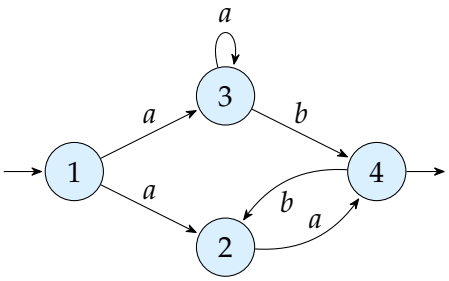
TP 3
Vcsn– 2

Version du 26 septembre 2016

Dans ce TP, nous allons utiliser à nouveau Vcsn. Si nécessaire, reprenez les instructions du TP 2.

Exercice 1 – Évaluation sur automate déterministe

1. Tout automate admet un automate déterministe équivalent. La méthode ‘`automaton.determinize`’ permet d’en calculer un.



Automate 1: Automate à déterminiser

   Déterminisez l’automate 1 et affichez le résultat cette opération. Vous pouvez aussi vérifier ceux des TDs !
2. Il est possible d’utiliser Vcsn pour évaluer un mot sur un automate à l’aide de ‘`automaton(word)`’, ou encore ‘`automaton.eval`’.

   Réutilisez les automates créés à partir d’expressions rationnelles du TP précédent pour évaluer des mots, qui selon vous, sont reconnus, ou pas, par ces automates.
3. La notation pointée (‘`obj.fun()`’) offre l’intérêt de pouvoir chaîner lisiblement plusieurs opérations à la suite, dans l’ordre de leur évaluation, de la gauche vers la droite. Par exemple :

   ```
   vcsn.context('lal_char(abc), b').expression('ab*').thompson().proper()
   ```

   crée l’automate de Thompson de l’expression ‘`ab*`’, puis en élimine les transitions spontanées (et les états devenus inutiles).

   Ajoutez à l’exemple précédent une étape de déterminisation, ainsi qu’une étape de minimisation (concept qui sera abordé plus tard en cours, mais qui consiste en la recherche de l’*unique* automate déterministe au plus petit nombre d’états).

   Ensuite, chaînez des commandes que vous connaissez déjà de manière à générer l’automate correspondant à l’expression rationnelle de votre choix. (Vous pouvez écrire une fonction si vous savez.)

   Sur cet automate, vous appliquerez les étapes suivantes :
   1. Élimination des transitions spontanées
   2. Déterminisation
   3. Évaluation d’un mot

Exercice 2 – Intersection de langages rationnels

Nous verrons demain une construction entre deux automates, le *produit synchronisé*. Elle produit un automate acceptant l’intersection des langages des deux automates.

1. Construire un automate qui accepte les mots sur $\{a, b\}$ ayant un nombre impair de b. Vous pouvez utiliser la méthode '*expression*.automaton' pour convertir une expression rationnelle en NFA.
   Vérifier votre automate grâce à '*automaton*.shortest'.
2. L'opération '*automaton* & *automaton*' (ou encore '*automaton*.conjunction') calcule le produit synchronisé de deux automates.
   Calculez un automate qui accepte les mots ayant un nombre impair de a et un nombre impair de b. Le vérifier grâce à 'shortest'.
3. En déduire une expression rationnelle qui dénote l'ensemble des mots sur $\{a, b\}$ ayant un nombre impair de a, et un nombre impair de b.
4. Vcsn supporte les expressions rationnelles *étendues* qui incluent deux opérateurs supplémentaires : 'e&f' pour la conjonction (intersection) et 'e{c}' pour la complémentation.
   Reprennez la question précédente, et traitez-la directement sur les expressions rationnelles, grâce à '*expression* & *expression*' (ou encore '*expression*.conjunction').

Exercice 3 – Le problème de Mans Hulden

Durant la conférence FSMNLP'08, Mans Hulden a mentionné un petit problème de langage dont une solution utilise une composition complexe de langages rationnels : étant donné un langage rationnel L, construire le langage L' des mots de Σ^* qui contiennent exactement un seul facteur dans L. Il se trouve que L' est rationnel.

Le propos de cet exercice est de traiter ce problème. Nous pourrions représenter les langages sur la forme d'automates et les transformer (concaténation, conjonction d'automates, etc.). Puisque Vcsn supporte les expressions étendues, nous le ferons avec des expressions, ce qui donnera des résultats équivalents, mais plus faciles à lire.

1. Définissez la fonction suivante, analysez-la, et essayez-la sur l'expression $ab + ba$.
   Pensez à '*expression*.shortest'.

```
ctx = vcsn.context('lal_char(abc), b')

# A helper function to define expressions on 'lal_char(abc), b'.
def exp(e):
    # If 'e' is not a real expression object, convert it.
    return e if isinstance(e, vcsn.expression) else ctx.expression(e)

def t1(re):
    re = exp(re)
    all = exp('[^]*')
    # * denotes the concatenation.
    return all * re * all
t1('a')
```

2. Soit e une expression rationnelle. Proposez une expression rationnelle qui désigne tous les mots qui contiennent au moins deux facteurs disjoints (sans recouvrement) qui sont des mots engendrés par e.
3. Définir une fonction Python qui construise cette expression rationnelle.
4. Vérifiez, à l'aide de 'shortest', que votre réponse est (probablement) correcte pour $e = ab + ba$.
5. Proposez une expression rationnelle qui désigne tous les mots qui ont un préfixe dans e et un suffixe propre dans e. Pensez à l'opérateur '&'.

6. Adaptez votre fonction pour engendrer cette expression rationnelle, et vérifiez-la pour $e = ab + ba$.
7. Adaptez votre fonction pour qu'elle reconnaisse tout mot qui contienne deux facteurs (au moins) de e, qui se recouvrent potentiellement.
8. L'opération '`expression % expression`' (ou encore '`expression.difference`') engendre une expression rationnelle qui désigne les mots acceptés par la première expression mais pas la seconde.

   Adaptez votre fonction pour qu'elle reconnaisse tout mot qui contienne exactement un seul facteur de e. Essayez-la.
9. Vérifiez la correction de votre réponse en prenant $L = \{ab, aba\}$. Qu'observez-vous ? Comment l'expliquer ?
10. Étant donné un langage rationnel L, caractérisez le langage des mots de L qui ont un (ou plusieurs) facteur propre dans L.

    Implémentez une fonction '`composite`' qui réalise ce calcul pour des expressions rationnelles, et testez-là sur $L = \{ab, aba\}$.
11. Proposer finalement une fonction qui prenne en argument une expression rationnelle e et retourne une expression rationnelle reconnaissant tous les mots qui contiennent exactement un unique facteur dans $L(e)$.