TP3 : Graphes et algorithmes combinatoires distribués avec MPI

TP a pour objectif de faire découvrir et d'illustrer l'intérêt du parallélisme sur des problèmes combinatoires de graphes. Les cas de la recherche d'isomorphismes et de sous-isomorphismes et du stable maximal/de la clique maximale sont utilisés. Ces problèmes sont des éléments clefs dans un large spectre de domaines. Les algorithmes de résolution peuvent être vus dans ces cas des recherches arborescentes: l'algorithme de backtracking pour les (sous-)isomorphismes est un cas particulier d'algorithme de Programmation Par Contraintes (PPC), stable maximal/clique maximale présentent un cas simple de Branch-and-Bound. Dans ces deux cas, les charges de travail sur les processus peuvent être très différentes en plus d'être mal connues a priori, ce qui en fait un cas d'étude intéressant pour illustrer des problématiques d'équilibrage de charge en programmation distribuée avec MPI.

Les classes `Graph` et `Mapping`

Ce TD va vous faire utiliser des graphes. Vous trouverez une implémentation maison dans `graph.h` et `graph.cpp`. Ces fichiers définissent deux classes utilisables : `Graph` et `Mapping`.

- La classe `Graph` comprend tout le nécessaire pour manipuler des graphes.
- La classe `Mapping` permet de représenter l'isomorphisme en cours de construction.

Les méthodes de `Graph` et `Mapping` sont expliquées dans les commentaires dans `graph.h`.

Attention : Ne modifiez pas `graph.h` ni `graph.cpp` !

Il est à noter que la bibliothèque Boost contient une implémentation générique de graphes et des algorithmes usuels, dont les performances sont optimisées pour des graphes denses ou épars. Les classes `Graph` et `Mapping` permettent d'avoir une implémentation simple pour ce TP.

Définitions

Un graphe G comprend un ensemble de nœuds que l'on notera G_v, et un ensemble d'arêtes que l'on notera G_e. On considérera les arêtes et les nœuds comme non-étiquetés et donc interchangeables, au moins pour commencer. On distinguera néanmoins les arêtes entrantes et les arêtes sortantes.
Deux graphes G et H sont **isomorphes** s'il existe f : $G_v \rightarrow H_v$ telle que $(u,v) \in G_e$ ssi $(f(u),f(v)) \in H_e$.
Un graphe G est un **sous-isomorphisme** du graphe H s'il existe $H' \subseteq H$ tel que G et H' sont isomorphes.

Partie I : Graphes, isomorphismes, et sous-isomorphismes

Exercice 1 : Isomorphisme simple

Nous allons commencer par le cas le plus simple : celui de l'isomorphisme. Ce problème appartient à la classe de complexité NP puisque une solution est vérifiable en temps polynomial mais le problème du test de l'isomorphisme de graphe n'a pas été prouvé NP-dur et nous ne connaissons pas d'algorithme polynomial dans le cas général. (C'est un des rares problèmes connus dans cette situation, avec aussi celui de la décomposition en facteurs premiers.)

Il existe cependant des algorithmes efficaces en temps polynomial pour des classes de graphes spécifiques, comme les arbres ou les graphes planaires ; mais soit ces restrictions ne conviennent pas à ce

que nous voulons faire, soit les algorithmes les exploitant sont beaucoup plus complexes à implémenter. Nous choisirons donc, dans un premier temps, une solution exhaustive basée sur le **backtracking**.

L'algorithme

On veut tester si G et H sont isomorphes.

- Soient G'_v et H'_v des ensembles de nœuds tels que G'_v ⊆ G_v et H'_v ⊆ H_v avec une fonction injective M telle que pour tout g' ∈ G'_v, il existe h'=M(g') ∈ H'_v.
- L'algorithme commence avec G'_v = H'_v = ∅ et termine (éventuellement) quand G'_v=G_v ;
- Les n nœuds de G (resp. H) seront appelés [g_0,...,g_{n-1}] (resp. [h_0,...,h_{n-1}]).

Le pseudocode de l'algorithme est le suivant :

```
Si |G| != |H| : G et H ne sont pas isomorphes ;
Initialement, M = {} ;
extendInjection(M) ;
Si M est complet : G et H sont isomorphes ;
Sinon : G et H ne sont pas isomorphes ;
```

... où la procèdure `extendInjection`, qui prend une injection d'un sous-graphe de G dans H et essaie de l'étendre à une injection de G dans H est définie comme

```
extendInjection(M) :
  Si M est complet : return ;
  Pour tout g_i dans G_v :
    Pour tout h_j dans H_v :
      Si (pour tout (g',h') dans M, g_i != g')
      et (pour tout (g',h') dans M, h_j != h')
      et (pour tout (g',h') dans M, (g_i,g') est dans G_e ssi (h_j,h') est dans H_e)
      et (pour tout (g',h') dans M, (g',g_i) est dans G_e ssi (h',h_j) est dans
H_e) :
        M = M + (g_i, h_j) ;
        extendInjection(M) ;
        Si M est complet : return ;
          Sinon : M = M - (g_i, h_j) ;
    return ;
```

Implémentation

La procédure `void findIsomorphism(Graph G, Graph H)` fait appel à `void extendInjection(Mapping& m)` pour trouver un isomorphisme entre G et H, s'il y en a, et l'imprimer sur `std::cout`. Il vous faut **implémenter cet algorithme** en complétant la procédure `extendInjection` dans `Isomorphism.cpp`.
Nous vous proposons d'utiliser les méthodes de la classe `Mapping` :

- `Mapping::isFull` renvoie `true` si et seulement si le mapping apparie un nœud de H à chaque nœud de G ; c'est à dire, si le mapping a bien été étendu à une fonction injective avec G comme domaine.
- `Mapping::areMappable(int idg, int idh)` vérifie si l'appariement g\mapsto h est compatible avec le mapping (i.e. vérifie que les nœuds g et h ne sont pas déjà dans le mapping, et qu'ils vérifient les conditions sur leurs voisins);
- `Mapping::addToMap(int idg, int idh)` rajoute un appariement g \mapsto h au mapping *sans aucune vérification que l'appariement est compatible avec ceux déjà dans le mapping*;
- `Mapping::deleteFromMap(int idg)` enlève un appariement g -> h du mapping

Attention : `extendInjection` peut (et souvent doit) modifier son argument, en essayant de l'étendre vers un isomorphisme complet.

Compilez

Vous pouvez compiler votre code avec la commande
`mpic++ graph.cpp Isomorphism.cpp TestExercice1.cpp -o TestExercice1`

Testez

Les deux graphes G_1 et H_1 isomorphes ci-dessous, ainsi que deux autres graphes G_2 et H_2 non isomorphes, sont initialisés pour vous dans la fonction `main` du fichier `TestExercice1.cpp` afin que vous puissiez tester votre algorithme.

Schéma des graphes G_1 et H_1 utilisés pour le test.

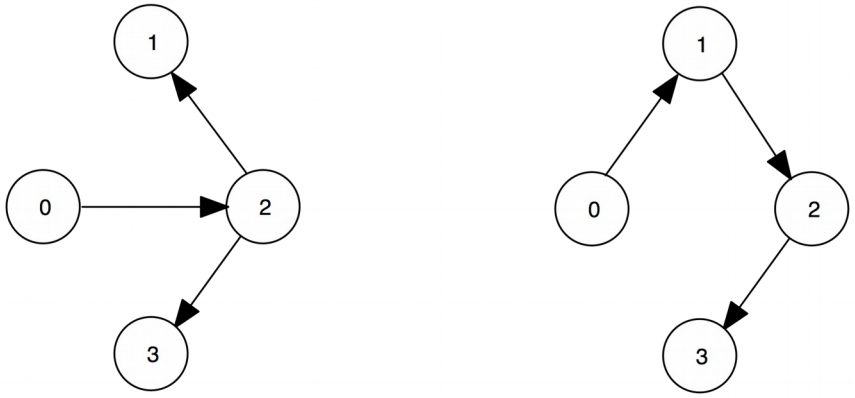

Schéma des graphes G_2 et H_2 utilisés pour le test.
La commande
`mpirun -np 1 TestExercice1` devrait donc afficher

```
Test d'isomorphismes entre graphes G_1 et H_1
[0 => 0] [1 => 2] [2 => 1]
Test d'isomorphismes entre graphes G_2 et H_2
Pas d'isomorphisme
```

Exercice 2: Sous-isomorphisme simple

Nous allons maintenant essayer de savoir si le graphe G est un sous-isomorphisme de H, c'est à dire s'il existe un sous-graphe H' contenu dans H et isomorphe à G.

Implémentation

Cette question ne nécessite pas de modifier `extendInjection`, mais simplement de l'appeler de la bonne manière depuis la procédure `void findSubIsomorphism(Graph G, Graph H)`. Cette procédure doit chercher un sous-isomorphisme entre G et H, et l'imprimer sur `std::cout` (s'il y en a).

Compilez

Vous pouvez compiler votre code avec la commande
`mpic++ graph.cpp Isomorphism.cpp TestExercice2.cpp -o TestExercice2`

Testez

La commande
`./TestExercice2` devrait afficher

```
recherche de sous-isomorphisme entre G_1 et H_1
[0 => 2] [1 => 4] [2 => 3]
recherche de sous-isomorphisme entre G_2 et H_2
Pas de sous-isomorphisme
```

Exercice 3 : Sous-isomorphisme distribué

Nous allons maintenant paralléliser la méthode précédente. Cette parallélisation va consister à lancer le travail sur h_i et h_j aussi souvent que possible en fonction du nombre de processeurs libres. L'enjeu étant bien sûr de ne pas refaire plusieurs fois le même calcul et d'optimiser l'utilisation des processus.
Les processus seront de deux types :

1. le processus *maître* qui créera les structures nécessaires, et
2. les *serviteurs* qui feront tourner la recherche de sous-isomorphismes.

Indice : Si G est un sous-isomorphisme de H alors chaque nœud de G, sera associé à un nœud de H. On peut donc essayer toutes les façons d'associer g_0 à un noeud de H...
Vous noterez qu'un processus peut finir une étape plus rapidement qu'un autre. Pour tirer parti de cette spécificité, vous devez utiliser les versions **non-bloquantes** de `MPI_Recv` et `MPI_Send` : `MPI_Irecv` et `MPI_Isend`.

- `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`:
  Cette fonction effectue une requête de transmission des `count` élements de type `datatype` du tableau `buf` au processus `dest`.
  Après, `*request` contient le "handle" de la requête, ce qui nous permet de savoir si la transmission est terminée.
- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`:
  Cette fonction effectue une requête de réception de `count` élements de type `datatype` depuis le processus `source`, pour les mettre dans le tableau `buf`.
  Après, `*request` contient le "handle" de la requête, ce qui nous permet de savoir si la réception est terminée.
- `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`:

  Cette fonction permet de vérifier l'état d'avancement de la requête de transmission/réception associée au handle `request`.
  Après, `*flag` vaut 0 si et seulement si la requête n'est pas encore terminée.

Implémentation

L'algorithme n'étant pas altéré, vous ne devriez pas avoir à modifier `extendInjection` mais seulement la procédure `void findSubIsomorphismMPI(Graph G, Graph H)`.
Attention : on n'est pas obligé d'arrêter tous les processus lorsque l'un d'eux trouve un sous-isomorphisme ; les autres processus peuvent eventuellement trouver d'autres sous-isomorphismes (voir le test proposé ci-dessous).

Compilez

Vous pouvez compilez votre code avec la commande
`mpic++ graph.cpp Isomorphism.cpp TestExercice3.cpp -o TestExercice3`

Testez

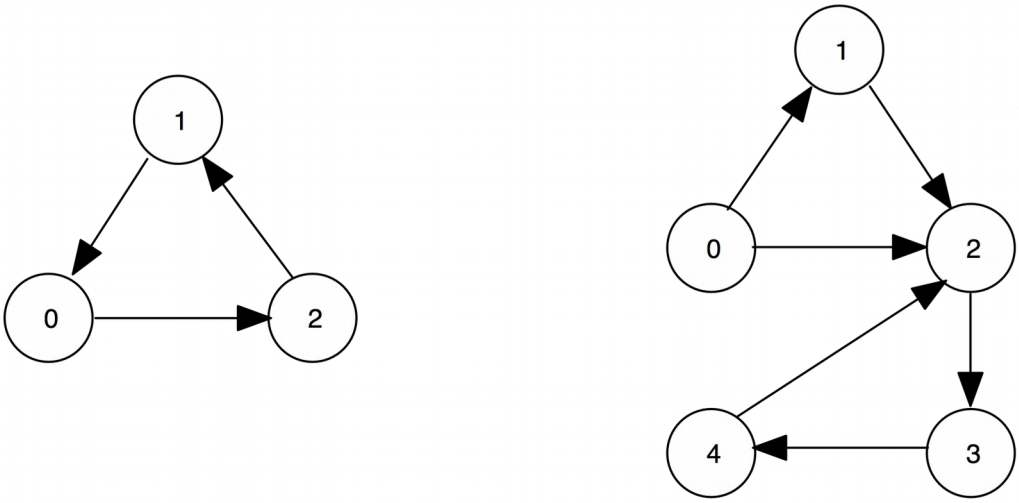

Schéma des graphes G et H utilisés pour le test.
La commande
`mpirun -np 5 TestExercice3` devrait afficher (modulo l'ordre des lignes):

```
[0 => 3] [1 => 2] [2 => 4]
[0 => 4] [1 => 3] [2 => 2]
[0 => 2] [1 => 4] [2 => 3]
```

Question supplémentaire : Comment procéder pour arrêter les calculs en cours de tous les processus lorsqu'un sous-isomorphisme a été trouvé ?

II : Problèmes de clique/stable max

Le problème de la clique maximale consiste à trouver une plus grande clique d'un graphe, i.e. un sous ensemble de sommets de cardinal maximal tels que les sommets sont tous reliés entre eux dans le graphe. Ce problème est NP-complet.

La recherche d'un stable de taille maximum dans un graphe est un autre problème classique, consistant à trouver un sous ensemble de sommets de cardinal maximal tels que les sommets ne sont jamais reliés dans le graphe.

En fait, ces deux problèmes d'optimisation sont très similaires: on passe de l'un à l'autre en considérant le graphe complémentaire. Tout algorithme de résolution d'un de ces problèmes permet de résoudre l'autre problème appliqué au graphe complémentaire. Dans la suite, nous nous intéresserons au problème du stable maximum.

On utilisera cette fois-ci des graphes non orientés. Une classe GraphNO est définie dans graphNO.hpp et graphNO.cpp . Vous pouvez compéter ces classes pour coder le problème de stable maximum.

Un algorithme de backtracking est également développé ici, suivant des questions de non optimalité, alors que précédemment, c'était la non faisabilité qui permettait d'arrêter des énumérations.

Exercice 4 : Branch&Bound séquentiel

L'algorithme présenté ici est un algorithme de Branch&Bound, algorithme d'énumération de solutions avec une borne supérieure pour couper des énumérations menant à des cas sous-optimaux.

On garde en mémoire la meilleure solution trouvée avec sa valeur O pour couper des cas non énumérables

A partir d'un ensemble partiel de sommets stables S, i.e tels qu'il n'existe pas d'arête joignant ces sommets, de cardinal s, si on note t le nombre d'éléments qui ne sont connectés à aucun des sommets de S, le stable maximal contenant S a au plus s+t éléments. Si s+t est inférieur à la meilleure solution trouvée, aucune solution optimale ne contient S et il ne sert à rien d'énumérer de tels cas.

Pour implémenter un tel algorithme d'énumération, la récursivité peut être utilisée, similairement à la fonction `extendInjection`

Coder un tel algorithme en séquentiel.

Exercice 5: Branch&Bound distribué

L'algorithme de Branch&Bound peut être parallélisé en explorant indépendamment plusieurs parties de l'arbre de Branch&Bound. Lorsqu'une meilleure solution est trouvée, la valeur O doit être diffusée à tous les processeurs, comme cela peut accélérer leurs calculs. (De telles situations permettent d'avoir des speed-ups meilleurs que du linéaire)

Schéma de parallélisation 1 : on décompose les cas suivant le plus petit indice d'un sommet du stable v, on retire tous les sommets < v du graphe partiel. Cela fait autant de calculs distribués que de sommets du graphe. Une décomposition master-worker permet au master d'affecter de nouveaux calculs à un nœud qui a terminé son travail. Cela est approprié ici comme le temps de calcul pour chaque worker n'étant pas prévisible.

Schéma de parallélisation 2 : on peut distribué plus de calculs en distinguant les cas selon les deux plus petits indice du stable v1,v2, qui ne sont pas connectés deux plus petites arrêtes. Une décomposition master-worker s'applique également, pour affecter de nouveaux calculs à un worker qui a terminé son

travail. Il y a plus de communications que le schéma 1, mais ce découpage permet a priori un meilleur équilibrage de charge

Coder avec MPI de tels schémas de décomposition.

Exercice 6: Branch&Bound distribué et équilibre de charge

Sur les deux schémas précédents, l'équilibre de charge peut être amélioré. Réindexer les points du graphe induit des parallélisation différentes. Dans quel ordre peut-on réaliser les calculs pour se rapprocher de LPT ? Un ordre aléatoire et un ordre type PLPT font ils des différences ?

Implémenter et analyser de telles décompositions.

Exercice 7: Initialisation distribuée avec un algorithme GRASP

Le fait de trouver initialement une solution de bonne qualité permet d'achever des calculs plus rapidement. Des heuristiques et méta-heuristiques peuvent être utilisées dans ce but. On implémentera ici un algorithme glouton randomisé, GRASP.

Un algorithme glouton construit une solution sommet par sommet en choisissant parmi les sommets disponibles (i.e. non voisins d'un des points de la solution partielle), celui qui a le degré minimal dans le sous graphe des sommets disponibles. Une solution est construite quand aucun sommet ne peut être rajouté au stable construit. Cela construit une solution.

Avec un algorithme GRASP, on choisit aléatoirement parmi les meilleures solutions locales (sur le critère de degré minimal). En répétant l'expérience avec plusieurs tirages aléatoires, on peut construire plusieurs solutions de bonne qualité, et potentiellement améliorer la qualité du glouton. De plus, des tirages peuvent être réalisés sur tous les processeurs disponibles.

Implémenter un algorithme GRASP distribué sur plusieurs processus. Analyser l'impact en temps de calculs d'une telle initialisation.
