

Rapport d'application

SAE – RPG

Ewen GILBERT, Noé COSTE – INF1A

06/06/2023



Plan

- I. Présentation de l'application
- II. Qualité de développement
- III. Conception de l'application
- IV. Bilan et conclusion
- V. Annexes

I. Présentation de l'application

L'application que nous avons réalisée a pour but de mesurer la durée de vie d'un jeu vidéo de type RPG, à partir de différents critères. Le but est alors de réaliser différents algorithmes permettant de calculer la durée de vie du jeu selon des critères précis avec des contraintes de quêtes comme l'expérience du joueur ou les préconditions de quêtes.

Les différentes quêtes qui composent un scénario sont rangées dans des fichiers textes qui doivent être lus et interprétés par l'application. Une fois cette lecture réalisée, les algorithmes peuvent être utilisés.

L'utilisateur peut choisir parmi les scénarios disponibles dans le menu de sélection, puis choisir sa méthode de recherche de solutions.

Il y a trois grands critères de recherche :

Le type de chemin à effectuer :

- **Efficace** : Le but est de trouver le chemin le plus efficace pour finir le jeu, en réalisant la quête finale sans avoir à finir toutes les quêtes
- **Exhaustive** : Le but est de trouver un chemin permettant de finir toutes les quêtes du jeu avant de faire la quête finale.

Le critère de recherche :

- **Gloutonne** : Le but est de prendre la quête la plus proche possible à chaque fois, sans vraiment réfléchir sur l'ordre de réalisation
- **En fonction de la durée** : Le but est d'avoir le chemin le plus court possible en termes de durée et d'arriver à la quête finale le plus vite possible.
- **En fonction du nombre de quêtes** : Le but est d'avoir le chemin le plus court possible en termes de nombre de quêtes et d'arriver à la quête finale en réalisant le moins de quêtes possible.

- **En fonction de la distance parcourue** : Le but est d'avoir le chemin le plus court possible en termes de distance parcourue et d'arriver à la quête finale en parcourant le moins de distance possible.

L'ordre des résultats :

- **Les meilleures solutions** : L'application affiche les meilleures solutions en fonction du critère de recherche choisi.
- **Les pires solutions** : L'application affiche les pires solutions en fonction du critère de recherche choisi.

Une fois tous les critères sélectionnés dans le menu, l'utilisateur peut **sélectionner le nombre de solutions** qu'il souhaite afficher, puis appuyer sur le bouton "Valider" pour afficher les solutions dans le tableau juste en dessous.

Toutes les solutions sont affichées dans le tableau avec les informations suivantes :

- Le n° de la solution dans la liste des solutions (la solution n°1 correspond à la meilleure solution)
- La suite de quêtes à réaliser pour cette solution
- La durée totale de la solution
- L'expérience accumulée à la fin de la solution
- La distance parcourue à la fin de la solution
- Le nombre de quêtes réalisées



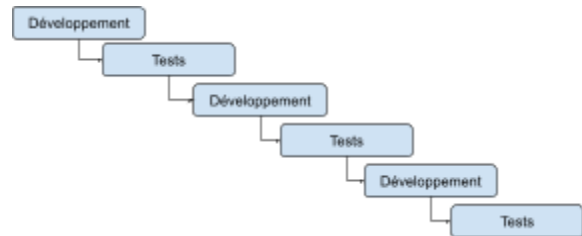
Lien vers la maquette de l'application : (réalisée sur figma.com)

<https://www.figma.com/proto/xDSoekZpSLfiQL3qUnvleY/Untitled?type=design&node-id=1-2&scaling=min-zoom&page-id=0%3A1>

II. Qualité de développement

A. L'organisation du travail

Le projet a été réalisé dans un **cycle de vie en cascade itératif**. Cela veut dire que nous avons choisi de réaliser notre développement par étapes, en alternant le développement et la réalisation de tests sur ce que nous venons de développer.



Cela permet alors de réaliser un projet structuré et testé au fur et à mesure de sa réalisation. Cette méthode est selon nous la plus appropriée pour un projet comme celui-ci, car il nous permet de s'assurer que l'application fonctionne correctement tout au long du développement. Celle-ci apporte également une **flexibilité** aux changements de dernière minute et **réduit les risques** de problèmes de calcul de solutions qui pourraient être rencontrés.

Du côté de la répartition des tâches, nous avons décidé de partager les tâches de la manière suivante :

Noé :

- Algorithmes de calcul des solutions.
- Tests des algorithmes et optimisation.
- Gestion et interprétation des scénarios et des quêtes.

Ewen :

- Réalisation de l'IHM et mise en relation de l'architecture MVC.
- Lecture, récupération et interprétation des données des solutions.
- Rédaction des rapports de tests et du rapport de l'application.

B. L'utilisation d'outils


Lors du processus de réalisation de ce projet, nous avons été amenés à utiliser [différents outils](#) qui nous ont aidés à réaliser le projet tel qu'il est aujourd'hui.

Pour commencer, nous avons pu nous organiser avec un outil nommé [Notion](#), qui est une plateforme permettant de [mettre en commun des documents et organiser des tâches](#) de manière pratique et efficace. Nous l'avons déjà utilisé dans le passé pour la partie gestion de projet, car elle permettait de noter et d'organiser nos tâches efficacement et proprement. Avec cette base créée lors de la partie de gestion de projet, nous avons pu nous organiser efficacement dans notre travail, en utilisant par exemple des fonctionnalités comme le tableau Kanban ou la liste de tâches.

Ensuite, comme expliqué plus tôt, nous avons utilisé le [site de conception Figma](#) qui a notamment servi lors de la réalisation de la SAE d'organisation du voyage au premier semestre, et qui permet de réaliser efficacement les maquettes d'un site Internet ou d'une interface comme celle de notre application. Cet outil nous a permis de visualiser notre interface avant même sa création, et analyser les besoins de celle-ci avant sa création plus efficacement, un véritable support qui nous a permis d'être plus assidus dans la création de l'interface de l'application.

Pour le versionnage de notre projet, nous avons choisi le [gestionnaire Git en ligne GitHub](#), nous permettant de mettre en commun notre travail et de le versionner de la meilleure façon possible, en créant diverses branches permettant de rendre notre travail homogène. Git et par extension GitHub permet de suivre efficacement l'avancée du projet, tout en conservant une trace de son avancée et de ses différentes versions. Cet outil est essentiel dans tout projet, surtout dans un projet en collaboration comme celui-ci, il est donc impératif de ne pas le négliger et de l'utiliser de la meilleure manière possible.

Quant aux tests, nous avons utilisé le [framework de test unitaire JUnit](#), qui permet de réaliser des tests unitaires en java. Celui-ci permet d'exécuter des tests de manière automatisée et structurée, permettant de garantir la fiabilité de



l'application tout au long du développement. Nous nous en sommes donc servi pour réaliser les [tests unitaires de notre application en boîte noire](#). Cela nous permet de comparer le résultat attendu avec le résultat obtenu et donc de savoir si notre programme fonctionne correctement. JUnit est un outil extrêmement utile dans la réalisation de nos tests et favorise une bonne qualité de développement, l'automatisation des tests, et la détection précoce des erreurs, contribuant ainsi à la robustesse et à la maintenabilité de notre application.

C. Analyse de notre méthode de développement

Comme dit plus tôt, nous avons opté pour un développement avec un [cycle de vie en cascade itératif](#), qui permet de rendre le déroulé de notre projet plus fiable. De plus, l'utilisation de tous les outils introduits juste avant nous a aidé dans la réalisation du projet. Grâce à celle-ci, nous avons pu nous mettre efficacement au travail et chaque outil nous a permis à sa manière de nous aider dans le processus de développement de l'application.

Pour ce qui est de [l'aspect collaboratif](#), nous avons commencé par réfléchir ensemble sur les diverses solutions et les moyens pour pouvoir satisfaire les exigences du projet et ses complexités. Nous nous sommes concertés afin de trouver le meilleur moyen de trouver une méthode algorithmique permettant de calculer les solutions efficacement et de manière optimisée dans chaque cas demandé, que ce soit une recherche efficace ou exhaustive. Toute cette organisation nous a permis d'être efficace dans notre travail, que ce soit à deux ou seuls, nous avons pu réaliser nos tâches correctement et efficacement. Nous avons pu communiquer efficacement et nous organiser avec le versionnage Git et l'utilisation de Notion et Figma, pour visualiser et définir les objectifs du projet, et répondre aux besoins de celui-ci.

III. La conception

A. Les diagrammes

DIAGRAMME DES CLASSES DE HAUT NIVEAU

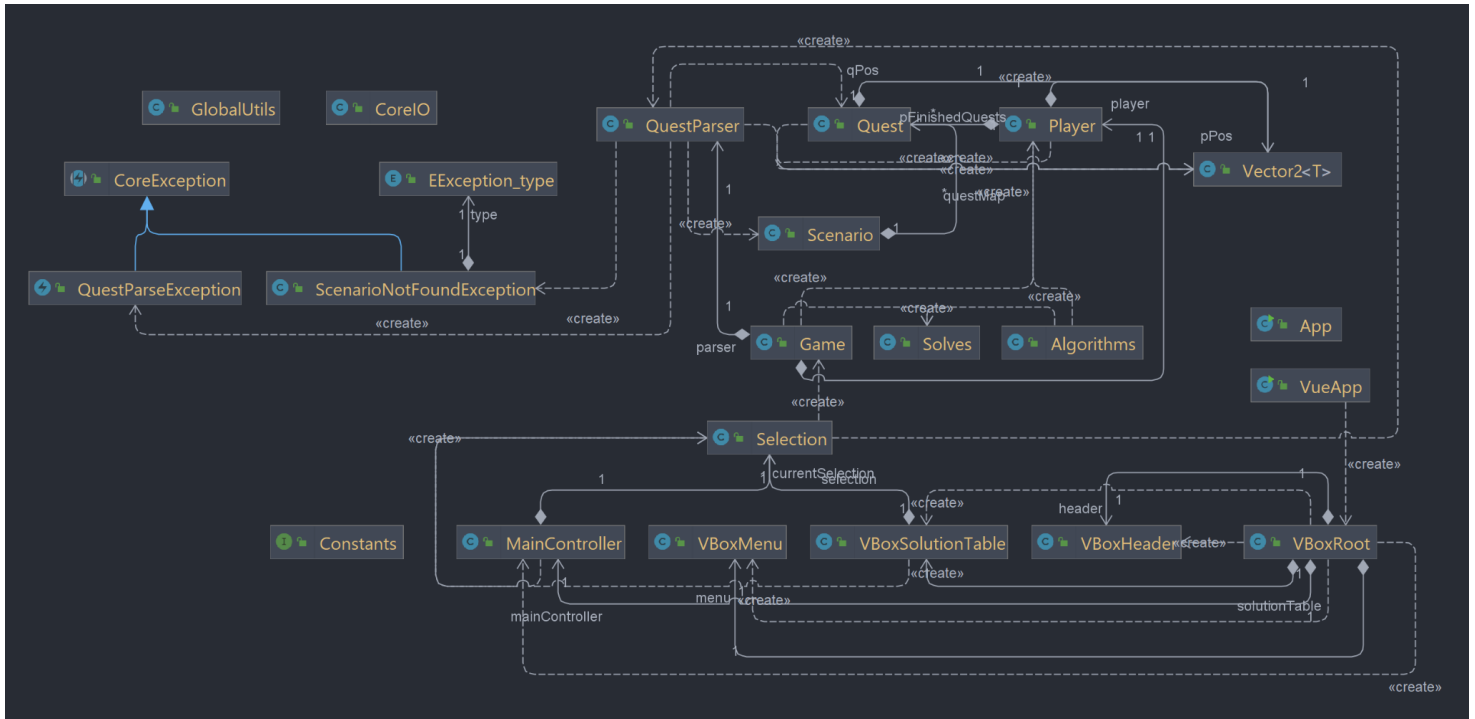


DIAGRAMME DES CLASSES - VUE

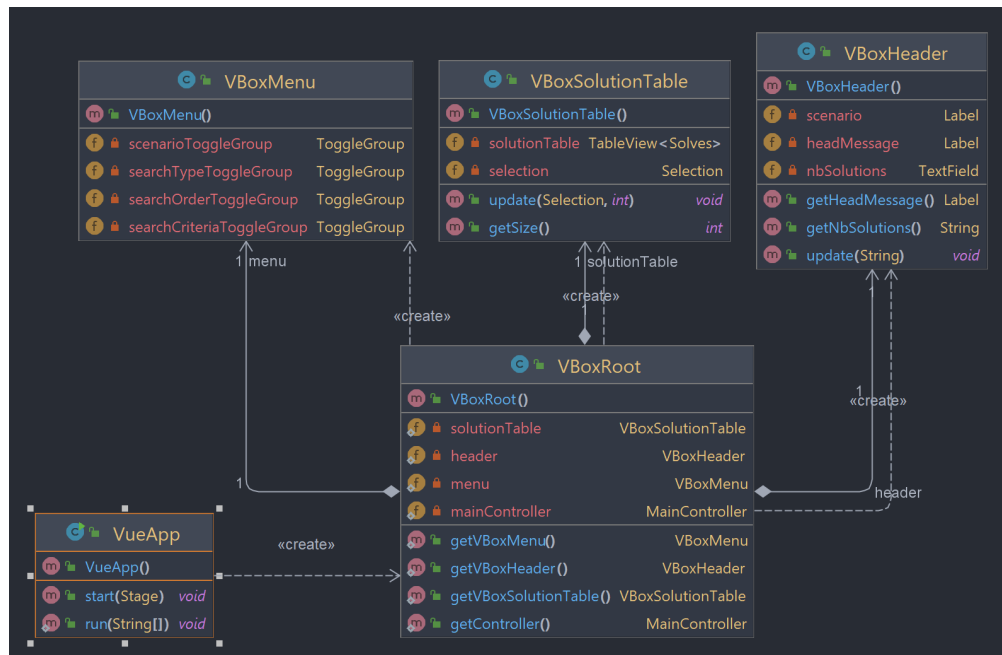


DIAGRAMME DES CLASSES – CONTRÔLEUR

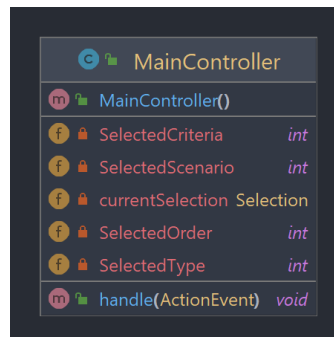
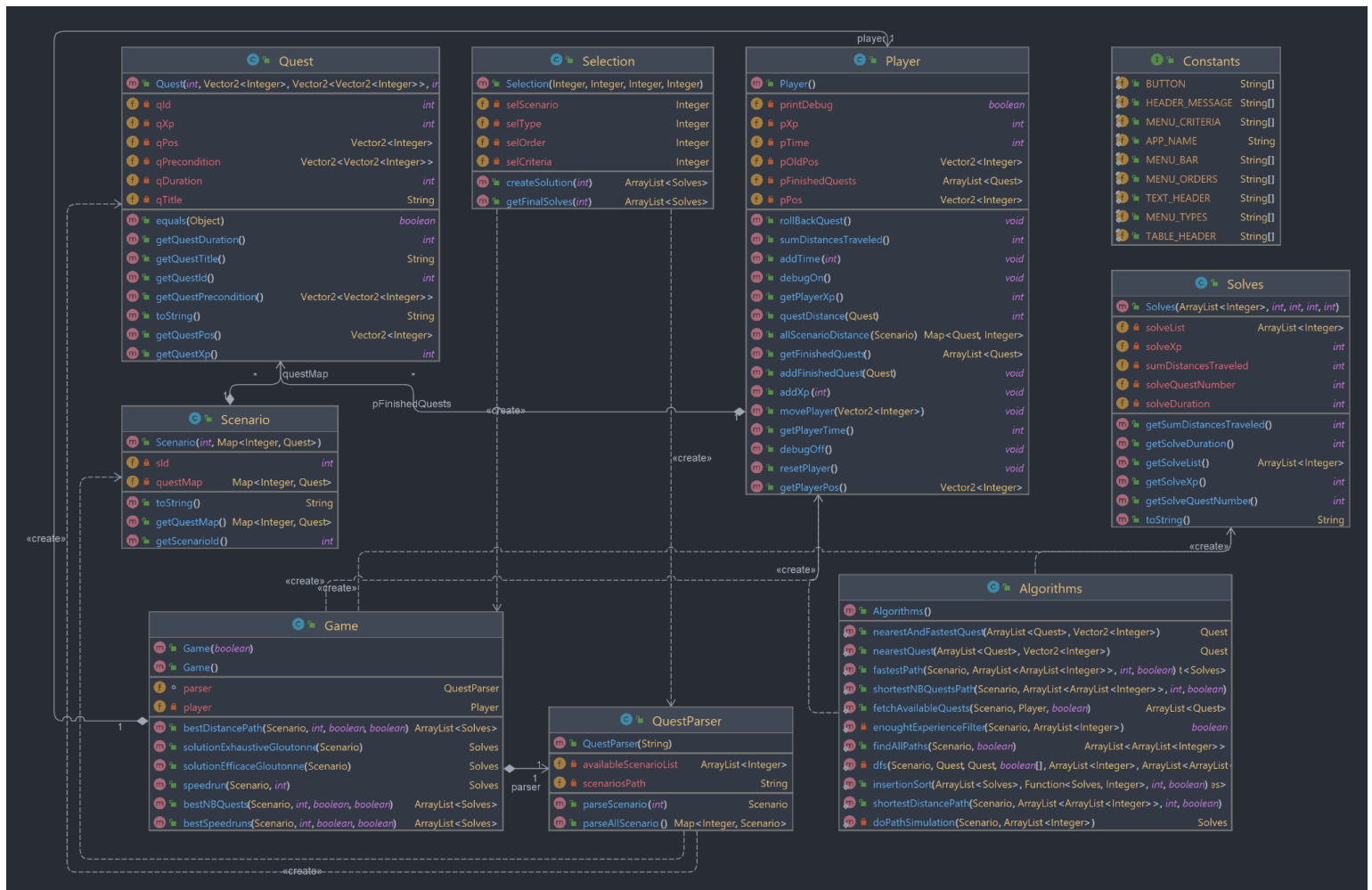


DIAGRAMME DES CLASSES – MODÈLE (PACKAGES MODÈLE ET CORE)



B. Les structures de données

1) La structure des scénarios

Pour la gestion des scénarios, nous avons utilisé une classe `Scenario` qui nous permet de stocker les données des scénarios efficacement et de pouvoir les réutiliser. Pour initialiser les scénarios, nous utilisons la classe `QuestParser`, qui va nous permettre de créer des objets de la classe scénario à partir des fichiers textes de scénarios fournis dans le sujet du projet. Chaque scénario contient deux champs, un champ `sld`, qui contient l'id du scénario, venant du nombre contenu dans les noms des fichiers. Le second champ, `questMap`, est une Map de quêtes qui prend pour clés des entiers qui représentent l'id des quêtes.

2) La structure des quêtes

Chaque scénario contient donc une Map d'objets de la classe `Quest`. La classe `Quest` est l'une des classes les plus importantes dans toute la structure du projet. En effet, c'est elle qui va être utilisée par les différents algorithmes pour interpréter les données des quêtes contenues dans les scénarios. Les objets de la classe `Quest`, contenus dans des objets de la classe `Scenario` sont tous instanciés dans la classe `QuestParser`, qui va éparpiller les données données dans le fichier dans les différents champs de la classe. Ces champs sont les suivants :

- `qId` représente l'id de la quête, qui pourra être utilisée dans les algorithmes comme élément repérant la quête.
- `qPos` représente la position de la quête, et est stockée sous forme d'un objet de la classe `Vector2` contenant des entiers, représentant une position en x et y. `Vector 2` est tiré des tuples contenus dans les lignes de quêtes du fichier texte.
- `qPrecondition` représente les préconditions de la quête, est stockée sous forme d'un objet de la classe `Vector2` contenant lui-même un objet de la classe `Vector2`, contenant des entiers, représentant les conditions de la quête.
- `qDuration` est le champ représentant la durée de la quête et est utilisé dans les algorithmes pour les calculs de temps permettant de déterminer le choix à suivre ou le temps total à retourner.

- **qXp** est le champ représentant l'expérience que va donner la quête au joueur, pour toutes les quêtes, sauf pour la quête 0 où c'est l'expérience requise pour la quête. Il est utilisé dans les algorithmes pour les calculs de temps permettant de déterminer le choix à suivre ou l'expérience totale à retourner.
- **qTitle** est utilisé pour afficher le titre de la quête, celui-ci est seulement utilisé dans la sortie en lignes de commandes.

3) La structure du joueur

Le joueur est un élément primordial de notre projet, c'est lui qui va faire les déplacements, accomplir les quêtes, et finir le jeu. Il va être le pilote de tous les algorithmes, et est donc composé de nombreux champs, et de nombreuses méthodes. La classe est composée de plusieurs champs :

- **pFinishedQuests** est une ArrayList de quêtes contenant la liste des quêtes terminées par le joueur. Celle-ci est par exemple utilisée dans le choix de la prochaine quête que doit réaliser le joueur par les algorithmes, en ne prenant que les quêtes qui ne sont pas contenues dans cette ArrayList. Elle est aussi utilisée dans l'affichage du nombre de quêtes terminées dans le tableau de solutions de l'application
- **pXp** est un entier qui est utilisé dans le choix des quêtes et les calculs pour comptabiliser l'expérience acquise par le joueur, afin de savoir si celui-ci a la possibilité de réaliser la dernière quête demandant un certain nombre de points d'expérience
- **pPos** est un champ contenant un objet de la classe Vector2 qui gère les informations concernant la position du joueur. Cette position permet de calculer la distance entre le joueur et les quêtes afin de choisir quelle quête réaliser pour chaque algorithme.
- **pOldPos** est un champ contenant un objet de la classe Vector2 qui gère la dernière position du joueur lorsqu'il est déplacé. Ce champ est utilisé dans l'algorithme calculant tous les chemins possibles pour le joueur car il permet au joueur de retourner sur son ancienne position.

- `pTime` est un champ contenant un entier utilisé dans la comptabilisation du temps, utilisé dans l'affichage du temps dans l'interface et dans le tri de solutions calculées par les algorithmes.
- `printDebug` est un champ booléen utilisé pour le debug, et utilisé dans une condition permettant d'afficher ou non des informations complémentaires dans la console.

4) La structure des sélections

La classe sélection est la classe appelée quand l'utilisateur fait ses choix dans l'interface de l'application. Elle récupère quatre paramètres qui représentent les choix dans les quatre menus de la barre de menus en haut de l'interface. Ces champs sont utilisés pour appeler la bonne méthode avec les bons paramètres qui donneront les résultats désirés par notre utilisateur.

Tout d'abord, un objet de la classe Selection va être instancié à chaque modification des attendus de l'utilisateur dans la barre de menus. On va alors remplir les champs suivants :

- `selScenario` contient le numéro du scénario qui va concerner la recherche de solutions, l'utilisateur a le choix entre tous les scénarios présents dans le répertoire des scénarios.
- `selType` contient un entier entre 0 et 1 indiquant quel type de recherche est choisi par l'utilisateur. Si le champ est à 0, l'utilisateur choisit une recherche de solution efficace, si le champ est à 1, l'utilisateur choisit une recherche de solutions exhaustive.
- `selCriteria` contient un entier entre 0 et 3 indiquant quel critère de recherche est choisi par l'utilisateur. Si il est à 0, c'est une recherche de solutions gloutonnes, à 1, une recherche de solution par rapport à la durée, à 2, par rapport au nombre de quêtes effectuées, et à 3, par rapport à la distance parcourue par le joueur. Les méthodes correspondantes sont appelées en fonction de la valeur donnée grâce à différentes conditions.

- `selOrder` contient un entier entre 0 et 1 indiquant l'ordre de recherche souhaité. Si il est à 0, on souhaite une recherche des meilleures solutions, si il est à 1, on recherche les pires solutions.

Il faut noter qu'il y a 5 méthodes algorithmiques pour le calcul des solutions, deux classes sont dédiées aux approches gloutonnes efficaces et exhaustives, et les trois autres correspondent aux trois autres critères de recherche (durée, distance, quêtes). Elles prennent en paramètre deux informations en plus du scénario concerné et du nombre de solutions. Ces paramètres sont les paramètres booléens `isExhaustive` et `worthFilter` qui donnent l'information aux méthodes pour le type de recherche et l'ordre de recherche.

5) La structure des solutions

Les solutions sont le résultat de tout notre travail jusqu'ici. La classe `Solves` contient des champs qui stockent les données des solutions créées. Dans notre application, les solutions sont rangées dans une `ArrayList` utilisée par l'interface et plus particulièrement le tableau créé dans `VBoxSolutionTable` qui va afficher toutes les solutions. La classe dispose de différents champs :

- `solveList` contient une `ArrayList` d'entiers qui va contenir la liste des quêtes à réaliser pour arriver à la solution donnée. Elle est utilisée dans l'affichage de la solution dans le tableau de l'interface et représente la solution en elle-même.
- `solveDuration` est un champ contenant un entier représentant la durée totale de la solution, permettant de connaître les meilleures solutions en termes de durée.
- `solveXp` est un champ contenant un entier représentant la totalité de l'expérience accumulée au cours du parcours du joueur. Ce champ est utilisé dans l'affichage des solutions.
- `solveQuestNumber` contient un entier représentant le nombre de quêtes réalisées au cours du parcours et permettant de connaître les meilleures solutions en termes de nombre de quêtes réalisées.

- `sumDistancesTraveled` contient un entier représentant la distance totale parcourue tout au long du parcours du joueur et permet de connaître les meilleures solutions en termes de distance parcourue.

C. Les stratégies algorithmiques

En ce qui concerne les algorithmes, chaque calcul de solutions suit le même schéma de fonctionnement. Tout d'abord, elles prennent en entrée le nombre de solutions souhaitées, puis deux booléens. Le premier booléen permet de savoir si c'est une solution efficace ou exhaustive qui est souhaitée. Le second booléen permet de savoir si on veut les meilleures ou les pires solutions.

A partir de ces informations, la première étape est de générer tous les chemins possibles du scénario à l'aide d'un algorithme de recherche en profondeur. Cet algorithme permet d'explorer efficacement toutes les possibilités et tous les nœuds du graphe représentant les quêtes entre elles tout en gardant un ensemble cohérent aux différentes dépendances des quêtes entre elles. Cet algorithme vu en cours de Graphes lors du second semestre compose la première étape importante de la recherche des solutions.

En même temps que cette génération, une simulation progressive en temps réel d'un objet de la classe joueur est réalisée. Le joueur change ses données au même rythme, pour enfin retourner une liste d'objets de la classe Solved à chaque solution trouvée. Si le booléen permettant de générer le chemin exhaustif est paramétré en true, l'algorithme de recherche en profondeur ne compose que des chemins passant par toutes les quêtes disponibles avec l'aide d'une condition d'arrêt qui empêche la solutions d'être ajouté si celle-ci est incomplète.

Enfin, on va trier les solutions avec un algorithme de tri par insertion, qui va trier la liste des solutions avec le critère choisi par l'utilisateur, et va également choisir l'ordre du tri, choisissant entre les pires et les meilleures solutions puis finalement découper la liste pour ne retourner que les n solutions demandées.

IV. Conclusion


Ce projet nous a permis de nous pencher sur plein de problématiques intéressantes et de pratiquer les choses apprises durant notre première année de BUT Informatique. Nous allons faire un petit historique de ce projet qui dure depuis déjà plusieurs mois.

Déjà, nous avons commencé à découvrir ce projet lors de la [SAÉ gestion de projet](#), qui consistait en l'organisation complète du projet, en passant par l'ordonnancement des tâches avec le WBS, le graphique d'ordonnancement ou le diagramme de Gantt, ou encore l'étude du projet avec l'étude de rentabilité. Cette partie de gestion de projet nous a permis d'organiser un véritable projet complexe et avec de nombreuses tâches à prévoir, et a été nécessaire dans le processus de réalisation tout au long du projet. L'utilisation des plateformes comme Notion ou Figma nous ont permis de nous organiser et de préparer au mieux le terrain face à l'épreuve qui nous attendait : le développement de l'application.

Une fois l'étape de l'organisation et de la gestion, nous commençâmes la véritable phase du projet, [le développement](#). Mais avant ça, il manquait une dernière étape. En effet, le cours de qualité de développement nous l'a démontré, il ne faut pas se jeter dans le développement les yeux fermés. Nous avons donc réfléchi et modélisé les différentes structures de données et classes dont nous aurions besoin, et avons décidé d'un cycle de vie pour le projet, pour finalement créer un projet gradle et le dépôt git sur GitHub.

Nous avons ainsi pu nous consacrer au développement et [mettre en oeuvre les connaissances](#) acquises lors des cours de développement, en développant les différentes classes de base, pour enfin arriver au développement des classes permettant de calculer les solutions et répondre au sujet initial, puis finalement réaliser les tests des algorithmes.

Pour finir, nous avons utilisé nos connaissances en IHM pour réaliser [l'interface de l'application](#) et ainsi mettre un point final sur le projet.



Ce grand projet rassemblant trois SAÉ nous a permis de mettre en œuvre nos connaissances et ainsi travailler sur un véritable sujet pendant plusieurs semaines.

Pour ce qui est des [idées de voies d'amélioration](#), nous pourrions par exemple ajouter un [affichage du jeu en lui-même](#), en affichant l'emplacement du joueur, des quêtes, et en affichant le meilleur chemin directement sur un plan en deux dimensions où l'utilisateur peut visualiser les déplacements sur la carte directement.

On pourrait aussi, avec cette méthode, calculer des solutions directement [à partir d'un certain avancement dans le jeu](#). En effet, si le joueur a déjà réalisé trois quêtes et est positionné à un certain endroit sur la carte, celui-ci pourrait calculer directement les solutions depuis l'avancement où il est, et ainsi agrandir les possibilités proposées par l'application.

On pourrait aussi proposer un calcul de solutions avec pour critère l'expérience gagnée lors du parcours par exemple, ou proposer la moyenne d'une colonne d'un critère.

Quelques exemples d'organisation peuvent aussi être mis sur la table. C'est un projet qui a l'avantage d'être malléable, et on peut alors l'améliorer de plein de façons différentes.

V. Annexe

LIEN VERS LE DÉPÔT GIT :

(informations importantes dans le fichier README.MD)

<https://github.com/NoXeDev/sae202>