurs de la k-coloration
- ***maCouleur*** : la couleur que j'ai choisie
- ***couleursLibres***[] : liste de toutes les couleurs libres (les couleurs utilisées sont celles de mes aïeuls)
- ***voisinsColorés***[] : liste de tous mes voisins colorés, c'est-à-dire mes aïeuls
- ***père*** : l'identifiant de mon père dans l'arbre recouvrant.

3.3.2.2. Algorithme du site i

Une fois que le site a reçu son message de confirmation, il lance une coloration. S'il n'a pas de voisin, il se considère déjà coloré. Dans le cas contraire, il choisit une couleur (la première libre), la diffuse à tous ses voisins, et demande à son premier voisin de se colorer.

```
initierColoration()
    Si (pas de voisins) Alors
           Etat = coloré ;
           maCouleur = 0 ;
    Sinon
           maCouleur = choisirCouleur() parmi les couleurs libres;
           envoyer(maCouleur) à tous mes voisins ;
           envoyer(« Coloration ») à mon premier voisin ;
    Fin
Fin
```

Quand un site reçoit une couleur d'un sommet voisin, il considère que cette couleur n'est plus libre, et sauve le fait que ce voisin est coloré.

```
sur_Reception_Couleur(couleur, j)
    monVoisin(j) est coloré ;
       la couleur reçue n'est plus une couleur libre ;
Fin
```

Quand un sommet reçoit un message de coloration d'un autre site, celui-ci est son père dans l'arbre. Si la couleur provient d'un autre sommet que le père (un des aïeuls), on envoie le message OK à celui-ci, car on a déjà été coloré. Dans le cas contraire, le site choisit une couleur, et la diffuse à tous ses voisins. Si tous ses voisins sont colorés, alors le site i est une feuille de l'arbre recouvrant, il renvoie OK à son père. Dans le cas contraire, il demande à un des ses voisins non coloré de se coloré. Si le site n'a plus de couleur libre, il renvoie Erreur à son père.

```
sur_Reception_Coloration(j)
    Si (je n'ai pas encore de père) père = j ;
    Si ((j'ai un père) && (père != j)) Alors
           J est un aïeul, j'ai déjà été coloré, envoie(« OK ») à j ;
    Sinon
           maCouleur = choisirCouleur() parmi les couleurs libres;
           Si (j'ai trouvé une couleur) Alors
                  envoyer(maCouleur) à tous mes fils ;
                  Si (tous mes voisins sont colorés) Alors
                         Etat = coloré ;
                         Envoie(« OK ») à mon père ;
                  Sinon
                         Envoie(« Coloration ») à un voisin non coloré
                  Fin
           Sinon
                  Etat = pasColoré ;
                  Envoie(« Erreur ») à mon père ;
           Fin
    Fin
Fin
```

Lorsqu'un site reçoit un message OK d'un autre, il considère que ce site est coloré. Si tous ses fils sont colorés et que le site n'a pas de père, la coloration s'est terminée avec succès. Si tous ses fils sont colorés et que le site a un père, on envoie à celui-ci un message OK signalant que tous les fils de cette branche de l'arbre recouvrant sont bien colorés. Dans le cas contraire, le site demande à un de ses fils non colorés de se colorer.

```
sur_Reception_Ok(j)
    Mon fils j est coloré
    Si (tous mes fils colorés) Alors
           Etat = coloré ;
           Si (je n'ai pas de père) Alors
                  « Coloration terminée avec succès »
           Sinon
                  Envoie(« Ok ») à mon père
           Fin
    Sinon
           Envoie(« Coloration ») à un des mes fils non coloré ;
    Fin
Fin
```

Lorsqu'un site reçoit un message d'erreur de ses fils, celui-ci considère d'abord que tous ses fils ne sont plus colorés. Il choisit à nouveau une couleur, la diffuse à tous ses fils, et demande à un de ses fils de se colorer. Si le site ne parvient pas à choisir une couleur, alors, si il a un père, il lui envoie un message d'Erreur. Dans le cas contraire, la coloration a échoué.

```
sur_Reception_Erreur()
    Tous mes fils ne sont plus colorés ;
    maCouleur n'est plus une couleur libre ;
    maCouleur = choisirCouleur() parmi les couleurs libres ;
    Si (j'ai trouvé une couleur) Alors
           Envoie(maCouleur) à tous mes fils ;
           Envoie(« Coloration ») à un de mes fils ;
    Sinon
           Si (j'ai un père) Alors
                  Envoie(« Erreur ») à mon père ;
           Sinon
                  « Coloration ratée »
           Fin
    Fin
Fin
```

4. Vers une solution distribuée

4.1. Vers une Coloration distribué…

On peut aborder les problèmes d'algorithmique distribuée de deux manières :

- En faisant fi de l'algorithmique classique, considérant que la nature des problèmes n'a rien à voir, et repartant sur des bases nouvelles.
- En se servant de ce qui existe en centralisé, et l'adaptant à un système distribué.

C'est sous cette deuxième optique que nous aborderons le problème.
Nous allons nous servir de l'algorithme de coloration le plus efficace que nous connaissons, à savoir le BackTrack, en l'adaptant pour qu'il s'exécute de manière distribuée.

Nous avons vu dans la partie précédente que l'algorithme de BackTrack, dans un système distribué, s'exécutait de manière répartie, mais également de manière linéaire.
Or, il est trivial de s'apercevoir que cette linéarité découle du fait qu'un sommet coloré demande de se colorer à **un seul de ses voisins à la fois**.

Aussi, si l'on veut distribuer cet algorithme, on va le modifier de manière à ce qu'un sommet coloré demande de se colorer **à tous ses voisins simultanément**. Ainsi, plusieurs sommets exécuteront la coloration en parallèle.

4.2. Les conflits de voisinage

La problématique de l'algorithme, résultant de ce parallélisme, réside dans le fait que certains de ces sommets peuvent être voisins. Un sommet déterminant sa couleur en fonction des couleurs de ses voisins, comment faire pour que deux sommets, voisins, choisissant leur couleur simultanément, l'un ignorant donc la couleur de l'autre, ne choisissent pas la même couleur ?
Une manière de garantir cela consiste à retarder la coloration d'un des sommets en conflit, le forçant à attendre de connaître la couleur de l'autre sommet. Toutefois, l'instauration de cette relation d'ordre total entre deux sommets conflictuels présente de nouveaux problèmes complexes.

On peut aussi admettre que chaque sommet choisit sa couleur avant de résoudre d'éventuels conflits avec ses voisins s'étant colorés simultanément.

- La résolution d'un conflit entre deux sommets, devant donner la priorité à l'un ou l'autre selon des critères communs :
- identifiants (max ou min)
- Dates (de coloration, de réception de message, etc.) introduisant ainsi une synchronisation
- Etc.

4.3. Nécessité d'une relation d'ordre

Par conséquent, l'instauration d'une relation d'ordre entre les sommets du graphe est inévitable. Cette relation d'ordre ayant pour principal objectif de déterminer des priorités entre des sommets voisins, de manière à empêcher tout conflit ; deux sommets voisins ne peuvent avoir le même numéro d'ordre. Ainsi, si la coloration respecte l'ordre, aucun sommet voisin ne se colorera en parallèle.

Etablir une telle relation d'ordre revient à construire un arbre recouvrant du graphe tel qu'aucun sommet voisin ne puisse avoir le même niveau dans l'arbre.
NB : on appelle ici niveau la distance, en nombre d'arcs, jusqu'à la racine de l'arbre.
Ce qui revient à dire que deux sommets voisins ne peuvent avoir le même père dans l'arbre recouvrant.

Cette solution est toute naturelle ; on comprend aisément que le conflit survient lorsqu'un sommet colorié demande à des fils voisins de se colorier. Or, si l'on construit un arbre tel que deux fils d'un père ne puissent être voisins, on élimine les cas de conflits.

Ainsi, la coloration dans l'arbre va se faire telle une diffusion, tous les sommets de mêmes niveaux pourront travailler en même temps car ils seront indépendants les uns des autres (du point de vue de la coloration).

4.4. Principe Général de l'algorithme

Le processus de coloration se décomposera donc en deux phases :

- Etablir la priorité des sommets par rapport à leurs voisins
- Colorier les sommets en fonction de leurs priorités

4.4.1. Etablir la relation d'ordre, Construire l'arbre de priorités

La problématique consiste à trouver un algorithme distribué qui permet de construire un arbre recouvrant du graphe tel qu'aucun sommet voisin ne puisse avoir le même père.

La difficulté est de déterminer lesquels des fils voisins doivent avoir la priorité sur les autres, sachant que ces fils, ne connaissant pas la structure du graphe, ne peuvent savoir si leurs voisins sont aussi des fils.

4.4.2. Une élection définit une priorité

Choisir un sommet parmi un ensemble de sommets de manière distribuée, revient précisément à effectuer une élection dans cet ensemble de sommets.

On va donc pouvoir déterminer ce sommet en effectuant un algorithme d'élection dans le sous graphe constitué du graphe de départ ôté du sommet père.

Cependant un algorithme d'élection pourra élire un sommet n'étant pas un voisin (ou fils) du sommet ôté, ce qui est contradictoire avec le but qui consistait à construire un arbre recouvrant.

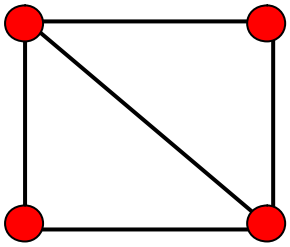
Néanmoins, une petite modification sur notre algorithme d'élection suffit à éviter ce problème : Seuls les voisins du sommet ôté vont se déclarer candidats, les autres sommets de a composantes ne servant qu'à relayer les messages d'élection. Autrement dit, les sommets non Candidat ne deviennent pas candidats en recevant des messages d'élection.

Par ailleurs, si le graphe de départ est connexe, le sous graphe pourra comporter plusieurs composantes connexes.
C'est la pluralité de ces composantes qui va rendre l'algorithme de coloration distribué car un algorithme d'élection exécuté sur un graphe donne autant d'élus que de composantes connexes.
On verra plus tard comment utiliser cette propriété pour optimiser l'algorithme.

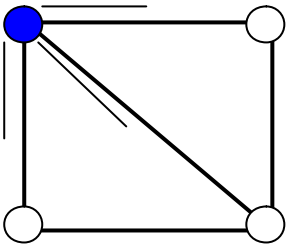
L'arbre recouvrant des priorités va donc se construire de manière distribuée, en parallèle dans toutes les composantes connexes générées par la suppression (virtuelle) d'un sommet.
On peut même affirmer que la vitesse de cet algorithme va augmenter de manière exponentielle au fur et à mesure:

- les tailles des composantes connexes vont se réduire, donc les élections vont être de plus en plus rapides générant ainsi plus vite de nouvelles composantes connexes plus petites.
- Le nombre des composantes connexes va augmenter, donc le nombre d'élections s'effectuant en parallèle va augmenter générant de plus en plus de composantes en un temps plus court.

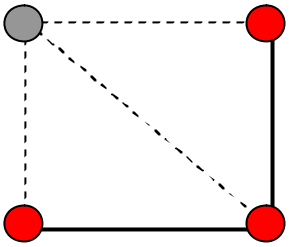
4.4.3. Une trace pour comprendre



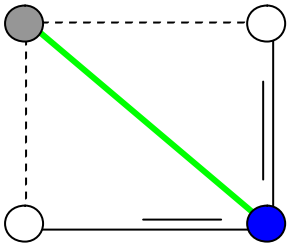
Phase 1 :
Election de la racine
Tous les sommets sont candidats



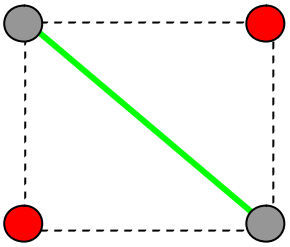
Phase 2 :
Le sommet élu en informe ses voisins



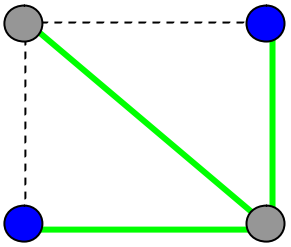
Phase 3 :
Ils lancent chacun une élection, ignorant le sommet déjà élu



Phase 4 :
Un des trois candidats est élu,
Il enregistre son père,
Il informe ses voisins



Phase 5 :
Ils lancent chacun une élection, ignorant le sommet déjà élu



Phase 6 :
Les 2 sommets sont élus
Ce sont des feuilles :
L'arbre est construit

4.4.4. La condition d'arrêt

L'arbre se construisant de la racine vers les feuilles, l'algorithme de construction se termine par conséquent sur une feuille.
Or, une feuille constatant la terminaison de l'algorithme, ne peut pas savoir si l'algorithme a fini de s'exécuter sur tous les sous arbres.

Le seul sommet pouvant constater la terminaison de l'algorithme sur tout le réseau est la racine de l'arbre. Il va donc être nécessaire que chaque feuille fasse remonter un avis de terminaison jusqu'à la racine.

Chaque sommet ne connaît de ses voisins que s'ils sont plus hauts ou plus bas dans l'arbre, sans distinguer d'ordre dans ces deux groupes de sommets.
Par conséquent, on projettera de remonter le message de terminaison, par la transmission de ce message à tous les sommets situés plus haut dès lors que l'on aura reçu ceux de tous les sommets situés plus bas :

- Une feuille envoyant donc un avis de fin à tous ces voisins
- La racine attendant un avis de fin de tous ses voisins

Parmi ses voisins, un sommet distingue ceux faisant déjà partie de l'arbre des autres car ceux-ci leur ont envoyé un avis d'élection lors de la construction de l'arbre.

4.4.5. L'algorithme

```
Initialisation
  ListeElus [ NbVoisins ] ← Faux  // i mémorise tous ses voisins déjà dans l'arbre
  Père ← i
  PèreTemp ← i
  NbAcqAttendus ← NbVoisins
  Je lance une élection dont je suis candidat

A la fin de l'élection :
      Si je suis élu
      Alors   Si PèreTemp != i
                    Alors Père ← PèreTemp
              Si NbVoisins >1
                    Alors   Pour tout voisin k tel que ListeElus[k]  = Faux
                      Envoyer <ELU> à k
                    Sinon   Pour tout voisin k tel que  ListeElus[k]  = Vrai // i
              est une feuille
                      Envoyer <FIN> à k

A la réception de <ELU> de j
  PèreTemp ← j
  ListeElu[j] ← vrai
  NbAcqAttendus ← NbAcqAttendus - 1
  Je lance une élection dont je suis candidat

A la réception de <FIN> de j
  NbAcqAttendus ← NbAcqAttendus + 1
  Si NbAcqAttendus = 0
        Si Père != i
              Alors Envoyer <FIN> à Père
              Sinon   // i est la racine
                    //L'arbre est construit, on va pouvoir lancer la coloration
```

4.4.6. Quel algorithme d'élection utiliser ?

Il reste, pour cette phase de la coloration, à déterminer sur quel algorithme d'élection implémenter la construction de notre arbre de priorités.

Nous ne connaissons que deux algorithmes d'élections dans un graphe quelconque :

- L'algorithme d'élection à jetons, exposé précédemment dans ce rapport
- L'algorithme d'élection par diffusion, présenté en cours

La différence entre ces deux algorithmes réside dans le seul principe que, dans l'élection par diffusion, on transmet le message d'élection à tous ses voisins (sauf son père) en même temps. De ce fait, l'exécution est plus distribuée que l'élection à jeton qui est linéaire pour chaque candidat.

Etant dans l'optique de rendre distribué le processus de coloration, à première vue, on pourrait en conclure que l'algorithme par diffusion correspond mieux à nos attentes.

Cependant, la construction de l'arbre donne lieu à plusieurs élections en parallèle, lesquelles comportent plusieurs candidats. Par conséquent, l'utilisation de l'algorithme à jetons, ne serait pas contraire à notre volonté d'une exécution distribuée.

Par ailleurs, on pourrait se baser sur des critères d'efficacité pour choisir l'algorithme d'élection.

A priori, il semble qu'une élection à jeton nécessite moins de messages qu'une élection par diffusion. En effet, il circule simultanément un seul jeton par sommet candidat sur le réseau, alors que des messages peuvent se croiser dans le cas de la diffusion.
En revanche, le fait de diffuser le message à tous ses voisins, doit logiquement accélérer la propagation du message d'élection de chaque sommet. Le temps d'exécution doit donc être plus court.

La valeur de ces deux critères dépend également fortement de la topologie du graphe.

Il semble que moins le nombre de candidats est important, plus l'algorithme à jeton est rapide par rapport à l'algorithme à diffusion. Or plus le graphe sera connexe, plus le nombre de candidats sera important (car plus il y aura de voisins parmi les fils).

A contrario, plus la structure du graphe s'apparente à un arbre (moins le graphe ne comportera de cycles), plus l'algorithme par diffusion sera rapide (car on ne rediffusera par vers des sommets déjà visités).

Par conséquent, le trop grand nombre de paramètres indépendants rend l'estimation de l'efficacité très compliquée. On considère donc les complexités de ces deux algorithmes approximativement équivalentes.

Notre choix se portera donc arbitrairement sur l'algorithme d'élection par diffusion. Il semble plus distribué, et, par ailleurs, il nous paraît instructif de l'implémenter.

4.5. Une Coloration distribuée

On considère que chaque sommet connaît le nombre k de couleurs avec lequel doit s'effectuer la coloration.
Dès lors que l'arbre est construit, la coloration devient relativement simple.

4.5.1. Choisir une couleur : quand et comment ?

Un sommet se coloriant selon son niveau dans l'arbre, il ne doit, pour choisir sa couleur, que tenir compte des couleurs choisies par ses voisins situés plus haut que lui dans l'arbre. Il peut ignorer les couleurs des sommets situés plus bas, ceux-ci étant moins prioritaires pour choisir leurs couleurs.

Un sommet doit donc attendre de connaître la couleur de tous ses voisins situés plus haut que lui pour choisir sa couleur.

Notre algorithme de coloration va s'exécuter en parcourant les sommets du graphe de la racine aux feuilles. Les sommets de mêmes niveaux pouvant se colorier en parallèle sans aucun risque de conflit, puisque non voisins.

Au moment de choisir sa couleur, un sommet i doit donc connaître les couleurs déjà choisies par ses voisins (par définition les voisins situés plus haut dans l'arbre). Il va ensuite communiquer sa couleur aux voisins n'ayant pas encore de couleurs (par définition situés plus bas dans l'arbre).
Parmi ces derniers, un sommet (ou plusieurs si certains sont voisins), a (ont) le sommet i comme père. Il(s) est (sont) donc le(s) suivant(s) dans l'ordre de priorité de coloration. C'est, par conséquent, son (leur) tour d'exécuter la coloration.

4.5.2. Condition de réussite

Lorsque un site est une feuille de l'arbre, tous ses voisins sont coloriés ; Lorsque toutes les feuilles sont coloriées, tout le graphe est colorié.

Comme lors de la construction de l'arbre, seul la racine va pouvoir constater la terminaison de la coloration sur tout le graphe.
Comme lors de la construction de l'arbre, il va suffire de faire remonter un avis de terminaison des feuilles vers la racine.
Un sommet va alors renvoyer cet avis à tous ses voisins dont il connaît la couleur (ceux situés plus haut dans l'arbre) lorsque les autres (situés plus bas) auront renvoyé un avis.

4.5.3. Condition d'échec

Bien évidemment, la problématique posée par un algorithme utilisant le BackTrack, réside précisément dans ce principe de retour en arrière.

Dans l'algorithme centralisé que nous connaissons, le BackTrack est nécessaire lorsqu'un sommet n'a plus de couleurs à sa disposition. On revient alors au sommet précédemment colorié, on lui attribue une nouvelle couleur puis on recommence la coloration du sommet ayant posé le problème. Le principe va être exactement le même ici.

Un sommet ne pouvant se colorer, par manque de couleurs disponibles, va informer son père (immédiatement précédent dans l'ordre de la coloration). Celui-ci va alors changer sa couleur, puis relancer la coloration sur tous ses voisins moins prioritaires.

Remarque : un sommet ayant plusieurs fils se coloriant en parallèle va devoir relancer la coloration sur tous ses sous arbres, même si un seul d'entre eux a échoué dans sa coloration. Ceci se justifie par la possibilité qu'un des fils ayant réussi sa coloration, ait choisi la même couleur que le second choix du père. Ce sommet devra alors changer sa couleur, et par extension, tous ses sous arbres devront faire de même.

On peut considérer l'échec total de la coloration lorsqu'une erreur remonte jusqu'à la racine. On a pu voir que, dans un algorithme de BackTrack, modifier la couleur de la racine et relancer une coloration ne modifiait pas l'issue de l'algorithme.

4.5.4. L'algorithme

```
Initialisation
      NbCouleurs ← k
      Ma Couleur ←  0 // couleur neutre
      CouleursVoisins[ NbVoisins ] ← 0   // i mémorise les couleurs des voisins
                                            plus prioritaires
      NbOkAttendus ← NbVoisins
      NbOkReçus ← 0
      Si Père = i
      Alors   Choisir Couleur
      Envoyer <COULEUR (MaCouleur) > à tous mes voisins

ChoisirCouleur
      Faire Ma couleur ← Ma Couleur + 1
      Tant que MaCouleur <= k et MaCouleur != CouleursVoisins[x] pour tout x
      Retourner MaCouleur

A la reception de <COULEUR (coul) > de j :
      CouleursVoisins[j] ← coul
      NbOkAttendus ← NbOkAttendus –1 //j est prioritaire sur moi
      Si Pere = j
      Alors    ChoisirCouleur
             Si MaCouleur < k
             Alors   Si il existe un voisin m tel que CouleursVoisins[m] = 0
               Alors  Pour tout voisin m tel que CouleursVoisins[m] =0
                      Envoyer <COULEUR(MaCouleur)>à m //m n'est pas colorié
                    Sinon      Pour tout voisin m tel que  CouleursVoisins[m] != 0
                      Envoyer <OK > à m //m est plus haut dans l'arbre
             Sinon   Envoyer <ERREUR> à Père
```

```
A la réception de <ERREUR> de j
  NbOkReçus ← NbOkAttendus
       Si Père != i
       Alors   ChoisirCouleur // on relance une coloration
             Si MaCouleur < k
             Alors   Pour tout voisin m tel que CouleursVoisins[m] = 0
                     Envoyer <COULEUR (MaCouleur) > à m //m n' est pas colorié
             Sinon  Envoyer <ERREUR> à Père
         Sinon // K-COLORATION RATEE

A la réception de <OK> de j
       NbOkReçus ← NbOkReçus + 1
       Si NbOkAttendus = NbOkReçus
             Si Père != i
             Alors  Pour tout voisin m tel que CouleursVoisins[m] != 0
                    Envoyer <OK > à m
             Sinon  // i est la racine
                    // K-COLORATION REUSSIE
```

4.6. Optimisation de l'algorithme

4.6.1. Pour ne pas construire une relation d'ordre inutilement…

Le principal inconvénient de cet algorithme provient du fait qu'il se déroule en deux phases.
En effet, la construction de l'arbre occupe un temps important dans le déroulement de l'algorithme (d'autant plus important que le graphe est fortement connexe).
Cependant, la coloration peut échouer au bout de quelques sommets. Dans un tel cas, la construction totale de l'arbre est inutile.

On constate, en lisant les deux algorithmes, qu'ils ont des fonctionnements similaires :

- On parcours le graphe de la racine aux feuilles, puis on remonte des messages de résultats
- Chaque sommet connaît ses voisins plus haut et plus bas que lui dans l'arbre

Notre idée consiste donc à réaliser simultanément les deux tâches.

Lorsqu'un sommet A est élu, il va choisir sa couleur et la diffuser à ses voisins non colorés. A la réception d'une couleur, les voisins de A non colorés vont déclencher l'élection. Le ou les sommets élus vont alors choisir leurs couleurs et vont la communiquer à leurs voisins non coloriés.

Ainsi, dans le meilleur des cas que serait une coloration réussie sans BackTrack, l'arbre ne serait parcouru qu'une seule fois.

Cependant, à chaque BackTrack sur un sous arbre, la racine de ce sous arbre, en changeant de couleur, va déclencher une élection parmi ses fils. Les critères d'élection et la topologie du réseau étant les mêmes que lors de l'envoi de la première couleur, le résultat de l'élection sera le même.

4.6.2. Pour ne pas refaire les élections en cas de BackTrack…

Il convient donc de trouver un moyen d'éviter refaire inutilement des élections en cas de BackTrack sur un sous arbre.

La solution à ce problème est simple :
A la réception d'un message de couleur, un sommet ne va déclencher l'élection que s'il n'a pas encore de père.

En revanche, s'il a déjà un père, cela signifie qu'il appartient à l'arbre de niveau. Si le sommet envoyant la couleur est son père, il va se colorer. Sinon, il va juste enregistrer la nouvelle couleur de ce voisin de plus haut niveau.

4.6.3. Pour Augmenter la distribution…

Il est évident que plus l'arbre de priorité sera large (on entend ici par largeur, le nombre moyen de sommet par niveau), et plus l'algorithme sera exécuté de manière distribuée. En cherchant donc à maximiser ce nombre moyen de sommets pour un même niveau, on va maximiser la distributivité de l'algorithme.
Nous allons donc faire en sorte qu'un sommet, quel qu'il soit, ait un maximum de fils.

Mais ce nombre de fils dépend directement de la topologie du graphe. La topologie résultant de la génération aléatoire du graphe, il est difficile de faire des hypothèses sur cette topologie.

Cependant, plus un sommet a de voisins candidats à être ses fils, plus le nombre de ses fils a des chances d'être élevé, et ce quelque soit la topologie. Par conséquent, notre idée consiste a modifier l'élection pour faire en sorte que le sommet élu soit celui qui a le plus de fils potentiels.

L'élection de départ va donc faire élire le sommet ayant le plus grand nombre de voisins (et, en cas d'égalité l'identifiant le plus grand). De ce fait, lorsqu'il enverra sa couleur à ses voisins, on aura statistiquement un nombre plus important d'élus. Par conséquent, le graphe va décomposer en un nombre plus grand de composantes connexes. La distributivité sera donc d'être plus importante.

4.7. Problèmes

4.7.1. Superposition d'élections

4.7.1.1. Le problème

Notre algorithme d'élection nécessite l'exécution d'une phase d'initialisation par tous les sommets participants (candidats ou non).
Ainsi, lors de la construction de l'arbre de priorités, les sommets doivent réinitialiser leurs variables (Nombre de voisins Max, Identifiant Max par exemple) pour procéder à l'élection du sommet le plus prioritaire du niveau.

Or, seul le sommet élu sait que l'élection est terminée. Lorsqu'il choisit sa couleur, qu'il la communique à ses voisins non coloriés, ceux ci vont déclencher une nouvelle élection dans la composante connexe.

Le problème, ici, est qu'un sommet qui ne fait que transmettre les messages d'élection entre les candidats peut participer successivement à des élections permettant d'élire des sommets de niveaux différents. Ce sommet ne peut donc savoir à quel moment une élection est terminée, ni à quel moment une nouvelle élection commence. Il ne peut donc pas savoir quand réinitialiser ses variables d'élection.

4.7.1.2. La solution

Nous avons tout simplement résolu ce problème en numérotant les élections.
Un sommet participe (non candidat) à une élection destiné à élire un sommet de niveau n dans l'arbre. S'il reçoit un message d'élection avec un numéro supérieur à n, il sait qu'il va devoir participer à une nouvelle élection, et sait donc qu'il doit réinitialiser ses variables.

4.7.2. Synchronisation des élections

4.7.2.1. Le problème

Comme il est courant en algorithmique distribuée, nous avons rencontré des problèmes dus à l'asynchronisme entre les différents processus. Certains messages se transmettent plus ou moins vite selon les liaisons, certaines actions mettent plus ou moins de temps à s'exécuter selon les processus.

Le seul problème de synchronisation que nous avons rencontré, se pose durant la phase d'élection.
En effet, durant cette phase, les sommets sont candidats à un élection si et seulement si ils ont reçu un message de couleur au préalable. Les autres sommets, non candidats, reçoivent des messages d'élection, mais, n'ayant pas reçu de couleur, ne font que diffuser le message d'élection.

Or, il est possible qu'un sommet devant recevoir la couleur, reçoive d'abord un message d'élection de la part d'un autre candidat du même niveau ayant déjà reçu le message de couleur.

On se rend bien compte de ce cas de conflit sur l'exemple suivant :

- *Une fois que le père a choisit sa couleur, il l'envoie à ses deux fils (1) et (2)*
- *Cependant, le Fils1 reçoit le message très vite, et se déclare candidat (3)*
- *Le problème vient du fit que, si le message de couleur (2) tarde à arriver, le Fils2 recevra (3) avant de recevoir (2), et croira qu'il n'est pas participant comme candidat à l'élection.*

Il est donc nécessaire de définir une politique pour gérer ces conflits.

4.7.2.2. Les solutions

Le sommet atteint trop tard passe son tour

Néanmoins, en prenant du recul par rapport à ce problème de synchronisation, on peut s'apercevoir que le fait qu'un sommet reçoive la couleur après un message d'élection et qu'il passe son tour, n'empêche pas l'algorithme de fonctionner.

Effectivement, le but de cette phase est de choisir un sommet pour chaque composante connexe du graphe ôté du sommet père.

Or, si un sommet A reçoit un message d'élection avant de recevoir la couleur du père, c'est qu'il y a, dans la même composante connexe, un sommet B qui est candidat. Par conséquent, A peut passer son tour en étant sûr qu'un sommet sera élu dans la composante connexe car il y a au moins un candidat à l'élection.

Cette solution est simple à mettre en œuvre, en revanche, on ne peut plus garantir que le sommet élu sera celui qui, parmi les fils, a le nombre de voisins non colorés le plus important. Mais cette exigence d'élire le sommet ayant le plus grand nombre de voisins non colorés n'est qu'une optimisation de l'algorithme.
Ainsi, dans le pire des cas, le sommet élu sera celui qui a le moins de voisins non colorés car tous les autres auront passé leur tour. De ce fait, selon les probabilités, l'algorithme sera moins distribué au niveau suivant. Mais ce n'est qu'une hypothèse statistique.

Toutefois, le message COULEUR en retard devra finir par arriver, que fait alors à ce moment là ?

Le père déclenche l'élection quand les fils sont prêts

La première solution envisagée consiste à ne déclencher l'élection parmi les fils que lorsque tous ont reçu la couleur. Cependant, ces fils ne se connaissent pas, il peut être compliqué pour chacun d'entre eux de connaître l'état des autres.

En revanche, ils sont tous, par définition, voisins du père. Le père peut donc assurer la synchronisation.

Chacun des fils, à la réception de la couleur, va renvoyer un accusé de réception à son père et va se tenir prêt pour l'élection. Lorsque le père aura reçu les accusés de tous ses voisins non coloriés, il leur intimera l'ordre de commencer l'élection. Ainsi, si un sommet reçoit un message d'élection avant l'ordre du père, comme il sera prêt pour l'élection il déclenchera son élection.

Toutefois, cette solution va augmenter le nombre de messages échangés. Aussi, nous allons voir qu'il est possible d'effectuer cette synchronisation sans rajouter de messages.

On simule la réception de la couleur

La deuxième solution proposée tire partie du fait que les sommets devant participer à l'élection sont tous voisins du père. Notre idée consiste à joindre l'identifiant et la couleur du père aux messages d'élection.

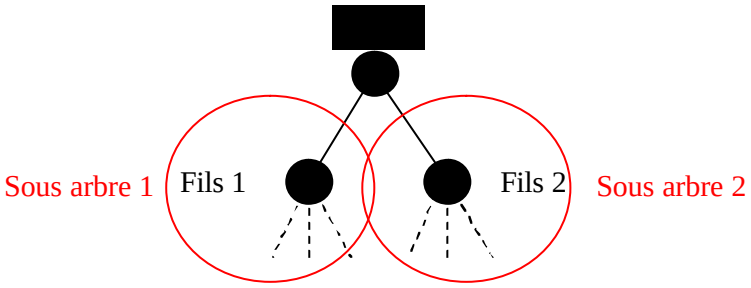
En effet, lorsqu'un sommet reçoit un message d'élection, s'il sait quel est le sommet qui, en envoyant sa couleur, a déclenché l'élection, il peut savoir s'il doit participer ou pas. S'il est voisin de ce sommet déclencheur, il va recevoir un message de couleur de ce sommet.

Il va alors simuler la réception du message couleur. Grâce à l'identifiant et la couleur, il va pouvoir enregistrer la couleur, et participer à l'élection en tant que candidat. Il n'aura plus ensuite qu'à ignorer le message de couleur arrivant en retard.

4.7.3. Vagues successives de coloration

4.7.3.1. Le problème

Pour comprendre ce problème, on servira d'un exemple. On considère le sous arbre suivant, issu de l'arbre de priorité :



Le père, ayant choisit sa couleur, l'envoie à ses deux fils qui répètent ce processus dans leurs sous arbre respectifs. On peut donc observer la propagation de la coloration de niveaux en niveaux.
Si la coloration dans l'un des sous arbre (1 par exemple) échoue, un message d'erreur parvient au père qui va changer de couleur et relancer la coloration de ses deux sous arbres.

Le problème vient ici du fait que, dans le sous arbre n'ayant pas renvoyé d'erreur, la 1ère coloration déclenchée peut être toujours en cours lorsque la 2ème est lancée. Ainsi, lorsque la 1ère coloration se termine, un message est renvoyé au père.
Que le résultat soit un succès ou un échec, on doit s'assurer que le père n'en tienne pas compte car ce résultat est obsolète : il répond à une coloration abandonnée.

4.7.3.2. La solution

Afin de résoudre ce problème, nous nous proposons de numéroter les vagues de colorations.
Nous allons ajouter des numéros au message de coloration : COULEUR, OK et ERREUR. Ainsi, un sommet va compter le nombre de colorations qu'il envoie à ses fils et le nombre de colorations que lui a envoyé son père.
S'il reçoit un message OK pour la dernière coloration qu'il a lancé, un sommet va renvoyer un message OK accompagné du numéro de la dernière coloration de son père.
En recevant un message OK ou ERREUR portant un numéro inférieur au nombre de colorations qu'il a envoyé, le sommet va l'ignorer. De cette manière, lorsque deux vagues, l'une descendante l'autre remontant vers le père OK ou ERREUR, se croisent, la vagues remontante va être stoppée.

4.8. L'algorithme final

```
Initialisation
      Père ← i
      PèreTemp ← i
      NiveauElection← 0
      IdMax ← i
      NbVoisMax ← nbVoisins
      Participant[nbSommet]←faux
      Route[nbSommet] ← i
      NbCouleurs ← k
      Ma Couleur ←  0 // couleur neutre
      CouleursVoisins[ NbVoisins ] ← 0  // i mémorise les couleurs des voisins plus
                                           prioritaires
      NbAcqAttendus[nbSommet] ← 0
      NbOkAttendus ← NbVoisins //nombre de fils potentiels
      NbOkReçus ← 0
      NbCoulReçues[ NbVoisins ]  ←0
      NbCoulEnvoyées ← 0
      MsgAIgnorer[NbVoisins] ← 0
      Faire Acte de Candidature(niveau) //tous les candidats font acte de  candidature
                                           au départ


Faire Acte de Candidature(niv)
     Participant[i] ← true
      niveau← niv
     NbAcqAttendus[nbSommet]← 0
      IdMax ← i
      NbVoisMax ← nbVoisins
      Si il existe un voisin m tel que CouleursVoisins[m] = 0
      Alors       Pour tout voisin m tel que CouleursVoisins[m] = 0
            NbAcqAttendus[m]← NbAcqAttendus[m]+1
            Envoyer <ELECTION(niveau, IdMax,NbVoisMax,0,0) > à m
      Sinon LancerColoration()

A la reception de <ELECTION(niv, id,nbVois, idPere, coulPere) >  de j :
     Si niveau<niv
      Alors       //on réinitialise
             NbAcqAttendus[nbSommet]← 0
             IdMax ← id
```

```
            NbVoisMax ← nbVois
            Participant[nbSommet]←faux
      Si Participant[id]
      Alors  Envoyer <ACQ(id)> à j
      Sinon  Participant[id]← true
            Si idPere est un de mes voisins && idPere !=0
            Alors  MsgAIgnorer[idPere] ← MsgAIgnorer[j]+1;
                  Simuler reception de <COULEUR (coulPere) > de idPere :
             Si (nbVoisMax<nbVois) ou ((nbVoisMax<nbVois)et(idMax<id))
            Alors  IdMax ← id
                  NbVoisMax ← nbVois
                  Si il existe un voisin m tel que CouleursVoisins[m] = 0
                  Alors       Route[id]=j
                        Pour tout voisin m tel que CouleursVoisins[m] = 0
                        Envoyer   <ELECTION(niveau,   IdMax,   NbVoisMax)   >   à   m
                        NbAcqAttendus[m]← NbAcqAttendus[m]+1
                  Sinon Envoyer <ACQ(id)> à j
```

A la reception de <ACQ(id) > de j :

```
      NbAcqAttendus[id]← NbAcqAttendus[id]-1
      Si NbAcqAttendus[id]==0
      Alors Si id==i
            Alors //Je suis élu
                  LancerColoration()
            Sinon Envoyer <ACQ(id)> à Route[id]
```

ChoisirCouleur

```
      NbCoulEnvoyées ← NbCoulEnvoyées+1
       Faire Ma couleur ← Ma Couleur + 1
       Tant que MaCouleur <= k et MaCouleur != CouleursVoisins[x] pour tout x
      Retourner MaCouleur
```

LancerColoration

```
      ChoisirCouleur
      Si MaCouleur < k
      Alors  Si il existe un voisin m tel que CouleursVoisins[m] = 0
            Alors       Pour tout voisin m tel que CouleursVoisins[m] = 0
                  Envoyer <COULEUR (MaCouleur) > à m
            Sinon  Pour tout voisin m tel que CouleursVoisins[m] != 0
                  Envoyer  <OK,  NbCoulReçue>  à  m  //m   est  plus  haut  que  moi  dans
                                                      l'arbre
      Sinon       Envoyer <ERREUR, NbCoulReçue > à Père
```

A la reception de <COULEUR (coul) > de j :

```
      Si MsgAIgnorer[j]==0
      Alors  CouleursVoisins[j] ← coul
            NbOkAttendus ← NbOkAttendus –1 //j est prioritaire sur moi
            NbCoulReçues[j] ← numcoloration
            Si Pere = j
            Alors LancerColoration()
            Sinon  Si Pere=i //i ne fait pas encore partie de l'arbre
                  Alors  PèreTemp ← j
                        NbAcqAttendus ← NbAcqAttendus - 1
                        Faire Acte de Candidature(niveau+1)
      Sinon MsgAIgnorer[j] ← MsgAIgnorer[j]-1
```

A la réception de <ERREUR(numColoration)> de j

```
      Si nbCoulEnvoyées == numColoration
      Alors   NbOkReçus ← NbOkAttendus
            Si Père != i
            Alors       Coloration()
            Sinon K-COLORATION RATEE
      Sinon //on ne fait rien le message concerne un coloration adandonnée
```

A la réception de <OK(numColoration)> de j

```
      Si nbCoulEnvoyées == numColoration
```

```
Alors NbOkReçus ← NbOkReçus + 1
       Si NbOkAttendus = NbOkReçus
       Alors  Si Père != i
                Alors       Pour tout voisin m tel que CouleursVoisins[m] != 0
              Envoyer <OK (nbCoulReçues[m])> à m
                Sinon       // i est la racine
                      K-COLORATION REUSSIE
       Sinon //on ne fait rien le message concerne une coloration abandonnée
```

5. Complexité des Algorithmes

5.1. L'élection Distribuée

5.1.1. Dans un graphe complet

Dans un graphe complet l'élection comporte quatre itérations :

a) Chaque sommet envoie sa candidature à tous les autres. Soit n x (n-1) messages.
b) En moyenne, 1 diffusion sur deux va être stoppée. Donc ½ (n x (n-1)) messages vont être échangés
c) A ce stade, tous les sommets ont déjà reçu les messages de la deuxième itération. Ils renvoient tous des acquittements. Donc ½ (n x (n-1)) messages vont être encore échangés
d) Enfin, tous les sommets ont reçu des acquittements de tous, ils envoient un acquittement au Max. Donc n-1 messages vont être échangés.

L'élection se réalise donc en 4 itérations, avec 2(n x (n-1)) + (n-1) messages. L'élection dans un graphe complet se réalise donc en $O(n^2)$ messages.

5.1.2. Dans un graphe quelconque

On considère un graphe quelconque, avec X ? [0,1], la probabilité de connexité.
Chaque sommet a donc (n-1)/X voisins.
La première itération de l'élection va donc comporter n x (n-1)/X messages.
On peut considérer qu'environ 1 diffusion sur 2 va être stoppée. La deuxième itération de l'algorithme va donc engendrer n x (n-1)/2X messages.
Ensuite, il est extrêmement difficile de prévoir le coût d'envoi de message de la troisième itération. En effet, certains sommets vont être atteints par des diffusions auxquelles ils participent déjà, et vont, en conséquence, envoyer des accusés de réception en retour. Les autres sommets vont faire suivre ou stopper la diffusion. La part de ceux qui vont envoyer des acquittements, de ceux qui vont faire suivre la diffusion et de ceux qui vont stopper la diffusion, est très complexe à calculer.
Faire une estimation de complexité du processus d'élection est donc très compliquée

5.2. La coloration Distribuée

Dans cette partie, nous tenterons de donner une approximation de la complexité de l'algorithme concernant la phase de coloration. Autrement dit, nous considérerons que l'arbre de priorité est construit. Or, il faut savoir que la construction de cet arbre nécessite autant d'élection qu'il y a de sommets dans le graphe. Chacune de ses élections comportant autant de diffusions qu'elles ont de candidats.

5.2.1. Dans un graphe complet

Dans un graphe complet, l'algorithme s'exécutera de manière séquentielle car tous les sommets sont voisins et ne pourront donc pas être de même niveau. L'arbre de priorité sera une chaîne de longueur n-1 arêtes.

Dans le meilleur des cas, sans BackTrack, on comptera n-1 messages COULEUR et n-1 messages OK. Soit 2n-2 messages au total.

Dans le pire des cas, le BackTrack a lieu sur la feuille de l'arbre (autrement dit que k = n-1). Remarquons que si k < n la k-coloration est impossible.

Dans un tel cas, la feuille va renvoyer ERREUR à son père. Son père qui a choisi la $k^{ième}$ couleur, ne peut en changer. Il va donc BackTracker. Son père, sommet de niveau n-2, a deux choix de couleur, il en change puis relance sur son fils, qui choisit la dernière couleur et en informe son fils. Celui-ci n'a plus de couleur disponible. Ce BackTrack a donné lieu à 2 niveaux x 2 messages supplémentaires. Mais il va falloir BackTracker à nouveau, et remonter puis redescendre 2 fois jusqu'au sommet de niveau n-3 (qui a 3 choix de couleur).
La somme des messages de BackTrack va donc être : 4 + 12 + …..+ (k-1) (2 x (n-2)) + (n-1). Les n-1 derniers messages remonte ERREUR jusqu'à la racine.
Soit au total 2(n-1) + ? niveau (k-niveau) (2 (n-1-niveau))

5.2.2. Dans un graphe quelconque

5.2.2.1. Un calcul très difficile

Tout comme pour le processus d'élection, estimer la complexité du processus de coloration, tant en nombre d'itérations, qu'en nombre de messages échangés, est très difficile.

En effet, un tel calcul devrait prendre en compte :

- Le nombre de sommets
- La probabilité de connexité
- Le nombre de couleurs
- Le nombre de BackTrack
- Le niveau d'où part chacun de ces BackTrack
- Le niveau jusqu'où remonte chacun de ces BackTrack

L'incertitude et la multiplicité de ces données représentent un trop grand nombre de suppositions pour permettre de donner une mesure de la complexité de l'algorithme avec une relative précision. Nous n'affirmons pas ici que ce calcul ne peut être fait, en revanche, nous voulons souligner le fait qu'un tel calcul relève d'opérations complexes.

5.2.2.2. Dans le meilleur des cas

Nous allons tout de même donner une estimation de complexité dans le cas où la coloration réussit sans aucun BackTrack. Pour cela nous considérons un graphe G à n sommets et dont l'arbre de priorité comporte en moyenne m arcs de la racine aux feuilles, et avec X la probabilité de connexité.

Dans un tel cas, qui est le meilleur des cas, la coloration réalise une descente en profondeur dans le graphe en explorant toutes les branches parallèlement, avant de remonter des messages OK jusqu'au père. L'algorithme se déroule donc en 2m itérations.

En moyenne, le nombre de sommet par niveau est n/m. A chaque itération, durant la descente, les n/m sommets d'un niveau envoient leurs couleurs à leurs voisins situés plus bas qu'eux dans le graphe. En moyenne, on peut considérer qu'environ 1 voisin sur 2 est situé en dessous (ou au dessus) dans l'arbre. Chacun des n/m sommets va donc envoyer son message à ½ de n/X ; c'est à dire n/2X messages. A chaque itération (sauf a la dernière, lorsqu'on est sur les feuilles), on compte en moyenne ce nombre de messages. La descente des messages couleurs représente donc m * n/2X messages.

La remontée des messages OK jusqu'au père représente le même nombre de messages. En effet, à chaque itération les n/m sommets d'un niveau envoient OK aux n/2X voisins situés plus haut qu'eux dans l'arbre. On a donc m * n/2X messages.

En moyenne, dans un graphe quelconque, sans BackTrack, la coloration se déroule en 2m itérations, avec (m * n)/X messages.

5.3. Mesure de la distributivité

Dans cette partie, pour simplifier le problème nous considérerons que tous les sommets ont le même temps de calcul. Sans cette simplification, l'estimation de la distributivité relève d'un problème d'ordonnancement NP-complet.

5.3.1. Dans un graphe complet

Dans un graphe complet, notre algorithme va construire un arbre de priorité qui aura la forme d'une chaîne. C'est le pire des cas : tous les sommets étant voisins, aucun ne pourront se colorier en même temps.
L'algorithme se déroulera donc de manière linéaire.

Dans un tel cas, la coloration ne sera pas plus distribuée que dans l'algorithme de coloration linéaire.

Nous pouvons noter également que les élections successives nécessaires à la construction de l'arbre vont ajouter un nombre important de messages. On peut donc considérer que, dans un graphe complet, notre algorithme sera moins performant.

5.3.2. Dans un arbre

Considérons que notre graphe est un arbre, binaire ou non.

Par définition de l'arbre, il n'existe qu'un seul chemin pour aller d'un sommet à un autre ; il n'existe pas de circuit. Par conséquent, il n'existe pas d'autre chemin entre deux voisins d'un sommet que le chemin qui passe par ce sommet. De ce fait, si ce sommet A envoie sa couleur à ses deux voisins, ils pourront alors se colorier en parallèle car dans le graphe ôté de A, les deux voisins ne peuvent se trouver sur la même composante connexe.

Dans un tel cas, l'algorithme va comporter deux fois plus d'itérations que la plus longue chaîne du graphe comporte d'arcs (deux fois car descente puis remontée de l'arbre). Si l'on note m ce nombre d'arcs, alors en moyenne, n/m sommets vont pouvoir travailler en parallèle.

Si le graphe est un arbre, notre algorithme sera donc obligatoirement distribué, et ce, quel que soit le sommet qui initie la coloration. C'est le meilleur des cas.

5.3.3. Dans un graphe quelconque

Il est très compliqué de généraliser l'estimation de la distributivité pour toutes les topologies. En effet, comme il a été expliqué précédemment, le trop grand nombre de paramètres rend le calcul de cette estimation compliqué.

Cependant, si l'on s'appuie sur les estimations du pire et du meilleur cas qui ont été faites plus haut, on peut conclure que la distributivité varie entre 1 et n/m sommets pouvant travailler parallèlement.

Plus généralement, on peut dire que l'algorithme sera distribué si au moins un de ses sous graphes comporte plusieurs composantes connexes.
Autrement dit, si le graphe n'est pas complet, au moins deux sommets du graphe ne seront pas voisins et pourront travailler ensemble.

Par conséquent, si la probabilité de connexité est inférieure à 1, nous sommes en mesure d'affirmer que notre algorithme s'exécutera de manière distribuée.

6. Architecture

6.1. Définition du graphe

6.1.1. Principe du graphe

Le but premier du projet est la coloration d'un réseau.
En vue de simplicité, on fera un amalgame entre la notion de processus et de processeur, entre le site physique de l'exécution et l'exécution elle-même.
Notre modèle d'exécution consiste en N processus qui communiquent entre eux par un échange de messages. C'est pourquoi, sur ce projet, chaque sommet du graphe représente un processus (thread en java) ; et la communication entre ces processus se fait par le biais de tubes.

6.1.2. Génération du graphe

A chaque liaison du graphe, est liée une probabilité d'existence qui représente la probabilité de connexité du graphe.
Une probabilité de connexité à 0 implique un graphe où tous les sommets sont isolés, sur notre interface, on saisit le pourcentage de connexité. Une probabilité à 1 implique un graphe complet où chaque sommet connaît tous les autres sommets du graphe.

6.2. Description des classes

6.2.1. Package graphe

Ce package contient tous les algorithmes.
Il s'y trouve la classe Sommet et Graphe pour définir la structure ; les classes d'élection, ElectionJeton, ElectionDistribuee ; et enfin les classes de la coloration ColorationJeton, ColorationDistribuee.

6.2.1.1. Classe Sommet

Le Sommet permet de communiquer avec les Sommets voisins du graphe. Il connaît tous ses voisins ainsi que le nombre de sommets total du graphe. Il permet aussi d'effectuer des statistiques sur le nombre de messages échangés ainsi que sur la durée de certains traitements.

6.2.1.2. Classe Graphe

Classe contenant toutes les informations du graphe à colorer, il est représenté par une matrice d'adjacence. C'est dans cette classe qu'apparaît la probabilité de connexité.

6.2.1.3. Classe ElectionJeton

Cette classe permet de gérer les élections dans le graphe sur le principe de la circulation de jeton. Le sommet chef qui est élu possède l'identifiant le plus grand.

6.2.1.4. Classe ElectionDistribuee

Cette classe permet de gérer les élections dans le graphe sur le principe de la diffusion des messages.
Le sommet qui est élu est celui qui a le plus de voisins. S'il y a égalité entre deux sommets, c'est le sommet qui a l'identifiant le plus grand.
Remarque : on a un chef par composante connexe.

6.2.1.5. Classe ColorationJeton

Cette classe permet de gérer une coloration du graphe de proche en proche, par circulation de jeton.

6.2.1.6. Classe ColorationDistribuee

Cette classe hérite de tous les attributs et méthodes de la classe Sommet, les élections en plus.

6.2.2. Package application

Ce package contient seulement les classes pour l'interface.

6.2.2.1. CadrePrincipal

Définit la fenêtre principale qui contient tous les objets graphiques : menu, champs de saisie, boutons, dessin du graphe ainsi que les commentaires.

6.2.2.2. CadreDessin

Définit le cadre où est dessiné le graphe.

6.2.2.3. SommetDessin

Définit un sommet graphique.

6.2.2.4. Ecouteur

L'écouteur est un Thread qui fait en permanence la correspondance entre un sommet graphique et la partie algorithmique correspondante. Quand, dans le déroulement de la coloration, un sommet change de couleur, il commande au sommet graphique d'en fait de même.

7. Conclusion

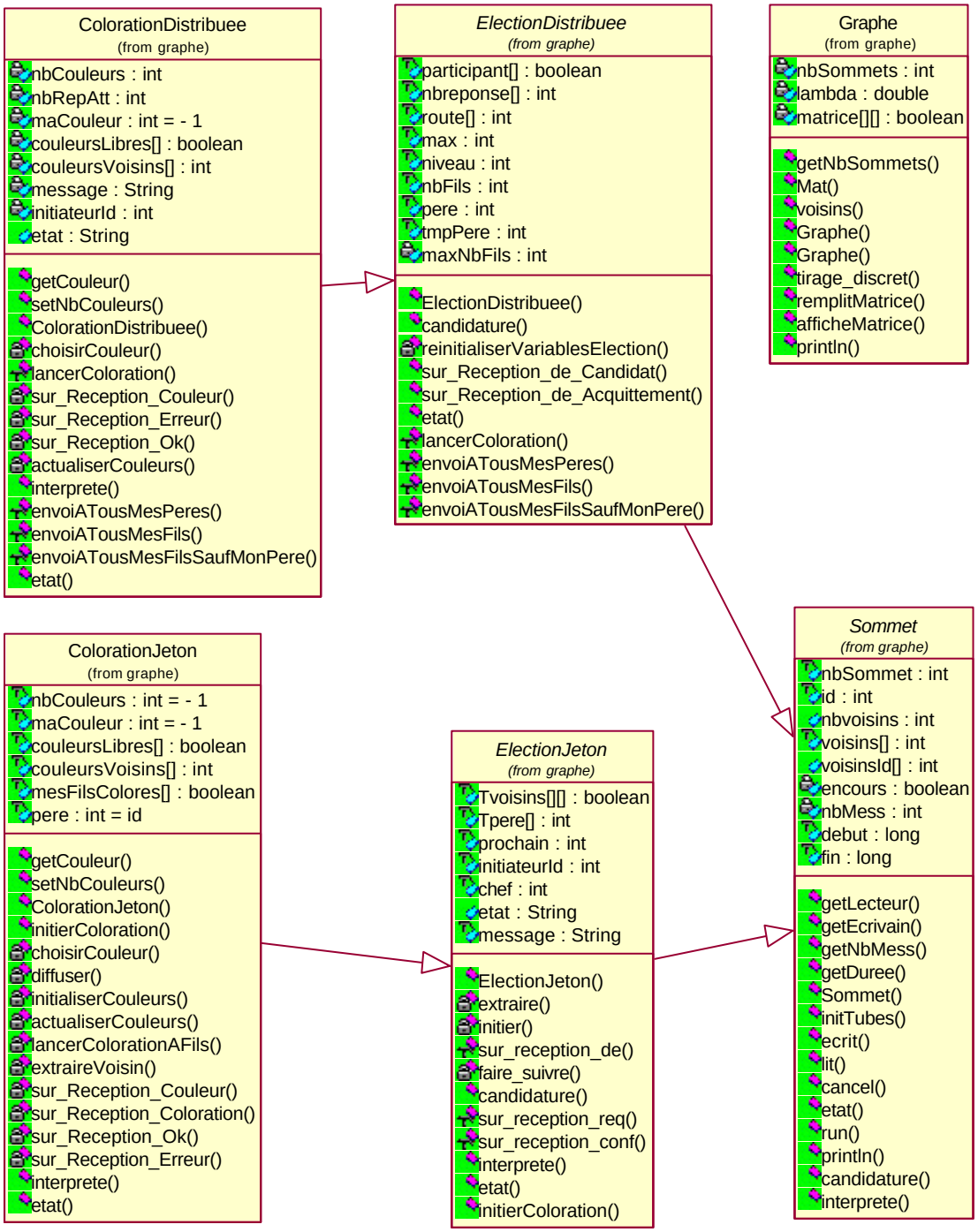
Du point de vue du résultat final, nous considérons avoir atteint les objectifs fixés par le cahier des charges initial.
Tout d'abord nous avons implémenté l'algorithme de BackTrack étudié en cours. Ceci nous a permis de cerner les réels problèmes d'un calcul distribué.
Nous appuyant sur ces acquis, nous avons imaginé puis implémenté un algorithme de BackTrack fonctionnant de manière distribuée ; algorithme que nous avons optimisé par la suite.
Enfin, nous avons réalisé une interface graphique afin d'illustrer de façon pédagogique le déroulement de notre algorithme.

Au travers de ce projet, nous avons pu avoir un aperçu de l'ampleur des problèmes spécifiques à l'algorithmique distribuée tels que la synchronisation de processus, le calcul de la complexité d'un algorithme, la localisation d'éventuels problèmes lors du déroulement du programme…..

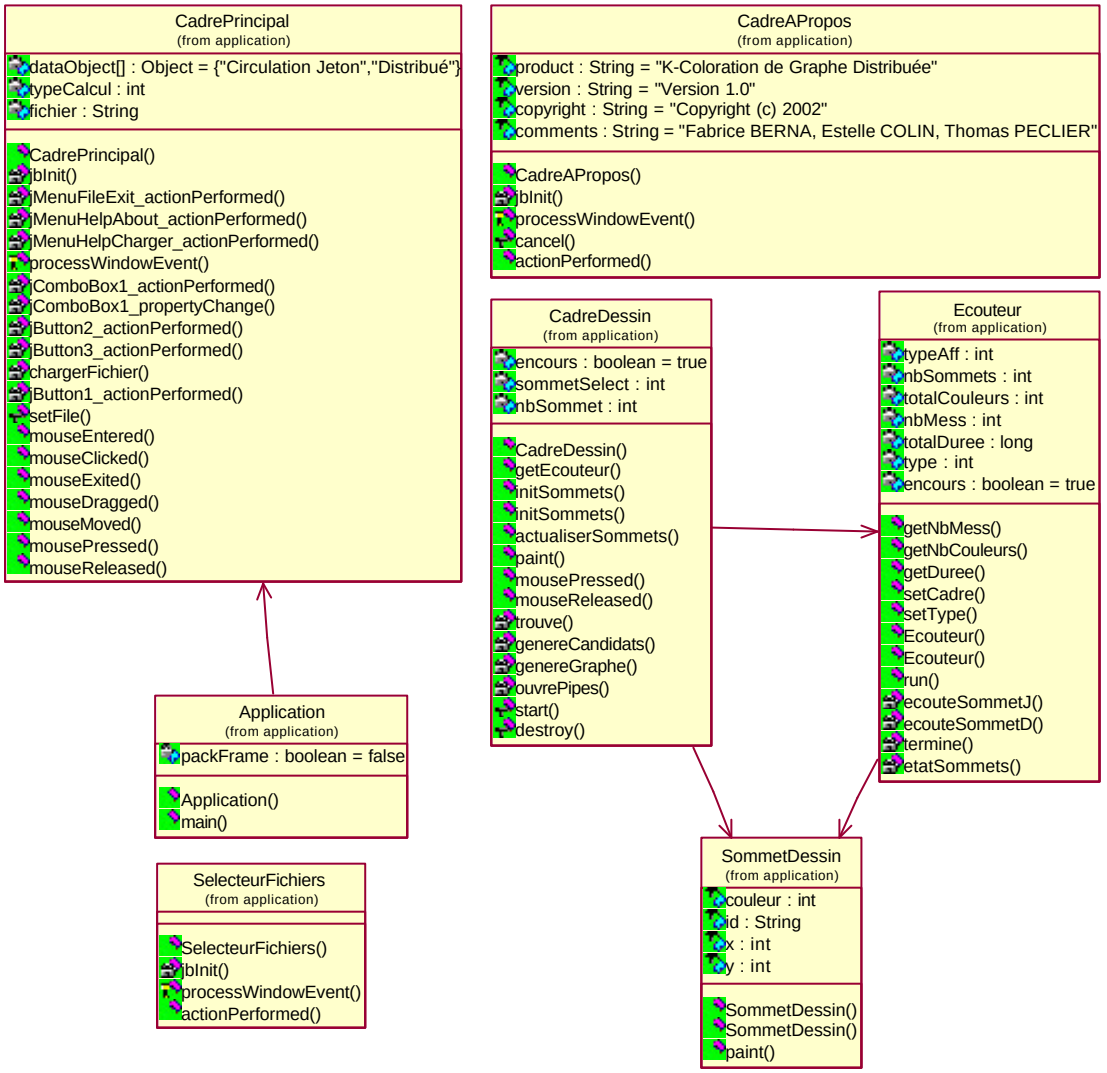
8. Annexes

8.1. Organisation des classes

8.1.1. Classes Coloration et Election



8.1.2. Interface d'utilisation



8.2. Interface d'utilisation

L'interface d'utilisation permet de voir une représentation du graphe de communication. Elle offre les fonctionnalités suivantes :

- Générer un graphe aléatoire en saisissant le nombre de sommets et la probabilité de connexité (en pourcent)
- Charger un graphe issu d'un fichier texte
- Lancer une coloration (distribuée ou à jeton) en saisissant le nombre de couleurs pour la k-coloration
- Afficher les statistiques de la k-coloration : Nombre de messages échangés, Distribution de la coloration, Nombre de couleurs utilisées, et Nombre de messages utilisés pour la coloration.
- Afficher la coloration de chaque sommet en temps réel.
- Afficher les étapes de la coloration