Rapport TER Extraction de réseaux

Réalisé par Jauroyon Maxime

TABLE DES MATIÈRES

CONTEXTE

Le sujet de ce TER en bio-informatique consiste à créer des autotests pour détecter la présence de différentes molécules (ou leur absence) grâce à un circuit logique de réactions avec une réponse colorimétrique. Pour comprendre son fonctionnement, prenons l'exemple d'un bocal rempli d'eau dans lequel nous avons ajouté des milliers de vésicules afin de constater un changement de couleur. Ces dernières sont isolantes aux molécules d'eau et aux grosses molécules, mais permettent aux petites molécules d'entrer ou aux grosses molécules qui ont été préalablement préparées pour entrer. Cela permet de provoquer les réactions en chaîne souhaitées à l'intérieur de la vésicule, ce qui entraîne la production d'une molécule qui change la couleur de la vésicule (qui était auparavant transparente). La partie complexe de ce processus est le circuit logique de réactions, qui se trouve dans les vésicules et impose certaines conditions à respecter pour que la chaîne de réactions se déroule correctement.

Mais qu'est-ce qu'une réaction ? Nous avons une enzyme, qui mesure entre 5 et 10 nm, sur laquelle les molécules, qui font souvent 1 nm, peuvent s'accrocher. Tout se déplace dans l'eau, et les molécules les plus petites se déplacent plus rapidement que les grosses molécules en raison de la résistance de l'eau. Plus il y a de molécules, plus elles se bousculent, augmentant ainsi les chances de se lier. C'est la saturation de molécules à laquelle nous avons recours lorsque nous voulons qu'une réaction se produise lorsqu'elle a attrapé la molécule A, mais qu'elle lui manque la molécule B pour activer la transformation. En effet, pour qu'une enzyme active la transformation, elle a besoin que tous ses substrats soient collés au même instant. Souvent, lorsqu'un substrat se colle, il ne reste que quelques secondes avant de se décrocher. Les produits de la transformation sont alors relâchés quasi simultanément, avant que l'enzyme ne retrouve sa forme originale pour attendre à nouveau ses substrats. Cependant, il existe un cas où la transformation ne se produit pas : si un inhibiteur est collé à l'enzyme, alors elle est bloquée dans son état jusqu'à ce qu'il se décroche.

Ce sont ces réactions et inhibitions qui vont nous permettre de créer des portes logiques pour nos vésicules. Si une enzyme a besoin de A et B pour produire P, alors c'est une porte "ET", car il faut beaucoup de A et B pour produire beaucoup de P. Si une enzyme E1 a besoin de A pour produire P et une enzyme E2 a besoin de B pour produire P, alors nous avons une porte "OU" en mettant les 2 enzymes dans la vésicule, car beaucoup de A ou de B produit beaucoup de P. Une enzyme qui a besoin de A pour produire P mais étant inhibée par I, alors si on sature A, nous aurons une porte "NEG", car si beaucoup de I alors pas beaucoup de P, mais si peu de I nous aurons plein de P. Il existe d'autres façons de faire la porte "NEG" avec 2 réactions dont une plus rapide que l’autre par exemple mais souvent on ne depassera pas 4 réactions pour faire une seule porte et cela nous donne une idée de la grande quantité de réactions que nous aurons besoin pour réaliser un circuit logique dans notre vésicule, qui est la raison de ce TER.

OBJECTIF

Il existe un programme complexe pour obtenir les circuits logiques à partir d'une liste de réactions et d'inhibitions, par exemple avec le fichier brenda.ssa contenant environ 60k réactions et inhibitions. Mais appliquer le programme sur autant de données prendrait beaucoup trop de temps surtout qu'il existe des réactions qui n'interviendront jamais dans un chemin entre les 2 molécules désirer, souvent entre un biomarqueur et de l'eau oxygénée ou un colorant. L'objectif est donc de faire un pré-traitement du fichier afin de renvoyer exclusivement les réactions pouvant participer à un chemin et donc d'avoir beaucoup moins de réactions à traiter pour l'algorithme qui vient après.

Pour cela nous avons codé un analyseur lexical et syntaxique pour parser la liste de réactions et inhibitions afin d'obtenir un objet Ocaml que l'on peut transformer pour obtenir un graphe sur lequel on va pouvoir appliquer nos algorithmes de recherches de chemins afin de renvoyer uniquement les réactions pouvant faire partie d'un chemin demandé.

METHODE

Lexing

La première étape est de tokeniser le fichier avec ocamllex, via alexer.mll, pour ce faire il faut prendre connaissance de la sémantique à respecter pour l'écriture des réactions et inhibitions. Voici comment se formule une réaction :

"enzyme" : "substrat1" + substrat2 -> "produit1" + produit2 | 910 uM, 1.3 mM - 365; // 0-14

Les noms d'enzymes et molécules peuvent être entre parenthèses ou non, on doit avoir autant de concentrations (910 uM) que de substrats, et les commentaires (après //) ne doivent pas être pris en compte. Le chiffre après les concentrations représente la quantité maximale produite par seconde et les concentrations sont ce qui est nécessaire pour chaque substrat afin d'en produire la moitié.

Mais toutes ces données ne sont pas utiles pour la recherche de chemins donc on les tokenize en string tandis que les noms d'enzymes et molécules sont ceux que l'on va comparer le plus donc on va les tokenizer avec des entiers pour accélérer l'exécution de notre programme. Pour ce faire nous avons une variable globale, dans atypes.ml, qui sera un tableau contenant les string des noms et nous stockerons dans les tokens les indices des emplacements. Ce tableau est rempli durant la phase de tokenisation avec une fonction de hachage, get_index(), afin d'encore accélérer l'exécution du programme comme montrer dans l'annexe 1.

```
| double_quote not_doube_quote* double_quote as i        { ID(Atypes.get_index i) }
| idS id* as i                                            { ID(Atypes.get_index i) }
```

Annexe 1 : règles de tokenisation des noms avec et sans parenthèses, alexer.mll

Parsing

La deuxième étape est donc de parser la suite de tokens obtenus avec menhir, via aparser.mly, en utilisant des règles pour capturer correctement la liste des réactions et inhibitions qui ne doit pas être vide et chaque réaction doit avoir au minimum 1 substrat, 1 produit et donc 1 concentration. Durant cette étape on différencie les réactions des inhibitions car la formulation d'une inhibition est différente :

"E1" : "M1" | 1 uM;

Ceci indique que l'enzyme E1 est inhibé par la molécule M1. Nous construisons donc une liste de réactions et inhibitions (2 objets Ocaml distincts) ce qui nous permet de compter le nombre de réactions dans le fichier en incrémentant une variable globale, dans atypes.ml, qui nous serviras pour faire un tableau de la bonne taille pour le graphe, voir annexe 2.

```
{ Atypes.reac_table_length := !Atypes.reac_table_length + 1; Reaction(i,l1,l2,c,v) }
```

Annexe 2 : incrémentation du compteur du nombre de réactions croisées pendant le parsing, aparser.mly

GRAPHE

Vérifications

Une fois le fichier parsé, avec parse_automata() de autil.ml, nous avons un objet Ocaml que nous pouvons manipuler. Nous pouvons donc afficher la liste des réactions et inhibitions après le parsing , seuls les commentaires et lignes vides devraient avoir disparu. Il est également possible de vérifier que nous n'avons pas d'erreurs dans le fichier, si aucune erreur n'a eu lieu durant le parsing, cela signifie que la seule erreur encore possible est de ne pas avoir autant de concentrations que de substrats, dans ce cas nous le détectons et cessons l'exécution du programme en affichant la ligne qui pose problème, via automata_is_interpretable() de autil.ml.

Construction

Notre graphe sera un simple tableau de noeuds représentant toutes les réactions, que nous avons compté donc la taille est exactement le nombre de réactions. Chaque noeud est composé de la réaction qu'il représente (pour l'affichage), des indices de son enzyme, ses substrats et ses produits (pour les vérifications heuristiques des chemins), de la liste des noeuds qui se connectent à lui (aretes entrantes) et des noeuds auxquelles il se connecte (aretes sortantes). Une connexion entre 2 noeuds existe lorsqu'un produit du premier apparaît dans un substrat du second, c'est-à-dire qu'il consomme la sortie de l'autre, voir annexe 3 pour le type ocaml. On le remplit donc en parcourant la liste des réactions.

```
(*          reaction   name  substrat     product      in_edges     out_edges *)
type node = reaction * int * (int list) * (int list) * (int list) * (int list)
```

Annexe 3 : definition du type pour les nœuds, atypes.ml

En addition de ce tableau représentant le graphe nous construisons aussi un tableau pour les inhibiteurs mais celui-ci est de la même taille que le tableau servant à stocker les noms de molécules. Dès que l'on croise un inhibiteur dans la liste au lieu d'une réaction nous ajoutons à la liste, se trouvant dans la case représentant l'enzyme inhibée, la molecule qui l'inhibe. Nous arrivons donc à partir de la liste des réactions et inhibitions obtenues, à construire le graphe avec les réactions et construire le tableau répertoriant tous les inhibiteurs de chaque molecule, voir annexe 4.

```
(* table for the nodes *)
let reac_table = Array.make !Atypes.reac_table_length None in
(* table like the one storing names but store inhibitors for each reaction name *)
let inhib_table = Array.make 65536 (ref []) in
```

Annexe 4 : création des tableaux pour le graphe et les inhibitions, atransitions.ml

CHEMINS

Definition

Nous devons trouver tous les chemins « correct » entre une molecule de départ et une molecule d'arrivée. L'idée est donc de diviser la recherche en 2, une partie qui part du début et l'autre de la fin puis de réunir les 2, ce qui réduit énormément la complexité en temps et espace. Pour qu'un chemin soit correct nous avons plusieurs conditions à respecter : aucun nœud ne doit produire la molecule de départ ou consommer la molecule de fin car on ne pourrait évidemment pas tester la présence d'une molecule si on l'a produit ou si on consomme la molecule finale qui est censée déclencher un changement de couler, aucun nœud (réaction) ne doit être inhibé par un produit d'un autre nœud (réaction) sinon l'enchainement des réactions ne marcherait plus après 1 seul passage, et enfin aucun nœud à par le dernier ne doit produire la molecule de fin et même raisonnement pour la molecule de début car on ne veut pas prolonger des chemins déjà finis ou commencés.

Implementation

Nous commençons par initialiser les connexions entre les nœuds, pour chaque nœud on parcourt tout le tableau et on lui rajoute dans les nœuds rentrant tous les nœuds qui ont un produit qui correspond à un de ses substrats et on lui rajoute dans les nœuds sortant tous les nœuds qui ont un substrat qui correspond à un de ses produits. Nous pouvons ensuite récupérer la liste de tous les nœuds de départs et d'arrivée qui vont nous servir pour savoir d'où partir. Ces 2 étapes ne faisant pas partie de la recherche de chemins, on ne prend pas en compte leur temps d'exécution.

Étant donné que la stratégie adoptée est de diviser la recherche en 2 nous avons 1 algorithme par longueur de chemin demandé, allant de 1 à 6. Voir find_index_n() dans ainterpreter.ml pour les détails des fonctions.

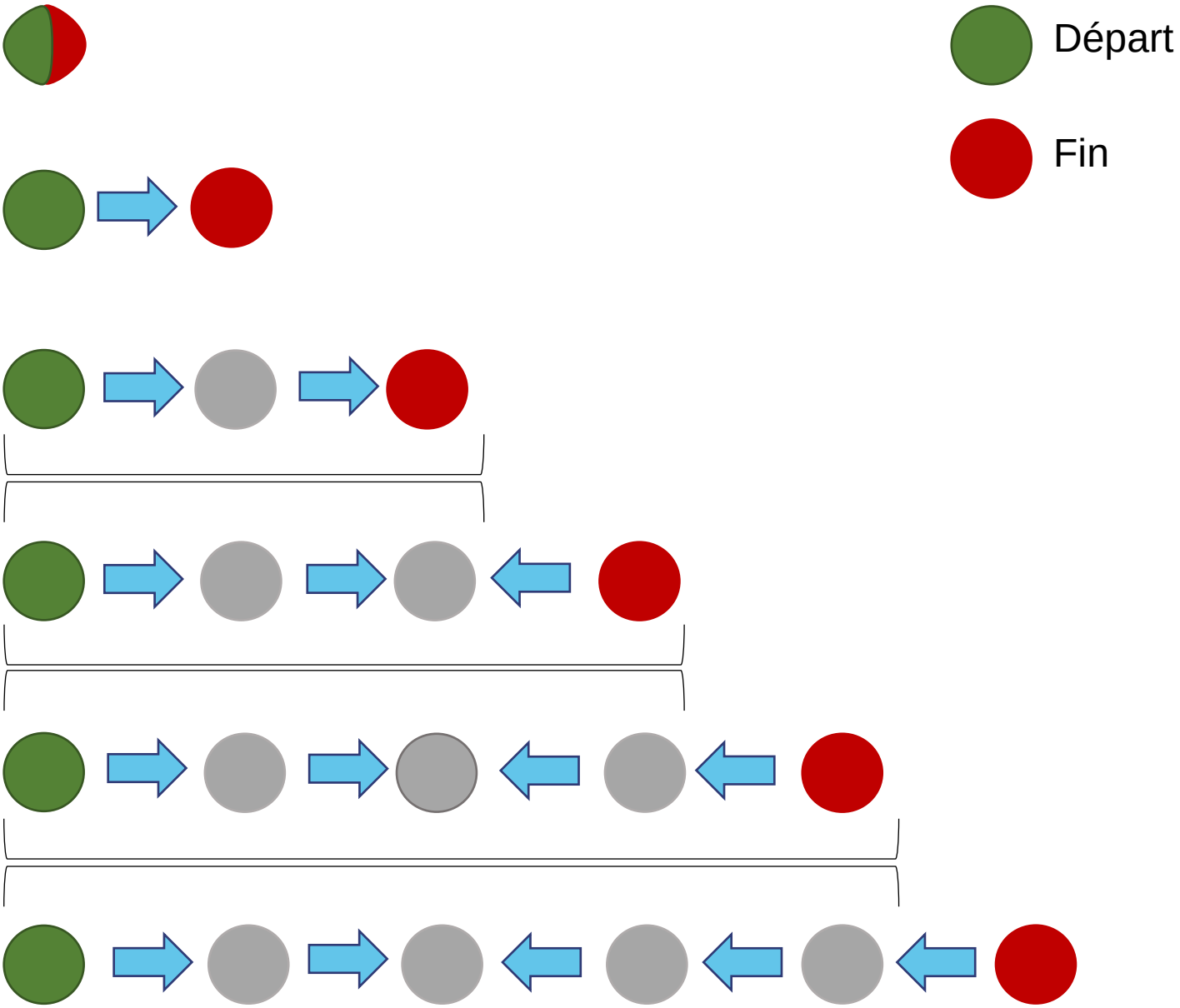
Longueur 1 : Trivial car nous prenons simplement les nœuds qui sont à la fois un départ et une arrivée et qui ne consomment pas la molecule d'arrivée ou produisent celles du départ (aussi qu'isl ne s'inhibent pas eux-mêmes mais vraiment très peu plausible).

Longueur 2 : Très simple aussi car nous prenons tous les nœuds de départs qui ne sont pas dans les arrivées et qui vérifient les heuristiques classiques (ne pas produire la molecule de départ +ne pas consommer la molecule de fin + ne pas s'inhiber) puis nous regardons les chemins corrects vers leurs voisins, c'est-à-dire les voisins qui sont une arrivée mais pas un départ, qui vérifient les heuristiques classiques et qui ne sont pas inhibés par un produit du départ et ne produisent pas un inhibiteur pour le départ.

Longueur 3 : comme d'habitude, nous prenons tous les nœuds de départs qui sont pas dans les arrivées et qui vérifient les heuristiques classiques (pas produire la molecule de départ + pas consommer la molecule de fin + ne pas s'inhiber) puis nous regardons les chemins vers leurs voisins (degré 1) et les voisins de leurs voisins (degré 2). Puis nous appliquons les heuristiques habituelles a l'exception que les voisins de degré 1 ne doivent être ni un départ ni une arrivée et les voisins de degré 2 doivent être une fin mais pas une arrivée.

Jusqu'à maintenant lorsque l'on a parler des voisins je faisais référence aux voisins sortants donc on n'a fait qu'aller depuis le départ vers la fin, or l'idée est de diviser la recherche ! c'est pourquoi les algorithmes de longueurs 4 à 6 permettent de rééquilibrer ceci en partant de la fin, voir annexe 5 pour les schémas.

Longueur 4-6 : L'idée est la même pour ces 3 dernières longueurs. On part des nœuds qui sont une fin mais pas un départ et qui verifient les heuristiques classiques puis on récupère la liste de leurs voisins entrants et on utilise l'algorithme de la longueur précédente pour trouver tous les chemins des départs vers cette liste de voisins sachant qu'on lui donne l'information du nœud d'arrivée (pour les heuristiques d'inhibitions) et un boolean a false pour dire qu'il doit vérifier que ses nœuds à la fin des chemins ne sont pas une fin car on est en train de l'utiliser pour une recherche de plus grandes longueurs que lui.



Annexe 5 : schémas des chemins calculés par les différentes longueurs

UTILISATION

Fonctionnalités

Pour faire tourner le projet il faut ouvrir un terminal dans le répertoire du projet et écrire : « ./automata » qui va compiler et lancer le programme automatiquement. Les différentes fonctionnalités sont additives dans notre programme, ce qui est très utile pour tester le programme lorsque l'on veut afficher le fichier que l'on traite (petit fichier test) et demander un chemin.

-help : pour afficher la liste des options (cela arrive par défaut lorsque vous lancez le programme sans donner de fichier de données)

-v : pour afficher la version du programme

-p : pour afficher le fichier de données après le parsing

-t : pour afficher les listes des réactions (via le tableau construit) et les inhibiteurs liés à leurs réactions (via le tableau construit)

-o filepath : pour que (quasiment) tout soit écrit a l'endroit donné (au lieu de stdout)

-l n : pour demander la liste des réactions intervenant dans un chemin de longueur n

-i : pour désactiver l'utilisation des inhibiteurs dans les heuristiques

Résultats

Nous avons reçu des fichiers contenant la liste des réactions attendues par leurs requêtes afin de pouvoir tester notre programme. Pour tous, excepter celui avec les inhibitions prises en compte dans les heuristiques, le programme renvoie exactement la même liste de réactions et quasiment dans le même ordre mais cette différence dans l'ordre s'explique par le fait que nous ne changeons pas l'ordre des lignes alors que le programme de l'encadrant change l'ordre pour optimiser la recherche. Le temps d'exécution de notre programme est rapide car seulement quelques secondes (jamais plus de 2 secondes pour les requêtes à tester) mais la recherche prend quelques milliemes de secondes tandis que l'initialisation des aretes entre les nœuds prend souvent entre 1 et 2 secondes, voir annexe 6.

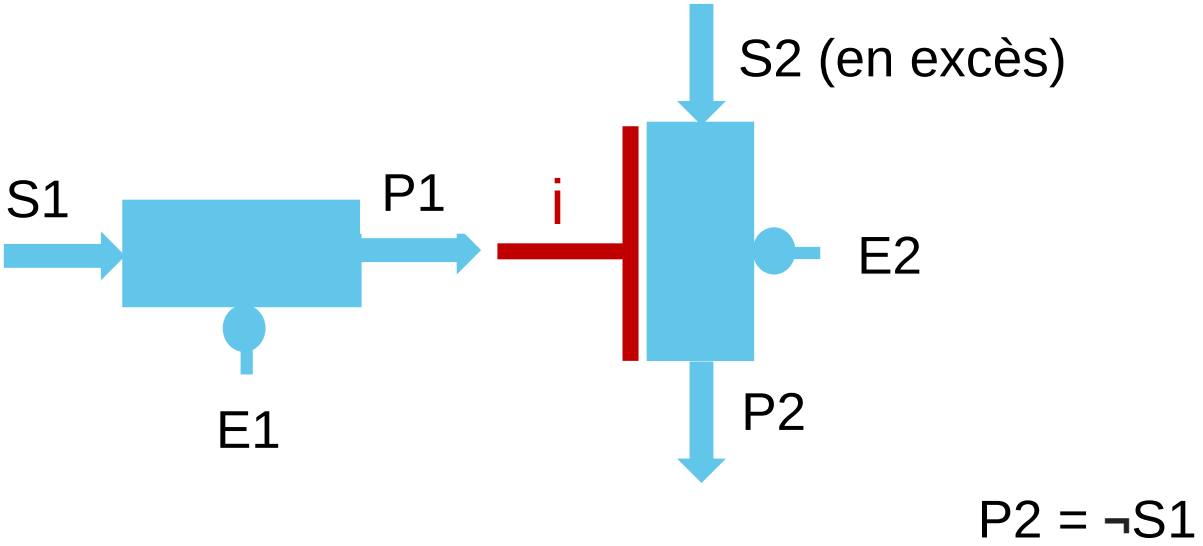
```
maxime@DESKTOP-CBR75GN:~/Univ-TER$ ./automata -o tests/d4-aceton-H2O2.txt brenda.ssa -i -l 4
Enter the start enzyme: "aceton"
Enter the end enzyme: "H2O2"
interpreting "aceton" to "H2O2" with length 4 :
edges construction time : 1.354011
start/end list construction time : 0.000313
search paths time : 0.021141
found 285 solutions
```

Annexe 6 : affichage d'une requête executée par le programme

Pour le fichier avec les inhibitions prises en compte dans les heuristiques, mon programme trouve moins de solutions que si les inhibitions étaient désactivées car les inhibitions ne peuvent qu'enlever des possibilités de chemins or ce n'est pas réellement le cas mais il n'était pas demandé par le sujet de gérer les cas où les inhibiteurs créent de nouveaux chemins. Donc notre programme fonctionne correctement dans le cadre du TER mais pourrait être amélioré pour gérer des cas de chemins utilisant des inhibiteurs.

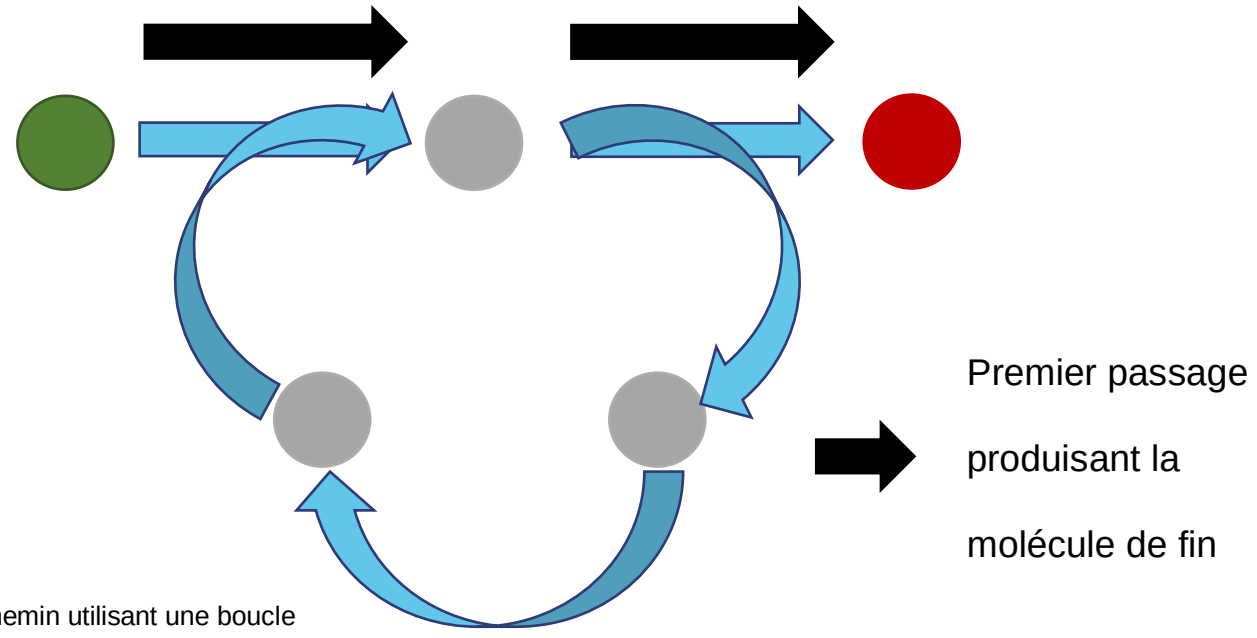
Pistes d'amélioration

Notre programme ne traite que les chemins composés de réactions et utilise les inhibiteurs pour supprimer des chemins. Or dans la réalité les inhibiteurs peuvent intervenir dans un chemin car ils servent par exemple de porte NEG, voir annexe 7, néanmoins cela change un peu l'architecture des chemins et complexifie grandement la recherche de chemins ce qui explique pourquoi ce n'était pas demander par le TER.



Annexe 7 : chemin utilisant un inhibiteur

Il existe aussi un autre critère que je n'ai pas incorporer dans mes recherches de chemins, les boucles ne devraient pas être des chemins valables car avant même le premier tour de boucle on constatera la production de la molecule de fin, voir annexe 8. L'implementation de ce critère devrait être possible mais nécessiterait de rajouter 1 argument à toutes les fonctions de recherche pour passer le tableau des nœuds actuellement traiter pour le chemin (car coder en Ocaml). Néanmoins sans avoir pris en compte cette contrainte, les résultats obtenus sont ceux attendus pour les tests car les cas de boucles sont très rares.



Annexe 8 : chemin utilisant une boucle