Évolution d'automates pour le traitement de séquences génomiques

Rapport de projet

Équipe **B-13** :

Bonjour Mickael (mickael.bonjour@heig-vd.ch)

Mettler Samuel (samuel.mettler@heig-vd.ch)

Pantic Nemanja (nemanja.pantic@heig-vd.ch)

Seville Nathan (nathan.seville@heig-vd.ch)

Simeonovic David(david.simeonovic@heig-vd.ch)

Table des matières

Introduction

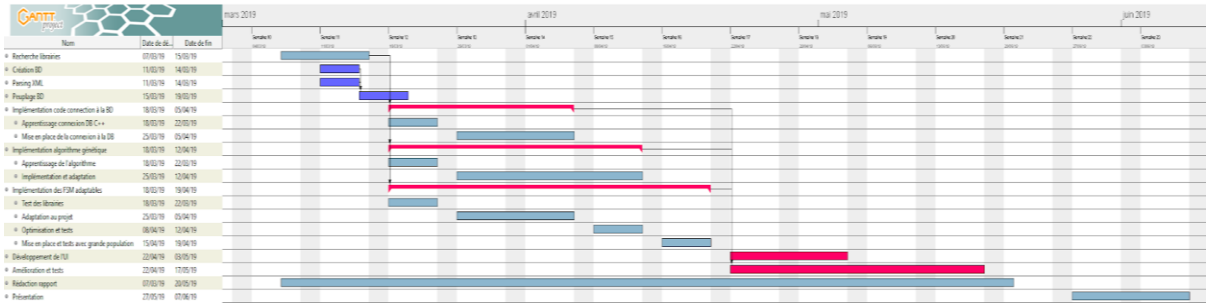
Une grande partie du comportement des êtres vivants est définie par son génome. L'étude du génome peut expliquer et faciliter la compréhension de nombreux phénomènes et c'est à quoi aspire ce projet. Plus particulièrement, nous nous intéressons au protéines qui découlent de ce génome. Les protéines, quant à elle, sont représentés par soit une séquence génétique (composé des 4 caractères ACGT) ou alors par une séquence protéique (composé de 20 caractères).

Notre application se base sur plusieurs points. D'un côté elle analyse les séquences génétique avec des automates (machine d'état) et de l'autre côté elle fait évoluer ces mêmes machine afin d'en construire des toujours plus performante et précise. En effet, construire des machines finis adéquates est particulièrement complexe, c'est pourquoi nous nous sommes orientés sur un algorithme évolutionniste.

Le rapport qui suit présente l'organisation interne lors du déroulement du projet, les choix lors de son déroulement ainsi que son développement. Une partie est aussi consacrée aux apprentissages ce qui représente une grande partie de ce projet.

Phase Initiale

Planification



Répartition du travail

Nous avons réparti les plus grandes tâches identifiées du projet comme suit, cette répartition représente la répartition initiale du travail qui a évolué lors du projet. Les référents sont néanmoins restés les mêmes et étaient chargés du bon déroulement de leurs tâches associées.

Tâches	Référents
Parsing XML	Samuel, David
Peuplage DB	Samuel
GA	Nemanja
States Machines	Nathan, Mickael (Peer programming)
Dispatcher	Nathan, Mickael (Peer programming)
UI	David

Déroulement du projet

Choix des technologies

Qt - State Machines

Nous avons choisi la technologie Qt afin d'implémenter nos machines d'états car elle présente de multiples avantages. Les QStateMachine ont l'avantage d'être mieux documentée, de posséder une syntaxe plus claire et d'avoir plus de fonctionnalité que tout autre librairie C++ de machine d'état.

Librairies que nous avons étudiées pour trouver celle qui correspondait le plus à nos attentes pour ce projet:

- https://digint.ch/tinyfsm
- https://doc.qt.io/qt-5/qstatemachine.html
- https://www.boost.org/doc/libs/1_64_0/libs/msm/doc/HTML/index.html
- https://www.etlcpp.com/fsm.html
- http://yasmine.seadex.de/yasmine.html

Galgo-2.0

Suite à différents tests et différentes comparaisons avec d'autres algorithmes génétiques, notre choix s'est porté vers Galgo-2.0 (https://github.com/olmallet81/GALGO-2.0). Nous l'avons choisis car tout d'abord la communauté sur Github semblait bien apprécier ce GA. Ensuite en comparaison à d'autres, il est plus facilement évolutif et a été plus simple à modifier. En effet, beaucoup des paramètres que nous cherchions sont déjà disponibles.

ElementTree

Suite à différentes recherches et comparaisons entre des librairies existantes, nous avons décidé d'utiliser la librairie ElementTree pour faire le parsing du XML. En effet, cette librairie semblait être très appréciée.

LibSSH

Durant nos recherche, nous n'avons pas trouvé grand chose sur la connexion SSH avec C++. La seul libraire qui permettait de le faire était LibSSH. Nous n'avons donc pas eu le choix de nous tourner vers celle-ci pour faire notre implémentation.

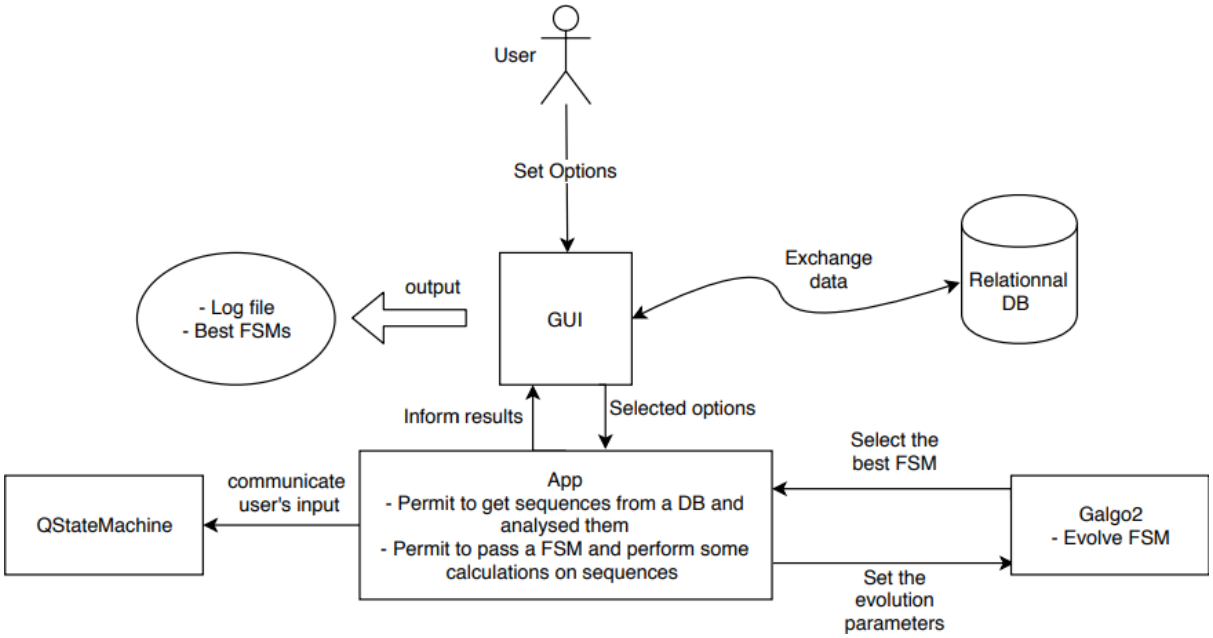
Malheureusement, LibSSH pouvez au mieux nous permettre d'accéder au shell de la machine distante. Nous avons donc basé notre implémentation de la récupération des données sur cette contrainte. Pour ce faire, nous envoyons donc une commande au bash de l'hôte distant. Cette commande enverra à la base de données la requête SQL donnée par l'utilisateur et va stocker le résultat de celle-ci dans un fichier en local. Une fois ce fichier crée, l'application va se charger de le récupérer et de le stocker sur votre machine.

Développement

Modélisation

En premier lieu, voici le container Diagram élaboré au début du projet. Cela explique le principe de base de notre application. Ce modèle a légèrement évolué pendant le projet mais rien de majeur, on avait une idée assez claire au début du projet.

Cependant, cette modélisation ne sert que de contexte et donc ne sera pas plus expliqué que cela.



Machines

Directions prises

QScxml

Nous sommes tout d'abord parti vers une structure se basant sur QScxml, une librairie du Framework QT qui permet de créer des machines d'états dynamiquement grâce à un fichier scxml qui est un standard pour la description des machines d'états. Cependant cette solution n'est pas très efficace. En effet, nous n'avons pas réussi à implémenter correctement ce genre de machine, à ce moment là nous n'avions pas encore conscience de la "Event Loop" de QT et de la nécessité de celle-ci pour les QStateMachine. De plus nous nous sommes heurtés à la lenteur des accès IO. Nous nous sommes éloigné de cette solution après avoir passé de nombreuses heures dessus. Mais cela a été important car cela nous a permis de nous initier réellement à ce que proposais QT et les raccourcis que cela apportait dans certains cas.

QStateMachine

Suite à cette remise en question sur QScxml nous nous sommes redirigés vers les QStateMachines (heureusement toujours dans le QT Framework). Nous avons donc pu réellement commencer à implémenter des machines d'états permettant d'analyser des séquences ADN. Au début nous avions implémenté la lecture/traitement de séquences nucléiques (rétrospectivement un mauvais choix, il

aurait été en effet plus judicieux de commencer par des séquences protéiques au vu de leur longueur). Par la suite, nous n'avons pas eu le temps d'implémenter la lecture de séquences protégiques.

Structure

MegaMachineManager

Nous avons décidé d'implémenter nos machines d'états à l'aide d'un manager qui permet de lancer plusieurs d'entre elles, toutes à l'écoute du même émetteur. Le manager agit comme intermédiaire entre les machines et l'émetteur dont la responsabilité est de lire les différentes séquences. Lorsqu'une séquence est terminée, le manager à la responsabilité de comptabiliser les scores et de réinitialiser à l'état initial les machines pour une prochaine séquence.

Les machines sont générées à partir de StateDescriptors qui permettent de définir un état de la machine, il contient ces transitions composée de l'état d'arrivée ainsi que du signal correspondant à la transition et de l'action de l'état (utilisé pour la comptabilisation des scores) qui peut être :

- Oui
- Non
- Rien

La machine retiendra le plus grand nombre de "Oui" ou "Non" comme résultat de la machine.

La structure utilisée est efficace et performante. Elle permet d'étendre ou d'adapter le travail déjà fait de manière simplifiée. Elle permet une extension avec le multithread des différentes machines afin d'améliorer la performance lorsque plusieurs d'entre elles sont lancées en même temps.

Le multithread à ce niveau n'a pas été implémenté lors de ce projet étant donné qu'une seule machine est lancée par le manager à chaque appel de la fonction objective de l'algorithme génétique car elle ne concerne qu'un seul individu.

En revanche le multithreading, comme expliqué dans la section "GA", est réalisé au niveau de la fonction objective pour améliorer ses performances car la fonction est gourmande en temps.

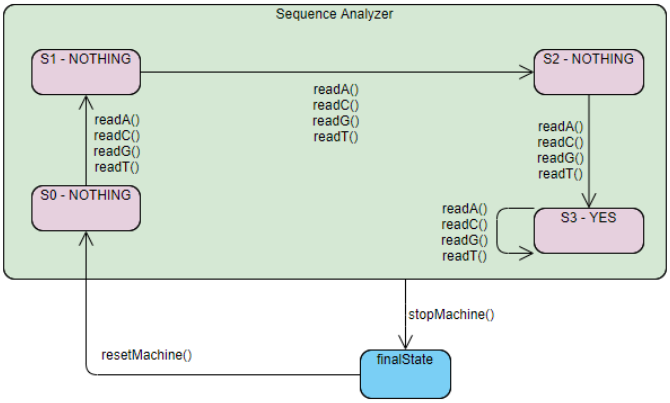
EmitterACGT

Cette classe permet la lecture des séquences et la transmission aux machines d'états afin d'avoir un traitement de la séquence par les machines. Cette classe souffre d'un problème d'optimisation, on lit toujours l'entier des séquences même si toutes les machines sont arrêtées, cela implique des problèmes de performances lors de séquences trop longues.

MegaMachine

La "MegaMachine" est ce qu'on peut appeler un "wrapper" sur des machines d'états (QStateMachine).Afin de spécialiser nos machines pour l'analyse des séquences. Nous implémentons un compteur d'alerte YES/NO que chaque état peut trigger en entrée. En effet, nous allons créer une machine selon les "StateDescriptor" (voir documentation développeur) et en plus de ces états (dans

le diagramme S0, S1, …) nous allons ajouter un *finalState* et des transitions propres au traitement de séquences.



Machines simple et simplifiée pour l'affichage du fonctionnement au propre.

Scénario

Typiquement quand l'émetteur passe à la prochaine séquence il émet d'abord un *finishedSequence()* afin d'arrêter toutes les machines en cours puis un *nextsequence()* afin de reset() les machines. Cela nous permet de repartir directement sur les machines sans problèmes. Quand une machine est stoppée (d'elle même ou par l'emitter) on calcule son score grâce à l'émetteur qui contient le résultat attendu et le MegaMachineManager qui tient un tableau des scores pour toutes les machines. On vérifie le nombre de YES et de NO de la machine et on prends le plus grand des deux pour avoir le résultat. Voir documentation développeur pour un diagramme de séquences plus détaillé du fonctionnement global.

Dispatcher

Le dispatcher est l'élément central de notre implémentation, c'est lui qui va permettre à nos différents composant de communiquer. Il permet de récupérer tous les différents paramètres relatif à l'algorithme génétique ainsi que le fichier de séquence à analyser. C'est lui qui va effectuer la traduction entre la représentation binaire de l'algorithme génétique et les machines d'états. A la fin d'une analyse celui-ci sauvegardera les données de la meilleure machine. Pour un fichier de séquence donné le dispatcher va effectuer autant d'analyse qu'il y a d'identifiant différent présent, pour chaque analyse il y aura autant de séquence attendant un oui (séquence relative à l'identifiant) que de non (séquence relative à une autre identifiant).

Le dispatcher permet également de récupérer une machine à partir d'un fichier et de l'exécuter sur un fichier de séquence fourni.

GA

Il a été nécessaire de modifier légèrement le code source de galgo afin qu'il corresponde à nos attentes. Nous avons du créer un nouveau constructeur pour celui-ci afin qu'il prenne des paramètres sous forme de vector plutôt que de parameter pack car non définissable en runtime.

GUI

L'interface utilisateur a été développé sur QT également. En effet, les machines d'états ayant été développées sur QT également, nous avons donc opté pour ce choix qui nous a paru évident.

Pour ce qui est de la connexion SSH, nous avons utilisé LibSSH. Nous avons donc créé plusieurs fonctions qui permettent le bon fonctionnement de la connexion SSH :

- sshConnect : créer la connexion SSH.
- channelConnect : créer un canal sur l'hôte distant afin d'accéder à son shell.
- sshWrite : envoie d'une commande bash à l'hôte distant.
- scpRead : télécharge un fichier depuis l'hôte distant et le stocke en local.

Malheureusement, la seul solution que nous avons trouvé pour récupérer les données depuis une base de données externe avec une connexion SSH, est de passer par un fichier temporaire sur le serveur. Ce fichier est généré avec la commande bash envoyé au serveur :

- PGPASSWORD=[password] psql -U [username] [DB name] -c \"Copy ([SQL request]) To stdout With CSV DELIMITER ';';\" > [remote file location]

Une fois ce fichier générer, nous le récupérons avec la fonction "scpRead" qui lit le fichier distant et le stocke en local dans ce fichier:

- /home/<username>/.local/share/MegaMachineEvolution/remoteSequences

Pour ce qui est du choix de fournir des données à l'application via un fichier local, nous avons utilisé QFileDialog qui permet d'ouvrir un explorateur de fichier. Une fois le fichier sélectionné, l'objet nous retourne simplement le chemin de celui-ci.

Une fois le fichier téléchargé ou sélectionné par l'utilisateur, nous passons simplement son chemin à la fenêtre "ParameterWindow".

La fenêtre de paramètre se charge simplement de récupérer les paramètres sélectionnés par l'utilisateur pour les passer au "Dispatcher". C'est ensuite lui qui va s'occuper d'envoyer ces paramètre aux machines d'états et à l'algorithme génétique.

Tout le reste du code sert simplement à l'affichage d'informations sur l'interface graphique.

Parsing XML et Base de donnée

Bien que notre projet n'ait pas comme aspect principal le traitement de données il nous était nécessaire de créer un outil permettant de traiter de très grands documents. En effet, les deux fichiers que nous avons utilisé pour peupler notre base de donnée faisaient respectivement 660 et 600 MB.

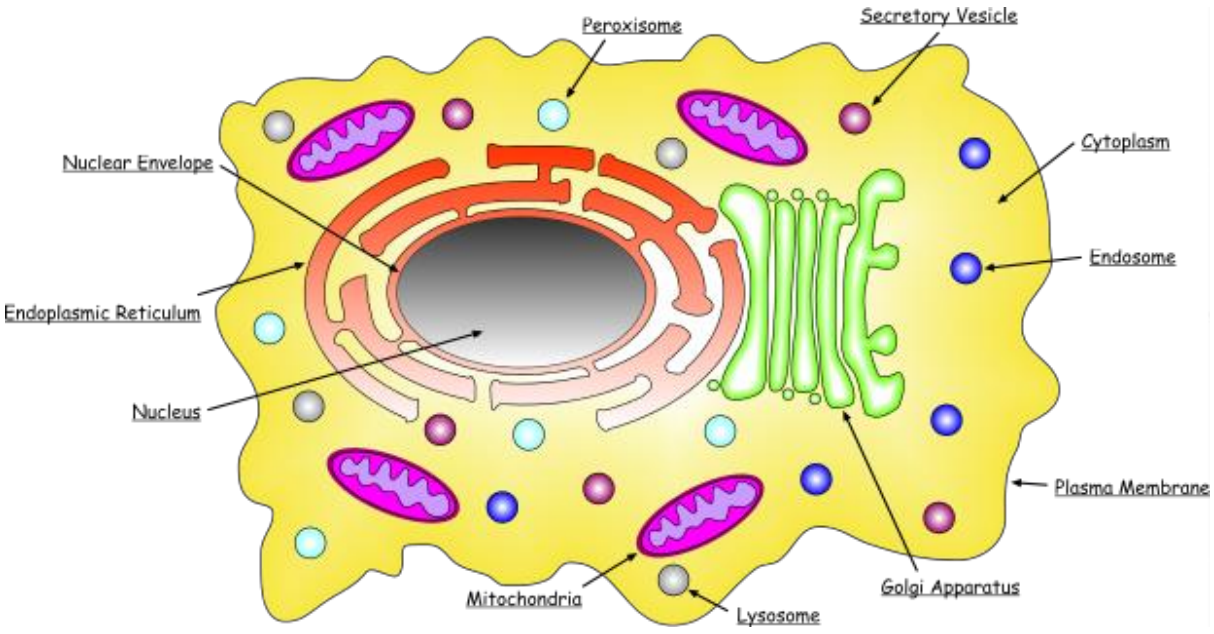
Les données nous viennent d'une base de donnée dédiée : *Locate subcellular localization database*. Cette base de donnée comprend les protéines contenues dans les humains et les souris.

Nous sommes intéressés par les localisations des protéines, donc nous avons besoin d'identifier les protéines. Les séquences nucléiques (composé des lettres ACGT) et les séquences protéiques (composé d'un alphabet de 20 lettres) étant liées l'une à l'autre, nous les avons enregistrés les deux. Un ID unique est également attribué à chaque paire de séquences.

Les localisations ont un GOID qui est unique et le nom correspondant à cette localisation. Après avoir parsé les deux documents, voici les 10 localisations que nous avons trouvé :

Golgi apparatus - cytoplasm -endoplasmic reticulum - plasma membrane - cytoskeleton - lysosome - peroxisome - mitochondrion - extracellular region - nucleus

Ces 10 emplacements sont les suivants :



Durant le déroulement du projet, le parsing nous a pris beaucoup plus de temps que prévu. En effet, nous pensions faire un parseur efficace en l'espace d'une semaine, hors, cela nous a pris beaucoup plus de temps. Il est possible d'expliquer cela par le dédoublement des informations dans la base de donnée. En effet, les XML n'étaient pas standards car certains ID étaient utilisés plusieurs fois sur des

éléments différents, ce qui nous a retardé. De plus, il s'agissait de quelques chose que nous n'avions jamais fait auparavant, donc, en plus de devoir trouver des technologies qui nous permettaient de parser un tel document, nous devions comprendre comment l'utiliser et l'adapter à nos besoins.

De plus, après avoir fourni une première version d'un parseur que nous pensions totalement fonctionnelle, nous avons réalisé bien plus tard que nous avions des doublons dans la base de donnée. Cela nous a permis de nous rendre compte que le parsing n'était pas optimal.

Nous avons donc recréé un nouveau parseur bien plus efficace au niveau du temps d'exécution et avec plus de fonctionnalités car nous utilisons des structures qui empêchent totalement les doublons.

Apprentissage

Parsing XML et Base de donnée

Interface utilisateur

L'interface graphique, nous a permis de découvrir l'assistant pour la création d'une interface graphique de QT. Nous avons donc appris à modéliser des interfaces utilisateurs grâce à cette outils. Nous avons aussi appris à interagir avec les objets de cette interface depuis le code.

Pour ce qui est de la connexion SSH. Nous avons appris à utiliser la librairie LibSSH. La prise en main de celle-ci n'a pas été des plus simples au début de l'implémentation. Heureusement pour nous, la libraire est bien documentée. Nous avons donc pu établir une session SSH entre notre machine local et un hôte distant. Nous avons aussi implémenté des fonctions qui nous permettent d'interagir avec la machine distante (sshWrite, scpRead).

Machines d'état

Dans cette partie nous avons pu réellement approfondir nos connaissances sur le QT Framework, c'est très motivant de mieux comprendre une programmation qui est plutôt orientée events. C'est une programmation assez dynamique et intéressante. On en a énormément appris sur les signaux QT et les slots et cela nous sert d'ailleurs pour des cours en parallèle on a donc été ravi d'avoir des clés en mains pour cela.

On a donc beaucoup appris sur l'Event Loop de QT car les QStateMachine ont besoin de cette fameuse Event Loop afin de fonctionner correctement. Je le mentionnes beaucoup car il a fallu bien comprendre ce genre de fonctionnement, en effet cela nous a consommé énormément de temps au départ du projet.

GA

Afin de nous faciliter le travail et avec l'accord du professeur nous avons choisis un GA déjà existant car il s'agit de quelque chose que nous ne connaissions pas. Les algorithmes génétiques sont de bonnes solutions pour différents problèmes d'optimisation, le principe de base est d'avoir une population de solutions, ces populations possèdent un score qui est attribué selon un objectif. À chaque génération nous allons :

1. Sélectionner les meilleurs individus(chromosomes) selon certaines conditions, ces différentes méthodes sont décrites plus loin dans cette section, principe de la théorie de Darwin.
2. Faire un croisement sur les chromosomes choisis lors de la sélection afin d'obtenir par reproduction un nouveau chromosome possédant des particularités des deux parents .
3. Effectuer une mutation sur une petite partie de la population, cela consiste en une altération de gènes d'un individu selon un taux de mutation.

Avec ces trois opérations nous obtenons une nouvelle population qui aura , en général, un score différent de la population précédente. On réitère tout ça *n* fois, *n* étant le nombre de générations.

Notre choix suite, à différents avis d'utilisateurs et de tests, s'est porté sur Galgo-2.0. Il s'agit donc d'un algorithme génétique développé en C++ par Olivier Mallet. Son principe est d'avoir une fonction de base que nous allons soit maximiser soit minimiser. Dans la documentation il est dit que ce GA est multi-threadable en utilisant OpenMP. Une section "Multithread" est disponible plus bas dans le rapport.

Nous avons la possibilité de choisir entre différentes méthodes de mutation, de sélection et de croisement. Voici celles qui sont implémentées de base dans Galgo-2.0

Sélection

- proportional roulette wheel selection (RWS)
- stochastic universal sampling (SUS)
- classic linear rank-based selection (RNK)
- linear rank-based selection with selective pressure (RSP)
- tournament selection (TNT)
- transform ranking selection (TRS)

Croisement :

- one point cross-over (P1XO)
- two point cross-over (P2XO)
- uniform cross-over (UXO)

Mutation :

- boundary mutation (BDM)
- single point mutation (SPM)
- uniform mutation (UNM)

Il est dit dans la documentation de Galgo-2.0 qu'il utilise par défaut ces paramètres avec :

- Aucune contrainte
- RWS
- P1XO
- SPM

Il s'agit d'un choix non expliqué par son créateur.

Les paramètres permettent d'instancier des chromosomes, nous avons la possibilité de choisir sa représentation en nombre de bits, il est possible de choisir une chaîne faisant entre 1 et 64 bits.

Il est également possible d'avoir un chromosome représenté sur 31 bits et un autre sur 63 bits. La taille n'est donc pas fixe à tous les chromosomes. Avec cette information Galgo va générer pour chaque population de chromosome un nombre aléatoire compris entre 0 et 2^*n* , *n* étant le nombre de bits.

Ce nombre est un entier non-signé. Il sera converti en string composé de 0 et de 1 et tronqué afin d'obtenir la taille souhaitée. Ensuite on travaillera sur cette valeur avec la mutation, etc. Les paramètres sont ensuite représentés soit comme un float, soit comme un double, il est impossible d'utiliser d'autres types que ces deux là.

Par défaut un paramètre est représenté sur 16 bits, nous pouvons également spécifier la range de la valeur du paramètre, par exemple, si nous voulons un nombre compris entre 32 et 78.

La classe GeneticAlgorithm possède des variables permettant d'influencer l'évolution des paramètres, ces derniers sont configurables dans notre GUI afin de laisser le choix à l'utilisateur. Il s'agit de :

- covrate : compris entre 0 et 1 (de base à 0.5), il s'agit du taux de croisement
- mutrate : compris entre 0 et 1 (de base à 0.05) il s'agit du taux de mutation
- SP : selective pressure pour la RSP, compris entre 1 et 2 par défaut à 1.5
- tolerance : condition de fin pour arrêter l'algorithme (inactif de base, il vaut 0)
- elitpop : taille de la population finale, ne peut pas être plus grand que la taille de la population de base (par défaut à 1)
- matsize : taille de la population mating , par défaut de la taille de la population
- tntsize : taille d'un tournoi, pour la méthode de sélection TNT (à 10 par défaut)
- genstep : affiche le résultat intermédiaire tous les <genstep> fois (à 10 par défaut)
- precision : nombre de décimales pour les résultats de sortis (5 par défaut)

Lorsqu'une contrainte est ajoutée elle est immédiatement prise en compte dans l'évolution des paramètres. Une contrainte est une pénalité appliquée sur le score de la population si une condition que nous avons choisis est atteinte. Cette pénalité se caractérise notamment par une perte de points dans son score.

Structure

Galgo est composé de plusieurs classes :

- Chromosome , cette dernière est la représentation d'un paramètre
- Converter , permet de convertir un string en long long et vice-versa
- Evolution , contient l'ensemble des méthodes de croisement,mutation,sélection et d'adaptation
- Galgo , inclut tous les outils nécessaires au fonctionnement de Galgo, c'est ici qu'on peut déclarer le comportement de OpenMP afin de faire du multi-threading
- GeneticAlgorithm , contient les variables permettant de paramétrer le comportement de notre algorithme génétique, on y spécifie les méthodes de croisements,etc ainsi que les paramètres, par exemple l'élitisme, le taux de croisement, taux de mutation, etc. Afin de faciliter l'implémentation nous avons créé une classe dispatcher, vous trouverez dans ce document une rubrique qui commente cette classe.
- Parameter , objets que nous faisons évoluer, on peut y spécifier des valeurs prédéterminées ou des aléatoires
- Population , ensemble de paramètres, on y stocke les populations courantes ainsi que la mating population
- Randomize , permet de générer des nombres aléatoires ou d'obtenir la valeur maximale d'un nombres sur N bits

Pour toutes les méthodes de sélection nous travaillons sur une population fournie en paramètre, les éléments choisis par cette méthode sont stockés dans une mating population.

Vous trouverez ci-dessous une description de l'algorithme théorique , suivi d'une description de son implémentation.

Evolution

Il s'agit d'une classe regroupant toutes les méthodes de croisement, de sélection , de mutation et d'adaptation, il est très simple d'en ajouter une nouvelle sans utiliser notre application, si vous souhaitez en ajouter une, il vous faudra tout d'abord l'ajouter dans l'interface graphique et l'ajouter dans la liste des méthodes de votre choix. Les listes se trouvent dans le fichier parameterWindow.

RWS

Cette méthode nommée “proportional roulette wheel selection” ou “roulette” en français, va donner une plus forte probabilité d’être sélectionnés aux individus ayant un meilleur score, cela permet donc de donner plus de chances à un individu qui semble le mieux résoudre le problème de départ tout en laissant quand même une chance à d’autres individus d’être choisis.

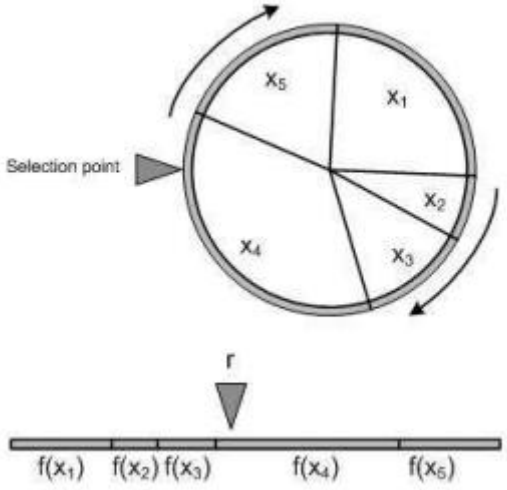


Image représentant un SUS, tiré de : http://www.lendek.net/teaching/opt_en/ga.pdf

Ce principe a été implémenté ainsi ,nous récupérons la somme totale des scores fitness de la population, ensuite pour chaque élément de la population nous allons soustraire sa valeur fitness à celle du fitness total. L’élément sélectionné sera celui qui rendra le fitness total négatif. Nous effectuons ces opérations tant que nous n’avons pas parcouru toute la population

SUS

Le “stochastic universal sampling” peut-être représenté comme un RWS, donc une roue possédant des segments représentatifs du fitness de l’élément de la population. Afin de sélectionner le chromosome, nous allons placer *c* pointeurs , *c* étant le nombre de chromosomes uniformément placés autour de la roue et faire tourner la roue, le chromosome ayant le plus de pointeur sur lui sera sélectionné

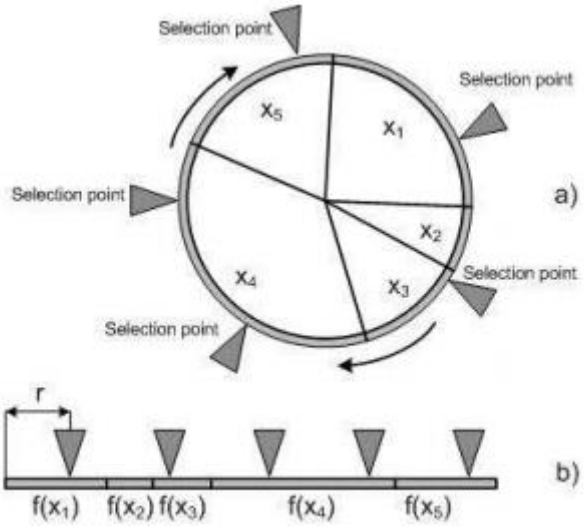


Image représentant un SUS, tiré de : http://www.lendek.net/teaching/opt_en/ga.pdf

Cette méthode est implémentée ainsi dans GALGO2 , nous récupérons la somme totale des fitness de la population ainsi que sa taille et nous les divisons afin d'obtenir la distribution entre les pointeurs et nous plaçons le pointeur dans la région comprise entre 0 et la distribution et nous effectuons les mêmes opérations que pour la RWS c'est à dire réduire le fitsum total tant que fitsum ne devient pas négatif. Une fois ceci atteint il récupère cette valeur et va incrémenter le pointeur de la même distance que la distribution calculée précédemment et ceci sera effectué *n* fois, *n* étant le nombre d'individus.

Sélection

RNK

RKN, aussi connue sous le nom de sélection par rang, nous ne nous basons plus cette fois sur le fitness à proprement parler mais nous nous baserons sur le rang que va obtenir l'individu, qui est un classement du fitness. En général nous donnons le rang N à l'individu possédant le meilleur fitness et 1 au pire.

Dans l'implémentation nous suivons ce schéma, nous remplissons un vecteur de rang de N à 1 et nous additionnons tous les rangs dans une variable ranksum et on procède de la même façon que pour les méthodes RWS et SUS en générant pour chaque élément de la population un nombre aléatoire compris entre 1 et ranksum et nous soustrayons le rang de l'individu à celui du rang total et tant que le total est positif, nous sélectionnons celui qui rend total négatif.

RSP

Il s'agit du même principe que le rank-based , mais cette fois-ci nous faisons entrer en matière la Selective Pressure, qui est un indice permettant de favoriser les meilleures individus, elle calcule le rang ainsi

$$rang = \frac{2 * (SP - 1) * (popsize - i)}{popsize} + 2 - SP$$

Équation permettant de donner une valeur au rang

Les autres opérations sont strictement les mêmes

TNT

Dans le mode de sélection tournoi on va définir une taille T , T étant la taille du tournoi. Nous allons faire s'affronter des individus choisis aléatoirement. Le vainqueur passe à l'étape suivante. Nous réitérons jusqu'à ce qu'il ne reste plus qu'un groupe. Il est en pratique recommandé d'utiliser des tailles de tournoi plutôt petite afin de maintenir une diversité dans la population.

Nous tirons au hasard un ID compris dans la range de la population et nous le définissons comme le meilleur, nous allons réitérer T fois un tirage aléatoire afin de prendre un individu aléatoire et nous allons voir si sa valeur fitness est meilleure que celle de bestFitness, si c'est le cas, nous assignons cet individu en tant que bestFitness. Le vainqueur du tournoi sera choisi à la fin du parcours T. Nous allons réitérer ces actions le nombre de fois nécessaire afin de remplir la mate population.

TRS

Nous allons ici remplir un vecteur de nombres aléatoires compris entre [0;1[, puis les trier du plus grand au plus petit et nous allons transformer les valeurs de fitness de toute la population par rapport à une valeur c initialement à 0.2 (et sera incrémenté après chaque génération de 0,1), le nouveau fitness va prendre cette valeur

$$newFitness = \frac{popsize * e^{-c*z} - popsize}{1 - e^{-c}}$$

On effectuera les mêmes opérations que précédemment en sélectionnant celui qui rendra *fsum* négatif.

Croisement

Toutes ces méthodes prennent en paramètre une population ainsi que deux références sur des chromosomes, ces deux chromosomes sont initialisés lorsque le GA va effectuer sa méthode évolution et va arriver dans la méthode *recombination.*

P1XO

Le principe du One Point Crossover est de placer un point de croisement, à partir de ce point nous allons croiser les valeurs afin d'avoir une partie du chromosome 1 et une autre partie du chromosome 2.

Image représentant un P1XO, tiré de : https://www.researchgate.net/figure

Nous allons donc dans cette méthode sélectionner deux chromosomes de manière aléatoire dans notre population et choisir un nombre aléatoire compris entre 0 et N , N étant le nombre de bits composant notre chromosome.

Une fois ceci fait nous allons écraser les valeurs des chromosomes passées en paramètre par celle que nous avons récupérées précédemment. Nous allons stocker les deux premières parties des chromosomes choisis allant de [0;pos], entre autre nous stockerons dans *chr1*, *idx1* et dans *chr2*, *idx2* , chr étant les chromosomes passés en paramètre et idx les chromosomes tirés au hasard. Ensuite nous allons effectuer le croisement en stockant de]pos; N] *idx2* dans *chr1* et *idx1* dans *chr2*.

P2XO

Le principe du Two Point Crossover est le même que pour P1XO sauf que cette fois-ci nous plaçons deux points autour desquels nous ferons des croisements de séquences.

Image représentant un P2XO, tiré de : https://www.researchgate.net/figure

Le principe est exactement le même que pour P1XO, sauf que nous tirons deux positions au lieu de une seule. Une fois cela fait nous allons stocker la plus petite des deux valeurs dans une variable *m* et la plus grande dans *M*. Le croisement se fera ainsi, nous allons stocker dans chr1 la valeur de *idx1* allant de [0;m], *chr1* étant le chromosome passé en paramètre et *idx1* le chromosome tiré au hasard. On stocke ensuite dans *chr2* la séquence de *idx2* allant de [0;m]. Après cette opération nous effectuons un croisement et nous stockons dans]m;M] la séquence correspondante de *idx1* dans *chr2* et *idx2* dans *chr1*. Finalement nous stockerons de]M;N] *idx1* dans *chr1* et *idx2* dans *chr2*.

UXO

Le Uniform Crossover va lui effectuer un croisement de manière uniforme entre les deux chromosomes, on va parcourir bit à bit notre séquence et on va avoir une chance sur deux d'effectuer un croisement

Image représentant un UXO, tiré de : https://www.researchgate.net/figure

Dans l'implémentation nous sélectionnons deux chromosomes de manière aléatoire et nous allons parcourir toute la séquence en générant un nombre aléatoire pour chaque bits, si ce nombre est < 0.5 nous ne ferons pas de croisement et si il est plus grand, nous effectuons un croisement.

Mutation

Pour toutes les méthodes de mutation nous passons en paramètre un chromosome déjà existant, la mutation s'effectue après l'opération de croisement, on peut le voir dans la méthode *recombination* située dans la classe Population.

Les méthodes utilisent une la méthode std::uniform_int_distribution afin de générer un nombre aléatoire entre [0;MAXVAL], *MAXVAL* valant par défaut 1.

BDM

La méthode de Boundary Mutation va remplacer la valeur du chromosome par sa valeur supérieure ou inférieure.

Nous allons tout d'abord vérifier si le taux de mutation est plus grand que 0, auquel cas nous arrêtons la méthode. Si ce test est correct nous allons créer un chromosome arrondi à la valeur au-dessus et un chromosome arrondi à la valeur en dessous. Ensuite on va générer un nombre aléatoire compris entre 0 et 1, si il est plus petit que le taux de mutation on va effectuer une mutation, il s'agit ici de simuler la probabilité d'évoluer, ensuite nous allons avoir une chance sur deux, en tirant un nombre aléatoire compris entre 0 et 1, d'avoir la valeur arrondie en dessous si le nombre tiré est plus petit que 0.5 autrement, si ce nombre est plus grand ou égal à 0.5 nous allons muter le chromosome en prenant la valeur arrondie du dessus.

SPM

Le Single Point Mutation est une mutation qui va parcourir l'ensemble des caractères du chromosome et qui va selon le *mutrate* inverser le bit. Le bit à 1 deviendra un 0 et vice-versa

Nous parcourons toute la séquence, pour chaque caractère nous tirons un nombre aléatoire compris entre 0 et 1, si ce dernier est plus petit ou égal au *mutrate* alors on va inverser le bit avec la méthode *flipBit*.

UNM

L'Uniform Mutation va remplacer un chromosome par un autre ou remplacer une partie du chromosome actuel.

Nous allons parcourir chaques caractères de notre chromosome, pour chacun de ses caractères nous vérifions si nous pouvons effectuer une mutation, si c'est le cas nous allons récupérer le paramètre[k], *k* étant la k-ième itération. Ce paramètre sera encodé en string de bits et va remplacer le chromosome actuel par le string généré précédemment.

Élitisme

L'élitisme est le principe de la conservation des meilleurs individus pour la génération suivante. Nous effectuons cette opération après l'étape de croisement en comparant le meilleur individu de la nouvelle génération avec celui de la génération précédente.

Si l'ancien meilleur chromosome est meilleur, il remplacera celui qui est dans la nouvelle génération. GALGO2 permet lui de sélectionner *elitpop* meilleurs chromosomes et de les comparer aux meilleurs chromosomes de notre nouvelle population.

L'élitisme est un bon compromis permettant d'obtenir une amélioration vers notre fonction objective sans avoir trop de variations dans les résultats, par contre, son soucis majeur réside dans sa tendance à vite atteindre un maximum local et non un maximum global.

Parameter

La classe paramètre est une classe qui hérite d'une classe abstraite BaseParameter, elle définit les méthodes encode() , encode(T value) ,decode(string), size() et getData(). Ce sont les méthodes absolument nécessaires à Galgo afin de pouvoir fonctionner.

Nous instancions un paramètre avec un vecteur de données composés de deux éléments, le premier élément est la valeur la plus petite que peut prendre notre paramètre et le deuxième représente la valeur maximale qu'elle peut avoir. Dans notre cas nous avons utilisés la valeur 0 pour le plus petit nombre et nous avons utilisés la méthode numeric_limits pour la valeur maximale afin de pouvoir obtenir le spectre le plus large possible.

Les paramètres seront utilisés plus tard par l'algorithme génétique.

Genetic Algorithm

Cet classe a subi une transformation par rapport à la version originale, nous gérons le constructeur d'une manière différente car il ne convenait pas à nos besoins, voici donc ses quelques modifications.

Nous avons notamment changé le constructeur de l'objet GeneticAlgorithm qui prend dorénavant ces paramètres :

- Une référence sur un Emitter afin de lui permettre d'envoyer des signaux
- Une référence sur une structure de paramètres
- Une fonction Objective que l'on devra maximiser
- un boolean pour indiquer si on souhaite afficher l'output
- Une référence sur un vector de Parameter

La structure de paramètres est composé des toutes les variables qui permettent de définir quelle méthode de croisement est utilisée, la taille de la population, le taux de mutation, le taux de croisement, etc. La plupart de ces attributs obtiennent leurs valeurs en les choisissant dans la fenêtre de paramètres de l'interface graphique.

Au lieu d'appeler la méthode init, on l'implémente directement dans le constructeur afin d'éviter les appels récursifs que produisaient cette méthode. On va copier le vector de data du paramètre et attribuer ses valeurs à des variables lowerBound et upperBound.

Ensuite la méthode run() va être appelée, cette dernière va tout d'abord vérifier si les valeurs saisies sont bien correctes, notamment la range les attributs. Une fois ces tests effectués ,nous allons initialiser une population en lui donnant une référence sur l'algorithme génétique actuel.

Finalement nous allons effectuer une boucle longue du nombre de générations que l'on souhaite créer en faisant évoluer la population, à la fin de chaque générations nous émettons un signal afin de pouvoir faire avancer la barre de progression de notre interface graphique.

Population

Lorsque la population est initialisée par l'algorithme génétique va instancier ses paramètres en allouant de la place pour les populations.Il va également calculer une variable nommée *nbrcrov* qui est le nombre de fois qu'elle doit effectuer un croisement suivi d'une mutation.

Directement après dans la classe GA on va appeler la méthode creation(). C'est cette classe qui est en quelques sortes le chef des opérations et va appeler les méthodes d'évolution et gérer ses chromosomes.

Dans la méthode création nous allons instancier des chromosomes en leur donnant une valeur, puis nous évaluerons ce string, et nous faisons ça jusqu'à ce que la population soit remplie.

La méthode d'évolution se fait de cette façon, tout d'abord nous appliquons la méthode de sélection que nous souhaitons, puis nous appliquons l'élitisme dans le cas où l'utilisateur souhaite utiliser ce principe , ensuite on va faire un croisement selon la méthode choisie sur deux chromosomes de la nouvelle population que nous créons, nous allons ensuite faire muter deux chromosomes, on réitérera les opérations de croisement et de mutation *nbrcrov* fois.

S'il manque des chromosomes à la fin de cette boucle on va appeler la méthode de complétion en la remplissant de chromosomes de la mating population choisis au hasard et les faire muter.Finalement on va définir cette nouvelle population comme étant la courante.

Les chromosomes sont toujours triés par leur fitness, on les classera du meilleur au pire au sein de la population.

Les évaluations des chromosomes sont effectuées ainsi:

1. On décode la valeur du chromosome
2. On le passe dans la fonction objective pour voir son résultat
3. Le résultat devient le fitness

Chromosome

Cette classe est donc la représentation d'un chromosome, elles sont stockées sous forme de make_shared. A son instanciation le chromosome va récupérer un pointeur sur l'algorithme génétique ainsi que la taille d'un chromosome et le numéro de génération actuel.

Le chromosome ne prendra sa valeur que lorsqu'il sera appelé par la méthode create() -> dans le cas où il n'a pas de valeurs de paramètres, on tirera un nombre tiré uniformément au hasard compris en lower et upper. S'il en possède une il appellera la méthode initialize et va encoder sa valeur.

Cette classe contient toutes les méthodes permettant d'interagir avec les séquences de bits.

Multithreading

Il est possible d'utiliser le multithreading dans GALGO2. Comme vaguement dit plus haut, la solution utilisée est OpenMP(Open Multi-Processing). Cette API est prise en charge sur Linux, OS X et Windows.

La librairie <opm.h> est implémentée dans le fichier *galgo.hpp.* Il y a une variable static permettant d'obtenir le nombres de threads disponibles.

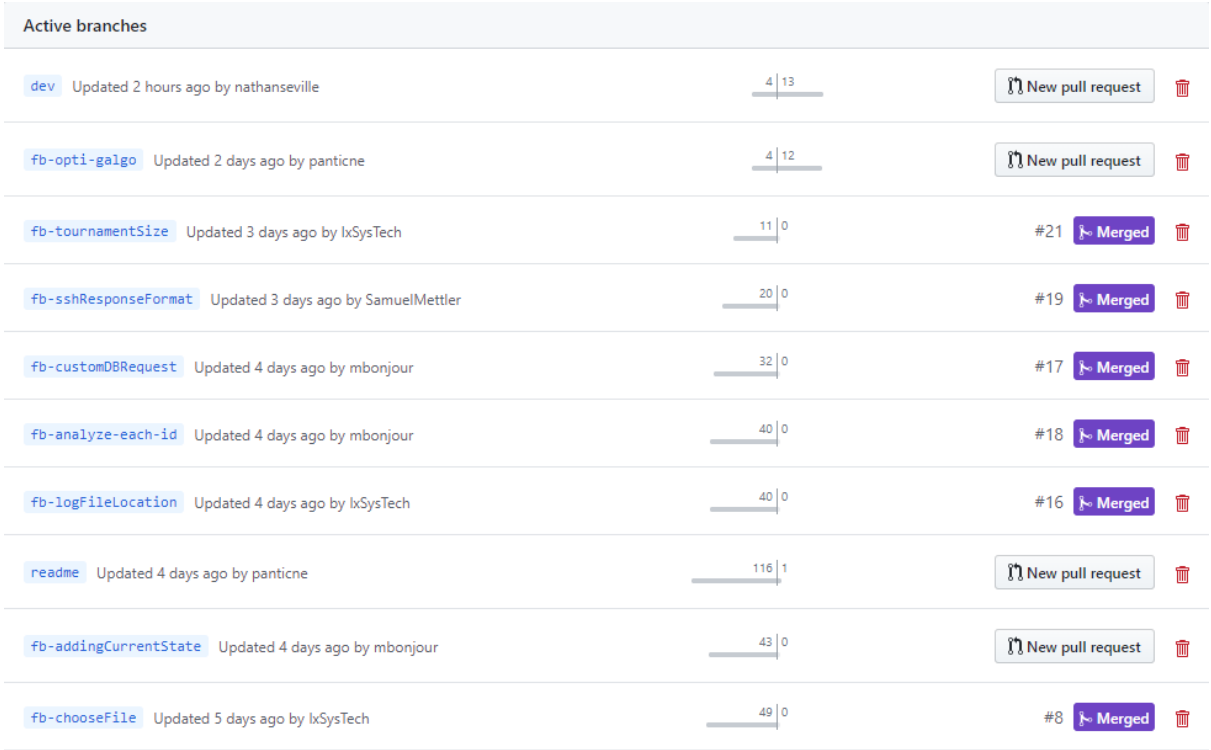
Comme il s'agit d'une technologie que nous ne connaissons pas tout à fait, nous n'avons pas réussis à correctement implémenter le multi-threading car en début de projet nous ne savions pas vraiment faire de programmation concurrente et que nous avions décidés d'aller vers le plus simple dans notre implémentation et de progressivement complexifier notre programme. Mais l'implémentation du multi-threading serait une tâche que nous pourrions creuser pour améliorer la performance de notre programme.

A noter que en dehors de Galgo, l'implémentation a été faites de façon à pouvoir être multi-threadable.

Méthode de développement

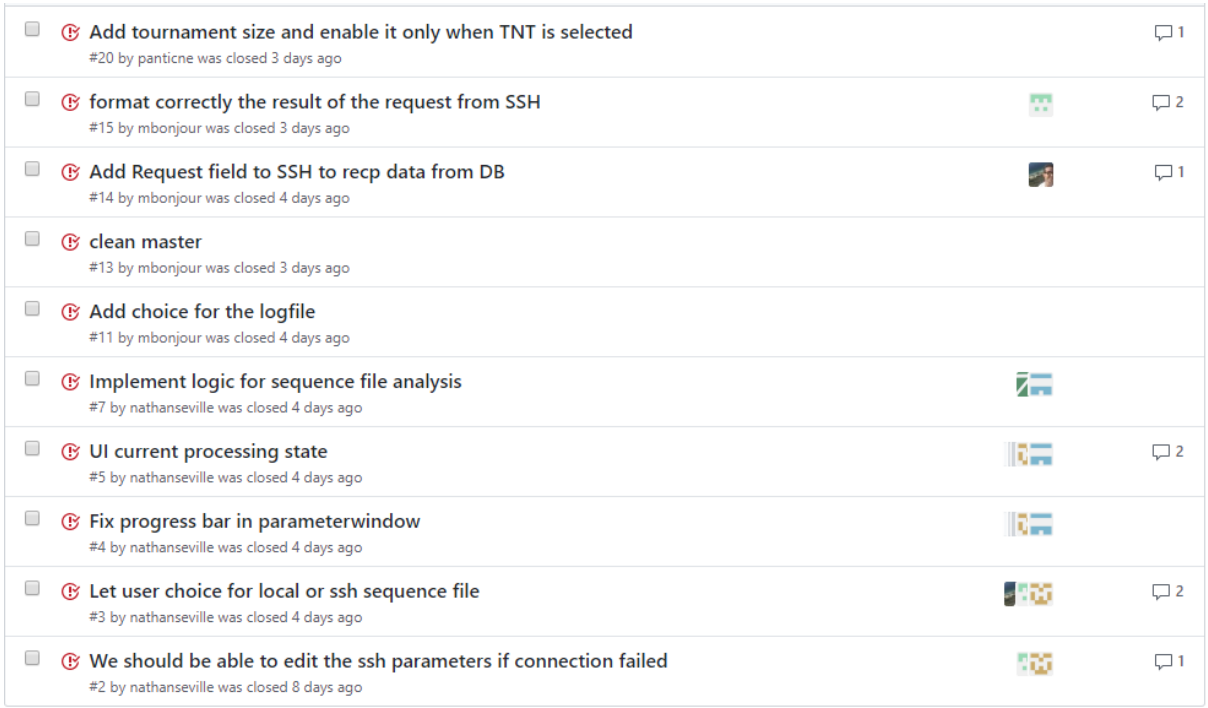
Afin de pouvoir travailler dans de bonnes conditions nous avons utilisés un système de gestion de développement de logiciels.

Notre choix a dû se porter sur GitHub. Nous avons créé différentes branches afin de pouvoir travailler en même temps sur le projet en évitant un maximum les conflits de fichiers.



Nous avons très souvent utilisés des feature branch afin de pouvoir ajouter des fonctionnalités et de pouvoir, en cas de problèmes, avoir la possibilité de revenir à un moment où la version de l'application était stable.

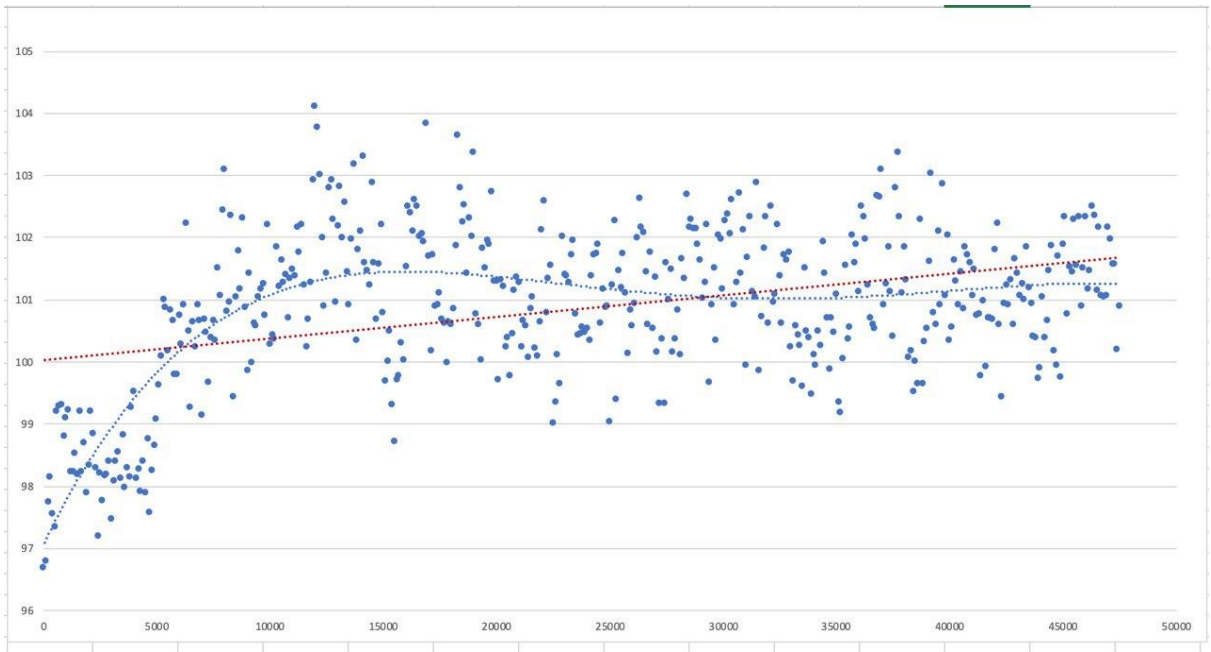
Sur la fin du projet nous nous sommes mis à créer des Issues afin que tout le monde puisse avoir un aperçu de ce qu'il restait à faire



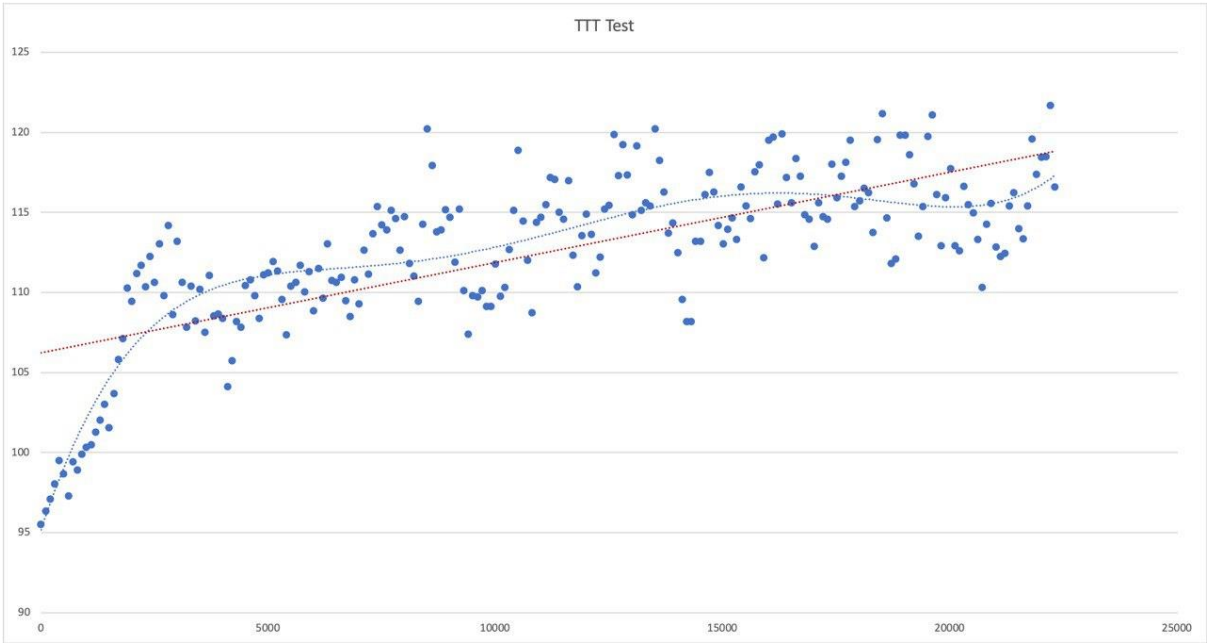
Cela nous a permis de mieux nous organiser dans la répartition de tâches, jusque là nous nous contentions de communiquer sur notre chat de groupe de travail.

Résultat & Statistiques

Afin de tester notre programme nous nous sommes basé sur l'opinion de notre client. Pour informations, chaque point représente la moyenne des scores obtenus dans une génération. Une de nos premières génération de machine nous a amené au résultat présenté ci-dessous:

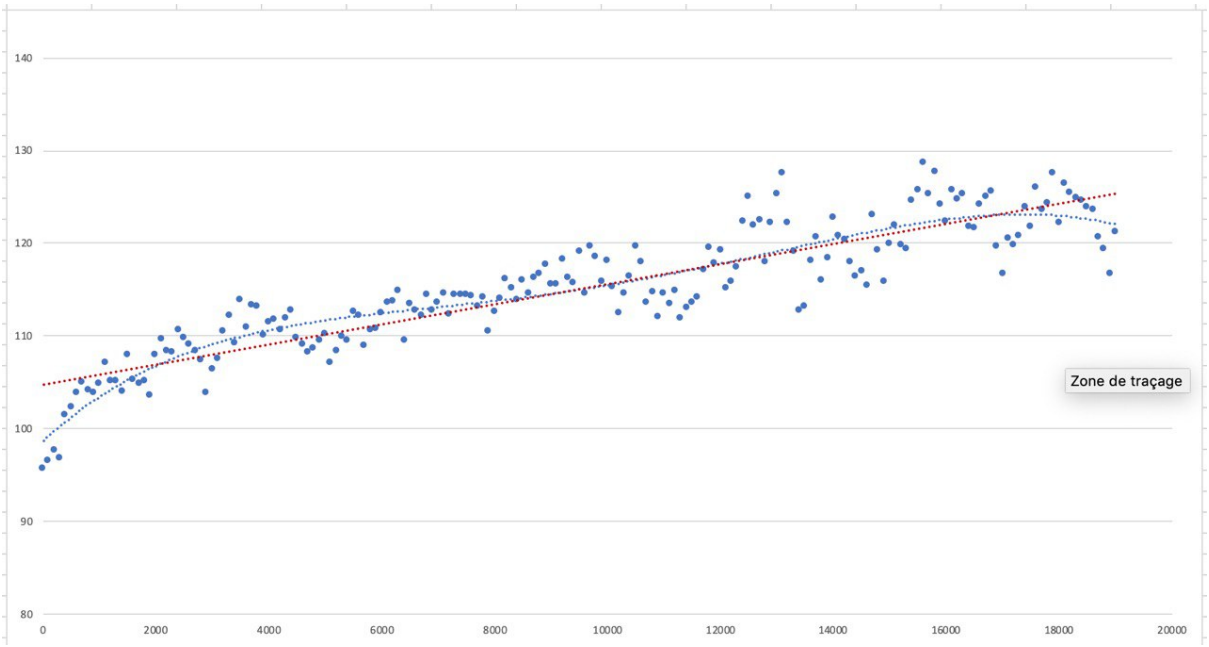


Nous remarquons que les points sont généralement très éparpillés, ce qui n'est pas bon, nous souhaiterions plutôt avoir une évolution plus linéaire avec une surface de points moins importante. Pour trouver une solution nous nous sommes renseignés auprès du client qui nous a expliqué l'importance d'avoir une faible mutation et un croisement plus important, cela nous a permis d'obtenir le graphique ci-dessous:



Cette fois-ci, nous pouvons constater que les points sont moins éparpillés et suivent une plus forte tendance à monter, ce qui se rapproche du comportement d'un algorithme génétique.

Pour finir, en augmentant la valeur de l'élitisme à 5, nous obtenons un graphique comme celui ci-dessous, nous remarquons qu'il est beaucoup plus régulier et se rapproche fortement de la tendance. En discutant avec le client, nous avons eu la confirmation que l'analyse se déroulait correctement et qu'il s'agissait bien du comportement souhaité.



Il est important de savoir que pour des mêmes configurations d'algorithme génétique, on obtiendra pas toujours le même graphique, car les chromosomes sont générés aléatoirement et qu'un facteur de croisement et de mutation s'appliquent sur ces derniers ce qui augmente l'aspect imprévisible.

Améliorations possibles

Interface graphique

Pour ce qui est de la compatibilité de notre application il est possible de la rendre plus générique. En effet, pour l'instant, elle n'est compatible qu'avec une base de données PostgreSQL. Un nouveau bouton dans l'interface graphique pourrait être ajouté afin de choisir le type de base de données.

Ensuite pour ce qui est du choix des données à récupérer de la base de données, il est possible d'implémenter un code qui récupérerait toutes les tables de la base de données avec leurs champs et les afficher à l'utilisateur. L'utilisateur pourra ensuite cocher les champs qu'il voudrait récupérer de la base de données. Une fois les choix effectués, le code pourrait générer une requête qui sera envoyé au serveur. Cela éviterai à l'utilisateur d'écrire lui-même la requête.

EmitterACGT

On pourrait apporter une amélioration à l'émetteur ACGT, en effet pour le moment on lit **toute la séquence** dans tous les cas. Et dans le cas où l'on prend les séquences disponibles sur le serveur via notre connexion SSH notre programme est **extrêmement lent**. En effet le fait de vouloir analyser des séquences aussi longues est très lent. On pourrait donc implémenter un **système d'arrêt** afin de passer à la prochaine séquence si toutes les machines de *MegaMachineManager* sont stoppées. Cela permettrait de ne pas analyser toutes les séquences en entier s'il y en a pas besoin. Les maxAlert permettent déjà d'arrêter les machines d'états mais l'emitter continuera d'émettre jusqu'à la fin de la séquence.

MegaMachineManager - MultiThread - Plusieurs MegaMachines

Pour le moment notre implémentation n'est pas optimale. Nous avons eu des soucis de communications dans le groupe par rapport à la recherche de l'algorithme génétique. En effet l'idéal aurait été de pouvoir passer notre population entière dans le MegaMachineManager (dans la fonction objective() de l'algo génétique) afin que la MegaMachineManager délivre son potentiel. Nous l'avons créé et designé pour cela cependant le GA n'étant pas fait pour ça cette fonctionnalité est caduc. Cela nous éviterait de relire toutes les séquences pour chaque individu. Notre MegaMachineManager peut gérer autant de machines que l'on veut et ainsi lire une fois les séquences pour toute la population.

Création paramètres GA dynamiques

Faire en sorte que l'application choisisse entre float ou double pour la génération des paramètres du GA.

Analyses séquences protéiques

Dans notre projet nous avons commencé par analyser des séquences nucléiques et cela est une erreur, malheureusement il était trop tard quand on l'a constaté (en effet les séquences protéiques sont 3x plus courtes et sont donc plus intéressantes). Heureusement pour mettre à disposition cette fonctionnalité cela ne devrait pas être trop complexe, en effet on peut "simplement" étendre les StateDescriptor afin d'accepter les acides aminés, ensuite il faudrait faire un émetteur pour les séquences protéiques et enfin pouvoir différencier les 2 dans le Dispatcher afin de lancer le bon type dans MegaMachineManager suivant les séquences. Enfin, il faudrait modifier la méthode getMachine ou en créer une pour les machines protéiques.

Mise en place d'une machine de départ

Proposer à l'utilisateur de redémarrer avec une certaine machine d'états donnée en paramètres.

Tests effectués

Analyse de mémoire valgrind intégré à QT Creator. Ceci nous a permis de régler beaucoup de fuites mémoires mais malheureusement pas toutes.

Lors de l'implémentation des différentes parties (MegaMachine & GA) on a effectué des tests au long de l'intégration (branche realteststatemachines sur github) pour valider nos implémentations respectives.

Erreurs et bugs connus

Des fuites mémoires ont été constatées, cependant on ne les a pas résolues car elles sont assez compliquées à gérer. L'analyseur utilisé n'indiquait pas correctement les zones mémoires touchées et l'analyse ayant été faite trop tardivement nous n'avons pas pu régler ces fuites.

L'analyse d'un fichier download est trop lente, ceci est dû à une mauvaise implémentation de l'émetteur ACGT comme cité plus haut.

Conclusion

Conclusion personnel

Mickael Bonjour

Ce projet a été vraiment très enrichissant, en effet j'ai pu apprendre à gérer l'event loop de Qt et le framework en général. Je suis content du travail effectué avec mon groupe et satisfait du résultat malgré le manque de temps pour l'amélioration des fonctionnalités présentes. Ce manque de temps est dû à la phase d'apprentissage un peu plus longue que prévue, mais cela n'a pas été vain. Dans l'ensemble on a atteint les objectifs que l'on s'était fixés dans le cahier des charges. J'aurais voulu en faire plus afin d'avoir un projet un peu plus fourni. Mon groupe a été impliqué au long du projet et c'était agréable de travailler avec eux. On a sous-estimer le temps qu'il nous fallait pour le merge des parties du projet et on a été pressés par le temps pour implémenter et corriger certaines erreurs vues sur la fin.

Samuel Mettler

Ce projet, bien que d'apparence difficilement abordable, m'a permis d'apprendre plein de nouveaux concepts qui m'étaient totalement étranger. En effet, le côté biologique sur lequel le projet est appliqué est particulièrement intéressant. De même pour la partie évolution : j'en ai souvent entendu parlé mais jamais concrètement donc je suis content d'avoir pu travailler avec dans le cadre de ce projet.
Il m'a également permis d'utiliser de nouveaux outils de développements et de mettre en place des structures ou éléments que je n'aurais jamais fait par moi-même.

Nathan Séville

Ce projet m'a permis de prendre conscience des avantages et difficultés que présente un travail de groupe sur un projet de moyenne envergure. De mon point de vue le projet c'est dans l'ensemble bien déroulé, j'en ai beaucoup appris tant au niveau technique que gestion de projet. Les différentes technologies utilisées tel que les machines d'états en c++, les algorithmes génétiques ainsi que l'UI Qt m'ont permis d'acquérir de nouvelles connaissances. L'ambiance au sein du groupe était très appréciable et à permis un bon développement du projet.

Nemanja Pantic

Pour ma part, j'ai trouvé le projet intéressant à mettre en place, il m'a permis de me rendre compte des différentes spécificités auxquelles nous avons affaire dans un travail de groupe. J'ai trouvé que nous étions assez bien soutenu par le client qui nous a permis de nous remettre sur les rails lorsque nous avions besoin de compléments d'informations. J'ai apprécié l'aspect que j'ai travaillé dans ce projet, les algorithmes génétiques sont intéressants et j'ai eu du plaisir à apprendre le fonctionnement de ces derniers.

David Simeonovic

En conclusion, j'ai trouvé ce projet très intéressant. Il nous a permis d'acquérir de nouvelles connaissances et découvrir de nouvelles technologies telles que LibSSH, Qt, Galgo. Il nous a appris à travailler avec un groupe relativement grand. Nous avons appris à bien répartir les tâches entre chaque membre du groupe.

Conclusion de groupe

Ce projet semestriel nous a opposé à une grande liste de challenges. Non seulement nous étions confrontés face à quelque chose que l'on ne connaissait pas (implémentation de machine d'état,

algorithme évolutionniste, connaissance biologiques), mais également nous devions mettre en place une gestion de travail propre et efficace.

Au niveau de l'organisation, la répartition des tâches ne nous a pas posé de problème particulier, tout le monde avait toujours quelques chose à faire. La communication entre les membres du groupes était bonne vu que l'on se connaît bien et qu'on a l'habitude de travailler ensemble.

L'application en elle-même à atteint son objectif qui était de faire évoluer des machines d'état et les évoluer ainsi que de parcourir des séquences. Nous sommes aussi satisfait du temps que prend notre application pour faire évoluer un certain nombre de génération. Cependant, l'application n'est pas utilisable sur des sets de données "libres" dans le sens où les séquences génétiques peuvent atteindre un nombre de caractères beaucoup trop grand. Nous avons donc atteint un de nos objectifs mais ayant commencé les tests réels tardivement, nous n'avons pas pu adapter sur le moment. Cela fait parti des améliorations possibles dans ce projet.

Sur le plan individuel, nous nous sommes tous impliqués dans ce projet et avons tous acquis de nouvelles connaissances grâce à des phases d'apprentissages et de recherches sur toutes les librairies qui pouvaient nous être utile lors du développement de notre application.

Pour conclure, malgré toutes les difficultées que l'on a rencontré nous avons pris du plaisir à réaliser ce projet.

Sources

GA

http://igm.univ-mlv.fr/~dr/XPOSE2013/tleroux_genetic_algorithm/fonctionnement.html

https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique

https://github.com/olmallet81/GALGO-2.0

https://www.researchgate.net/profile/Murat_Yildizoglu/publication/5085159_Presentation_des_algorithmes_genetiques_et_de_leurs_applications_en_economie/links/0912f505ca0731d7a5000000/Presentation-des-algorithmes-genetiques-et-de-leurs-applications-en-economie.pdf

http://www.lendek.net/teaching/opt_en/ga.pdf

http://wpmedia.wolfram.com/uploads/sites/13/2018/02/09-3-2.pdf

https://www.researchgate.net/post/What_is_meant_by_the_term_Elitism_in_the_Genetic_Algorithm

http://www8.umoncton.ca/umcm-cormier_gabriel/SystemesIntelligents/AG.pdf

LibSSH

https://www.libssh.org/

https://www.libssh.org/get-it/

http://api.libssh.org/stable/libssh_tutor_scp.html

http://api.libssh.org/stable/libssh_tutor_guided_tour.html

http://api.libssh.org/stable/libssh_tutor_shell.html

Diagrammes

https://online.visual-paradigm.com

Évolution d'automates pour le traitement de séquences génomiques

Documentation développeur

Équipe **B-13** :

Bonjour Mickael (mickael.bonjour@heig-vd.ch)

Mettler Samuel (samuel.mettler@heig-vd.ch)

Pantic Nemanja (nemanja.pantic@heig-vd.ch)

Seville Nathan (nathan.seville@heig-vd.ch)

Simeonovic David(david.simeonovic@heig-vd.ch)

Table des matières

Introduction

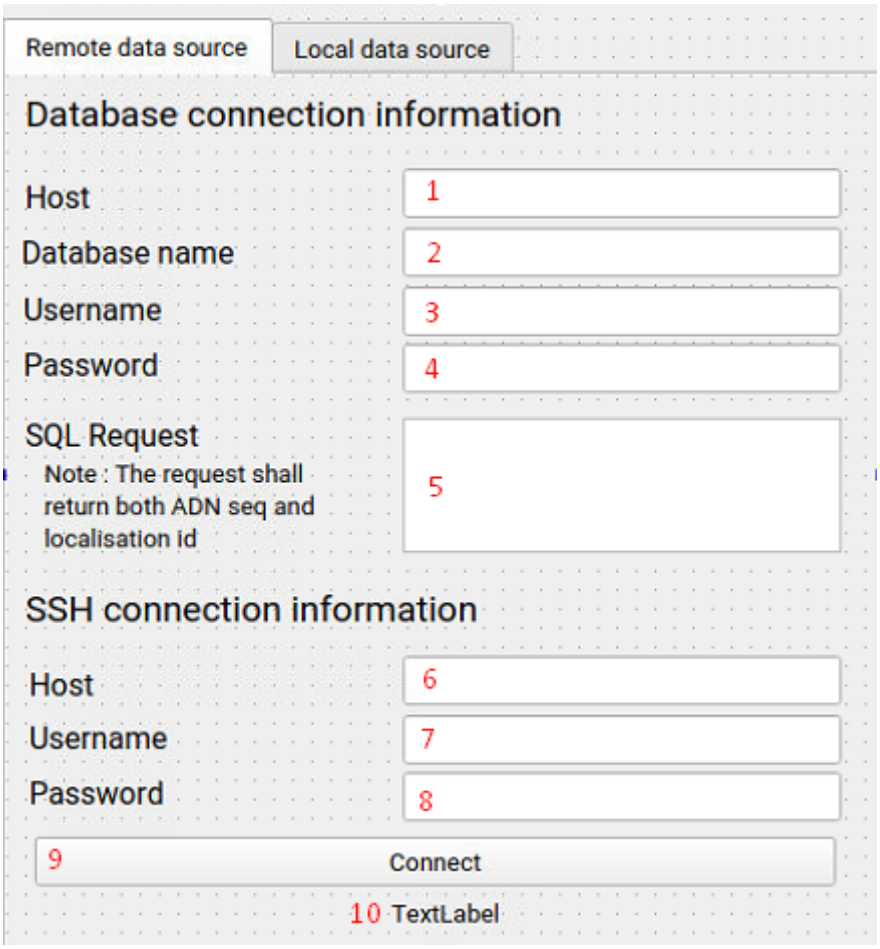
Ce document a pour but d'informer un développeur sur les choix d'implémentations que nous avons fait lors de la création de notre application

Interface graphique

Partie graphique

Toute la partie graphique de notre application a été construite grâce à l'interface que QT Deisgner met à disposition.

MainWindow

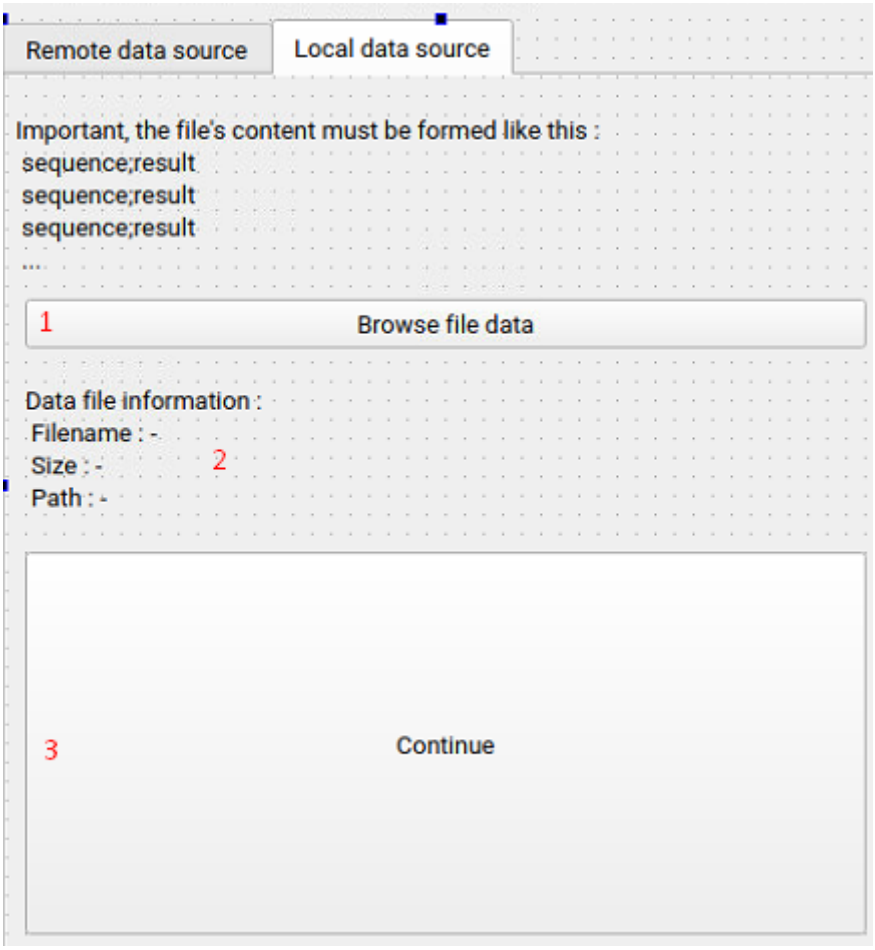


Listes des éléments avec leur fonctions :

1. txtDBHost : hôte de la base de données
2. txtDBName : nom de la base de données
3. txtDBUsername : nom d'utilisateur pour la connexion à la base de données
4. txtDBPassword : mot de passe pour la connexion à la base de données
5. sqlRequest : requête qui sera envoyé à la base de données
6. txtSSHHost : hôte pour la connexion SSH

7. txtSSHUsername : nom de l'utilisateur pour la connexion SSH
8. txtSSHPassword : mot de passe pour la connexion SSH
9. btnConnect : établi la connexion SSH ainsi que la connexion à la base de données et récupère les données
10. lblInfo : informe l'utilisateur de l'avancement du programme et indique s'il y a eu des erreurs lors du processus

Il y a une barre de progression (pgbDownload) qui est cachée. Elle se trouve au même endroit que le bouton "Connect". Elle informe l'utilisateur sur l'avancement de la récupération des données depuis la base de données. Elle s'affiche lorsque la connexion a été initiée et que l'on reçoit des données



Liste des éléments avec leur fonctions

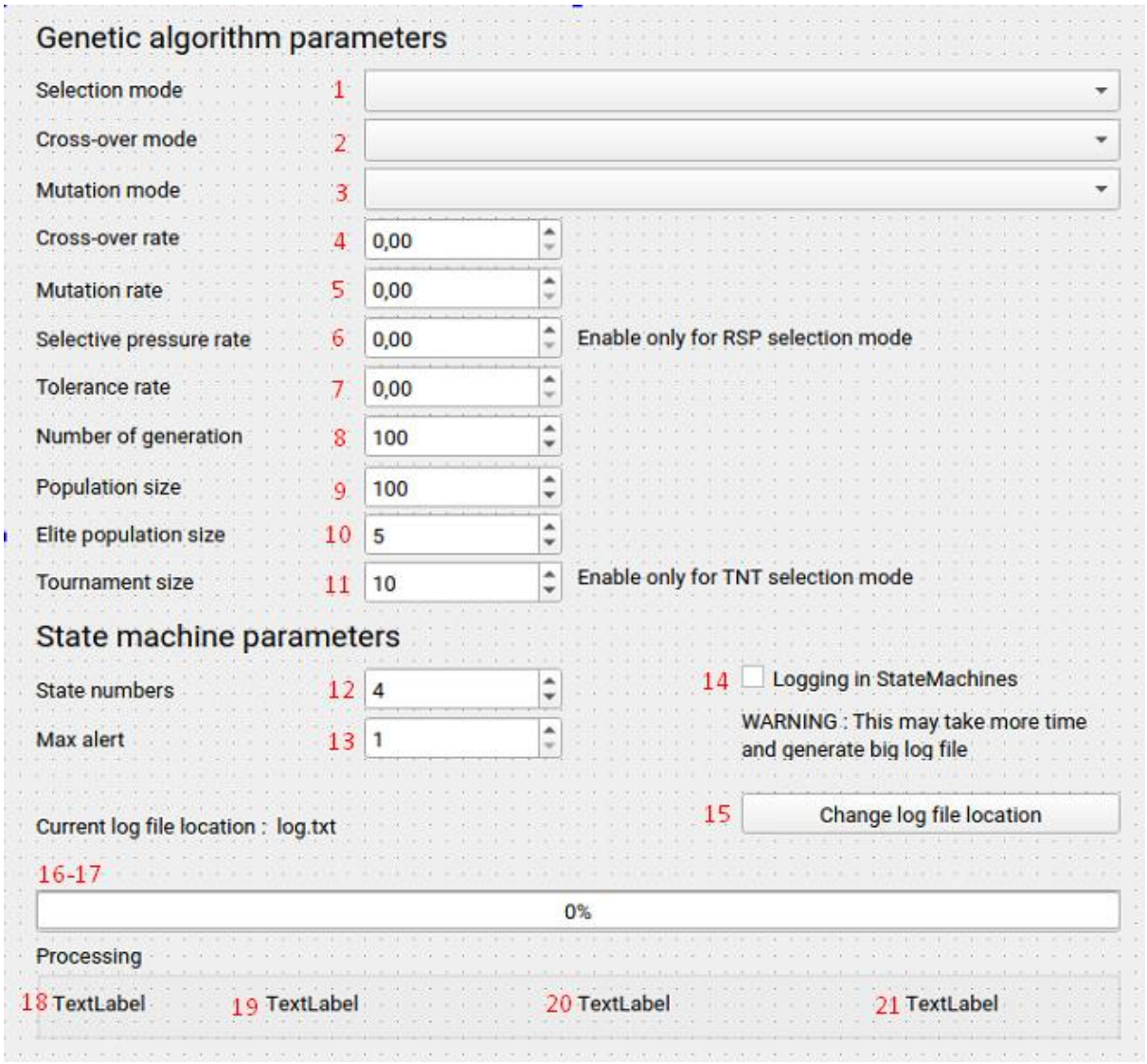
1. btnBrowse : Ouvre un explorateur de fichier pour permettre à l'utilisateur de sélectionner le fichier de donnée de son choix
2. lblFileInfo : informe l'utilisateur sur le fichier qu'il a sélectionné
    a. Nom du fichier

b. Taille en bytes du fichier

c. Chemin du fichier

3. btnContinue : Permet de cacher cette fenêtre et d'afficher la fenêtre ParameterWindow

ParameterWindow



1. cmbSelectionMode : permet de choisir le mode de sélection :

a. Proportional roulette wheel selection (RWS)

b. Stochastic universal sampling (SUS)

c. Classic linear rank-based selection (RNK)

d. Linear rank-based selection with selective pressure (RSP)

e. Tournament selection (TNT)

f. Transform ranking selection (TRS)

2. cmbCrossOverMode : permet de choisir le mode de croisement :
    a. One point cross-over (P1XO)
    b. Two point cross-over (P2XO)
    c. Uniform cross-over (UXO)
3. cmbMutationMode : permet de choisir le mode mutation :
    a. Boundary mutation (BDM)
    b. Single point mutation (SPM)
    c. Uniform mutation (UNM)
4. dsbCrossOverRate : permet de choisir le taux de croisement (double)
    a. Minium : 0
    b. Maximum : 1
    c. Par défaut : 0.5
5. dsbMutationRate : permet de choisir le taux de mutation (double)
    a. Minium : 0
    b. Maximum : 1
    c. Par défaut : 0.5
6. dsbSpRate : permet de choisir le taux de la pression sélective (double)
    a. Uniquement disponible pour le mode de sélection RSP
    b. Minium : 1
    c. Maximum : 2
    d. Par défaut : 1.5
7. dsbToleranceRate : permet de choisir le taux de tolérance (double)
    a. Par défaut 0 (inactif)
8. sbGenerationNumber : permet de choisir le nombre de générations (int)
    a. Minium : aucun
    b. Maximum : aucun
    c. Par défaut : aucun
9. sbPopulationSize : permet de choisir la taille de la population (int)

    a. Minium : aucun
    b. Maximum : aucun
    c. Par défaut : aucun

10. sbElitePopulationSize : permet de choisir la taille de la population d'élite (int)
    a. Minium : aucun
    b. Maximum : aucun
    c. Par défaut : 1
11. sbTournamentSize : permet de choisir la taille du tournoi (int)
    a. **Uniquement disponible pour le mode de sélection TNT**
    b. Minium : aucun
    c. Maximum : aucun
    d. Par défaut : 5
12. sbStateNumbers : permet de choisir le nombre d'états des machines d'états (int)
    a. Minium : aucun
    b. Maximum : aucun
    c. Par défaut : aucun
13. sbMaxAlert : permet de d'indiquer au machine d'état à combien d'alerte elles devraient s'arrêter
    a. Minium : aucun
    b. Maximum : aucun
    c. Par défaut : aucun
14. cbLogMachines :
15. browseLog : permet à l'utilisateur de choisir où stocker le fichier de log
16. pgbGeneration : barre de progression qui indique l'état de l'analyse des séquences
17. btnRun : permet de lancer l'analyse des séquences
18. CurrentGen : indique la génération en cours d'analyse
19. MaxFitness : indique les meilleur score obtenu au moment de l'analyse
20. MeanFitness : moyenne des scores de la génération courante
21. CurrentAnalysis : indique le numéro de l'analyse en cours

Partie code

MainWindow

Pour la connexion SSH, nous avons choisis d'utiliser la librairie LibSSH. Celle-ci permet l'implémentation d'une connexion SSH entre l'application et l'hôte distant.

Afin de pouvoir utiliser LibSSH, il vous suffit de suivre le tutoriel sur ce site :

- https://www.libssh.org/get-it/

sshConnect

Cette fonction prend en paramètre un pointeur de l'objet "ssh_session". Elle va créer une nouvelle session ssh lors de l'appelle à "ssh_new()". Elle se charge ensuite de récupérer les données utilisateurs concernant la connexion SSH pour les fournir à l'objet :

- ssh_option_set → fournit l'information sur le nom de l'hôte à l'objet
- ssh_userauth_password → fournit le nom d'utilisateur et le mot de passe

Si l'initialisation de la session SSH échoue, la fonction retourne la valeur "SSH_ERROR", si tout se passe bien elle retourne "SSH_OK". Cette valeur est une valeur entière.

channelConnect

Cette fonction prend en paramètre un objet "ssh_session" et un objet "ssh_channel". Elle permet de créer un canal qui donne un accès directe au shell de la machine distante. L'appel à "ssh_channel_new" permet de créer le canal puis l'appel à "ssh_channel_open_session" permet d'ouvrir ce canal pour qu'il puisse être utilisé.

Si l'initialisation du canal SSH échoue, la fonction retourne la valeur "SSH_ERROR", si tout se passe bien elle retourne "SSH_OK". Cette valeur est une valeur entière.

sshWrite

Cette fonction prend en paramètre un objet "ssh_channel" et un pointeur de "char". Elle permet d'envoyer la commande passé en paramètre au shell de la machine distante.

Si l'envoie de la commande échoue, la fonction retourne la valeur "SSH_ERROR", si tout se passe bien elle retourne "SSH_OK". Cette valeur est une valeur entière.

scpRead

Cette fonction prend en paramètre un objet “ssh_session” et un “QString”. Elle permet la copie d’un fichier se trouvant sur la machine distante vers la machine local. Elle va créer un objet “ssh_scp” qui servira pour la copie du fichier. Avec l’appel à “ssh_scp_new” nous allons initialiser cette objet. Nous allons aussi instancier un objet de type “FILE” ce sera le fichier qui contiendra celui copié.

Nous allons d’abord initialiser la copie avec l’appel à “ssh_scp_init”. Nous allons ensuite envoyer une pull request à l’hôte distant avec “ssh_scp_pull_request”. Nous allons ensuite récupérer certaines informations sur le fichier qui sera copié en local :

- “ssh_scp_request_get_size” donne la taille du fichier
- “ssh_scp_request_get_filename” donne le nom du fichier
- “ssh_scp_request_get_permission” donne les droits du fichier

La pull request est ensuite acceptée avec “ssh_scp_accept_request”.Nous allons ensuite allouer de la mémoire pour le fichier récupéré avec un appel à “malloc”. Une boucle “while” va ensuite s’occuper de récupérer le fichier par bloc. Une fois le fichier récupéré, nous allons faire un dernier appel à “ssh_scp_pull_request” pour vérifier que toutes l’opérations se sont passées avec succès.

Si la copie du fichiers échoue, la fonction retourne la valeur “SSH_ERROR”, si tout se passe bien elle retourne “SSH_OK”. Cette valeur est une valeur entière.

setGUIEnabled

Cette fonction prend en paramètre une valeur booléenne. Elle active ou désactive simplement les objets de la fenêtre :

- False → désactive
- True → active

on_btnConnect_clicked

Cette fonction correspond à l’action du bouton “btnConnect”. Cette fonction utilise toutes les fonctions citées précédemment. Au moment du clic sur ce bouton, il va appeler la fonction “ssh_connect” pour initialiser la connexion SSH. Elle va ensuite appeler la fonction “channel_connect” pour initialiser le canal qui donne accès au shell de la machine distante. Une fois la connexion établie, la fonction va se charger de formater la commande à envoyer en y ajoutant la requête SQL de l’utilisateur. Finalement elle va appeler “scp_read” pour récupérer le fichiers distants. Ce fichier contient le résultat de la requête SQL. S’il y a une erreur lors de l’appel de ces fonctions, il affichera un message d’erreur à l’utilisateur. Une fois toutes ces opérations effectuées, la fonction cachera cette fenêtre et affichera la fenêtre “parameterWindow” à l’’utilisateur.

on_btnBrowse_clicked

Cette fonction correspond à l'action du bouton "btnBrowse". Elle utilise l'objet "QFileDialog" qui permet d'ouvrir un explorateur de fichiers pour que l'utilisateur puisse choisir un fichier de données à fournir à l'application. Une fois le fichier sélectionné, nous afficherons à l'utilisateur certaines informations à propos du fichier. Ces informations sont affichées dans un label "lblFileInfo". Une valeur booléenne est mise à true lorsqu'un fichier est sélectionné.

on_btnContinue_clicked

Cette fonction correspond à l'action du bouton "btnContinue". La fonction vérifie simplement qu'un fichier a été sélectionné. Si c'est le cas, elle affichera la fenêtre "ParameterWindow" et cachera la fenêtre "MainWindow"

ParameterWindow

setDataSource

Cette fonction prend en paramètre un "QString". Elle permet d'indiquer la localisation du fichier de données.

currentState

Cette fonction prend trois paramètres :

- **unsigned int** gen
- **double** maxFit
- **double** currentMean

Elle permet d'afficher des informations dans les labels suivants :

- N° 18 : numéro de la génération en cours
- N° 19 : meilleur score obtenu au moment de l'analyse
- N° 20 : moyenne des scores de la génération courante

nextAnalysis

Cette fonction prend 2 paramètres :

- **unsigned int** current
- **unsigned int** total

Permet d'afficher le nombre total d'analyse ainsi que le numéro de l'analyse en cours dans le label N° 21

setGUIEnabled

Cette fonction prend en paramètre une valeur booléenne. Elle active ou désactive simplement les objets de la fenêtre :

- False → désactive

- True → active

incrementProgressBar

Cette fonction prend en paramètre une valeur double. Elle permet d'incrémenter la valeur de la barre de progression.

on_cmbSelectionMode_currentIndexChanged

Cette fonction correspond à l'action de changer de choix pour le mode de sélection. Permet de bloquer certaines options selon les choix de l'utilisateur :

- Le taux de pression sélective est disponible seulement pour le mode RSP
- La taille du tournoi est disponible seulement pour le mode TNT

in_browseLog_clicked

Cette fonction correspond à l'action du bouton "browseLog" qui affiche à l'utilisateur un explorateur de fichier. L'utilisateur pourra choisir où stocker le fichier de log de l'analyse.

on_btnRun_clicked

Cette fonction correspond à l'action du bouton "btnRun". Elle s'occupe de récupérer tous les paramètres choisis par l'utilisateur. Avec ces données, elle va générer un objet "gaParameters" qui va tous les contenir.

Tous les paramètres vont envoyer des données à la classe "Dispatcher" qui va s'occuper de les envoyer à l'algorithme génétique.

Finalement la fonction va lancer l'analyse.

FSM - Machines d'états finies

Choix d'implémentation

Pour les machines d'états nous avons décidé de "wrapper" la classe QStateMachine du Qt Framework afin de les spécialiser dans l'analyse de séquences nucléiques. Pour cela notre classe MegaMachineManager (qui gèrent les MegaMachines , elles mêmes wrappant les QStateMachine) a besoin de tourner dans une EventLoop de Qt car les QStateMachine ne fonctionnes pas sinon (voir https://doc.qt.io/qt-5/qstatemachine.html).

Utils

Nous avons mis en place un fichier utils.h permettant de partager les StateDescriptor. Ce StateDescriptor est en fait une structure permettant de définir formellement un état pour nos machines d'états. En effet c'est grâce à cela que le Dispatcher fera la conversion de GA -> MegaMachine. Cette structure permet aussi l'évolutivité des stateMachines, il nous faut uniquement modifier cette structure, et pour chaque transition ajoutée dans cette structure il faudra ajouter son signal correspondant dans emitterACGT puis dans MegaMachine lors de la création de la machine. Il faudra par la suite ajouter la logique du décodage du nouveau signal dans le getMachine du dispatcher pour avoir un décodage des signaux ajouté correctement.

Un StateDescriptor est défini comme suit :

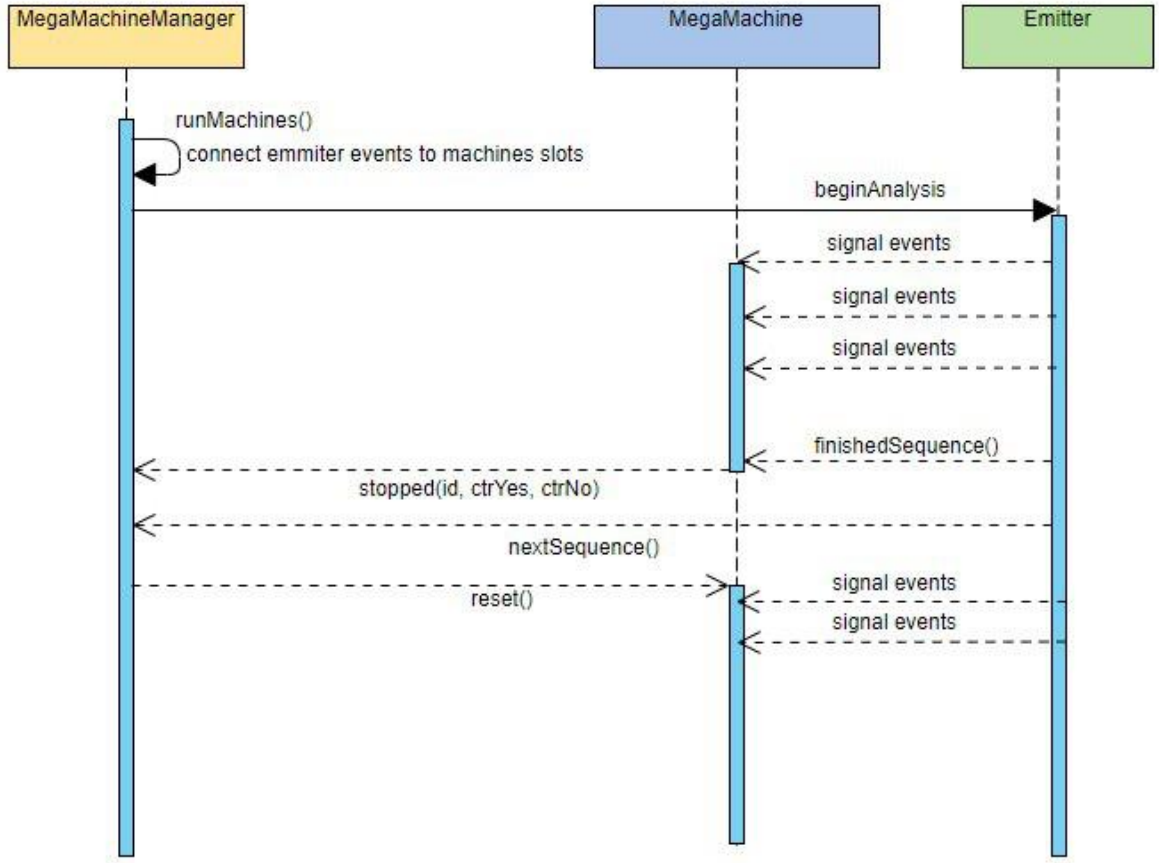
Un état est défini par une action (en entrée de l'état l'état émettra un signal sur MegaMachine), c'est cette action qui incrémente les compteurs ctrYes/ctrNo définis ci-après. De plus chaque état contient un vecteur de Transition. Ces transitions sont définies par un signal et un destinationState.

Fonctionnement

Structure

La structure se base un peu sur un concept de maître-esclave, en effet on a un MegaMachineManager qui va gérer des MegaMachine. De plus nous avons une classe émetteur ACGT qui permet de lire les séquences et de lancer des signaux sur la QStateMachine via le MegaMachine. Nous avons fait en sorte que ce manager puisse avoir autant de machine que possible sous son aile. Cependant nous ne l'utilisons pas à son plein potentiel dans notre implémentation du dispatcher.

Schémas



MegaMachineManager

Le MegaMachineManager permet de faire les **connexions nécessaires entre les MegaMachine et l'émetteur.** Puis entre lui-même et les machines. Il s'occupe aussi de tenir les scores pour chaque machine avec un vecteur de scores.

Ces scores sont calculés à chaque arrêt de la machine grâce au signal stopped() et ses paramètres, décrivant l'état de la machine au moment de son arrêt ce qui va nous permettre de calculer les scores. Si il y a une égalité entre ctrYes et ctrNo nous n'incrémentons pas le score. On regarde lequel de **ctrYes et ctrNo est le plus grand et on le prend comme résultat de la machine**, on va ensuite interroger le emitterACGT qui connaît le résultat attendu pour la séquence actuellement analysée. Si cela correspond, on incrémente le score de la machine.Sinon, on ne fait rien. **Cette méthode est modifiable aisément si l'on veut changer le calcul du score.**

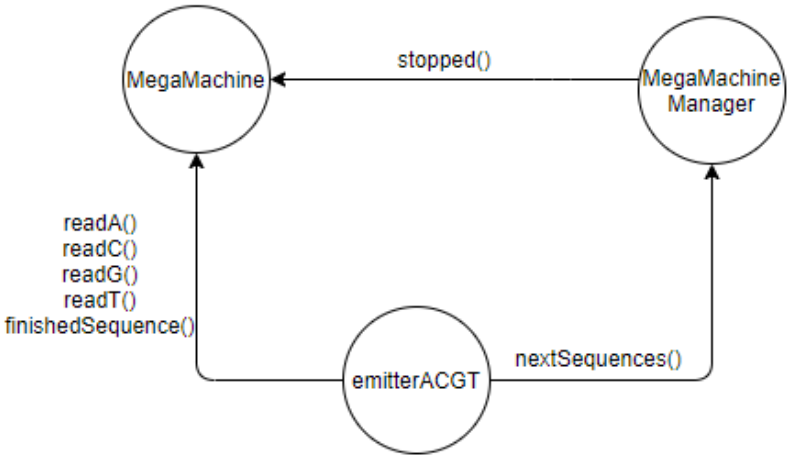
EmitterACGT

Cette classe permet uniquement de **lire des séquences nucléiques** afin d'émettre un signal pour chaque lettre lue dans le fichier de séquences. Cette classe nécessiterait une amélioration, en effet l'intégralité de chaque séquence est lue mais on pourrait simplement s'arrêter lorsque toutes les machines sont arrêtées.

Une fois la séquences finie on envoie un *finishedSequence()* aux machines qui va stopper toutes les machines. Puis ,un *nextSequence()* qui va effectuer un reset de toutes les machines. Nous verrons dans les prochaines sections comment les états sont générés.

Connexions

Les connexions ont étés importantes dans cette partie du projet afin que tous ces composants communiquent correctement entres eux. On peut voir ici les connexions nécessaire (connexions entre emitter et MegaMachine faites pour chaque MegaMachine du manager):



MegaMachine

Les états des QStateMachine ont des actions en entrée, ainsi lorsque l'on entre dans un état la QStateMachine émettra un signal, dans notre cas: NOTHING, YES ou NO.

Pour chaque signal émis depuis la machine on incrémente les ctrYes ou ctrNo.

Si ces compteurs atteignent un certain seuil (maxAlert, défini par l'utilisateur) la machine s'arrête (on émet un stop) et attends dans le finalState la prochaine séquence.

Les états ont des transitions définies pour chaque transition possible (une par lettre possible A, C, G, T, X) et chaque transition a un état de destination. Ce sont en fait des *slots* au sein de la machine qui nous permettent de passer d'état en état.

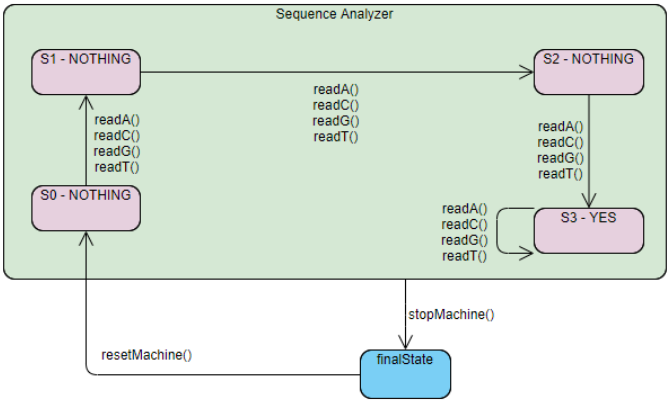
La classe MegaMachine permet de *wrapper* les QStateMachine afin de les spécialiser dans l'analyse des séquences.

On informe le manager de l'arrêt des machines (pour calculer le score) grâce au signal stopped().

Mise en place des états

Le groupe d'états Sequence Analyzer représentes les états propres à la machine créer grâce au paramètre donné au MegaMachineManager, c'est en fait la machine qui nous intéresse. **On prendra**

comme convention que le premier état donné est l'état initial. On ajoute aussi un *finalState* pour le traitement des séquences, ainsi grâce à cet état on peut arrêter des machines et les reset facilement.



Dispatcher

La classe dispatcher permet de lancer l'algorithme génétique ou une machine définie dans un fichier .machine au format json à l'aide des fonctions run() et runOneMachine(). C'est la classe centrale de notre implémentation qui permet la **gestion et l'interaction entre machine d'état et algorithme génétique**.

Initialisation des séquences

Les séquences sont récupérées à partir du fichier transmis dans le constructeur et sont insérées en premier lieux dans une *multimap<int, string>* avec comme clé l'identifiant de leur localisation et en contenu la séquence correspondante. Un second traitement sera effectué au lancement de la fonction *runOneMachine()* et à chaque itération intrinsèque à la fonction *run()*. Ce traitement consiste à l'insertion des séquences correspondante à la localisation recherchée lors du run dans une *multimap<string, bool>.* Cette multimap est définie comme suit : avec comme **clé la séquence** et comme **contenu un booléen** à true signifiant que la sortie attendue est vrai (séquence cherchée). Puis dans la même multimap l'insertion de séquence ne correspondant pas à la localisation cherchée (bool à false). Les séquences ne correspondant pas sont sélectionnées dans l'ordre d'apparition à l'intérieur du fichier.

Le nombre de séquences n'ayant aucune correspondance est le même que le nombre de séquence insérée correspondant à la localisation cherchée. Actuellement toute les séquences du fichier correspondant à la localisation recherchée sont insérée est analysée.

Run

```
void run(unsigned int stateNb, unsigned int maxAlert, const gaParameters& gaParam);
```

Afin de récupérer les paramètres nécessaires au fonctionnement de l'algorithme génétique, Dispatcher fournit une structure de donnée à cet effet qui sera demandée à l'appel de la fonction run().

```
struct gaParameters {
  int mutMode;
  int crossMode;
  int selectMode;
  float crossOverRate;
  float mutationRate;
  float selectivePressureRate;
  int popsize;
  int genNb;
  int elitpop;
```

```
  float tolRate;
  int tntsize;
};
```

Les deux autres paramètres nécessaires au lancement de l'algorithme sont les paramètres des machines d'états qui se résument aux nombres d'états qu'elles posséderont ainsi qu'au *maxAlert* CF. Section FSM - Machines d'états finies

Notre dispatcher inclut l'algorithme génétique en l'utilisant avec des paramètres définis en float ce qui permet au maximum la représentation de 32 états. Cette limitation est dues au fait qu'un état est représenté par un state action de 3 bits et que chaque destination state est représenté par 5 bits dans le cas de 32 états. Avec 5 transitions (A, C, G, T, X) par état cela fait 5*5+3 = 28bits / 32bits utilisés.

Fonction objective

La fonction objective est utilisée par l'algorithme génétique afin d'évaluer ses paramètres, cette fonction va s'occuper de transformer les paramètres binaires du ga (transférées sous forme de float ou de double) en machine d'état représentée sous forme de vector de StateDescriptor (Structure détaillée en section FSM - Machines d'états finies) puis lancera le State Machines Manager responsable de l'exécution de celle-ci.

Fonction getMachine

La fonction getMachine définit la façon dont nos State Machine seront créées par rapport au paramètres envoyés par l'algorithme génétique.

Afin de récupérer les paramètres réels de l'algorithme génétique en représentation binaire, une union a été utilisée:

```
uint32_t Dispatcher::convert(float from) {
  union {
    float value;
    uint32_t converted;
  } c32;

  c32.value = from;
  return c32.converted;
}
```

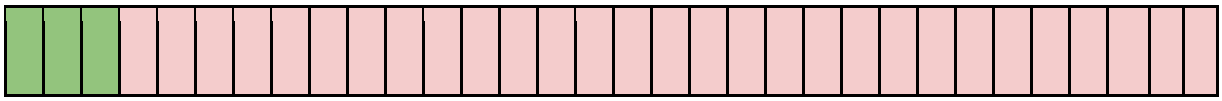
```
uint64_t Dispatcher::convert(double from) {
  union {
    double value;
    uint64_t converted;
  } c64;

  c64.value = from;
  return c64.converted;
}
```

Représentation binaire (binRepresentation):

State Action (taille de représentation binaire constante) 3bits

Destination States (Taille de représentation des états de destinations dépendant du nombre d'état)



Il y a une transition par type de signal (A, C, G, T, X) composée d'un destination states encodé du côté LSB correspond au signal A, puis dans l'ordre en direction du MSB ,C, G ,T et X.

Le nombre de bit nécessaire pour représenter un état en binaire est définit par le logarithme en base deux du nombre d'état de la machine arrondi à sa valeure supérieure. C'est le nombre de bit minimum pour représenter un nombre entier.

```
unsigned int nbBitState = static_cast<unsigned int>(ceil(log2(stateNb)));
```

Ainsi on peut en déduire le masque afin de les récupérer petit à petit les états de destination des transitions côtés LSB, on effectuera un shift du nombre de bit nécessaire à la représentation d'un état afin de récupérer le suivant: binRepresentation >>= nbBitState;

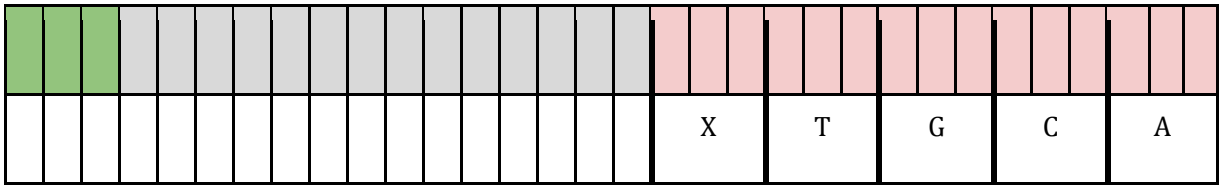
```
unsigned int MASK_TRANSITIONS = static_cast<unsigned int>(pow(2, nbBitState) -1);
```

Exemple pour 6 états:

State Action (Taille de représentation binaire constante) 3bits

Destination States (Taille de représentation des états de destinations dépendant du nombre d'état)

Inutilisé



Sauvegarde meilleur machine par ID

En fin de chaque itération (analyse d'un ID), la meilleur machine associée est sauvegardée au format json dans un fichier bestmachineAnalysis{ID}.machine.

```
{
    "GARepresentation": [
        nombre réel,
        ...
    ],
    "bestScore": nombre réel,
    "localisationTreated": num localisation,
    "machine": [
        {
        "stateAction": state action id,
        "transitions": [
                {
                    "destinationState": uint,
                    "signal": signal id
                },
                ...
        ]
        },
        …
    ],
    "maxAlertSet": uint
}
```

RunOneMachine

RunOneMachine permet de lancer une analyse sur le fichier de séquence sélectionné avec une seule machine bien définie dans un fichier .machine. Ce fichier est créé par le run d'une analyse dans l'algorithme génétique. La machine est récupérée à l'aide de la fonction parseJson() et parseJsonMachine() qui permettent respectivement de parser les informations générales tel que

GARepresentation, bestScore, … ainsi que parser la machine en elle même afin de récupérer un vector de StateDescripor (représentation d'une machine au sein de notre implémentation).

Lancement des machines

Les machines sont lancées à l'aide d'un MegaMachineManager ne lançant qu'une machine étant donnée qu'à l'appel de la fonction objective une seule machine lui est passée. Afin de garantir le bon fonctionnement des machines d'état, il est nécessaire de créer une EventLoop sans quoi elle ne pourront fonctionner (CF. Section FSM - Machines d'états finies). Le machine manager envoie un signal finished() lorsqu'il a terminé son exécution, ce signal sera mappé sur le slot quit() de l'event loop qui lui signale de sortir normalement.

```
QEventLoop loop;

QObject::connect(manager, SIGNAL (finished()), &loop, SLOT (quit()));

QTimer::singleShot(0, manager, &MegaMachineManager::runMachines);

loop.exec();
```

Algorithme génétique

Un constructeur a été ajouté par nos soins à la librairie galgo afin de permettre le choix du nombre de paramètre passé à l'algorithme génétique en runtime. Effectivement le constructeur de base de galgo prends les paramètres sous forme de parameter pack ce qui n'est pas définissable en runtime.

Le nouveau constructeur prend cette forme:

```
GeneticAlgorithm(Emitter *gaEmitter, const gaParameters& p, Func<T> objective,
        bool output, const std::vector<Parameter<T, N>>& parameters);
```

Le gaEmitter permet de notifier le dispatcher avec des informations détaillées sur l'avancement de l'analyse, la structure gaParameters permet de se simplifier le passage des paramètres relatif à la configuration de l'algorithme génétique. Le vector de de paramètre permet de définir au runtime le nombre qu'il en contiendra.

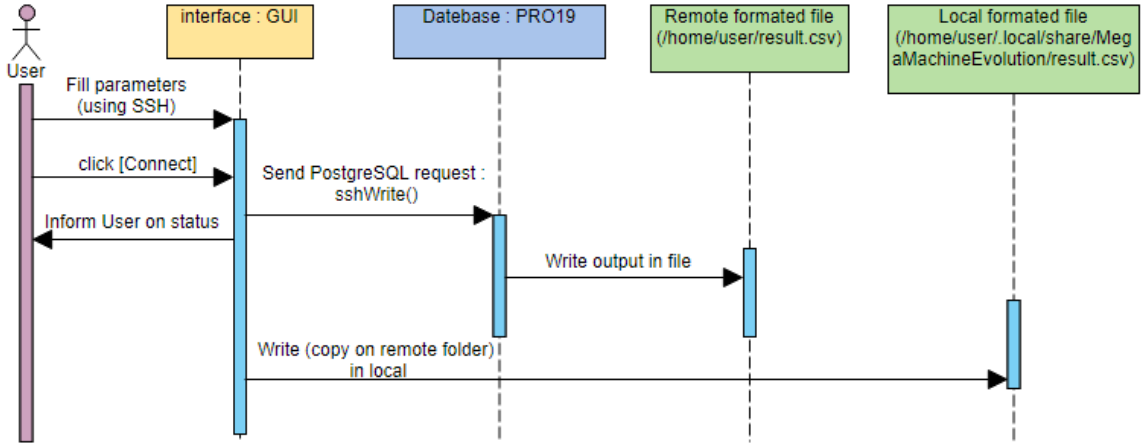
Interaction entre les classes

Maintenant que la plupart des parties du projet ont été présenté, nous allons présenter l'interaction entre les classes.

Interaction Client - Application

Le client aura deux manières pour lancer l'application comme précisé dans la partie GUI, soit en faisant une requête sur la DB soit en utilisant un fichier local déjà formaté.

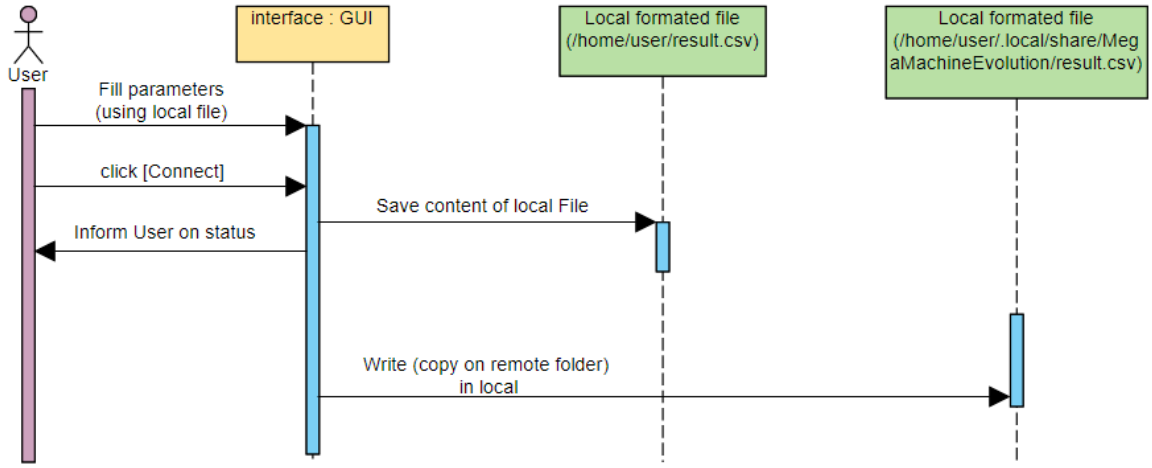
Utilisation du SSH



Lors de l'utilisation de la connexion en SSH, une requête à la base de donnée sera exécutée. Cette requête est formatée grâce à la méthode COPY de PostgreSQL l'output est formaté et est enregistré sur le /home/user du SSH. Ce fichier est ensuite copié en local (de l'utilisateur) dans le répertoire /home/user/.local/share/MegaMachineEvolution/result.csv.

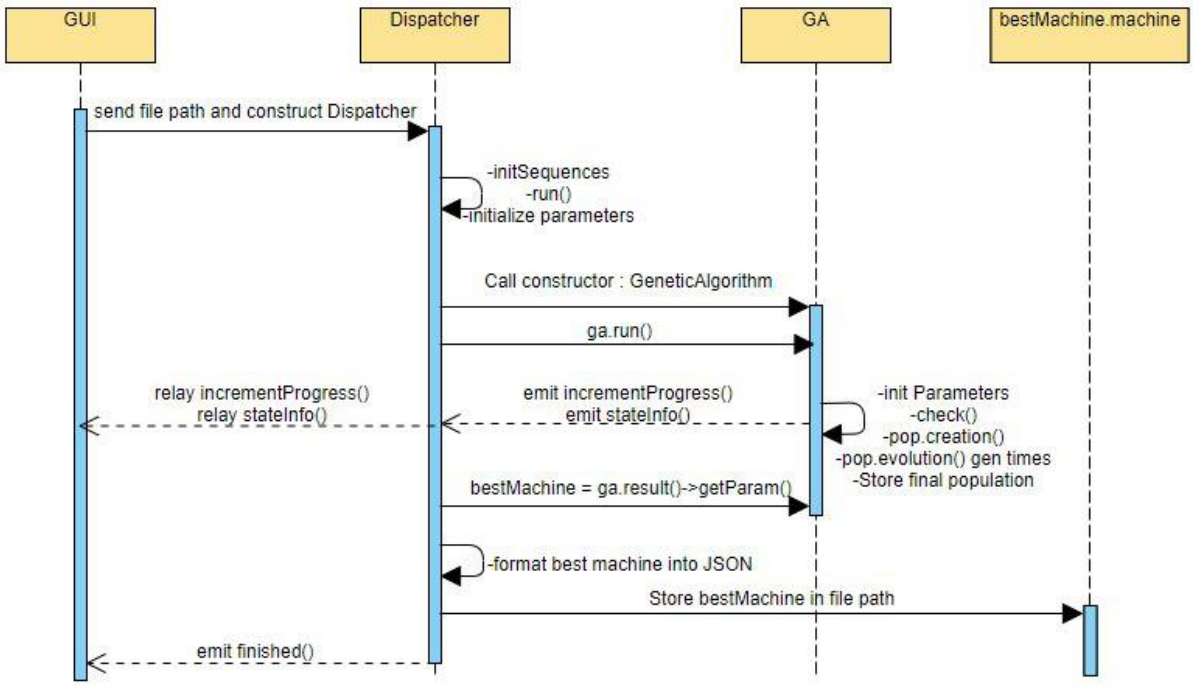
Ce fichier est après envoyé au dispatcher, voir ci dessous.

Utilisation d'un fichier local



Lors de l'utilisation du fichier en local, la GUI copie le fichier local à un autre endroit (même emplacement que lorsqu'on utilise SSH). Ce fichier est ensuite envoyé au dispatcher.

Interaction Application interne



Une fois le fichier source récupéré, celui-ci est transmis au dispatcher et est traité par la fonction initSequences(). Le dispatcher va alors pouvoir lancer l'algorithme génétique qui, à chaque génération émettra un signal pour incrémenter la barre de progression. Le dispatcher récupère à la fin de l'exécution de l'algorithme les paramètres de la meilleure machine et la formate en json afin de la sauvegarder.

Parsing XML

Présentation de la structure

La structure de l'XML que nous avons utilisé pour peupler la base de donnée était la suivante : (pour des soucis de lisibilité, seulement les parties qui nous intéressent sont préservées)

```
<LOCATE_Protein>
    <protein>
        <organism>
        <protein_sequence>
    </protein>
    <transcript>
        <transcript_sequence>
    </transcript>
    <scl_prediction>
        <source>
            <location>
            <goid>
        <source>
        <source> .. </source>
        <source> .. </source>
    </scl_prediction>
</LOCATE_Protein>
```

Dans l'élément organism qui se trouve dans l'élément protein on trouve l'hôte de la protéine (dans notre cas soit une souris soit un humain). Dans l'élément protein_sequence on retrouve, comme son nom l'indique la séquence protéique.

Dans l'élément transcript_sequence qui est l'enfant de l'élément transcript on retrouve la séquence ADN de la protéine.

Puis au final, dans les éléments sources on retrouve les différentes localisations des protéines. Une protéine pouvant avoir plusieurs localisation, cet élément source se répète jusqu'à couvrir toutes les localisations.

Cette séquence se répète autant de fois qu'il y a de protéines dans le XML.

Récupération des données

La gestion de récupération des données a été faites en fonction du degrés de "profondeur" des éléments recherchés. Ils ont été séparé en arbre de hiérarchie, de parents à petits enfants :

Parents	**protein**		**transcript**	**scl_prediction**	
Enfants	organism	protein_sequence	transcript_sequence	source	
Petits enfants				location	GOID

L'algorithme utilisé cherche donc pour chaque parents, les enfants et éventuelles petits enfants recherchés. Ces éléments sont changeables pour pouvoir rechercher n'importe quel élément dans le document.

Pour chaque protéine trouvée, une entrée est ajoutée dans un tableau résultat. Le tableau comporte la structure suivante :

[protein_sequence, organism, transcript, [location, GOID]] (pour chaque entrée)

Création script insertion des données

Afin d'éviter les doublons, des structures appropriées ont été créés pour préparer les scripts d'insertions. Nous avons utilisé 5 dictionnaires (en python).

5 dictionnaires sont créés (pour chaques tables, voir section Base de donnée) et toutes les valeurs du tableau sont insérées dans tous les maps.

On parcourt ensuite les différentes maps et on écrit dans un fichier les commandes à entrer dans la base de donnée.

Après chaque parcours de fichier XML, les 5 maps sont enregistrés dans un fichier afin d'être sûr de ne pas insérer des données déjà existante dans la base de donnée.

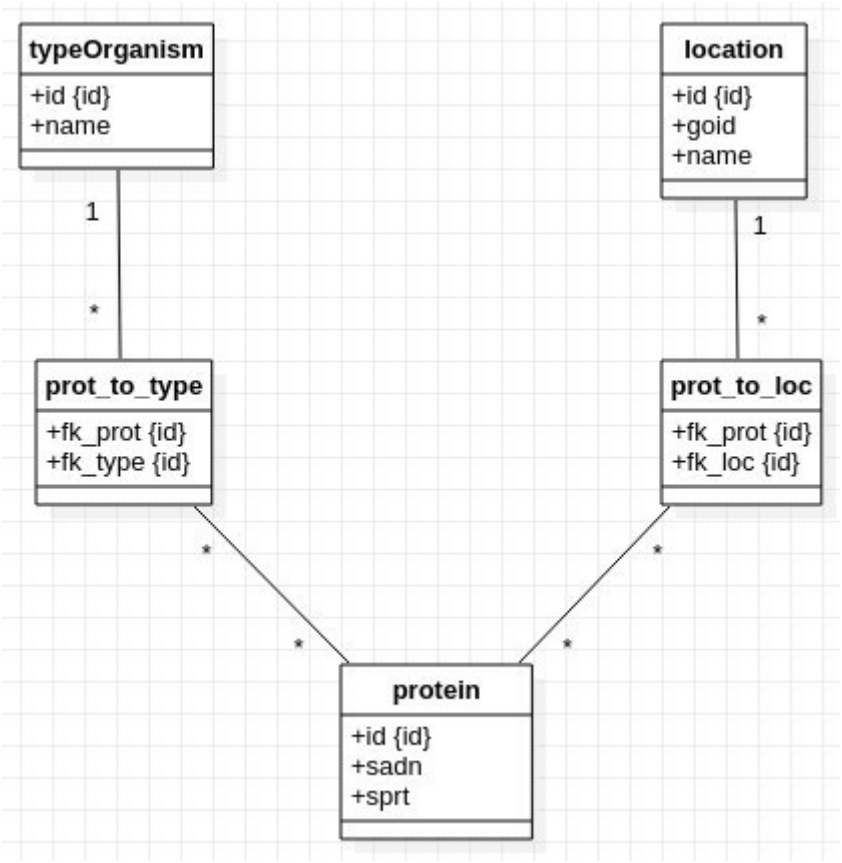
Note : ce fichier peut prendre une taille conséquente au fur et à mesure du temps, plusieurs Go de données.

Base de donnée

La base de donnée étant hébergée sur le serveur de l'école, la base de donnée utilise PostgreSQL.

Comme expliqué précédemment dans la partie XML, différents éléments étaient importants à récupérer. Nous avons donc créé une base de donnée afin de satisfaire ce besoin.

Voici la structure de la base de donnée



Quick help

Changer la méthode de calcul de score

Dans megamachinemanager.cpp on peut changer la logique du calcul du score aisément sur le slot *stop()*.

Changer le type des paramètres

dispatcher.h : runOneMachine()

dispatcher.cpp :

```
std::vector<galgo::Parameter<float,32>> parameters(
            this->stateNb,
            galgo::Parameter<float,32>({0.0, std::numeric_limits<float>::max()})
);

// initiliazing genetic algorithm
galgo::GeneticAlgorithm<float> ga(gaEmitter, gaParam, Dispatcher::objective<float>, true, parameters);

QObject::connect(gaEmitter, SIGNAL(incrementProgress(double)), this, SLOT(relay(double)));
QObject::connect(gaEmitter, SIGNAL(stateInfo(uint,double,double)), this, SLOT(relayState(uint,double,double)));

// running genetic algorithm
ga.run();
std::vector<float> bestMachine = ga.result()->getParam();
std::vector<StateDescriptor> * theBestMachine = getMachine(bestMachine);


debug << "Score : " << static_cast<float>(scores->at(0)) << endl;
delete theMachines;

return {static_cast<float>(scores->at(0))};
```

Vous pouvez remplacer les float mis en évidence par des doubles.

Changer l'encodage GA -> MegaMachine

Dans la fonction getMachine() on peut changer la logique du décodage des params du GA afin de pouvoir créer des machines. C'est cette fonction qui est utilisée dans tous les cas pour le décodage des params, on peut ainsi simplement la modifier pour changer la logique de l'encodage.

Évolution d'automates pour le traitement de séquences génomiques

Manuel d'utilisation

Équipe **B-13** :

Bonjour Mickael (mickael.bonjour@heig-vd.ch)

Mettler Samuel (samuel.mettler@heig-vd.ch)

Pantic Nemanja (nemanja.pantic@heig-vd.ch)

Seville Nathan (nathan.seville@heig-vd.ch)

Simeonovic David(david.simeonovic@heig-vd.ch)

Table des matières

Introduction

Ce document est destiné aux utilisateurs de notre application, il fournit des informations sur ce que l'utilisateur peut faire et les interactions qu'il peut avoir avec les différentes interfaces graphiques.

User guide

Nous traiterons ici de façon simplifiée du fonctionnement de notre programme. Si vous avez besoin de plus d'informations, n'hésitez pas à vous référer au document "Interface utilisateur".

Fenêtre de connexion

Lorsque vous lancerez l'application, il s'agira de la première fenêtre qui se lancera. Elle vous permettra de fournir des données à l'application afin qu'elles soient analysées. Pour cela, vous aurez 2 choix :

Le premier choix est de récupérer des données depuis un serveur externe (Remote data source). Il faudra fournir à l'application les informations suivantes :

1. Pour la base de données :
    a. Nom de l'hôte (généralement "localhost")
    b. Nom de la base de données
    c. Nom de l'utilisateur
    d. Mot de passe
    e. Requête SQL permettant de récupérer les données voulues
2. Pour la connexion SSH :
    a. Nom de l'hôte
    b. Nom de l'utilisateur
    c. Mot de passe

Une fois ces informations fournies, il vous suffira de cliquer sur le bouton "Connect". L'application se chargera ensuite de télécharger les données sur votre ordinateur.

Le deuxième choix et celui-ci de fournir un fichier à l'application (Local data source). Pour cela il vous suffira de cliquer sur le bouton "Browse file data", l'application vous ouvrira un explorateur de fichiers pour que vous puissiez sélectionner votre fichier de données. Cependant, il faut faire attention au format de données que celui-ci contient. Elles doivent être sous cette forme :

- *[séquence];[numéro d'identification du résultat]*
- *[séquence];[numéro d'identification du résultat]*
- *[séquence];[numéro d'identification du résultat]*
- …

Une fois le fichier sélectionné, il vous suffira simplement de cliquer sur le bouton "Continue".

Fenêtre de paramètres

Dans cette fenêtre, vous pourrez modifier plusieurs paramètres de l'algorithme génétique ainsi que des machines d'états. Vous pourrez aussi choisir le *filepath* du fichier de log. Ce fichier vous permettra de voir en détail le résultat de chaque génération. Une fois les options sélectionnées, il vous suffira de cliquer sur le bouton "Run Simulation" pour que l'analyse des séquences démarre. L'application vous affichera en bas de la fenêtre l'avancement de celle-ci.

Interface graphique

Dans cette partie, nous allons vous expliquer en détail comment utiliser notre application et son fonctionnement.

Attention : notre application n'est compatible qu'avec une base de données PostgreSQL

Fenêtre de connexion

Source de données externe

Pour commencer, voici la fenêtre qui apparaît quand vous lancez le logiciel :

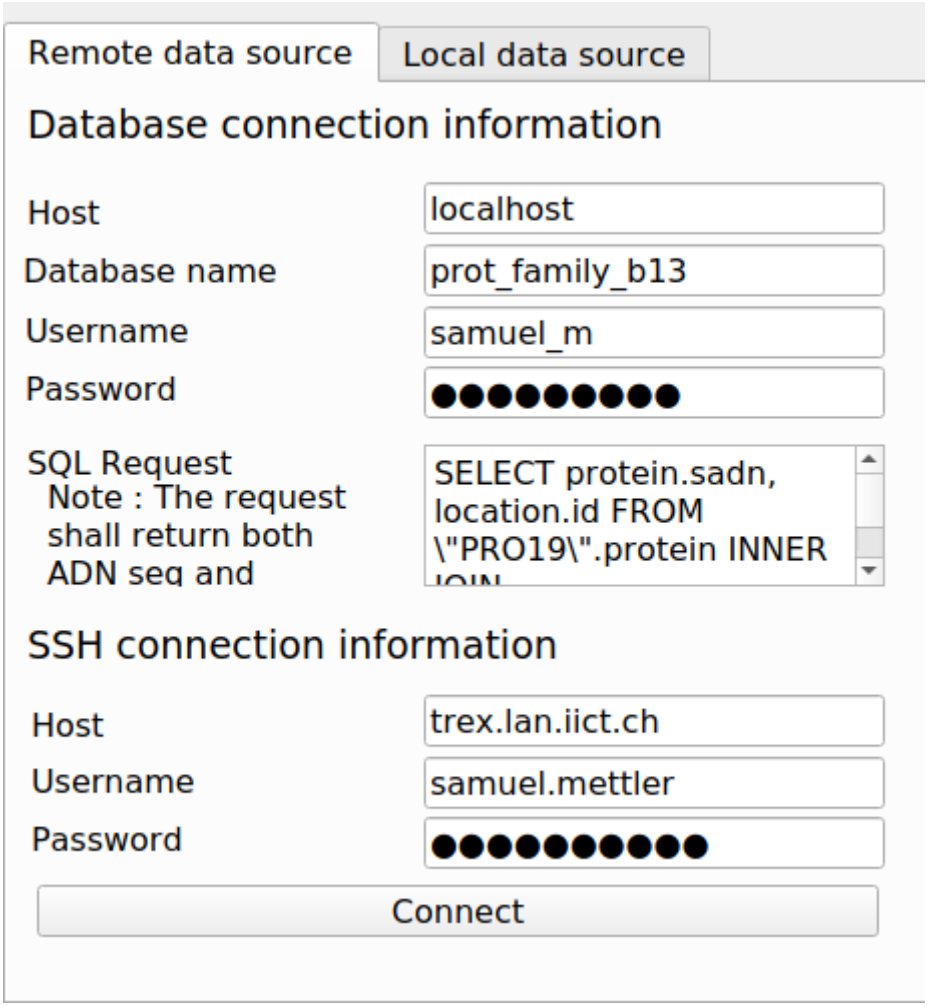


FIGURE 1 INTERFACE CONNEXION "REMOTE DATA SOURCE"

C’est la fenêtre de connexion à la base de données. Elle vous permet de vous y’ connecter afin de pouvoir récupérer les données à tester. Vous devez d’abord fournir les informations concernant la connexion à la base de données :

- Le nom de l'hôte qui contient la base de données (toujours localhost car l’application se connecte en SSH à la machine qui contient la base de données)
- Le nom de la base de données sur laquelle se connecter
- L’utilisateur qui va se connecter à la base de données
- Le mot de passe de l’utilisateur

Dans cette partie, nous avons un dernier champ, celui-ci concerne la requête SQL qui sera envoyé à la base de données. En effet vous pourrez choisir les champs que vous souhaitez récupérer depuis la base de données. Cependant, pour des raisons de formatage, vous serez **obligés d’ajouter un “\”** avant chaque utilisation des doubles guillemets pour que la requête soit correctement envoyée à la base de données. De plus, contrairement à une requête SQL/PostgreSQL vous ne devez pas terminer la requête par un “ ;”

Ensuite, nous avons la partie SSH. Dans celle-ci, vous devrez donner les informations suivantes afin que l’application puisse se connecter en SSH à l'hôte :

- Nom de l'hôte auquel se connecter
- L’utilisateur avec lequel se connecter
- Le mot de passe de l’utilisateur

Une fois toutes ces informations fournies à l'application, vous pouvez cliquer sur le bouton "Connect". Le logiciel va se connecter en SSH, puis , à la base de données et va récupérer les données de celle-ci :

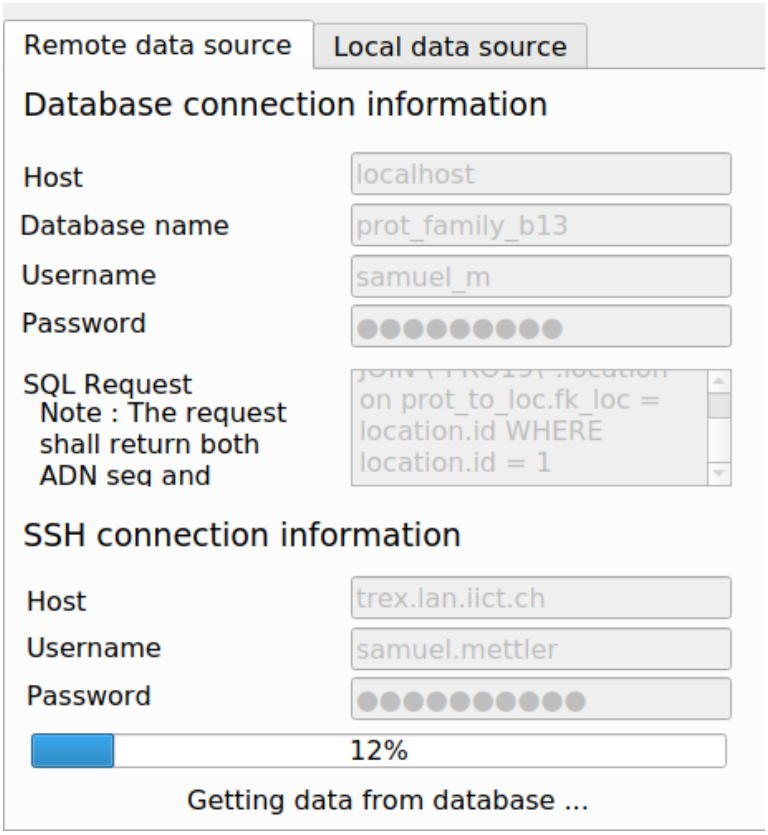


FIGURE 2 EXEMPLE DE TÉLÉCHARGEMENT DE DONNÉES

Les erreurs et leur signification :

- *Connection to host via SSH failed*
    - L'application n'arrive pas à accéder à la machine distante pour la connexion SSH
    - Le nom d'utilisateur ou le mot de passe est incorrect
- *Connection to the database failed*
    - Le nom de la base de données est incorrect
    - Le nom d'utilisateur ou le mot de passe est incorrect
- *Error during the download of data*
    - La syntaxe de la requête SQL est incorrecte

Source de données local

L'application vous permet aussi de fournir des données via un fichier local :

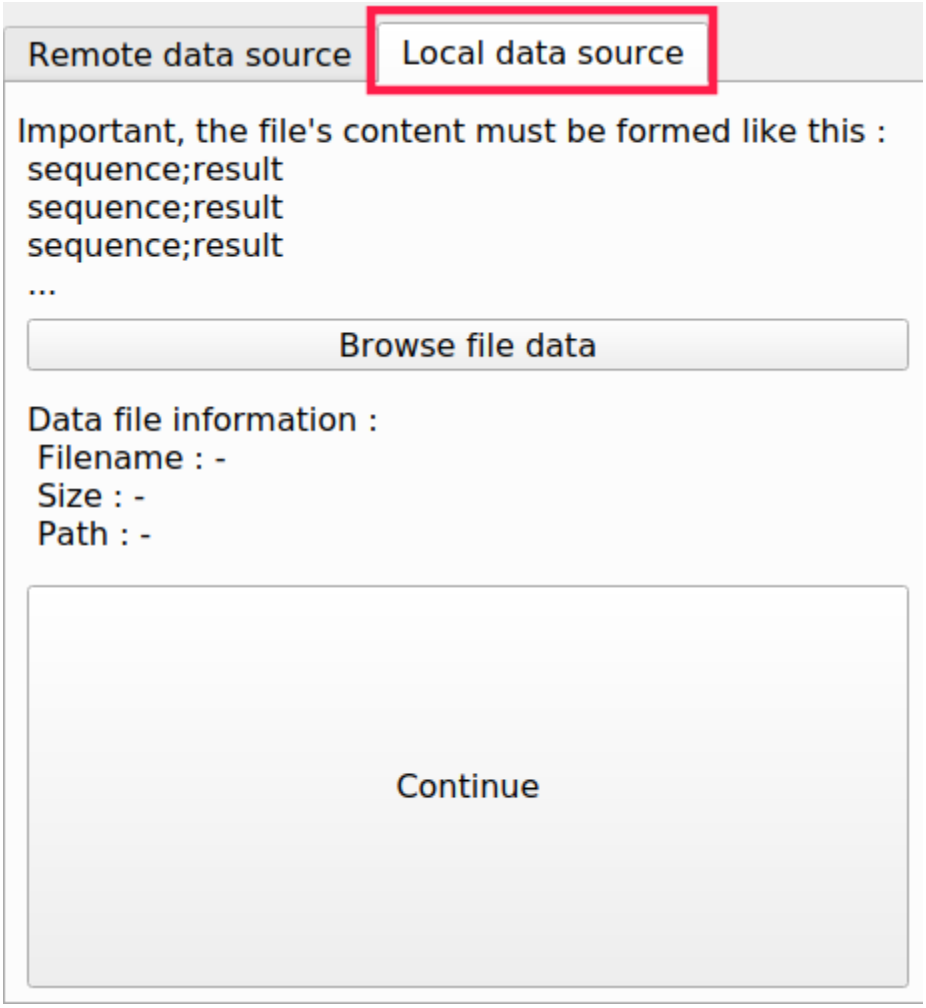


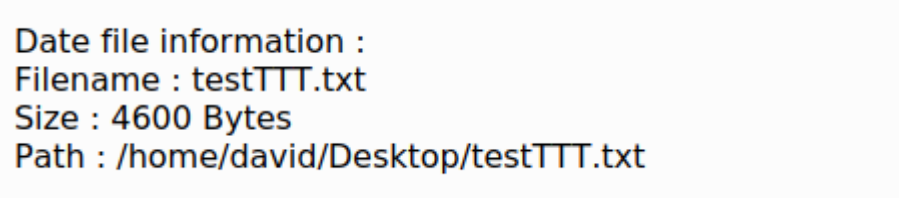
FIGURE 3 FENÊTRE "LOCAL DATA SOURCE"

Dans cette partie de la fenêtre, vous pourrez sélectionner un fichier en local afin de fournir des données à l'application. Cependant, le contenu du fichier doit avoir une forme bien spécifique pour que l'application fonctionne correctement :

- *[séquence];[numéro d'identification du résultat]*
- *[séquence];[numéro d'identification du résultat]*
- *...*

Attention, le nombre de séquence de l'identifiant qui a le plus de séquence associée ne doit pas être plus grand que le nombre de toute les autres séquences réunies afin de garantir une répartition à 50 % de séquences juste et 50 % de séquences fausses et ainsi effectuer une analyse optimale.

Une fois le fichier sélectionner, l'application vous affichera des informations basiques celui-ci :



Vous pouvez ensuite cliquer sur le bouton “Continue” afin d’arriver sur la fenêtre principale du logiciel qui vous permettra de choisir les paramètres de l’algorithmes génétique ainsi que des machines d’états.

Fenêtre de paramètres

Une fois la récupération des données à analyser, la fenêtre de connexion va se fermer et la fenêtre principale de l’application va apparaître :

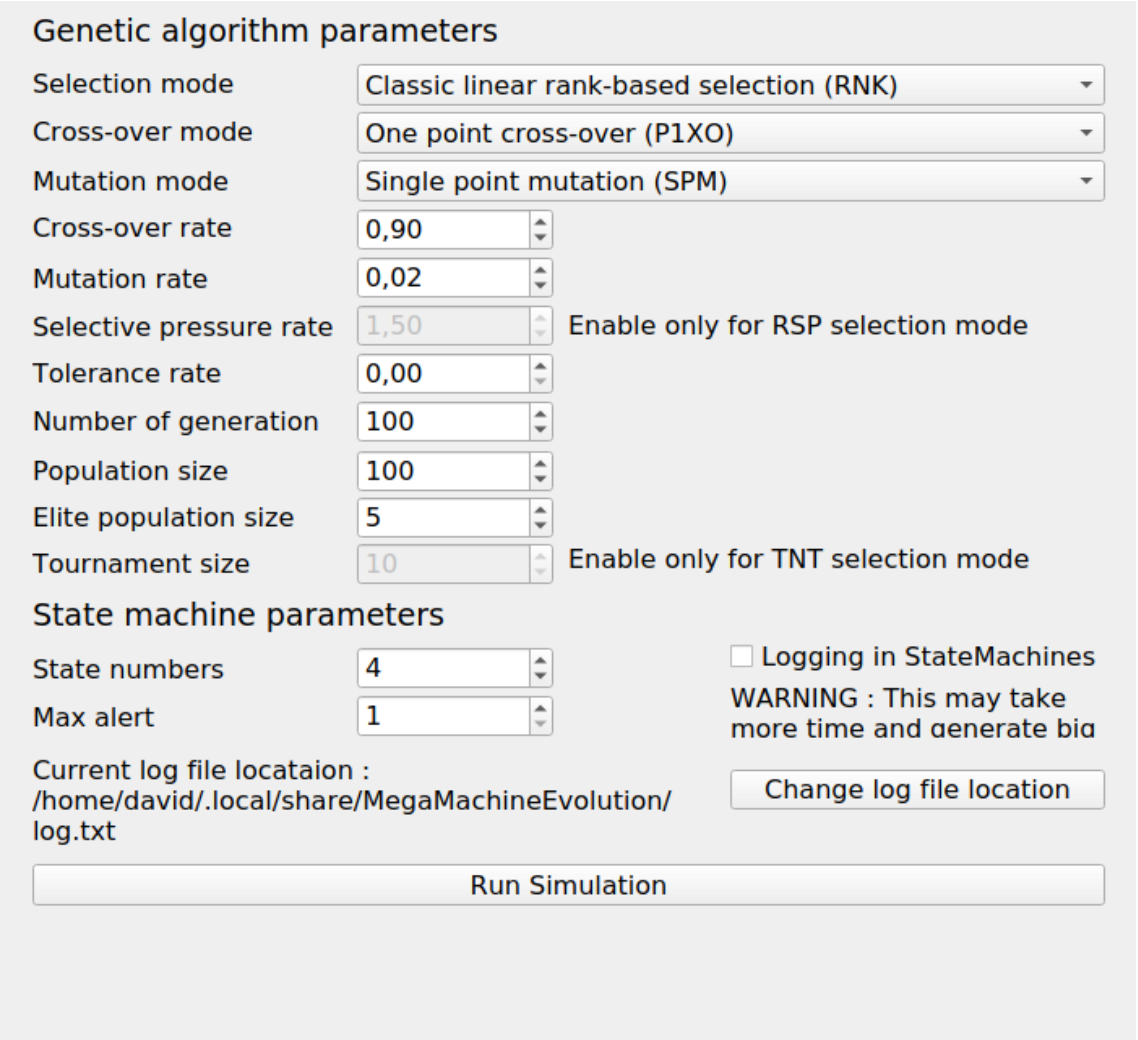


FIGURE 4 FENÊTRE DE PARAMÈTRES

Dans cette fenêtre, vous pourrez choisir tous les paramètres de l’application avant de lancer l’analyse des séquences :

Paramètre de l'algorithme génétique

Mode de sélection :

- Proportional roulette wheel selection (RWS)
- Stochastic universal sampling (SUS)
- Classic linear rank-based selection (RNK)

- Linear rank-based selection with selective pressure (RSP)
- Tournament selection (TNT)
- Transform ranking selection (TRS)

Mode de croisement :

- One point cross-over (P1XO)
- Two point cross-over (P2XO)
- Uniform cross-over (UXO)

Mode de mutation :

- Boundary mutation (BDM)
- Single point mutation (SPM)
- Uniform mutation (UNM)

Choix du taux de :

- Croisement : valeur de 0 à 1 (par défaut = 0.5)
- Mutation : valeur de 0 à 1 (par défaut = 0.5)
- Pression sélective (uniquement disponible pour le mode de sélection RSP) : valeur de 1 à 2 (par défaut = 1.5)
- Tolérance : 0 (inactif par défaut)

Choix du nombre de générations : aucune valeur par défaut

Choix de la taille de :

- Population : aucune valeur par défaut
- Population élite : 5 par défaut
- Tournoi (uniquement disponible pour le mode de sélection TNT) : 5 par défaut

Paramètre des machines d'états

- Nombre d'états de la machine
- Nombre maximum d'alerte

Autres paramètres

L'application vous offre le choix de changer la localisation du fichier de log qui sera produit lors de l'analyse des séquences. Un dossier par défaut est assigné de base. Une fois un nouveau dossier sélectionné, le logiciel affichera le nouveau dossier de destination :

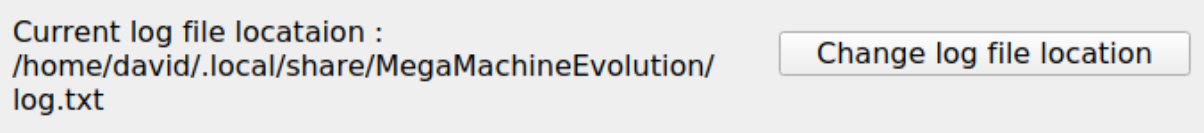


FIGURE 5 OPTION DE CHANGEMENT DE DESTINATION

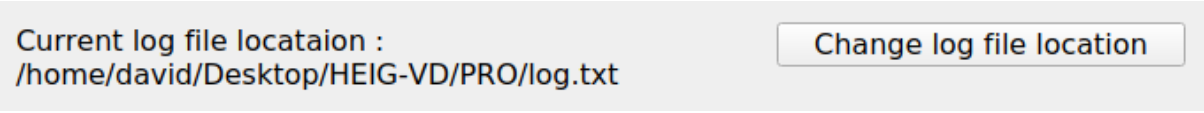


FIGURE 6 OPTION DE CHANGEMENT DE DESTINATION APRÈS CHANGEMENT

Lancement de l'analyse

Une fois tous les paramètres choisis, vous pourrez lancer l'analyse des séquences en cliquant sur le bouton "Run Simulation".

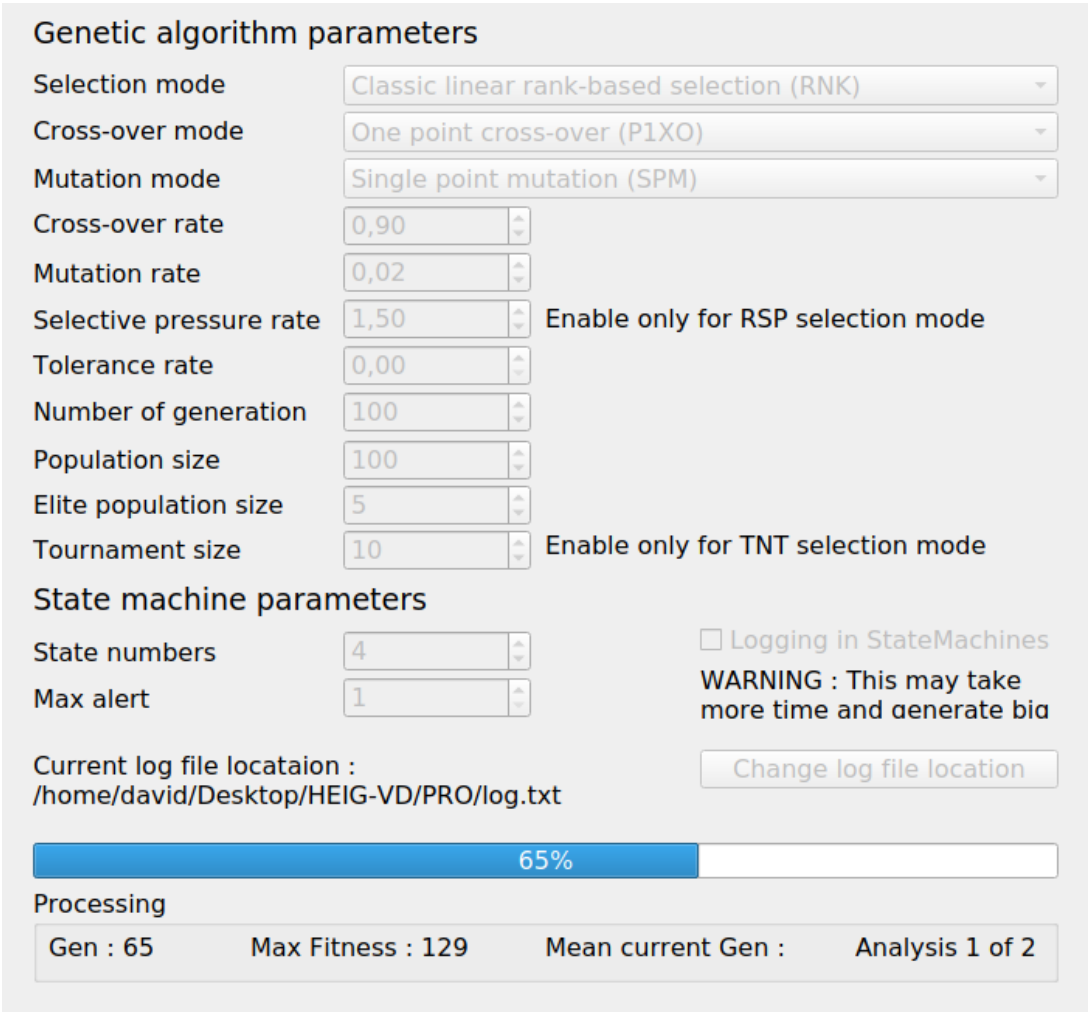


FIGURE 7 NOTRE APPLICATION EN PLEINE EXÉCUTION

Pendant la simulation, une barre de progression vous indique l'avancement de la simulation. Vous aurez aussi plus d'information en dessous de celle-ci :

- "Gen" : indique la génération en cours

- “Max fitness” : indique les meilleur score obtenu au moment de l’analyse
- “Mean current Gen” : moyenne des scores de la génération courante
- “Analysis” : indique le numéro de l’analyse en cours

Une fois l’analyse terminée, vous pourrez consulter le fichier de log pour avoir plus de détails sur chaque génération.