ICC (SV) – Mini-projet

Aurélien Ferlay, Barbara Jobstmann, Ehsan Pajouheshgar, Jamila Sam

(Version 1.0)

Ce document est en couleur et contient des liens cliquables. Il est préférable de le consulter en format numérique. Il est très important de bien le lire, **entièrement** et de **comprendre les structures de données fournies** et le rôle des fonctions demandées avant de vous lancer dans le codage.

1 Introduction

Les processus métaboliques sont des ensembles de réactions chimiques qui se produisent à l'intérieur des organismes vivants. Ces processus impliquent des réseaux complexes d'enzymes et de réactions biochimiques qui sont responsables de la synthèse de molécules essentielles, de la décomposition des nutriments et de la production d'énergie. (Voir Fig. 1 pour un exemple de réseau métabolique à l'échelle du génome). Les chercheurs étudient les processus métaboliques pour comprendre leurs fonctions et leurs implications dans les maladies humaines, et pour concevoir des médicaments qui ciblent des enzymes ou des processus biochimiques spécifiques. En outre, l'ingénierie métabolique a conduit au développement de nouvelles biotechnologies et de nouveaux bioproduits. La compréhension des processus métaboliques est cruciale pour faire progresser notre compréhension de la biologie, améliorer la santé humaine et développer de nouveaux bioproduits.

Cependant, l'identification d'un processus en laboratoire est un exercice compliqué. C'est pourquoi les chercheurs ont développé des techniques informatiques pour analyser les réseaux métaboliques. Ces techniques peuvent être classées en (1) **recherche de chemin** et (2) **approches stœchiométriques** [1]. Les approches de recherche de chemin se concentrent sur la recherche d'un chemin direct d'un composé source donné à un composé cible donné. Les approches stœchiométriques visent à trouver le processus « complet », qu'elles définissent comme un ensemble minimal de réactions dans un pseudo-état stable. En général, un processus plus court est souvent préféré à un processus plus long parce qu'il nécessite un plus petit nombre d'enzymes et peut donc être stocké dans un génome plus court, qui à son tour est plus susceptible d'être conservé par l'évolution.

2 Objectifs

Dans les deux premières parties de ce projet, nous nous concentrons sur la recherche de processus métaboliques à partir d'un graphe. Nous considérons le réseau métabolique

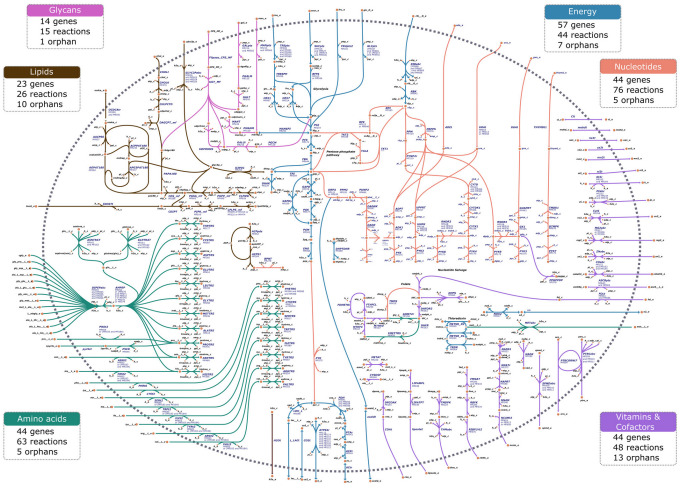

FIG. 1 : Réseau métabolique à l'échelle du génome provenant de [2]. Les cercles représentent les composés et les lignes indiquent les réactions métaboliques. La ligne en pointillé représente la membrane cellulaire, avec des réactions de transport reliant le milieu intracellulaire à l'environnement extracellulaire.

comme un graphe et recherchons un chemin qui transforme un composé donné (appelé réactif) en un autre composé donné (appelé produit). La longueur d'un processus est le nombre de réactions impliquées. Comme les processus les plus courts sont souvent les plus probables, les approches de recherche de chemin utilisent les concepts de chemin le plus court et de k-chemin le plus court. Dans la partie 1 de ce projet, vous mettrez en oeuvre diverses fonctions sur des graphes. Dans la partie 2, vous implémenterez une fonction pour trouver un processus le plus court et une fonction pour trouver tous les processus les plus courts. Dans la troisième et dernière partie de ce projet, nous souhaitons donner une idée des techniques utilisées dans les approches stœchiométriques. Par conséquent, nous montrons comment calculer les concentrations à l'état d'équilibre pour un chemin linéaire simple[1] et utiliser ces concentrations pour calculer la vitesse des réactions impliquées dans le chemin donné. Cela nous permettra de déterminer **le processus le plus rapide parmi tous les processus les plus courts** (c'est-à-dire différents processus comportant le même nombre de réactions). Les approches stœchiométriques impliquent la résolution d'équations différentielles et sont souvent appliquées à des sous-réseaux. Dans notre projet, nous nous concentrons sur un cas spécial simple dans lequel le sous-réseau est un chemin simple.

En résumé, le projet vise à trouver l'ensemble des réactions qui convertissent un réactif en un produit en passant par des réactions intermédiaires. Le flux sera déterminé dans des conditions d'équilibre et nous permettra de déterminer la vitesse de la séquence de réactions (processus). Une fois que tous les chemins les plus courts possibles auront été trouvés, il s'agira de déterminer le chemin le plus « rapide ». Dans un processus, la vitesse de transformation du réactif en produit est limitée par la vitesse de la réaction intermédiaire la plus lente (Étape cinétiquement déterminante). Dans le cadre du projet, les réactions seront toujours réversibles. Le modèle de cinétique réversible Michaelis-Menten déterminera la vitesse de réaction (flux).

[1] pas le réseau complet, ce qui serait beaucoup trop complexe

3 Les structure et le code fournis

Ce projet est divisé en 3 étapes :

- Implémenter des fonctions qui traitent le réseau métabolique, créent un graphe d'adjacence à partir de celui-ci et effectuent une Parcours en largeur sur ce graphe.
- Implémenter des fonctions pour trouver les chemins les plus courts entre deux nœuds d'un graphe.
- Implémenter des fonctions pour trouver les concentrations des produits chimiques à l'état d'équilibre et utilisation de ces concentrations pour trouver le chemin le plus court avec la vitesse de réaction la plus élevée.

Chaque partie de ce projet a plusieurs fonctions à compléter. Des types et fonctions utilitaires sont fournis dans le fichier `utilis.hpp`. Les types et structures à utiliser sont définis dans le fichier `pathsearch.hpp`.

> Il est important de consulter les types et fonctions fournis dans le fichier `pathsearch.hpp`.

Vous devez compléter le code du fichier `pathsearch.cpp`. Des instructions précises vous sont données dans ce document. Les prototypes des fonctions à implémenter se trouvent dans le fichier `pathsearch.hpp` et **ne doivent pas être modifiés**.

Notez qu'en dehors des fonctions imposées, vous êtes libre de définir toutes fonctions supplémentaires que vous jugez pertinentes. C'est une bonne pratique de créer un code propre et modulaire.

3.1 Types et Structures

Dans ce projet, vous utiliserez les structures des données suivantes.

3.1.1 Types de base

Les trois types suivants sont utilisés pour identifier les composés et les réactions dans notre réseau métabolique. Un composé est une molécule qui peut être soit le substrat, soit le produit d'une réaction. Une molécule (substrat) qui, par une réaction enzymatique, est transformée en une autre molécule (produit).

```
typedef std::string CompoundName;
typedef int ReactionID;
typedef int CompoundID;
```

3.1.2 La structure `Reaction`

Une réaction $S \rightarrow P$ est définie par les deux composés impliqués dans la réaction, le substrat S et le produit P. Une enzyme catalyse la réaction et sa capacité à catalyser la réaction est déterminée par les quatre constantes suivantes :

- K^S : Plus la constante K^S est élevée, plus la concentration de substrat nécessaire à une activité enzymatique significative est importante. Cela reflète généralement une faible affinité de l'enzyme pour son substrat. Plus la constante K^S est faible, plus l'activité enzymatique maximale obtenue pour une faible concentration de substrat est importante. L'affinité de l'enzyme pour le substrat est élevée.
- K^P : Plus la K^P est élevée, plus la concentration de produit requise pour une activité significative de l'enzyme dans la direction opposée est élevée. Ceci reflète généralement une faible affinité de l'enzyme pour son produit.
- V^+ représente l'activité maximale de l'enzyme dans le sens direct.
- V^- représente l'activité maximale de l'enzyme dans la direction opposée.

Voici le code de la structure `Reaction` :

```
struct Reaction {
  std::pair<CompoundID,CompoundID> compounds;
  double V_plus;
  double V_minus;
  double K_S;
  double K_P;
};
```

3.1.3 La structure `Network`

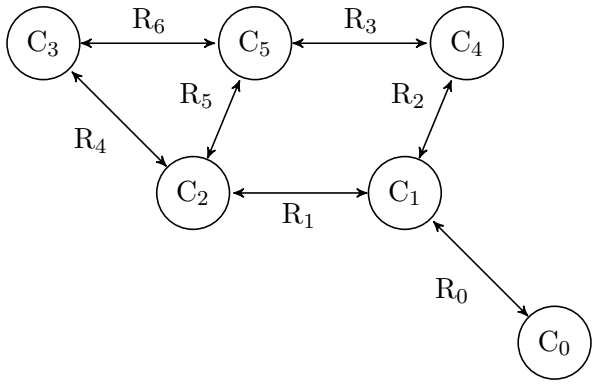

Fig. 2 : Un réseau métabolique simple avec 6 composés et 7 réactions

La structure `Network` stocke les informations complètes d'un réseau métabolique. Elle comprend la liste des composés et la liste des réactions permettant à un composé de se transformer en un autre.

```
struct Network {
  std::vector<CompoundName> compounds;
  std::vector<Reaction> reactions;
};
```

L'indice dans ces vecteurs correspond respectivement à un identifiant de composé (`CompoundId`) et un identifiant de réaction (`ReactionId`).

La figure 2 est un exemple d'un réseau métabolique simple avec 6 composés et 7 réactions.

3.1.4 La structure `AdjacencyGraph`

Le structure `AdjacencyGraph` est une structure de donnée aidant à implémenter les algorithmes calculant les plus courts chemin dans un graphe (voir la section A.1).

```
typedef std::vector<std::map<CompoundID, ReactionID>>
   AdjacencyGraph;
```

Pour chaque nœud stocke l'index des nœuds qui en sont adjacents avec la réaction les reliant[2]. Notez qu'un entrée dans une `map` n'est autre qu'une `pair`.

Par exemple, le `AdjacencyGraph` pour le graphe de la Figure 2 serait :

```
{
  {{1, 0}},                 // C 0: edge to C 1 with React. 0
  {{0, 0}, {2, 1}, {4, 2}}, // C 1: -> 0(0), -> 2(1),...
  {{1, 1}, {3, 4}, {5, 5}}, // C 2: -> 1(1), -> 3(4),...
  {{2, 4}, {5, 6}},         // C 3: -> 2(4), -> 5(6)
  {{1, 2}, {5, 3}},         // C 4: -> 1(2), -> 5(3)
  {{2, 5}, {3, 6}, {4, 3}}  // C 5: -> 2(5), -> 3(6), ...
}
```

3.1.5 La structure `BFS`

Comme décrit dans l'annexe (voir la section A.1), vous allez utiliser un algorithme de parcours en largeur (breadth-first search en anglais) pour trouver les chemins les plus courts dans un graphe partant d'un nœud vers tous les autres nœuds.

La structure de donnée `BFS` est un utilitaire stockant le résultat de l'algorithme de parcours en largeur de graphe :

```
struct BFS {
  CompoundID start;
  std::vector<std::vector<int>> parents;
  std::vector<int> distances;
};
```

[2] voir https ://cplusplus.com/reference/map/map/ pour plus de documentation sur le type `map`

Le champ `start` correspond au composé source de notre parcours de graphe. L'algorithme trouvera la distance la plus courte entre chaque composé du graphe et le composé source. Le vecteur `parents` est utilisé pour stocker le dernier composé visité **dans les chemins les plus courts venant du composé `start`**.
Par exemple, pour le graphe de la figure 2, le résultat du parcours en largeur en partant du noeud source 0, serait un objet BFS ayant pour contenu :

```
0, // noeud source
{{-1}, // parents du noeud 0  : -1 (source universelle)
 {0},  // parents du noeud 1  :  0
 {1},  // parents du noeud 2  : 1
 {2},  // ...
 {1},
 {2,4} // parents du noeud 5 : 2 et 4
 }
,
{0, 1, 2, 3, 2, 3} // liste des distances
```

La liste des distances est telles que son ième élément est la plus courte distance entre le noeud source et le noeud i. Ainsi les plus courts chemins entre les noeuds 0 et 5 sont ceux qui arrivent à 5 en passant soit par le noeud parent 2 soit par le noeud parent 4. Ils auront tous deux comme longueur total (distance) 3.

3.1.6 Les types représentants des processus métaboliques et des concentrations

Les types `Path` et `Paths` stockent les réactions impliquées dans un processus métaboliques et une liste de processus métaboliques respectivement.

```
typedef std::vector<ReactionID> Path;
typedef std::vector<Path> Paths;
```

Le type `Concentractions` stocke la concentration chimique de chaque composées. Il est utilisé pour calculer les concentrations en conditions d'équilibre.

```
typedef std::map<CompoundID, double> Concentrations;
```

3.1.7 Les Fonctions Utilitaires fournies

Dans le fichier `utils.cpp` sont fournies plusieurs fonctions utilitaires pour initialiser un réseau métabolique ainsi que les concentrations chimiques depuis un fichier.

La fonction `Network read_network(std::string network_filename);` lit un réseau métabolique depuis un fichier texte et renvoie une instance de la structure `Network`.

La fonction `Concentrations read_initial_concentrations(const Network& network, std::string filename);` lit les concentrations chimiques d'un fichier texte. Ces données seront utilisées pour les concentrations en conditions d'équilibre.

Il n'est pas nécessaire de lire ni de comprendre cette partie du programme. La fonction `main` dans le fichier `main.cpp` vous montre comment appeler ces deux fonctions.

Nous vous fournissons quelques fonctions `to_string()` pour vous aider à afficher les structures de données utilisées dans ce projet.

3.1.8 Fichier de test fourni

Nous vous fournissons plusieurs fichiers de données dans le dossier `data` afin de tester votre code. Chaque fichier texte (`.txt`) correspond à un réseau métabolique. Comme mentionnée dans la section précédente, nous vous fournissons les utilitaires pour initialiser une structure `Network` à partir de fichiers. Vous trouverez des exemples de leurs utilisations dans les fichiers `main.cpp` et `unit_test.cpp`. Le fichier `data/basic.txt` correspond au réseau de la Figure 2. Pour générer des données de graphes, nous avons notamment utilisé le site suivant : http ://hpat.kavrakilab.org.

Test et vérification des cas limites

Par souci de simplicité, vous considérerez que toutes les données fournies en tant que paramètres aux fonctions que nous vous demandons d'implémenter sont correctes. Dans cette optique, il est de votre responsabilité de vérifier que votre programme se comporte correctement. Il est important de **vérifier soigneusement à chaque étape que vous produisez bien des données correctes avant de passer à l'étape suivante qui utilisera ces données.**

Pour vous aider dans vos vérifications, nous vous fournissons quelques exemples de tests disponibles dans le fichier `unit_test.cpp` (et qui sont déjà invoqués pour vous dans le fichier `unit_test.cpp`). Vous pouvez utiliser ces tests et les compléter pour vous assurer que tout fonctionne. Notez que le code dans `main.cpp` et `unit_test.cpp` ne seront pas testé en tant que tel pendant la correction. Modifiez-les comme bon vous semble, en fonction de vos besoins. Pour tester le code que vous avez écrit pour chaque partie, vous pouvez appeler la fonction `run_unit_tests(part)` en lui passant l'index `part` de la partie. Par exemple, si vous voulez tester votre code pour la partie 3 de ce projet, appelez la fonction `run_unit_tests(3)` dans le fichier `main.cpp`.

Attention les tests fournis ne sont pas exhaustifs. Il est recommandé d'essayer de les compléter par soit même, notamment pour les cas limites.

4 Implémentation

Ce projet est composés de 3 parties. Dans chaque partie, vous devez implémenter quelques fonctions dans le fichier `pathsearch.cpp`

> Les parties 2 et 3 peuvent en principe être abordées en parallèle si vous le souhaitez. Il est cependant important de bien comprendre comment elles sont censées s'enchaîner.

4.1 Partie 1

L'objectif de la première partie du projet est d'implémenter quelques fonctions de base permettant de travailler avec réseau métabolique. Il s'agira en particulier de créer une instance de la structure `AdjacencyGraph` à partir d'un réseau. Cette structure sera utilisée par un algorithme de parcours en largeur (« breadth-first search ») qui permettra plus tard d'identifier les chemins les plus courts. **Commencez par lire l'annexe A.1 pour en savoir plus sur cet algorithme et la façon dont il sera utilisé**.

Voici la liste des fonctions à implémenter :

- `CompoundID find_compoundID(const Network& network, CompoundName name);` Cette fonction permet de trouver l'identifiant d'un composé à partir de son nom. La valeur -1 sera retournée si le composant n'existe pas.
  La fonction `test_part1` appelée par le `main` fourni montre le résultat attendu pour le réseau de la Figure 2 :

```
std::cout << " ======= Testing findCompoundID ======= " <<
    std::endl;
Network basic_network(read_network("data/basic.txt"));
std::cout << find_compoundID(basic_network,"C4") <<
    std::endl; // Should display: 4
```

- `AdjacencyGraph build_adjacency_graph(const Network& network);` Cette fonction crée l'instance de la structure `AdjacencyGraph` pour le réseau métabolique donné. La fonction `test_part1` fournie montre le résultat attendu pour le réseau de la Figure 2 :

```
std::cout << " ======= Testing build_adjacency_graph
    ======= " << std::endl;
/* Should display:
  0:C0: {{1,0}}
  1:C1: {{0,0}{2,1}{4,2}}
  2:C2: {{1,1}{3,4}{5,5}}
  3:C3: {{2,4}{5,6}}
  4:C4: {{1,2}{5,3}}
  5:C5: {{2,5}{3,6}{4,3}}
 */
```

```
std::cout << to_string(basic_network,
   build_adjacency_graph(basic_network));
```

- `ReactionID find_reactionID(const AdjacencyGraph& graph, CompoundID index1, CompoundID index2)`. À partir d'un graphe d'adjacence et deux ID de composés, cette fonction trouve l'identifiant de la réaction entre ces deux composés. Cette fonction est utile pour exprimer les processus comme des séquences de réactions (arêtes du graphe). La valeur -1 sera retournée s'il n'existe pas de réaction entre les deux composés.

  La fonction `test_part1` fournie montre le résultat attendu pour le réseau de la Figure 2 :

```
std::cout << " ======= Testing find_reactionID ======= " <<
   std::endl;
AdjacencyGraph graph {
    {{1, 0}},
    {{0, 0}, {2, 1}, {4, 2}},
    {{1, 1}, {3, 4}, {5, 5}},
    {{2, 4}, {5, 6}},
    {{2, 1}, {5, 3}},
    {{2, 5}, {3, 6},{4, 3}}
};
std::cout << find_reactionID(graph, 3, 5) << std::endl; //
   Should display: 6
```

- `BFS bfs(const AdjacencyGraph& graph, CompoundID start)`. Étant donné un graphe d'adjacence et une source `start`, cette fonction effectue un parcours en largeur et renvoie une instance de la structure `BFS`. La fonction `test_part1` fournie montre le résultat attendu pour le réseau de la Figure 2 :

```
std::cout << " ======= Testing bfs ======= " << std::endl;
/* Should display:
  start:0
  Node0: {-1 }, distance : 0
  Node1: {0 }, distance : 1
  Node2: {1 }, distance : 2
  Node3: {2 }, distance : 3
  Node4: {1 }, distance : 2
  Node5: {2 4 }, distance : 3
 */
BFS bfs_result(bfs(graph, 0));
std::cout << to_string(bfs_result) << std::endl;
```

Tests

Pour tester cette partie, vous pouvez utiliser le programme principal fourni. Ouvrez le fichier `main.cpp`, vous y verrez que de nombreux tests sont fournis et **vous êtes libre de décommenter les instructions pertinentes pour tester votre code**. La fonction `test_part1` vous donne un exemple de tests simples que vous pouvez compléter à votre guise pour la première partie du projet. Par ailleurs, les dernières instructions exécutent la fonction `run_unit_tests`. Elle est définie dans `unit_test.cpp`. Ouvrez le fichier `unit_test.cpp` et examinez la fonction `run_unit_tests`. Vous verrez qu'elle appelle différents tests préprogrammés pour chaque partie du projet. Pour cette partie du projet, vous pouvez appeler cette fonction `run_unit_tests(1)` qui exécutera le test préprogrammé pour la partie 1 du projet. L'exécution devrait afficher le message `[Passed]` pour tous les tests. En cas d'erreur, vous devriez obtenir un message d'erreur avec `[Failed]` vous indiquant la différence entre la valeur attendue et la valeur calculée. Par exemple :

```
---------------
test_find_compoundId
---------------
Testing with network 7paths.txt
[Passed]
[Passed]
[Passed]
[Passed]
[Passed]
[Passed]
[Passed]
```

qui indique que la fonction `find_compoundId` a passée avec succès les tests. Sinon elle affichera par exemple :

```
 --------------
test_find_compoundId
---------------
Testing with network 7paths.txt
[Failed]
   expected: 0
   computed: 2
```

ce qui indique que `find_compoundID` a donné la valeur erronée qui est affichée sous `computed:`, au lieu de la valeur correcte affichée sous `expected`.

4.2 Partie 2

L'objectif de la deuxième partie du projet est d'implémenter les fonctions nécessaires pour trouver tous les chemins les plus courts entre deux nœuds d'un graphe. Vous devrez implémenter deux fonctions dans cette partie :

- `Path find_shortest_path(const AdjacencyGraph& graph, CompoundID srcID, CompoundID destID)`. À partir un graphe d'adjacence, cette fonction trouve un plus court chemin entre le composé source avec `srcID` et le composé de destination avec `destID`. Par exemple, si nous appelons cette fonction sur le graphe illustré dans la Figure 2 avec `srcID=0` et `destID=6`, la fonction devrait renvoyer l'un des deux chemins de réaction les plus courts (soit React0 → React1 → React5 ou React0 → React2 → React3).
- `Paths find_all_shortest_paths(const AdjacencyGraph& graph, CompoundID srcID, CompoundID destID);` Étant donné un graphe d'adjacence, cette fonction trouve tous les plus courts chemins entre les composés source et destination. Par exemple, si nous appelons cette fonction sur le graphe de la Figure 2 avec `srcID=0`, et `destID=6`, la fonction devrait renvoyer les deux processus les plus courts (React0 → React1 → React5 et React0 → React2 → React3).

Une description des algorithmes ainsi que des conseils pour les mettre en œuvre sont fournis dans l'annexe A.1. La fonction `test_part2` fournie montre le résultat attendu pour le réseau de la Figure 2 :

```
// Create the graph shown in Figure 2
AdjacencyGraph graph (
{
  {{1, 0}},                  // C 0: edge to C 1 with React. 0
  {{0, 0}, {2, 1}, {4, 2}}, // C 1: -> 0(0), -> 2(1),...
  {{1, 1}, {3, 4}, {5, 5}}, // C 2: -> 1(1), -> 3(4),...
  {{2, 4}, {5, 6}},          // C 3: -> 2(4), -> 5(6)
  {{1, 2}, {5, 3}},          // C 4: -> 1(2), -> 5(3)
  {{2, 5}, {3, 6}, {4, 3}}  // C 5: -> 2(5), -> 3(6),...
});

// Find a single shortest path from compound 0 to 5
auto path (find_shortest_path(graph, 0, 5));
std::cout  << "Found path: " << to_string(path) << std::endl;
/* Should display :
  Found path: Reactions: 0 1 5
 */

// Find all the shortest paths from compound 0 to 5
Paths paths (find_all_shortest_paths(graph, 0, 5));
std::cout << "Number of shortest paths: " << paths.size() <<
   std::endl;
std::cout << to_string(paths);
/* Should display :
```

```
 Number of shortest paths: 2
 Path Number 1 - Reactions: 0 1 5
 Path Number 2 - Reactions: 0 2 3
*/
```

Tests

Pour tester cette partie, vous pouvez utiliser la fonction `test_part2`. Vous pouvez aussi appeler la fonction de test unitaire comme suit : `run_unit_tests(2)` Si votre code est correct, vous ne devriez obtenir que les messages `[Passed]`. Voici les deux fonctions de tests unitaires proposées pour la partie 2 du projet :

```
test_find_shortest_path();
test_find_all_shortest_paths();
```

4.3 Partie 3

Dans la troisième partie, vous devez implémenter des fonctions pour trouver les concentrations à l'état d'équilibre des composés chimiques et vous utiliserez ces concentrations pour trouver, parmi les plus courts chemins possibles, celui ayant la vitesse de réaction la plus élevée. Voici les fonctions qu'il vous est demandé de coder (pour chacune d'elle, vous trouverez un exemple d'exécution dans la fonction `test_part3`) :

- `double michaelis_reversible_rate(const Reaction& R, const double& S, const double& P);` Étant donné une structure de réaction `R` et les concentrations des substrats source et produit, cette fonction évalue la vitesse de réaction de Michaelis-Menten à l'aide de la formule suivante :

$$r(S \to P) = \frac{V^+ \frac{S}{K^S} - V^- \frac{P}{K^P}}{1 + \frac{S}{K^S} + \frac{P}{K^P}}$$

  Où S et P correspondent à la concentration des composés de la source et du produit, respectivement[3].

- `Concentrations compute_ss_concentration(const Network& network, const Path& path, const Concentrations& initial_concentrations, double dt=1e-3)` Étant donné un processus (une liste de réactions métaboliques) à l'intérieur d'un réseau métabolique, cette fonction calcule la concentration à l'état d'équilibre de chaque composé chimique impliqué en utilisant la méthode d'intégration d'Euler. Dans cette fonction, `double dt` est la constante de temps utilisée pour la méthode d'intégration d'Euler. Veuillez vous référer à l'annexe A.2 pour plus de détails. Pour calculer les différences relatives de concentration, vous pouvez utiliser la fonction C++ `fabs` (valeur absolue pour des doubles).

[3] Pour la signifcation des autres constantes, revoir au besoin le paragraphe 3.1.2

- `double compute_path_rate(const Network& network, const Path& path, const Concentrations& ss_concentrations);` Étant donné un processus et la concentration à l'état d'équilibre des composés impliqués, cette fonction calcule la vitesse de réaction combinée. Pour un processus $p = c_0, c_1, c_2 \ldots, c_n$, où les c_i sont les composés impliqués, la vitesse de réaction combinée $f(p)$ est définie comme la plus petite vitesse de réaction le long de ce processus :

$$f(p) = \min(\{r(c_i \to c_{i+1}) \mid n > i \geq 0\})$$

- `Path find_fastest_path(const Network& network, const Paths& paths, const Concentrations& initial_concentrations, double dt);` Cette fonction évalue les vitesses de réactions combinées pour tous les processus donnés et renvoie le processus ayant la vitesse de réaction la plus élevée.

La fonction `compute_ss_concentration` est la plus exigeante. Il est recommandé de bien la modulariser par exemple au moyen d'une fonction

```
Concentrations euler_implicite(const Network& network,
                               const Path& path,
                               const Concentrations
                                  c_in, double dt)
```

qui calcule les concentrations obtenues par la méthode de Euler, sur les composés d'un chemin donné, après écoulement d'un pas de temps `dt` en partant de concentrations initiales données.

Tests

Pour tester cette partie, vous pouvez appeler la fonction `test_part3` ainsi que la fonction de test unitaire comme suit : `run_unit_tests(3)` Si votre code est correct, vous ne devriez obtenir que les messages `[Passed]`. Voici les fonctions de tests unitaires proposés pour la partie 3 du projet :

```
test_michaelis_reversible_rate();
test_compute_ss_concentration();
test_compute_path_rate();
test_find_fastest_path();
```

A Annexe

A.1 Notes sur l'algorithme de parcours en largeur pour trouver les chemins les plus courts

Nous considérons un graphe tel que celui représenté dans la Figure 3. Notre objectif est de calculer tous les chemins les plus courts entre un nœud donné (la source) et un autre (la destination). La longueur d'un chemin est son nombre d'arêtes.

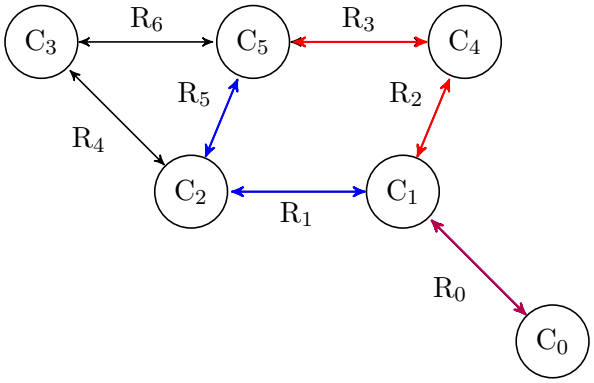

FIG. 3 : Les deux chemins les plus courts du nœud C0 au nœud C5

A.1.1 La fonction `bfs`

Un algorithme couramment utilisé pour atteindre cet objectif consiste à explorer le graphe à l'aide d'une stratégie de parcours en largeur [Lien]. Cet algorithme explore le graphe en partant du nœud source puis traversent les nœuds les plus proches aux nœuds les plus éloignés. Le pseudo-code de cet algorithme est fourni ci-dessous :

```
Input : source
Output :  pour tous nœuds, sa liste de parents sur ses chemins les plus courts partant de la source + la longueur de ses chemins.
// Nous utilisons une file (queue) pour stocker les nœuds à visiter
queue.push(source)
parents[start] = { -1 }
for each node do distance[node] = infinity // Utilisez la constante INT_MAX
end for
distance[start] = 0
while queue is not empty do
    current_node = queue.front() // le premier élément de la file
    queue.pop() // enlevé le premier élément de la file
    for  all the nodes, adj, adjacent to current_node do
        if distance[adj] > distance[current_node] + 1 then
            // Nous avons trouvé une distance plus faible
            // Donc nous écrasons les parents précédents
            // et insérons current_node dans parents[adj]
            distance[adj]=distance[current_node] + 1
            // En ajoutant adj à la fin de la file, cela force l'algorithme
            // à explorer les nœuds plus proche de la source avant les nœuds
            // adjacent de adj.
            queue.push(adj)
            parents[adj]={current_node}
        else
            if  distance[adj] == distance[current_node] + 1 then
                // Nous avons trouvé un autre plus court chemin de même distance
                parents[adj].push_back(current_node)
```

end if
end if
end for
end while

Par exemple, pour le graphe de Figure 3, l'algorithme de parcours en largeur donnera le résultat suivant :

```
Node 0: parents= { -1  } , distance: 0
Node 1: parents= { 0  }  , distance: 1
Node 2: parents= { 1  }  , distance: 2
Node 3: parents= { 2  }  , distance: 3
Node 4: parents= { 1  }  , distance: 2
Node 5: parents= { 2, 4 }, distance: 3
```

Dans ce projet, la fonction `bfs` sera chargée de calculer ce résultat qui prendra la forme de la structure de données `BFS` (voir `pathsearch.hpp`). Pour l'implémentation en C++, vous pouvez utiliser le type prédéfini `queue` (https ://cplusplus.com/reference/queue/queue/).

A.1.2 La fonction `build_adjacency_graph`

Pour implémenter plus facilement l'algorithme de parcours en largeur, nous avons besoin d'un moyen simple de récupérer les nœuds qui sont *adjacents* à un autre. C'est pourquoi, avant d'implémenter la fonction `bfs`, il vous est demandé d'implémenter la fonction `build_adjacency_graph`.

Le graphe d'adjacence de l'exemple de la figure 3 prendra la forme suivante :

```
{{{1, 0}},
 {{0, 0}, {2, 1}, {4, 2}},
 {{1, 1}, {3, 4}, {5, 5}},
 {{2, 4}, {5, 6}},
 {{1, 2}, {5, 3}},
 {{2, 5}, {3, 6}, {4, 3}}
 }
```

qui se lit comme :

- le nœud 0 est adjacent au nœud 1 via la réaction 0
- le nœud 1 est adjacent au nœud 0 via la réaction 0, au nœud 2 via la réaction 1 et au nœud 4 via la réaction 2
- etc.

A.1.3 La fonction `find_shortest_path`

Il est facile de calculer le plus court chemin entre une `source` et une `destination` en utilisant les informations calculées par l'algorithme de parcours en largeur :

Input : Le graphe d'adjacence, la source et la destination
Output : Un des plus courts chemins entre la source et la destination représenté par une liste de réactions
shortest_path_as_list_of_nodes = {}
current_node = destination
// Le premier parent n'est pas trouvé dans la structure BFS
first_parent = first parent of current_node
while first_parent != -1 **do**
    add first_parent at the end of shortest_path_as_list_of_nodes
    current_node = first_parent
    first_parent = first parent of current_node
end while
return shortest_path_as_list_of_nodes

A.1.4 La fonction `find_all_shortest_path`

Pour trouver un plus court chemin à l'aide de la structure BFS, il suffit de suivre le premier parent de chaque nœud. Nous pouvons également explorer récursivement tous les parents pour trouver tous les plus courts chemins. Pour implémenter la fonction `find_all_shortest_path`, nous recommandons donc d'implémenter une fonction d'aide `recursive_find_paths` selon l'algorithme suivant :

Function `recursive_find_paths`
Input : la liste des plus cours chemin à construire
Input : le chemin actuellement en construction
Input : les parents de tous les nœuds (comme calculer par `bfs`)
Input : la destination
if destination == -1 **then**
    if `current_shortest_path` not empt **then**
        add `current_shortest_path` at the end of `shortest_paths`
    end if
end if
for all parent p of destination **do**
    `current_shortest_path`.push_back(destination)
    `recursive_find_paths` (`shortest_paths` , `current_shortest_path`, parents, p)
    `current_shortest_path`.pop_back()
end for

A.2 Notes sur le calcul des concentrations à l'état d'équilibre

Nous illustrerons cette fonction par l'exemple suivant. Imaginons que nous voulions calculer les concentrations à l'état d'équilibre pour le chemin de réaction suivant (extrait de la glycolyse) :
$\xrightarrow{v_{in}} glc \xrightarrow{HEX} g6p \xrightarrow{PGI} f6p \xrightarrow{PFK} fdp \xrightarrow{k_{out}}$

Il y a trois enzymes impliquées dans cette réaction, HEX, PGI, PFK et nous voulons déterminer leur vitesse de réaction. On suppose que chaque réaction suit la cinétique réversible de Michaelis-Menten, et donc que les vitesses de réaction se calculent comme suit :

$$r_{HEX} = r(glc \to g6p) = \frac{V^+_{HEX}\frac{glc}{K^{glc}_{HEX}} - V^-_{HEX}\frac{g6p}{K^{g6p}_{HEX}}}{1 + \frac{glc}{K^{glc}_{HEX}} + \frac{g6p}{K^{g6p}_{HEX}}}$$

$$r_{PGI} = r(g6p \to f6p) = \frac{V^+_{PGI}\frac{g6p}{K^{g6p}_{PGI}} - V^-_{PGI}\frac{f6p}{K^{f6p}_{PGI}}}{1 + \frac{g6p}{K^{g6p}_{PGI}} + \frac{f6p}{K^{f6p}_{PGI}}}$$

$$r_{PFK} = r(f6p \to fdp) = \frac{V^+_{PFK}\frac{f6p}{K^{f6p}_{PFK}} - V^-_{PFK}\frac{fdp}{K^{fdp}_{PFK}}}{1 + \frac{f6p}{K^{f6p}_{PFK}} + \frac{fdp}{K^{fdp}_{PFK}}}$$

(pour la signification des constantes impliquées, revoir le paragraphe 3.1.2).

Le taux de changement pour chaque substrat est quant à lui déterminé par le flux d'entrée et le flux de sortie des réactions pour ce substrat. Il existe également un flux d'entrée externe v_{in} pour le premier substrat et un flux de sortie externe v_{out} pour le dernier substrat :

$$\frac{d}{dt}glc = v_{in} * (1.0 - glc) - r_{HEX}$$
$$\frac{d}{dt}g6p = r_{HEX} - r_{PGI}$$
$$\frac{d}{dt}f6p = r_{PGI} - r_{PFK}$$
$$\frac{d}{dt}fdp = r_{PFK} - fdp * v_{out}$$

Pour déterminer les concentrations des composés à l'état d'équilibre, nous allons procéder par simulation à l'aide de la méthode dite d'Euler. Cette méthode, permet de calculer la concentration à un temps $t + 1$ connaissant la concentration au temps t. Pour notre exemple, cela se ferait comme suit :

$$glc(t+1) = glc(t) + \Delta t \times \frac{d}{dt}glc$$

$$g6p(t+1) = g6p(t) + \Delta t \times \frac{d}{dt} g6p$$
$$f6p(t+1) = f6p(t) + \Delta t \times \frac{d}{dt} f6p$$
$$fdp(t+1) = fdp(t) + \Delta t \times \frac{d}{dt} fdp$$

où Δt est une petite constante Notez que les concentrations ne peuvent pas être négatives et qu'elles doivent être bridées en conséquence (c'est-à-dire que si elles deviennent négatives, elles doivent être ramenées à zéro).

La simulation répète ces calculs tant que la différence entre les concentrations au pas $t+1$ et celles au pas t ne sont pas négligeables. Dans le cas de notre exemple, nous arrêtons la simulation lorsque toutes les conditions suivantes sont remplies :

$$\frac{|glc(t+1) - glc(t)|}{glc(t+1)} < 10^{-8}$$
$$\frac{|g6p(t+1) - g6p(t)|}{g6p(t+1)} < 10^{-8}$$
$$\frac{|f6p(t+1) - f6p(t)|}{f6p(t+1)} < 10^{-8}$$
$$\frac{|fdp(t+1) - fdp(t)|}{fdp(t+1)} < 10^{-8}$$

La concentration initiale de chaque composé est fournie dans un fichier d'entrée (par exemple le fichier `7paths_concentrations.txt` pour le réseau `7paths.txt`).

Les constantes v_{in} et v_{out} sont données comme `const double V_IN` et `const double V_OUT` dans `pathsearch.hpp`.

Références

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, Afnizanfaizal Abdullah, and Hoshang Kolivand. Identification of metabolic pathways using pathfinding approaches : a systematic review. *Briefings in Functional Genomics*, 16(2) :87–98, 03 2016.

[2] Jean-Christophe Lachance, Dominick Matteau, Joëlle Brodeur, Colton J Lloyd, Nathan Mih, Zachary A King, Thomas F Knight, Adam M Feist, Jonathan M Monk, Bernhard O Palsson, Pierre-Étienne Jacques, and Sébastien Rodrigue. Genome-scale metabolic modeling reveals key features of a minimal gene set. *Molecular Systems Biology*, 17(7) :e10099, 2021.