Implémentation de réseau de Petri

T.Bersoux et N.Compère, Gr 681

Table des matières

1 Introduction aux réseaux de Petri

Les réseaux de Petri se modélisent en général à l'aide d'un n-uplet, souvent un quadruplet $(T, P, A, C(p))$

— T est un ensemble de transitions, représentés graphiquement par une barre.

— P est un ensemble de "places", représentées graphiquement par un rond.

— A est un ensemble d'arcs orientés, représentés par des flèches. Un arc se fait toujours entre une place et une transition, jamais entre une place et une place, ni entre une transition et une transition.

— $C(p) \in \mathbb{N}$ est la capacité de la place p en nombre de jetons. Une capacité non indiqué graphiquement signifie aucune limite de capacité.

On rajoute parfois aussi :

— $Pre(t)$ qui indique combien de jetons doivent être consommés pour atteindre la transition t. Pas d'indication sur graphique = 1 (par convention).

— $Post(t)$ qui indique sur l'arc combien de jetons sont produits par la transition t dans la place qui suit t. Pas d'indication sur graphique = 1 (par convention).

— M_0 le marquage au temps 0 qui indique le nombre de jetons au départ sur chaque place.

Les ensembles Pre et Post peuvent se marquer sous forme matricielle. Les colonnes sont alors les transitions, les lignes sont les places, et les valeurs sont les jetons requis/produits (respectivement pour Pre/Post). Par exemple :

$$\text{PRE} = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \text{ et POST} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

2 Implémentation

2.1 Différentes approches possibles

Notons d'abord qu'en utilisant le module matrix, il était très simple d'implémenter ces réseaux, en utilisant juste un tableau pour le marquage, ainsi que deux matrices POST et PRE pour les arcs. N'ayant pas vu en cours ce module, et ayant d'autres idées, nous n'avons pas choisi cette approche.
Nous avions tout d'abord pensé de manière trop impérative (En même temps, les réseaux de Petri s'y prêtent bien !) comme on peut le voir dans l'annexe A, qui est la première version de notre code. Cette version aurait presque pu fonctionner si on y ajoutait l'utilisation de références. Mais pour coller de plus près aux cours et TP, nous avons décidé de repartir de zéro ou presque, et de réfléchir "fonctionnel".

2.2 Notre approche

Nous représentons les réseaux de Petri par un ensemble d'arcs entrants, un ensemble d'arcs sortants, et un marquage (Nombre de jetons dans des places). Cela permet, lors de l'activation d'une transition de modifier seulement le marquage et non le reste du réseau (puisque les arcs restent les même). Cela résolvait ainsi le problème de notre première approche, où c'était le rôle des arcs que de stocker les jetons (La mise à jour était alors difficile, puisque il y avait conflits entre les arcs qui n'avaient pas les mêmes nombres de jetons pour une même place !).

2.3 Définitions des types et getters/setters (Lignes 4 à 70)

L'approche "Constructeur" de place et transition, et arcs permet de compacter la définition de réseau tout en la gardant claire.
Un marquage est tout simplement une liste de place associées a leur nombre de jetons.
Un arc est soit entrant, soit sortant. Un entrant va d'une place à une transition, et un sortant fait l'inverse. Les arcs ont tous des poids.
Un réseau est alors une liste d'arcs entrants, une liste d'arcs sortants et un marquage.
Pour les types marquage, arc, et réseau qui sont des enregistrements, nous avons crée des setters et getters afin d'augmenter la clarté du code. Sans ces getters et setters, nous faisions des dizaines de matchs identiques dans le reste du code...

2.4 Utilitaires (Lignes 72 à 107)

Les deux fonctions arcs_of_transition, de fonctionnements similaires, renvoient l'ensemble des arcs (resp. entrants, sortants) d'une transition. Cela permet de construire des fonctions récursives sur ces ensembles d'arcs.
La fonction poids_ok est l'une de ces fonction récursives, car elle regarde un ensemble d'arcs entrants pour savoir si la consommation de ces arcs ne va pas créer un nombre de jetons négatifs sur une place.
est_franchissable a pour but de dire si une transition est... franchissable. Pour cela, on doit regarder si la consommation des jetons par la transition ne nécessite pas plus de jetons que le contenu des places qui la précèdent. On devrait aussi regarder, si on avait implémenté la capacité maximum des places, si la transition ne produisait pas plus de jetons que les places qui la suivent ne pouvaient supporter. Par manque de temps, nous n'avons pu implémenter cette capacité maximum (est_franchissable se contente donc juste d'appeler poids_ok).

2.5 Fonctionnement (Lignes 109 à 146)

Pour un réseau de Pétri, on doit pouvoir activer une transition.
Pour cela, nous avons divisé le problème en trois parties : Obtenir le réseau, c'est obtenir le marquage modifié par l'activation de la transition, et l'activation de la transition c'est activer la consommation/production de tous ses arcs.
La fonction activer_arc modifie ainsi le marquage en ajoutant ou retirant des jetons à la place qui lui est associé.
activer_transition regarde alors si la transition est franchissable, et active tous ses arcs (en commençant par les entrants, et en faisant bien attention à faire passer le nouveau marquage à chaque activation d'arc, afin de bien garder le marquage à jour).
Ensuite, step renvoie le réseau avec le nouveau marquage modifié par l'activation de la transition.

2.6 Utilisation

On peut définir un réseau de façon plus ou moins condensé. La version condensé est plutôt lisible et c'est donc celle que l'on donne ici (voir l'exemple pour une plus exhaustive) :

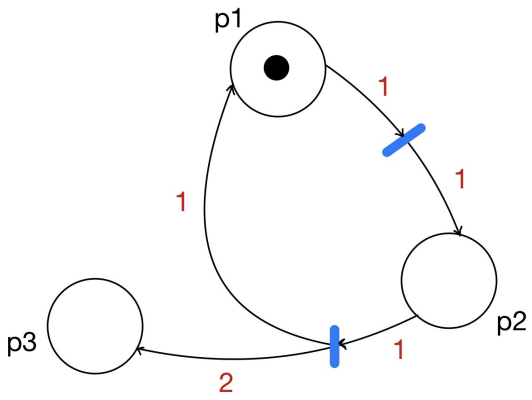
```
let _nomDuReseau_ = (
[
ArcEntrant(Place _numPlace_,Transition _numTransition_,_numPoids_);
...
(*definir autant d'arcs que l'on veut*)
],[
ArcSortant(Transition _numTransition_,Place _numPlace_,_numPoids_);
...
(*definir autant d'arcs que l'on veut*)
],[
(Place _numPlace_,_nbJetonsInitial_);
...
(*definir autant de places que l'on veut*)
]
)
```

Une fois le réseau définit, il suffit d'appeler "step _numDeLaTransitionAActiver_ _nomDuReseau_" et la fonction donnera alors le réseau résultant.

Attention, lors de la définition, on suppose que l'utilisateur entre un réseau de Petri "bien formé" : Pas de nombre négatif pour les jetons, pas de transition vers ou depuis une place qui n'existe pas... (Avec un peu de temps on pourrait implémenter des fonctions de vérifications pour faire un code sécurisé, mais pas vraiment le sujet du projet...).

3 Exemple

Nous avons défini le réseau suivant :



Ce qui donne en ocaml les lignes 153 à 173 du code (Ou lignes 177 à 192 pour la version condensée).

On construit également une petite fonction "doubler" qui utilise le réseau pour doubler un entier donné en paramètre :

```
let rec doubler i r =
match i with
|0 -> jetons_of_place 3 (marquage_of_reseau r)
|_ -> doubler (i-1) (step 2 (step 1 r))
```

Enfin, on a défini trois variables : resPostInutile,resPostT1,resPostT2, qui sont respectivement les réseaux après transition 2, puis 1, puis 2. La première activation de la transition 2 est sans effet car pas assez de jeton dans la place 2, mais les autres ont un effet (consommation et production) : nos fonctions font bien ce qu'on attend d'elles.
L'utilisateur pourra tester "doubler" avec l'entier de son choix (pas trop grand, cela reste lent...) et le réseau initial "res".

4 Conclusion

Nous partions avec une certaine appréhension, car les réseaux de Pétri nous semblaient quand même peu adaptés pour la programmation fonctionnel. D'ailleurs, notre première version était plutôt tournée itératif avec en tête des pointeurs par exemple. Ce projet nous a donc permis d'apprendre à mieux réfléchir d'un point de vue fonctionnel. Nous sommes assez satisfaits de notre programme. Il est possible de définir de nombreux réseaux, d'activer les transitions, et de définir des fonctions qui les utilisent. Le code est concis et clair. Il y a plusieurs améliorations possibles : Rajouter les capacités maximales des places, ajouter des messages pour l'utilisateur et des vérifications des données entrées... . Une interface graphique pourrait aussi être intéressante, car les réseaux de Petri s'y prêtent bien !

A Première version inaboutie

```
type place = {num : int ; mutable jetons : int;} (* etiquette,
jetons actuellement dans la place, jetons max simultanés*)
type arcE = {p : place; t : int ; poidsE : int} (* place, transition, poids *)
type arcS = {t : int ; p : place ; poidsS : int} (* transition, place, poids *)
type reseau = {arcsEntrants : arcE list ; arcsSortants : arcS list}

let place1 = {num =1; jetons = 0;}
let place2 = {num =2; jetons = 1;}
let place3 = {num =3; jetons = 0;}

let arcE1 = {p = place1; t = 1; poidsE = 1}
let arcE2 = {p = place2; t = 2; poidsE = 1}
let arcS1 = {t = 1; p = place2 ; poidsS = 1}
let arcS2 = {t = 2; p = place3 ; poidsS = 2}
let arcS3 = {t = 2; p = place1 ; poidsS = 1}

let net = {arcsEntrants = [arcE1;arcE2];arcsSortants = [arcS1;arcS2;arcS3]};;

let nTransitions net=
let rec dansEntrants arcsE m=
match arcsE with
|[] -> m
|({p = _; t = i ; poidsE = _})::q -> if(i>m) then dansEntrants q i else dansEntrants q m
in dansEntrants net.arcsEntrants 0
;;

let rec poidsOk arcsE =
match arcsE with
|[] -> true
|({p = pl ; t = _ ; poidsE = po })::q -> (pl.jetons >= po)&&(poidsOk q)
;;

let rec listeArcsE t arcsE =
match arcsE with
|[] -> []
|({p = pl ; t = i ; poidsE = po })::q -> if (i=t) then [{p = pl;t=i;poidsE=po}]@(listeArcsE t q)
else (listeArcsE t q)
;;

let rec listeArcsS t arcsS =
match arcsS with
|[] -> []
|({t = i ; p = pl ; poidsS = po })::q -> if (i=t) then [{t=i;p = pl;poidsS=po}]@(listeArcsS t q)
else (listeArcsS t q)
;;

let estFranchissable t net =
poidsOk (listeArcsE t net.arcsEntrants)
;;

let consommer t net =
```

```
let arcsCourants = listeArcsE t net.arcsEntrants in
let rec aux arcs =
match arcs with
|[] -> []
|({p = {num = n; jetons = j} ; t = tr ; poidsE = po }::q ->
[{p = {num = n;jetons=(j-po)};t=tr;poidsE=po}]@(aux q)
in
if (estFranchissable t net) then (aux arcsCourants) else arcsCourants
;;

let produire t net =
let arcsCourants = listeArcsS t net.arcsSortants in
let rec aux arcs =
match arcs with
|[] -> []
|({t = tr; p = {num = n; jetons = j};poidsS = po})::q ->
[{t = tr; p = {num = n; jetons = j+po};poidsS = po}]@(aux q)
in
if(estFranchissable t net) then (aux arcsCourants) else arcsCourants
;;


(* Fonctionnement similaire a listeArcs mais comme on recree
le reseau on ne peut se contenter d' un appel a liste *)
let stepTransition t net =
let rec filtreEntrants transi arcsE =
match arcsE with
|[] -> []
|({p = pl ; t = i ; poidsE = po })::q -> if (i!=transi) then
[{p = pl;t=i;poidsE=po}]@(filtreEntrants transi q) else (filtreEntrants transi q)
in let nouveauxE = (filtreEntrants t net.arcsEntrants)@(consommer t net) in
let rec filtreSortants transi arcsS =
match arcsS with
|[] -> []
|({t = i;p = pl;poidsS = po})::q -> if (i!=transi) then
[{t = i;p = pl;poidsS = po}]@(filtreSortants transi q) else  (filtreSortants transi q)
in let nouveauxS = (filtreSortants t net.arcsSortants)@(produire t net) in
{arcsEntrants = nouveauxE;arcsSortants = nouveauxS }
;;
```