Algorithmique distribuée – GHS

Rapport de projet de développement

Audrey Lagardère

Deroi Tieffi

Thibault Bernard
Enseignant

Projet RT0803 – Algorithmique distribuée

Master Réseaux et Télécommunications
Parcours Administration et
Sécurité des Réseaux

Université de Reims Champagne-Ardennes
UFR des Sciences Exactes et Naturelles
Moulin de la Housse
BP 1039 - 51687 Reims Cedex 2

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

PLAN

I.Table des figures....3
II.Introduction....4
Consigne....4
Algorithme choisi....4
III.Choix technologiques....5
A/ Le fichier graphe....5
B/ La simulation....5
La représentation de l'algorithme....5
La représentation des données....5
Le déroulement de l'algorithme....6
C/ Le langage....7
IV.Modélisation....8
A/ Format des messages....8
B/ Format de graph....8
C/ Format de set_....9
V.Implémentation....11
A/ Fichiers utilitaires....11
Graphs.py....11
Files_basics.py....11
Register.py....12
Exchanges_queues.py....12
B/ Ghs.py....13
Bloc_initialisation....13
Traiter le message plus tard....14
Test dans procédure TEST....14
Test dans procédure REPORT....15
Changement de valeur....15
Terminaison d'algorithme....15
C/ ScriptG.py....15
VI.Conclusion....18
ANNEXES....19

I. Table des figures

Table des figures

Figure 1 : Exemple de fichier graphe....5
Figure 2 : Exemple de déroulement d'algorithme sur site A....6
Figure 3 : Format d'un message échangé....8
Figure 4 : Format de graph....9
Figure 5 : Exemple de variable graph....9
Figure 6 : Format de set_....10
Figure 7 : Pseudo-code read_graph_from_file....11
Figure 8 : Pseudo-code SEND_TO et RECV_FROM....12
Figure 9 : Pseudo-code bloc_initialization....14
Figure 10 : Pseudo-code test procedure TEST....14
Figure 11 : Pseudo-code test procedure REPORT....15
Figure 12 : Pseudo-code scriptG.py....16

II. Introduction

Consigne

Le projet consiste à réaliser une simulation d'un algorithme distribué, il devra être réaliser en binôme, ou en monôme. Les choix technologiques sont libres, mais doivent être assumés.

Deux algorithmes sont proposés:

- L'algorithme Gallager Humblet Spira
- L'algorithme de Tajibnapis

Vous devez fixer le cadre de manière à illustrer chaque étape de l'algorithme choisi. Votre simulateur doit pouvoir prendre en paramètre un graphe (pondéré ou non, suivant le choix de l'algorithme) sous forme de fichier descriptif (Matrice/Liste d'adjacence ou format plus complexe, au choix).

Par ailleurs pour l'algorithme de Tajibnapis, il faudra permettre au graphe d'évoluer de manière à illustrer les messages fail et repair.

La simulation sera fera de manière discrète, c'est-à-dire que le simulateur traitera événement par événement (au sens du modèle à passage de messages) et sans hypothèse sur le synchronisme.

Les descriptifs des algorithmes se trouveront sur la page Moodle du cours.

Un rapport sera à fournir. La note globale sera composée de 50% du projet et de 50% du rapport.

La date de restitution du projet (codes sources + rapport) est fixée au 17 juin de manière stricte.

Algorithme choisi

Ce document est le rapport accompagnant les codes sources du projet de notation de la matière RT0803 – Algorithmique distribuée. Il tend à expliquer les différents choix de développement de la démonstration de l'algorithme choisi : Gallager Humblet Spira (GHS par la suite).

III. Choix technologiques

Cette partie présente les premiers choix technologiques effectuées durant la mise en place du projet, ces choix posent les bases de notre implémentation.

A/ Le fichier graphe

La consigne précise que la simulation doit accepter en entrée un fichier contenant un graphe.

Nous avons choisi le format suivant : la première ligne contient le nombre de sites puis les lignes suivantes une arrête chacune. Une arrête est représentée par le premier sommet impliqué, un espace, le deuxième sommet impliqué, un espace, et enfin le poids de l'arrête.

Soit un fichier de la forme suivante :

```
3
A B 10
A C 2
B C 4
```

FIGURE 1 : EXEMPLE DE FICHIER GRAPHE

B/ La simulation

Pour simuler un système, il nous faut simuler ses sites et les échanges entre eux. Nous avons décidé d'implémenter **chaque site à travers un processus** et d'utiliser des **files de messages** afin de simuler les canaux d'échange.

La file de message a l'avantage de nous fournir une ressource partagée dont la synchronisation est gérée par le système. Ainsi, chaque site peut se positionner en lecture dans sa file, et lire les messages les uns à la suite des autres. Les autres sites doivent être capables d'écrire dans la file.

La consigne indique qu'aucune hypothèse ne doit être faite sur le synchronisme. Ici, les messages sont envoyés dans la file sans synchronisme, car les processus-sites sont indépendants, ce qui simule des temps de trajet aléatoires.

La représentation de l'algorithme

L'algorithme GHS nous a été fourni sous forme de blocs. Nous avons choisi d'implémenter ces blocs sous la forme de fonction. C'est notre fichier ***ghs.py*** qui contient ces différentes fonctions. Nous reviendrons dessus plus en avant dans ce rapport.

La représentation des données

Les données (variables) utilisées par nos différents sites doivent donc être transmises aux fonctions-blocs de l'algorithme. Afin de faciliter ces échanges et de les rendre

homogènes, nous avons choisi de regrouper toutes les variables propres à un site dans un set de variables. Ce set est nommé ***set***_.

Le déroulement de l'algorithme

Afin de visualiser les différentes étapes de l'algorithme avec notre programme multi-processus, nous avons choisi d'accorder un fichier à chaque processus-site. Ainsi, un site peut stocker chaque étape de son avancée dans l'algorithme sans interférer avec un autre.

Nous avons décider d'enregistrer les étapes suivantes :

- ✔ Le changement de valeur d'une variable
- ✔ La réception d'un message
- ✔ L'envoi d'un message
- ✔ L'entrée dans une procédure
- ✔ L'état final du ***set***_ de variables

Un fichier après une exécution du programme peut donc ressembler à :

```
    A envoie à D : (connect, 0, None, None) ...
A recoit de D : (test, 1, 1, None) ...
    D envoie à A : (test, 1, 1, None) ...
A recoit de D : (initiate, 1, 1, find) ...
    niv : 0 --> 1
    nom : None --> 1
    etat : found --> find
    pere : None --> D
    mcan : None --> None
    mpoids : 0.1 --> inf
    recu : 0 --> 0
    testcan : None --> B
    A envoie à B : (test, 1, 1, None) ...
A recoit de D : (test, 1, 1, None) ...
    A envoie à D : (reject, None, None, None) ...
A recoit de B : (reject, None, None, None) ...
    canal[B] : basic --> reject
    testcan : B --> None
    etat : find --> found
    A envoie à D : (report, inf, None, None) ...
A recoit de father : (termine, None, None, None) ...

----------------------
Set final :
canal : {'D': 'branch', 'B': 'reject'}
recu : 0
f_ : RAPPORT_graphecours__060620_233955_nodeA.txt
edges : {'D': 3, 'B': 4}
niv : 1
pere : D
mpoids : inf
testcan : None
mcan : None
etat : found
i : A
nom : 1
```

FIGURE 2 : EXEMPLE DE DÉROULEMENT D'ALGORITHME SUR SITE A

C/ Le langage

Notre attention s'est portée sur le python. C'est un langage très flexible et que nous souhaitions tout simplement manipuler d'avantage.

Ainsi, afin d'utiliser des files de messages et des processus, nous nous sommes intéressés au module python nommé ***multiprocessing***. Nous avons donc établi que chaque processus de site serait créé à travers un appel système ***fork()***, et que les files seraient des ***SimpleQueues***.

IV. Modélisation

Nous avons établi précédemment que les sites sont représentés par des processus indépendants et les échanges entre eux passent par des files de message.

Une file de message attribuée à chaque site permet à ceux-ci de lire les messages qui leur sont envoyés les uns à la suite des autres.

La ***SimpleQueue*** proposée par le module ***multiprocessing*** permet à différents processus de se synchroniser pour écrire dans la ressource grâce à un verrou intégré. C'est très bien car cela nous évite d'avoir à le gérer nous même. De plus, la ***SimpleQueue*** permet d'envoyer et recevoir tout type d'objets dont il faut simplement connaître le format au moment de vouloir l'exploiter.

A/ Format des messages

Pour pouvoir exploiter nos messages après leur lecture dans la file, nous avons besoin d'en connaître le format.

Pour faciliter la lecture du code et sa compréhension, nous avons principalement utilisées les structures de dictionnaire proposées par python qui permettent d'accéder à la valeur d'une variable en passant par a clef.

Ainsi, tous les messages transmis dans les files de messages (***SimpleQueue***) sont de la forme suivante :

```
{
        "type" : string,
        "val1" : X,
        "val2" : Y,
        "val3" : Z,
        "sender" : id
}
```

FIGURE 3 : FORMAT D'UN MESSAGE ÉCHANGÉ

Chaque message contient donc un type, qui est en réalité le nom du bloc-fonction qui sera appelé au moment de la réception.

Ensuite il contient, trois valeurs, qui seront les paramètres du bloc-fonction. Les trois valeurs ne sont pas toujours toutes utilisées mais toujours envoyées, avec la valeur ***None*** si nécessaire.

Enfin, l'identifiant de l'envoyeur est joint au message afin de palier au fait que nous utilisons une unique file pour contenir tous les messages reçus.

B/ Format de graph

Notre variable contenant un graph est un enchevêtrement de dictionnaires. Elle est de la forme suivante :

```
{       "size" : X,
        "edges" : {     node : {        node : value,
                                            …
                                },
                        …
                }
}
```

FIGURE 4 : FORMAT DE GRAPH

Notre dictionnaire de graph contient donc deux clefs : ***size*** et ***edges***.

La valeur attribué à ***edges*** est un dictionnaire dont chaque clef est un ***nœud***.

La valeur attribuée à chaque ***nœud*** est encore une fois un dictionnaire dont les clefs sont les ***voisins*** et les valeurs le poids de l'arrête qui relie le nœud courant au voisin courant.

Ainsi, la variable graph contenant le graph stocké dans notre fichier en Figure1 est la suivante :

```
{       "size" : 3,
        "edges" : {     'A' : {   'B' : 10,
                                  'C' : 2
                                },
                        'B' : {   'A' : 10,
                                  'C : 4
                                },
                        'C' : {   'A' : 2,
                                  'B' : 4
                                }
                }
}
```

FIGURE 5 : EXEMPLE DE VARIABLE GRAPH

Ainsi, chaque arrête est stockée deux fois dans notre variable : dans la liste d'adjacence de chaque sommet qu'elle implique.

Cette structure est intéressante car elle permet de récupérer la totalité des arrêtes d'un nœud donné très facilement : **graph["edges"]["noeud"]**.

C/ Format de set_

Comme nous l'avons déjà indiqué, nous avons choisi de réunir la totalité des variables propres à un site dans un dictionnaire nommé ***set_***. Ce set a donc la forme suivante :

```
{
        "i" : id,
        "canal" : {     node : "branch|basic|reject",
                        ...
                },
        "niv" : entier,
        "etat" : "found|find",
        "recu" : entier,
        "nom" : reel,
        "pere" : id,
```

```
"mcan" : id,
"testcan" : id,
"mpoids" : reel,
"edges" : graph,
"queues" : {    node : queue_associée,
                ...
            },
"f_" : "nom_de_fichier.txt"
}
```

FIGURE 6 : FORMAT DE SET_

Nous souhaitons souligner qu'ici, la clef ***edges*** contient seulement les voisins du nœud. Ce n'est donc pas **graph["edges"]** qui y est insérée mais **graph["edges"]["noeud"]**.

Les clefs ***canal*** et ***queues*** contiennent ici encore des dictionnaires dont les clefs sont des ***nœuds***.

V. Implémentation

Cette partie présente l'implémentation de la simulation à travers l'étude de chacun des fichiers et de leurs algorithmes associés.

A/ Fichiers utilitaires

Les fichiers que nous nommons utilitaires sont les fichiers contenant les fonctions appelées tout au long du programme (à l'exception des fonctions-blocs de l'algorithme GHS) :

- ✔ graphs.py
- ✔ exchanges_queues.py
- ✔ files_basics.py
- ✔ register.py

Graphs.py

Ce fichier contient une unique fonction chargée de créer une structure graph depuis un fichier contenant un graph : **read_graph_from_file(fname)**. Cette fonction prend donc en paramètre un nom de fichier et retourne une structure graph.

Voici le pseudo-code de cette fonction très simple :

```
(1)   read_graph_from_file(fname)
(2)   |   ouvrir le fichier fname
(3)   |   graph["size"] ← la première ligne du fichier
(4)   |   Pour chaque ligne du fichier
(5)   |   |   Si mot1 n'est pas un nœud connu
(6)   |   |   |   Ajouter mot1 comme clef dans graph["edges"]
(7)   |   |   Fin Si
(8)   |   |   Ajouter mot2&mot3 comme clefs&valeurs dans graph["edges"][mot1]
(9)   |   |   Si mot2 n'est pas un nœud connu
(10)  |   |   |   Ajouter mot2 comme clef dans graph["edges"]
(11)  |   |   Fin SI
(12)  |   |   Ajouter mot1&mot3 comme clefs&valeurs dans graph["edges"][mot2]
(13)  |   Fin Pour
(14)  |   fermer fichier
(15)  Fin
```

FIGURE 7 : PSEUDO-CODE READ_GRAPH_FROM_FILE

Le point à noter est que chaque ligne du fichier (3) à (12), qui contient une arrête, implique d'eux entrées dans les listes d'adjacence : l'un pour le sommet 1 (5) à (7) et l'autre pour le sommet 2 (8) à (11).

Files_basics.py

Ce fichier contient une procédure permettant d'écrire un texte dans un fichier : **write_in_file(f_name, txt)**.

Cette fonction enrobe simplement des appels très classiques en python aux fonctions open() et write().

Register.py

Ce fichier contient les fonctions permettant d'enregistrer le déroulement de l'algorithme dans des fichiers :

- ✔ **send_msg(sender, receiver, name, val1, val2, val3, f_)**
- ✔ **recv_msg(sender, receiver, name, val1, val2, val3, f_)**
- ✔ **change_var(name, old, new, f_)**
- ✔ **set_state(set_, f_)**

Ces fonctions prennent simplement en paramètre les variables dont le contenu doit être écrit dans le fichier et bien entendu le nom de ce fichier.

Il n'est pas très intéressant de détailler ces fonctions étant donné qu'elles ne font que mettre en forme les variables à enregistrer puis appellent la fonction **write_in_file**.

Exchanges_queues.py

Ce fichier contient les fonctions permettant d'envoyer et de recevoir des messages à travers des files de messages :

- ✔ **send_to(q_, sender, receiver, name, val1, val2, val3, f_)**
- ✔ **recv_from(q_, receiver, f_)**.

La procédure d'envoi prend en paramètre la queue dans laquelle écrire ainsi que les données à envoyer.

La fonction de réception prend en paramètre la queue dans laquelle lire et l'identifiant du nœud appelant pour retourner un objet "message".

Les deux fonctions prennent également en paramètre le nom de leur fichier de déroulement d'algorithme.

```
(1)   send_to(q_, sender, receiver, name, val1, val2, val3, f_)
(2)   |     créer objet message avec name, val1, val2, val3 et sender
(3)   |     q_.put(message)
(4)   |     register.send_msg(sender, receiver, name, val1, val2, val3, f_)
(5)   Fin
(6)
(7)   recv_from(q_, receiver, f_)
(8)   |     objet ← q_.get()
(9)   |     register.recv_msg(objet["sender"], receiver, objet["name"], objet["val1"],
(10)  |                        objet["val2"], objet["val3"], f_)
(11)  Fin
```

FIGURE 8 : PSEUDO-CODE SEND_TO ET RECV_FROM

Les fonctions put() (3) et get() (8) sont les fonctions proposées par le module ***multiprocessing*** sur des objets ***SimpleQueue***. Donc les variables ***q_*** (1) et (7) doivent être de ce type.

B/ Ghs.py

Ce fichier contient l'implémentation de tous les blocs de notre algorithme GHS ainsi qu'une petite fonction utilitaire :

- ✔ **find_min_val_return_key_in_dict(edgesList)**
- ✔ **bloc_initialization(edgesList, q_, i, f_name)**
- ✔ **bloc_connect(L, j, set_)**
- ✔ **bloc_initiate(L, F, S, j, set_)**
- ✔ **TEST(set_)**
- ✔ **bloc_test(L, F, j, set_)**
- ✔ **bloc_accept(j, set_)**
- ✔ **bloc_reject(j, set_)**
- ✔ **REPORT(set_)**
- ✔ **bloc_report(poids, j, set_)**
- ✔ **CHANGEROOT(set_)**
- ✔ **bloc_changeroot(set_)**

La première fonction prend en paramètre une dictionnaire et retourne la clef associée à la valeur la plus petite contenue dans le dictionnaire.

Les fonctions-blocs prennent toutes en paramètre les arguments dont les blocs de notre algorithme ont besoin pour fonctionner additionnés du ***set***_ et retournent le ***set***_ mis à jour.

L'implémentation de notre algorithme est pour la plupart une simple transcription en python du pseudo-code. Nous nous intéressons donc ici simplement aux lignes de codes dont la transcription a nécessité des ajustements, aux points d'intérêt de notre sode.

Bloc_initialisation

Nous avons modifié le pseudo-code du bloc 1 : initialisation afin de pouvoir y initialiser la totalité de notre ***set***_ : c'est à dire y placer les queues, le nom de fichier ainsi que la liste d'adjacence du nœud concerné mais également préparer toutes les variables pour leur utilisation future.

Ainsi, le pseudo-code de la fonction-bloc initialisation est le suivant :

```
(1)   bloc_initialization(edgesList, q_, i, f_name)
(2)   |     set_["i"] ← i
(3)   |     Pour chaque voisin dans edgesList
(4)   |     |     set_["canal"]["voisin"] ←basic
(5)   |     Fin Pour
(6)   |     set_["niv"] ← 0
(7)   |     set_["etat"] ←found
(8)   |     set_["recu"]←0
(9)   |     set_["nom"], set_["pere"], set_["mcan"], set_["testcan"] ← None
(10)  |     set_["mpoids"] ← 0,1
(11)  |     set_["edges"] ← edgesList
(12)  |     set_["queues"] ← q_
(13)  |     set_["fichier"] ← f_name
(14)  |     k ← find_min_val_return_key_in_dict(edgesList)
(15)  |     set_["canal"][k] ← branch
```

```
(16)    |       envoie connect 0 à k
(17)    Fin
```

FIGURE 9 : PSEUDO-CODE BLOC_INITIALIZATION

Ici il est important de savoir que cette fonction est appelée par le script maître (celui qui crée les processus-sites). Cela permet de comprendre que la variable ***i*** (1), passée en paramètre, permet au maître d'indiquer à son fils quel est son identifiant. Nous reviendrons la dessus plus en aval de ce document, lorsque nous aborderons le pseudo-code du processus père.

Cet algorithme est très simple à comprendre. La variables sont initialisées avec les valeurs proposées dans l'algorithme et celle données en paramètre. La variable ***mpoids*** (10) est initialisée avec la valeur de 0,1 arbitraire, afin de lui attribuer le type de réel.

Traiter le message plus tard

L'intégration de la consigne "traiter le message plus tard" a été effectuée sous la forme d'un message que le nœud se renvoie à lui même. En effet, étant donné l'utilisation d'une file de messages pour simuler la réception de ceux-ci, les messages reçus sont "consommés" et le moyen le plus simple de les traiter plus tard et de les renvoyer dans la file en faisant attention de lui attribuer le bon expéditeur.

Dans notre code, nous avons également attendu une seconde avant de ré-envoyer le message, ce afin d'éviter au programme de boucler inutilement. Un message envoyé trop rapidement sera relu trop rapidement et les variables n'auront pas eu le temps de changer de valeur.

Test dans procédure TEST

La procédure TEST repose sur le test donné dans l'algorithme suivant :

```
if (il existe j appartenant à vois|canal[j] = basic) then
        Choix de j|canal[j] = basic ^ poids(i, j) minimal pour tout j appartenant à vois
        ...
```

Le pseudo-code que nous avons utilisé pour mettre en place ce code dans notre programme en python est le suivant (par simplicité nous omettons le **set**_) :

```
(1)     existe ← dictionnaire vide
(2)     Pour chaque j dans voisin
(3)     |       Si canal[j] = basic
(4)     |       |       Ajouter j,poids(i,j) à existe
(5)     |       Fin Si
(6)     Fin Pour
(7)     Si existe n'est pas vide
(8)     |       j ← find_min_val_return_key_in_dict(existe)
(9)     |       ...
```

FIGURE 10 : PSEUDO-CODE TEST PROCEDURE TEST

Le code couleur permet de bien visualiser les transcriptions.

Nous avons ici utilisé une variable complémentaire ***existe*** (1), un dictionnaire, afin de stocker seulement les adjacences dont le ***canal*** est à la valeur basic (3) à (5). Pour pouvoir utiliser notre fonction de recherche de valeur minimum sur ce dictionnaire (8).

Test dans procédure REPORT

La procédure REPORT repose sur le test donné dans l'algorithme suivant :

```
if recu = #{j/canal[j] = branch ^ j != pere} ^ testcan = undef then
    ...
```

Le pseudo-code que nous avons utilisé pour mettre en place ce code dans notre programme en python est le suivant (par simplicité nous omettons le **set**_) :

```
(1)     test ← 0
(2)     Pour chaque j dans voisins
(3)     |       Si canal[j] = branch et j != pere
(4)     |       |       test = test + 1
(5)     |       Fin Si
(6)     Fin Pour
(7)     Si recu = test et testcan = none
(8)             ...
```

FIGURE 11 : PSEUDO-CODE TEST PROCEDURE REPORT

Nous avons donc ici utilisé une variable ***test*** (1) et en plus par rapport à l'algorithme donné afin de pouvoir compter le nombre de voisins concernés (4).

Changement de valeur

Afin de garder trace du déroulement de l'algorithme, nous choisissons d'entrer dans les fichiers chaque changement de valeurs de variables de l'algorithme initial. Ces changements sont opérées depuis les fonctions-blocs de ***ghs.py***.

C'est pourquoi chaque ligne d'assignation de valeur de l'algorithme initial est dédoublée dans notre implémentation.

```
Testcan ← udef
```

Devient :

```
register.change_var("testcan", set_["testcan"], "None", set_["f_"])
set_["testcan"] = None
```

Terminaison d'algorithme

Pour notre implémentation, lorsque l'algorithme se termine, le processus-site envoie à son maître (le processus scriptG qui l'a créé) un message de terminaison. Ce message étant interprété par le père pour terminer proprement le programme comme nous le voyons dans le paragraphe suivant.

C/ ScriptG.py

Le fichier le ***scriptG.py*** permet de créer tous les processus-sites. Voici son pseudo-code, détaillant précisément les passages les plus importants et utilisant les fonctions présentées auparavant :

```
(1)     Vérifier que argument est donné (le nom du fichier contenant le graph)
(2)     g_ ← graps.read_graph_from_file(argument)
```

```
(3)   Pour chaque nœud dans g_["edges"]
(4)   |     queues[noeud] ← Nouvelle SimpleQueue
(5)   Fin Pour
(6)   queues["father"] ← Nouvelle SimpleQueue
(7)   fileName ← Créer un nom de fichier pour l'instance du programme à partir de la date
(8)
(9)   Pour chaque nœud dans g_["edges"]
(10)  |     set_ ← ghs.bloc_initialization(g_["edges"][noeud], queues, noeud,
(11)  |                                    fileName+"_node"+node+".txt")
(12)  |     n = os.fork()
(13)  |     Si n = 0
(14)  |     |     break ;
(15)  |     Fin Si
(16)  Fin Pour
(17)
(18)  Si n = 0
(19)  |     Tant que set_ != TERMINE
(20)  |     |     msg ← eq.recv_from(set_["queues"][set_["i"]], set_["i"], set  _["f_"])
(21)  |     |     Si msg["type"] = "connect"
(22)  |     |     |     set_ = ghs.bloc_connect(msg["val1"], msg["sender"],  set_)
(23)  |     |     Sinon Si msg["type"] = "initiate"
(24)  |     |     |     set_ = ghs.bloc_initiate(msg["val1"], msg["val2"], m  sg["val3"],
(25)  |     |     |                              msg["sender"], set_)
(26)  |     |     Sinon Si msg["type"] = "test"
(27)  |     |     |     set_ = ghs.bloc_test(msg["val1"], msg["val2"], msg["  sender"],
(28)  |     |     |                          set_)
(29)  |     |     Sinon Si msg["type"] = "accept"
(30)  |     |     |     set_ = ghs.bloc_accept(msg["sender"], set_)
(31)  |     |     Sinon Si msg["type"] = "reject"
(32)  |     |     |     set_ = ghs.bloc_reject(msg["sender"], set_)
(33)  |     |     Sinon Si msg["type"] = "report"
(34)  |     |     |     set_ = ghs.bloc_report(msg["val1"], msg["sender"], set_)
(35)  |     |     Sinon Si msg["type"] = "changeroot"
(36)  |     |     |     set_ = ghs.bloc_changeroot(set_)
(37)  |     |     Sinon Si msg["type"] = "termine"
(38)  |     |     |     end_set_ = set_
(39)  |     |     |     set_ = "TERMINE"
(40)  |     |     Fin Si
(41)  |     Fin Tant que
(42)  |     Afficher "Done !"
(43)  Sinon
(44)  |     msg = eq.recv_from(queues["father"], "father", None)
(45)  |     Attendre 2 secondes
(46)  |     Pour tout nœud dans queues
(47)  |     |     eq.send_to(q_, "father", None, "termine", None, None, None,  None)
(48)  |     Fin Pour
(49)  |     Afficher "J'ai fini !"
(50)  Fin Si
(51)  Attendre 3 secondes
(52)  Quitter
```

FIGURE 12 : PSEUDO-CODE SCRIPTG.PY

L'algorithme peut être divisé en 4 parties : l'initialisation (1) à (16), la partie propre au processus-site (18) à (42), la partie terminaison (43) à (49) et la partie fermeture (50) à (51).

La partie initialisation (1) à (16) se charge comme son nom l'indique d'initialiser les processus-sites et de les lancer. Le graph est récupéré dans le fichier donné en paramètre (2), puis des queues sont créés à partir des nœuds présents dans le graph (3) à (5) auxquelles vient s'ajouter une queue ***"father"*** qui sert à contacter le processus maître (6).

Une nouvelle boucle sur les nœuds du graph (9) à (16) se charge enfin de créer et initialiser les processus-sites en utilisant ***os.fork()*** (12). Cette fonction crée un processus indépendant en clonant toutes les informations du programme père. Cela est intéressant dans notre cas car il suffit au père d'initialiser le ***set***_ (10) puis d'appeler juste ensuite os.fork() pour que le processus résultant ait la valeur de ***set***_ lui correspondant. Cependant, étant donné que le clone du processus père se situe lui également en pleine boucle, il nous faut l'en faire sortir (13) à (15) pour éviter les créations intriquées de processus fils.

La partie processus-site (18) à (42) est bornée par une condition permettant d'identifier le processus dans lequel on se trouve. La variable ***n*** est initialisée dans le maître mais toujours égale à 0 dans les fils. A l'intérieur se trouve une unique grande condition "Tant que" (19) permettant de boucler sur une lecture dans la queue du site (20) tant que le ***set***_ n'est pas égal à "TERMINE".

La valeur du message lu dans la queue est comparée avec toutes celles possibles afin d'appeler le bloc-fonction correspondant (21) à (36). Si cette valeur n'est pas celle d'un bloc mais "termine" (37), alors le set final est stocké (38) puis le fils quitte sa boucle (39).

Cette solution permet à notre programme de terminer proprement les processus-fils qui ne parviennent pas d'eux même à la partie de terminaison de l'algorithme.

La partie terminaison (43) à (49) est dédiée au processus maître. Celui-ci se place en lecture sur sa propre file (44) afin d'être tenu informé que l'algorithme s'est termine. Ainsi, il envoie après un court instant un message de type "termine" à tous ses fils (46) à (48) pour qu'ils puissent se fermer correctement.

La partie fermeture (50) à (51) se contente d'attendre 3 secondes (50) avant de quitter (51). Ce délai a été apposé afin de palier aux problèmes rencontrés par les programmes python lorsqu'un processus lié à une queue est fermé tandis qu'un autre processus tente d'accéder à cette queue. Dans notre cas, les trois secondes d'attente suffisent à ce que tous les fils parviennent au point de fermeture dans leur code, ce qui nous évite les erreurs.

VI. Conclusion

Parfois une erreur python se produit en fin de programme. Après de nombreuses recherches il semblerait que, quelques fois, les files de messages queues en python produisent cette erreur au moment de leur fermeture. Après beaucoup de recherches et de tests, nous ne sommes pas parvenus à faire disparaître l'erreur et il semblerait que nous ne soyons pas un cas isolé parmi les utilisateurs de python. Étant donné que cette erreur n'affecte pas notre simulation et que l'algorithme est déjà terminé au moment où elle se produit, nous avons considéré que l'objectif était rempli tout de même.

Nous avons essayé, dans ce rapport, de retranscrire le processus de développement de notre simulation.

Nous aurions pu commenter le code, mais avons jugé que les commentaires directement intégrés à celui-ci suffiraient ; dans la mesure où nous avons pris un soin tout particulier à les rendre très complets.

Ce document vient donc se positionner en accompagnement du code du programme, pour la totale compréhension de ce dernier.

Annexes

Annexes

Annexes

A/ Contenu du dossier

Voici le dossier du projet dans son ensemble :

```
|__projet_GHS_Lagardère_Tieffi
        |__fichiers_traces
        |       |__RAPPORT_txtGcours__150620_1522_nodeA.txt
        |       |__RAPPORT_txtGcours__150620_1522_nodeB.txt
        |       |__RAPPORT_txtGcours__150620_1522_nodeC.txt
        |       |__RAPPORT_txtGcours__150620_1522_nodeD.txt
        |       |__RAPPORT_txtGcours2__150620_1522_nodeA.txt
        |       |__RAPPORT_txtGcours2__150620_1522_nodeB.txt
        |       |__RAPPORT_txtGcours2__150620_1522_nodeC.txt
        |       |__RAPPORT_txtGcours2__150620_1522_nodeD.txt
        |
        |__programme
        |       |__exchanges_queues.py
        |       |__files_basics.py
        |       |__ghs.py
        |       |__graphs.py
        |       |__register.py
        |       |__scriptG.py
        |       |__txtGcours
        |       |__txtGcours2
        |       |__txtPetitG
        |       |__txtTtPetitG
        |
        |__algo_GHS.pdf
        |
        |__rapport_projet_GHS.pdf
        |
        |__README.txt
```

Les fichiers trace sont fournis au cas ou un problème empêcherait l'exécution du programme.

Le programme est donc composé des fichiers sources additionnés de fichiers graphs préparés : dont les graphs vus en cours.

Le fichier algo_GHS.pdf est l'algorithme donné en cours.

Le fichier rapport_projet_GHS.pdf est ce fichier même.

Le fichier README.txt explique comment utiliser le programme.