Automates à compteur avec limite d'inversion
Le test du vide

Nicolas Feron

Université Libre de Bruxelles, Bruxelles, Belgique
nicolas.feron@ulb.ac.be

Abstract

Les automates à compteur avec limite d'inversion sont déjà connus et introduits dans un article de Oscar H. Ibarra. Le but de ce mini-mémoire était de chercher et implémenter un test du vide pour ces derniers. Est-ce un test réalisable ? En quelle complexité ? Quels éléments rendent le problème décidable ? Ce sont toutes des questions auxquelles on va tenter de répondre lors de ce travail.

L'article nous emmène donc dans la réflexion et l'implémentation des différentes stratégies imaginées et mises en oeuvre.

Pour résumer le résultat en quelques mots, nous avons un algorithme assez simple et très efficace pour de petits automates, lorsque cet automate se complique, le test du vide peut très vite devenir complexe et long. Il est bien sûr possible d'ajouter plus d'améliorations et il ne faut pas oublier que chaque algorithme peut être plus efficace que l'autre suivant l'automate en entrée.

Introduction

Les automates sont souvent décrits comme des machines donnant une réponse suivant une entrée à fournir. De manière plus précise et scientifique, ce sont des 5-tuples

$$M = (Q, \Sigma, q_0, \delta, F)$$

avec respectivement un ensemble d'états, un alphabet des symboles d'entrée, un état initial, une fonction de transition et un ensemble d'états acceptants. On commence toujours par l'état initial, et on voyage dans les états de Σ en suivant la fonction δ pour les valeurs en entrée.

Cependant cet article se concentre sur le test du vide des automates à compteur avec limite d'inversion. Ce sont des automates finis [1] déterministe [2] à compteurs [3] avec limite d'inversion [4], ils peuvent être décrits par un 6-tuple

$$M = (k, Q, \Sigma, \delta, q_0, F)$$

1. un nombre fini d'états
2. pour un état et une entrée, la fonction δ donnera toujours le même résultat
3. des compteurs sont maintenus et incrémenté/décrémenté lors de changement d'états, ils peuvent être comparés à 0 dans la fonction δ
4. les compteurs peuvent changer de sens incrémentation <–> décrémentation un nombre fini de fois

respectivement la limite sur les compteurs, les états, les entrées, la fonction de transition, l'état initial et les états acceptants. Ces contraintes simplifient le problème étudié (test du vide) et notamment la limite d'inversion qui le rend décidable en temps polynomiale pour les automates à compteur en ajoutant une limite à la **taille** des mots d'entrée (détaillé dans un article d'Oscar H. Ibarra, Journal 22).

Environnement automate

Avant de commencer à travailler sur un test du vide, il était nécessaire de développer un environnement de travail. Après une étude des automates, le travail a donc commencé. En résultat, nous avons maintenant un environnement permettant de créer des graphes représentant des automates. Ces derniers sont actuellement uniquement importés via des fichiers texte. Le parcours du graphe est possible suivant un mot et correspond au parcours d'un automate avec comme point de départ son état initial et en entrée toujours ce même mot.

Il est possible d'utiliser plusieurs types d'automate. En effet, l'environnement ne s'est pas développé d'un coup, il a fallu commencer par travailler avec de simples automates. Chaque ajout n'empêche pas l'utilisateur d'utiliser les automates acceptés précédemment, il y a donc un travail de maintenance sur les versions précédentes. Dans l'ordre, les ajouts furent : l'import depuis un fichier, le parcours de l'automate, les compteurs avec opérateur de comparaison =, le fait de pouvoir ignorer des compteurs, les opérateurs < et > et enfin les limites d'inversion.

Voici quelques commandes basiques pour l'utilisation de l'environnement :

Création d'un graphe automate :

Attention, le format du contenu du fichier d'import est strict : il faut, dans l'ordre, en première ligne, la taille du header (le nombre de ligne(s) contenant des caractéristiques de l'automate), les dites caractéristiques, les états et enfin les règles de transition (liens).

```
Graph newGraph("automaton3.txt");
```

Affichage des nombres de noeuds et liens :

```
newGraph.uglyPrint();
```

Affichage détaillé des noeuds et liens :

```
newGraph.print();
```

Entrée d'un mot :

La fonction renvoie un booléen d'où la condition ternaire

```
std::string word;
std::cout << "enter your word: ";
std::cin >> word;
(newGraph.wordEntryWithCounters(word)) ?
std::cout<<"accepted"<<std::endl :
std::cout<<"refused"<<std::endl;
```

Lancement d'un test du vide :

La fonction renvoie un booléen d'où la condition ternaire

```
std::string accepted = "accepts at least
one word";
std::string refused = "no word accepted";
(newGraph.voidTest(Graph::DEPTH_FIRST)) ?
std::cout << accepted << std::endl :
std::cout << refused << std::endl;
```

Le test du vide

Au niveau du test du vide, le plus simple à mettre en oeuvre est un algorithme qui va tester chaque mot jusqu'à en trouver un qui est accepté. Ce fut donc la première étape. Cependant un algorithme pareil peut tourner indéfiniment si nous ne lui donnons pas de condition d'arrêt. Au niveau des conditions d'arrêt, nous pouvons lui fournir plusieurs choses, une limite d'itération de l'algorithme, une limite en taille de mot, une limite en nombre de noeuds parcourus, une limite sur le nombre de fois qu'on parcourt un même noeud,... Il y a beaucoup de possibilités à ce niveau.

Taille des mots

La première limite introduite fut sur la taille des mots, en effet, comme précédemment cité, Oscar H. Ibarra a introduit une limite sur la taille des mots pour ce type d'automate (avant d'être certain d'avoir un état acceptant ou un boucle dans l'automate). Cette limite prend en compte plusieurs caractéristiques de l'automate étudié : le nombre de compteur(s) 'm', le nombre de lien(s) 's' et une constance 'c' :

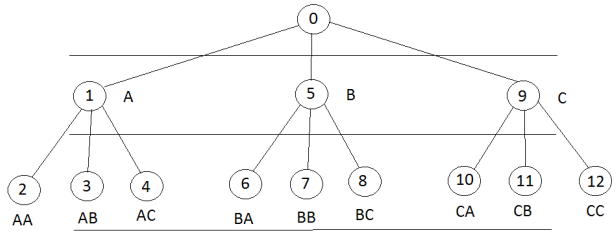
$$(m\ s)^{sc}$$

Cette version du test du vide a une complexité dans le pire des cas en $O(a!^n)$ avec n, la taille maximale d'un mot, et a, le nombre de lettres dans l'alphabet de l'automate.

De manière plus générale, ce n'est pas réellement la taille du mot d'entrée qui est limitée mais le nombre de transitions possibles avant d'être certain d'avoir atteint un état acceptant ou d'être coincé dans une boucle. Dans un simple automate sans transition ε (transition qui a pour seul but d'incrémenter ou décrémenter un et un seul compteur), c'est donc la taille du mot qui est limitée.

Parcours en profondeur

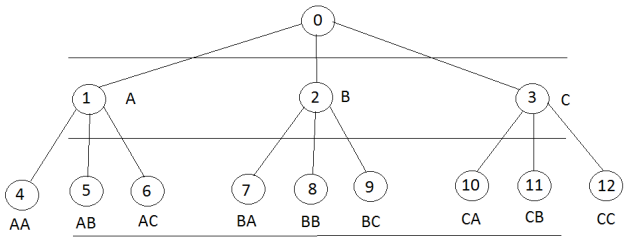
C'est la première version du test du vide développée. Un parcours en profondeur signifie qu'on travaille avec un stack (tas). On commence par mettre un mot vide dans le stack, ensuite on fait trouner une boucle qui va sortir le premier élément du stack (mot vide), on ajoute à ce mot une lettre de l'alphabet, on regarde si le mot est accepté puis on met le mot sur le stack (à faire pour chaque lettre), etc.. Pour un alphabet A, B et des mots de 4 lettres, cela ressemble à : A, AA, AAA, AAAA, AAAB, AABA, AABB, ABAA, etc..



Bien sûr ceci est la représentation de l'ordre des mots, l'image ne représente en rien l'avancée dans le graphe.

Parcours en largeur

Dans cette version, nous utilisons un heap (pile). C'est le même algorithme en remplaçant le stack par un heap, le premier rentré sera donc le premier servi, un parcours va ressembler à : A, B, AA, AB, BA, BB, AAA, AAB, ABA, ABB, etc..



Bien sûr ceci est la représentation de l'ordre des mots, l'image ne représente en rien l'avancée dans le graphe.

Parcours en profondeur dynamique

Dans le simple parcours en profondeur, on testait mot par mot, ce qui signifie parcourir le graphe pour AAA (2 transitions d'états) et pour AAAA (3 transitions d'états), des étapes sont ici répétées. En effet, deux transitions sont identiques.

On peut alors améliorer l'algorithme en regardant lettre par lettre, c'est le parcours dynamique. Nous regardons les mots dans le même ordre que le simple parcours en profondeur mais sans les répétitions de transition. Pour tenter d'être plus clair, voici un exemple d'étapes :

Pour un automate d'alphabet {A,B,C} qui n'accepte que les mots avec un C en 3^{eme} position. Cet automate boucle sur le premier état et incrémente un compteur pour chaque lettre, si le compteur est égal à 2 et la lettre suivante est un C, on passe à l'état 2, état acceptant.

1. compteur = 0, état 1 (initialisation)
2. A -> compteur = 1, état 1 (mot testé == A)
3. A -> compteur = 2, état 1 (mot testé == AA)
4. A -> compteur = 3, état 1 (mot testé == AAA)
5. A -> compteur = 4, état 1 (mot testé == AAAA)
6. taille maximale d'un mot => on fait un pas en arrière et continue avec la lettre suivante
7. B -> compteur = 4, état 1 (AAAB)
8. taille maximale d'un mot => on fait un pas en arrière et continue avec la lettre suivante
9. C -> compteur = 4, état 1 (AAAC)
10. taille maximale d'un mot => on fait un pas en arrière et continue avec la lettre suivante
11. B -> compteur = 3, état 1 (AAB)
12. ... les 3 lettres en dernière position ... retour en arrière ...
13. C -> compteur = 2, état 2 (AAC)
14. le mot est accepté donc l'algorithme s'arrête, l'automate accepte au moins un mot.

Grâce à cette amélioration nous réduisons la complexité dans le pire des cas à $O(a^n)$

Parcours en profondeur dynamique avec sauvegardes d'états

On rajoute encore une couche supplémentaire, nous allons maintenant nous souvenir d'états dans l'algorithme. Une sauvegarde d'état est un 2-tuple composé de l'état q_i dans lequel nous sommes ainsi que la liste C des i compteurs c_i.

$$save = (q_i, C)$$

Pour chaque sauvegarde, nous allons ajouter les différentes lettres au mot puis garder cette sauvegarde dans un vecteur. Si cette sauvegarde est déjà présente dans le vecteur, nous sommes déjà passés par ce même état avec les mêmes valeurs de compteurs, nous sommes soit dans une boucle soit, simplement, dans deux parcours identiques. Si tel est le cas, nous arrêtons de parcourir le graphe dans cette direction.

En résumé, nous évitons des répétitions et surtout de nous prendre dans des boucles interminables.

Cette amélioration va augmenter la complexité dans le pire des cas à $O(a^{2n})$, en effet pour chaque état rencontré, nous devons regarder si il a déjà été parcouru. **Cependant**, nous allons réduire la complexité moyenne pour des automates complexes. En effet, ces automates sont destinés à contenir des boucles.

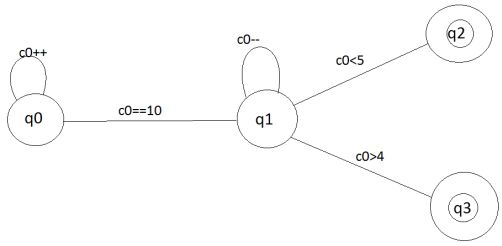
Parcours depuis un état acceptant :

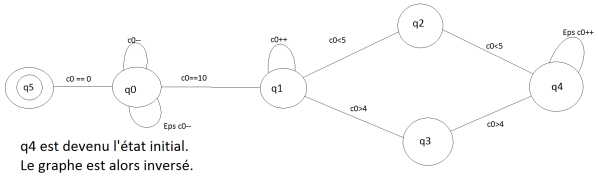
Il est possible de partir d'un état acceptant et de voir si on arrive à retourner à l'état initial. Le but étant d'éviter que notre parcours s'écarte de la solution pour n'y revenir qu'à la fin (pire des cas d'un parcours entier depuis un état initial).

Cependant cela requiert d'inverser notre graphe, donc inverser l'ordre des noeuds et le sens des liens ainsi qu'un update au niveau des comparaisons lorsqu'un lien compare un compteur ET incrémente/décrémente le même compteur.

En addition de cette inversion, il faut compléter le graphe inversé au niveau du nouvel état initial avec des transitions ε pour avoir les valeurs acceptantes de compteur.

Il est encore possible de pousser plus loin, dans le cas d'un automate avec plusieurs états acceptants, nous pouvons créer un nouvel état acceptant ainsi que des liens transitions ε en bouclant sur ce dernier. On peut dès lors ajouter des liens vers les anciens états acceptants avec des comparaisons sur les compteurs pour avoir leurs valeurs acceptantes.





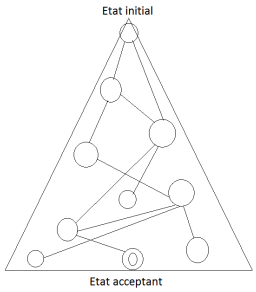
Toutes les précédentes versions peuvent être utilisées avec un graphe inversé

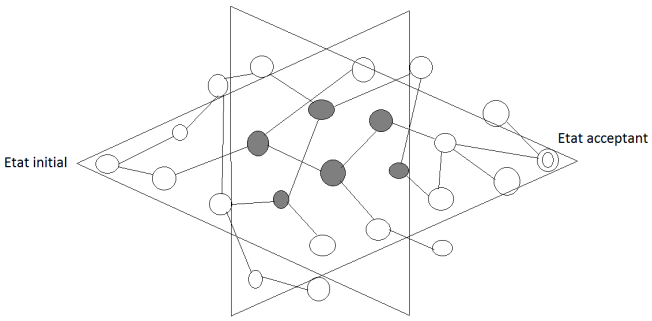
Améliorations possibles

Voici des améliorations qui n'ont pas été implémentées.

Parcours dans les deux sens : Il est possible de réaliser un parcours qui part dans les deux sens et tente de trouver un même état avec les mêmes valeurs de compteur.

Pourquoi est ce que ce serait une amélioration ? Il faut s'imaginer un parcours comme un triangle, plus on avance dedans, plus on a de chance de s'écarter de l'état initial (pas forcément, vu que c'est un graphe, tout est possible) ou des valeurs de compteurs initiales.





Nous visons dés lors ces noeuds surlignés en gris, en tentant d'éviter de s'aventurer dans les noeuds qui peuvent nous mener loin de l'état acceptant. Il faudrait obligatoirement utiliser un parcours en largeur pour ce type d'amélioration.

Tout en un : Nous pourrions aussi tester un algorithme qui en combine plusieurs (par exemple un algorithme qui parcourt en profondeur et en largeur en même temps) et, ce, dans le but d'en tirer les bénéfices des deux.

Mais comme dit précédemment, chaque automate est fort différent et un algorithme en profondeur peut être instantané dans le meilleur des cas ou extrêmement long dans le pire des cas. Imaginons que le seul mot accepté soit un mot rempli de la dernière lettre de l'alphabet.

Exemples d'exécution

L'automate utilisé pour la plupart des tests est un automate qui reconnaît tout mot ayant un 'C' en 3^{eme} lettre. Cet automate peut être importé grâce à un fichier de format : state>isResponse link >origin :target :value :comparaisonOperator :counterValue :counterModification

```
HEADER_SIZE 6
NUMBER_OF_COUNTERS 1
REVERSAL_BOUND 5
ALPHABET A B C END
LINE_BEGIN_SIZE 6
FILE_SEPARATOR :
state>0
state>1
link >0:1:C:=:2:0
link >0:0:A:=:-1:1
link >0:0:B:=:-1:1
link >0:0:C:=:-1:1
```

Nous Créons le Graphe grâce à la ligne :

```
Graph newGraph("automaton3.txt");
```

Nous l'affichons ensuite :

```
newGraph.print();
```

Ce qui a pour résultat :

Les lignes trop grandes ont été tronquées par un '/'.

```
Graph:

state> ID: 0 | isResponse: 0
link > ID: 0 > origin: 0 -> target: 1 | /
    value: C > counter: =:2:0:
link > ID: 1 > origin: 0 -> target: 0 | /
    value: A > counter: =:-1:1:
link > ID: 2 > origin: 0 -> target: 0 | /
    value: B > counter: =:-1:1:
link > ID: 3 > origin: 0 -> target: 0 | /
    value: C > counter: =:-1:1:

state> ID: 1 | isResponse: 1
```

Nous pouvons entrer un mot grâce à

```
std::string word;
  while(word != "exit")
  {
    std::cout << "enter your word: ";
    std::cin >> word;
    (newGraph.wordEntryWithCounters(word))\
    ? std::cout<<"accepted"<<std::endl :\
    std::cout<<"refused"<<std::endl;
  }
```

Ce qui va donner :

```
enter your word: bababababa
refused
enter your word: ccaccca
refused
enter your word: aacaaaa
accepted
enter your word: ccccccc
accepted
enter your word: exit
```

Un test du vide peut être appelé grâce à la ligne :

```
newGraph.voidTest(Graph::DEPTH_FIRST)
```

Graph : :DEPTH_FIRST est un des multiples types de test du vide implémentés, voici une liste exhaustive :

— DEPTH_FIRST
— BREADTH_FIRST
— DEPTH_FIRST_DYNAMIC
— DEPTH_FIRST_DYNAMIC_STATES_SAVE
— DEPTH_FIRST_FROM_END
— BREADTH_FIRST_FROM_END

Pour notre petit automate, nous avons comme résultat :

DEPTH_FIRST Parcours en profondeur, trouve 'AAC' en 0.010s à 0.012s.

DEPTH_FIRST Parcours en largeur, trouve 'AAC' en 0.010s à 0.014s.

DEPTH_FIRST_DYNAMIC

Parcours en profondeur dynamique, trouve 'AAC' en 0.009s à 0.012s.

On remarque donc que dans le cas d'un automate si petit, la différence est minime. Simplement car une partie du temps d'exécution est utilisée pour la construction du graphe et la lecture du fichier. Ces éléments deviendront mois significatifs pour de plus grands automates.

Conclusion

Au terme de ce travail, qui fut principalement de la recherche / réflexion mixées avec une partie d'implémentation et ses tests, nous avons un environnement automate flexible et modulable qui permet plusieurs types de tests du vide.

Une remarque à souligner est que chaque automate aura un algorithme plus efficace qu'un autre, il n'y a pas d'algorithme "le plus rapide" pour tous les automates. Par contre, nous pouvons tenter d'avoir une complexité moyenne la plus faible possible. La meilleure amélioration apportée durant ce travail est certainement la manière dynamique qui évite énormément de répétitions.

Il reste sans aucun doute une multitude de moyens pour arriver à réduire cette complexité moyenne.

Références

Oscar H. Ibarra (1973) Journal of Computer and System Sciences 7, pages 28 - 36

Oscar H. Ibarra (1981) Journal of Computer and System Sciences 22, 220 - 229

Michael Sipser (2006) Introduction to the Theory of Computation (2nd edition) chapitres 0 et 1