Calcul avec un automate non-déterministe

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.

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

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.

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.

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.

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`.

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.

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

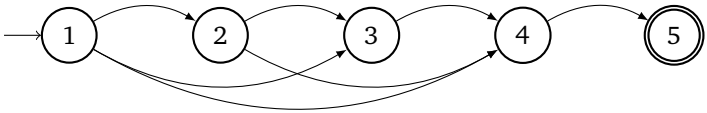

FIGURE 1 – L'automate $\mathcal{A}_2$; les étiquettes sont omises pour une meilleure lisibilité

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.

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.

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.

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, ...).