Développement
en environnement de base de données

Projet 3

Table des matières

Énoncé

Vous devez concevoir et réaliser une application capable de résoudre un problème d'optimisation géométrique.

Objectif général

Ce laboratoire vise la réalisation d'un projet complet visant la pratique des éléments suivants :

- programmation orientée objets,
- réalisation d'une petite librairie générique d'un algorithme génétique.

Objectifs spécifiques

Plus spécifiquement, ce projet vise ces considérations :

- conception orientée objets mettant de l'avant la notion d'encapsulation, d'héritage et de polymorphisme;
- développement favorisant la modularité et la réutilisabilité;
- utilisation d'une architecture partiellement imposée;
- réalisation d'outils d'analyse géométriques;
- réalisation d'un algorithme génétique
- programmation moderne en C++.

Contraintes

Vous devez respecter ces contraintes:

- Vous devez travailler en équipe de 4. Aucun étudiant ne peut travailler seul!
- Il est très important que tous les membres de l'équipe travaillent équitablement, car l'évaluation finale tient compte de plusieurs critères, dont la répartition de la tâche de travail.
- Vous devez utiliser :
    - le langage C++;
    - Visual Studio sous Windows.
- Vous devez faire une conception orientée objets soignée de votre projet en utilisant la structure imposée. Vous pouvez toutefois l'étendre et la bonifier.
- Vous devez faire l'interface graphique en mode console. Sans y être contraint, vous pouvez utiliser la petite librairie fournie pour faciliter l'utilisation de la console.
- La date de remise est non négociable.

Présentation de la problématique

Problème d'optimisation géométrique

Dans plusieurs domaines du génie, les problématiques d'optimisation font partie du quotidien. Plusieurs tâches différentes requièrent des concepts similaires et l'optimisation est une branche importante de tous les systèmes modernes.

Pour ce projet, on vous demande de résoudre le problème suivant :

- vous disposez d'un canevas (surface rectangulaire);
- sur le canevas se trouvent des points :
    - ces points correspondent à des obstacles et sont identifiés comme tels;
    - le nombre de points peut être variable;
    - les points sont répartis aléatoirement sur le canevas au début du problème mais fixe pendant la résolution de ce dernier;
- on tente de disposer **une** forme géométrique **quelconque** sur le canevas;
- la forme géométrique :
    - peut :
        - toucher au contour du canevas,
        - toucher aux obstacles;
    - ne peut pas :
        - dépasser le contour du canevas,
        - posséder un obstacle à l'intérieur;
    - est :
        - une forme quelconque : cercle, ellipse, rectangle orthogonal, polygone régulier, polygone convexe, polygone concave, …;
    - il est possible :
        - d'effectuer une transformation affine incluant l'une ou plusieurs de ces transformations :
            - translation;
            - rotation;
            - homothétie (« *scaling* »);
- au final, on cherche la plus grande forme pouvant correspondre à ces critères.

Mise en place d'un scénario

Pour chaque résolution de problème, trois paramètres fondamentaux doivent être mis en place et rester immuables tout au long de la résolution :

- la taille du canevas;
- le nombre d'obstacles et la disposition de ces derniers;
- la forme géométrique.

La forme peut subir une ou plusieurs transformations affines, mais ne peut être modifiée autrement. Par exemple, si la forme est un pentagone, elle le restera jusqu'à la fin de la résolution.

Objectifs de la résolution

Au final, il est attendu que la forme soit disposée de façon telle à maximiser sa taille sans enfreindre les règles énoncées.

Algorithme génétique

Pour résoudre un tel problème, plusieurs approches existent et certaines sont plus performantes que d'autres. Néanmoins, tenter de résoudre un tel problème est plus difficile qu'il n'y paraît.

Une solution viable serait une approche de force brute « *brut force* ». Cette technique consiste à estimer toutes les solutions possibles. Imaginons que nous avons une forme de type polygone régulier possédant les 5 paramètres suivants : position x, position y, rayon, nombre de côtés, angle de rotation. Supposons que la taille du canevas soit de 1000 x 1000 et qu'une approche naïve est mise de l'avant en tentant d'essayer toutes ces combinaisons :

- position x : [0, 1000[= 1000 possibilités
- position y : [0, 1000[= 1000 possibilités
- rayon : [1, 500] = 500 possibilités
- nombre de côtés : [3, 9] = 7 possibilités (choisi arbitrairement)
- rotation : [0, 360[= 360 possibilités (supposant que 1° soit suffisant comme résolution angulaire)

Si on test toutes ces possibilités, il existe 1 260 000 000 000 solutions différentes. Soit 1.15 tera-solutions où environ 350 heures à raison d'une évaluation à chaque μsec (10^{-6} sec). Pourtant, ce problème est tout simple!

Plusieurs algorithmes d'intelligence artificielle permettent d'aborder un tel problème en prétendant identifier une solution sous-optimale. C'est-à-dire que la solution trouvée ne sera pas la meilleure, mais certainement intéressante.

L'algorithme génétique fait partie des algorithmes évolutifs par convergence qui donne généralement des résultats satisfaisants pour des petits problèmes. Dans sa forme fondamentale, son implémentation est relativement simple et ne requiert pas des connaissances mathématiques avancées. C'est cet algorithme que vous devez obligatoirement implémenter.

Espace de solution

L'espace de solution représente un espace à *n* dimensions où chaque dimension est un axe orthogonal représentant une variable liée à la représentation de la solution. Par exemple, notre exemple précédent présentant 5 variables et par conséquent, nécessite un espace à 5 dimensions pour le résoudre.

Solution

Une solution est une hypothèse de réponse au problème donné. Reprenant l'exemple précédent, voici un exemple de solution :

- position x : 434
- position y : 777
- rayon : 89
- nombre de côtés : 4
- rotation : 167

Cette hypothèse représente une solution possible au problème. Néanmoins, il est difficile de dire si elle est adéquate sans pouvoir la comparer à d'autres ni pouvoir estimer sa performance.

Au début de l'algorithme génétique, plusieurs solutions sont créées et tous les paramètres sont déterminés aléatoirement.

Population

Les algorithmes évolutifs utilisent souvent un grand nombre de solutions simultanément. Cette approche permet plusieurs avantages :

- meilleure recherche de l'espace de solution (meilleure couverture);
- permets d'échanger de l'information entre les solutions;
- permets à certains individus d'explorer plus librement des zones moins intéressantes à priori.

Ainsi, la population consiste en une liste de *n* solutions.

Évaluation de la performance

Intimement lié à la solution, il est essentiel d'établir une métrique pouvant estimer la performance d'une solution. On appelle souvent cette métrique « *fitness* ». Il est important que cette métrique soit cohérente avec la problématique à résoudre.

Un « bon fitness » peut tendre vers l'infini, vers l'infini négatif ou vers zéro. Selon le cas, le problème en deviendra un de maximisation ou de minimisation.

Gène

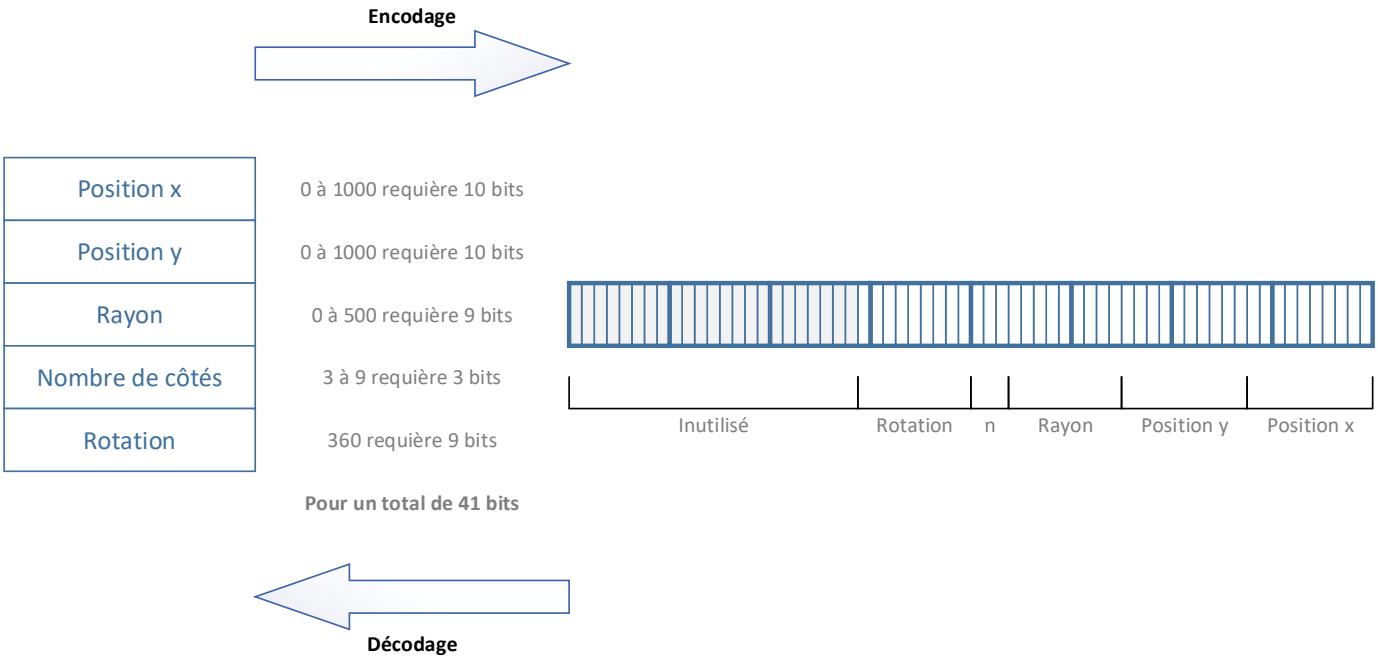
Un gène est ce qui représente toute l'information d'une solution. Dans notre exemple précédent, ce sont les 5 paramètres à optimiser. Un gène représente ces informations codées dans une série de bits contigus.

Les opérations génétiques sont faites sur les gènes eux-mêmes. Pour ce projet, on réalisera des opérations sur des entiers uniquement. Ainsi, si vous désirez travailler avec des réels, il faudra déterminer la résolution souhaiter et les manipuler vous-même comme étant des entiers.

Encodage et décodage

Il est essentiel de créer une fonction d'encodage et de décodage qui permet de passer d'une représentation à l'autre. D'ailleurs, ce sont ces fonctions qui passe d'une représentation en nombre réel vers une représentation en entier et vice versa.

Par exemple ceci :



Sélection

L'une des étapes importantes est la création d'une nouvelle génération de solutions. Cette étape permet de garantir (sous conditions) une évolution de la population afin de déterminer une meilleure solution.

Le processus de sélection permet de choisir deux solutions qui, en partageant leurs bagages génétiques (leur gène), engendreront une progéniture se rapprochant d'une meilleure solution.

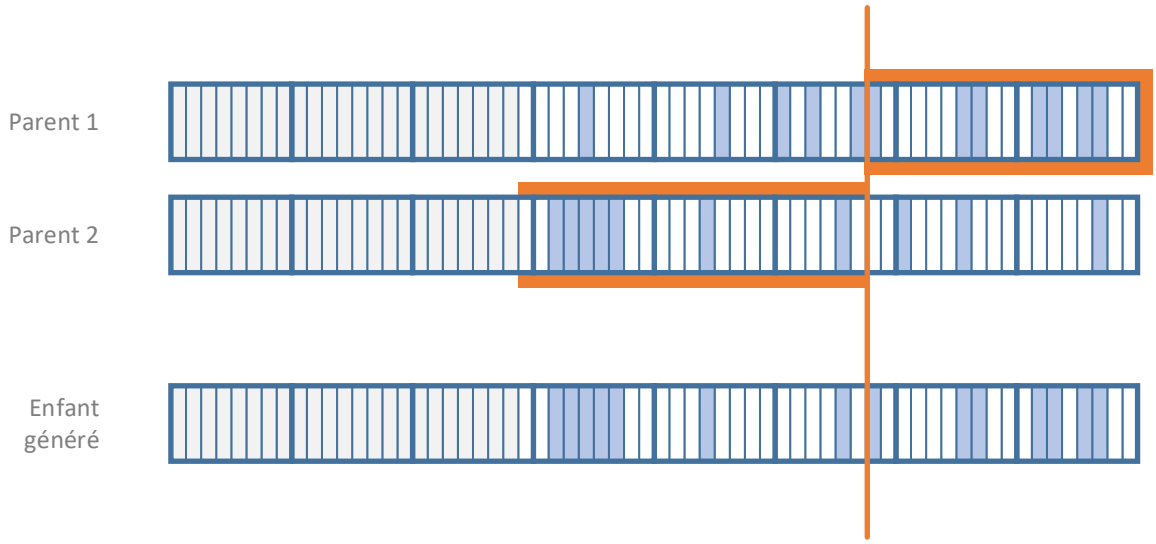
Il existe plusieurs stratégies, mais nous envisagerons deux techniques :

- L'élitisme consiste à retenir les meilleurs candidats de la population précédente et de les injecter directement dans la population de la nouvelle génération. Cette stratégie garantie de ne pas perdre les meilleurs candidats.
- La sélection proportionnelle à la performance (nommé « *Roulette Wheel* ») permet de sélectionner des parents en favorisant ceux qui sont les plus performants. C'est-à-dire, ceux dont la métrique de « fitness » est plus favorable.

Croisement

Lorsque deux solutions sont sélectionnées pour engendrer une nouvelle solution, le processus de croisement est mis de l'avant. Ce dernier consiste à déterminer aléatoirement un lieu de séparation parmi tous les bits utilisés (pivot). Ensuite, on crée l'enfant en prenant la partie de droite d'un parent et la partie de gauche de l'autre parent. Cette procédure garantit un échange d'information simple et efficace.

Par exemple :



Mutation

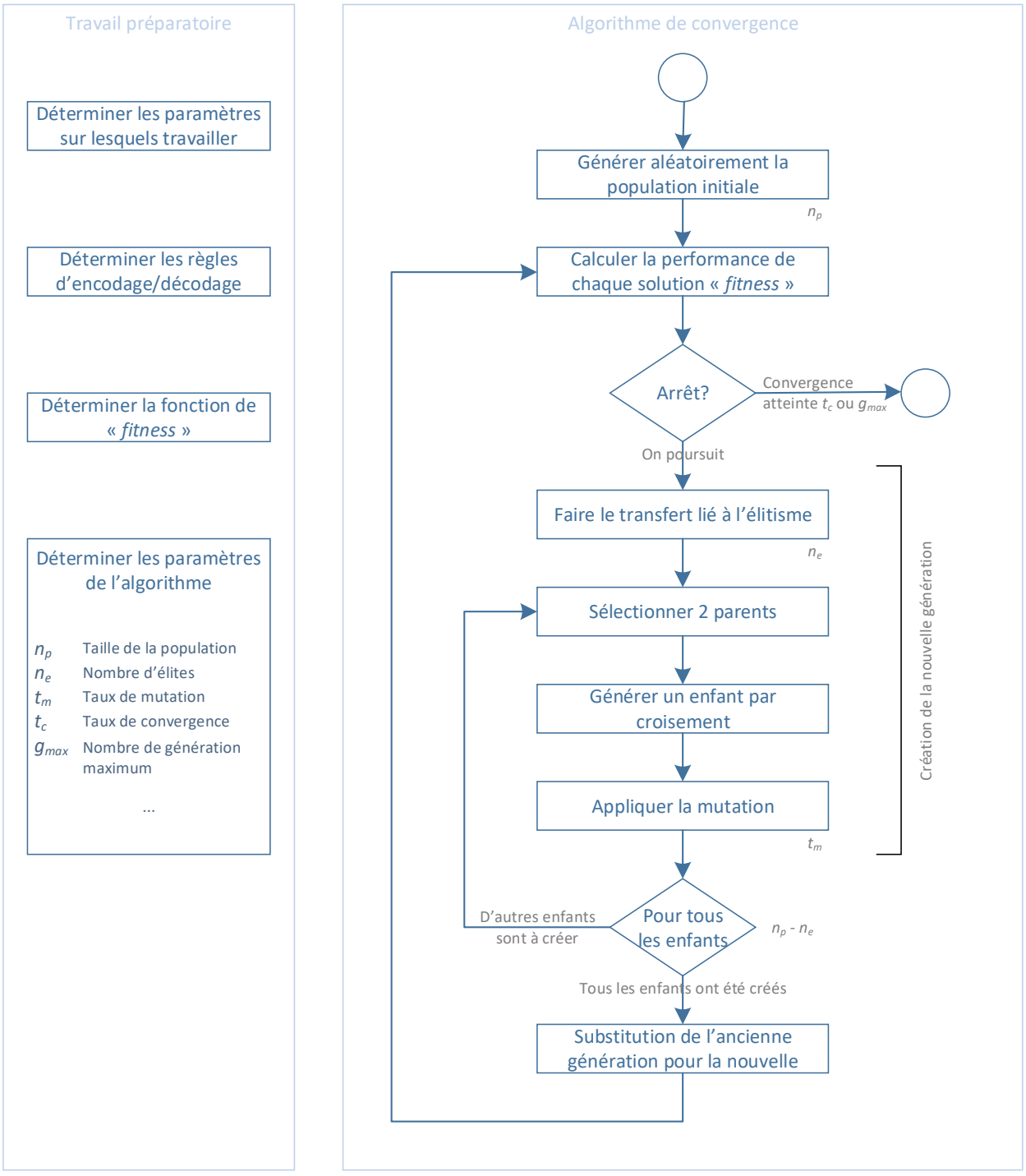
La mutation est un processus essentiel et intrinsèque à l'algorithme génétique. C'est ce processus qui garantit la recherche constante de nouvelle zone dans l'espace de solution. C'est aussi ce mécanisme qui permet, dans certaines occasions, de sortir d'un minimum local.

Le processus est très simple et se base sur un pourcentage établi qui déterminer la probabilité d'avoir une mutation. Cet indicateur est nommé le taux de mutation (typiquement entre 1% et 15% selon la nature du problème).

À la création d'un nouvel enfant, on déterminer si une mutation a lieu selon le taux de mutation. Si aucune mutation n'a lieu on laisse le gène tel quel. Sinon, on bascule aléatoirement un bit du gène.

Mécanisme de résolution

Voici un schéma illustrant l'algorithme génétique dans sa forme la plus simple :



Critère d'arrêt

Cet algorithme pourrait être exécuté en boucle indéfiniment sans pour autant trouver de meilleure solution (même s'il en existe une). Néanmoins, il faut établir un critère d'arrêt qu'il faut appliquer.

Voici les deux critères d'arrêt à implémenter :

- un certain nombre d'itérations est atteint;
- un taux de convergence est atteint, par exemple : moins de *x*% de variation de la performance depuis les *y* dernières itérations.

Extension de l'algorithme présenté

L'algorithme présenté est une version minimaliste de l'algorithme génétique. Il existe évidemment un très grand nombre de variantes et d'améliorations possibles.

Vous pouvez laisser libre cours à vos idées pourvu que vous respectiez les constituants présentés.

Plusieurs populations simultanées

Il est justement demandé d'implémenter le fait d'avoir plusieurs populations qui évolue en parallèle. Si vous réussissez à bien modulariser votre code, avoir plusieurs populations est pratiquement aussi facile qu'en avoir une seule.

Si vous le désirez, vous pouvez faire des migrations de sous-populations entre les populations ou vous pouvez aussi réaliser du métissage.

Injection de connaissances à priori

Selon la nature du problème et sa représentation, il est souvent frustrant et parfois incompréhensible d'observer les solutions converger vers un minimum local et qu'elles restent coincer là.

Il est fréquent et pertinent d'ajouter des connaissances à priori dans l'algorithme génétique. Ainsi, selon la connaissance à injecter, on se glisse dans le processus pour faire l'insertion ou la modification identifiée.

Il est important que cet ajout ne dénature pas l'algorithme, mais vous aurez de bien meilleurs résultats ainsi.

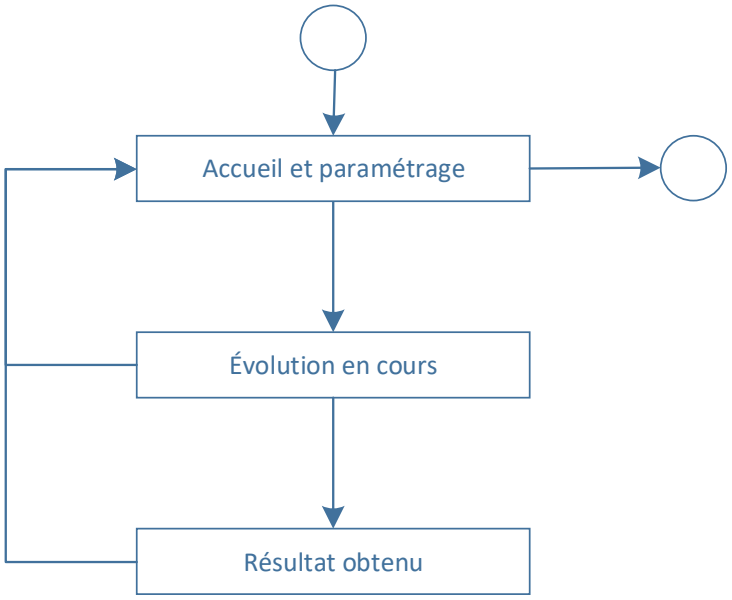
Réalisation technique

Le logiciel réalisé doit offrir une interface utilisateur minimale mais fonctionnelle incluant :

- la présentation de :
    - la problématique,
    - la résolution en cours ,
    - la solution finale;
- une interaction minimum avec le logiciel.

Déroulement du logiciel

Selon l'état du logiciel, voici les actions et affichage à réaliser :



Selon l'état courant, il existe certaines actions possibles et un affichage associé.

État	Touche	Action	Affichage
Accueil	`q / w`	Augmente / réduit de 10 le nombre d'obstacles. défaut : 25 – minimum : 5 – maximum : 255	Le canevas est affiché. Selon les options, on affiche les obstacles et la forme sélectionnée centrée.
	`e`	Réinitialise aléatoirement la position des obstacles.	
	`1 - 4`	Détermine le nombre de population.	
	`a`	Bascule la forme géométrique à traiter : Cercle – Rectangle – Polygone	
	`z`	Bascule l'affichage des obstacles : aucun – tous	
	`x`	Bascule l'affichage de la forme géométrique choisie : cache – affiche	
	`barre esp.`	Démarre la résolution (va à *Évolution*).	
Évolution	`barre esp.`	Mise en pause ou reprise de la résolution.	Le canevas est affiché. Selon les options, on affiche les obstacles et les solutions. Chaque population possède sa couleur.
	`s`	Si en pause, fait un seul pas d'évolution.	
	`z`	Bascule l'affichage des obstacles : aucun – tous	
	`x`	Bascule l'affichage des solutions : aucune – toutes – la meilleure	
	`esc.`	Quitte la résolution et reviens à *Accueil*.	
Résultat	`barre esp.` `esc.`	Retour à *Accueil*.	Le canevas est affiché. Selon les options, on affiche les obstacles et les solutions. Chaque population possède sa couleur.
	`z`	Bascule l'affichage des obstacles : aucun – tous	
	`x`	Bascule l'affichage des solutions : aucune – toutes – la meilleure	

Note, lorsqu'on affiche tous les obstacles, une couleur plus foncée est utilisée pour afficher les différentes solutions mais la même couleur plus pâle est utilisée pour afficher la meilleure solution.

Paramètres de l'application :

Il est attendu que **tous** les paramètres de l'application soient complètement paramétrables à partir du fichier où se trouve le `main`. Voici quelques constituants essentiels à paramétrer :

- taille du canevas et disposition relative pour l'affichage;
- nombre d'obstacles disposés aléatoirement;
- les paramètres de l'algorithme génétique : taux de mutation, nombre associé à l'élitisme, taille de la population, nombre de populations, …
- toutes les couleurs utilisées pour l'affichage;
- …

Pour y arriver, il pourrait être pertinent de créer une `struct` ou une `class` permettant de bien encapsuler ces paramètres.

Formes géométriques et transformations affines

Vous devez réaliser l'implémentation de trois formes géométriques différentes et permettre la résolution du problème pour chacune;

1.	cercle	translation		homothétie
2.	rectangle orthogonal	translation		homothétie
3.	polygone quelconque	translation	rotation	homothétie

Il est fortement recommandé de réaliser l'implémentation des formes dans l'ordre présenté. La réalisation des deux premières formes est plus facile étant donné leur invariance à la rotation. Ça facilite la représentation et les calculs géométriques associés.

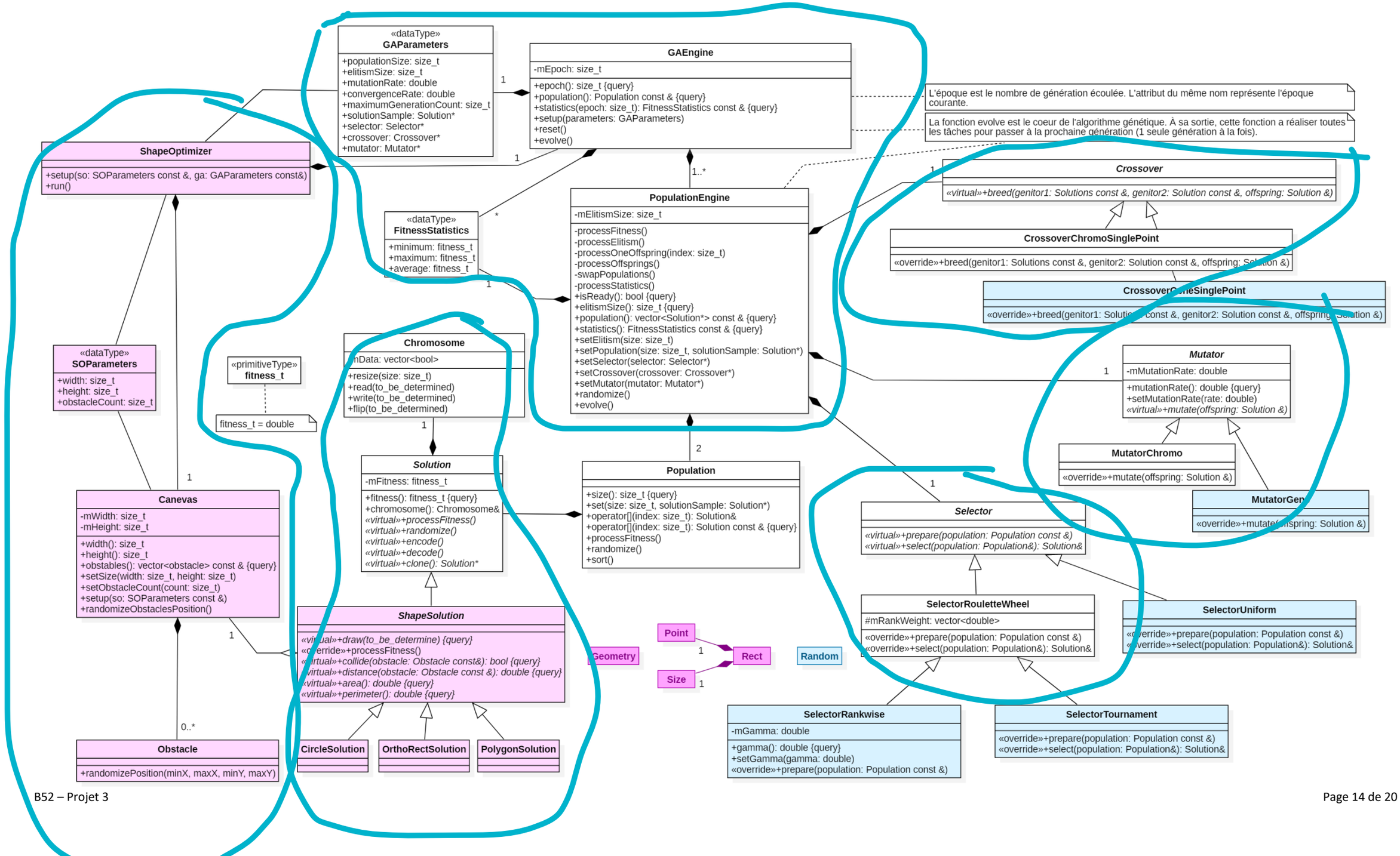
Ne vous méprenez pas, la difficulté de ce projet réside davantage dans les calculs géométriques associés au polygone que dans l'algorithme génétique lui-même.

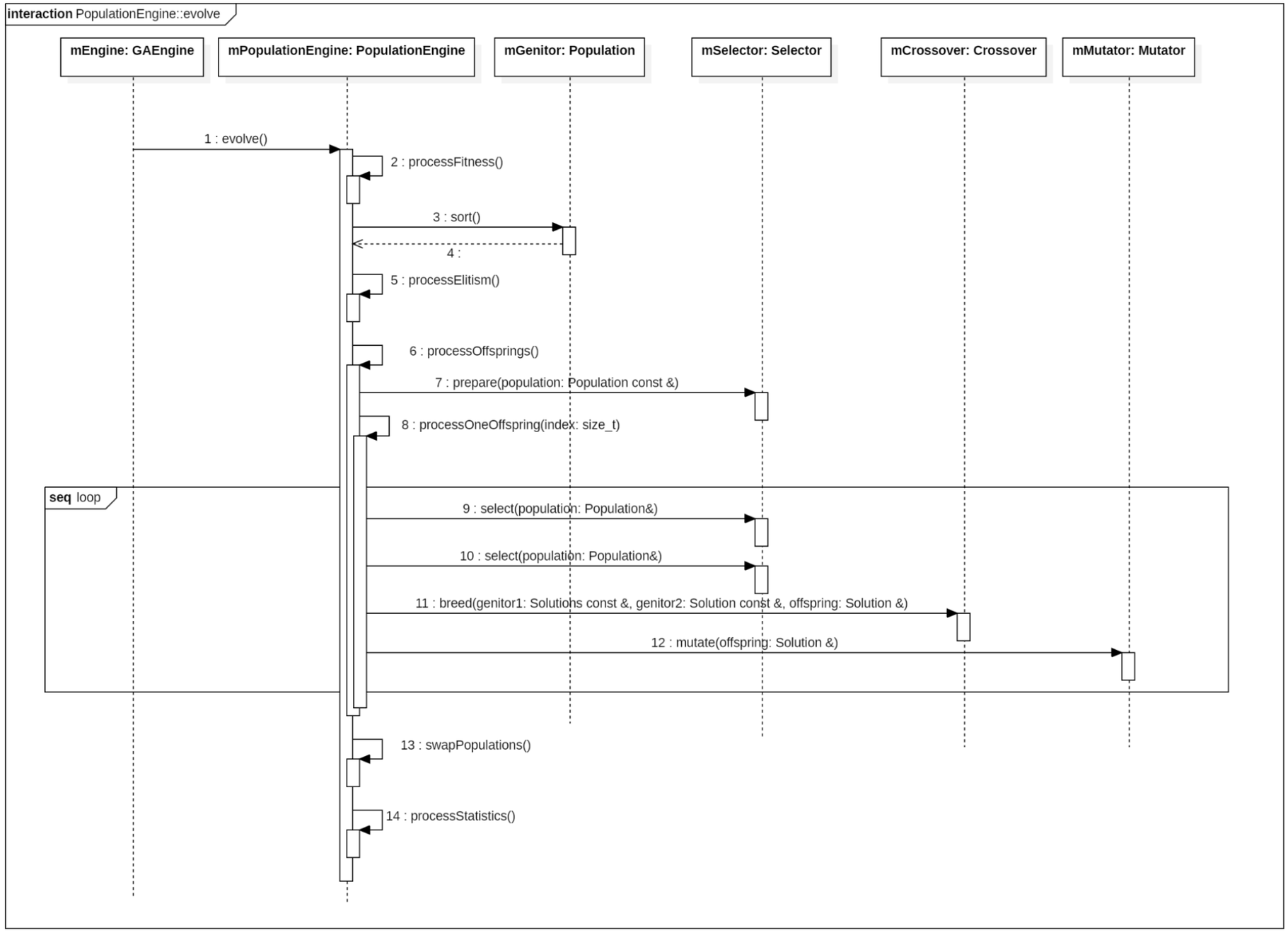
Contraintes reliées à la conception orientée objets

Vous devez mettre en place une solution orientée objets correspondant au schéma UML de la page suivante.

On peut d'abord remarquer les éléments suivants :

- Coloration :
    - les classes blanches indiquent les classes à réaliser liées à l'algorithme génétique;
    - les classes bleues indiquent des suggestions de classes pouvant être réalisées et qui sont liées à l'algorithme génétique;
    - les classes mauves (contour noir) sont les classes à réaliser liées à ce programme spécifique;
    - les classes dont le contour et texte sont colorées représentent des suggestions pouvant vous aider pour le reste de ce travail.
- La classe `ShapeOptimizer` est la classe centrale de l'application. C'est elle qu'on instancie dans le `main` et qu'on exécute.


«dataType»
GAParameters
+populationSize: size_t
+elitismSize: size_t
+mutationRate: double
+convergenceRate: double
+maximumGenerationCount: size_t
+solutionSample: Solution*
+selector: Selector*
+crossover: Crossover*
+mutator: Mutator*
GAEngine
-mEpoch: size_t
+epoch(): size_t {query}
+population(): Population const & {query}
+statistics(epoch: size_t): FitnessStatistics const & {query}
+setup(parameters: GAParameters)
+reset()
+evolve()
L'époque est le nombre de génération écoulée. L'attribut du même nom représente l'époque courante.
La fonction evolve est le coeur de l'algorithme génétique. À sa sortie, cette fonction a réaliser toutes les tâches pour passer à la prochaine génération (1 seule génération à la fois).
ShapeOptimizer
+setup(so: SOParameters const &, ga: GAParameters const&)
+run()
«dataType»
FitnessStatistics
+minimum: fitness_t
+maximum: fitness_t
+average: fitness_t
PopulationEngine
-mElitismSize: size_t
-processFitness()
-processElitism()
-processOneOffspring(index: size_t)
-processOffsprings()
-swapPopulations()
-processStatistics()
+isReady(): bool {query}
+elitismSize(): size_t {query}
+population(): vector<Solution*> const & {query}
+statistics(): FitnessStatistics const & {query}
+setElitism(size: size_t)
+setPopulation(size: size_t, solutionSample: Solution*)
+setSelector(selector: Selector*)
+setCrossover(crossover: Crossover*)
+setMutator(mutator: Mutator*)
+randomize()
+evolve()
Crossover
«virtual»+breed(genitor1: Solutions const &, genitor2: Solution const &, offspring: Solution &)
CrossoverChromoSinglePoint
«override»+breed(genitor1: Solutions const &, genitor2: Solution const &, offspring: Solution &)
CrossoverGeneSinglePoint
«override»+breed(genitor1: Solution const &, genitor2: Solution const &, offspring: Solution &)
«dataType»
SOParameters
+width: size_t
+height: size_t
+obstacleCount: size_t
«primitiveType»
fitness_t
fitness_t = double
Chromosome
mData: vector<bool>
+resize(size: size_t)
+read(to_be_determined)
+write(to_be_determined)
+flip(to_be_determined)
Mutator
-mMutationRate: double
+mutationRate(): double {query}
+setMutationRate(rate: double)
«virtual»+mutate(offspring: Solution &)
MutatorChromo
«override»+mutate(offspring: Solution &)
MutatorGene
«override»+mutate(offspring: Solution &)
Solution
-mFitness: fitness_t
+fitness(): fitness_t {query}
+chromosome(): Chromosome&
«virtual»+processFitness()
«virtual»+randomize()
«virtual»+encode()
«virtual»+decode()
«virtual»+clone(): Solution*
Population
+size(): size_t {query}
+set(size: size_t, solutionSample: Solution*)
+operator[](index: size_t): Solution&
+operator[](index: size_t): Solution const & {query}
+processFitness()
+randomize()
+sort()
Canevas
-mWidth: size_t
-mHeight: size_t
+width(): size_t
+height(): size_t
+obstables(): vector<obstacle> const & {query}
+setSize(width: size_t, height: size_t)
+setObstacleCount(count: size_t)
+setup(so: SOParameters const &)
+randomizeObstaclesPosition()
Selector
«virtual»+prepare(population: Population const &)
«virtual»+select(population: Population&): Solution&
SelectorRouletteWheel
#mRankWeight: vector<double>
«override»+prepare(population: Population const &)
«override»+select(population: Population&): Solution&
SelectorUniform
«override»+prepare(population: Population const &)
«override»+select(population: Population&): Solution&
ShapeSolution
«virtual»+draw(to_be_determine) {query}
«override»+processFitness()
«virtual»+collide(obstacle: Obstacle const&): bool {query}
«virtual»+distance(obstacle: Obstacle const &): double {query}
«virtual»+area(): double {query}
«virtual»+perimeter(): double {query}
Geometry
Point
Rect
Size
Random
Obstacle
+randomizePosition(minX, maxX, minY, maxY)
CircleSolution
OrthoRectSolution
PolygonSolution
SelectorRankwise
-mGamma: double
+gamma(): double {query}
+setGamma(gamma: double)
«override»+prepare(population: Population const &)
SelectorTournament
«override»+prepare(population: Population const &)
«override»+select(population: Population&): Solution&


interaction PopulationEngine::evolve
mEngine: GAEngine
mPopulationEngine: PopulationEngine
mGenitor: Population
mSelector: Selector
mCrossover: Crossover
mMutator: Mutator
1 : evolve()
2 : processFitness()
3 : sort()
4 :
5 : processElitism()
6 : processOffsprings()
7 : prepare(population: Population const &)
8 : processOneOffspring(index: size_t)
seq loop
9 : select(population: Population&)
10 : select(population: Population&)
11 : breed(genitor1: Solutions const &, genitor2: Solution const &, offspring: Solution &)
12 : mutate(offspring: Solution &)
13 : swapPopulations()
14 : processStatistics()

Résumé de la notation UML utilisée:

- Le premier diagramme est un diagramme de classe :
  - Un rectangle représente une classe :
    - Les trois sections représentent :
      - en haut, le titre
      - au centre, les attributs (variables)
      - en bas, les opérations (fonctions)
    - Le masquage est illustré par :
      - `-` privé
      - `#` protégé
      - `+` public
    - Les types sont mis après les attributs et les opérations séparé par un : (deux points).
    - Si une opération est italique, ça indique qu'elle est abstraite.
    - Si le titre de la classe est italique, ça indique qu'elle est abstraite.
  - Les lignes représentent les associations :
    - La ligne pleine indique que ces deux classes travaillent ensemble sans que l'une possède l'autre (généralement via une fonction).
    - La flèche blanche indique la notion d'héritage – la flèche pointe de l'enfant vers le parent.
    - Le losange noir indique la composition, c'est-à-dire que la classe du côté où se trouve le losange possède une ou plusieurs instances de la classe reliée. De plus, la cardinalité est le numéro se trouvant de l'autre côté du losange et indique le nombre d'instance que le possesseur possède. L'étoile indique *n* instances. Par exemple, sur le graphique, on peut lire que la `Population` possède *n* `Solution`.
    - Le losange blanc indique l'agrégation. L'agrégation est similaire à la composition mais la durée de vie de la classe liée est indépendante du référent.
  - Les `<< … >>` indiquent des stéréotypes. En UML, on les utilise pour ajouter de l'information. Ici, `virtual` et `override` font référence aux concepts du même nom du langage C++. `dataType` fait référence à une structure et `primitiveType` fait référence à un `typedef` d'un type fondamental.
  - Le rectangle avec le coin supérieur droit plié indique une note donnant des détails sur l'élément lié.
  - UML présente plusieurs contraintes notées entre accolades mais une seule a été utilisée pour illustrée le concept du « lecture seule » (`const`) du C++. La contrainte `{query}`.
- Le deuxième diagramme représente un diagramme de séquence L :
  - L'axe vertical représente le temps.
  - Les rectangles se trouvant tout en haut des lignes verticales pointillées représente des signaux provenant d'instances. Dans notre cas, elles représentent les appels de fonctions des classes concernées.
  - Les lignes horizontales représentent les transferts de signaux (les appels de fonctions). On remarque un numéro séquentiel ainsi que la fonction appelée.

- Les rectangles verticaux superposés aux lignes pointillées représentent la durée d'un appel de fonction.
- Le rectangle intitulé `seq loop` représente une boucle de type `for` ou `while` selon l'implémentation.
- Le schéma réalisé représente la fonction principale de l'algorithme génétique. Malgré sa complexité initiale, on remarque qu'elle est découpée en plusieurs sous tâches utilisant ou non le polymorphisme.

On présente ici une courte description des classes à réaliser. D'abord, les classes reliées à la petite librairie :

- `Solution` : Représente une unique solution de l'algorithme génétique. Cette classe est abstraite et possède 5 fonctions devant être réimplémentée.
  - La fonction `processFitness` représente une partie importante car c'est cette fonction qui, réimplémentée dans une autre classe vous permettra de résoudre le problème adressé.
  - La fonction `randomize` permet d'allouer aléatoirement le chromosome.
  - Les fonctions `encode` et `decode` permettent de réaliser l'encodage et le décodage.
  - La fonction `clone` permet de créer un nouvel objet à partir d'un objet existant. L'objet créé doit être identique à l'objet créateur.
- `Chromosome` : Représente l'encodage de votre solution. C'est avec les données de cette classe que les algorithmes de croisement et de mutation travaillent.
- `Population` : Cette classe n'est qu'un conteneur de `Solution`. Autrement dit, elle offre tous les services reliés à la gestion d'une population (c'est-à-dire d'un ensemble de solutions) sans faire d'algorithmes spécifiques. Elle sert d'interface pour réaliser une tâche à toutes les solutions.
- `PopulationEngine` : Représente l'une des parties les plus importante du projet. C'est cette classe qui possède le moteur principal de l'algorithme génétique, c'est-à-dire du traitement relié à la création d'une nouvelle génération. On remarque plusieurs fonctions privées. Ces fonctions sont toutes utilisées par la fonction `evolve` qui supervise la tâche de créer la nouvelle génération.
- `GAEngine` : Représente la classe mère de l'algorithme génétique. C'est cette classe qui possède les *n* populations (civilisation) et synchronise toutes les opérations.
- `GAParameters` : Représente les différents paramètres requis par l'algorithme génétique.
- `Selector` : Classe abstraite implémentant un algorithme de sélection. La fonction `prepare` prépare les données internes à l'algorithme si nécessaire. Il est tout à fait possible que cette fonction ne fasse rien. La fonction `select` retourne une solution selon sa méthode.
- `SelectorRouletteWheel` : Réalise l'algorithme expliqué en classe. La fonction `prepare` calcule et stock les informations nécessaires à l'algorithme.
- `Crossover` : Classe abstraite réalisant l'algorithme de croisement. La fonction `breed` permet de modifier les valeurs d'une progéniture selon deux géniteurs.
- `CrossoverChromoSinglePoint` : Réalise l'algorithme expliqué dans ce document. Le terme `Chromo` fait référence au fait que le croisement est fait sans considération pour la structure interne du chromosome et que le découpage ne tient pas compte des gènes.
- `Mutator` : Classe abstraite implémentant la mutation. La fonction `mutate` doit effectuer la mutation sur le chromosome de la progéniture passée.

- `MutatorChromo` : Réalise l'algorithme expliqué dans ce document. Encore une fois, le terme `Chromo` fait référence au fait que la mutation est faite sans considération pour la structure interne du chromosome et qu'on ne tient pas compte des gènes
- `SelectorTournament`, `SelectorRankwise`, `SelectorUniform`, `CrossoverGeneSinglePoint` et `MutateGene` : Si vous voulez implémenter ces algorithmes, demander à l'enseignant de vous expliquer ces algorithmes. Toutefois, il est intéressant de mentionner que `SelectorRankwise` utilise cette formule : $w_i = \frac{g-1}{g^{i+1}}$ où w_i est le poids du i^e chromosome trié et g le facteur de correction gamma $g \in [1, \infty$ $centré$ à 2. On remarque que le premier chromosome vaut $\frac{g-1}{g}$.

Les autres classes sont reliées à l'application elle-même :

- `ShapeOptimizer` : C'est la classe qui encapsule l'application à réaliser. C'est la fonction `run` qui possède la « *main loop* » et qui redirige toutes les interactions clavier et écran.
- `Canevas` : Représente l'espace 2d et toutes les informations reliées (taille, obstacles, …).
- `Obstacles` : Représente les points 2d de l'espace auxquels les formes ne doivent pas entrer en collision.
- `ShapeSolution` : C'est la classe qui permet de faire le lien entre votre librairie et votre application. Lorsque votre librairie sera terminée, la réalisation de cette classe représente en fait le seul développement spécifique au problème donné. La fonction draw est très pertinente et pratiquement inévitable. Néanmoins, les quatre autres fonctions virtuelles proposées (`collide`, `distance`, `area` et `perimeter`) ne sont que des suggestions en fonction de votre calcul fait dans `processFitness`.

Les éléments suivants utilisent tous le polymorphisme pour généraliser et modulariser cet algorithme autant que possible :

- `Solution`
- `Selector`
- `Crossover`
- `Mutator`

Il est important de se rappeler que le polymorphisme requiert l'utilisation de pointeur (ou de référence) pour référencer les objets polymorphiques créés.

IMPORTANT : ce design n'est pas complet. Il manque plusieurs éléments que vous devez déterminer en équipe. Néanmoins, l'infrastructure en place vous donne une assise solide. Vous pouvez ajouter quoique ce soit en autant que vous respectiez la logique en place.

Cette conception n'est pas optimale et correspond à un exemple pédagogique visant la mise en pratique de la programmation orientée objets et dont la réalisation soit réaliste en classe.

Sachez que pour une meilleure performance, le polymorphisme dynamique utilisé ici est une mauvaise stratégie. Il aurait été préférable d'utiliser des pointeurs de fonctions (pas nécessairement orienté objets) ou des notions plus avancées comme le polymorphisme statique que permet les *template* (le `CRTP` aurait été très intéressant ici).

Ajout personnel

Comme pour les laboratoires précédents, 20% du projet est attribué à l'ajout d'une fonctionnalité supplémentaire.

Voici quelques suggestions :

- ajoutez des stratégies différentes pour la sélection, le croisement et la mutation;
- tentez de résoudre une forme à géométrie variable;
- tentez de résoudre un problème différent avec votre petite librairie.

Il est fortement encouragé d'en discuter avec l'enseignant.

Outils fournis

Librairie d'interface

Vous pouvez utiliser la librairie Console ou BAI déjà utilisée. Veuillez utiliser les fichiers donnés précédemment.

Librairie polygone

Vous avez à votre disposition une petite librairie réalisant certaines manipulations liées à un polygone. Vous pouvez l'utiliser pour ce projet. Néanmoins, si vous réussissez le projet sans son usage, des points bonis vous seront donnés. Pour avoir les points bonis, il ne suffit pas d'utiliser une autre librairie, il suffit de s'approprier les connaissances liées à ce type de problématique et d'en faire une implémentation adéquate.

Stratégie d'évaluation

L'évaluation se fera en 2 parties. D'abord, l'enseignant évaluera le projet remis et assignera une note de groupe pour le travail. Ensuite, chaque équipe devra remettre un fichier Excel dans lequel sera soigneusement reportée une cote représentant la participation de chaque étudiant dans la réalisation du projet. Cette évaluation est faite en équipe et un consensus doit être trouvé.

Une pondération appliquée sur ces deux évaluations permettra d'assigner les notes finales individuelles.

Ce projet est long et difficile. Il est conçu pour être réalisé en équipe. L'objectif est que chacun prenne sa place et que chacun laisse de la place aux autres.

Ainsi, trois critères sont évalués :

- **participation** (présence en classe, participation active, laisse participer les autres, pas toujours en train d'être sur Facebook ou sur son téléphone, concentré sur le projet, pas en train de faire des travaux pour d'autres cours …)
- **réalisation** (répartition du travail réalisé : conception, modélisation, rédaction de script, documentation …)
- **impact** (débrouillardise, initiative, amène des solutions pertinentes, motivation d'équipe …)

Remise

Vous devez créer un fichier de format `zip` dans lequel vous insérez :

- votre schéma de conception `schema.pdf`
- votre solution Visual Studio **bien nettoyée** `solution.zip`
- le fichier Excel rempli sur la participation active des membres du groupe `evaluation.xlsx`

Vous devez remettre votre projet une seule fois sur Lea après avoir nommé votre fichier :

`NomPrenomEtudiant1_NomPrenomEtudiant2[_NomEtudiantN].zip`