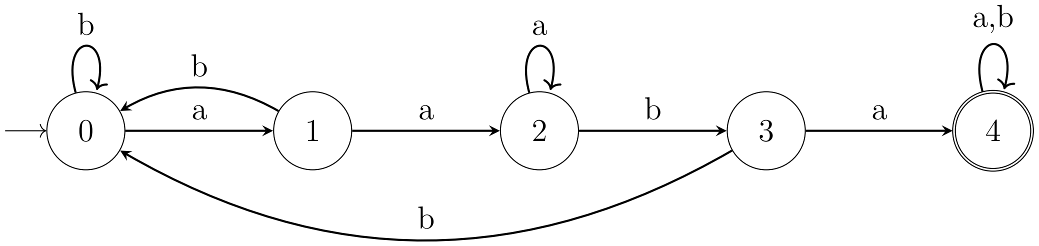
Prédécesseurs et successeurs

Les **successeurs** d'un état q sont l'ensemble des états de destination des transitions partant de q.
Par exemple si l'on reprend l'automate du $TP1$:



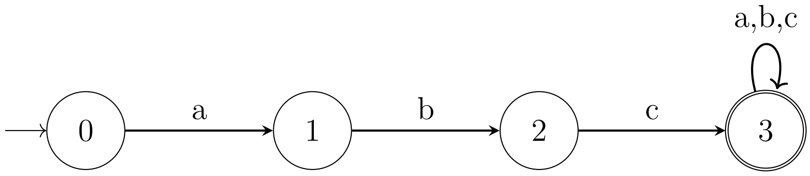
On a `successeurs(0)=[0,1]`, `successeurs(1) = [0, 2]`, `successeurs(2) = [2, 3]`, `successeurs(3) = [0, 4]` et `successeurs(4) = [4].`
De la même manière on définit l'ensemble des prédécesseurs d'un état q correspond aux états qui ont (au moins) une transition allant vers q. Ce qui revient à dire que les prédécesseurs de q sont les états qui ont q comme successeur.

Dans l'exemple ci-dessus cela donne donc :
`predecessseurs(0) = [0, 1, 3]`, `predecesseurs(1)=[0]`, `predecesseurs(2) = [1, 2]`, `predecesseurs(3) = [2]`, `predecesseurs(4) = [3, 4]`.

Exercice 1. Implémenter les méthodes `successeurs(self,etat)` et `predcesseurs(self,etat)` qui prennent en entrée un automate et un état puis retournent la liste des successeurs (respectivement des prédécesseurs) de cet état.

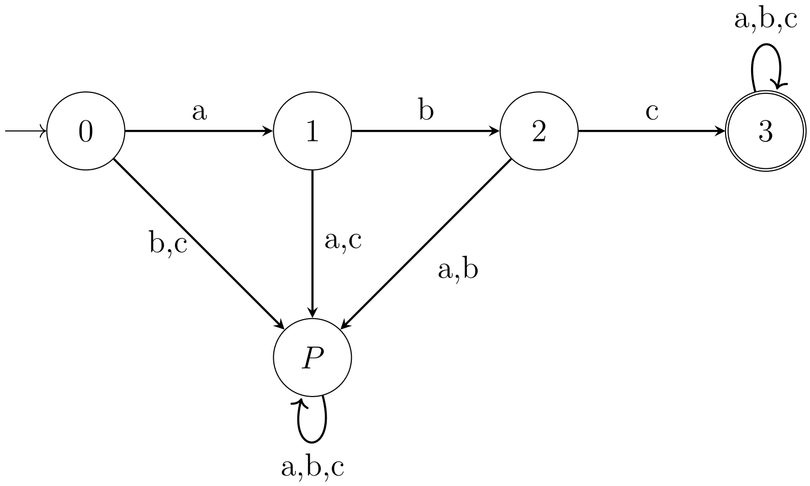
Automate complet

Un automate est dit **complet**, s'il chaque état possède une transition pour chaque symbole de l'alphabet. Par exemple, l'automate précédent est complet, tandis que le suivant ne l'est pas :



Exercice 2. Coder une méthode `est_complet(self)` qui permet de tester si un automate est complet ou non.

En pratique, il est toujours possible de transformer un automate en un automate complet **équivalent**, c'est à dire un automate qui reconnait le même langage.
L'algorithme consiste à ajouter un **état puits**, qui ne soit pas final, vers lequel on enverra toutes les transitions manquantes. Pour l'automate de l'exemple précédent, on obtient :



On obtient bien un automate équivalent puisque P n'est pas final. À noter que la boucle de P sur lui-même est nécessaire pour respecter la définition de la complétude.

Exercice 3. Implémenter une méthode `complete(self)` de complétion d'automate.

États accessibles et co-accessibles

Un état est **accessible**, s'il existe un mot qui permet de l'atteindre.
Si tous les états sont accessibles, alors l'automate est **accessible**.

Exercice 4.
1. Coder une méthode `etats_accessibles(self)` qui retourne la liste des états accessibles d'un automate.
   Penser aux parcours en profondeurs et en largeurs à partir de l'état initial.
2. Ajouter deux méthodes `accessible(self, etat)` et `est-accessible(self)` qui permettent de savoir si un état particulier est accessible et si un automate est accessible.

Un état est **co-accessible**, s'il existe un mot qui permet d'aller vers un état final depuis cet état.
L'algorithme est quasiment identique : on initialise la liste d'états à visiter avec l'ensemble des états finaux, puis on cherche leurs prédécesseurs.

Exercice 5.
1. Implémenter des méthodes `etats-coaccessible(self)`, `coaccessible(self,etat)` et `est_coaccessible(self)` qui retournent une liste de tous les états co-accessibles de l'automate spécifié, respectivement retourne vrai si l'état spécifié est co-accessible, respectivement retourne vrai si l'automate est co-accessible.
2. Un automate qui est à la fois accessible et co-accessible est dit **émondé**. Définir une méthode `est_emonde(self)` qui retourne vrai si l'automate est émondé.