Développement Logiciel

Ordonnancement de communications

Dixneuf Julen

31 mars 2017

Table des matières

Chapitre 1

Présentation du problème

1.1 Contexte

Ce projet est consacré à l'ordonnancement de l'interaction d'éléments dans un graphe, ici présentés comme des serveurs. Ce problème peut naturellement se prolonger à d'autres exemples, tel que celui des poignées de main, mais on se contentera de traiter l'exemple des serveurs qui se synchronisent.

Considérons des serveurs, répartis sur la surface du globe. Ils sont interconnectés par un réseau de câbles, qui les relient de proche en proche. Après un certain temps de calcul, ils doivent synchroniser leurs résultats avec leurs voisins directs. Cependant, chaque serveur ne peut se synchroniser qu'avec un voisin à la fois, il faut donc qu'il réalise autant de synchronisations qu'il a de voisins, chacune prenant un temps non négligeable.

Le problème revient donc à optimiser ce temps de synchronisation. En effet, un algorithme naïf consistant à choisir le prochain voisin de manière aléatoire (ou selon un ordre prédéfini mais non optimisé) présente une grande inefficacité : certains serveurs sont prêts à communiquer avec un voisin, mais ces derniers sont tous occupés, ils attendent et retardent la terminaison de la synchronisation globale.

La solution consiste donc à optimiser cet ordre de sélection des voisins pour la synchronisation, afin de minimiser les temps où un serveur est inactif. Pour réaliser cette optimisation, il faut dégager des groupes de liens entre serveurs qui peuvent se synchroniser en même temps sans qu'il n'y ait conflit. Pour cela, il faudra utiliser un algorithme de coloration de graphe.

Une fois cet ordre optimisé établi, une phase de simulation sur des réseaux représentés par des graphes vient compléter le projet, pour pouvoir observer le gain de temps associé à l'ordonnancement. Cette simulation peut être illustrée en images grâce à une bibliothèque *Python* **Paraview**.

Il est alors possible de comparer les performances sur des algorithmes de coloration différents.

1.2 Formalisation

Ce réseau peut être représenté par un graphe, les nœuds du graphe étant des serveurs, et un lien entre deux serveurs étant représenté par une arête.

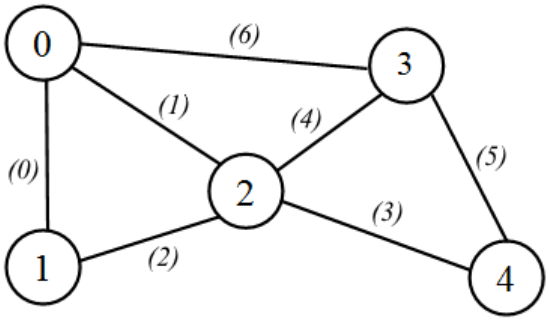

FIGURE 1.1 – Exemple de graphe simple

On peut formaliser un graphe G mathématiquement, en le représentant comme un tuple

$$G = (V, E)$$

où V est l'ensemble des *sommets* (*vertex* en anglais) et E est l'ensemble *arêtes* (*edge* en anglais). Chaque arête est un tuple $(v_1, v_2) \in V$ indiquant qu'une arête relie les sommets v_1 et v_2. Il existe plusieurs manières de représenter ce graphe en machine, qui seront détaillées plus loin dans le rapport.

Le problème posé ici demande de colorier les arêtes et non pas les sommets du graphe, or les algorithmes de coloration permettent de travailler sur les sommets. Il suffit alors de créer le graphe des arêtes, appelé *graphe adjoint* $L(G)$ (*line graph* en anglais).

$$L(G) = (E, V')$$

où E est toujours l'ensemble des arêtes, mais V' représente les sommets communs à deux arêtes. Ainsi V' est composé de tuples $(e_1, e_2) \in E$ indiquant que l'arête e_1 et l'arête e_2 ont un sommet en commun dans le graphe G.

1.3 Résolution

1.3.1 Coloration

Sans rentrer, dans les détails des différents algorithmes de coloration possibles qui seront détaillés plus loin dans le dossier, il est possible de proposer une coloration des arêtes pour l'exemple simple de graphe proposé. Pour cela, il suffit de regrouper entre elles les arêtes non voisines, facilement

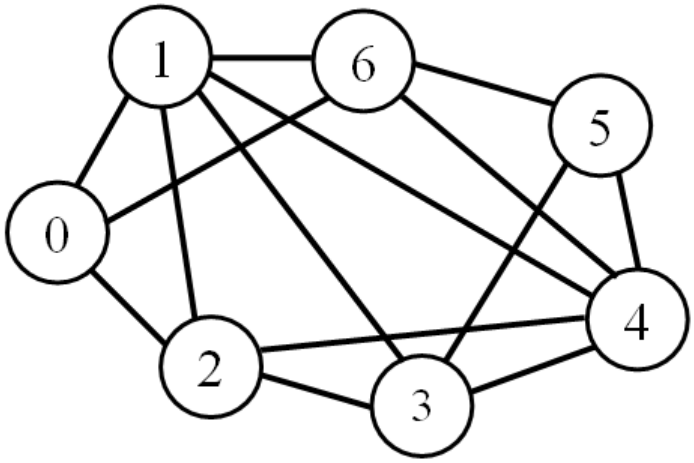

FIGURE 1.2 – Graphe adjoint du graphe de l'exemple précédent

visibles dans le graphe adjoint. Ainsi, en recherchant ces groupes par ordre croissant du nom de l'arête, on obtient les groupes :

$$A_0 = \{0, 3\}; A_1 = \{1, 5\}; A_2 = \{2, 6\}; A_3 = \{4\}$$

où chacune des arêtes propres à un groupe est isolée, sans aucun sommet en commun avec une autre arête du groupe.

1.3.2 Synchronisation

Sans coloration

Réalisons à la main l'algorithme de synchronisation. Sans coloration, les sommets initient la communication dès que c'est possible, selon leur ordre croissant de nom. Si le sommet en question n'est pas disponible, il attend. (On fait l'hypothèse pour simplifier que les arêtes ont toutes le même poids, et de plus les nœuds sont tous prêts à communiquer.) Sans coloration, on comprend que le sommet **2** cherche à communiquer avec le sommet **0**, qui est occupé avec le sommet **1**. Il devra attendre que cette première synchronisation soit terminée pour commencer à faire quelque chose, alors que le lien avec **3** était disponible. Cet exemple montre pourquoi sans coloration il y beaucoup de pertes de temps.

Avec coloration

Cette fois-ci, les sommets se synchronisent selon l'ordre fixé par les groupes de liens cités plus haut. Ainsi, la première synchronisation verra les synchronisations de **(0, 1)** et **(2, 3)**. Par rapport au fonctionnement

précédent, on a déjà gagné en vitesse car le sommet **2** est cette fois occupé dès le début des synchronisations.

1.4 Mesure de l'efficacité

Il existe de multiples moyens de colorier le graphe adjoint, certaine colorations étant plus efficaces pour notre problème. On cherche ainsi à minimiser le nombre de couleurs utilisées pour la coloration du graphe adjoint $L(G)$, la solution optimale étant appelée *indice chromatique* $\chi'(L(G))$. On cherchera donc à se rapprocher de l'*indice chromatique*, sachant que pour un graphe G quelconque,

$$\chi'(G) \in [\Delta(G), \Delta(G) + 1]$$

où $\Delta(G)$ est le degré maximum de G (nombre maximal d'arêtes que possède un sommet).

Chapitre 2

Différents formats et structures de données

2.1 Représentation des graphes

Dans la grande majorité des graphes, de nombreux liens possibles ne sont pas effectués, c'est à dire que la plupart des sommets n'ont que quelques voisins. Si l'on représente le graphe sous la forme d'une matrice d'adjacence $[G]$ tel que

$$G_{i,j} = \begin{cases} 1 & \text{si il existe une arête reliant i et j} \\ 0 & \text{sinon} \end{cases}$$

celle-ci sera alors constituée de nombreux 0 et de quelques 1 éparses. Elle est de plus symétrique, l'information est donc redondante. Au lieu de lister si chaque couple de sommets est voisin, il suffit de donner la liste des voisins de chaque sommet, c'est la *liste d'adjacence*.

Pour représenter cette dernière en machine, on utilise un format de données spécial, le *CSR* (Compressed Sparse Row), qui utilise deux tableaux, un de données, et l'autre de pointeurs.

2.1.1 Principe général

Expliquons sur un exemple très général... Un système est composé de $n+1$ éléments $a_0, a_1, ..., a_n$ qui peuvent avoir différents attributs $x_0, x_1, ..., x_k$. Sous forme d'un stockage CSR, on va stocker dans le premier tableau les différents attributs de a_0, puis de a_1, et ainsi de suite. Ainsi si a_0 a pour attributs x_2 et x_5 et a_1 a x_6, le début du tableau ressemblera à

x_2	x_5	x_6	...

Nous voyons bien les deux attributs de a_0, et celui de a_1, mais avec ce seul tableau un lecteur ne peut savoir si ce sont 3 attributs de a_0, aucun, ou

autre. Il faut rajouter une information pour savoir à quel élément correspond chaque attribut : on utilise un tableau de pointeurs, qui va permettre d'indiquer que les deux premiers attributs sont associés à a_0, puis le suivant à a_1 et ainsi de suite. Il se présente alors ainsi :

0	2	3	...	N

Pour l'élément a_0, ses attributs sont listés de la case 0 à la case 2 exclue (en commençant à lister à partir de 0). De même, pour l'élément a_1, ses attributs vont de la case 2 à la case 3 exclue. De manière plus générale, les attributs de l'élément a_i sont listés de la case l à la case m, où l et m sont respectivement les valeurs des cases i et $i+1$ du tableau *pointeur*. La dernière case indique le nombre total d'attributs dans le premier tableau.

On va ainsi nommer le premier tableau *a2x* (élement vers attributs) et le deuxième *p_a2x* (p comme pointeur). Pour résumer, en utilisant les mêmes notations qu'en *Python* :

Les attributs de l'élément a_i sont $a2x$[p_a2x**[i]** : p_a2x**[i+1]**]

2.1.2 Application aux graphes

Quel rapport avec la représentation en machine d'un graphe et notre matrice d'adjacence ? Il suffit de prendre comme éléments les sommets ou les arêtes, et comme attribut d'être voisin d'un sommet ou d'une arête. Cela donne plusieurs représentations possibles d'un graphe. On travaillera à nouveau sur l'exemple simple introduit auparavant.

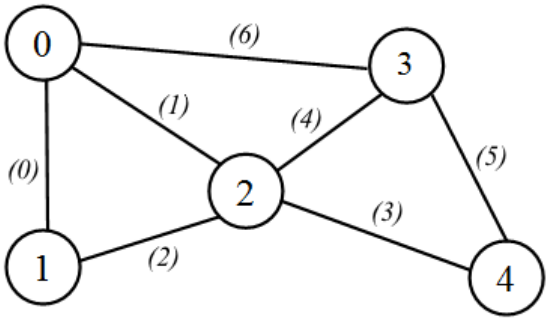

FIGURE 2.1 – Exemple de graphe simple

Sommets vers sommets

Ici on crée *vert2vert* et *p_vert2vert* (*vert* comme *vertex*, sommet en anglais). Ici, chaque sommet (élément) a comme voisin plusieurs autres sommets (attributs).

vert2vert :

1	2	3	0	2	0	1	3	4	0	2	4	2	3

p_vert2vert :

0	3	5	9	12	14

Dans les cases, 0 à 3 exclu, on retrouve les sommets voisins du sommet 0 : 1, 2 et 3. Et ainsi de suite...

Sommets vers arêtes

Ici on crée *vert2edge* et *p_vert2edge* (*edge*, arête en anglais). Ici, chaque sommet (élément) possède plusieurs arêtes (attributs).

vert2edge :

0	1	6	0	2	1	2	3	4	...

p_vert2edge :

0	3	5	9	...

Dans les cases 0 à 3 exclu, on retrouve les arêtes du sommet 0 : 0, 1 et 6. Et ainsi de suite...

Arêtes vers sommets

Ici on crée *edge2vert* et *p_edge2vert* . Ici, chaque arête (élément) est connectée à 2 sommets (attributs).

edge2vert :

0	1	0	2	1	2	2	4	2	3	3	4	0	3

p_edge2vert :

0	2	4	6	8	10	12	14

Dans les cases 0 à 2 exclu, on retrouve les sommets de l'arête 0 : 0 et 1. Remarquez la forme très simple du tableau *pointeur* étant donné qu'une arête a exactement deux sommets.

Arêtes vers arêtes

Ici on crée *edge2edge* et *p_edge2edge* . Ici, chaque arête (élément) a un sommet en commun avec d'autres arêtes (attributs).

edge2edge :

1	2	6	0	2	3	4	6	...

p_edge2edge :

0	3	8	...

Dans les cases 0 à 3 exclu, on trouve les arêtes voisines de l'arête 0.

2.2 Stockage de données

Pour stocker les données avant et après leur passage dans des algorithmes, il faut les stocker sur des fichiers. Ces derniers doivent pouvoir être facilement interprétéss et lu par un algorithme.

2.2.1 Stockage des graphes

Ici, un exemple de stockage pour les graphes est le format de fichier DIMACS. Sa syntaxe est relativement simple :

```
c ligne de commentaires
c
p edge nb_vertex nb_edge
e 2 1
e 3 2
...
```

La ligne commençant par un `p` indique la représentation du problème, ici on donnera le graphe en listant toute ses arêtes comme indiqué par `edge`, puis on précise le nombre de sommets et d'arêtes. Enfin, chaque ligne commençant par un `e` indique donc la présence d'une arête entre les deux sommets listés à la suite. Connaissant cette syntaxe, il est facile d'importer ce fichier dans un programme python.

2.2.2 Maillage et stockage

Lien maillage et graphe

La visualisation nécessite Paraview, qui travaille à l'aide de **Mesh** (*maillage* en anglais). Ce format de données va plus loin que les graphes, il sert usuellement à réprésenter de manière discrète des surfaces dans l'espace pour du calcul par éléments finis. Un maillage est donc constitué de polygones qui sont les *elements* (éléments en anglais), et chaque polygone est formé en reliant ses points adjacents, formant alors un contour fermé. Ces points sont les *nodes* (nœuds en anglais), et sont repérés par des coordonnées dans l'espace. On peut déjà remarquer que la représentation en machine sous forme CSR est encore une fois possible, et se comprend immédiatement. On peut ainsi définir `element2node`, `node2node`, etc..

De plus, si les éléments (des polygones), n'ont que deux sommets, ce sont alors des arêtes, et les deux nœuds l'encadrant ses sommets. On voit alors que dans le cas de maillage où tout les éléments n'ont que deux nœuds, les notions de *nœud* et de *sommet* ainsi que *élément* et *arête* coïncident respectivement. De même les représentations `element2node` et `edge2vert` sont alors identiques (et de même pour les autres variantes).

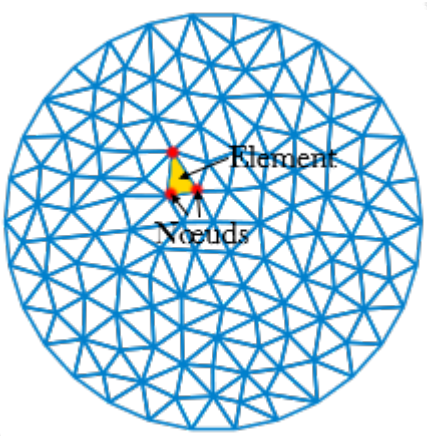

FIGURE 2.2 – Exemple de maillage

Format de données VTK

Pour stocker les maillages, Paraview utilise le format de fichier VTK, qui est écrit comme ceci :

```
# vtk DataFile Version 3.0
Titre
ASCII | BINARY
DATASET type
...
```

Ces quatre premières lignes indiquent quel type de données et comment elles sont stockées dans le fichier. La première indique la version de VTK utilisée, la deuxième est le Titre, la troisième vaut soit `ASCII` soit `BINARY` selon l'encodage. Enfin la quatrième indique comment est décrit le maillage, ici le type sera `UNSTRUCTURED_GRID`.

```
POINTS nb_nodes data_type
x_0 y_0 z_0
...
CELLS nb_elements array_size
nb_nodes_0 p_{0, 0} p_{0, 1} ...
...
CELL_TYPES nb_elements
3
...
POINT_DATA nb_nodes
...
CELL_DATA nb_elements
...
```

On commence d'abord par lister les coordonnées des nœuds, en précisant le nombre de nœuds et le type de données (`data_type` : *int, float, etc...*).

Ensuite vient la liste des éléments, avec le nombre d'éléments total précisé en amont. Chaque élément est représenté par un nombre de nœuds qu'il possède, et leur identifiant.

Le type de l'élément ensuite indiqué correspond à sa forme, ici 3 correspond à une arête.

Enfin il est possible d'ajouter plus d'information sur les éléments et les nœuds grâce à `POINT_DATA` et `CELL_DATA`.

Connaissant cette syntaxe, il est alors aisé de lire ce fichier avec un programme Python.

Chapitre 3

Fonctionnement de l'algorithme

3.1 Approche générale

La fonction de l'algorithme est de lire un fichier contenant un graphe ou un maillage, réaliser une coloration des arêtes, puis réordonner la liste d'adjacence des sommets en fonction. Un autre algorithme va alors simuler un comportement de ce réseau en fonction de la coloration appliquée, et enfin donner en sortie une représentation graphique de la simulation, grâce à des modules du logiciel *Paraview*.

3.1.1 Programmation objet

Python est un langage de programmation orientée objet, c'est pourquoi nous utiliserons cette approche pour ce projet. Ainsi, l'algorithme est découpé en modules (les objets) qui sont détaillés plus loin. Mais tout d'abord, il s'agit d'expliquer la programmation objet. En programmation, un objet est un conteneur, qui contient des attributs et des méthodes. Les attributs sont des variables propres à l'objet, pouvant être de n'importe quel type : entier, chaîne de caractères, liste ou même un autre objet. Les méthodes sont des fonctions, propres à l'objet. On commence d'abord par définir une classe, c'est à dire une maquette générale pour un objet. Par exemple, définissons la classe `Voiture`.

```
1  class Voiture(object):
       def __init__(self, couleur = ""):
           self.NB_ROUES = 4
           self.couleur = couleur
5
       def klaxon(self):
           print("Honk ⌴ !")
```

A partir de cette classe, ce modèle, on peut créer plusieurs instances de cette classe, des objets. Par exemple on peut créer `Monospace` de la manière suivante `Monospace = Voiture()`. Lors de la création de l'objet, la méthode (c'est une fonction) `__init__` crée les variables stockant la couleur et le nombre de roues de la voiture (attributs). Notez que pour dans toutes les classes créées pour ce projet, les attributs qui ne devraient pas dépendre de l'instance `NB_ROUES` (si toutes les voitures ont 4 roues) seront notés en majuscules tandis que les attributs non invariants (la couleur) seront écrits en minuscules. Notez ici que sans couleur spécifiée, l'objet n'a aucune couleur attribuée, mais on aurait pu lui en attribuer dès la création à en le passant en argument lors de l'appel de la classe. Pour appeler ou modifier l'attribut `couleur`, il suffit d'écrire `Monospace.couleur` et `Monospace.klaxon()` pour utiliser la méthode écrite dans la classe.

3.1.2 Application aux graphes et maillages

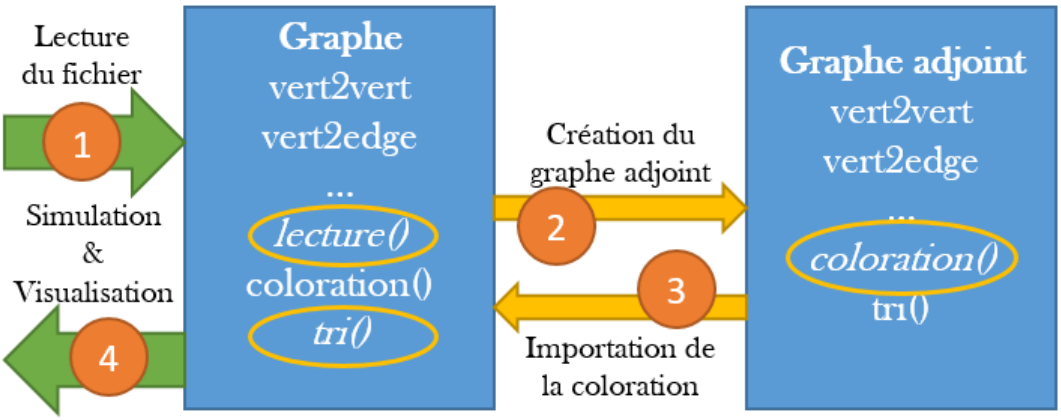

FIGURE 3.1 – Utilisation de la programmation objet pour les graphes

Les fichiers nous donnent des graphes, et on veut effectuer la coloration sur les arêtes. Au lieu de créer un algorithme spécial pour colorier les arêtes, on peut se contenter de créer un deuxième objet de la classe qui représentera le graphe adjoint, et dans lequel on appellera la méthode de coloriage des graphes.

La programmation objet a ainsi pour avantage de nous proposer de ne coder qu'une fois une classe adaptée, puis de créer plusieurs objets dont les attributs et l'usage pourront différer. Dans notre cas, la classe **Graphe** devra gérer la lecture de fichiers, la création de différentes représentations de graphe (vert2vert, edge2vert, etc...), la coloration des sommets et le réordonnancement des sommets en fonction de cette coloration. Cet objet pourra

ensuite être traité par deux autres objets, un dédié à la simulation et l'autre à la visualisation.

3.2 Details des différentes classes

3.2.1 mesh.py

Cette classe permet de lire les fichiers VTK afin de les représenter à l'aide de `elem2node`. Pour ce faire, il utilise les bibliothèques classiques de Python de lecture de fichier, puis lit ligne par ligne en se servant de la syntaxe identifiée plus haut pour interpréter les données. Ci dessous un code qui ouvre un fichier dont l'adresse (string) est dans la variable `address` .

```
file = open(address)
line = file.readline()
line = line.split()
if line[0] == "POINTS":
        ...
```

On peut alors lire une nouvelle ligne avec `readline`. Pour simplifier l'utilisation, on peut alors séparer et stocker chaque mot de la ligne séparé par un espace avec la méthode `split` sur les chaînes de caractères.

Cette méthode n'est pas tout à fait celle utilisée dans le fichier, mais donne les mêmes résultats. On ne détaillera pas plus ce fichier qui même s'il est long, n'est pas conceptuellement compliqué.

3.2.2 graph.py

Ce fichier constitue le gros du projet, il s'agit de détailler bloc par bloc les différentes méthodes programmées.

Variables

numb_vert : nombre de sommets ; *int*

numb_edge : nombre d'arêtes ; *int*

edge2vert : représentation par défaut du graphe (CSR) ; *array*

p_edge2vert : pointeur associé ; *array*

edge2edge ; *array*

p_edge2edge ; *array*

vert2edge ; *array*

p_vert2edge ; *array*

vert2vert ; *array*

p_vert2vert ; *array*

vert_degree : degré ou nombre de voisins pour chaque sommet ; *array*

numb_color : nombre de couleurs utilisées pour colorier le graphe ; *int*

vert_color : couleur de chaque sommet ; *array*

edge_color : couleur de chaque arête ; *array*

ReadFromFileDimacs() et ReadFromFileDimacsAscii()

Ce sont des méthodes simples pour lire les fichiers DIMACS décrits précédemment. Comme pour la lecture des fichiers VTK dans la classe **Mesh**, cette méthode se contente de lire par ligne par ligne le fichier texte, en se servant de la syntaxe pour interpréter le contenu du fichier. A noter que cette méthode donne une représentation du graphe de la forme `edge2vert`, celle disponible dans le fichier DIMACS.

BuildEdge2Edge()

Cette méthode permet de construire la représentation CSR `edge2edge` à partir de `edge2vert`. Pour ce faire, nous allons utiliser la représentation matricielle du graphe, tout en conservant un format CSR grâce au module `scipy.sparse.csr_matrix`, qui permet du calcul matriciel à partir des listes d'adjacence.

On va ainsi représenter à partir de `edge2vert`, la matrice d'incidence I^G et sa transposée qui sont définies de la sorte (V comme vertex, E comme edge pour le graphe G) :

$$\forall i \in V, j \in E, I^G_{i,j} = \begin{cases} 1 & \text{si l'arête j a pour sommet i} \\ 0 & \text{sinon} \end{cases}$$

Pour construire cette matrice référons-nous à la documentation de `scipy.sparse.csr_matrix`.

```
1 csr_matrix((data, indices, indptr), [shape=(M, N)])
  is the standard CSR representation where the column indices for
      row i are stored in indices[indptr[i]:indptr[i+1]] and their
      corresponding values are stored in data[indptr[i]:indptr[i
      +1]]. If the shape parameter is not supplied, the matrix
      dimensions are inferred from the index arrays.
```

Ici, `data` correspond aux valeurs de la matrice, ici que des 1. On crée donc un array de deux fois le nombre d'arêtes (nombre de 1 au total dans la matrice) à placer dans `data` à l'aide de `(numpy.ones(self.numb_edge*2)`. Ensuite, on remplace naturellement `indices` par `edge2vert` et `indptr` par `p_edge2vert`.

On obtient alors la transposée de la matrice d'incidence, et I^G s'obtient grâce à la méthode de transposition `transpose()`.

```
1 incidence_matrix_T = scipy.sparse.csr_matrix((numpy.ones(self.
      numb_edge*2), self.edge2vert, self.p_edge2vert))
  incidence_matrix = incidence_matrix_T.transpose()
```

Remarquons ensuite que la multiplication matricielle ${}^tI^G \times I^G$ permet de remonter à la représentation matricielle de `edge2edge`. En effet,

$$[{}^tI^G \times I^G]_{i,j} = \begin{cases} 2 & \text{si i = j} \\ 1 & \text{si les arêtes on un sommet en commun0} \quad \text{sinon} \end{cases}$$

Lorsque le calcul est effectué dans ce sens, on itère sur les arêtes, et le résultat en (i, j) donne le nombre de sommets en commun entre l'arête i et j. Si l'on conserve le format `scipy.sparse.csr_matrix`, on peut alors récupérer les attributs correspondant au format CSR une fois la diagonale retranchée.

```
identity_matrix = scipy.sparse.identity(self.numb_edge)
edge_incidence_matrix = (incidence_matrix_T.dot(incidence_matrix)
    - 2 * identity_matrix)
self.edge2edge = edge_incidence_matrix.indices
self.p_edge2edge = edge_incidence_matrix.indptr
```

On a ainsi récupéré `edge2edge` à l'aide de `edge2vert`

BuildVertDegree()

Cette méthode permet de construire le degré des sommets à partir de différentes représentations CSR du graphe.

Avec vert2vert C'est alors très simple, il suffit de regarder `p_vert2vert`, où par définition on retrouve entre la case i et $i + 1$ le nombre de sommets voisins au sommet i.

```
for i in range(self.numb_vert):
            self.vert_degree[i] = (self.p_vert2vert[i+1] - self.
                p_vert2vert[i])
```

Avec vert2edge On retrouve le même principe, car un sommet a autant d'arêtes partant de lui que de voisins puisqu'il y a un sommet au bout de chaque arête.

```
for i in range(self.numb_vert):
            self.vert_degree[i] = (self.p_vert2edge[i+1] - self.
                p_vert2edge[i])
```

Avec edge2vert Cette fois-ci, il faut aller chercher dans la partie donnée directement, en itérant sur `vert2edge`, chaque sommet est représenté autant de fois qu'il a d'arêtes, et donc de sommets voisins.

```
for x in self.edge2vert:
            self.vert_degree[x] += 1
```

ColorVertByWelshPowell()

On arrive enfin à l'algorithme de coloration de Welsh Powell des sommets du graphe.

Tout d'abord, on va ordonner les sommets par ordre de degré décroissant. Ensuite, on colorie avec la première couleur le sommet de degré maximal, puis on colorie tous les sommets qui ne sont pas déjà coloriés ni voisins avec un sommet colorié de la couleur actuelle, en les parcourant par degré décroissant. Lorsque plus aucun coloriage n'est possible, on incrémente alors la couleur et on reprend avec tous les sommets non coloriés, par ordre décroissant (sans reprendre le tri).

Pour le tri des sommets, et la future coloration, j'ai décidé d'utiliser une Queue, afin d'être sur de parcourir tous les sommets non coloriés, tout en évitant de changer l'ordre de tri. La partie tri est tout à fait naïve et non optimisée (en $O(n^2)$) :

```
indice_sorted = Queue.Queue(self.numb_vert)
    vert_sorted = numpy.zeros(self.numb_vert) # 1 if the vert i is
         already sorted
    for i in range(self.numb_vert): #sort one vertex each run
        degree_max = 0
        indice_deg_max = -1
        for j in range(self.numb_vert): #check all the vertex
            if vert_sorted[j] == 0:
                if self.vert_degree[j] > degree_max:
                    degree_max = self.vert_degree[j]
                    indice_deg_max = j
        indice_sorted.put(indice_deg_max) #add the max degree
            vertex to the queue
        vert_sorted[indice_deg_max] = 1
```

Pour la coloration, on choisit une couleur, puis on colorie tous les sommets qui le peuvent. Avec la structure de Queue, si l'on parcourt toute la file, et qu'on replace seulement les sommets non coloriés, on ne se retrouve qu'avec les sommets non coloriés, toujours triés.

```
while not (indice_sorted.empty()): #run until all vertex colored

        # color the highest vertex
        self.vert_color[indice_sorted.get()] = self.numb_color

        numb_vert_uncolored = indice_sorted.qsize()
        # color other vertex if there is no conflicts, by highest
            degree
        for i in range(numb_vert_uncolored):

            # extracting the vertex
            vert = indice_sorted.get()

            # checking conflicts
            no_conflicts = True
```

```
            j = 0
            while ((j < self.vert_degree[vert]) and (no_conflicts)
                ):
                # for all neigbour vertex, check if they have the
                    same color

                if (self.vert_color[self.vert2vert[self.
                    p_vert2vert[vert] + j]] == self.numb_color  ):
                        no_conflicts = False
                j+= 1

            if no_conflicts:
                # color the vertex
                self.vert_color[vert] = self.numb_color

            else:
                # put back the vertex in the queue, which stay
                    sorted
                indice_sorted.put(vert)

        # select a new color (also works as a counter)
        self.numb_color += 1
```

BuildVert2Edge

C'est ici très simple lorsqu'on comprend comment on a construit `edge2edge` avec la représentation matricielle. Là où `scipy.sparse.csr_matrix` nous donnait à partir de `edge2vert` la transposée de la matrice d'incidence, la matrice d'incidence correspond à la matrice d'incidence elle-même. Il suffit alors de passer en représentation matricielle, de faire la transposée puis de repasser en représentation CSR en prenant les bons attributs.

```
# -- build transpose of incidence matrix

    incidence_matrix_T = scipy.sparse.csr_matrix((numpy.ones(self.
        numb_edge*2), self.edge2vert, self.p_edge2vert))

    # numpy.ones generates "1" for incidence matrix, two for each
        edges

    incidence_matrix = scipy.sparse.csr_matrix(incidence_matrix_T.
        transpose())

    # -- extract vert2vert and p_vert2vert

    self.vert2edge = incidence_matrix.indices
    self.p_vert2edge = incidence_matrix.indptr
```

SortVert2EdgeByColor()

Il s'agit de trier la liste des arêtes de chaque sommet en fonction de la coloration `edge_color` des arêtes. Pour cette fois, j'ai pris avantage d'une fonction de tri implémentée dans Python `sorted`, qui prend en argument un tableau, et une fonction donnant un poids pour chacun des éléments de ce tableau. Le tableau sera alors trié en fonction du poids associée à chaque élément. Ici la fonction est définie à la volée avec lambda, et renvoie la couleur associé à chaque arête.

```
for i in range(self.numb_vert):
        ind_start, ind_end = self.p_vert2edge[i], self.p_vert2edge
            [i+1]
        # sort his edges, using edge_color as key
        self.vert2edge[ind_start : ind_end] = sorted(self.
            vert2edge[ind_start:ind_end], key=lambda j: self.
            edge_color[j])
```

On trie ainsi chaque liste d'arêtes d'un sommet en fonction de la couleur.

BuildVert2Vert()

Enfin, on veut pouvoir construire `vert2vert`, mais en tirant profit d'une éventuelle coloration des arêtes et d'un tri de `vert2edge`. Pour cela il faut donc utiliser `vert2edge` comme brique fondamentale de construction. On rappelle que `scipy.sparse.csr_matrix` permet d'avoir la matrice d'incidence I^G.

Dans ce cas, on mettra à profit la multiplication matricielle $I^G \times {}^tI^G$, qui donne

$$[I^G \times {}^tI^G]_{i,j} = \begin{cases} \text{degré du sommet i} & \text{si i = j} \\ 1 & \text{si les sommets sont voisins} \\ 0 & \text{sinon} \end{cases}$$

En effet, cette multiplication matricielle nous fait parcourir les sommets, et nous donne en (i, j) le nombre d'arêtes reliant les sommets i et j. Or il y a en soit 1, soit 0. Le cas $i = j$ est lui un peu spécial, et nous donne le nombre d'arête partant du sommet, soit le degré.

En retranchant la diagonale (en construisant une matrice dont les coefficients diagonaux sont les degrés des sommets), on retombe sur la représentation matricielle de `vert2vert`, on peut récupérer sa représentation CSR avec les attributs.

```
    incidence_matrix = scipy.sparse.csr_matrix((numpy.ones(self.
        numb_edge*2), self.vert2edge,   self.p_vert2edge))
    # -- build incidence matrix
    incidence_matrix_T = incidence_matrix.transpose()

    # -- build diagonal vertex degree matrix
    row = list(range(self.numb_vert))
```

```
    col = row
    vert_degree_diag_matrix = scipy.sparse.csr_matrix((self.
        vert_degree, (row,col)))

10              # -- compute vertex graph adjacency matrix
    vert_incidence_matrix = (incidence_matrix.dot(
        incidence_matrix_T) - vert_degree_diag_matrix)

            # -- extract vert2vert and p_vert2vert
    self.vert2vert = vert_incidence_matrix.indices
15  self.p_vert2vert = vert_incidence_matrix.indptr
```

3.2.3 netsim.py

Cette classe ne fait pas partie du code à écrire pour le projet, c'est pourquoi je ne ferai qu'une explication succincte des quelques méthodes qui sont directement appelées pour la simulation. On rappelle que la simulation consiste en un réseau de serveurs (sommets) qui calculent un certain temps (selon leur poids) puis se synchronisent en utilisant des liens (arêtes) qui ont aussi un certain poids (distance), sachant qu'ils ne peuvent se synchroniser sur plusieurs liens en même temps.

___init___()

La méthode de construction est importante car elle permet de connaître les attributs nécessaires au bon fonctionnement de l'algorithme.

numb_node : nombre de nœuds, ie de sommets ; *int*

numb_link : nombre de liens, ie d'arêtes ; *int*

node2link : vert2edge ; *array*

node2node : vert2vert ; *array*

p_node2node : p_vert2vert ; *array*

node_size : poids de chaque sommet ; *array*

link_size : poids de chaque arête ; *array*

numb_iter : nombre d'itération de la simulation ; *int*

Le poids des sommets et des arêtes est stocké dans le fichier VTK à la suite des `POINT_DATA` et `CELL_DATA`. Une méthode dans la classe **Mesh** permet de récupérer ces données.

Il y aussi d'autres attributs créées, qui servent à mesurer dans quelle étape de la simulation l'on se trouve.

node_step : Etat de chaque serveur, se charge de 0 jusqu'au poids du serveur ; *array*

link_step : Etat de chaque lien, se charge de 0 jusqu'au poids du lien ; *array*

flag_end : Indique si la simulation est terminée ; `False` tant qu'elle est en cours

numb_step : Nombre de pas nécessaires pour chaque itération ; *array*

Step()

Cette méthode permet de réaliser un pas élémentaire de simulation selon les règles définies précédemment, c'est-à-dire un temps de calcul du serveur ou une synchronisation à travers un lien.

Il suffit d'itérer cette méthode tant que le `flag\_end` n'indique pas que la simulation est terminée.

3.2.4 meshviz.py

Cette classe est assez compliquée car faisant appel à des fonctions internes à Paraview pour permettre la visualisation du graphe et de l'avancement de la synchronisation. Il y a notamment plusieurs options de caméra pour la visualisation qui ne seront pas traitées ici. On y retrouve également des méthodes semblables à `mesh.py` qui permettent de lire les fichiers VTK.

Config()

Cette méthode présente des options intéressantes pour déplacer la caméra, changer la taille de la fenêtre ou changer la couleur de fond. Dans notre exemple laisser les options par défaut suffira. Si le fond noir ne convient pas, on peut par exemple mettre en argument `Config((255, 255, 255))` pour l'avoir en blanc.

ReadMeshFromFile

Comme son nom l'indique, permet de lire un maillage depuis un fichier VTK.

WriteScreenshotToFile()

De même, permet d'enregistrer une image de la visualisation sur le disque dur.

Cette méthode est utile si l'on veut par la suite réaliser une vidéo montrant l'algorithme à l'œuvre sur un graphe.

Les formats jpg et png sont supportés.

SelectCellData() & SelectPointData()

Cette méthode permet de récupérer les données de "poids" des nœuds et des liens du maillage utilisés pour la simulation.

BuildCellColorScale() & BuildPointColorScale()

Permet de construire une échelle de couleurs en fonction de valeurs extrêmes prises par le poids des éléments du maillage.

BuildGlyph()

Permet de construire des sphères au niveau des nœuds, on y indique notamment le rayon de la sphère.

Render()

C'est la dernière méthode à être appelée, c'est celle-ci qui construit le rendu 3D.

3.2.5 sim.py

Maintenant que l'on a vu a toutes les méthodes à utiliser, il faut bien les utiliser pour réaliser le projet décrit précédemment. Pour rappel, il s'agit d'importer un graphe, d'assurer la coloration de ses arêtes, de trier en fonction de cette coloration et enfin de réaliser la simulation et la visualisation.

Il faut tout d'abord sélectionner un maillage. Pour le développement, un maillage simple pour pouvoir effectuer les algorithmes manuellement a été choisi.

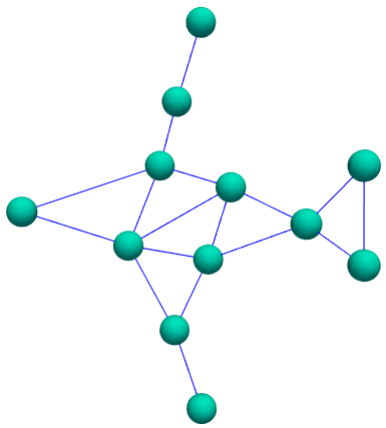

FIGURE 3.2 – Maillage simple

```
1 path_to_file = "../../data/in/MRG/AP3D-H0750B-SS0-LGM12.vtk"
```

Il faut ensuite importer le fichier avec **Mesh**, le transcrire dans **Graph**, créer le graphe adjoint (*line graph* en anglais), importer la coloration des arêtes sur le graphe et enfin trier en fonction de cette coloration.

```
# Create the Mesh object
m = Mesh()

# Import the file into Mesh object
m.ReadFromFileVtk(path_to_file)

# Create a Graph object with the attribute read by Mesh
g = Graph(m.numb_node,     # numb_vert
          m.numb_elem,     # numb_edge
          m.elem2node,     # edge2vert
          m.p_elem2node)   # p_edge2vert

# create line graph, with numb_edge, numb_vert, vert2edge,
    p_vert2edge
# create vert2edge
g.BuildEdge2Edge()

# build line graph (edge --> vert and vert --> graph)
lg = Graph()
lg.numb_vert = g.numb_edge
lg.numb_edge = g.numb_vert
lg.vert2vert = g.edge2edge
lg.p_vert2vert = g.p_edge2edge

# coloring edges, which are vertex of line graph
lg.ColorVertByWelshPowell()

# adding edges colors to the graph
g.edge_color = lg.vert_color

# sort edges by color
g.SortVert2EdgeByColor()

# build sorted vert2vert
g.BuildVert2Vert()
```

Le graphe est maintenant prêt pour la simulation et la visualisation. On initialise donc les deux objets avec les bonnes variables, notamment pour bien régler la taille des sphères et des couleurs.

```
# Create first simulation
s1 = NetSim(g.numb_vert, g.numb_edge, g.vert2edge, g.vert2vert, g.
    p_vert2vert)

# Add the size for the simulation, read from the mesh file
s1.node_size = m.ReadFieldFromFileVtkAscii(path_to_file, '
    subdomain_numb_nodes')[:,0]
s1.link_size = m.ReadFieldFromFileVtkAscii(path_to_file, '
    subdomain2numb_interface_node')[:,0]
```

Le `[:,0]` permet de transformer le tableau 2D fournit par la méthode de **Mesh** en un tableau 1D exploitable par la classe de simulation.

Il faut maintenant assurer la partie Visualisation, en réglant correctement les différents paramètres, notamment au niveau de l'échelle de couleurs.

```
# Create the MeshViz object
viz = MeshViz()
# Import the file
viz.ReadMeshFromFile(path_to_file)
viz.SelectCellData('subdomain2numb_interface_node')
viz.SelectPointData('subdomain_numb_nodes')

# Set the background color to white
viz.Config((255, 255, 255))

# Find the extrema for the size of nodes and links
point_data, cell_data = s1.node_size, s1.link_size
min_point, max_point = 0, max(point_data)
min_cell, max_cell = 0, max(cell_data)

# Set the ColorScale
viz.BuildCellColorScale(min_cell, max_cell)
viz.BuildPointColorScale(min_point, max_point)
# Select the radius of the sphere for nodes
viz.BuildGlyph(0.5)
```

Il ne reste maintenant plus qu'à simuler, et à générer le rendu au fur et à mesure, en prenant pour indice de couleur l'avancement des calculs sur les seveurs et les liens accessibles avec `node_step` et `link_step`. La simulation est terminée quand `flag_end` passe à `True`.

```
output = "../../data/out" # path for output files

while not(s1.flag_end): # run until the simulation is over
        point_data, cell_data = s1.node_step, s1.link_step #
            actual state of the simulation
        viz.Render(point_data, cell_data) # render
        s1.Step() # run the simulation for one step
        viz.WriteScreenshotToFile(output + str(i) + '.jpg') #
            saving the vizualisation
```

On fait donc un rendu à chaque étape, ce qui est assez lourd en termes de calculs. Pour mesurer seulement la performance des algorithmes, il suffit de compter le nombre d'étapes nécessaires pour terminer le processus, stocké dans l'attribut `numb_test` de la classe **NetSim**.

Chapitre 4

Résultats

4.1 Simulation

Pour mesurer l'efficacité des colorations, il suffit de regarder le nombre d'étapes nécessaires pour réaliser un cycle de calcul & synchronisation total. Pour ces tests, j'ai pris un maillage plus complexe. Il est possible de

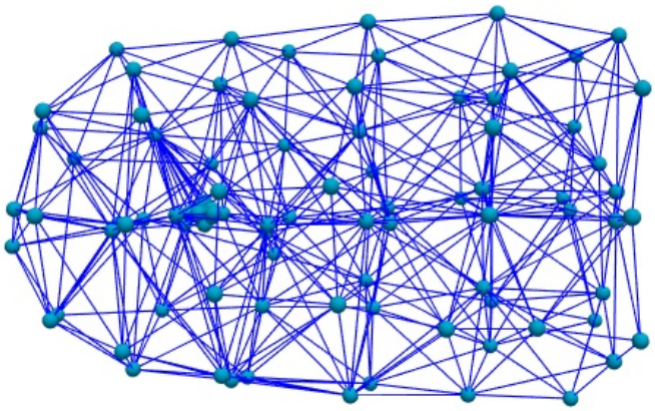

FIGURE 4.1 – Maillage pour les essais

configurer la simulation en indiquant de faire plusieurs cycles de calcul & synchronisation à la suite.

Bien que l'algorithme soit totalement déterministe, et donc que la durée totale ne devrait à priori pas changer, il se trouve que le deuxième cycle démarre même si tous les nœuds n'ont pas terminé l'étape 1. Il y a donc une certaine optimisation, mais pas dans les itérations au-delà, car on est alors dans le cas d'un fonctionnement en continu.

Sans coloration

Sans coloration, l'algorithme de simulation effectue **34 853 pas** pour la 1ère itération, **26 660** pour les itérations suivantes.

Avec coloration de Welsh Powell

Cette fois, la simulation est bien plus rapide, avec **10 962 étapes** pour la 1ère itération, et **10 215** pour les suivantes. On a donc un important gain en vitesse puisque la 1ère itération n'a besoin de réaliser que **31%** des étapes nécessaires sans coloration, 38% en fonctionnement en continu. En bref, on a **triplé** la vitesse de notre réseau !

4.2 Visualisation

Une fois que l'on a bien compris les différentes fonctions de la classe **MeshViz**, il n'y a pas de grandes difficultés.

Le rendu étant assez long, il peut être pertinent de ne faire que le rendu de quelques étapes sur toutes celles calculées, par exemples 1 sur 10.

Une fois toutes les images générées, on peut réaliser une vidéo de l'évolution du graphe pour voir la simulation en direct.

4.3 Correction des algorithmes

Au fur et à mesure du développement, il est pertinent de mettre en place des programmes testant les codes. Pour celà, il suffit de prendre un exemple simple, comme le graphe présenté avec l'explication de `sim.py`. Il faut alors comparer le résultat de l'algorithme testé avec le résultat attendu, préalablement trouvé à la main. Plusieurs de ces codes sont présents dans le dossier.

Annexe A

Installation de Paraview et VTK

Pour assurer la bonne visualisation du graphe, il faut utiliser le module VTK installé avec Paraview. Notons d'ailleurs que tous les codes écrits ne fonctionnent à priori qu'avec Paraview 4.1 , c'est donc cette version qu'il faut installer. La version de Python utilisée est la version 2.7 .

Pour pouvoir importer les classes VTK dans Python, il faut bien paramétrer les variables d'environnement pour que Python sache où chercher les classes qu'on lui demande d'importer. Les variables d'environnement permettent aux programmes du système de retrouver ce qui pourrait leur être utile. Par exemple, un programme qui a besoin d'accéder au dossier *Mes documents*, dont le chemin sera *C :/Users/[votre pseudo]/Mes documents* (*/home/[pseudo]/* sur Linux) n'est pas censé connaître votre pseudo. Il utilisera donc la variable d'environnement *$HOME* qui pointera sur votre dossier personnel.

Pour ce projet, il faut en configurer trois : ajouter à *PYTHONPATH* les emplacement de Paraview et des modules VTK, et ajouter à *PATH* l'emplacement de Paraview. Attention à bien l'ajouter à l'utilisateur avec lequel vous lancez votre programme : si le programme est lancé en mode administrateur, il faut toucher aux variables système.

Comment ajouter une variable d'environnement ?

Sous Windows

Taper "Variables" dans la recherche suffit pour trouver l'option *Modifier les variables d'environnement* (elle est accessible dans le Panneau de Configuration dans les paramètres systèmes avancés). Ensuite, un dernier clic sur *Variables d'environnement* et on arrive sur une interface claire pour pouvoir les ajouter avec la syntaxe suivante : dans *Nom de la variable*, mettre PATH ou PYTHONPATH, puis dans *valeur*, mettre le chemin, séparés par un **;** .

Sous Linux

Il suffit de rajouter une ligne dans le fichier de configuration du shell, *.bashrc* dans le home (*.zshrc* si vous utilisez *zsh*). Il faut alors ajouter une ligne du type `export PATH = "chemin"`, enregistrer le fichier et enfin charger les modifications avec un `source .bashrc`.