TP 17 - diff

5 juillet 2023

1 diff

L'outil `diff` permet de visualiser les changements entre deux fichiers texte. Il est en particulier très utilisé par les systèmes de versionnement de fichiers comme `git` qui décrivent un fichier comme une succession des modifications à partir d'un état initial. Cette approche permet de garder en mémoire les versions successives du fichier (son historique), et est très pratique si on veut revenir en arrière sur des modifications passées. C'est l'usage de `diff` qui permet de déterminer quelles sont les modifications qui permettent de passer d'une version à la suivante. Vous pouvez utiliser `diff` en ligne de commande avec la syntaxe suivante : `diff -u <fichier_1> <fichier_2>`.

1. Testez `diff` sur deux fichiers de votre choix ne différant que très peu.

2 Tables de hachage

Dans cette section on s'intéresse à l'implémentation d'une table de hachage pour servir de dictionnaire dans la suite. Une table de hachage consiste en deux éléments principaux : un tableau contenant les couples clé/valeurs contenus dans le dictionnaire et une fonction de hachage qui indique dans quelle case placer un couple clé/valeur. Dans ce TP les collisions seront gérées à l'aide de listes : une case du tableau contiendra une liste des couples clé/valeurs dont l'image est identique par la fonction de hachage. On utilise donc le type suivant pour une fonction de hash dont les clés sont de type `'a` et les valeurs de type `'b` :

```
type ('a, 'b) hash_table = {data: ('a * 'b) list array; hash_function : 'a -> int}
```

`data` est le tableau contenant les couples clé/valeur et `hash_function` est la fonction de hachage à utiliser pour déterminer la case associée à une clé. On attend d'une table de hachage l'interface suivante :

- `init_hash_table : int -> ('a -> int) -> ('a,'b) hash_table` qui initialise une table à partir de la taille du tableau et de la fonction de hachage.
- `member : 'a -> ('a, 'b) hash_table -> bool` qui détermine l'existence d'une valeur associée à une clé dans une table.
- `get_value : 'a -> ('a, 'b) hash_table -> 'b option` qui renvoie la valeur associée à la clé dans la table si elle existe via un type option.
- `add_value : 'a -> 'b -> ('a, 'b) hash_table -> unit` qui ajoute un couple clé/valeur dans la table. S'il existe déjà un élément associé à la clé, la fonction lève une exception
- `remove_element : 'a -> ('a, 'b) hash_table -> unit` qui supprime un couple associé à une clé dans la table. S'il n'y a aucun élément associé à cette clé dans la table, la fonction lève une exception

2. Implémentez les fonctions de l'interface.
3. (Bonus) Implémentez une fonction `iterate: ('a,'b) hash_table ->('a,'b) list` qui renvoie la liste de tous les couples clé valeurs présents dans la table.
4. (Bonus) Implémentez une fonction `replace` qui remplace la valeur associée à une clé dans une table, et l'ajoute si aucune valeur n'était associée à cette clé.
5. (Bonus) Implémentez un mécanisme de redimensionnement de la table quand celle-ci contient trop ou trop peu d'éléments. Vous préciserez les conditions de déclenchement de ce mécanisme et prendrez soin de modifier la fonction de hachage en conséquence.
6. On souhaite tester ces fonctions avec une table dont les clés sont des `int` et les valeurs sont des `int`. Le code suivant implémente une fonction de hachage qui ramène la clé dans les indices valides du tableau via un modulo. Le double modulo permet de gérer les nombre négatifs et `size` est le nombre de cases de la table. Testez les fonctions de l'interface de la table.

```
1 let simple_hash_fn size key = ((key mod size) + size) mod size
```

On souhaite générer de manière pseudo aléatoire des fonctions de hachage pour initialiser une table afin que deux clés de même modulo ne soient pas nécessairement hachées de la même manière et que chaque table n'utilise pas nécessairement la même fonction de hachage. On se propose donc de choisir une fonction de hachage de manière aléatoire dans un ensemble de fonctions de hachage. On se fixe un entier premier p plus grand que la taille de la table. On note $H =$

$\{h_{a,b} | (a,b) \in \mathbb{Z}/p\mathbb{Z},\ a > 0\}$ où $h_{a,b} = x \mapsto (ax+b)[p]$ et $\mathbb{Z}/p\mathbb{Z}$ est l'ensemble $\mathbb{Z}$ quotienté par la relation d'équivalence "égal modulo p".
Choisir une fonction au hasard dans H revient donc à choisir au hasard un couple a, b qui convient.

Remarque: Cette famille de fonction de hachage intervient dans la méthode de construction de tables de hachage appelée *hachage parfait* et qui garantit un temps d'accès moyen constant pour l'ajout, la suppression et la récupération des éléments d'un dictionnaire. Cette méthode demande cependant de savoir à l'avoir quelles seront les clés qui seront utilisées lors de l'utilisation de la table.

Vous pourrez obtenir un nombre pseudo aléatoire via la fonction `Random.int : int -> int` qui donne un entier entre 0 et l'argument (exclu) donné à la fonction.
Ce générateur pseudo aléatoire peut être initialisé avec une graine particulière via la fonction `Random.init: int -> unit`. Il peut être utilisé avec une graine pseudo aléatoire sur la base de l'état de la machine au moment de l'exécution avec la fonction `Random.self_init : unit -> unit`

7. Implémentez une fonction `sample_hash_fn: int -> (int -> int)` qui choisit au hasard une fonction de H pour un p donné.
8. On modifie ensuite cette fonction pour ramener son image à un indice valable. Pour ça on peut directement lui appliquer `simple_hash_fn`. Implémentez une fonction `gen_hash_fn : int -> (int -> int)` qui calcule la fonction qui sera utilisée par une table dont la taille est donnée en argument.
9. Testez les fonctions d'interface et cette nouvelle fonction de hachage.
10. (Bonus) Implémentez une fonction de hachage quand les clés sont des mots. Attention aux dépassements de capacité.
11. (Bonus) Testez les tables de hachage dont les clés sont des mots.

Les tables de hachage permettent, via leur usage comme dictionnaire, d'implémenter des approches par mémoïsation. Le calcul de la suite de Fibonacci n'est pas réellement un problème de programmation dynamique, mais on pourrait le considérer comme tel à la marge : les sous problèmes à résoudre pour le calcul de f_{n+2} sont les calculs de f_{n+1} et f_n. Il y a bien entendu chevauchement entre ces sous problèmes.

12. En donnant une majoration et une minoration de chacune des branches de l'arbre d'appel, rappelez la complexité d'un calcul de f_n récursif. Comparez ce résultat au nombre de sous problèmes différents nécessaires pour ce calcul de f_n.
13. L'approche par mémoïsation permet d'éviter ce chevauchement en notant le résultat des calculs déjà réalisés dans un dictionnaire (ici implémenté par une table de hachage). Implémentez une fonction qui calcule le n-ième terme de la suite de Fibonacci avec mémoïsation.
14. Comparez le résultat avec une implémentation récursive naïve (sans mémoïsation) (pour $n = 42$ vous devriez voir une différence).

OCaml fournit un module `Hashtbl` qui permet l'utilisation directe de tables de hachage sans avoir besoin de fournir la fonction de hachage et dont l'interface est la suivante :

— `create : int -> ('a, 'b) t` qui crée une table de taille spécifiée. La taille de la table est mise à jour automatiquement si nécessaire.

— `add : ('a, 'b) t -> 'a -> 'b -> unit` qui ajoute un couple clé valeur à la table passée en argument.

— `find : ('a, 'b) t -> 'a -> 'b` qui renvoie la valeur associée à la clé passée en argument et lève une exception quand aucune valeur n'est associée à la clé dans cette table.

— `mem : ('a, 'b) t -> 'a -> bool` qui vérifie si une valeur est associée à une clé dans la table.

— `remove : ('a, 'b) t -> 'a -> unit` qui supprime le couple clé/valeur de la table. Attention, si jamais lors d'une utilisation de `add` le couple clé/valeur est déjà présente dans la table, un second couple est ajouté, masquant le premier. `remove` ne supprimera que le dernier couple, levant le masquage. Par exemple :

```
let _ = Hashtbl.add table key_1 value_1
let _ = Hashtbl.add table key_1 value_2
let _ = Hashtbl.remove table key_1
let a = Hashtbl.mem table key_1
```

va associer la valeur `true` à `a` : après ces opérateurs, il existe une valeur associée à `key_1` dans `table` et c'est `value_1`. Si la clé n'est pas présente dans la table, rien ne se passe.

— `replace : ('a, 'b) t -> 'a -> 'b -> unit` Qui remplace la valeur associée à la clé donnée si elle est présente et sinon ajoute le couple clé/valeur à la table.

— `iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit` qui permet d'appliquer une fonction à tous les couples clé/valeur de la table. Par exemple dans le code suivant, on utilise iter pour afficher tous les couples clé/valeur d'une table dont les clés sont des `string` et les valeurs des `int` :

```
let print_key_and_value key value =
  Printf.printf "%s %d\n" key value
let table = (Hashtbl.create 5)
let _ = Hashtbl.add table "cinq" 5
let _ = Hashtbl.add table "sept" 7
let _ = Hashtbl.iter (print_key_and_value) table
```

15. Implémentez une fonction `symmetry : (int,int) Hashtbl.t -> unit` qui prend une table dont les clés sont des entiers et les valeurs égales aux clés et fait en sorte que dans cette table, si la clé k est présente, la clé $-k$ est aussi présente.

3 Distance d'édition

La *distance d'édition* ou distance de *Levenshtein* est une distance sur l'ensemble des mots qui compte le nombre minimal d'opérations pour passer d'un mot à un autre. Dans le cas général cette distance permet les opérations suivantes :

— Suppression d'une lettre

— Insertion d'une lettre

— Substitution d'une lettre par une autre sous conditions

Il est courant que ces opérations soient associées à un coût, la distance d'édition correspond alors au coût minimal pour passer d'un mot à un autre. On peut remarquer que la substitution d'une lettre par une autre doit avoir un coût inférieur à la succession d'une insertion puis d'une suppression si on souhaite que cette action de substitution soit utilisée, sinon, il suffit de supprimer la lettre concernée puis de la remplacer par la lettre voulue pour opérer une substitution à l'aide des deux autres opérations. Cette valeur vérifie bien les propriétés attendues d'une fonction de distance (cf le DS6, sous conditions de symétrie sur les coûts de substitution).
Dans ce TP, nous ne nous intéresserons pas à la substitution qu'on considérera impossible. Dans ce cas, la distance d'édition se confond avec le problème de *plus longue sous séquence commune* qui consiste à trouver le plus long sous mot commun à deux mots. On passe alors du premier mot au second en supprimant toutes les lettres nécessaires pour obtenir la plus longue sous séquence commune, puis en insérant toutes les lettres pour obtenir le second mot à partir de cette plus longue sous séquence commune. On rappelle que la définition d'un sous-mot d'un mot w est un mot w' dont les lettres sont une sous suite des lettres de w. Ainsi, un facteur est un sous mot, mais un sous-mot n'est pas nécessairement un facteur. *abed* est un sous mot de *abcded* mais ce n'est pas un de ses facteurs. On peut calculer cette distance d'édition entre deux mots x et y par programmation dynamique : les sous problème consistent à calculer la distance d'éditions sur les préfixes de taille i et j et x et y notés x_i et y_j et à ne pas confondre avec $x[i]$ et $y[j]$ qui sont les lettres numéro i et j de x et y. On a donc $(|x|+1)*(|y|+1)$ sous problème distincts. On peut déduire la distance d'édition dans un cas à partir de sa valeur sur trois sous problèmes particuliers en remarquant que c'est le plus petit nombre d'opération parmi les cas suivants

— Si $i=0$, il faut insérer toutes les lettres, la distance est donc j, idem si $j=0$ il faut supprimer toutes les lettres.

— Si $x[i-1]=y[j-1]$, les deux dernières lettres de x_i et y_j sont égales et on peut garder cette lettre et continuer en calculant la distance entre les préfixes de taille $i-1$ et $j-1$ de x et y.

— On peut supprimer la i-ième lettre de x qui est la dernière lettre du préfixe de taille i puis calculer la distance entre les préfixes de taille $i-1$ et j

— On peut aussi ajouter la j-ième lettre à y qui est la dernière lettre du préfixe de taille j de y puis calculer la distance entre les préfixes de taille i et $j-1$

En notant $d_{i,j}$ la distance entre les préfixes de taille i de x et j de y on a la relation suivante :

$$d_{0,j}=j, d_{i,0}=i$$

$$d_{i,j}=\min(1+d_{i-1,j}, 1+d_{i,j-1}, d_{i-1,j-1}) \text{ si } x[i-1]=x[j-1]$$
$$\min(1+d_{i-1,j}, 1+d_{i,j-1}, d_{i-1,j-1}) \text{ sinon}$$

Pour des raisons explicitées plus tard, on choisit de représenter les mots par la liste de leurs caractères, dans l'ordre inverse, le dernier caractère étant donc en tête de liste.

16. Implémentez une fonction `string_to_list: string -> char list` qui renvoie la liste des caractères de ce mot dans l'ordre inverse : le dernier caractère est en tête de liste.
17. Implémentez une fonction `edit_distance: 'a list -> 'a list -> int` qui calcule la distance d'édition entre deux mots en utilisant la mémoïsation pour ne pas refaire les calculs plusieurs fois. La table utilisée pour la mémoïsation aura alors comme clé le couple de préfixes x_i, y_j indiquant les préfixes courants et représentés sous forme de listes de caractères, et comme valeur la distance minimale de x_i à x_j.

On peut en plus garder trace des opérations utilisées pour obtenir cette distance : lors du calcul de $d_{i,j}$ on sait quelle est l'opération utilisée, on peut donc noter la nature de cette opération dans une seconde table de hachage et déduire la case vers laquelle mène cette opération. On peut ainsi remonter la liste des opérations qui permettent de passer de y à x.

18. Sur la base du code précédent, notez dans une seconde table les opérations nécessaires. On pourra représenter ces opérations par le type `type op = Insertion | Deletion | Identity |End`. La table aura alors comme clé le couple x_i, y_j indiquant les préfixes courants, et comme valeur des `op` où `op` correspond à la première opération appliquée pour passer de x_i à y_j avec le nombre minimal d'opérations. `End` correspond au cas spécifique des préfixes réduits au mot vide, c'est le cas où on a fini.
19. Implémentez une fonction `get_op_list` qui récupère la liste des opérations retenues et le caractère modifié par l'opération à partir de la seconde table contenant les opérations.
20. (Bonus) à partir de la liste précédente, affichez le processus de transformation d'un mot x vers y.

4 Implémentation de diff

`diff` consiste en fait à calculer la distance d'édition sur les lignes d'un texte plutôt que sur les caractères.

21. Implémentez une fonction `text_to_lines` qui calcule la liste des lignes qui composent ce texte. Vous pourrez utiliser judicieusement `String.sub: string -> int -> int` qui permet d'extraire la sous-chaîne d'une chaîne à l'aide de sa position de départ et de sa longueur. `String.sub "aabaa" 2 1` renvoie la chaîne `"b"`.
22. Adaptez le code calculant la distance d'édition sur des listes de mots pour calculer des distances d'éditions sur des listes de lignes. Vous prendrez soin de calculer aussi la liste des opérations.
23. Implémentez une fonction d'affichage similaire à `diff` qui affiche dans l'ordre croissant des lignes les éléments insérés, supprimés et identiques. Devant les lignes insérées vous afficherez un +, devant les lignes supprimées un − et devant les identiques un =.
24. Testez la fonction sur des petits exemples

On souhaite maintenant transformer cette fonction en un exécutable qui affiche la différence et peut être appelé directement sur des fichiers depuis le terminal.

25. Implémentez une fonction `lines_from_file` qui renvoie la liste des lignes contenues dans un fichier. Vous pourrez vous rappeler le TP 10 pour ce qui concerne la lecture de fichiers.

En OCaml, les arguments de la ligne de commande sont stockés dans le tableau `Sys.argv`. Ce sont bien entendu des chaînes de caractères qu'il faut convertir dans le type adéquat si besoin, et comme en C, le premier de ces arguments est le nom du programme. Ainsi le programme suivant (contenu dans un fichier *args.ml*), compilé avec la commande `ocamlop -o args.exe args.ml` et executé avec la commande `./args.exe arg_1 arg_2` affiche :

```
let _ = for i = 0 to Array.length (Sys.argv) -1 do
  Printf.printf "Argument %d: %s\n" i Sys.argv.(i)
done
```

```
Argument 0: args.exe
Argument 1: arg_1
Argument 2: arg_2
```

26. Implémentez un programme qui calcule la `diff` entre deux fichiers dont les noms sont passés via la ligne de commande.
27. Comparez les résultats obtenus à ceux de la commande `diff`.