Projet de Théorie des Graphes

Mehrad RAFII - Robin XAMBILI

March 10, 2018

1 Introduction

Ce projet consiste à apparier des squelettes afin de faire de la reconnaissance d'objets. Un squelette est un modèle qui permet de représenter un objet en 2D ou en 3D. On associe donc à chaque squelette un graphe sans cycle, i.e. un arbre, et il s'agit donc d'estimer la distance d'édition entre deux graphes G_1 et G_2 qui représentent chacun un squelette.

2 Sections 1 à 3

2.1 Opérations sur les graphes

On commence d'abord par implémenter des opérations basiques sur des graphes qui nous serviront à estimer la distance d'édition ultérieurement.

2.1.1 Associate/Separate

La fonction *associate* met 2 noeuds v_1 et v_2 des deux graphes en correspondance, en les marquant de la façon suivante :
$v_1 < -label(v_2)$
$v_2 < -label(v_1)$

Pour que cette opération soit réversible, il faut coder l'opération inverse, i.e. *separate*, qui réinitialise la marque de chaque sommet à 0.

2.1.2 Contract/Insert

L'opération *contract* contracte deux arêtes o et a dans un graphe, de la façon suivante :

- Retrait de l'arête (o,a)
- Pour tout voisin s de a (autre que o), retrait de l'arête (a,s) et création de l'arête (o,s)
- Retrait du sommet a

Pour que cette opération soit elle aussi réversible, il faut fournir en sortie de cette fonction la liste des voisins de l'arête a (autres que o). Cette liste des voisins nous permet alors d'implémenter l'opération réciproque *Insert*, qui insère l'arête (o,a) dans le graphe, en recréant les arêtes entre a et ses voisins de façon symétrique à la fonction *contract*.

2.2 Test d'égalité de deux arbres

Nous implémentons maintenant une fonction qui teste l'égalité de deux arbres donnés en entrée. Il s'agit en fait d'une fonction *equals* qui fait appel à une fonction auxiliaire récursive *equals-aux*. L'algorithme de base nous est fourni.

Le principe de cet algorithme est d'apparier deux sommets dans les deux graphes à comparer (ce qui revient, en des termes intuitifs, à les "disposer" de la même façon sur un papier), puis d'associer récursivement les sous-arbres gauche, puis droit de chaque graphe. Cela se fait en considérant à chaque étape la liste ordonnée des successeurs non marqués de chacun des sommets "racines". Si l'on arrive à marquer les sommets des deux graphes jusqu'à avoir parcouru l'intégralité des deux graphes, alors cela signifie que les deux graphes sont égaux.
En effet, les cas d'arrêt de la fonction *equals-aux* sont au nombre de 2 :

- Soit on s'arrête lorsque les deux listes de successeurs non marqués sont vides, auquel cas les deux graphes sont nécessairement égaux
- Soit on s'arrête lorsque une et une seule des deux listes est vide, auquel cas on conclut que les deux graphes sont différents.

Enfin, nous avons dû implémenter la fonction *unmarked* qui à partir d'une liste de sommets renvoie la liste des sommets non marqués. Cette fonction est utilisée dans la fonction *equals-aux* et son implémentation est simplement basée sur l'utilisation d'un fold-right.

2.3 Évaluation de la distance d'édition

On s'intéresse maintenant à la fonction essentielle de ce projet, qui est la fonction *distance*, qui calcule la distance d'édition entre deux graphes, i.e. le nombre d'opérations de contraction/insertion d'arêtes minimal nécessaire pour passer d'un graphe à l'autre.
Il nous est conseillé de nous inspirer de la fonction *equals* : en effet, dans ceette fonction, l'opération que l'on effectue à chaque étape est le marquage de deux sommets. Ici , on va compléter cet algorithme en ajoutant deux opérations : contracter une arête dans le premier graphe, contracter une arête dans le deuxième graphe. On a ainsi le schéma suivant :

1. On associe les deux sommets de départ, et on calcule la distance d'édition entre les deux graphes obtenus, puis on sépare les deux sommets.
2. On contracte une arête dans le premier graphe, puis on calcule la distance d'édition entre les deux graphes obtenus, enfin on réinsère l'arête dans le premier graphe.
3. On contracte une arête dans le deuxième graphe, puis on calcule la distance d'édition entre les deux graphes obtenus, enfin on réinsère l'arête dans le deuxième graphe.

Une fois cela fait, on a alors 3 distances, et la distance d'édition entre nos deux graphes initiaux est égale au minimum de ces 3 distances.

Remarques :

- Lors de l'opération d'association, on appelle en fait 2 fois la fonction récursive *equals-aux* : 1 fois sur les graphes de racines h_1 et h_2 (i.e. les sommets que l'on vient de marquer), et 1 fois sur les graphes de racines v_1 etv_2 (i.e. les racines initiales). On obtient donc deux distances c_h et c_q, et la distance d'édition correspondant à la branche "associate" est égale à $c_h + c_q$.
- A chaque appel de la fonction *distance-aux*, on calcule aussi les 3 listes que la fonction doit retourner (liste des sommets appariés, liste des arêtes contractées dans le premier graphe, liste des arêtes contractées dans le deuxième graphe).

3 Accélération de l'algorithme

L'algorithme que l'on vient d'utiliser est en fait assez catastrophique en terme de complexité. En effet, on se rend compte en effectuant les tests que les temps d'exécution sont souvent très élevés (de l'ordre de plusieurs dizaines de secondes), et le nombre d'appels récursifs pour le jeu de tests fourni dépasse le million.
Cela s'explique par la complexité exponentielle de l'algorithme : à chaque étape de la fonction récursive, on effectue une opération, on fait un appel récursif, puis on effectue une deuxième opération, puis on fait un appel récursif, puis on effectue une troisième opération et on fait un appel récursif. Cela veut dire que quoi qu'il arrive, on testera toutes les opérations possibles à

chaque étape de l'algorithme, sans même se soucier de leur pertinence. Ainsi, imaginons que la première branche d'associations ait renvoyé une distance d'édition de zéro (i.e. les graphes sont égaux), on va tout de même effectuer toutes les contractions sur les deux graphes puis comparer les 3 distances obtenues, alors même qu'on connaît déjà le résultat à l'avance, d'où le très grand nombre d'appels récursifs inutiles.
Pour solutionner ce problème, on modifie donc la fonction *distance-aux*, de façon à ne plus explorer les branches de l'arbre des appels récursifs si celles-ci renvoient une distance d'édition supérieure aux branches précédentes :

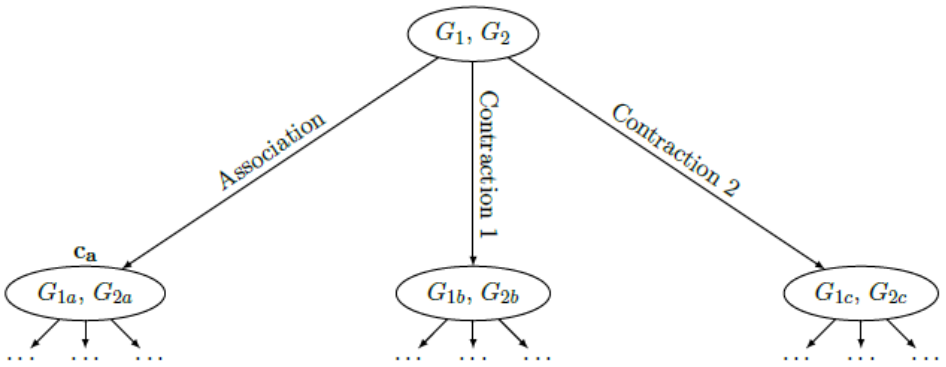

Figure 1: Arbre des appels récursifs de l'algorithme

D'un point de vue algorithmique, les modifications que l'on a apportées sont :

- L'ajout de deux paramètres dans la fonction récursive auxiliaire : un paramètre *nb-edit* qui correspond au nombre d'éditions effectuées dans la branche courante ; et un tuple *min-edit* qui contient notamment le nombre minimal d'éditions effectuées jusqu'à présent dans l'algorithme.
- L'ajout d'une condition sur le nombre d'éditions courant
- La mise à jour du nombre minimal d'éditions au cours de l'algorithme

Ainsi, à chaque appel récursif de la fonction auxiliaire, on ajoute 1 au paramètre *nb-edit*. Ensuite, l'ensemble des opérations auxiliaires n'est effectué que si le paramètre *nb-edit* est strictement inférieur au nombre minimal d'éditions *c-min* ; dans le cas contraire, on retourne directement le tuple *min-edit*, ce qui coupe immédiatement la branche courante de l'arbre ci-dessus, et qui évite les appels récursifs non pertinents. Enfin, il faut également mettre à jour le paramètre *min-edit* au cours de l'algorithme. Ce tuple contient 4 valeurs : le nombre minimal d'éditions *c-min*, ainsi que les 3 listes *l0-min, l1-min, l2-min* correspondant à ce nombre minimal d'éditions. Avant chaque opération, on met à jour la valeur de *min-edit* en comparant la valeur *c-min* avec le nombre d'éditions de la branche que l'on vient de parcourir :

```
(* Association *)
associate h1 h2;
let (ch,lh0,lh1,lh2) = distance_aux_opti g1 h1 g2 h2 nb_edit min_edit in
let (cq,lq0,lq1,lq2) = distance_aux_opti g1 v1 g2 v2 nb_edit min_edit in
separate h1 h2;

let new_min =
    if (ch+cq < c_min) then
        (ch+cq, ((h1,h2)::lh0)@lq0, lh1@lq1, lh2@lq2)
    else min_edit in


(* Contraction de g1 (v1,h1)*)
let (cg1,lg1_0,lg1_1,lg1_2) = contract_g1_opti g1 v1 g2 v2 h1 nb_edit new_min in
```

Figure 2: Mise à jour de *min-edit*

Enfin, on retourne alors le minimum des 3 distances obtenues à partir de chaque branche du graphe, comme précédemment.

Remarque : Lors du premier appel de la fonction récursive auxiliaire, il faut fournir une valeur du minimum d'éditions courant (i.e. la valeur seuil au delà de laquelle on arrêtera les appels récursifs). Nous avons choisi de positionner cette valeur à $2 * nbaretes(g_1) + 1$.

4 Tests comparatifs

Nous avons écrit un fichier de tests qui contient d'abord les tests unitaires des opérations élémentaires sur les graphes. Puis nous avons ajouté des tests sur les fonctions *distance* et *distance-opti* afin de comparer leurs performances :

```
(* Tests distance et distance_opti *)
let t1 = Sys.time ();;
let (c1, l1_0, l1_1, l1_2) = distance obj1_1 obj1_1v5 obj1_3 obj1_3v3;;
let t2 = Sys.time ();;
t2-.t1;;

c1 == 2;;

let t1 = Sys.time ();;
let (c1, l1_0, l1_1, l1_2) = distance_opti obj1_1 obj1_1v5 obj1_3 obj1_3v3;;
let t2 = Sys.time ();;
t2-.t1;;

c1 == 2;;
```

Figure 3: Tests de performance

```
- : float = 30.5840000000000174
- : bool = true
```

Figure 4: Distance non optimisée

```
- : float = 0.019999999999818101
- : bool = true
```

Figure 5: Distance optimisée

On voit que sur cet exemple, on passe d'un temps de plus de 30 secondes à un temps de moins de 2 centièmes de seconde. Enfin, en lançant le jeu de tests fourni, on obtient les résultats suivants :

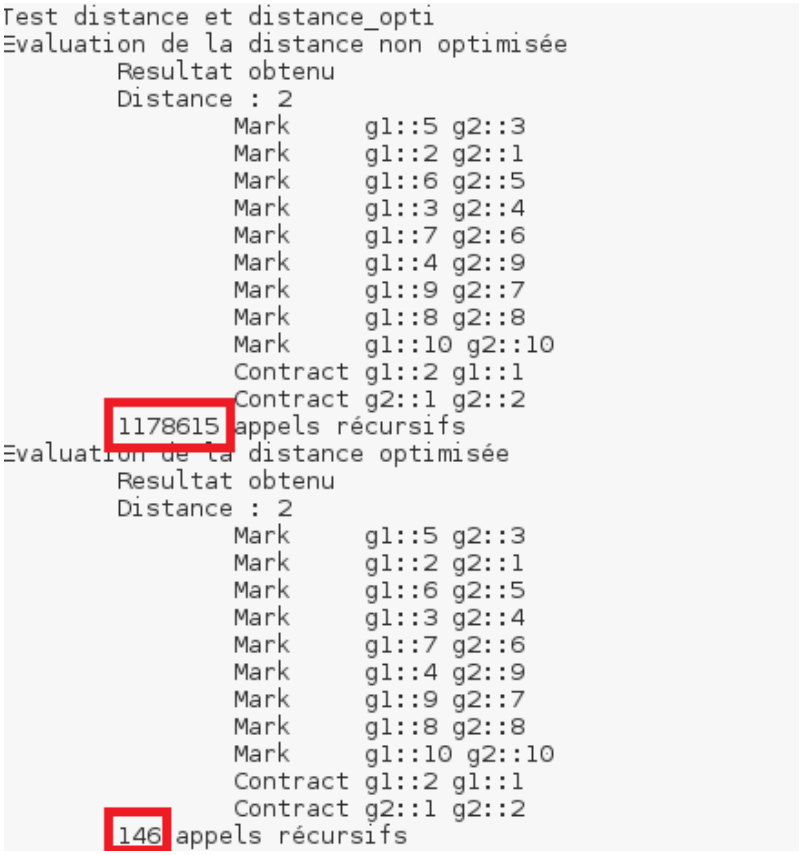

Figure 6: Résultats du jeu de tests

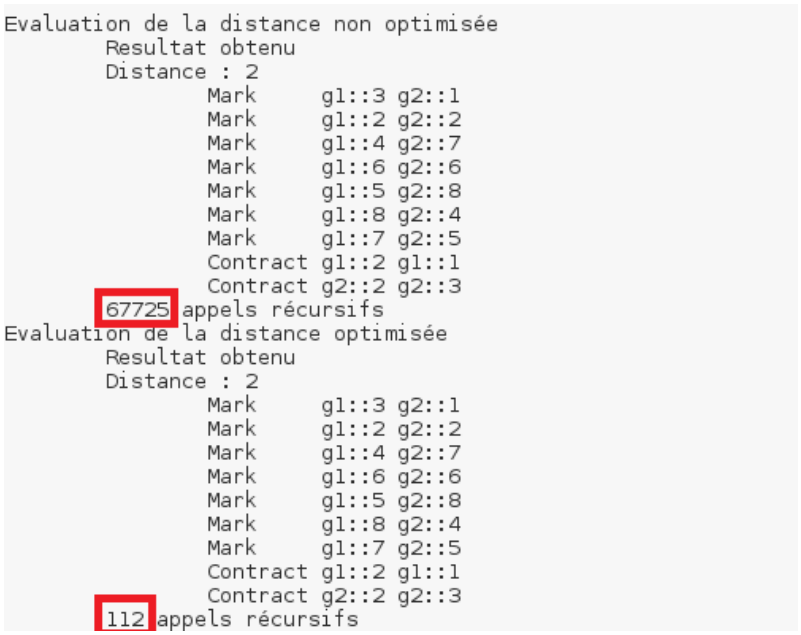

Figure 7: Résultats du jeu de tests

On note là aussi une amélioration très significative puisque l'on divise le nombre d'appels récursifs par 100 voire par 1000 selon les cas.

5 Conclusion

Ce travail a été pour nous une bonne occasion de se confronter à un projet de programmation fonctionnelle, avec les problèmes classiques de complexité des algorithmes et de temps de calcul. On a aussi pu constater une nouvelle fois que la solution à un problème complexe nécessite souvent plusieurs heures de réflexion pour aboutir à des modifications de moins de dix lignes de code (comme pour l'algorithme de distance optimisée par exemple), d'où l'importance de bien comprendre comment fonctionne l'algorithme.
En ce qui concerne l'organisation, les trois premières sections se prêtaient à une parallélisation du travail, c'est ce que nous avons fait, notamment pour coder les opérations et leurs réciproques. Pour la partie 4, nous avons surtout réfléchi en binôme sur une même machine jusqu'à obtenir des résultats concluants.