Rapport de stage

Synet

3 juin 2002 au 12 juillet 2002

Igor Rosenberg

Introduction

Mon sujet de stage était assez théorique (j'ai passé dix jours à lire des articles avant de regarder du code). Je vais donc commencer par des définitions, pour bien poser le domaine considéré.

1 Présentation des différents objets

1.1 Langages

Un langage est un ensemble de mots formés sur un vocabulaire. Pour un langage régulier (ou rationnel), on peut, plutôt que d'énumérer tous les éléments, énoncer les règles pour former les mots. Par exemple, le langage $L = \{a, aa, aaa, aaaa...\}$ peut être décrit par $aa*$. A l'aide du vocabulaire, et des opérateurs $*$ et $|$, on décrit le langage en entier.

Langage clos par préfixe : dans la clôture par préfixe, on considère tous les mots du langage, ainsi que tous les mots qui peuvent être préfixe d'un mot du langage : si $L = bbba*$, alors sa clôture par préfixe va être $\overline{L} = \epsilon|b|bb|bbba*$

1.2 Automates

Rappeler correspondance automate <->langage

Isomorphisme d'automate : deux automates sont isomorphes ssi il existe deux bijections (l'une sur les arcs, l'autre sur les états) entre les deux automates. Cela ne concerne pas les états initiaux et finaux. Dans la suite, on décide donc que tous les états sont accepteurs.

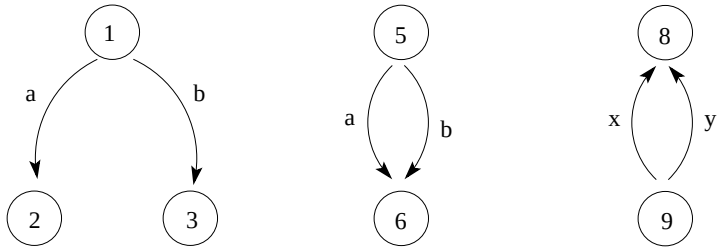
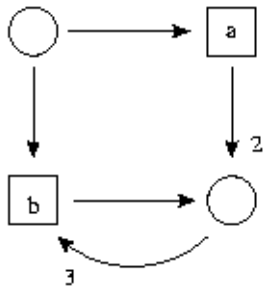


FIG. 1 – Le premier a trop d'états mais les deux derniers sont isomorphes

1.3 Réseaux

$N = (P, E, F)$ est un **réseau de Pétri** où

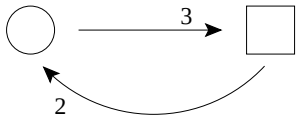
- P est un ensemble de places
- E un ensemble d'évenements ,
- E et P sont disjoints et
- $F : (P \times E) \cup (E \times P) \to \mathbb{N}$



En fait on a deux ensembles, avec un "graphe" entre ces deux ensembles, composé d'arcs, pondérés par un poids positif.

N est un réseau de **Pétri pur** si
$\forall p \in P \; \forall e \in E, \; F(p, e) = 0$ ou $F(e, p) = 0$.
D'un point de vue graphique, ça veut dire qu'on n'a pas de boucles.

Réseau pur



Réseau impur

Un **marquage** M est une fonction $M : P \to \mathbb{N}$. En fait, c'est dire combien de jetons se trouvent dans un place. On verra que c'est important pour le tirage

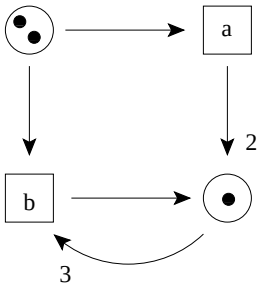
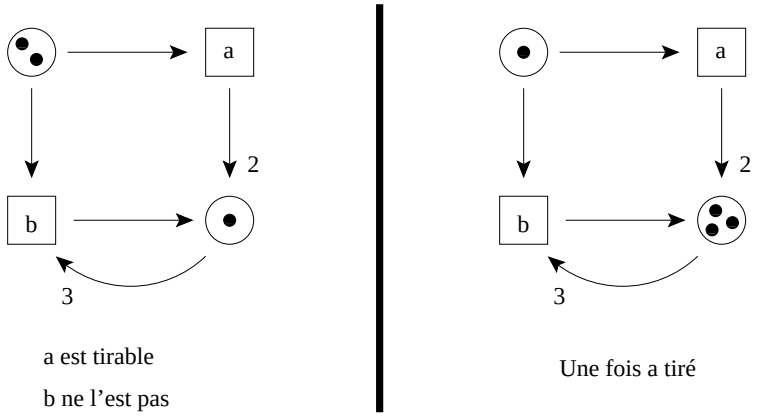


FIG. 2 – Un réseau et son marquage : $M = (2, 1)$

e **est tirable sur** M si $\forall p \in P, M(p) \geq F(p, e)$ Cela veut dire que pour être tirable, il faut qu'il y ait plus de jetons dans la place que le poids de

l'arc. Lorsque c'est le cas, l'évenement peut être tiré : on obteint la transition $M[e > M'$, avec M' un nouveau marquage, $M'(p) = M(p) - F(p,e) + F(e,p)$. (Si le réseau est pur, on ajoute, ou on soustrait mais pas les deux)



un réseau marqué : $N = (P, E, F, M_0)$ avec M_0 marquage initial.

L'ensemble des marquages accessibles, c'est $\{M : P \to \mathbb{N} | M_0[* > M\}$. En fait, c'est tirer toutes les transitions possibles, à partir du marquage initial.
Le **graphe des marquages accessibles** est l'automate $N^* = (MA(N), E, T, M_0)$, où

- $MA(N)$ est l'ensemble des marquages accessibles,
- $T = \{M \xrightarrow{e} M' | M, M' \in MA(N) et M[e > M'\}$ et
- M_0 le marquage initial.

C'est un automate déterministe, codéterministe, accessible, mais en général infini.

Un réseau est borné si son graphe des marquages accessibles est fini. Non-borné sinon.

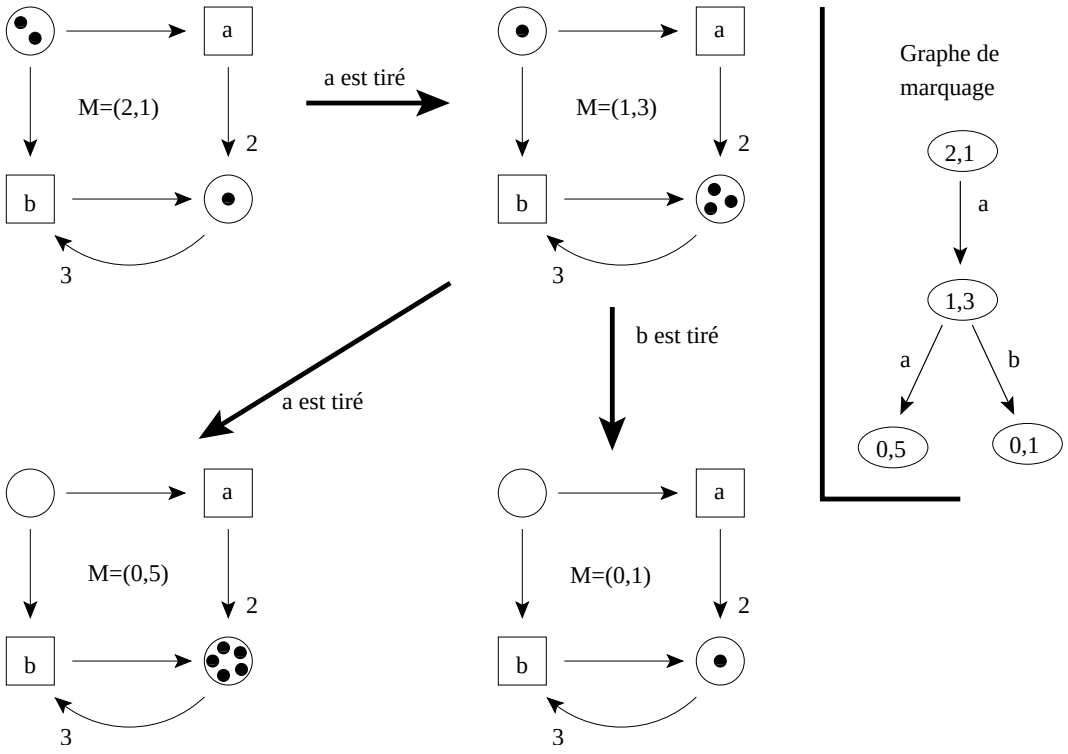


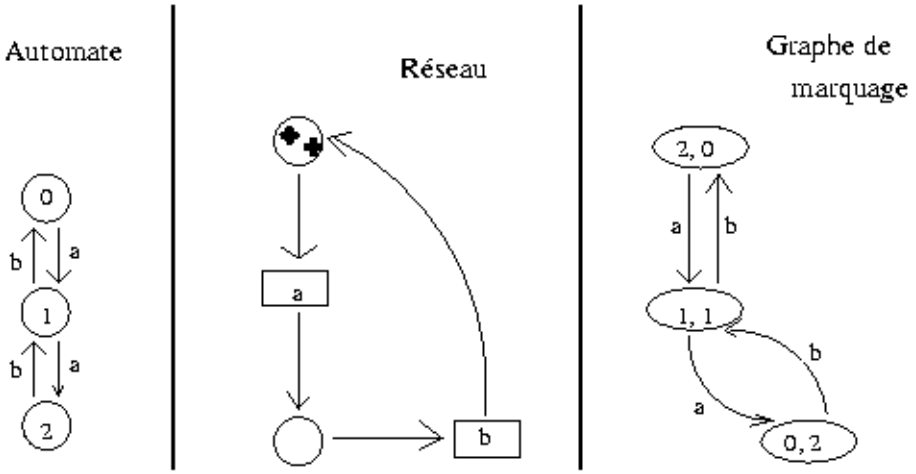
FIG. 3 – Graphe de marquage d'un réseau

2 Présentation de synet , logiciel de synthèse de réseaux

2.1 Le logiciel

Synet est un logiciel développé par Benoît Caillaud de S4 à l'IRISA. A partir d'un automate, on obtient le réseau qui se rapproche le plus de la solution, en s'appuyant sur le calcul de régions. On peut aussi, à l'aide de **synet** , fabriquer des réseaux distribuables, c'est-à-dire des réseaux qui peuvent être répartis sur des sites distincts et travailler de manière indépendante.

Un exemple : à partir d'un automate, on cherche à obtenir un réseau, qui, s'il existe, a un graphe de marquage isomorphe à l'automate de départ :



2.2 Historique

Six mois avant le début du développement de **synet** , a été écrit un logiciel pour synthétiser des réseaux élémentaires (presque comme des Réseaux de Pétri mais les places sont bornées par 1). Il s'appelle bf Petrify, écrit par Kishinevski, Cortella et Lavagno. On pouvait alors s'en servir pour synthétiser des circuits VLSI asynchrones : en pratique, ça nous donne des circuits életroniques (hardware) sans horloge. C'est très pratique pour faire du hardware, mais inappliquable au logiciel (software). En effet, les booléens sont idéaux pour les circuits, mais pas pour des programmes qui fonctionnnent avec des entiers (en fait, on souhaite synthétiser des réseaux de Pétri généraux). **Synet** a alors été developpé.

2.3 bons et mauvais points

Le concurrent de **synet** , avec des réseaux élémentaires, marche dans 100 % des cas, car le logiciel est capable de modifier le graphe de départ par étiquettage pour qu'une solution convienne. Par contre, **synet** ne donne pas toujours une réponse ; il faut alors réétiquetter l'automate de départ, mais on n'a pas d'indications par le logiciel. Parfois,le réétiquettage de quelques transitions suffit. Parfois, presque toutes les étiquettes doivent être changées. Mais **synet** n'a pas de restriction sur les réseaux générés : ils peuvent être généraux, non-bornés, impurs...

2.4 Support théorique

Synet est un logiciel tout à fait fiable dans le sens qu'il n'est construit que sur des théorèmes démontrés. Pour construire le réseau que l'on cherche, on introduit le concept de régions. Elles nous fourniront, grâce à la structure de l'automate, les places du futur réseau. On utilise aussi de l'algèbre linéaire sur des inéquations.

Le **théorème d'existence** est :

$$\text{Pour tout graphe } G$$
$$\exists \text{ réseau } N | N^* = G <=> G \text{ vérifie SSA et ESSA}$$

SSA (state separation axiom) : Il existe une région telle que
$s \neq s' \implies \sigma(s) \neq \sigma(s')$
ESSA (event/state separation axiom) : Il existe une région telle que
$s \xrightarrow{e} \implies \sigma(s) < \dot{} \eta(e)$

N^* est le graphe de marquage du réseau N.

Une **région** d'un automate $A = (S, E, T)$ c'est la donnée d'un triplet de fonctions $R = (\sigma, \dot{}\eta, \eta\dot{})$ avec

$$\sigma : S \to \mathbb{N} \quad , \quad \dot{}\eta : E \to \mathbb{N} \quad , \quad \eta\dot{} : E \to \mathbb{N}$$

et vérifiant les propriétés

- $s \xrightarrow{e} \implies \sigma(s) \geq \dot{}\eta(e)$
- $s \xrightarrow{e} s' \implies \sigma(s) = \sigma(s) - \dot{}\eta(e) + \eta\dot{}(e)$

On note M le marquage du réseau avec $M : P \to \mathbb{N}$ et le poids des arcs est la fonction $F : (P \times E) \cup (E \times P) \to \mathbb{N}$

Chaque place p d'un réseau détermine une région de son graphe de marquage par :

- $\sigma(M) = M(p)$
- $\forall e \in E \quad \dot{}\eta(e) = F(p, e)$ et
- $\forall e \in E \quad \eta\dot{}(e) = F(e, p)$

Inversement, chaque région d'un automate donne un réseau atomique. Le travail de synthèse consiste donc à fabriquer un ensemble de régions assez grand pour vérifier les deux axiomes en chaque point. On obtient alors (quand on a de la chance) un réseau grâce aux deux axiomes SSA et ESSA. On peut alors le réduire à sa forme minimal.

2.5 A quoi ça sert ?

2.5.1 Distribution

On part d'un protocole que l'on se donne. A partir de ça un automate se déduit très facilement. Avec **synet** , on obtiendrait une spécification qui permetrait de trouver une implémentation d'un réseau **distribué**. C'est un problème de distribution d'automates. On peut le faire parfois via la synthèse de réseau distribuable. Mais on n'obtient pas toujours une réponse, quand il y en a une.

2.5.2 Contrôle de supervision

Synet peut aussi servir pour synthetiser un contrôleur. On se donne la description d'un système sous la forme d'un graphe d'état, où on passe d'un état à un autre par des transitions qui représentent des actions. Dans ce système, il existe des actions contrôlables, que l'on peut donc empêcher, et d'autres incontrôlables, dont le mouvement nous est caché ainsi que des états interdits. On souhaite enlever du système (les rendre inaccessibles) les états interditsainsi que tous les états qui pourraient y mener par des actions incontrôlables. Ca se fait par rétropropagation. En fait, pour interdire des

accès, on peut créer des places de réseaux avec ..... " Les places à ajouter sont des régions du résidu ". En fait c'est **synet** mais sans l'axiome séparation-état-état.

2.6 Orientations futures ?

Il semblerait que la synthèse de réseaux distribuable ne soit pas efficace avec **synet** : trop souvent, le logiciel ne donne pas de réponse. Par contre, le contrôle est la voie présentie par l'équipe de recherche S4. Des extensions possibles seraient de permettre des données imprécises, par exemple en donnant seulement des parties de l'automate. L'entrée non pas d'une vision concrète des données, mais logique, est aussi envisagée : rentrée de règles de logique pour définir un automate.

3 Mon travail

J'avais un logiciel qui marchait dans les mains, et l'on m'a demandé de rajouter des fonctionnalités. J'ai passé beaucoup de temps à lire le code, pour savoir comment marchait l'algorithme, et repérer les endroits où il fallait que je fasse des modifications.

3.1 Synthèse de langage

On sait que expressions rationnelles (langages) et automates ont le même pouvoir expressif : un automate définit un langage régulier et vice-versa. Une extension possible de **synet** est d'adapter les algorithmes pour que l'on ne regarde plus l'équivalence entre l'automate de départ et le graphe de marquage du réseau, mais entre le langage définit par le graphe de marquage et le langage donné par l'automate du départ. Par exemple, les automates suivants sont distincts (pas le même nombre d'états) mais définissent le même langage **a**|**b** :

Dans le logiciel : Dans la comparaison des automates, il ne faut plus s'intéresser à la séparation état-état : si tous les arcs sont représentés, alors les langages seront équivalents. J'ai fait un double de l'algorithme original, et ensuite je l'ai modifié. Un problème est apparu : dans le nouvel algorithme, il y a un dépliage, [voir paragraphe suivant] où l'automate est modifié dans sa structure même. Les étiquettes des états changent de type : au lieu d'être des chaînes, elles deviennent des couples (chaîne - entier). Toute la suite du code était alors inopérante, vu que le type des objets avait changé. Benoît m'a alors aidé à implémenter une version objet de la description des états.

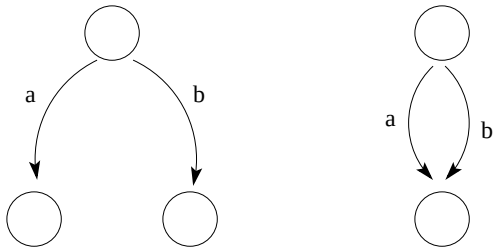


FIG. 4 – Deux automates définissant le même langage

Ainsi, on obtient des fonctions génériques qui s'appliquent sur des types différents. Aussi, les états étant maintenant des objets, il apparaît le problème de l'égalité entre états : ne sont égaux deux objets non pas si le contenu est le même, mais si ils ont la même adresse mémoire. Il a donc fallut créer une table de hachage.

Dépliage d'un automate Lorsque l'on donne un automate pour la synthèse de langage, il faut qu'il soit déplié. Cela signifie que les transitions sont toutes descendantes, ou alors qu'elles remontent dans la même branche de l'arbre. Voir figure 5.

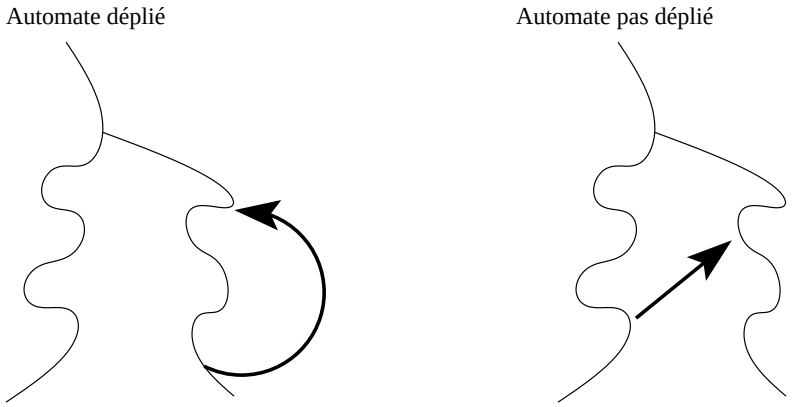


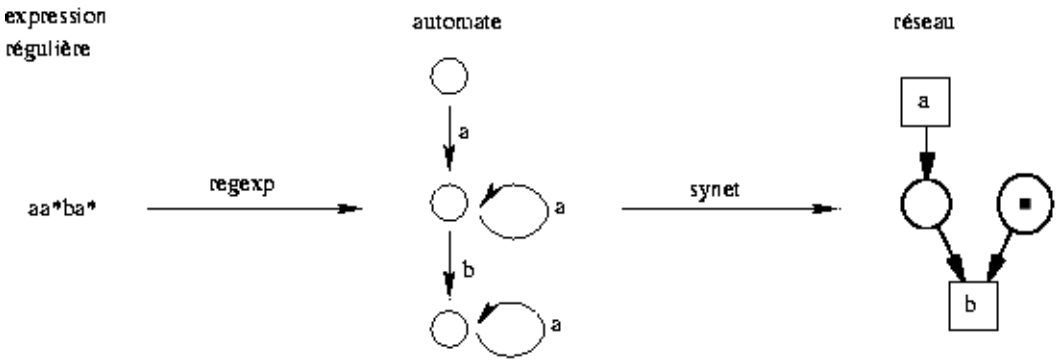
FIG. 5 – Dépliage

Mais ce dépliage pose un gros problème : des copies d'états sont créées, et le nombre de copies dépend de la complexité de l'automate. On numérote donc les états avec le nom précédent de l'état couplé à un naturel, le "numéro" de la copie. On ne se retrouve donc plus avec des états référencé par des chaînes, mais par des couples.

3.2 Langage donné par son expression régulière

Vu que dans la synthèse de langage, on ne regarde pas l'automate, mais le langage, pourquoi ne pas donner le langage de départ sous forme d'expression régulière ? La donnée initiale sera strictement la même au point de vue théorique, mais l'utilisateur peut préférer une visualisation à l'autre.

J'ai trouvé sur Internet un module nommé **regexp** : il transforme des expressions régulières en automates. Ce module m'a posé beaucoup de problèmes car les fonctionnalités que je recherchais (reconnaissance d'identificateurs de plusieurs caractères, automates étendus [pas de simplification 'a' et 'b' -> 'a b') n'était pas toutes là, ou ne faisait pas tout ce que je voulais. Il m'a donc fallu bidouiller là-dedans aussi, et ce travail-là a été fait un peu dans la précipitation.



3.3 Réseaux non bornés

Un réseau non-borné est un réseau dont le graphe de marquage n'est pas borné, c'est-à-dire que cet automate comporte un nombre infini d'états. Mais il est possible de décrire cet automate infini par une expression régulière finie (avec un nombre fini de caractères). Lors de la synthèse de langage, on veut parfois autoriser la génération d'un réseau non borné, vu qu'au final on ne s'intéresse qu'au langage. Il fallait implémenter cette option aussi.

Un exemple pour clarifier : Considérons le langage rationnel **aa*ba***. Il n'est pas possible d'obtenir un réseau non borné, mais par contre, le réseau suivant définit le même langage (le réseau n'est pas borné : il n'y a pas de limite sur le nombre de a tirés)

Dans le logiciel : Comme pour le langage, il m'a suffit de supprimer quelques lignes, et de rajouter des conditionnelles pour que les équations caractérisant la finitude se retrouvent court-circuitées.

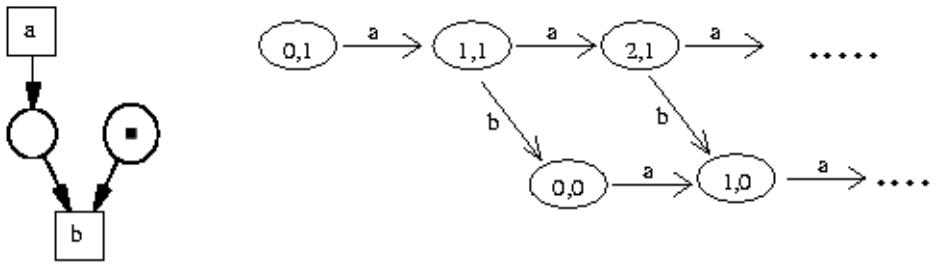


FIG. 6 – Un réseau non-borné et son graphe de marquage

3.4 Makefile

Comme **synet** est un gros projet, il est divisé en modules de quelques kilo-octets, et la compilation du logiciel se fait à l'aide de Makefiles. J'ai du à plusieurs reprises modifier ces fichiers pour pouvoir ajouter de nouveaux modules. Vu que c'était la première fois de ma vie que je tripotais des Makefiles, j'en ai fait un bazar assez monstrueux. J'y ai passé beaucoup de temps, sans en tirer de solution élégante. C'est une des choses que je regrette : ne pas avoir eu assez de temps (ni les connaissances) pour en faire un joli paquet-cadeau.

3.5 Au final

Le fait que j'ai eu le temps de faire autant de petites choses (synthèses de langage, expressions rationnelles, réseaux non bornés) en ces six semaines fait que ma contribution au logiciel n'a pas été vaine. Une partie de mes ajouts, après avoir été revus et corrigés, a été intégré dans le logiciel tel qu'il se présente à l'heure actuelle.

4 Conclusion ?

Utilité pour moi Ce stage m'a semblé très intéressant, m'obligeant à programmer de manière ordonnée et lisible par d'autres qui passeraient après. J'y ait appris une foule de choses en informatique et plus précisément en programmation, comme l'utilité réelle des objets, les Makefiles, et une connaissance plus détaillée de `caml`.

Utilité pour l'équipe de recherche Mon ajout de la gestion des expressions régulière a été utilisée par Philippe Darondeau de l'équipe S4 pour infirmer une conjecture.
On se donne un réseau, sur lequel après quelques tirages, on obtient comme vecteur de Parikh nX. La question est : sur le même réseau, existe-t-il un marquage initial tel que le vecteur X soit tirable ? (le problème initial était donné sous une autre forme équivalente)

La réponse est non, un contre-exemple est donné par : figure 7.

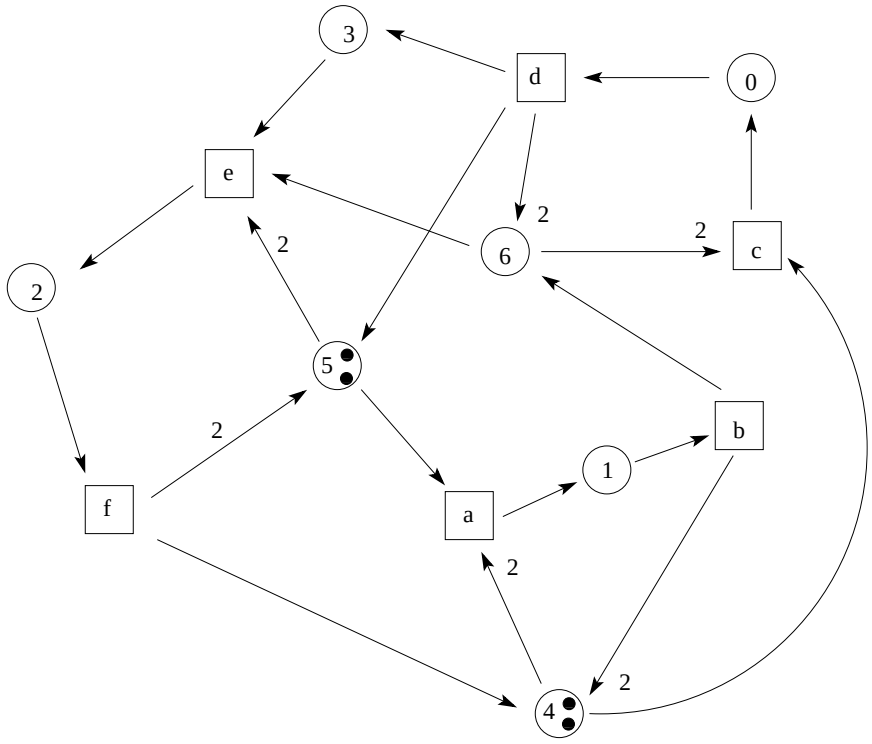


FIG. 7 – Un réseau à 7 places...

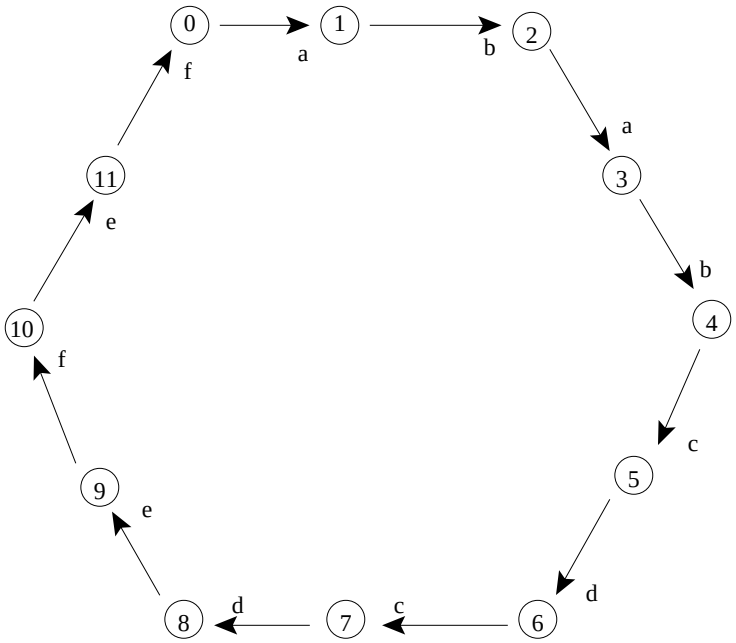


FIG. 8 – Et son graphe de marquage

Ce qu'on voit bien sur le cycle de marquage : En faisant un cycle complet, on trouve $2(a, b, c, d, e, f)$ mais il est impossible d'obtenir (a, b, c, d, e, f), car si l'on veut toutes les lettres, au moins deux sont doublées. Cet exemple, trouvé en quelques minutes avec **synet** , est beaucoup plus long à voir lorsque tout le travail est fait à la main...

Table des matières

Table des figures