Sorbonne Université
Campus Pierre et Marie Curie - 75005
Professeur référent : Alix MUNIER-KORDON | LIP6

Informatique
UE 2I013 - 2019

Rapport de projet

Génération aléatoire de réseaux de Petri pondérés vivants à choix libre

Emilie BIEGAS & Benjamin WEBER

Mots-clés

Génération de graphes, réseaux de Petri à choix libre, normalisation, génération aléatoire, calcul de marquages initiaux.

Key-words

Graph generation, Choice-Free Petri nets, normalization, random generation, initial marking computation.

Résumé

Les réseaux de Petri pondérés vivants à choix libre sont très utilisés pour la résolution de problèmes fondamentaux au sein des systèmes communicants. Y tester une propriété requiert de pouvoir générer aléatoirement ce type de réseaux. Chose qui jusqu'à maintenant n'existait pas.
Ce rapport aborde la résolution de ce problème au travers de quatre grands leviers que sont : - la génération aléatoire de graphe fortement connexe afin d'obtenir un SDF - la génération aléatoire de multiples valeurs dont la somme est fixée au préalable pour obtenir un T-semiflot aléatoire - le calcul d'un marquage initial vivant du SDF - et pour finir la transformation du SDF en Choice-Free.
La formalisation des réseaux, l'implémentation et les performances obtenues y seront également présentées.

1 Introduction

Ce projet a été réalisé dans le cadre d'un stage de 7 mois (de février à septembre 2019) à *Sorbonne Université* dans le département informatique LIP6 sous la direction d'Alix Munier-Kordon, enseignante-chercheuse.

Les réseaux de Petri sont largement utilisés depuis les années 1970. Leur construction simple ne leur enlève en rien de nombreuses propriétés mathématiques utiles. Ils sont utilisés, en outre, afin d'étudier le calcul distribué, qui n'est autre que la réalisation de calculs sur des unités de traitement séparées.
Les réseaux de Petri à choix libre, aussi dit Choice-Free, sont une sous-classe de réseau de Petri avec des propriétés mathématiques et algorithmiques permettant de résoudre plus facilement des questions fondamentales liées aux systèmes communicants.
Ainsi, les Choice-Free ont de nombreuses applications concrètes qui font de leur étude un sujet important, comme par exemple, la modélisation d'une chaîne de production de voitures dans laquelle plusieurs ateliers produisent les roues convergeant vers un même autre atelier pour y être assemblées.

Le stage avait donc comme objectif d'étudier les champs de recherches encore inexplorés des réseaux de Petri pondérés vivants à choix libre, aussi dit Weighted Choice-Free. Et, jusqu'à maintenant, leur génération aléatoire n'était pas possible. Cette génération aléatoire répond pourtant à un besoin en recherche lors de l'étude des propriétés de ces réseaux.
Alors, incité par l'existence d'un générateur aléatoire de SDF vivant (sous-classe des Weighted Choice-Free) développé par le laboratoire du LIP6, nommé *Turbine* [1], nous avons orienté nos recherches sur une possible transformation des SDF vers les Weighted Choice-Free, auparavant inexistante. *Turbine* datant de quelques années, nous avons choisi de reprendre intégralement la génération de SDF en cherchant de nouvelles solutions, plus optimisées.

La section 2 présente les réseaux étudiés en eux-même, ainsi que les outils mathématiques à notre disposition. La section 3 établit et formalise la transformation des SDF vers les Weighted Choice-Free. La section 4 présente pas à pas les grandes étapes rendant possible la génération aléatoire de Weighted Choice-Free vivant. Elle contient entre autres la présentation de quatre étapes, à savoir la génération de graphes fortement connexes dans l'optique de générer des SDF, la génération aléatoire d'un T-semiflot (autrement dit, générer aléatoirement de multiples valeurs dont la somme est fixée préalablement), le calcul d'un marquage initial vivant pour le SDF et la transformation du SDF vivant en Weighted Choice-Free vivant. La section 5 fait état de l'implémentation en elle-même. Et pour finir, la section 6 dévoile les résultats sur les performances de notre générateur.
Les annexes contiennent des exemples aux théorèmes et définitions présentées, ainsi que des explications plus détaillées sur le fonctionnement du générateur et sur son implémentation. Des résultats expérimentaux supplémentaires s'y joignent également.

2 Formalisation des réseaux étudiés

2.1 Synchronous DataFlow Graph

Le **Synchronous DataFlow Graph** [2] raccourcit en SDFG et plus couramment en SDF, est un formalisme utilisé à l'origine à des fins de modélisation des systèmes électroniques communicants.
Il s'agit d'un graphe biparti composé de deux types d'éléments que sont les processus, aussi dit tâches, et les FIFO, First In First Out buffer, aussi dit files. Les processus s'échangent, au travers des files chargées de temporiser les transferts, des jetons symbolisant l'échange de

paquets d'informations. Les arcs file-processus ou processus-file sont pondérées par une valeur entière strictement supérieure à 0.
Chaque file est mono-lectrice et mono-écrivaine, c'est-à-dire qu'une file doit avoir une seule entrée et une seule sortie.
Les processus sont graphiquement représentés par des cercles tandis que les files sont représentées par des rectangles.

Sur la figure 1, le processus t_1 consomme 4 jetons, et, pour chaque paquet de 4 jetons consommés, produit 1 jeton. Quant au processus t_2, il consomme 4 jetons et en produit 5.
Les files a_1 et a_2 permettent de temporiser l'arrivée des jetons. Par exemple, la file a_1 ne distribue aucun jeton avant d'en avoir 4. À ce moment, elle distribue 4 jetons d'un seul coup au processus t_2.
À l'étape initiale, la file a_2 dispose dors et déjà de 2 jetons à contrario de la file a_2 qui n'en dispose d'aucun.

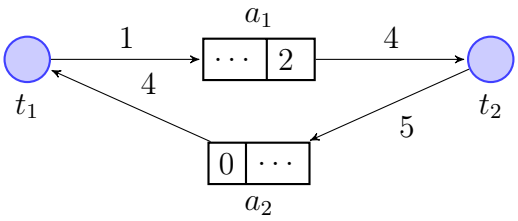


FIGURE 1 – Exemple de SDF composé de deux files a_1, a_2 et de deux processus t_1, t_2. La file a_1 possède 2 jetons à l'étape initiale.

Les processus peuvent être vus comme des unités de traitement indépendantes qui prennent un certain nombre d'informations en entrée et qui, après traitement, distribuent d'autres informations. À la différence près que la nature des informations n'a pas d'importance ici : les informations d'entrées et de sorties peuvent ne pas avoir la même forme. Dans les SDF, ce ne sont que des paquets d'informations qui entrent et qui sortent.
C'est la raison pour laquelle les SDF sont étudiés : ils permettent de modéliser l'échange d'informations dans un système entre différents éléments.

2.2 Réseau de Petri

Les SDF constituent une sous-classe des réseaux de Petri. Les réseaux de Petri étendent donc la définition des SDF.

Définition 1. *Un **réseau de Petri** [3] est un graphe biparti composé de transitions, représentées graphiquement par des rectangles, et de places, représentées par des cercles. Dans ces réseaux, les jetons sont graphiquement représentés par des points.*
*Si de plus, chaque arc place-transition et transition-place est pondéré par une valeur entière strictement positive, alors on parle de **réseau de Petri pondéré**.*

Dans un réseau de Petri, les places jouent le rôle des files dans les SDF et les transitions le rôle des processus. Par convention, la pondération de l'arc reliant une place p à une transition t est noté $W(p,t)$, et celui entre une transition t et une place p est noté $W(t,p)$.
De plus, un réseau de Petri est multi-lecteur et multi-écrivain. Autrement dit, dans un réseau de Petri chaque élément peut avoir plusieurs entrées et sorties.

Définition 2. *Marquage*
*Pour chaque place a, le nombre entier de jetons à un instant t est appelé **marquage** de la place a, noté $M_t(a)$, nombre devant en tout temps rester positif ou nul. En particulier, à l'étape*

initiale, on parle de ***marquage initial****, noté* $M_0(a)$.
Un réseau est dit ***marqué*** *si toutes les places possèdent un marquage initial.*

Puisque la relation de passage entre un réseau de Petri et un SDF est immédiate (comme le montre la figure ci-dessous), nous appliquerons par la suite les notations des réseaux de Petri aux SDF.

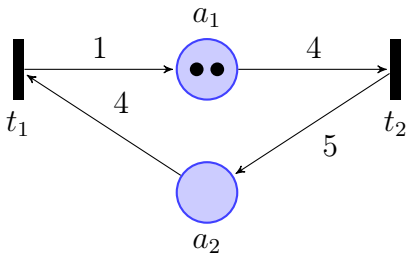
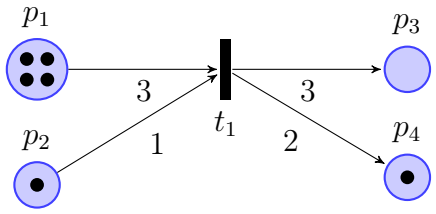


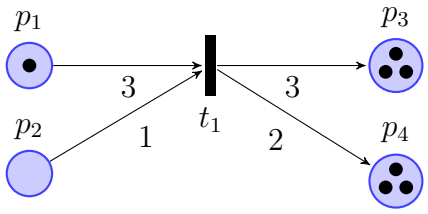
FIGURE 2 – Présentation du SDF de la figure 1 sous la forme d'un réseau de Petri

Le marquage initial, appuyé sur les valeurs de pondération, conditionne la vivacité du réseau. Autrement dit, l'enjeu du marquage initial est de trouver le bon nombre de jetons pour chaque place afin qu'à aucun moment il n'y ait un blocage, à savoir qu'après un certain nombre d'échanges, certaines places n'aient plus assez de jetons à distribuer, bloquant par la même occasion l'échange de jetons dans tout le réseau.

Par exemple, sur la figure ci-dessous, la place p_1 dispose de 4 jetons et p_2 possède 1 jeton. La transition t_1 peut donc être franchie. Cette transition va distribuer 3 jetons à la place p_3 et 2 à p_4.
Si la place p_1 avait eu moins de 3 jetons ou que p_2 avait eu moins de 1 jeton, la transition n'aurait pas pu être franchie et les places en sortie de t_1 auraient été bloquées, sauf si un nombre suffisant de jetons arrivent dans p_1 et p_2 après un certain moment.



(a) Exemple de marquage initial au temps t=0

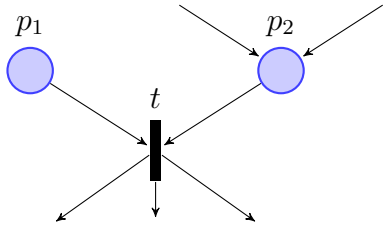


(b) Marquage obtenu au temps t=1

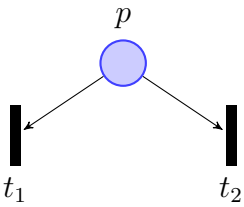
2.3 Réseau de Petri à choix libre pondéré (Choice-Free)

Définition 3. *Choice-Free*

Un réseau de Petri est dit à choix libre (Choice-Free) [4] si chaque place de ce réseau a exactement une transition en sortie. Le Choice-Free est multi-écrivain et mono-lecteur pour les places.



(a) Exemple de réseau à choix libre



(b) Exemple de réseau non à choix libre

Remarquons que les SDF forment une sous-classe des réseaux de Pétri à choix libre pondéré. Nous avons donc les relations suivante : SDF $\subset$ Weighted Choice-Free $\subset$ Weighted Petri net $\subset$ Petri net.

2.4 Définitions & Propriétés

Définition 4. *Vivacité*

Un réseau de Petri est dit ***vivant*** *s'il vérifie la condition de vivacité, à savoir chaque transition peut être exécutée une infinité de fois.*
Dans les cas des SDF et des Choice-Free pondérés, cette condition peut se reformuler, non trivialement, en : il existe un temps au bout duquel le réseau retrouve son marquage initial. C'est la ***réversibilité*** *du réseau. Dans ces cas, le T-semiflot est appelé* ***vecteur répétition****.*

On remarque ainsi que la vivacité d'un Choice-Free pondéré (et d'un SDF) dépend entièrement du marquage initial de ce dernier.

Définition 5. *T-semiflot*

On pose C la matrice places-transitions de taille $n*m$, où chaque ligne représente une place et chaque colonne une transition, telle que :

$$\forall p_i \text{ place}, i \in [\![1,n]\!], \forall t_k \text{ transition}, k \in [\![1,m]\!], c[i,k] = W(t_k,p_i) - W(p_i,t_k)$$

où $c[i,k]$ est le coefficient d'indice (i,k) de la matrice C.
Le ***T-semiflot*** *est un vecteur colonne Y non nul de taille $m*1$ vérifiant $C*Y=0$.*

Un exemple de calcul du T-semiflot est disponible en annexe A.

S'il existe un T-semiflot, on dit que le réseau est **consistant**. Si le réseau est de plus fortement connexe, alors il est **bien formé**.
Si un Choice-Free est pondéré et vivant alors, avant de revenir au marquage initial, chaque transition est exécutée exactement le même nombre de fois que la composante correspondante du T-semiflot.

Définition 6. *Séquence réalisable*

Une ***séquence*** *est une exécution séquentielle des transitions, toute ou en partie, du réseau. Par exemple, dans un réseau de 5 transitions notées $(t_i)_{i\in[\![1,5]\!]}$, (t_1,t_2,t_1,t_2,t_3) est une séquence dans laquelle la transition t_1 s'exécute en premier, puis t_2 s'exécute, etc.*
Une ***séquence réalisable*** *est une séquence pour laquelle, à chaque exécution d'une transition, le nombre de jetons dans chaque place reste positif.*
Pour un réseau de Petri vivant, il existe au moins une séquence réalisable infinie.

Définition 7. *Jetons utiles*

Pour toute place p, on appelle $pgcd_p$ le pgcd des pondérations des arcs entrants et sortants de cette dernière.
Dans un réseau, si le marquage initial de chaque place p n'est pas un multiple entier de $pgcd_p$,

alors certains jetons ne seront pas utiles. Autrement dit, dans chaque place il restera un certain nombre de jetons qui ne fera qu'attendre, n'étant pas assez nombreux pour franchir les transitions de sorties. Les autres jetons sont appelés des ***jetons utiles****.*

Définition 8. *Normalisation d'un SDF*

Pour un SDF, la ***normalisation*** *[5] consiste à trouver pour chaque transition une valeur des ses poids d'entrée et de sortie qui soit la même, tout en conservant les contraintes de précédence (pour une place p qui a comme entrée la transitions t_i et comme sortie t_j, alors t_i doit être exécutée avant t_j).*

La normalisation d'un Choice-Free pondéré n'existe pas, puisqu'un jeton peut arriver dans une place depuis plusieurs transitions. Autrement dit, pour une place on ne sait pas quelle transition en entrée est exécutée avant la transition de sortie. Autrement dit, on ne peut pas définir de relation de précédence entre les transitions entrantes d'une même place.

Théorème 1. *Tout SDF consistant est normalisable et se calcule selon la formule ci-dessous.*

Soit un réseau de Petri comportant $k \geq 1$ transitions, notées $(t_i)_{i\in[\![1,k]\!]}$ de T-semiflot $Y = Y(t_1)...Y(t_k)$.
Dès lors, la valeur normalisée pour une transition t_l, avec $l \in [\![1,k]\!]$, notée $Z(t_l)$, est reliée à $R := ppcm_{i\in[\![1,k]\!]}\big(Y(t_i)\big)$ par la relation suivante : $Z(t_l) = \frac{R}{Y(t_l)}$

Un exemple de normalisation est disponible annexe D.

Théorème 2. *Condition suffisante de vivacité pour un SDF*

Soit un SDF normalisé à n places.
Pour toutes places p_i, $i \in [\![1,n]\!]$, on pose $t_e(p_i)$, respectivement $t_s(p_i)$, la transition d'entrée de p_i, respectivement de sortie.

Alors une condition de vivacité suffisante [5] est :

$$\textit{Pour tout circuit } c\ , \sum_{p_i\in c} M_0(p_i) > \sum_{p_i\in c}\big(W(p_i,t_s(p_i)) - pgcd\big(W(p_i,t_s(p_i)), W(t_e(p_i),p_i)\big)\big)$$

Un exemple est disponible en annexe B.

3 Transformation Synchronous DataFlow vers Choice-Free

Grâce à *Turbine*, on sait qu'il est dors et déjà possible de générer aléatoirement des SDF vivants vérifiant la condition de vivacité (théorème 2). Ainsi, l'objectif de cette partie est de trouver une transformation permettant de passer d'un SDF vivant à un Choice-Free vivant. La transformation qui suit s'inspire d'une existante [6] donnant une relation de passage d'un Choice-Free vivant à un SDF vivant.
Lors de cette transformation, on souhaite que les deux réseaux conservent le même T-semiflot, puisque l'existence de ce dernier (sa consistance) est une condition nécessaire à l'existence d'un marquage initial vivant. On souhaite également qu'une séquence réalisable pour le SDF, le soit aussi pour le Choice-Free. Ainsi, cela suffira à garantir la vivacité du Choice-Free.
Succinctement, l'idée de la transformation est de fusionner plusieurs places en entrée d'une même transition en changeant les valeurs de pondération d'entrée et de sortie, et le marquage initial de ladite place.

Pour trouver une relation de passage entre les deux graphes (la transformation d'un SDF vers un Choice-Free étant celle qui nous est essentielle pour la suite), nous appuierons nos démonstrations des figures 5, représentant un sous graphe d'un SDF, et 6, représentant quant

à elle le sous graphe souhaité d'un Choice-Free après transformation. Les notations de ces deux figures seront conservées pour la suite. Les transformations sont valables pour un nombre de transition $k \geq 2$.
Pour le Choice-Free on note : $\vec{a}(x, y)$ l'arc orienté reliant l'élément x à y, et $\mathscr{T}$ l'ensemble des transitions du réseau. On conserve les mêmes notations indicées par $'$ pour le SDF.

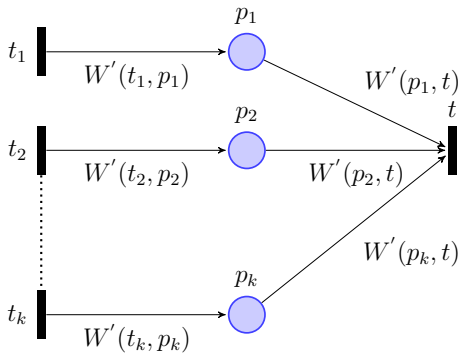


FIGURE 5 – Illustration d'un SDF menant à la transformation.
Chaque place pi, $i \in [\![1, k]\!]$ où $k \geq 2$, a un marquage initial noté $M_0'(p_i)$.

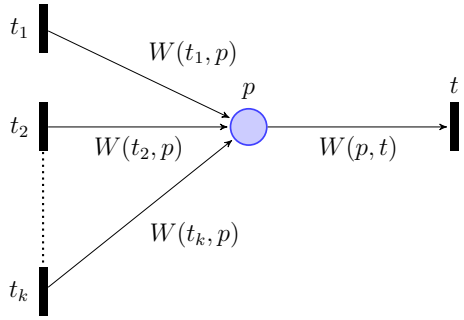


FIGURE 6 – Illustration de la transformation souhaitée en partant d'une situation présentée sur la figure 5 pour $k \geq 2$ transitions. La place p a un marquage initial noté $M_0(p)$.

Soient $Y = Y(t_1)...Y(t_k)$ un T-semiflot du Choice-Free et $Y' = Y'(t_1)...Y'(t_k)$ un T-semiflot du SDF.
On suppose que le SDF est normalisé (la transformation d'un SDF quelconque vers un normalisé étant connue). De plus, afin de conserver des propriétés similaires au niveau de la périodicité, nous supposons que les deux graphes ont le même T-semiflot.

Calcul des poids

On pose pour tout $i \in [\![1, k]\!]$, $Z'(t_i)$ la valeur normalisée de la pondération de l'arc $\vec{a}(t_i, p)$ du SDF et en remarquant que, par hypothèse, le SDF est normalisé (c'est-à-dire $W'(p_1, t) = ... = W'(p_k, t)$), on pose $Z'(t)$ celle d'un des arcs $\vec{a}(p_i, t)$. Ainsi par définition :

$$\forall i \in [\![1, k]\!], \begin{cases} Z'(t_i) = W'(t_i, p_i) \\ Z'(t) = W'(p_i, t) \end{cases}$$

Alors, pour tout $a \in [\![1, k]\!]$, on a $Z'(t) = w'(p_a, t)$. Afin de faciliter l'écriture, on fixe a pour la suite.

Puisque le SDF est normalisé (d'après la relation entre le T-semiflot et les valeurs des poids normalisées) :

$$\begin{cases} \forall i [\![1,k]\!], Y^{'}(t_i) = \frac{R^{'}}{Z^{'}(t_i)} \\ Y^{'}(t) = \frac{R^{'}}{Z^{'}(t)} \end{cases}$$

où $R^{'} := ppcm_{t_i \in \mathscr{T}}\big(Y^{'}(t_i)\big)$

Puis comme par hypothèse $Y = Y^{'}$ on obtient pour le Choice-Free :

$$\begin{aligned} \sum_{i=1}^{k} W(t_i,p) * Y(t_i) = W(p,t) * Y(t) &\iff \sum_{i=1}^{k} W(t_i,p) * Y^{'}(t_i) = W(p,t) * Y^{'}(t) \\ &\iff \sum_{i=1}^{k} \frac{W(t_i,p)}{Z^{'}(t_i)} = \frac{W(p,t)}{Z^{'}(t)} \end{aligned}$$

En posant :

$$\boxed{\begin{cases} W(t_i,p) := Z^{'}(t_i) = W^{'}(t_i,p_i) \\ W(p,t) := k * Z^{'}(t) = k * W^{'}(p_a,t) \end{cases}} \tag{1}$$

On obtient bien :

$$\sum_{i=1}^{k} \frac{W(t_i,p)}{W^{'}(t_i,p_i)} = k = \frac{W(p,t)}{W^{'}(p_a,t)}$$

Calcul d'un marquage initial

Puisque l'on souhaite que la transformation conserve la vivacité, nous supposons qu'une séquence réalisable pour le SDF l'est aussi pour le Choice-Free.
Par définition d'un réseau de Petri, pour chaque place le nombre de jetons doit en tout instant être positif ou nul.
Soit σ une séquence de tir réalisable pour le SDF.

On pose $\delta^{'}(\sigma,t)$, respectivement $\delta(\sigma,t)$, le nombre de fois que la transition t est exécutée dans la séquence σ pour le SDF, respectivement pour le Choice-Free.
Alors pour le SDF :

$$\forall i \in [\![1,k]\!], M_0^{'}(p_i) + W^{'}(t_i,p_i) * \delta^{'}(\sigma,t_i) - W^{'}(p_i,t) * \delta^{'}(\sigma,t) \geq 0$$

En sommant ces k équations on obtient :

$$\sum_{i=1}^{k} M_0^{'}(p_i) + \sum_{i=1}^{k} W^{'}(t_i,p_i) * \delta^{'}(\sigma,t_i) - \sum_{i=1}^{k} W^{'}(p_i,t) * \delta^{'}(\sigma,t) \geq 0$$

Or comme le SDF est normalisé on a l'égalité $W^{'}(p_i,t) = W^{'}(p_a,t)$ pour tout $i \in [\![1,k]\!]$. D'où

$$\sum_{i=1}^{k} M_0^{'}(p_i) + \sum_{i=1}^{k} W^{'}(t_i,p_i) * \delta^{'}(\sigma,t_i) - k * W^{'}(p_a,t) * \delta^{'}(\sigma,t) \geq 0$$

Puisque la séquence σ est réalisable pour le SDF alors elle l'est pour le Choice-Free par hypothèse. D'où $\delta(\sigma, x) = \delta^{'}(\sigma, x)$. Alors, une inégalité vérifiée par σ est :

$$M_0(p) + \sum_{i=1}^{k} W(t_i, p) * \delta(\sigma, t_i) - W(p, t) * \delta(\sigma, t) \geq 0$$

Puis :

$$M_0(p) + \sum_{i=1}^{k} W^{'}(t_i, p_i) * \delta^{'}(\sigma, t_i) - k * W^{'}(p_a, t) * \delta^{'}(\sigma, t) \geq 0$$

Les deux inégalités étant identiques au marquage initial près, en posant l'égalité qui suit on s'assure que l'inégalité précédente est vérifiée :

$$\boxed{M_0(p) := \sum_{i=1}^{k} M_0^{'}(p_i)} \tag{2}$$

Conclusion

Soit $f : S \longrightarrow CF$ la transformation d'un SDF pondéré vivant S en un Choice-Free pondéré vivant CF selon les formules 1 et 2.

Lemme 1. *La transformation f conserve les T-semiflot. À savoir, pour un SDF pondéré vivant, noté S, S et $f(S)$ ont le même T-semiflot.*

Démonstration. Voir annexe E □

Lemme 2. *Soit S un SDF pondéré vivant.*
Alors toute séquence réalisable de S l'est également pour $f(S)$.

Démonstration. Voir annexe F □

Théorème 3. *Si S est un SDF normalisé vivant alors $CF = f(S)$ est vivant.*
En particulier, si S vérifie la condition suffisante de vivacité (théorème 2) alors CF est vivant.

La preuve du théorème est immédiate d'après les lemmes 1 et 2.

4 Génération d'un Choice-Free vivant

La création d'un Weighted Choice-Free vivant se scinde en quatre grandes étapes : premièrement, la création d'un SDF, dont l'étape déterminante pour ce faire est l'obtention d'un graphe fortement connexe, puis vient la génération aléatoire d'un T-semiflot permettant de normaliser le SDF, ensuite le calcul d'un marquage initial pour le SDF, pour finir avec la transformation du SDF vivant en Choice-Free vivant.
Le générateur prendra alors comme entrée le nombre de noeud, la densité du graphe et la somme des composantes du T-semiflot voulu.

4.1 Génération d'un SDF

4.1.1 Génération d'un graphe fortement connexe

La création d'un graphe fortement connexe est un problème difficile de par le fait que la recherche de composantes fortement connexes est exponentielle. Toutefois, un algorithme proposé par M. Maueur [7] et appuyé sur l'algorithme de Tarjan pour la recherche des composantes fortement connexe, dont la validité n'a pas été prouvée, permet de rendre un graphe simplement connexe en un graphe fortement connexe, avec une complexité linéaire.

Pour cet l'algorithme, il était préconisé d'utiliser des arborescences comme entrée. Néanmoins, après des expériences (voir annexe G), il s'est avéré que le nombre d'entrées pour chaque nœud était bien trop faible pour satisfaire la situation présentée lors de la transformation d'un SDF en Weighted Choice-Free (figure 5).
Alors, nous avons implémenté notre propre générateur de graphe connexe, prenant comme paramètres le nombre de noeuds n et la densité $d = \frac{m}{n(n-1)}$ où m est le nombre d'arcs du graphe.

L'idée est de tirer aléatoirement les arcs entre les noeuds selon le degré entrant et sortant de ces derniers. Succinctement, chaque noeud a une probabilité d'être tiré inversement proportionnelle à son nombre d'entrées ou de sorties selon qu'il est considéré comme une entrée ou comme une sortie. Autrement dit, plus un noeud aura d'entrées, respectivement de sorties, moins il aura de chance d'être tiré comme entrée, respectivement comme sortie, pour un autre noeud.
Selon le nombre d'arcs m du réseau, on fixe une valeur moyenne du degré entrant et sortant, notée avg_degree, de sorte que $\sum_{i=1}^{n} avg\_degree \approx m$. Puis, au moment de tirer les noeuds d'entrée et de sortie pour un noeud, on choisit aléatoirement un nombre plus ou moins proche de avg_degree (selon une valeur de dispersion fixée) comme degré entrant et un autre pour le degré sortant. Il est à noter qu'en sortant de cet algorithme, chaque noeud à un degré entrant et sortant au moins égale à 1.
Lorsque le nombre d'arcs souhaités est atteint, on effectue un parcours en profondeur du graphe pour le rendre connexe.
Le fonctionnement détaillé de l'algorithme est présenté annexe I.

Après analyse, on constate que le nombre moyen d'entrées par nœud correspond environ à celui fixé lors de la construction, comme le montre la figure ci-dessous. Toutefois, il est à noter que ce nombre s'éloigne plus il se rapproche du nombre de noeud, et ce car la densité se rapproche de 1. Et, en cause de cet écart se trouve la dispersion (proportionnelle au nombre d'arcs, qui augmentent plus la densité augmente) du tirage aléatoire autour de avg_degree.

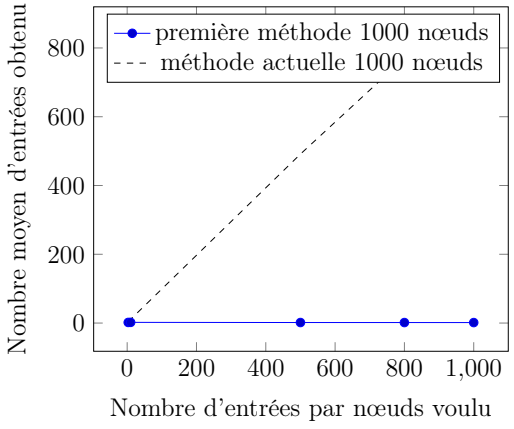
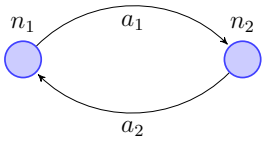


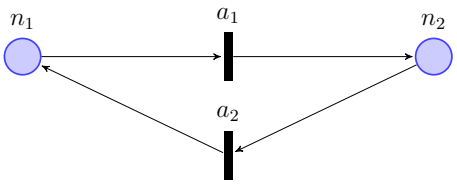
FIGURE 7 – Moyenne créée sur une génération aléatoire de 100 graphes

4.1.2 Transformation en SDF

La transformation d'un graphe fortement connexe en SDF est immédiate puisque chaque nœud devient une transition et chaque arc une place, comme illustré sur la figure suivante :



(a) Graphe fortement connexe



(b) SDF après conversion

4.2 Génération d'un T-semiflot

On dispose désormais d'un SDF fortement connexe. Il nous faut maintenant normaliser ce dernier afin de calculer un marquage vivant par la suite. Pour ce faire, nous devons disposer d'un T-semiflot nous permettant alors pour chaque transition du réseau de mettre la pondération des arcs entrants et sortants de cette dernière à la valeur correspondant à cette dite transition dans le T-semiflot.
La somme des composantes du T-semiflot est un paramètre qu'il est nécessaire de pouvoir contrôler du fait que plus ses composantes sont grandes, plus les séquences admissibles seront conséquentes à simuler. Autrement dit, la somme des composantes du T-semiflot est un paramètre de complexité pour les algorithmes. C'est pourquoi, cette somme est une entrée standard des algorithmes sur ces réseaux.
Ainsi, ci-dessous est énoncée une méthode permettant la génération aléatoire de plusieurs entiers naturels dont la somme est fixée. Il est à noter qu'à l'heure actuelle il s'agit d'un problème ouvert.

Différentes méthodes pour la génération des poids ont été envisagées (voir annexe H). Toutefois, à cause de la non-uniformité des résultats ou du dépassement de la somme fixée, nous n'avions pas abouti à une solution correcte.
Le dépassement de la somme fixée avec la méthode précédente, donnant néanmoins des résultats uniformes, est dû à l'intervention d'une partie entière supérieure dans le calcul qui augmente chaque valeur, plus que nécessaire. Intuitivement, l'alternance entre parties entières inférieure et supérieure pour les valeurs des x_i devrait permettre de résoudre ce problème.

Pour la suite on note S pour la somme fixée et x_i pour la valeur de la case d'indice i du tableau des composantes du T-semiflot de taille n.
On pose $\sum_{i=1}^{n} x_i := A$ et $f = \frac{S-n}{A}$

En tirant les valeurs des x_i dans $[\![1, S]\!]$, on pose alors :

$$N_i = \begin{cases} \lceil x_i * f \rceil + 1 \text{ si } i \text{ pair} \\ \lfloor x_i * f \rfloor + 1 \text{ sinon} \end{cases}$$

L'ajout de 1 permet de s'assurer que tous les x_i ont une valeur au moins égale à 1 (et non à 0). En réalité, seul celui associé à la partie entière inférieure est nécessaire mais cela introduirait des calculs en flottants supplémentaires.
On obtient donc :

$$\sum_{i=1}^{n} N_i \approx \sum_{i=1}^{n} \left(x_i * f + 1\right) \approx f * A + n \approx S$$

Finalement, comme le montre la figure ci-dessous, cette méthode permet d'obtenir une distribution quasiment uniforme avec une somme proche de la norme du vecteur.

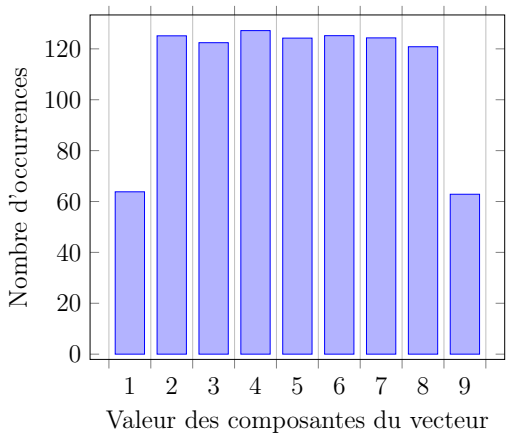


FIGURE 9 – Moyenne faite sur 100 essais - (nombre nœuds, somme fixée, somme atteinte)=(1000, 5000, 5005.53)

Une fois les composantes du vecteur obtenues, il suffit de s'assurer que le plus grand diviseur commun entre ces dernières vaut 1. En fait, puisqu'il existe une infinité de T-semiflot, il est plus intéressant de choisir le plus petit.

4.3 Calcul d'un marquage initial vivant pour le SDF

Une résolution possible [1], et pour le moment la seule envisageable, pour s'assurer de la vivacité du SDF est de résoudre un système linéaire en nombre entiers dont les contraintes sont exprimées pour chaque arc, et non plus pour chaque sous circuit du graphe (qui est un problème de complexité exponentielle). Un exemple est disponible en annexe C.
Pour chaque place p_i du graphe, comportant p places, comme défini sur la figure 10, on introduit deux variables réelles propre à la transition t_j, respectivement t_k où $k \neq j$, notée $\gamma_j \in \mathbb{R}$, respectivement $\gamma_k \in \mathbb{R}$, tel que $\gamma_k - \gamma_j + M_0(p_i) > Z_k - pgcd(Z_k, Z_j)$.

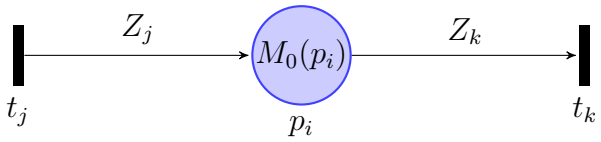


FIGURE 10 – Paramétrisation du problème de vivacité sur les arcs

La résolution de ce système est trop longue pour l'envisager telle quelle. À la place, le système est résolu en tant que système réel en posant $x_i \in \mathbb{R}^+$ comme nouveau marquage initial. La résolution de système linéaire étant valable pour des inégalités larges, on introduit un biais $\epsilon > 0$ faible (que l'on pose ici à $\epsilon = \frac{1}{p}$) :

$$\text{Minimiser } \sum_{i=1}^{p} x_i$$
$$\text{contraint à :}$$
$$\begin{cases} \gamma_k - \gamma_j + x_i - \epsilon \geq Z_k - pgcd(Z_k, Z_j) \\ x_i > 0, \gamma_k \in \mathbb{R}, \gamma_j \in \mathbb{R} \\ M_0(p_i) = \lceil \frac{x_i}{pgcd(Z_k, Z_j)} \rceil * pgcd(Z_k, Z_j) \in \mathbb{N} \end{cases} \tag{3}$$

Alors, pour tout circuit $(t_1, p_1, t_2, p_2, ..., t_m, p_m, t_1)$, avec $(t_i)_{i \in [\![1,m]\!]}$ les transitions et $(p_i)_{i \in [\![1,m]\!]}$ les places, en sommant les m inégalités et en notant Z_j^i le poids de l'arc entrant dans la place

p_i, pour tout $i \in [\![1, m]\!]$, et Z_k^i le poids de celui sortant, on obtient :

$$\begin{cases} \sum_{i=1}^{m} x_i \geq \sum_{i=1}^{m} \left(Z_k^i - pgcd(Z_j^i, Z_k^i)\right) + m * \epsilon > \sum_{i=1}^{m} \left(Z_k^i - pgcd(Z_j^i, Z_k^i)\right) \\ \left(\forall i \in [\![1, m]\!], x_i \leq \lceil \frac{x_i}{pgcd(Z_k, Z_j)} \rceil * pgcd(Z_k, Z_j)\right) \Longrightarrow \sum_{i=1}^{m} x_i \leq \sum_{i=1}^{m} M_0(p_i) \end{cases}$$

La condition de vivacité est ainsi vérifiée.

4.4 Transformation en Choice-Free

La dernière étape est de transformer le SDF vivant et consistant en Weighted Choice-Free vivant comme défini par la transformation du théorème 3. On dispose d'un SDF normalisé et vivant comme définit à la figure 5. Il est désormais nécessaire de fusionner les places entre elles, selon un partitionnement aléatoire.

Le principe de partitionnement est le suivant. Supposons que l'on dispose d'une transition t avec $k \geq 1$ places en entrées. Dès lors, on tire aléatoirement un nombre $nb_p \geq 1$ de partitions sur l'ensemble $\mathcal{E} := \{i | i \in [\![1, k]\!]\}$ tel que $nb_p \leq card(\mathcal{E})$. Pour chaque partition p_i, avec $i \in [\![1, nb_p]\!]$, on tire aléatoirement $nb_p(p_i)$ éléments deux à deux distincts de $\mathcal{E}$ à mettre dans cette partition.
On a lors : $\bigcup_{i=1}^{nb_p} p_i = \mathcal{E}$ et $\bigcap_{i=1}^{nb_p} p_i = \emptyset$.
Puis, en indiçant les places en entrée de t de 1 à k, on fusionne entre elles les places dont l'indice se trouve dans la même partition.

Le standard utilisé pour les réseaux de Petri étant le PNML [8], une fois la génération de Choice-Free terminé, il faudra exporter le réseau à ce format.
Les fichiers de sortie au format PNML ont ensuite été vérifiés à l'aide du logiciel libre développé par le *LIP6* [9].

5 Implémentation

Le générateur a été implémenté en C, et en Python pour l'interface graphique. Il contient approximativement 5000 lignes de code.

5.1 Pré-requis

Les codes sources sont compilés avec le standard C99. Il est également nécessaire de disposer de la librairie *libglpk-dev* (version 4.61-1) ; librairie du solveur GLPK utilisée pour résoudre le problème du marquage initial.

5.2 Organisation du code

Voir annexe J

5.3 Structures utilisées

La transformation d'un SDF en Choice-Free demande, pour chaque transition, de retrouver les places en entrées, puis de supprimer ces places pour les fusionner. En tenant compte de la faible densité des graphes, le nombre d'entrées et de sorties par nœud est très faible par rapport au nombre total de nœuds, le choix a donc été fait de rendre accessible pour chaque élément (place/transition) du réseau leurs entrées et leurs sorties.
Une structure a donc été mise en place, plus lourde en mémoire que deux matrices stockant les poids d'entrées et de sorties, mais facilitant la manipulation des réseaux dans le code.

Des explications détaillées sur les structures principales du code se trouvent en annexe K.

6 Résultats expérimentaux

Chaque test a été réalisé 100 fois afin de faire une moyenne. Le nombre d'entrées et de sorties varie entre 2 et 5. La moyenne des composantes du T-semiflot est de 5. La machine utilisée est un *Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz* 64-bits avec 4Go de RAM.

Temps moyen (en s) - nombre moyen entrées 5/nombre moyen sorties 5				
Nombre de transitions	10	100	1000	10000
Création graphe connexe	0.000016	0.000544	0.041428	4.849092
Transformation fortement connexe	0.000003	0.000036	0.000272	0.004467
Conversion graphe/SDF	0.000023	0.000252	0.002188	0.037861
Génération poids	0.000001	0.000015	0.000047	0.000386
Normalisation	0.000002	0.000018	0.000284	0.009046
Calcul marquage	0.000319	0.021249	5.625320	1386.603827
SDF vers Choice-Free	0.000025	0.000259	0.002868	0.056213
Libération mémoire	0.000010	0.000079	0.002494	0.070104
Total	0.000399	0.022452	5.674901	1391.630996

À la vue de ces résultats, la première constatation est la validité de la structure utilisée pour manipuler les réseaux de Petri puisque la taille du réseau n'impacte que très peu la création de la structure à l'étape *Conversion graphe/SDF* et la manipulation de cette dernière à l'étape *SDF vers Choice-Free*.
De surcroît, l'étape prépondérante dans la génération d'un graphe est la résolution du marquage initial. Celle-ci prend d'autant plus de temps que le nombre de places (lié au nombre d'arcs, donc d'entrées et sorties) augmente. D'autres expérimentations sont disponibles en annexe L.

Le générateur de SDF *Turbine* [1] étant la référence en la matière, il est intéressant de comparer les résultats obtenus. Malheureusement, il fut impossible de proprement télécharger le générateur, dont certains composants semblent obsolètes.

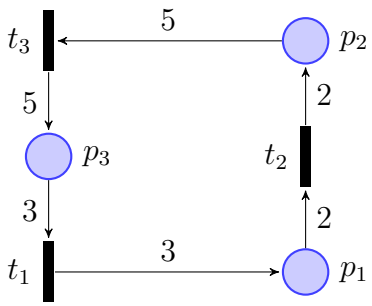
7 Conclusion

Pour la première fois, la transformation d'un SDF vivant et consistant vers un Weighted Choice-Free a été détaillée. Cette transformation a ensuite pu être utilisée pour concevoir le tout premier générateur aléatoire de Weighted Choice-Free vivant.
Ce générateur résout de nombreux problèmes ouverts auxquels il a fallu construire une solution comme la génération aléatoire de graphe fortement connexe à l'aide de l'algorithme de Tarjan pour la recherche des composantes fortement connexe et la génération aléatoire de poids pour le T-semiflot.
En outre sa portabilité sur diverses plateformes, grâce à son implémentation en C, ce générateur permet également d'être utilisé comme entrée dans les algorithmes de la communauté des réseaux de Petri, et ce grâce à sa sortie au format PNML. Ce travail à également permis de mettre à jour le travail sur la génération de SDF vivant et consistant réalisé avec *Turbine*. Ces aspects permettent alors d'envisager une intégration au *GitHub* du *LIP6*.

Cette insertion dans le monde de la recherche nous a introduits à de nombreuses notions nouvelles, comme la théorie des SDF et Choice-Free, la programmation linéaire, et la construction de graphes. À cela, se rajoute la méthode de recherche pas à pas d'une solution pour des problèmes qui au départ n'avaient pas été envisagés.

Annexes

A Exemple de calcul d'un T-semiflot



La matrice de représentation C de ce réseau est la suivante : $\begin{pmatrix} 3 & -2 & 0 \\ 0 & 2 & -5 \\ -3 & 0 & 5 \end{pmatrix}$

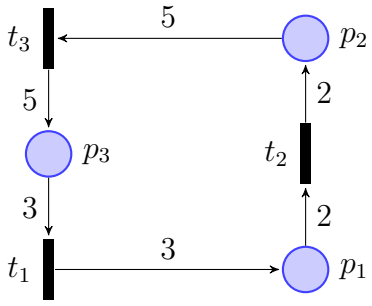
Nous devons trouver un vecteur Y=$\begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \end{pmatrix}$ tel que C*Y = 0_3 où * représente la multiplication matricielle et $0_3 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. Effectuons d'abord la multiplication matricielle :

$$\begin{pmatrix} 3 & -2 & 0 \\ 0 & 2 & -5 \\ -3 & 0 & 5 \end{pmatrix} \quad \begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} \quad = \begin{pmatrix} 3Y_1 - 2Y_2 \\ 2Y_2 - 5Y_3 \\ -3Y_1 + 5Y_3 \end{pmatrix}$$

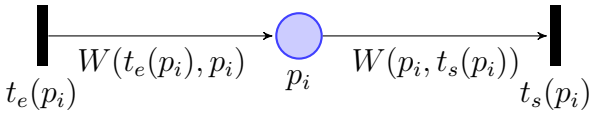
On veut donc que Y vérifie $\begin{cases} 3Y_1 - 2Y_2 = 0 \\ 2Y_2 - 5Y_3 = 0 \\ -3Y_1 + 5Y_3 = 0 \end{cases} \iff \begin{cases} 3Y_1 = 2Y_2 \\ 2Y_2 = 5Y_3 \\ 3Y_1 = 5Y_3 \end{cases}$

On veut donc Y sous la forme Y = $\begin{pmatrix} Y_1 \\ \frac{3}{2}Y_1 \\ \frac{3}{5}Y_1 \end{pmatrix}$. On peut par exemple prendre Y = $\begin{pmatrix} 10 \\ 15 \\ 6 \end{pmatrix}$.

B Exemple de la condition de vivacité



Chaque place p_i, $i \in [\![1, 3]\!]$, a un marquage initial noté $M_0(p_i)$. Pour toutes places p_i, on pose $t_e(p_i)$, respectivement $t_s(p_i)$, la transition d'entrée de p_i, respectivement de sortie. On note $W(t_e(p_i), p_i)$ et $W(p_i, t_S(p_i))$ comme suit :



Pour que ce circuit c soit vivant il suffit que

$$M_0(p_1) + M_0(p_2) + M_0(p_3) > \sum_{p \in c} W(p, t_s(p)) - \sum_{p \in c} pgcd(W(t_e(p), p), W(p, t_s(p)))$$

Or,

$$\sum_{p \in c} W(p, t_s(p)) = 2 + 5 + 3 = 10$$

et

$$\sum_{p \in c} pgcd(W(t_e(p), p), W(p, t_s(p))) = pgcd(3, 2) + pgcd(2, 5) + pgcd(5, 3) = 1 + 1 + 1 = 3$$

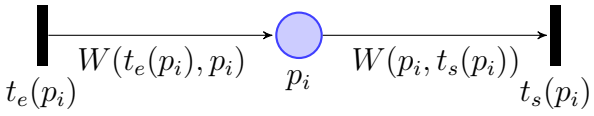
Pour que ce circuit soit vivant il suffit donc que

$$M_0(p_1) + M_0(p_2) + M_0(p_3) > 7$$

C Exemple de résolution du marquage initial

L'idée est d'utiliser la condition suffisante de vivacité pour construire des Choice-Free pondérés vivants de n places.

On cherche maintenant à minimiser le nombre de jetons lors du marquage initial.
Pour chaque place p_i, $i \in [\![1, n]\!]$, on note $x_i = M_0(p_i) \geq 0$ son marquage initial, et $t_e(p_i)$, respectivement $t_s(p_i)$, la transition d'entrée de p_i, respectivement de sortie. On définit enfin $W(t_e(p_i), p_i)$, $W(p_i, t_s(p_i))$ comme suit :



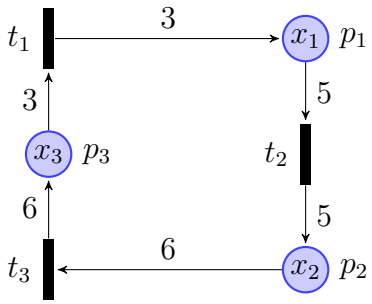
Pour tout i dans $[\![1, n]\!]$:
On veut que $x_i = k_i$ pgcd$(W(t_e(p_i), p_i), W(p_i, t_s(p_i)))$ avec $k_i \in \mathbb{N}$ (marquage utile).
On veut que $\sum_{i=1}^{n} x_i > \sum_{i=1}^{n} W(p_i, t_s(p_i)) - \sum_{i=1}^{n} pgcd(W(t_e(p_i), p_i), W(p_i, t_s(p_i)))$ (condition suffisante de vivacité).

À chaque transition indicée par l, notée t_l, on associe une variable $\gamma_l \in \mathbb{R}^+$ avec l $\in [\![1, m]\!]$ où m est le nombre de transitions.

À chaque place p_i, i $\in [\![1, n]\!]$, on associe l'équation suivante : $\gamma_k - \gamma_j + x_i > W(p_i, t_s(p_i)) - pgcd(W(t_e(p_i), p_i), W(p_i, t_s(p_i)))$.

Alors, en sommant les n équations, les variables γ_l s'annulent les unes avec les autres vérifiant alors :

$$\sum_{i=1}^{n} x_i > \sum_{i=1}^{n} W(p_i, t_s(p_i)) - \sum_{i=1}^{n} pgcd(W(t_e(p_i), p_i), W(p_i, t_s(p_i)))$$



Marquage :

$$\begin{cases} x_1 = k_1 (= k_1 pgcd(3,5)) \in \mathbb{N} \\ x_2 = k_2 (= k_2 pgcd(5,6)) \in \mathbb{N} \\ x_3 = 3k_3 (= k_3 pgcd(6,3)) \in \mathbb{N} \end{cases}$$

On veut que $x_1 + x_2 + x_3 > 3 + 5 + 6 - 1 - 1 - 3 = 14 - 5 = 9$ et que

$$\begin{cases} \gamma_2 - \gamma_1 + x_1 > 5 - 1 = 4 \\ \gamma_3 - \gamma_2 + x_2 > 6 - 1 = 5 \\ \gamma_1 - \gamma_3 + x_3 > 3 - 3 = 0 \end{cases}$$

En faisant la somme des trois équations on retrouve bien l'équation $x_1 + x_2 + x_3 > 9$.

On aura ensuite besoin de remplacer l'inégalité stricte par une égalité large (car nous allons utiliser la programmation linéaire pour résoudre ce système et cette dernière requiert des inégalités larges). Pour ce faire, on va ajouter un biais ε comme suit :

$$\gamma_k - \gamma_j + x_i - \varepsilon \geq W(p_i, t_s(p_i)) - pgcd(W(t_e(p_i), p_i), W(p_i, t_s(p_i)))$$

avec $\varepsilon > 0$ petit.

Quand on fera la somme sur un circuit c, on obtiendra donc :

$$\sum_{i=1}^{n} x_i - |c|\varepsilon \geq \sum_{i=1}^{n} (W(p_i, t_s(p_i)) - pgcd(W(t_e(p_i), p_i), W(p_i, t_s(p_i))))$$

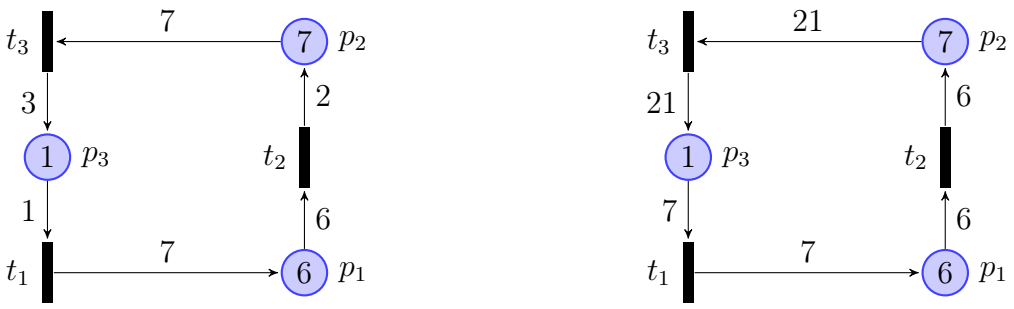
où $|c|$ est le nombre de places du circuit. En effet, il y a une équation par place et on somme ces dernières. Il y a donc $|c| = n$ équations.

Il faut que $|c|\varepsilon$ soit inférieur à 1 mais pas trop petit à cause des erreurs d'arrondi. Il faut que cette valeur soit prise en considération.

Soit $\overline{x_i}$ la solution donnée par le solveur (solution optimale du système relâché). Alors x_i* est une solution dans $\mathbb{N}$ (qui n'est pas forcement optimale mais qu'on espère proche de l'optimale) où $x_i* = \lceil \overline{x_i} \frac{1}{pgcd(W(t_e(p_i),p_i),W(p_i,t_s(p_i)))} \rceil \; pgcd(W(t_e(p_i), p_i), W(p_i, t_s(p_i)))$.

D Exemple de normalisation

Prenons le réseau de Petri suivant de T-semiflot $Y = [6, 7, 2]$



(a) Avant normalisation

(b) Après normalisation

Normalisation d'un SDF

En effet, on obtient $R = ppcm(6, 7, 2) = 42$, puis en appliquant la formule précédente : $Z(t_1) = \frac{42}{6} = 7$, $Z(t_2) = \frac{42}{7} = 6$, $Z(t_3) = \frac{42}{2} = 21$.

E Démonstration du lemme 1 : T-semiflot SDF $\Longrightarrow$ T-semiflot Choice-Free

Démonstration. Soit $Y^{'} = Y^{'}(t_1)...Y^{'}(t_k)$ un T-semiflot du SDF
Alors :

$$\forall i \in [\![1, k]\!], W^{'}(t_i, p_i) * Y^{'}(t_i) = W^{'}(p_i, t) * Y^{'}(t)$$

Puis :

$$\forall i \in [\![1, k]\!], W^{'}(t_i, p_i) * Y^{'}(t_i) = W^{'}(p_a, t) * Y^{'}(t)$$

Or comme (par hypothèse d'après l'équation 1) $\begin{cases} \forall i \in [\![1, k]\!], W^{'}(t_i, p_i) = W(t_i, p) \\ W^{'}(t, p) = \frac{1}{k} * W(t, p) \end{cases}$ alors

$$\forall i \in [\![1, k]\!], W(t_i, p) * Y^{'}(t_i) = \frac{1}{k} * W(p, t) * Y^{'}(t)$$

En sommant les k équations, on obtient :

$$\sum_{i=1}^{k} W(t_i, p) * Y^{'}(t_i) = \sum_{i=1}^{k} \frac{1}{k} * W(p, t) * Y^{'}(t)$$

$$\iff \sum_{i=1}^{k} W(t_i, p) * Y^{'}(t_i) = k * \left(\frac{1}{k} * W(p, t) * Y^{'}(t)\right)$$

$$\iff \sum_{i=1}^{k} W(t_i, p) * Y^{'}(t_i) = W(p, t) * Y^{'}(t)$$

Donc $Y^{'}$ est un T-semiflot pour le Choice-Free. □

F Démonstration du lemme 2 : séquence SDF $\Longrightarrow$ séquence Choice-Free

Démonstration. Nous allons démontrer que, pour une séquence σ admissible pour le SDF, lorsque la transition t est exécutée pour le SDF dans cette séquence, elle l'est au même moment dans la séquence pour le Choice-Free. Ainsi, la transformation d'une place sera transparente

pour les places non transformées situées en amont et en aval de cette dite place. Cela assure alors l'admissibilité de la séquence σ pour le reste du Choice-Free (avant et après l'exécution de t dans σ pour le SDF les mêmes places seront exécutées dans le même ordre pour le Choice-Free).
Pour ce faire nous procédons par récurrence sur la taille de la séquence σ.

Supposons dans la suite qu'une seule place p ait subit la transformation dans le SDF pour obtenir un Choice-Free.

Initialisation : La séquence σ est vide. Alors son exécution est trivialement admissible pour les deux réseaux.

Hérédité : Soit $\sigma = \sigma_1 t$ une séquence de taille $n \geq 1$ admissible pour le SDF et telle que l'hypothèse de récurrence soit vérifié par σ_1.

Alors, on dispose d'une séquence σ admissible pour le SDF dans laquelle la transition t est exécutée après une sous-séquence σ_1. Il nous faut désormais montrer que t est exécutée pour le Choice-Free (σ_1 étant admissible pour le Choice-Free par hypothèse de récurrence) pour prouver que σ est admissible pour le Choice-Free.

On note M_0 le marquage initial, M_1 le marquage après l'exécution de la séquence σ_1 et M le marquage après l'exécution de t pour le Choice-Free. On conserve les mêmes notations indicées par $'$ pour le SDF comme l'illustre la figure ci-dessous :

Choice-Free : $M_0 \xrightarrow{\sigma_1} M_1$

SDF : $M_0^{'} \xrightarrow{\sigma_1} M_1^{'} \xrightarrow{t} M^{'}$

— Puisqu'aucune des places en amont de la place transformée p n'a subit de transformation pour obtenir le Choice-Free, elles ont été exécutées le même nombre de fois dans σ_1 pour le Choice-Free (puisque σ_1 est admissible pour le Choice-Free par hypothèse de récurrence). D'où pour toutes ces places p_n : $M_(p_n) = M^{'}(p_n)$.
Alors la transition t est tirable pour l'étape du marquage M.

Plus simplement dit, les transitions $(t_i)_{i\in[\![1,k]\!]}$ de la figure 5 vont être exécutées au même moment dans le Choice-Free que dans le SDF pour la séquence σ_1 car les places en amont n'ont pas été modifiées ; il n'y a aucun blocage en amont de la place p.

— Montrons que les valeurs des poids et du marquage initial de p permettent l'exécution de la transition t dans le Choice-Free.
Puisque σ est admissible pour le SDF :

$$\forall i \in [\![1,k]\!], M_0^{'}(p_i) + W^{'}(t_i,p_i) * \delta^{'}(\sigma,t_i) - W^{'}(p_i,t) * \delta^{'}(\sigma,t) \geq 0$$

Puis d'après le point précédent :

$$\forall i \in [\![1,k]\!], M_0^{'}(p_i) + W^{'}(t_i,p_i) * \delta(\sigma,t_i) - W^{'}(p_i,t) * \delta(\sigma,t) \geq 0$$

En sommant et par construction (formule 1 et 2), on obtient :

$$M_0(p) + \sum_{i=1}^{k} W(t_i,p) * \delta(\sigma,t_i) - W(p,t) * \delta(\sigma,t) \geq 0$$

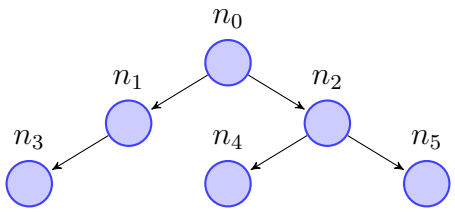
Donc t est exécutable dans le Choice-Free.

Conclusion : Une séquence σ admissible pour le SDF l'est aussi pour le Choice-Free. La transition t étant exécutée au même moment dans les deux réseaux et délivrant le même nombre de jetons (puisque la sortie de t n'est pas impactée par la transformation), alors cette transformation est transparente pour le Choice-Free. Autrement dit, plusieurs places dans le SDF peuvent être transformées pour obtenir un Choice-Free tout en conservant la propriété voulue sur les séquences admissibles.

□

G Autre solution pour la génération d'un graphe connexe

Une première solution, simple à mettre en place, est la génération d'un graphe orienté acyclique, qui peut être assimilé à un arbre orienté dans notre cas : un graphe orienté dont tous les nœuds sont reliés à une racine et ne possédant pas de circuit ; comme illustré sur la figure ci-dessous. Dès lors, pour chaque nœud du graphe, il suffit de tirer un nombre prédéfinit d'enfants à relier à ce dernier.



Toutefois, les résultats obtenus ne satisfont pas un nombre suffisant d'entrées par nœud, condition nécessaire pour transformer le SDF en Choice-Free, avec une moyenne du nombre d'entrées ne dépassant pas 2 comme le montre la figure suivante :

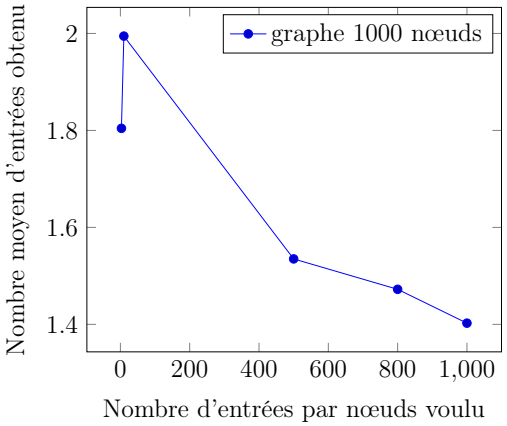


FIGURE 15 – Moyenne créée sur une génération aléatoire de 100 graphes

H Différentes solutions abordées pour la génération des poids

Dans la suite, on souhaite générer aléatoirement de multiples entiers naturels strictement positifs dont la somme est fixée.

Première idée

Une idée simple est d'initialiser un tableau de n cases à 1, puis d'incrémenter de 1 des cases choisies aléatoirement jusqu'à atteindre la somme voulue.

Cependant, la distribution obtenue avec cette méthode n'est pas uniforme (condition nécessaire pour produire des SDF de manière aléatoire) comme le montre la figure ci-dessous :

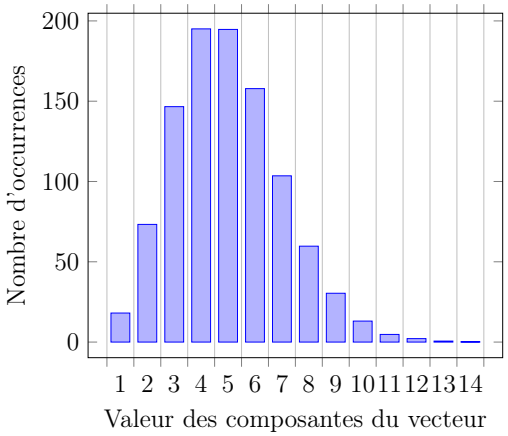


FIGURE 16 – Moyenne faite sur 100 essais - (nombre nœuds, somme fixée, moyenne somme atteinte)=(1000, 5000, 5001.0)

Deuxième idée

Une autre méthode est de remplir un tableau de n cases avec des nombres aléatoires tirés entre 1 et la somme fixée, notée S. En posant f le rapport de la somme fixée sur la somme des valeurs du tableau, il suffit de prendre, pour chaque case, la partie entière supérieure du produit de la valeur de cette case par f.
En effet, en notant x_i la valeur de la case d'indice i du tableau on obtient :

$$\sum_{i=1}^{n} \lceil x_i * f \rceil \approx \sum_{i=1}^{n} x_i * f \approx \frac{S}{\sum_{i=1}^{n} x_i} * \sum_{i=1}^{n} x_i \approx S$$

Si la répartition est quasiment uniforme avec cette méthode, la somme obtenue diffère trop de la somme fixée avec des écarts relatifs moyens de près de 10%, comme l'illustre la figure suivante :

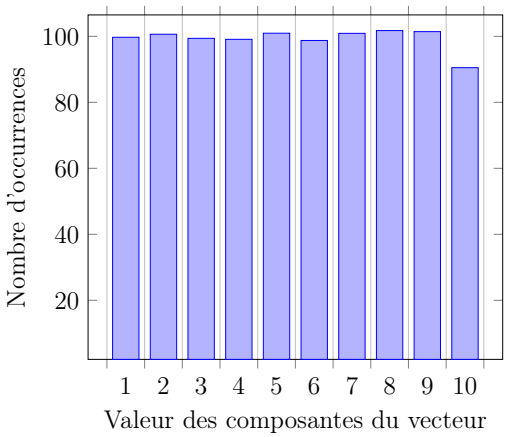


FIGURE 17 – Moyenne faite sur 100 essais - (nombre nœuds, somme fixée, somme atteinte)=(1000, 5000, 5506.43)

I Fonctionnement de l'algorithme de génération de graphes simplement connexes

Soit n le nombre de noeud du graphe, m son nombre d'arcs et d sa densité.

Afin d'être certain que le nombre d'arcs soit atteint ($m \approx \lceil d * n(n-1) \rceil$), on fixe une valeur moyenne avg_degree que chaque noeud devrait avoir. On pose alors $avg\_degree = \lceil \frac{m}{n} \rceil$.
On dispose de deux tableaux deg_i et deg_o pour sauvegarder le degré entrant maximal et celui sortant maximal de chaque noeud. Au départ, ces tableaux sont initialisés avec une même valeur égale à $n-1$; valeur maximale possible comme degré dans un graphe.

Pour sélectionner un noeud d'entrée on tire aléatoirement une valeur, notée $alea$, entre 0 et $\left(\sum_{k=1}^{n} deg_o[k]\right) - 1$. On considère ici le tableau des degrés sortants car tirer un noeud comme entrée pour un autre, revient à choisir un noeud qui peut avoir comme sortie le noeud actuel, autrement dit, un noeud qui peut accepter d'avoir une sortie de plus. Puis, pour choisir un noeud, il suffit de parcourir le tableau deg_o depuis le début et de s'arrêter sur la case dont la somme des valeurs avant celle-ci dépasse strictement la valeur tirée. En d'autres termes, le noeud k tiré vérifie $\sum_{l=1}^{k} deg_o[l] > alea$.
Dans ces conditions, un noeud k dont sa valeur $deg_o[k]$ vaudrait 0 ne pourra plus être tiré. Ainsi, afin de s'assurer qu'on ne relie pas un noeud à lui-même ou bien à un noeud précédemment relié au noeud actuel comme entrée, on fixe temporairement leur valeur à 0 dans le tableau des degrés sortants.
Le processus est le même lorsqu'il s'agit de tirer une sortie pour un noeud, à la différence que le tableau considéré est deg_i.

Pour chaque noeud k, on tire aléatoirement un nombre (autour de avg_degree) d'entrées et de sorties (tel que $1 \leq avg\_degree \leq n-1$). Pour chaque entrée p tirée, $deg_i[k]$ et $deg_o[p]$ sont décrémentés de 1. On effectue alors cette opération pour chaque noeud du graphe.
À chaque étape $l \in [\![0, n-1]\!]$, on vérifie si le nombre actuel d'arc du graphe est plus grand que $m-(n-l)$. Si tel est le cas, cela signifie que pour les $(n-l)$ noeuds restant leurs degrés entrants et sortants doit valoir 1.

Dans les faits, même après avoir tiré un nombre de sortie pour un noeud, il pourra être choisi par le suite (sauf si ce nombre est $n-1$) et ce parce qu'il n'est pas possible de tirer aléatoirement au fur et à mesure des valeurs de degré cohérentes pour le graphe. En effet, il faut que la somme des degrés entrants soit égale à la somme des degrés sortants et que cette somme vaille à peu près m. Autrement dit, cela nécessite en amont de générer aléatoirement n valeurs comprises entre 1 et $n-1$ et dont la somme vaut environ m ; ce qui à l'heure actuelle est un problème ouvert et sans solution réellement efficace.
Alors, pour chaque noeud on fixe le degré entrant et sortant aléatoirement tout en laissant la possibilité qu'il ait d'autres entrées et sorties par la suite.

Finalement, un parcours en profondeur du graphe, dont la complexité est linéaire et en $\Theta(n+m)$, permet de le rendre connexe.

J Organisation du code

README.md : introduction au problème et tutoriel simple pour utiliser le code.
/**Algorithme_C/Graph :** Code source permettant la génération de Choice-Free vivant.
main.c : interface permettant à l'utilisateur de générer des Choice-Free.

GraphGenerator.c GraphGenerator.h : contient les fonctions de création d'un Choice-Free dont les étapes ont été définies précédemment.

Array.c Array.h : contient les définitions des structures utilisées pour manipuler les graphes, ainsi que leurs fonctions associées.

Rand.c Rand.h : contient les fonctions propres à la génération de nombres aléatoires.

Display.c Display.h : contient les fonctions d'affichage et de sauvegarde d'un Choice-Free vers un fichier au format PNML.

Tools.c Tools.h : contient diverses fonctions non relatives à la génération de graphes simplifiant le code.

/Algorithme_C/Py_interface : Serveur socket permettant la communication entre le code source C et l'interface graphique Python pour le débogage. La visualisation de graphes en C étant inutilement compliquée, il a été plus simple d'utiliser Python pour afficher les graphes générés par le code source C.

main_py.c : programme lancé par l'interface graphique de Python pour communiquer avec le code source C.

/Algorithme_C/Examples : Exemples d'utilisation des structures de graphes et de la génération de Choice-Free.

/Algorithme_C/Library :

choiceFreeGeneratorLib.a choiceFreeGenerator.h : API utilisateur pour intégrer le code source à d'autres programmes.

choiceFreeGeneratorDevLib.a choiceFreeGeneratorDev.h : API programmeur pour tester le code depuis d'autres programmes.

/Python : Interface graphique en Python.

main.py : script principal permettant de lancer les commandes depuis Python pour dialoguer avec le code source C.

/Tests : Code source des tests réalisés sur les performances.

K Structures utilisées lors de l'implémentation

Liste simplement chaînée

La première structure utilisée est une liste simplement chaînée, c'est-à-dire qu'à partir d'un nœud de la liste il est possible d'atteindre le nœud suivant. Cette structure pointe sur une donnée au travers du champ *data*. Le type de données est défini préalablement comme étant un type de base, ou bien un type personnalisé. Dans le second cas, il est nécessaire de fournir la fonction de libération de la mémoire de la donnée *data*.

```
typedef enum Types {custom_t=-1, uInt_t, petri_t, petriElem_t, petriNode_t,
    petriLink_t, wrapper_t, array_t, list_t, fixedSizeList_t, directedGraph_t}
    types;
```

```
typedef struct Array * pArray;
typedef struct Array
{
  types type;
  void (*freeFunction)(void * pData);
    void * data;
    struct Array * next;
} array;
```

Graphe orienté

Les graphes orientés sont stockés sous forme de tableaux dont la taille correspond au nombre de nœuds. Chaque case d'indice i du tableau correspond à l'ensemble des nœuds qui ont comme entrée le nœud i ; cet ensemble étant lui-même stocké sous la forme d'une liste simplement

chaînée.
La structure contient également la taille du tableau (autrement dit le nombre de nœuds), ainsi que le nombre d'arcs dans le graphe ; utile pour la suite.

```
typedef struct _directedGraph * pDirectedGraph;
typedef struct _directedGraph
{
  pArray * links_list;
  unsigned int nb_nodes;
  unsigned int nb_edges;
} directedGraph;
```

Par exemple, si le nœud 0 a comme enfant les nœuds $1, 4, 3$, alors dans la case d'indice 0 du tableau la liste chaînée y étant contenue sera composée des valeurs $1, 4, 3$.

Réseau de Petri

La structure utilisée pour stocker les réseaux de Petri est volontairement plus lourde en mémoire pour faciliter la manipulation de ces derniers. Cette structure permet, à partir de n'importe quel nœud ou transition, de trouver toutes ses entrées et sorties sans avoir à parcourir un tableau pour savoir quel élément est connecté à quel autre élément. Cette facilité de recherche est possible par une plus grande redondance des données en mémoire.

Chaque élément (place et transition) est stocké sous la même structure *petriElem*. Ils sont alors différenciés par leur type :
PETRI_PLACE_TYPE pour une place et *PETRI_TRANSITION_TYPE* pour une transition. Les éléments possèdent également un nom *label* sous la forme d'un entier positif. Puis, pour les places uniquement, le champ *val* est utilisé pour conserver le marquage initial.

```
#define PETRI_PLACE_TYPE 1
#define PETRI_TRANSITION_TYPE 0
```

```
typedef struct PetriElem * pPetriElem;
typedef struct PetriElem
{
  int type;
  unsigned int label;
  int val;
} petriElem;
```

Les liens entre les places et les transitions sont stockés sous la forme de la structure *petriLink*. Cette dernière prend un élément correspondant au point de départ *input* de l'arc, et un autre *output* pour le point d'arrivée. À ceci s'ajoute le champ *weight* permettant de stocker le poids de l'arc.
Les éléments du lien ne sont que des références vers les éléments place/transition et ce afin d'affecter tous les liens en même temps si une modification de paramètres d'un élément est nécessaire.

```
typedef struct PetriLink * pPetriLink;
typedef struct PetriLink
{
  pPetriElem input;
  pPetriElem output;
  unsigned int weight;
} petriLink;
```

La structure *petriNode* est celle qui permet de centraliser toutes les entrées et les sorties pour un élément place/transition. Elle contient le nombre d'entrées de cet élément *nb_inputs*, son nombre de sorties *nb_outputs*, une liste simplement chaînée *input_links* des liens dans

lesquels l'élément intervient comme une sortie, et une autre *output_links* dans laquelle l'élément intervient comme une entrée.

```
typedef struct PetriNode * pPetriNode;
typedef struct PetriNode
{
  unsigned int nb_inputs;
  pArray input_links;
  unsigned int nb_outputs;
  pArray output_links;
} petriNode;
```

Finalement, toutes ces structures sont centralisées dans une seule *petri* : celle utilisée pour stocker les réseaux de Petri. Toutes les références des places sont conservées dans le tableau *pl_elems* de taille *nb_pl*. De même pour toutes les références vers les transitions stockées dans *tr_elems* de taille *nb_tr*.
L'ensemble des liens est conservé sous la forme d'une liste doublement chaînée *links*, une liste pour laquelle il est possible d'atteindre les nœuds précédent et suivant un nœud
L'ensemble des liens d'entrées et de sorties pour chaque nœud est conservé dans les tableaux *places* pour les places et *transitions* pour les transitions. La case i du tableau des places correspond à l'ensemble des liens d'entrées et de sorties de la place i. De même pour le tableau des transitions.

```
typedef struct Petri * pPetri;
typedef struct Petri
{
  unsigned int nb_pl;
  pPetriNode * places;
  unsigned int nb_tr;
  pPetriNode * transitions;
  unsigned int nb_links;
  pArray2 links;
  pPetriElem * pl_elems;
  pPetriElem * tr_elems;
} petri;
```

Il est nécessaire de comprendre que la position d'un élément dans un des deux tableaux doit impérativement correspondre à son nom, dans le champ *label* de la structure des éléments. Par exemple, à la position i du tableau des places *places* dans la structure *petri*, le champ *label* de la structure *petriElem* stocké comme référence dans la case i du tableau doit avoir comme valeur i.

Dans cette structure, l'accès à l'ensemble des relations pour une place/transition est en $\Theta(1)$ puisqu'étant stocké dans un tableau. Une fois la case atteinte, il suffit de parcourir la liste simplement chaînée des liens d'entrées ou de sorties pour trouver les éléments qui y sont reliés. Or, comme la densité des réseaux n'est pas grande (pour chaque nœud, le nombre d'entrées/sorties par rapport au nombre total de nœuds est très faible), les listes chaînées sont composées de peu d'éléments.
De surcroît, la modification d'un paramètre est également en $\Theta(1)$ puisque chaque élément est stocké dans un tableau et la modification est répercutée instantanément dans le reste du réseau. En effet, seule la référence vers les éléments est stockée à chaque étape.

La suppression d'un élément est également assez simple. Une fois le nœud dans le tableau de places/transitions atteint, il suffit de lire chaque lien d'entrées et de sorties pour trouver avec quels éléments le nœud est relié. Il suffit ensuite de supprimer le lien correspondant dans ces éléments (une fois un élément trouvé, l'accès à ce dernier est immédiat et il suffit de parcourir la liste chaînée de cet élément pour trouver la référence du lien correspondant et la supprimer

de la liste), puis le lien est ensuite définitivement supprimé de la liste doublement chaînée.
Remarquons qu'il n'est pas nécessaire de parcourir la liste doublement chaînée pour trouver le lien puisque dans la structure *petriNode* ce ne sont pas les références des liens qui sont conservées mais la référence du nœud dans la liste doublement chaînée contenant le lien. L'utilisation d'une liste doublement chaînée permet ici de supprimer un élément de la liste depuis cet élément (puisqu'on dispose alors de l'élément précédent et suivant de la liste), ce qui n'est pas possible avec une lite simplement chaînée (il aurait fallu stocker l'élément précédent celui référençant le lien).
Ainsi, la suppression d'un élément est relativement rapide.

L Performances du générateur

Temps moyen (en s) - nombre moyen entrées 3/nombre moyen sorties 3				
Nombre de transitions	10	100	1000	10000
Création graphe connexe	0.000015	0.000392	0.032444	4.252117
Transformation fortement connexe	0.000003	0.000026	0.000219	0.002952
Conversion graphe/SDF	0.000017	0.000146	0.001474	0.022075
Génération poids	0.000001	0.000015	0.000052	0.000357
Normalisation	0.000002	0.000009	0.000893	0.007468
Calcul marquage	0.000187	0.005312	1.559445	369.987137
SDF vers Choice-Free	0.000018	0.000134	0.001389	0.029095
Libération mémoire	0.000006	0.000052	0.001148	0.040077
Total	0.000249	0.006086	1.597064	374.341278

Bibliographie

[1] Y. L. B. Bodin, "Fast and efficient dataflow graph generation," *Proceedings of the 17th international workshop on software and compilers for embedded systems*, 2014.

[2] E. Lee and D. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE, 75(9) :1235-1245*, 1987.

[3] M. O'Brien, "Petri nets : Properties, applications, and variations," University of Pittsburgh - lecture. [Online]. Available : http ://-people.cs.pitt.edu/ chang/231/y16/231sem/semObrien.pdf

[4] J. Desel and J. Esparza, *Free Choice Petri Nets.* Cambridge tracts in theoretical in computer science 40 : Press Syndicate of the university of Cambridge, 1995.

[5] O. Marchetti and A. Munier-Kordon, "A sufficient condition for the liveness of weighted event graphs," *European Journal of Operational Research*, 2009.

[6] P. Thomas, "Contribution to the study of weighted petri nets," Ph.D. dissertation, Sorbonne université, F-75005, 2014.

[7] P. M. Maueur, "Generating strongly connected random graphs," *Int'l Conf. Modeling, Sim. and Vis. Methods - MSV'17*, 2017.

[8] "Petri nets standard pnml." [Online]. Available : http ://www.pnml.org/

[9] "Pnml document checker." [Online]. Available : http ://pnml.lip6.fr/pnmlvalidation/usage.html