Tower Defense

Mise en œuvre d’un graphe interactif

Création d’une variante du célèbre jeu Tower Defense dans le cadre d’un cours d’algorithmique.

ASD2

Lazhar Farjallah / Aurélien Da Campo / Pierre-Dominique Putallaz

Heig-vd

ASD

TD

Sommaire

[1 Analyse préliminaire 4](#_Toc250982530)

[1.1 Introduction [LAZHAR] 4](#_Toc250982531)

[1.2 Organisation [LAZHAR] 4](#_Toc250982532)

[1.3 Objectifs [LAZHAR] 5](#_Toc250982533)

[1.4 Planification initiale [AURELIEN] 6](#_Toc250982534)

[2 Analyse 7](#_Toc250982535)

[2.1 Concepts algorithmiques [LAZHAR] 7](#_Toc250982536)

[2.1.1 Outils algorithmiques 7](#_Toc250982537)

[2.1.2 Complexité 9](#_Toc250982538)

[2.2 Etude de concurrentielle [AURELIEN] 9](#_Toc250982539)

[2.3 Etude de faisabilité [AURELIEN] 9](#_Toc250982540)

[2.4 Planification [AURELIEN] 10](#_Toc250982541)

[3 Conception 11](#_Toc250982542)

[3.1 Dossier de conception 11](#_Toc250982543)

[3.1.1 Systèmes d’exploitation [LAZHAR] 11](#_Toc250982544)

[3.1.2 Outils logiciels [LAZHAR] 11](#_Toc250982545)

[3.1.3 Librairies externes [LAZHAR] 11](#_Toc250982546)

[3.1.4 Architecture de l’application [AURELIEN] 11](#_Toc250982547)

[3.1.5 Interface graphique [AURELIEN] 12](#_Toc250982548)

[3.1.6 Schémas UML [PIERRE-DO] et [AURELIEN] 12](#_Toc250982549)

[3.1.7 Gestion de la concurrence [PIERRE-DO] 12](#_Toc250982550)

[4 Réalisation 13](#_Toc250982551)

[4.1 Dossier de réalisation 13](#_Toc250982552)

[4.1.1 Résultat de l’interface graphique [AURELIEN] 13](#_Toc250982553)

[4.1.2 Résultat de l’implémentation du maillage [PIERRE-DO] 13](#_Toc250982554)

[4.2 Description des tests effectués [LAZHAR] 13](#_Toc250982555)

[4.2.1 Génération du maillage (graphe associé) avec *JGraphT* 13](#_Toc250982556)

[4.2.2 Recherche du chemin le plus court 15](#_Toc250982557)

[4.2.3 Enregistrement des scores et leur tri 18](#_Toc250982558)

[4.2.4 Test sur un autre environnement 21](#_Toc250982559)

[4.2.5 Test global de l’application 21](#_Toc250982560)

[4.3 Erreurs restantes [LAZHAR] 23](#_Toc250982561)

[*5* Conclusions *[LAZHAR]* 23](#_Toc250982562)

[5.1 Objectifs atteints / non-atteints 23](#_Toc250982563)

[5.1.1 Illustrer le concept de graphe de manière ludique et interactive. 23](#_Toc250982564)

[5.1.2 Acquérir de l’expérience dans la planification et l’accomplissement d’un projet conséquent. 23](#_Toc250982565)

[5.1.3 Utiliser et découvrir des librairies existantes implémentant le concept de graphe 23](#_Toc250982566)

[5.1.4 Apprendre à mettre en œuvre une interface graphique en Java. 24](#_Toc250982567)

[5.1.5 Séparer le travail en plusieurs niveaux d’abstraction pour faciliter l’élaboration et l’évolutivité de ce projet. 24](#_Toc250982568)

[5.1.6 Comprendre la nécessité d’utiliser des algorithmes complexes dans les applications informatiques. 24](#_Toc250982569)

[5.1.7 Mettre en œuvre un algorithme de recherche de chemin le plus court (ACPC). 24](#_Toc250982570)

[5.2 Points positifs / négatifs 25](#_Toc250982571)

[5.2.1 Points positifs 25](#_Toc250982572)

[5.2.2 Points négatifs 25](#_Toc250982573)

[5.3 Difficultés particulières 25](#_Toc250982574)

[5.4 Avenir du projet 25](#_Toc250982575)

[6 Annexes 26](#_Toc250982576)

[6.1 Sources – Bibliographie [LAZHAR] 26](#_Toc250982577)

[6.2 Journal de bord de chaque participant [LAZHAR] 26](#_Toc250982578)

[6.3 Manuel d'Utilisation [LAZHAR] 26](#_Toc250982579)

[6.4 Archives du projet [LAZHAR] 26](#_Toc250982580)

# Analyse préliminaire

## Introduction [LAZHAR]

Ce projet de fin de semestre consiste à créer une application ludique mettant en œuvre des algorithmes et structures de données étudiées en cours. Nous avons choisi pour cela de créer une variante du célèbre jeu « Tower Defense », dans lequel des personnages se déplacent d’un point A à un point B selon un chemin optimal. En effet, on aura besoin pour cela d’une structure de graphe ainsi que les algorithmes associés, ce qui colle parfaitement avec la contrainte de départ car nous les avons étudiés en cours.

Le but est également ici de créer une application « didacticiel » (il s'agit d'un logiciel interactif destiné à l'apprentissage de savoirs) qui sera présentée dans les futurs cours de cette unité d’enseignement. Ce projet permettra en effet de montrer une application réelle de l’utilisation d’algorithmes associés à des graphes (en particulier celui de recherche du chemin le plus court entre deux nœuds).

Nous commencerons par effectuer une analyse du projet, notamment en ce qui concerne l’organisation, les objectifs ainsi que la planification initiale. Puis nous détaillerons les étapes de conception et de réalisation. Enfin, avant de conclure, nous présenterons les fonctionnalités de l’application finale sous forme de mode d’emploi et nous présenterons des captures d’écran.

## Organisation [LAZHAR]

Les membres participant à ce projet sont :

**Etudiant 1** :

Aurélien Da campo, *aurelien.dacampo@heig-vd.ch*

**Etudiant** **2** :

Pierre-Dominique Putallaz, *pierre-dominique.putallaz@heig-vd.ch*

**Etudiant 3 (responsable de projet)** :

Lazhar Farjallah, *lazhar.farjallah@heig-vd.ch*

Nous prévoyons de nous répartir les tâches de manière suivante entre chaque entité du groupe :

1. *Responsable de projet*
   1. Rédactions, administrations
   2. Suivi des rendus (*deadlines*)
   3. Surveillance et coordination
   4. Développement
2. *Etudiant 1*
   1. Création de l’interface graphique
   2. Gestion de l’affichage
   3. Rendu graphique
   4. Interaction avec l’utilisateur
3. *Etudiant 2* 
   1. Algorithmique
   2. Implémentation des algorithmes de graphe
   3. Fournir les briques logicielles pour permettre la construction de la partie fonctionnelle de l’application

## Objectifs [LAZHAR]

Les objectifs de ce projet sont les suivants :

* Illustrer le concept de graphe de manière ludique et interactive.
* Acquérir de l’expérience dans la planification et l’accomplissement d’un projet conséquent.
* Utiliser et découvrir des librairies existantes implémentant le concept de graphe
* Apprendre à mettre en œuvre une interface graphique en Java.
* Séparer le travail en plusieurs niveaux d’abstraction pour faciliter l’élaboration et l’évolutivité de ce projet.
* Comprendre la nécessité d’utiliser des algorithmes complexes dans les applications informatiques.
* Mettre en œuvre un algorithme de recherche de chemin le plus court (ACPC).
* Respecter le design pattern MVC (*Model* – *View* – *Controller*) qui structure un programme en trois couches principales.

## Planification initiale [AURELIEN]

# Analyse

## Concepts algorithmiques [LAZHAR]

Le principal concept algorithmique utilisé dans ce projet est celui de graphes. En effet, ce concept, bien connu dans le domaine algorithmique, constitue un point clé dans la réalisation de notre cahier des charges. Il intervient principalement dans le problème de la recherche d’un chemin (le plus court) entre deux points donnés A et B.

Si on regarde à un niveau plus abstrait, le but premier de notre application est de lancer des objets (créatures) à partir d’un point A pour qu’ils se dirigent vers un point B selon un chemin optimal, calculé à la volée en fonction d’un graphe qui ne cesse de se modifier (dynamique).

Nous avons également besoin d’un algorithme de tri afin de trier les meilleures scores du jeu obtenus et de les afficher dans l’ordre.

Cette simple vision est tout à fait suffisante pour comprendre les outils algorithmiques dont nous devons s’armer.

Nous allons maintenant développer plus en détails quels sont les structures et algorithmes de graphes dont nous avons besoin.

### Outils algorithmiques

Partant du constat du paragraphe 2.1, on comprend assez facilement que nous aurons besoin des structures suivantes :

* un graphe pondéré non orienté
* des nœuds composant le graphe
* des arcs pour relier les différents nœuds du graphe
* un algorithme de recherche d’un chemin optimal entre deux nœuds (connu sous l’appellation « algorithme de Dijkstra ») appliqué à un graphe pondéré non orienté
* un algorithme de tri afin de trier les meilleures scores des joueurs

#### Graphe pondéré non orienté

Nous avons décidé d’utiliser un tel graphe car il correspond parfaitement à ce dont nous avons besoin. En effet, chaque arc possède un poids (pondéré), car une créature doit savoir à partir de n’importe quel nœud du graphe quel est la longueur du chemin jusqu’au prochain nœud. De plus, nous avons choisi d’utiliser un graphe non orienté, c’est-à-dire que les arcs n’ont pas de sens (si un nœud A est voisin d’un nœud B alors B est aussi un voisin de A). Les nœuds sont donc traversables en double sens, ce qui se prête bien à nos besoins car il se peut très bien qu’une créature doive tout d’un coup reculer et revenir sur ces pas.

#### Nœuds composants le graphe

Les nœuds de notre graphe contiennent des informations telles que la position x et y (en fonction du repère cartésien du jeu) afin de connaître le parcours à suivre en coordonnées cartésiennes, mais aussi un « drapeau » (*flag* en anglais) permettant de savoir si un tel nœud est actif ou non.

#### Arcs pour relier les nœuds du graphe

Les arcs reliant les nœuds du graphe sont pondérés, c’est-à-dire qu’ils possèdent une valeur associée (un nombre) strictement positif déterminant la longueur (tout simplement) de l’arc en question. Cette longueur est calculée selon la position cartésienne des extrémités de l’arc (autrement dit à partir des 2 nœuds reliés par l’arc) grâce au théorème de Pythagore. Ces arcs sont également non orientés, c’est-à-dire qu’ils ne possèdent pas de sens (chaque arc peut être parcouru dans n’importe quel sens).

#### Algorithme de dijkstra (sur un graphe pondéré non orienté)

Voici, en pseudo-code, l’algorithme de recherche de tous les chemins les plus courts à partir d’un sommet initial vers tout autre sommet du graphe (un sommet est un synonyme de nœud) que nous utilisons:

*(partie générale)*

marquer tous les sommets comme non visités

visiter le sommet initial

*(partie spécifique à un sommet : visiter le sommet)*

**si** le sommet est non visité **alors**

déposer le sommet avec priorité nulle dans la queue de

priorité

**tant que** la queue de priorité n’est pas vide **boucler**

prélever le sommet de tête (il devient le sommet courant)

marquer le sommet courant comme visité

traiter le sommet courant

**pour** chaque sommet voisin du sommet courant **boucler**

**si** le sommet voisin est non visité **alors**

calculer la priorité du sommet voisin (c’est-à-dire

prendre la somme de l’attribut de l’arc, entre le

sommet courant et lui-même, et de la priorité du

sommet courant)

déposer le sommet voisin et sa priorité dans la

queue de priorité

**fin si**

**fin boucler**

**fin boucler**

**fin si**

Algorithme 2.1 : recherche des chemins les plus courts dans un graphe (Dijkstra).

#### Algorithme de tri pour le tri des meilleurs scores des joueurs

Pour trier les meilleurs scores obtenus par le joueurs, nous utilisons la méthode sort() disponible dans l’API Java, applicable sur un objet de type ArrayList (une collection). En effet, Java utilise pour ce faire le tri Merge Sort (tri fusion) dont voici le pseudo-code :

procedure triFusionI(entier[] tab)

entier[] tmp <- tableau de taille N;

entier i <- 1;

entier debut <- 1;

entier fin <- debut + i + i - 1;

tant que (i < N) faire

debut <- 1;

tant que (debut + i -1 < N) faire

fin <- debut + i + i - 1;

si (fin > N) alors

fin <- N;

fusion(tab, tmp, debut, debut + i - 1, fin);

debut <- debut + i + i;

fin tant que

i <- i + i;

fin tant que

fin procedure

Algorithme 2.2 : tri fusion sur un tableau d’entiers.

#### Algorithme pour la génération de la santé des créatures

### Complexité

#### Algorithme de Dijkstra

Si le graphe possède *m* arcs et *n* nœuds, en supposant que les comparaisons des poids d'arcs soient à temps constant, et que le tas soit binomial, alors la complexité de l'algorithme est .

#### Tri fusion

On voit que la procédure de fusion nécessite un tableau intermédiaire aussi grand que le nombre d'éléments à trier. C'est en effet le principal inconvénient du tri fusion, car sa complexité est dans tous les cas en, (prix du tableau annexe aussi grand que le tableau initial).

## Etude de concurrentielle [AURELIEN]

## Etude de faisabilité [AURELIEN]

## Planification [AURELIEN]

# Conception

## Dossier de conception

### Systèmes d’exploitation [LAZHAR]

Le système d’exploitation sur lequel nous travaillons est Windows (*XP* et *Seven*). Cependant, grâce au choix qui a été fait d’utiliser un encodage de type UTF-8 ainsi que celui du langage portable Java, nous pouvons sans soucis travailler sur un environnement Linux ou Mac par exemple.

### Outils logiciels [LAZHAR]

Nous développons avec l’IDE Eclipse (version 3.4 et supérieure) intégrant tous les outils nécessaires au développement d’applications Java. Ce logiciel est très largement répandu dans le monde des développeurs et est très utilisé. Il possède de nombreuses fonctions spécialement conçues pour augmenter la productivité des développeurs et leur simplifier la vie (comme le *refactoring* par exemple). De plus, cette plateforme nous permet d’ajouter toute une série de plugins qui peuvent ajouter des fonctionnalités, telles que SVN (logiciel de gestion des versions du code). En ce qui concerne la génération des diagrammes de classe UML, nous utilisons le plugin ***eUML 2.0*** de chez ***Soyatec***. Finalement, pour la gestion des versions en fonction de l’avancement du projet, nous utilisons le fameux logiciel ***svn*** ainsi que le service ***GoogleCode*** afin de garder une trace de notre code à chaque nouvelle étape du projet, le tout étant sauvegardé sur un serveur.

### Librairies externes [LAZHAR]

#### JGraphT

Nous utilisons la librairie (libre) externe ***JGraphT*** codée en Java. En effet, cette librairie possède toutes les briques logicielles nécessaires à la création de graphes (de tout type). Nous l’utilisons comme une boîte noire sans se soucier de son implémentation. Le choix de cette librairie n’a pas été sans réflexion. En effet, après quelques recherches effectuées sur Internet ainsi qu’une discussion avec le professeur, cette librairie nous parait correcte.

#### JLayer

Nous utilisons la librairie (libre) externe ***JLayer*** codée en Java. Cette librairie nous permet de jouer des musiques codées en divers formats tel que le mp3 par exemple (ou le wav). Elle sert uniquement à jouer de la musique, ce qui apporte un petit plus au projet.

### Architecture de l’application [AURELIEN]

Nous avons choisi comme architecture du programme le fameux design pattern MVC (Modèle – Vue – Contrôleur).

### Interface graphique [AURELIEN]

### Schémas UML [PIERRE-DO] et [AURELIEN]

#### Maillage [PIERRE-DO]

#### Interface graphique (vue et contrôleur) [AURELIEN]

#### Gestionnaire du jeu (model) [AURELIEN]

### Gestion de la concurrence [PIERRE-DO]

# Réalisation

## Dossier de réalisation

### Résultat de l’interface graphique [AURELIEN]

### Résultat de l’implémentation du maillage [PIERRE-DO]

Ici nous allons discuter de l’implémentation de la partie algorithmique dure du projet, à savoir le graphe dynamique. Nous avons modélisé ce graphe sous la forme d’une classe Maillage qui contient en interne un graphe implémenté par la librairie externe Jgrapht (voir section « Outils Logiciels »).

L’implémentation du TDA maillage est définie par le cahier des charges suivant (description des méthodes) :

1. Constructeurs : La liste des différents constructeurs permettant de créer le maillage en fonction de différents paramètres. L’unité utilisée par default est le pixel, depuis la zone graphique selon les standards. Une exception est levée si les paramètres ne sont pas probables.
   1. + Maillage (int,int,int,int,int) : ce constructeur public prend en paramètres dans l’ordre :
      1. La largeur totale en pixel du maillage.
      2. La hauteur totale en pixel du maillage.
      3. Le côté du nœud en pixel. Un nœud est toujours carré.
      4. La position de départ du maillage dans la zone graphique en x.
      5. La position du départ du maillage dans la zone graphique en y.
   2. + Maillage (int,int,int)*:* mêmes paramètres que le constructeur précédent, avec la position du maillage par default en (0 ;0) .
2. Méthodes d’éditions : ces méthodes permettent d’éditer la structure du maillage de manière dynamique après l’avoir crée. Si le rectangle définissant les zones à éditer n’est pas dans la zone, une exception est levée. Méthodes « thread-safe ».
   1. + void activerZone(Rectangle) : Permet d’activer une zone dans le maillage, c'est-à-dire de définir les nœuds contenus dans le rectangle passé en paramètre comme actifs et de les relier aux nœuds actifs adjacents. Le rectangle a une dimension en pixels relatifs à la zone de jeu.
   2. + void desactiverZone(Rectangle) : Permet de désactiver une zone dans le maillage, c'est-à-dire de marquer comme accessibles les nœuds contenus dans le rectangle passé en paramètre et de supprimer les arcs dans le graphe relatifs à ce nœud. Le rectangle a une dimension en pixels relatifs à la zone de jeu.
3. Méthode de calcul : Permet d’obtenir des informations sur le maillage, notamment le chemin le plus court d’un point à un autre. Méthode « thread-safe ».
   1. + ArrayList<Point> plusCourtChemin(int,int,int,int) : C’est la méthode principale du maillage. Elle permet à partir des coordonnées des points de départ et d’arrivée de calculer selon le maillage le chemin le plus court, qu’elle retournera sous forme d’un ArrayList de points représentant les différentes étapes du chemin. Dans l’ordre des paramètres :
      1. La coordonnée x du point de départ
      2. La coordonnée y du point de départ
      3. La coordonnée x du point d’arrivée
      4. La coordonnée y du point d’arrivée
4. Accesseurs : Cet ensemble des méthodes permet de consulter l’état du maillage à n’importe quel moment, par exemple de récupérer la liste des nœuds internes au graphe. Nous ne détaillerons pas ici l’ensemble des méthodes, pour les détails voir la JavaDoc.

La représentation interne de la structure se fait selon le principe suivant. L’ensemble des nœuds est stocké dans un tableau, puis passé au graphe pour qu’ils soient organisés logiquement et qu’on puisse leur ajouter des arcs. Comme il s’agit de références passées, on peut accéder de manière équivalente aux nœuds par le graphe (que les nœuds actifs) et par le tableau interne du maillage. Cela permet d’ajouter et de supprimer dynamiquement des nœuds dans le graphe sans à avoir à les régénérer.

Pour la gestion des arcs, c'est-à-dire l’ajout lors de l’activation dynamique d’une zone ou à la création du maillage, on utilise l’algorithme suivant :

Pour un nœud cible :

Ajout du nœud dans le graphe.

Marquer le nœud comme actif.

for tout ses voisins do

if le voisin est actif do

Ajout dans le graphe d’un arc cible <-> voisin.

Assignation du poids de l’arc, c'est-à-dire la distance en pixel qui sépare les deux points.

End if

End for

Il est essentiel de retenir que le graphe n’est donc pas orienté. La figure ci-contre illustre l’algorithme, le nœud bleu est le nœud cible, il est relié aux nœuds actifs et lui-même marqué comme actif. Cet algorithme permet d’utiliser la même méthode lors de l’initialisation des nœuds que lors de la réactivation d’une zone après avoir été désactivée.

Donnons également un petit point sur les unités. En effet, c’est un détail qui nous a posé soucis. Nous avons distingué deux types d’unités :

* Le pixel : C’est l’unité avec laquelle est créé, édité et lu le maillage. Les classes qui implémente le maillage travaillent dans une zone de dessin en pixel, toutes les opérations prennent et retourne cette unité. Le maillage a conscience de sa dimension en pixel, de la largeur du nœud du pixel ainsi que du décalage par rapport à l’origine, toujours en pixel.
* La coordonnée nodale : C’est l’abstraction utilisée à l’intérieur de la classe Maillage pour représenter un nœud dans le tableau de nœud. Par définition une coordonnée nodale égale une largeur de nœud en pixel, à quelques subtilités prêt.

L’essentiel de la difficulté de travail avec des deux types d’unités (uniquement en interne), est qu’il a fallu non seulement définir une relation pixel->nodale, mais également la relation inverse. Nous avons utilisé l’algorithme suivant pour la conversion :

Pixel -> nodale :

Pour chaque coordonnée x et y en pixel do

Calcul de la coordonnée modulo taille du nœud.

Soustraction de cette valeur à la coordonnée.

Division de la valeur obtenue par la largeur du nœud.

Retour de x et y en nodal.

Nodale -> pixel :

Pour chaque coordonnée x et y en nodale do :

Multiplication de la coordonnée nodale par la largeur du nœud en pixel.

Retour de x et y en pixel.

En effet, la coordonnée nodale converti en pixel représente le centre du nœud, tandis qu’une coordonnée en pixel est converti vers la coordonnée nodale la plus proche.

Résumons donc les propriétés du maillage :

* Graphe non orienté (pour permettre une navigabilité dans les deux sens).
* Graphe pondéré (pour permettre de représenter la distance entre les points pour l’algorithme de Dijkstra).
* Toujours un chemin entre le point de départ et d’arrivée.
* Edition dynamique du maillage par activation ou désactivation d’une zone.
* Lecture sur le maillage par une liste des arcs actifs.
* Lecture sur le maillage par une liste de points représentant le chemin le plus court entre un point A et un point B donné.
* Utilisation de méthodes de conversions pour les différentes unités.

Afin de faire fonctionner la classe Maillage, il y a quelques classes annexes (pour chaque classe voir la JavaDoc pour des informations plus détaillées) :

* Arc : Il s’agit d’un arc dans le maillage. Il étend la classe DefaultWeightedEdge (de Jgrapht) pour pouvoir être intégré dans le maillage et pour pouvoir jouer avec le poids de l’arc. Il contient également une référence vers le nœud de départ et le nœud d’arrivée. Une méthode permet également de le convertir en Line2D pour un affichage plus propre dans la zone graphique.
* GenerateurDArcs : Requis dans la structure du JGrapht, permet de générer les arcs. Utilisé lors de la création du graphe dans la classe Maillage. Il en a été fait une classe externe pour une éventuelle réutilisation ultérieure.
* Nœud : Représente un nœud dans le graphe. Il étend la classe Point pour le stockage des coordonnées x et y en pixel et pour un affichage plus facile. Il contient des attributs comme son état (actif ou inactif), sa largeur en pixel, ainsi que quelques méthodes statiques de conversion de coordonnée et de calcul du centre du nœud en fonction d’une coordonnée x et y passée en paramètre.
* PathNotFoundException : Exception levée par le maillage si aucun chemin n’est trouvé. Classe externe pour permettre aux classes utilisant la classe maillage de traiter correctement cette exception.

## Description des tests effectués [LAZHAR]

### Génération du maillage (graphe associé) avec *JGraphT*

Pour ce test, nous avons mesuré le temps de génération du graphe associé à un terrain (une « map ») du jeu. Nous avons décidé d’effectuer ce test avec un graphe doté d’un nœud tous les 2 pixels, ainsi que 8 arcs par nœuds, répartis uniformément. Les nœuds sont tous reliés les un aux autres.

Voici un schéma du graphe généré pour un terrain de 4x4 pixels :

x [pixels]

y [pixels]

(0,0)

(2,0)

(4,0)

(0,2)

(0,4)

(4,4)

noeuds

arcs

Figure 4.2.1 : graphe généré avec un nœud tous les 2 pixels.

Le résultat, après avoir pris les mesures correspondantes, figurent sous deux formes. Dans la première, on mesure le temps de génération du maillage en fonction d’une taille de terrain en pixels (par exemple 800x600). Dans la seconde, on mesure l’espace mémoire nécessaire en fonction toujours de la taille du terrain en pixels.

Voici les résultats obtenus, sous forme de graphiques :

Figure 4.2.1.1 : temps de génération d’un maillage (graphe).

Figure 4.2.1.2 : mémoire utilisée lors de la génération d’un maillage (graphe).

En conclusion de ce premier test, on peut voir plusieurs choses intéressantes. Tout d’abord, rappelons que nous utilisons pour générer les maillages (graphes) une librairie externe, d’où l’utilité de ces tests, car nous ne sommes que des utilisateurs de cette librairie.

Premièrement, on voit sur le premier graphique que le temps de génération d’un maillage est en , grâce à la fonction y d’approximation. Il en va de même pour le second graphique, cette fois pour la mémoire. Cette première conclusion n’est pas surprenante, puisque pour créer le maillage, il faut créer n nœuds puis ensuite pour chaque nœud, il faut créer m arcs.

Deuxièmement, on voit qu’au niveau du temps de génération ainsi que de la mémoire, il va falloir choisir une valeur « correcte », c’est-à-dire une valeur optimale de la taille en pixels du terrain pour optimiser le temps et la mémoire lors de la génération du maillage, qui se fait au lancement de notre application. Nous avons donc choisi d’utiliser un terrain de jeu composé d’au maximum 500x500 pixels. En effet, d’après les graphiques, cela signifie que le temps de génération est d’environ 1 seconde et la mémoire utilisée d’environ 30 mégaoctets, ce qui est relativement raisonnable.

Cependant, pour baisser encore ces valeurs, nous avons décidé de créer finalement un nœud tous les 10 pixels, et non plus un nœud tous les 2 pixels. De ce fait, nous aurons un graphe qui aura 5 fois moins de nœuds et d’arcs que le graphe que nous générons dans ce test (car pour une zone de 10x10 pixels, on avait avant 5 nœuds, alors que maintenant on en a plus qu’un). On peut alors s’attendre à un temps de génération d’environ 0.2 seconde ainsi qu’une mémoire utilisée d’environ 6 mégaoctets, ce qui est cette fois parfaitement raisonnable.

### Recherche du chemin le plus court

Dans ce test, nous nous bornerons à faire des captures d’écran de l’application et à démontrer le bon fonctionnement de l’algorithme de recherche de chemin le plus court (*Dijkstra*).

Voici une série de trois captures d’écran illustrant les trois situations les plus intéressantes, suivies d’explications :

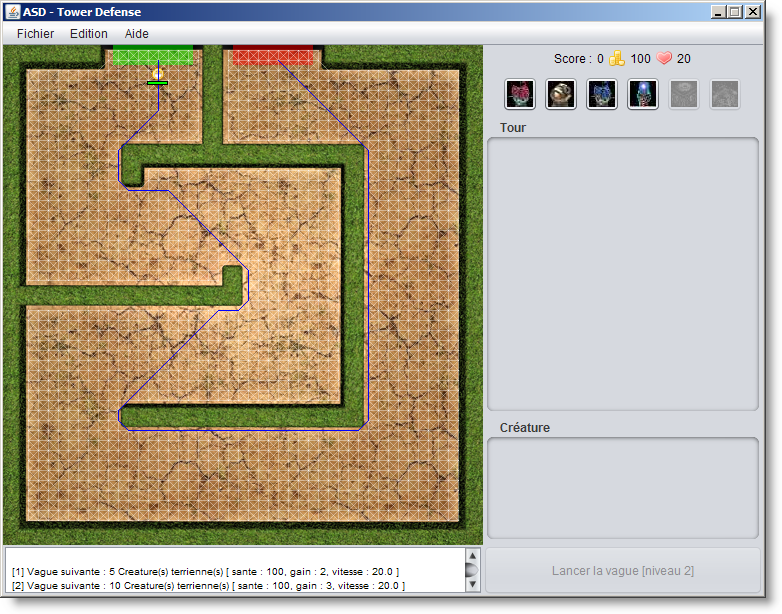


Figure 4.2.2.1 : terrain de jeu avec le chemin le plus court en bleu, situation 1.

Dans cette première situation, aucun obstacle n’est placé. Le chemin optimal est celui tracé en bleu. On voit nettement que ce chemin est le plus court. Cette situation est la plus triviale et ne nécessite pas de remarques particulières, si ce n’est que l’algorithme de recherche du chemin optimal fonctionne parfaitement.

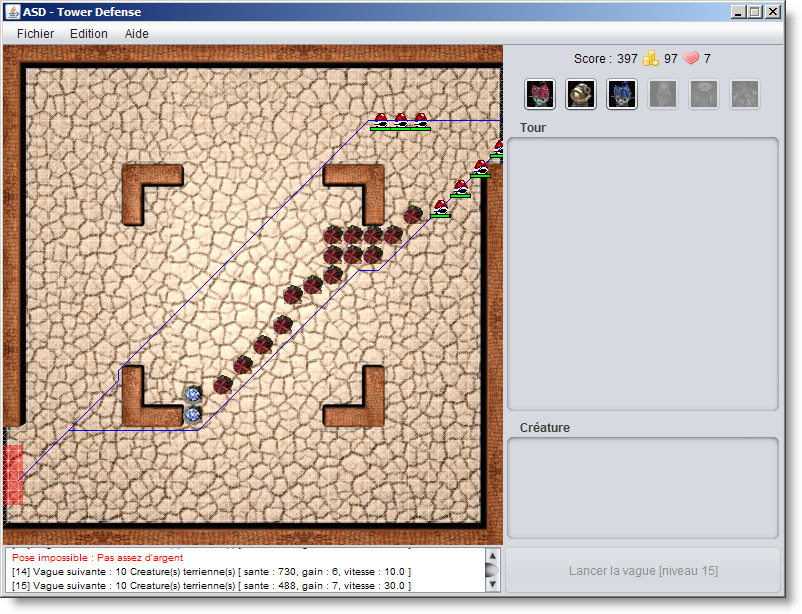


Figure 4.2.2.2 : terrain de jeu avec le chemin le plus court en bleu, situation 2.

Dans cette deuxième situation, il est intéressant de constater qu’il peut y avoir plusieurs chemins. En fait, c’est parce qu’on peut bouger dynamiquement les tours faisant office d’obstacle pendant le jeu. Ici par exemple, les trois premières créatures en haut avaient à la base un autre chemin optimal, en fonction des tours qui étaient placées à ce moment-là. Puis, après avoir supprimé/ajouté des tours, les quatre créatures suivantes se sont attribuées un nouveau chemin. Il est donc intéressant de voir que la recherche du chemin optimal est bel et bien dynamique : le chemin peut varier en tout temps. De plus, on constate que chaque créature possède son propre chemin optimal, toutes les créatures n’étant pas au même endroit (c’est-à-dire au même nœud en terme de graphe) à un temps donné *t*.

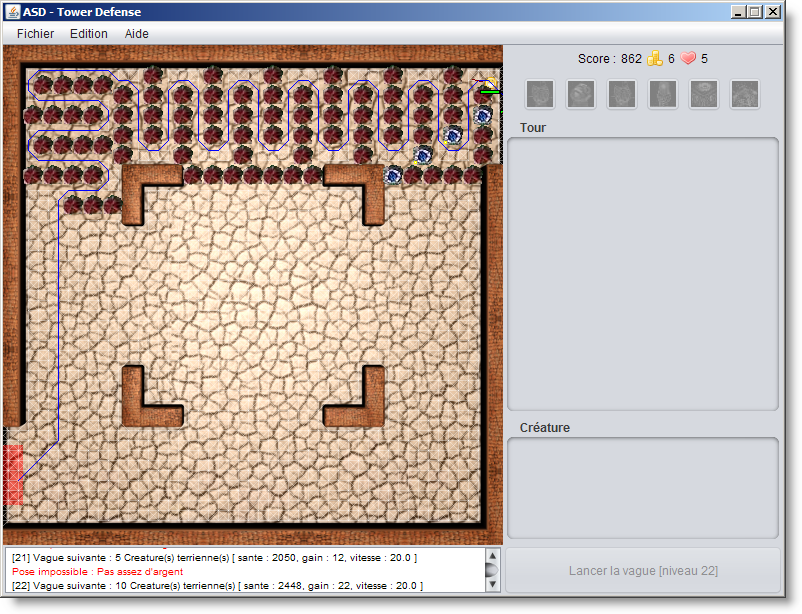


Figure 4.2.2.3 : terrain de jeu avec le chemin le plus court en bleu, situation 3.

Dans cette troisième et dernière situation, on voit que le joueur a pris le soin de créer une espèce de labyrinthe. Comme les créatures doivent se rendre du point de départ (en haut à gauche) jusqu’à l’arrivée (en bas à droite) coûte que coûte, elles prendront le meilleur chemin existant. En effet, ici elles seront obligées de parcourir tout le labyrinthe, car le chemin résultant, malgré qu’il soit long, devient un chemin optimal. De plus, on voit nettement qu’à la sortie du labyrinthe créer par le joueur, le chemin est à nouveau « livré à lui-même » car il n’y a plus aucun obstacle (d’où la ligne droite allant vers l’arrivée qu’on voit après la sortie du labyrinthe).

### Enregistrement des scores et leur tri

Lorsqu’un joueur termine une partie, son score est sauvegardé dans un fichier « *sérializé »* sur le disque dur dans le dossier /donnees. Nous allons ici simplement tester cette fonctionnalité et fournir des captures d’écran démontrant son bon fonctionnement.

Voici un schéma de jeu classique :

1. Le joueur lance la partie et joue jusqu’à ce qu’il perde.
2. Le joueur a perdu ; il est invité à inscrire son nom.
3. Le joueur est ensuite invité à consulter la liste des dix derniers meilleurs scores classés par ordre décroissant (le meilleur score en premier), le score qu’il vient d’obtenir étant lui aussi inclus.
4. Sans le voir directement, les scores sont automatiquement sauvegardés dans un fichier sur le disque dur pour que le joueur puisse à tout moment revoir ses anciens scores.

Voici à présent trois captures d’écran destinées à démontrer les étapes 2, 3, et 4 décrites ci-dessus :

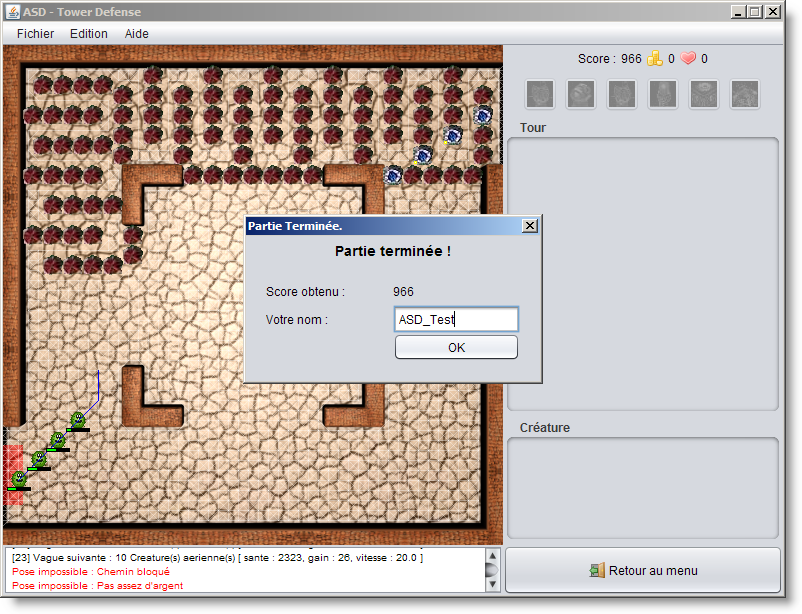


Figure 4.2.3.1 : le joueur a perdu ; il est invité à inscrire son nom.

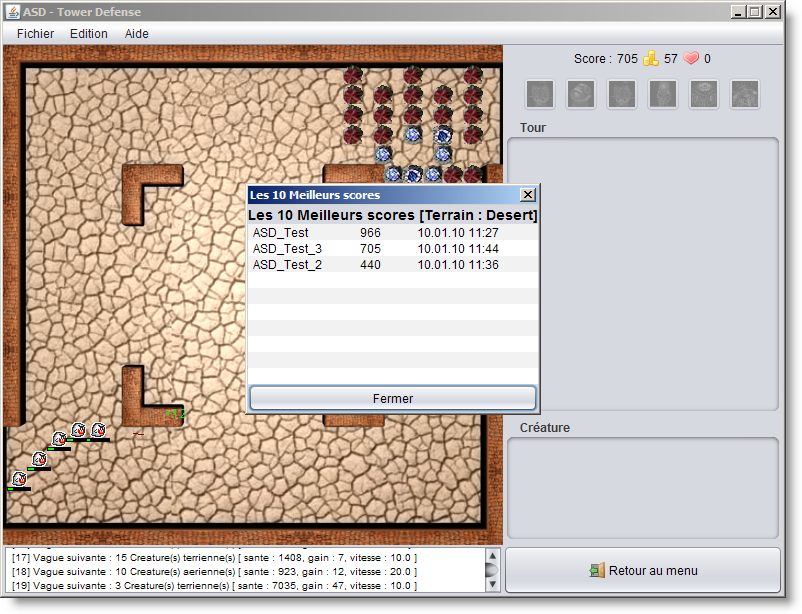


Figure 4.2.3.2 : le joueur voit la liste des 10 derniers meilleurs scores triés.

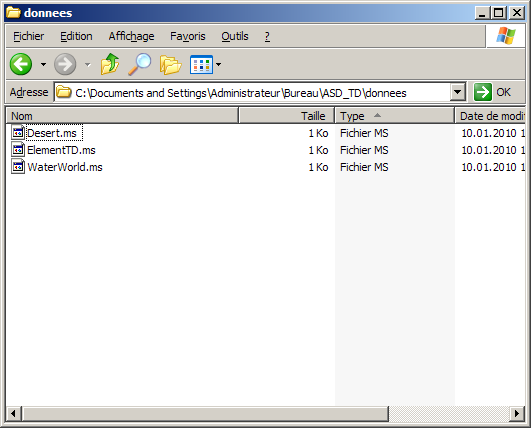


Figure 4.2.3.3 : Les scores sont sauvegardés dans le dossier /donnees.

### Test sur un autre environnement

Nous avons testé notre application sur un environnement Linux. Comme elle est codée en Java, elle est de toutes façons portable, car cette tâche revient aux concepteurs du langage. Malgré cela, nous avons jugé bon de le tester.

Voici une capture d’écran de l’application tournant sur une machine doté de Linux Ubuntu 9 :



### Test global de l’application

Nous avons décidé, peu de temps avant le rendu de ce laboratoire, de diffuser, en « *beta-test »*, une version beta de notre jeu sur Internet, à l’adresse http://code.google.com/p/asd-tower-defense/. Ainsi, après avoir envoyé quelques e-mails, plusieurs personnes ont joué à notre jeu et nous ont fait des petits feedbacks.

Voici par exemple la capture d’écran qu’un contact nous a envoyé, pendant qu’il testait notre application :



Figure 4.2.4.1 : Capture d’écran fournie par un de nos « beta-testeurs ».

Ce test comporte beaucoup d’informations intéressantes.

Tout d’abord, nous pouvons voir que le joueur est arrivé au niveau 90 sans bug ni crash de l’application, ce qui représente plus d’une heure de jeu. L’application est donc robuste à ce niveau-là.

Deuxièmement, on voit que le joueur a pleinement utilisé les fonctionnalités du jeu : il utilise tous les types de tours et d’améliorations possibles, ce qui est très positif pour nous car cela signifie qu’aucune des fonctionnalités n’est inutile aux yeux du joueur.

Troisièmement, nous pouvons voir qu’au niveau des chiffres, c’est-à-dire le score, les prix de vente des tours, les prix d’achat des tours, l’argent économisé ainsi que le nombre de vies, ils évoluent parfaitement bien avec l’avancement du jeu. Il n’y a effectivement pas de valeur aberrante ou totalement inadaptée.

Quatrièmement, en ce qui concerne la fenêtre de log, en bas, nous pouvons constater qu’elle est parfaitement à jour.

Enfin, ce joueur ainsi que plusieurs autres nous on rapporté qu’à partir du niveau 100, certains bugs survenaient, Ceci est principalement dû à la gestion des threads qu’il faut encore améliorer, mais qui n’entre pas dans les objectifs principaux de ce projet. Pour plus d’informations, consulter les sections 5.3 – 5.4.

## Erreurs restantes [LAZHAR]

Notre application ne comporte pas d’erreurs « graves », si on s’en tient au cahier des charges. Tout fonctionne parfaitement et nous en sommes très contents. Cependant, beaucoup d’améliorations sont à considérer (qui dépassent évidemment le cadre de ce projet) que nous détaillons dans la section 5.3 - 5.4. Il s’agit principalement du caractère *Temps réel* de l’application ainsi que de la gestion des threads. Mais ceci dépasse le cadre de ce projet, c’est pourquoi nous ne le considérons pas comme étant une erreur.

# Conclusions *[LAZHAR]*

Mener à bien un projet n’est pas une chose aisée, et nous sommes très satisfaits de pouvoir conclure de manière très positive ce travail. En effet, si on s’en tient au cahier des charges initial, les objectifs sont largement atteints. En ce qui concerne l’organisation et la répartition des tâches ainsi que la collaboration des membres, une bonne entente a régné durant tout le projet. Nous sommes donc satisfait et le résultat est au rendez-vous.

Dans les sous-sections suivantes, nous développons en détails quelques conclusions importantes.

## Objectifs atteints / non-atteints

Voici un récapitulatif des objectifs et de leur état, repris du cahier des charges initial.

### Illustrer le concept de graphe de manière ludique et interactive.

L’objectif a été atteint avec succès.

Notre application permet de jouer tout en observant, si on le souhaite, le concept de graphe permettant aux créatures de se déplacer selon un chemin précis.

### Acquérir de l’expérience dans la planification et l’accomplissement d’un projet conséquent.

L’objectif a été atteint avec succès.

Nous avons mené a bien cette tâche grâce notamment à l’utilisation du logiciel ***svn*** permettant de faire des versions à tout moment du travail en cours ainsi que d’organiser au mieux l’évolution du projet.

### Utiliser et découvrir des librairies existantes implémentant le concept de graphe

L’objectif a été atteint avec succès.

En effet, nous avons pour ce projet utilisé deux librairies externes codées en Java. Elles nous ont été très utiles et nous avons appris à manier les librairies externes.

### Apprendre à mettre en œuvre une interface graphique en Java.

L’objectif a été atteint avec succès.

Notre application n’aurait aucune raison d’être sans interface graphique. C’est le point-clé d’un jeu tel que celui que nous avons créé. Il va sans dire que ce point a été plus qu’approfondi.

### Séparer le travail en plusieurs niveaux d’abstraction pour faciliter l’élaboration et l’évolutivité de ce projet.

L’objectif a été atteint avec succès.

En effet, nous avons réparti notre code en fonction de l’architecture bien connue *MVC*. Cette étape n’a pas été facile mais le résultat est très satisfaisant.

### Comprendre la nécessité d’utiliser des algorithmes complexes dans les applications informatiques.

L’objectif a été atteint avec succès.

Une célèbre personnalité de l’informatique a dit un jour : « Algorithms + data structures = programs ». Cette citation est toujours d’actualité, simplement car sans algorithme, notre jeu n’aurait jamais vu le jour. Tout repose en effet principalement sur le fameux algorithme de recherche du chemin le plus court, plus connu sous l’appellation *« Algorithme de Dijkstra »*.

### Mettre en œuvre un algorithme de recherche de chemin le plus court (ACPC).

L’objectif a été atteint avec succès.

L’algorithme ACPC (*Dijkstra*) décrit à la section 2.1.1.4 est l’algorithme-clé de notre application. Il est utilisé tout au long du jeu et est mis en œuvre avec succès.

## Points positifs / négatifs

### Points positifs

Travail en groupe

### Points négatifs

Projet ambitieux, on reste malgré tout un peu sur notre fin

## Difficultés particulières

Gérer un projet en groupe, répartition des tâches, ampleur du projet

Apprendre à utiliser correctement une librairie externe

Aspect *« temps réel »* de l’application

## Avenir du projet

Ce projet, ambitieux au départ, n’est pas sur le point d’être définitivement terminé. En effet, toute une série d’améliorations et de fonctionnalités pourraient être ajoutées. De plus, grâce à l’utilisation d’un logiciel de *« versionage »* avec son serveur associé, le projet peut sans problème rester ouvert. Les membres participant au développement peuvent même évoluer et on pourrait voir naître de nouveaux contributeurs. Tout le principe du développement logiciel en groupe *« Open Source »* est ici mis en œuvre grâce au logiciel ***svn***. De ce fait, le projet reste complètement ouvert à toute perspective d’avenir.

Voici une liste non exhaustive des idées principales et améliorations qui pourraient survenir, par ordre de priorité décroissante :

* Revoir complètement la partie *« temps réel »* de l’application et supprimer la mauvaise gestion des threads pour n’en faire plus que deux : un pour la mise à jour de l’affichage et un pour le calcul des données.
* Corriger tous les bugs existants et ceux qui peuvent survenir.
* Améliorer l’interface graphique pour être encore plus convivial et simple.
* Ajouter une option permettant de couper le volume.
* Pouvoir envoyer chaque nouveau score de chaque joueur sur un serveur afin d’avoir la liste de tous les meilleurs scores existant, et le afficher sur une page Internet.
* Séparer la difficulté du jeu en trois modes : facile, normal et expert.
* Pouvoir enregistrer la partie en cours et la charger à un moment ultérieur.
* Pouvoir mettre le jeu en mode pause.

# Annexes

## Sources – Bibliographie [LAZHAR]

## Journal de bord de chaque participant [LAZHAR]

Voir annexe 6.2.

## Manuel d'Utilisation [LAZHAR]

## Archives du projet [LAZHAR]