École Nationale Supérieure de l'Informatique pour l'Industrie et l'Entreprise

PRFO 2023-2024

Rapport : Programmation fonctionnelle

GOELLER Karim

Rapport : Programmation fonctionnelle

GOELLER Karim

January 2, 2024

Contents

1 Introduction

Ce projet a pour finalité d'aider M. Forest à maximiser le flot de café entre le bar de l'ENSIIE et son bureau. Entre le bureau de M. Forest et le bar, il y a des salles et des halls, qui sont reliés entre eux par des couloirs de différentes tailles.

La maximisation se fera en deux phases distinctes :

- Phase 1 : on cherchera le chemin le plus rapide (le plus court) entre le bureau et le bar
- Phase 2 : On prendra en compte la capacité réelle des couloirs, on cherchera à maximiser le flot sortant du bar et arrivant au bureau.

2 Phase 1

Il nous faut implémenter différentes structures et fonctions pour réussir cette phase. Nous modéliserons l'école par un graphe orienté non pondéré pour la phase 1 (sans prendre en compte leur capacité pour l'instant). Les salles et les halls seront les sommets, reliés entre eux par des couloirs qui constitueront les arêtes. Pour trouver tous les chemins les plus courts, nous utiliserons un parcours en largeur. Dans cette fonction, nous nous servirons d'une file d'attente dont j'expliciterai la structure.

2.1 le type graphe

Pour cette partie, nous n'aurons pas besoin d'un graphe pondéré, mais cela sera nécessaire dans la phase 2. J'utiliserai donc la même structure de graphe pour les deux parties. Cependant, dans la phase 1, toutes les pondérations seront fixées à zéro et n'interviendront pas dans les étapes du processus.
Pour implémenter les graphes, nous utiliserons un dictionnaire (une Map) dont les clés seront les sommets du graphe et la valeur associée à chaque clé sera un autre dictionnaire. Dans ce second dictionnaire, les clés seront un sommet successeur et la valeur associée sera un doublet d'entiers, que j'expliciterai dans la phase 2. Pour l'instant, ils seront initialisés à (0, 0).
en Ocaml c'est le type : `type graph = ((int*int) NodeMap.t) NodeMap.t`

2.2 file d'attente

Un parcour en largeur nécessite de de connaitre qu'elle est le noeud courant, c'est quoi le chemin que l'on a parcourus pour arrivé ici. Cette méthode garantit que les nœuds sont visités de manière systématique, en commençant par ceux qui sont les plus proches du point de départ et en progressant ensuite vers les nœuds plus éloignés. Pour implémenter cette file d'attente, nous utiliserons une FIFO (First In, First Out). Dans notre cas, elle consistera en une liste de doublets, où le premier élément est le nœud courant et le second représente le chemin parcouru jusqu'à ce nœud. Grâce à cette structure FIFO, dès qu'un nœud est visité,

tous ses voisins non visités sont ajoutés à la file, assurant ainsi une exploration uniforme et complète du graphe. On arrête de parcourir le graphe lorsque tous les sommets ont été visités ou que la destination a été trouvée.

2.3 Parcour en largeur

Dans le cadre de la première phase de notre projet, nous avons choisi d'utiliser un parcours en largeur (BFS) pour identifier les plus courts chemins entre le bureau de M. Forest et le bar. Cet algorithme débute à partir du nœud source, ici le bureau, et s'étend progressivement en explorant systématiquement tous les nœuds adjacents. Chaque successeur non visité du nœud courant est ajouté à la file d'attente, avec une mise à jour du chemin parcouru pour y parvenir, assurant ainsi que les nœuds sont abordés selon l'ordre de leur découverte.

Lorsqu'un nœud atteint le point de destination, à savoir le bar, l'algorithme évalue le chemin utilisé. Si ce dernier s'avère être plus court que les précédents chemins identifiés, il est alors enregistré comme l'un des chemins les plus courts. L'algorithme continue son exploration jusqu'à avoir couvert tous les chemins possibles. Cette approche garantit une exploration complète et efficace.

3 Phase 2

Lors de la Phase 2, nous prenons en compte la capacité des différents couloirs, ce qui implique l'utilisation de pondérations. La pondération de chaque couloir est représentée par un doublet défini comme suit :

- Premier élément : il représente le flux actuellement en transit dans le couloir. Initialement, pour chaque couloir, cette valeur est fixée à 0 au démarrage de l'algorithme.
- Deuxième élément : il indique le flux maximal que le couloir peut accueillir. Cette valeur reste constante tout au long de l'exécution de l'algorithme.

Pour résoudre ce problème, nous utiliserons l'algorithme de Dinic, qui s'appuie sur la création de graphes intermédiaires tels que le graphe résiduel et le graphe de niveau. Ces structures permettent à l'algorithme de Dinic d'optimiser la recherche de chemins et de maximiser efficacement le flux à travers le réseau de couloir de l'ENSIIE.

3.1 Graphe Résiduel

Dans le cadre de l'algorithme de Dinic, le graphe résiduel joue un rôle crucial. Il représente le graphe original mais avec des capacités de flux ajustées en fonction des flux déjà présents dans le graphe. Dans ce graphe, chaque arête possède une capacité résiduelle, qui correspond à la différence entre la capacité maximale de l'arête et le flux actuellement en transit. Ce graphe résiduel est constamment mis à jour à chaque itération de l'algorithme, reflétant ainsi

les changements dans les flux à travers le réseau de couloirs. Si une arête (u,v) existe dans le graphe avec un flux f et une capacité c, dans le graphe résiduel, si (c-f) > 0, alors l'arête (u,v) sera aussi présente avec un flux (c-f) et (v,u) sera reliée par une arête avec un flux f.
Pour le construire, on effectue un `fold_edge` pour parcourir toutes les arêtes du graphe. Puis, on vérifie la capacité résiduelle de chaque arête ; si elle est strictement positive, on ajoute les arêtes correspondantes au graphe résiduel.

3.2 Graphe de Niveau

Une composante clé dans l'implémentation de l'algorithme de Dinic est la construction du graphe de niveau. Ce graphe, dérivé du graphe résiduel, est essentiel pour identifier efficacement les chemins augmentants. Chaque arête du graphe de niveau connecte uniquement des nœuds dont les niveaux sont consécutifs.
La construction du graphe de niveau débute par l'initialisation d'une carte des niveaux, c'est un dictionnaire où chaque noeud du graphe est associé à son niveau, initialisé avec une valeur de niveau de -1, signifiant qu'ils ne sont pas encore visités, est mis à chaque noeud. Ensuite, un parcours en largeur est effectué à partir du nœud source. À chaque étape, le niveau du nœud courant est mis à jour dans la carte des niveaux, assurant que chaque nœud est visité une seule fois et que son niveau le plus bas possible est enregistré.
Pour chaque nœud visité, ses voisins dans le graphe résiduel sont examinés. Si un voisin n'a pas encore été visité ou si son niveau actuel peut être réduit, il est ajouté à la file d'attente du BFS avec un niveau incrémenté. Cette approche garantit que seuls les chemins valides et augmentants sont pris en compte, car elle élimine les chemins qui reviennent en arrière ou qui ne contribuent pas à l'augmentation du flux.
Une fois le parcours en largeur terminé, le graphe de niveau est construit en ajoutant des arêtes entre les nœuds dont les niveaux sont adjacents et pour lesquels il existe un flux résiduel positif dans le graphe résiduel. Ainsi, le graphe de niveau résultant fournit un moyen efficace de naviguer dans le réseau, en concentrant la recherche sur les chemins qui sont les plus susceptibles d'augmenter le flux global. Cet outil est indispensable pour l'optimisation de l'algorithme de Dinic et joue un rôle important dans la maximisation du flux.

3.3 Détermination et Ajout du Flot Bloquant

L'étape de détermination et d'ajout du flot bloquant est cruciale dans l'algorithme de Dinic. Cette phase commence par identifier le flot bloquant sur un chemin augmentant dans le graphe de niveau. On utilise une fonction récursive pour traverser le chemin et trouver le flux le plus faible, qui est le flot bloquant.
Une fois ce flot identifié, le graphe est mis à jour pour refléter le nouveau flux. Cela implique d'augmenter le flux sur chaque arête du chemin dans la direction du flux bloquant, tout en ajustant le flux opposé pour maintenir l'équilibre. Cette mise à jour est essentielle pour assurer l'exactitude du graphe résiduel, qui reflète les capacités de flux restantes après chaque

itération.
Ce processus d'ajustement du flux est fondamental pour optimiser l'utilisation des capacités de flux disponibles dans le réseau et pour garantir que chaque chemin augmentant contribue efficacement à l'accroissement global du flux.

3.4 Algorithme de Dinic

L'algorithme de Dinic, en utilisant les graphes résiduels et de niveau, optimise la recherche de chemins augmentants et la mise à jour des flux. À chaque itération, il construit un graphe de niveau, trouve les chemins augmentants et les flots bloquants, et met à jour le graphe résiduel en conséquence. Cette méthode itérative est répétée jusqu'à ce qu'aucun chemin augmentant ne soit trouvé dans le graphe de niveau, indiquant que le flux maximal a été atteint.

Ces étapes de l'algorithme de Dinic illustrent une approche efficace pour maximiser le flux de café entre le bar et le bureau de M. Forest, en prenant en compte les capacités variables des couloirs de l'ENSIIE.

4 Conclusion

Pour s'assurer que mon implémentation répond aux exigences, il serait pertinent de la tester avec une série de graphes standards. L'objectif serait de contrôler sa performance en termes de consommation de ressources et de rapidité d'exécution. Concrètement, cela implique de soumettre le programme à divers cas de graphes représentatifs, afin d'évaluer son efficacité en termes de mémoire et de temps de traitement. Cette démarche permettrait de confirmer la viabilité de l'implémentation dans des contextes pratiques et d'identifier les éventuelles optimisations nécessaires.

La complexité temporelle et en mémoire de mon implémentation pourrait poser problème. Bien que le projet ne dispose pas d'un cahier des charges précis, il est possible que j'ai dépassé les limites envisageables. En ce qui concerne la complexité temporelle, j'aurais pu améliorer la recherche du flot bloquant. Dans mon implémentation actuelle, on cherche un chemin de la source au puits dans le graphe de niveau, puis on actualise le graphe résiduel avec ce seul chemin. Une piste d'optimisation serait de considérer comme flot bloquant l'ensemble de tous les chemins les plus courts trouvés dans le graphe de niveau pour mettre à jour le graphe résiduel. Une autre piste concerne la construction du graphe de niveau : actuellement, je parcours le graphe en largeur, je stocke les niveaux dans un dictionnaire, puis je ne relie que ceux séparés par des niveaux adjacents. Pour améliorer la complexité temporelle, on pourrait envisager de construire le graphe de niveau directement pendant le parcours en largeur, sans recourir à un dictionnaire intermédiaire.

Pour mieux respecter les contraintes liées à la complexité en mémoire, il serait judicieux

de minimiser l'utilisation de structures intermédiaires. Par exemple, stocker les successeurs dans un Set ou conserver les niveaux pour la construction du graphe de niveau dans un dictionnaire intermédiaire peut alourdir significativement le programme. En simplifiant ces aspects, par exemple en intégrant directement ces informations dans la structure principale du graphe ou en optimisant la manière dont ces données sont stockées et traitées, il est possible de rendre le programme plus léger et efficient en termes d'utilisation de la mémoire.

Pour optimiser le projet, ...