Calcul avec un automate non-déterministe (avec solutions)

Le but de ce TP est d'implémenter de deux façons la procédure de calcul d'un automate fini non-déterministe. C'est à dire qu'étant donné un automate et un mot, on veut décider si l'automate accepte le mot. On propose dans un premier temps d'implémenter cette procédure avec une approche par *backtracking*, puis avec une approche s'apparentant à la programmation dynamique. Une troisième partie consiste à construire une famille d'automates mettant en évidence les limites de l'approche par backtracking.

Représentation des données. Dans tous le sujet, les automates seront représentés à l'aide des types suivants :

```
type state_id = S of int

(* la relation d'adjacence pour un état :
   étant donnée une lettre, l'état peut mener de 0
   à plusieurs états énumérés dans une liste *)
type adj = (char, state_id list) Hashtbl.t

(* un automate est un ensemble d'états et
   de relation d'adjacence pour chaque état ;
   une liste d'états initiaux ;
   une liste d'états finaux (ou acceptants) *)
type nfa = {states : (state_id, adj) Hashtbl.t;
            init_s : state_id list;
            final_s : state_id list}

(* les lettres sont représentées par le type char,
   et les mots par le type string *)
```

Ces types sont prédéfinis dans un fichier nfa_base.ml, qui contient également deux exemples d'automates vous permettant de partiellement tester vos fonctions.

1. Écrivez une fonction OCaml récursive :
   `_bt_decide (a:nfa) (w:string) (s:state_id) : bool`
   qui prend en entrée un automate `a` représenté comme ci-dessus, un mot (éventuellement vide) `w` et un état `s` de `a`, et qui en suivant une approche par backtracking renvoie `true` ss.'il existe dans `a` un chemin partant de `s` et aboutissant dans un état acceptant, en suivant uniquement des transitions étiquetées par les lettres de `w`.
   Un certain nombre de fonctions de modules OCaml peuvent être utiles ici, par exemple :

   — `List.mem` ; `List.exists`
   — `Hashtbl.find` ; `Hashtbl.find_opt`
   — `String.length` ; `String.sub`

   Toutes ces fonctions (à l'exception de la dernière) sont au programme et doivent pouvoir être utilisées à condition de faire l'objet d'un rappel.

   On propose :

```
let rec _bt_decide (a:nfa) (w:string) (s:state_id) : bool =
    (* mot vide : succès ssi. on est dans un état final
       (ceci aurait pu être fait plus efficacement avec
       une meilleure structure de données type « dictionnaire »
       pour les états finaux... *)
    if w = "" then List.mem s a.final_s
    else
        let c = w.[0] in (* prochaine lettre *)
        let w'    = (String.sub w 1 (String.length w - 1)) in (* reste du mot *)
```

```
        let s_adj = Hashtbl.find a.states s in (* ne doit pas échouer *)
        let nxt   = Hashtbl.find_opt s_adj c in (* prochains états possibles *)
        match nxt with
        | None -> false (* dead-end *)
        (* essaie toutes les possibilités une à une *)
        | Some nxt' -> List.exists (_bt_decide a w') nxt'
```

2. Écrivez une fonction OCaml :
   `bt_decide (a:nfa) (w:string) : bool`
   qui prend en entrée un automate `a` et un mot `w`, et utilise votre fonction `_bt_decide` pour renvoyer `true` ssi. `w` est accepté par `a`.

   Il suffit d'appeler la fonction précédente pour chaque état initial :
   ```
   let bt_decide (a:nfa) (w:string) : bool =
       List.exists (_bt_decide a w) a.init_s
   ```

3. Testez votre fonction, par exemple avec les automates fournis en exemple.

On peut remarquer (et l'on observera cela expérimentalement à la fin du sujet) que l'approche par backtracking souffre d'un coût temporel excessivement mauvais dans le pire cas.

4. Sans l'implémenter, conjecturez un coût pire cas (en fonction du nombre d'états N de l'automate et de la longueur ℓ d'un mot) d'une procédure de calcul pour un automate *déterministe*.

   Un automate déterministe est un automate non-déterministe : il peut donc être représenté avec le même type `nfa` déjà utilisé, où l'on aura simplement des relations d'adjacence de longueur 0 ou 1 (en tant que listes) et un seul état initial.
   Les fonctions `_bt_decide` & `bt_decide` déjà implémentées fonctionnent sans modifications pour ces automates déterministes. Leur coût est alors un $O(\ell + N)$, ou plus précisément $O(\ell + N_f)$ avec N_f le nombre d'états acceptants : pour chaque lettre de `w` on effectue des traitements en coût constant (par hypothèse sur les fonctions de `Hashtbl` et `String` utilisées et le fait que `nxt` est de longueur au plus 1, ce qui implique que `List.exists` ne cause qu'au plus un appel récursif), et la vérification que le dernier état atteint est acceptant se fait ici en $O(N_f)$ (améliorable en $O(1)$).

5. Rappelez succinctement le procédé permettant de construire un automate déterministe reconnaissant le même langage qu'un automate non-déterministe.

   Soit Q l'ensemble des états de l'automate non-déterministe $\mathcal{A}$ en question, il suffit de construire un automate déterministe $\mathcal{A}'$ dont les états représentent les éléments de l'ensemble des parties de Q et les transitions déterministes reflètent la relation de transition non-déterministe de $\mathcal{A}$ étendue aux sous-ensembles d'états. L'état initial de $\mathcal{A}'$ est celui correspondant à l'ensemble des états initiaux de $\mathcal{A}$, et ses états finaux sont ceux correspondant à tous les ensembles d'états incluant au moins un état final de $\mathcal{A}$.

6. Pourquoi le problème de décider si un automate non-déterministe $\mathcal{A}$ reconnaît un mot w ne peut-il pas être résolu efficacement dans le pire cas en construisant un automate déterministe $\mathcal{A}'$ reconnaissant le même langage que $\mathcal{A}$ et en utilisant une procédure de calcul (pour automate déterministe) sur $\mathcal{A}'$ et w ?

   La construction de l'automate déterminisé esquissée à la question précédente a un coût $\Omega(2^N)$ en temps et en espace, quand appliquée à un automate non-déterministe de N états, et il n'existe pas de procédé qui soit meilleur dans le pire cas (ce que l'on admet ici). En conséquence, même si le calcul lui-même avec un automate ainsi déterminisé peut être efficace, la construction préalable de l'automate ne l'est pas forcément.

On va maintenant implémenter une autre procédure de calcul avec un automate non-déterministe, qui se base sur le calcul de la relation de transition δ étendue aux ensembles d'états.

7. Écrivez une fonction OCaml (non récursive) :
   `xdelta (a:nfa) (s_set:(state_id, unit) Hashtbl.t) (c:char)`
   `: (state_id, unit) Hashtbl.t`
   qui prend en entrée un automate `a`, un ensemble d'états $\mathcal{S}$ de `a` représenté par une table de hachage `s_set` de type `(state_id, unit) Hashtbl.t` (plus précisément, un état est présent dans l'ensemble ss.'il est présent comme clef dans `s_set` (avec la valeur associée `()` ; non utilisée)), une lettre `c`, et qui

calcule et renvoie un nouvel ensemble d'états $\mathcal{S}'$ (représenté de la même façon que $\mathcal{S}$) correspondant aux états accessibles depuis $\mathcal{S}$ en suivant les transitions étiquetées par `c`.
Un certain nombre de fonctions de modules OCaml peuvent être utiles ici, par exemple :

— `List.iter`
— `Hashtbl.find` ; `Hashtbl.find_opt` `Hashtbl.create` ;
`Hashtbl.replace` ; `Hashtbl.iter`

Toutes ces fonctions (à l'exception incompréhensible de `Hashtbl.replace`) sont au programme et doivent pouvoir être utilisées à condition de faire l'objet d'un rappel.

On propose :
```
let xdelta (a:nfa) (s_set:(state_id, unit) Hashtbl.t) (c:char)
    : (state_id, unit) Hashtbl.t =
    let ns_set = Hashtbl.create (Hashtbl.length a.states) in
    let add_all x = List.iter (fun s -> Hashtbl.replace ns_set s ()) x in
    let add_chr s _ =
        let s_adj = Hashtbl.find a.states s in
        let nxt = Hashtbl.find_opt s_adj c in
        match nxt with
        | None -> ()
        | Some nxt' -> add_all nxt'
    in
    (Hashtbl.iter add_chr s_set ; ns_set)
```
On crée un nouvel ensemble d'états vierge `ns_set` et l'on définit une fonction `add_all` qui y « ajoute » tous les éléments de la liste qu'elle prend en argument. Cette fonction est utilisée dans `add_chr` pour « ajouter » tous les états possibles après lecture de `c` depuis l'état qu'elle prend en argument. Enfin, on appelle `add_chr` pour tous les états présents dans l'ensemble `s_set` initial.

8. Déterminez le coût pire-cas de votre fonction `xdelta` en fonction de la taille S de `s_set` et du nombre d'états N de `a`, et en supposant un coût constant pour les opérations de `Hashtbl` faisant intervenir *une* clef.

Avec les hypothèses de la question, la fonction `add_all` a un coût linéaire en la longueur de son entrée, qui est un $O(N)$; la fonction `add_chr` a donc elle aussi un coût en $O(N)$, et elle est appelée S fois : le coût de `xdelta` est donc un $O(NS)$ (qui est un $O(N^2)$).

9. Écrivez une fonction OCaml (non-récursive) :
`dp_decide (a:nfa) (w:string) : bool`
qui prend en entrée un automate `a` et un mot `w`, et utilise votre fonction `xdelta` pour renvoyer `true` ssi. `w` est accepté par `a`.
Un certain nombre de fonctions de modules OCaml peuvent être utiles ici, par exemple :

— `List.iter` ; `List.exists`
— `Hashtbl.create` ; `Hashtbl.add` ; `Hashtbl.length` ; `Hashtbl.mem`
— `String.for_all`

Toutes ces fonctions (à l'exception incompréhensible de `Hashtbl.length` et (celle plus compréhensible) de `String.for_all`) sont au programme et doivent pouvoir être utilisées à condition de faire l'objet d'un rappel.

On propose :
```
let dp_decide (a:nfa) (w:string) : bool =
    let cs = ref (Hashtbl.create (List.length a.init_s)) in
    let () = List.iter (fun x -> Hashtbl.add !cs x ()) a.init_s in
    let process c =
        let ns = xdelta a !cs c in
        (cs := ns ; Hashtbl.length ns > 0)
    in
    let accept () = List.exists (fun s -> Hashtbl.mem !cs s) a.final_s in
    String.for_all process w && accept ()
```

On utilise une référence pour mettre à jour l'ensemble d'états à considérer, et l'on s'arrête dans le parcours du mot si celui-ci devient vide (dans ce cas, le mot n'est pas accepté).
L'utilisation de `String.for_all` n'est pas essentielle, mais elle permet d'interrompre le calcul de façon anticipée dans certains cas.
La fonction `accept` cherche à exploiter le fait que contrairement à `List.mem`, `Hashtbl.mem` a un coût (moyen) constant, mais ce n'est pas une nécessité.

10. Déterminez le coût pire-cas de votre fonction `dp_decide` en fonction de la longueur ℓ de `w` et du nombre d'états N de `a`.

Sans hypothèses particulières sur l'automate, on suppose un coût $O(N^2)$ pour `xdelta`, qui est appelée au plus ℓ fois. Le coût initial de création de `cs` et celui d'`accept` sont tous deux en $O(N)$ (ou plus précisément $O(N_i)$ et $O(N_f)$ avec N_i et N_f le nombre d'états initiaux et finaux respectivement, mais cela n'a pas d'importance ici). Les autres opérations (notamment `Hashtbl.length`) on un coût constant, ce qui donne un coût total en $O(N^2\ell)$.

11. Testez votre fonction, par exemple avec les automates fournis en exemple.

On souhaite maintenant construire et étudier une famille $(\mathcal{A}_n)$ d'automates pour laquelle l'approche par backtracking est particulièrement inefficace. L'automate $\mathcal{A}_n$ de cette famille reconnaît le langage très simple $\{a^i\}_{n\leq i\leq 2n}$, ce qui montre au passage que ce mauvais comportement n'est pas directement lié à la complexité du langage.
L'automate $\mathcal{A}_n$ de cette famille est construit comme suit (l'alphabet étant $\{a\}$, toutes les transitions mentionnées ci-dessous sont étiquetées par a) :

— il possède $2n+1$ états numérotés de 1 à $2n+1$
— son unique état initial est l'état 1 ; son unique état final est l'état $2n+1$
— tous les états $i < 2n+1$ possèdent une transition vers l'état $i+1$
— tous les états $i \leq n$ possèdent en plus une transition vers *chacun* des états j vérifiant $i+1 < j \leq n+2$.

On représente graphiquement dans la Figure 1 l'automate $\mathcal{A}_2$ de cette famille.

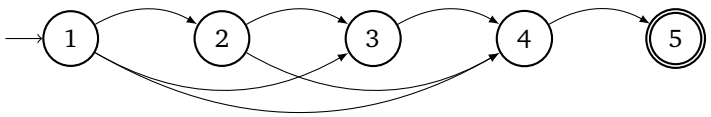

FIGURE 1 – L'automate $\mathcal{A}_2$; les étiquettes sont omises pour une meilleure lisibilité

12. Montrez que les automates $\mathcal{A}_n$ construits comme décrit ci-dessus reconnaissent bien le langage $\{a^i\}_{n\leq i\leq 2n}$.

Le graphe représentant $\mathcal{A}_n$ est acyclique et possède $2n+1$ sommets s_i (qu'on indice de la même façon que les états qu'ils représentent), ce qui donne une majoration en $2n$ pour la longueur des mots qu'il accepte.
Par construction de $\mathcal{A}_n$ et notamment le fait que le degré entrant des sommets $s_{n+3}, \ldots, s_{2n+1}$ est 1, on a que tout chemin du graphe de longueur $k < n$ qui aboutit à l'unique sommet acceptant s_{2n+1} est nécessairement de la forme :

$$s_{2n+1-k} \to s_{2n+1-k+1} \to \cdots \to s_{2n+1}$$

Aucun des sommets $\{s_i\}_{n+2\leq i\leq 2n+1}$ ne correspondant à un état initial de l'automate $\mathcal{A}_n$, on déduit un minorant n pour la longueur des mots qu'il accepte.
De même, pour tout $1 \leq k \leq 2n$ il *existe* un chemin de longueur k de la forme :

$$s_{2n+1-k} \to s_{2n+1-k+1} \to \cdots \to s_{2n+1}$$

Comme le sommet s_1 est à distance 1 des sommets $\{s_i\}_{2\leq i\leq n+2}$, il existe des chemins :

$$s_1 \to s_{n+2} \to \cdots \to s_{2n+1}, \quad s_1 \to s_{n+1} \to \cdots \to s_{2n+1}, \quad \ldots, \quad s_1 \to s_2 \to \cdots \to s_{2n+1}$$

qui sont de longueur $n \leq k \leq 2n$ et relient s_1 à s_{2n+1}.
Ceci permet de conclure, en combinaison avec les minoration et majoration établies précédemment et le fait qu'un mot a^i est accepté par $\mathcal{A}_n$ ss.'il existe un chemin de longueur i entre s_1 et s_{2n+1}.

13. Écrivez une fonction OCaml

`create_an2n_explo (n:int) : nfa`
qui prend en entrée `n` et construit et renvoie une représentation de l'automate $\mathcal{A}_n$.

Un certain nombre de fonctions de modules OCaml peuvent être utiles ici, par exemple :

— `List.init`
— `Hashtbl.create` ; `Hashtbl.add`

Toutes ces fonctions (à l'exception relativement incompréhensible de `List.init`) sont au programme et doivent pouvoir être utilisées à condition de faire l'objet d'un rappel.

On propose :

```
let create_an2n_explo n =
    let sn = 2 * n + 1 in
    let states = Hashtbl.create sn in
    let () = for i = 1 to n do
                 let adj = Hashtbl.create 1 in
                 let () = Hashtbl.add adj 'a' (List.init (n + 2 - i)
                                                 (fun x -> S (x + i + 1))) in
                 Hashtbl.add states (S i) adj done in
    let () = for i = n + 1 to sn - 1 do
                 let adj = Hashtbl.create 1 in
                 let () = Hashtbl.add adj 'a' [S (i + 1)] in
                 Hashtbl.add states (S i) adj done in
    let () = Hashtbl.add states (S sn) (Hashtbl.create 0)
    in {states = states; init_s = [S 1]; final_s = [S sn]}
```

14. Testez vos fonctions `bt_decide` et `dp_decide` sur des automates construits avec la fonction ci-dessus, par exemple avec une représentation de $\mathcal{A}_{20}$ et les mots a^{19} et a^{20}.

On pourrait faire la remarque que la famille d'automate décrite ci-dessus est inutilement compliquée pour le langage que l'on cherche à reconnaître. C'est effectivement vrai, mais cela ne doit pas pour autant occulter les performances catastrophiques d'une implémentation par backtracking dans certains cas. Il est d'ailleurs malheureux que ces implémentations ne soient pas rares dans les moteurs d'expression régulière, ce qui peut demander un travail supplémentaire lors de l'écriture d'une expression régulière afin d'essayer d'éviter de tomber dans un mauvais cas, *cf.* la référence donnée en fin de sujet.

15. Proposez une famille d'automate $(\mathcal{A}'_n)$ t.q. $\mathcal{A}'_n$ reconnaît le même langage que $\mathcal{A}_n$, où $\mathcal{A}'_n$ est obtenu à partir de $\mathcal{A}_n$ en en retirant certaines transitions bien choisies.
Vous *devez* montrer que les automates ainsi construits reconnaissent bien le bon langage.

On peut construire $\mathcal{A}'_n$ comme $\mathcal{A}_n$ dont on a ôté les transitions supplémentaires des états $2, \dots, n$. La preuve donnée en réponse à la question 12 n'utilisant pas ces dernières transitions, on a immédiatement que $(\mathcal{A}'_n)$ reconnaît la bonne famille de langage.

16. Écrivez une fonction OCaml
`create_an2n (n:int) : nfa`
qui prend en entrée `n` et construit et renvoie une représentation de l'automate $\mathcal{A}'_n$ suivant votre construction.

On propose :

```
let create_an2n n =
    let sn = 2 * n + 1 in
    let states = Hashtbl.create sn in
    let adj_1 = Hashtbl.create n in
    let () = Hashtbl.add adj_1 'a' (List.init (n + 1) (fun x -> S (x + 2))) in
    let () = Hashtbl.add states (S 1) adj_1 in
    let () = for i = 2 to sn - 1 do
                 let adj = Hashtbl.create 1 in
                 let () = Hashtbl.add adj 'a' [S (i + 1)] in
                 Hashtbl.add states (S i) adj done in
    let () = Hashtbl.add states (S sn) (Hashtbl.create 0)
    in {states = states; init_s = [S 1]; final_s = [S sn]}
```

17. Testez vos fonctions `bt_decide` et `dp_decide` sur des automates construits avec la fonction ci-dessus, et comparez avec la construction précédente.

Une référence. Pour plus de détails sur le sujet, des comparatifs de performance, des références historiques etc., on peut consulter l'article de blog suivant (d'où la famille d'automates $\mathcal{A}_n$ est implicitement tirée) : Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).