PRO3600: Mars Lander

Augustin Bresset | Zacharie March

January 2024

FIGURE 1 – Logo TSP

Table des matières

1 Introduction

1.1 Présentation du problème : Mars Lander

Mars Lander est un problème d'optimisation proposé sur la plateforme *CodinGame* s. d. Ce jeu consiste en le contrôle d'un vaisseau spatial et de ses caractéristiques (position, vitesse, puissance des moteurs, angles de rotation...) afin de le faire atterrir en toute sécurité et en douceur sur Mars sur une surface plane, quelle que soit sa position de départ. L'idée est donc de trouver une trajectoire fonctionnelle.

1.1.1 Environnement

La surface de Mars est représenté localement par une suite continue de segments. Parmis ces segments un et un seul est rigouresement vertical, celui-ci définit la zone d'atterissage que la navette doit viser.

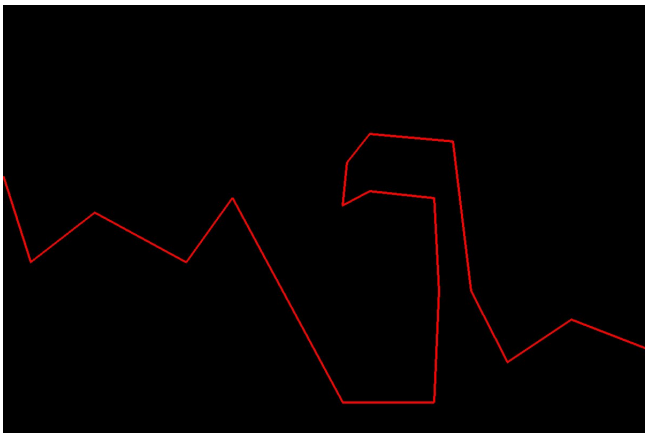

FIGURE 2 – Exemple de carte du problème

1.1.2 Espace des Etats

L'état du vaisseau est représenté par un vecteur de 7 valeurs :
— x : position horizontale
— y : position verticale
— $hSpeed$: vitesse horizontale
— $vSpeed$: vitesse verticale
— $fuel$: quantité de carburant restante
— $rotate$: angle de rotation
— $power$: puissance des moteurs
Certains de ces paramètres est borné par une valeur minimale et maximale que voici :
— x : $[0, 7000]$
— y : $[0, 3000]$
— $fuel$: $[0, \infty[$
— $rotate$: $[-90, 90]$
— $power$: $[0, 4]$

1.1.3 Espace des Actions

Toutes les secondes, en fonction des paramètres d'entrée (position, vitesse, fuel, etc.), le programme doit fournir le nouvel angle de rotation souhaité ainsi que la nouvelle puissance des fusées de Mars Lander. Mais les commandes de puissance des fusées et de l'angle de rotation sont bornés par les valeurs suivantes qui forme l'espace des actions :
— rotation : $[-15, 15]$
— puissance : $[-1, 1]$

1.1.4 Conditions d'atterrissage

Les conditions d'atterissage doivent être représentées :
— atterrir sur un terrain plat
— atterrir en position verticale (angle d'inclinaison = 0°)
— la vitesse verticale doit être limitée ($\leq 40 m.s^{-1}$ en valeur absolue)
— la vitesse horizontale doit être limitée ($\leq 20 m.s^{-1}$ en valeur absolue)

1.2 Algorithme Génétique

Afin de résoudre ce problème, nous avons choisi d'utiliser une approche heuristique, les algorithmes génétiques.

Les algorithmes génétiques sont des algorithmes d'optimisation stochastique inspirés de la théorie de l'évolution naturelle. Ils sont basés sur le principe de **sélection naturelle**.

Dans un premier temps on génère une **population** de solution aléatoire, puis on évalue la qualité de chaque solution. On sélectionne fait ensuite évoluer cette population à travers des phénomènes semblables à la sélection naturelle tel que la **sélection**, la **reproduction** et la **mutation**. On répète ce processus jusqu'à ce qu'une solution satisfaisante soit trouvée. Dans notre problème, une solution est définie par une trajectoire, c'est à dire une suite de commande à effectuer par le vaisseau.

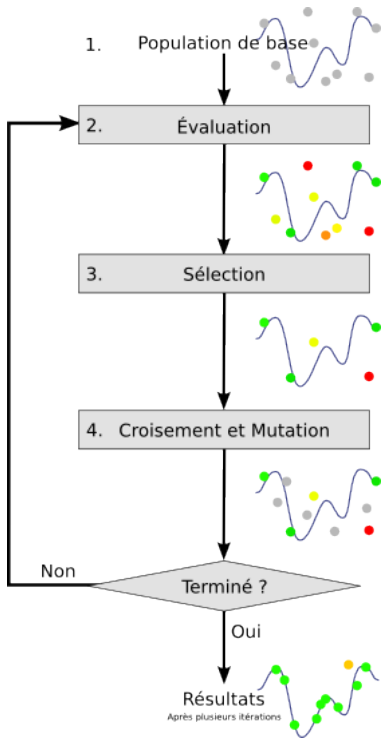

FIGURE 3 – *Schéma d'un algorithme génétique* s. d.

Le calcul du score peut dépendre de nombreux paramètres, dans notre cas on a choisi de prendre en compte les paramètres suivants :

— La **distance** au sol qui sépare la zone de collision et d'atterissage
— Les **vitesses** verticale et horizontale
— L'**angle** d'inclinaison du vaisseau à l'atterissage

1.3 Objectif

Afin de bien étudier le problème et l'influence des hyperparamètres de la solution , nous avons décidé de créer une interface graphique. Pour cela on à définit plusieurs points afin de mener à bien notre projet :

— Création d'une **interface graphique** sur pygame
— Mise en place d'une **interface client** permettant de piloter la navette
— Calcul de **solution** notamment à l'aide d'algorithme génétique

1.3.1 Cahier des charges

Développer une applicattion avec interface graphique. Les fonctionnalités attendues sur l'application sont les suivantes :

— Choix de la solution à utiliser
— Choix de la carte à utiliser
— Exécution de la simulation
— Adapter l'affichage en fonction de la solution

1.4 Organisation du projet

1.4.1 Répartition des tâches

En claire la répartition des tâches est la suivante :

— **Augustin Bresset**
Développement de l'environnement et des solutions
— **Zacharie March**
Développement de l'interface graphique et de l'interface client

Dans les deux cas, un apprentissage de la librairie pygame est nécessaire. Que ce soit pour la création de la solution "manuelle" (contrôle par l'utilisateur) ou pour la création de l'interface graphique.

1.4.2 Outils de travail

Afin de communiquer, nous nous reposons pour la communication active sur plusieurs outils, étant deux, nous n'étions pas restreint par le besoin de créer un groupe sur un outil de communication. Pour la commun passive nous avons créée une organization sur Github dans laquelle on peut trouver deux repertoire :

— **backend** : Contient le code source du projet
— **meta** : Contient la documentation du projet (et ce livrable)

1.4.3 Organisation du code

Afin de faciliter la lecture du code ainsi que le développement, nous avons séparer en plusieurs modules le code source :

— **environment** : Contient le code source de l'environnement
— **gui** : Contient le code source de l'interface graphique
— **solution** : Contient le code source des solutions
— **utils** : Contient le code source des fonctions utilitaires

2 Développement

2.1 Architecture du projet

Afin de mener notre projet à bien, nous avons décidé de découper le code en plusieurs modules.

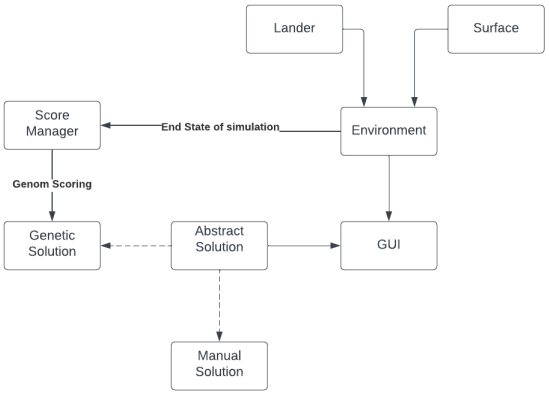

FIGURE 4 – Architecture du projet

Comme vue dans la figure 4, le projet est composé de plusieurs modules :

— **Environment** : Module permettant de gérer l'environnement
— **Gui** : Module permettant de gérer l'interface graphique
— **Solution** : Module permettant de gérer les solutions
— **Score** : Module permettant de gérer le scoring

Et à cela s'ajoute un module **Utils** qui contient des fonctions utilitaires.

2.1.1 Environnement

L'environnement est composé de plusieurs classes :

— **Environment** : Objet permettant de gérer les interactions entre les entités et l'environnement
— **Surface** : Classe permettant de gérer la surface de Mars
— **Lander** : Classe permettant de gérer le vaisseau
— **Action** : Classe permettant de gérer les actions du vaisseau
— **Utils** : Ensemble de fonctions utilitaires et constantes
— **Entity** : Classe abstraite permettant de gérer les entités
— **Constants** : Classe permettant de gérer les constantes

2.1.2 Gui

— **Gui** : Classe permettant de gérer l'interface graphique
— **Menue** : Fonction permettant l'affichage du menue
— **Utils** : Ensemble de fonctions utilitaires et constantes

2.1.3 Solution

Une solution est définie par une classe abstraite **AbstractSolution** qui permet de définir les méthodes communes à toutes les solutions. On a ensuite deux types de solutions :
— **ManualSolution** : Solution manuelle permettant de piloter le vaisseau à l'aide du clavier
— **GeneticSolution** : Solution génétique permettant de piloter le vaisseau à l'aide d'un algorithme génétique

2.1.4 Score

Le score est calculé à l'aide de la classe **ScoreManager** qui permet d'attribuer un score à une trajectoire. Pour cela il va venir calculer différents résultats sur l'environnement avant qu'il ne soit réinitialisé avant une prochaine simulation.

2.1.5 Utils

Le module **Utils** contient des fonctions utilitaires et des constantes utilisées dans les autres modules et notamment les classes **Point** et **Segment**.

Point
La classe Point permet de représenter un point dans un espace à deux dimensions. Il lui ait associé des fonctions permettant de calculer la distance et de gérer les égalités entre deux points. On a considérer ici que deux points étaient égaux si leurs coordonnées étaient assez proches.

Segment
La classe Segment permet de représenter un segment dans un espace à deux dimensions. Sa méthode la plus utile est la méthode **collision** qui à l'aide d´une fonction *CCW* s. d., CCW permet de savoir si deux segments se croisent.

En considérant un instant de trajectoire comme étant un segment, on peut vérifier si il ne rentre pas en collision avec la surface grâce à cette méthode.

2.2 Tests

Afin de vérifier le bon fonctionnement de notre code, nous avons mis en place des tests. Ces tests se divisent en trois catégorites qui composent ces parties. On va voir dans un premier temps les **tests unitaires** qui permettent de tester les fonctions et méthodes importantes et de manières indépendantes au reste du code. Il enn suivra les **tests d'intégration** qui permettent de tester le bon fonctionnement des modules entre eux. Et enfin les **tests fonctionnels** qui permettent de tester le bon fonctionnement de l'application dans son ensemble.

Nous utiliserons la librairie **unittest** pour l'écriture de nos tests.

2.2.1 Tests unitaires

Environnement Afin de tester l'environnement, nous avons récuperer les réponses dynimiques de l'environnement présent sur *CodinGame* s. d., CodinGame et nous avons comparer les états à chaque instant à ceux calculer par notre environnement. De plus on a fait des tests pour vérifier que les collisions avec la surface étaient bien détectées. Et notamment que les collisions avec avec le site d'atterissage est bien détecté et que si le vaisseau sort de la carte, ce soit bien détecté par l'environnement.

Score Les fonctions étant simplement des fonctions mathématiques dépendant d'un paramètre du problème, nous n'avons pas jugé nécessaire de faire des tests unitaires excepté pour la fonction qui permet de calculer la distance entre la zone de collision et la zone d'atterissage. En effet, ce n'est pas une simple distance euclidienne, mais plutot la distance en suivant la surface de l'environnement. On a pu vérifier le bon fonctionnement de cette fonction à l'aide de cas caculé à la main. Et les trois tests nous permettent de nous laisser penser que :
— La distance est bien décroissante si l'on se dirige vers la zone d'atterissage
— La distance est bien calculée

Algorithme Génétique Ces tests unitaires sont destinés à évaluer le fonctionnement de certaines fonctionnalités de l'algorithme génétique mise en œuvre.

On test tout d'abord la génération d'une population. Il crée une population de 10 individus, chacun ayant 100 gènes du type "ActionChromosome". Les assertions comparent la longueur de la population, la longueur des gènes du premier individu, et s'assurent que la longueur des gènes est la même pour tous les individus, tout en vérifiant qu'il y a des différences entre les gènes des deux premiers individus.

Puis on test la **cumulative wheel**, ce test évalue la fonction qui génère une roue cumulative pour la sélection d'individus. Une population de 10 individus est créée avec des scores aléatoires. La population est triée en fonction des scores, puis la roue cumulative est générée. Les assertions vérifient que la longueur de la roue cumulative est correcte, et que chaque élément de la roue est une paire d'individus de type "ActionChromosome", avec des identifiants différents.

Enfin on vérifie **la sélection** du/des meilleur individu dans une population.Ce test évalue la fonction de sélection du meilleur individu dans une population. Une population de 10 individus est créée, chaque individu ayant un score égal à son index. La population est triée en fonction des scores, et la meilleure solution est sélectionnée. Les assertions comparent le score de la meilleure solution avec l'index attendu, soit 9 dans ce cas. En résumé, ces tests garantissent le bon fonctionnement de différentes parties de la solution génétique, y compris la génération de populations, la création de roues cumulatives pour la sélection, et la sélection du meilleur individu dans une population.

2.3 Problèmes rencontrés

Tout au long du projet, on a essayé de garder une architecture de code permettant une meilleure répartition des processus tout en gardant une cohérence dans le code.

Nous voulions diviser les repertoires github entre le frontend et le backend, mais nous avons finalement pas rencontrés ce besoins étant donné que l'on travaillait sur le même langage.

N'ayant pas de deadline claire, on a pu prendre le temps de bien réfléchir à l'architecture du code et à la répartition des tâches mais ca nous a aussi conduit a prendre plus de temps que nécessaire pour certaines tâches. Finalement, on s'est remis à faire des deadlines pour pouvoir avancer plus rapidement.

2.4 Manuel utilisateur

Afin de lancer le programme, il faut tout d'abord installer les dépendances du projet présentes dans le fichier **requirements.txt**. Pour cela on peut utiliser la commande suivante :

```
pip install -r requirements.txt
```

Une fois les dépendances installées, on peut lancer le programme à l'aide de la commande suivante :

```
python src/launcher.py
```

Références

CodinGame (s. d.). URL : https://www.codingame.com/multiplayer/optimization/mars-lander.
CCW (s. d.). URL : https://bryceboe.com/2006/10/23/line-segment-intersection-algorithm/.
Schéma d'un algorithme génétique (s. d.). URL : https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique.