Projet d'Informatique :
k shortest path and constrained shortest path

Raphaël LESCOT, Alexandre MATTON

1 Algorithme de Dijkstra

Question 1

Pour trouver le plus court chemin dans un graphe orienté, nous avons choisi d'implémenter l'algorithme de Dijkstra. Cet algorithme construit progressivement un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. Au départ, on considère que tous les sommets sont à distance infinie du sommet de départ, sauf celui-ci, qui est à une distance nulle de lui-même. Ces distances sont stockées dans un tableau d. Le sous-graphe de départ est l'ensemble vide.

Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet v de distance minimale et on l'ajoute au sous-graphe. Ensuite, on met à jour les distances des sommets voisins de v : la nouvelle distance du sommet voisin est le minimum entre sa distance stockée dans le tableau d et celle obtenue en ajoutant le poids de l'arc entre ce sommet voisin et v à la distance de v stockée dans d.

On continue ainsi jusqu'à atteindre le sommet d'arrivée, ou jusqu'à épuisement des sommets (s'il n'y a pas de chemin entre s et t).

Version naïve

Algorithme séquentiel Nous avons tout d'abord codé une première version naïve. L'information du sous-graphe mis à jour au cours de l'algorithme est contenue dans un tableau *visited*. Ce tableau sera à chaque fois parcouru en entier pour trouver le sommet voisin v le plus proche. Nous avons codé la recherche de ce sommet v dans une fonction membre auxiliaire $distanceMin()$, qui a donc une complexité linéaire en le nombre de sommet $\mathcal{O}(n)$. L'algorithme peut dans le pire de cas parcourir tous les sommets du graphe avant de tomber sur le sommet de départ : le nombre d'itérations effectuées pour la mise à jour du sous-graphe est donc un $\mathcal{O}(n)$. Au total, la complexité de cette première version naïve de l'algorithme de Dijkstra est donc quadratique en le nombre de sommets $\mathcal{O}(n^2)$. Une version plus élaborée permettra d'obtenir une complexité significativement meilleure si le graphe n'est pas complet (i.e. si m est inférieur à n^2).

Algorithme parallèle Nous avons ensuite utilisé la bibliothèque MPI pour paralléliser cette première version de l'algorithme de Dijkstra. Plus précisément, nous avons parallélisé le calcul du sommet voisin v non encore visité le plus proche de la source. Pour cela, chacun des p processeurs gère *grosso modo* n/p noeuds. Notre algorithme met à jour en conséquence la liste des voisins de chaque noeud, contenue dans *successor*. Chacun des $p-1$ processeurs envoie son sommet v de distance minimale au processeur racine, indexé par 0. Celui-ci retient ensuite, parmi ces p sommets, celui qui est réellement le plus proche du sommet source. Il *broadcaste* le noeud v obtenu à l'ensemble des processeurs. Le processeur exécute ensuite la fin de l'algorithme, qui est identique à la version séquentielle.

À chaque itération, l'algorithme
— effectue plusieurs opérations en temps constant (dont le MPI_Send et le MPI_Recv),
— le calcul du sommet le plus proche pour chaque processeur via la fonction membre $distanceMin$, qui a cette fois-ci une complexité en $\mathcal{O}(n/p)$,
— le calcul du sommet effectivement le plus proche parmi les p envoyés par les processeurs, ce qui se fait en $\mathcal{O}(p)$,
— et enfin le MPI_Broadcast de la valeur de ce sommet, qui est un $\mathcal{O}(\log p)$.

Au total, la complexité de cette version parallèle de l'algorithme de Dijkstra est donc en $\mathcal{O}\left(\frac{n^2}{p} + n * p\right)$

Version optimale

Algorithme séquentiel Nous avons aussi implémenté l'algorithme de Dijkstra qui utilise des tas afin d'avoir une meilleure complexité. Celle-ci résulte essentiellement de la complexité des opérations qui sont faites sur les tas. On retrouve les fonctions sur les tas dans la classe MinHeap. La création du tas se fait en temps linéaire en le nombre de données qu'elle contient (on remplit juste simplement un tableau), et les fonctions qui consistent à faire descendre, remonter ou insérer un objet dans un tas se font en temps logarithmique. Ainsi, dans l'algorithme séquentiel de Djikstra avec tas, la complexité finale est de $\mathcal{O}(m*\log(n))$ où m est le nombre d'arêtes et n de sommets du graphe, puisqu'à chaque fois qu'un nouveau sommet est découvert, on parcourt l'ensemble des arêtes provenant de ce sommet et on fait remonter dans le graphe les sommets aux bouts de ces arêtes si besoin est : dans le pire des cas, on parcourt toutes les arêtes du graphe, et pour chaque arête découverte on fait remonter un sommet, d'où cette complexité.

Algorithme parallèle Pour la parallélisation de cet algorithme, on donne à chaque processeur un certain nombre de noeuds à gérer (dans l'idéal, chaque processeur gère $\frac{n}{p}$ noeuds). Ensuite, chaque processeur p modifie les listes des successeurs de chaque sommet pour qu'elles ne contiennent que les noeuds gérés par lui-même, afin d'améliorer la complexité (cette étape se fait donc en $\mathcal{O}(m)$. Puis, chaque processeur p construit un tas uniquement formé de ses noeuds.
A chaque tour de boucle, le root demande à tous les processeurs leur meilleur sommet (c'est-à-dire celui dont la distance à l'origine calculée en fonction des sommets déjà découverts est la plus faible). Il prend alors le meilleur de ces meilleurs sommets (en $\mathcal{O}(p)$), celui-ci étant par conséquent le sommet le plus proche de l'origine. Il le découvre, retransmet les informations à tous les autres processeurs ($\mathcal{O}(p)$), et le processeur qui devait gérer ce sommet le supprime de son tas. Enfin, chaque processeur met à jour son tas en remontant les successeurs du sommet qui vient d'être découvert. Chaque processeur p peut donc faire remonter au plus $\sum_{\text{v géré par p}} \text{nombre de predecesseurs de v}$ fois des sommets au cours de l'algorithme, chaque fois avec une complexité d'environ $\mathcal{O}(\frac{n}{p})$. La boucle fait dans le pire des cas n tours (si le noeud qu'on veut atteindre est atteint en dernier), la complexité finale est donc :

$$\mathcal{O}\left(m + n * p + \max_{\text{processeur p}}\left(\sum_{\text{v géré par p}} \text{nombre de prédécesseurs de v}\right) * \log\left(\frac{n}{p}\right)\right)$$

Question 2

Algorithme séquentiel La littérature disponible sur la toile donne plusieurs moyens d'implémenter l'algorithme du plus courts chemins avec contraintes, une grande partie de ceux-ci reposant sur des calculs de lagrangien. Nous avons décidé de chercher nous-même comment faire un algorithme plus dans l'esprit de ce que nous avons vu précédemment, et qui puisse être parallélisable.

L'idée principale est d'utiliser la programmation dynamique pour faire un algorithme de type Bellman-Ford, mais avec 2 dimensions : on note d[j][w][k] la distance calculée entre l'origine et le sommet j, qui correspond à un chemin avec un poids qui vaut exactement w, et qui est atteignable en traversant moins de k arêtes. Alors on a

l'égalité suivante, où $d_{i->j}$ est la distance entre les sommets i et j et qui vaut 0 si i=j, et $w_{i->j}$ est le poids de l'arête entre i et j, qui vaut aussi 0 si i=j :

$$c[j][w][k] = min_{i \in V \ voisins \ ou \ egal \ a \ j}(d[i][w - w_{i->j}][k-1] + d_{i->j})$$

On calcule à chaque tour k les $d[j][w][k]$ pour tous les poids possibles w inférieurs au poids maximal donné par l'énoncé. Dans le pire des cas, le chemin final est constitué de n arêtes, donc on arrête les calculs lorsque k vaut n, et on cherche pour quel w on trouve la plus faible distance parmi les d[destination][w][k].

On obtient ainsi une complexité pseudo-polynomiale, puisque dans chacune des n boucles, on utilise dans les calculs une fois chaque arête pour tout w d'où une complexité finale de l'ordre de $\mathcal{O}(n * m * w)$

Algorithme parallèle La parallélisation de cet algorithme se fait de la manière suivante :
On attribue à chaque processeur p un nombre $\frac{m}{p}$ d'arêtes. Pour chacune de ces arêtes et à chaque tour, celui-ci met à jour la nouvelle valeur de la distance du sommet à l'extrémité de l'arête, Puis via des ReduceAll, Broadcast et Send/recv, chaque d[v][k][w] (pour tout v et w), est calculé en trouvant le minimum des valeurs calculées par chaque processeur, puis partagé entre tous les processeurs.
En considérant que les Recv/Send sont des opérations unitaires en complexité, et que les Reduce et Broadcast sont au plus linéaires (donc prennent moins de temps que p communications de type Recv/Send), on obtient qu'à chaque boucle on fait environ $n * w * p$ opérations pour cette partie-là, la partie précédente correspondant à mettre à jour les nouvelles distances prenant un temps de l'ordre de $\mathcal{O}(n * w + \frac{m}{p} * w)$ opérations pour chaque processeurs. D'où une complexité finale de :

$$\mathcal{O}((n * w + \frac{m}{p} * w + n * w * p) * n) = \mathcal{O}((\frac{m}{p} + n * p) * n * w)$$

Le résultat est donc le plus probant lorsque $p = \sqrt{\frac{m}{n}}$, la complexité étant alors : $\mathcal{O}(\sqrt{m * n} * n * w)$

Question 3

Algorithme séquentiel Pour calculer les k chemins les plus courts d'un graphe, nous avons implémenté une version de l'algorithme de Yen. Son fonctionnement est le suivant :

- Le chemin le plus court est donné par l'algorithme de Dijkstra. Il est le premier élément d'un vecteur $shortestPaths$ destiné à contenir les k chemins les plus courts.
- Soit p le dernier chemin trouvé par l'algorithme, qui est donc le dernier élément de $shortestPaths$. Notons le $p = (1) - (2) - ... - (l)$. Pour chacun des $l - 1$ premiers sommets (j) de p, l'algorithme calcule un chemin q, qui est une déviation de p à partir de (j).
  - La première partie $rootPath$ du chemin q est identique à celle de p : ce sont les sommets de p jusqu'à (j) ;
  - Pour trouver la seconde partie, i.e. les sommets jusqu'à t, le graphe est mis à jour. Il ne faut en effet pas obtenir les mêmes sommets que ceux de p. Pour cela, sont supprimés de G d'une part les $j - 1$ sommets de p avant (j), et d'autre part, les arêtes $(j) - (j + 1)$ précédemment utilisées : en particulier celle du chemin p, mais aussi les arêtes $(j) - (j + 1)$ des chemins déjà ajoutés à $shortestPaths$ qui partagent le même $rootPath$ que p. On applique l'algorithme de Dijkstra au graphe ainsi obtenu, pour récupérer la seconde partie $spurPath$ de q.
  - q est donc la concaténation de $rootPath$ et $spurPath$
- On ajoute enfin à $shortestPaths$ le chemin q le plus court parmi les $l - 1$ obtenus
- On réitère ensuite cette procédure sur le chemin q, jusqu'à obtenir k chemins - ou moins, s'il en existe moins de k.

Pour chacune des k itérations, l'algorithme fait des appels à la fonction Dijkstra sur presque tout le graphe, ce qui a une complexité $\mathcal{O}(m * \log n)$. Le nombre de ces appels est à chaque fois égal à $l - 1$, où l est la longueur du chemin q calculé précédemment. Toutes ces longueurs l sont des $\mathcal{O}(n)$. La complexité de l'algorithme de Yen séquentiel est donc $\mathcal{O}(k * m * n * \log n)$.

Algorithme parallèle La parallélisation se fait dès le début de l'algorithme, pour chacun des k tours de boucle : chacun des p processeurs gère l/p noeuds, où l est la longueur du chemin q calculé précédemment, qui est le dernier élément de $shortestPaths$. Il calcule donc l/p candidats q, et envoie le plus court au processeur racine. Celui-ci ajoute enfin à $shortestPaths$ le chemin le plus court parmi les p qu'il a reçus.

Au cours de chacune des k itérations, l'algorithme :
— effectue plusieurs opérations en temp constant, dont les MPI_Send et MPI_Recv
— fait l/p appels à l'algoritme de Dijkstra, où l est un $\mathcal{O}(n)$
— et le processeur racine choisit le plus court chemin parmi les p candidats qui lui ont été envoyés, ce qui se fait en $\mathcal{O}(p * n)$

La complexité de la version parallèle de l'algorithme des k plus courts chemins est donc un $\mathcal{O}\left(m * \log n * \frac{n}{p} + n * p\right)$