Bot pour le jeu NetHack

Chargé de TD : FLEURY Emmanuel **Client : LE BORGNE Yvan**

ARNE Jean-Philippe COLENO Thomas GARNAUD Eve
TESSIER Aymeric

7 Avril 2009

Remerciements

Nous tenons à remercier Monsieur Fleury, notre chargé de TD, pour sa patience et ses conseils. Il nous a guidé au niveau des tâches à accomplir pour réaliser ce PdP mais il a aussi su orienté notre réflexion afin de permettre l'aboutissement du projet. Un grand merci également à Monsieur Le Borgne, notre client, qui nous a permis de travailler sur un sujet passionnant et enrichissant. Il nous a accompagner tout au long de ce semestre avec bonne humeur et enthousiasme en nous encourageant et nous guidant habilement.

Résumé

NetHack est un jeux vidéo inspiré par le jeu de rôle “Donjons et Dragons”. Le but de notre projet est de développer un bot capable de jouer (et éventuellement gagner) à NetHack. Ce projet peut se décomposer en trois objectifs. Tout d'abord, on souhaite insérer du code à la place de l'interface afin de permettre à nos bots de jouer sans tricher. Ensuite, on développera quelques bots spécialisés dans des tâches précises. On validera ces bots par la production de statistiques sur un grand nombre de parties jouées. Et enfin, on essaiera de mettre en place un module décisionnel capable de gérer l'ensemble des spécificités des bots en un seul et unique.

NetHack est écrit en C et est disponible sur le site de référence du jeu : `http://www.nethack.org`.

Table des matières

Chapitre 1

Domaine d'application

Notre intelligence artificielle devra remplacer un joueur humain dans NetHack pour accomplir un certain nombre de tâches. Elle devra être capable d'analyser des situations précises afin de prendre des décisions pertinentes pour remplir au mieux ses objectifs, ou tout simplement de survivre dans un donjon qui est rempli de monstres et de pièges mortels.

1.1 Définition Rogue-like

Rogue est un jeu-vidéo créé en 1980 s'inspirant du célèbre jeu de rôle "Donjons et Dragons". Le joueur y incarne un aventurier qui doit évoluer dans un donjon aux multiples niveaux, rempli de monstres et de trésors. Le jeu se joue sur une grille en deux dimensions, où chaque couloir, chaque porte, chaque escalier, chaque monstre et chaque trésor sont générés aléatoirement. Etant un jeu de rôle, le joueur gagne de l'expérience au cours de ses combats contre les monstres, ce qui le rend plus puissant et plus apte à affronter des monstres encore plus puissants. Un Rogue-like est donc un jeu fonctionnant sur ce même principe.

1.2 Intelligence artificielle (cf : [JN03])

1.2.1 Histoire

Le terme d'intelligence artificielle est probablement apparu pour la première fois dans l'article d'Alan Turing "*Computing Machinery and Intelligence*" (Mind, October 1950), où Turing propose une expérience maintenant connue sous le nom de *test de Turing*. Ce test consiste en un échange textuel entre un homme d'un côté et une machine et un homme de l'autre. Si le premier homme n'arrive pas à différencier la machine de l'humain au cours de la conversation, le test est considéré comme validé.

Ce n'est que quelques années plus tard, en 1956 durant le congrès de Dartmouth, qu'a ensuite été créée l'intelligence artificielle en tant que domaine de recherche.

1.2.2 Définition

C'est la simulation par une machine d'un comportement humain tel que l'apprentissage, l'analyse d'un certain nombre de paramètres, la prise de décision et autres. Les intelligences artificielles sont très utilisées dans le domaine de la finance comme aide décisionnelle. Etant capable de traiter un grand nombre d'informations diverses, elles peuvent suggérer des directions stratégiques à prendre afin de remplir tel ou tel objectif. Elles servent également dans des applications du type « reconnaissance » : reconnaissance vocale, faciale, digitale ... Ici, ce sont les propriétés d'apprentissage des réseaux neuronaux (cf : section 2.5 page 9) qui sont utilisées.

L'un des créateurs de l'intelligence artificielle, Marvin Lee Minsky, définie l'intelligence artificielle comme "la construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisantes par des êtres humains car elles demandent des processus mentaux

de haut niveau tels que : l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique."

1.2.3 Deux types d'intelligence artificielle

Les intelligences artificielles peuvent cependant être scindées en deux parties distinctes. D'un côté il y a les intelligences artificielles évolutives et de l'autre les non évolutives.

Intelligence artificielle évolutive

Une intelligence évolutive est capable d'apprendre de situations déjà vécues et de s'enrichir au fur et à mesure de son cycle de vie. Ce type d'intelligence artificielle utilise les réseaux bayesiens[1] (voir [NWL+07]).

Intelligence artificielle non évolutive

Au contraire, une intelligence non évolutive peut être vue comme un automate ayant un nombre d'états fini où chaque comportement est codé par le développeur. On ne peut pas dire que ce type d'intelligence artificielle soit intelligente, dans le sens où ses actions ne sont pas réfléchies mais programmées à l'avance.

1.2.4 Domaines de l'intelligence artificielle

L'intelligence artificielle est utilisée dans grand nombre de domaines dont en voici des exemples :
- Les systèmes experts : elle rassemble des grandes quantités de données et peut donner des indications stratégiques à une entreprise, une fois toutes les données traités ;
- Le calcul formel : qui traite des expressions symboliques ;
- Le traitement de langage naturel : qui consiste à traduire un texte ou encore à le résumer ;
- La reconnaissance de l'écriture : grâce notamment aux réseaux neuronnaux ;
- La reconnaissance des visages : qui a longtemps été considéré comme un des problèmes les plus difficiles mais qui s'améliore grâce à l'utilisation des réseaux neuronaux.

[1] Les réseaux bayesiens sont définis chapitre 2

Chapitre 2

Analyse de l'existant

2.1 NetHack (cf : [LKM])

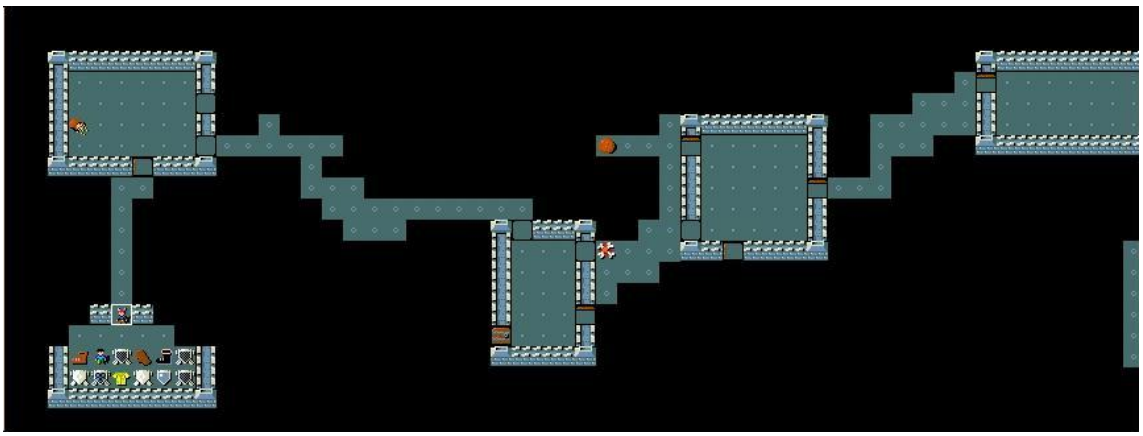

FIG. 2.1 – NetHack

NetHack est un jeu en grille sorti en 1987 qui a été conçu en réseau, c'est-à-dire que plusieurs personnes ont contribué à son développement à travers le monde. NetHack est un logiciel libre et fait parti de la famille des rogue-like (cf : 1.1 page 4). C'est un jeu difficile où le personnage peut trépasser de nombreuses façons comme être tué par un monstre, mourir de faim et bien d'autres ... Le but du jeu est de descendre dans les oubliettes pour récupérer l'Amulette de Yendor et sortir du donjon par le toit. Toute la difficulté du jeu réside dans le fait que le héro doit se nourrir sans avaler de poison, ramasser des objets sans se faire ensorceler ou visiter des pièces sans tomber dans des pièges.

NetHack a été développé pour un grand nombre de systèmes comme Windows, Unix, Mac, ...
L'implémentation de NetHack se décompose en deux partie principales : le noyau du jeu et l'interface du jeu. Lors de la compilation, seuls les fichiers correspondants au système d'exploitation courant sont compilés, ce qui implique donc qu'une interface par plateforme ait été développée et qu'il y ait des fichiers spécifiques par plateforme.

Au début d'une partie de NetHack, le donjon est generé aléatoirement. Ensuite, au cours de la partie, le noyau et l'interface s'envoient des données suivant les déplacements du héro, ses actions, ... Progressivement, nous découvrons une plus grande partie de la carte comme nous pouvons le voir sur la figure 2.1.

2.2 Création d'un niveau personnalisé pour NetHack

Il est possible de créer des niveaux personnalisés pour NetHack et de les insérer dans un donjon. Ceci nous a aidé pour pouvoir faire nos tests dans des niveaux spécifiques afin de savoir précisément comment était fais le niveau.

Nous allons voir dans cette partie comment créer un niveau, l'inclure dans un donjon et le compiler. Pour que cela soit plus simple nous prendrons par la suite le niveau *fortress* et le donjon *dungeon*.

2.2.1 Créer un niveau

Emplacement du fichier

Pour créer un niveau, il faut tout d'abord créer un fichier avec l'extension ".des", ce qui donne pour notre exemple *fortress.des*. Ce fichier est à mettre dans le dossier dat.

Contenu du fichier

Le fichier *fortress.des* doit contenir notament le nom du niveau tel qu'il sera dans NetHack et la configuration de votre niveau entre les balises MAP et ENDMAP. Ensuite, il peut contenir des lignes de commande supplémentaires concernant les objets, les portes (ouvertes ou fermées par exemple) ou encore les monstres.

Attention au nom du niveau que vous donnez car dans NetHack les noms de niveaux doivent être au maximum de 8 caractères.

La liste des caractères permettant la construction du niveau est la suivante :

```
'-'     horizontal wall
'|'     vertical wall
'+'     a doorway (state is specified in a DOOR declaration)
'A'     open air
'B'     boundary room location (for bounding unwalled irregular
        regions)
'C'     cloudy air
'I'     ice
'S'     a secret door
'H'     a secret corridor
'{'     a fountain
'\'     a throne
'K'     a sink (if SINKS is defined, else a room location)
'}'     a part of a moat or other deep water
'P'     a pool
'L'     lava
'W'     water (yes, different from a pool)
'T'     a tree
'F'     iron bars
'#'     a corridor
'.'     a normal room location (unlit unless lit in a REGION
        declaration)
' '     stone
```

Voici un extrait de fichier ".des" :

```
MAZE : "fortress", random
GEOMETRY : center , center
MAP
}}}}}}}}}
}}}|-|}}}
}}|-.-|}}
}|-...-|}
}|.....|}
}|-...-|}
}}|-.-|}}
}}}|-|}}}
}}}}}}}}}
ENDMAP
```

Pour de plus amples informations, vous pouvez consulter la page de la référence [CC] qui contient les commandes lex de l'analyseur de fichiers, ainsi qu'un exemple de niveau.

2.2.2 Insérer un niveau dans le donjon

Une fois le niveau créé, il faut l'inclure dans un donjon. Pour ce, il faut éditer le fichier *dungeon.def* du répertoire dat. Dans ce fichier, il ne reste maintenant plus qu'à insérer votre niveau suivant l'endroit dans le donjon où vous souhaité qu'il apparraisse.

Pour de plus amples informations, vous pouvez consulter la page de la référence [Ste] qui contient l'explication de la grammaire d'un fichier ".def" ainsi qu'un exemple de donjon.

2.2.3 Compiler un niveau

Pour Unix

Il faut maintenant inclure le niveau dans le bon Makefile afin qu'il soit compilé et généré en un fichier portant l'extension ".lev", ce qui nous donne *fortress.lev*. Pour ce, il faut modifier le Makefile du dossier dat en rajoutant le niveau avec les autres.

Pour Windows avec Visual Studio

Pour compiler le niveau sous Windows, il faut ouvrir le projet sous Visual Studio et aller dans les propriétés du projet levcomp. Dans les propriétés, il faut aller dans la section "Evenements de génération" puis "Evenement après génération". Il suffit maintenant de rajouter dans "Ligne de commande" la ligne permettant de compiler le niveau, de la même façon que les autres niveaux sont compilés.

Voici la ligne qu'il faudrait rajouter pour notre exemple :

```
..\util\levcomp.exe fortress.des
```

2.3 Les interfaces graphiques dans nethack

Nethack est un jeu multi-plateformes, et pour chaque plateforme dispose d'une ou plusieurs interfaces graphiques. Lors de la compilation, il est possible de compiler toutes les interfaces graphiques d'une plateforme. Ensuite, il faut modifier le fichier *config.h* pour choisir l'interface graphique qui devra être utilisée pour lancer le jeu. Exemple :

```
#define MSWIN_GRAPHICS
```

Concrètement, les noms des fonctions de chaque interface graphique compilée sont intégrés dans une structure window_procs. Il suffit ensuite d'utiliser les fonctions du window_procs correspondant à l'interface graphique choisie. Pour une plus grande clarté d'écriture du code, des macros ont été faites.

2.4 TAEB (cf : [FSF])

TAEB, Tactical Amulet Extraction Bot, est un bot pour NetHack qui remplie le rôle d'un joueur. On peut lui affecter des priorités pour remplir plusieurs objectifs précis. Il a été écris en Perl et conçu en deux parties :

- d'une part les besoins du bot, comme par exemple se nourir, stockés dans le répertoire `lib/TAEB/AI/Behavoiral/Personality` ;
- et d'autre part une partie qui est capable de prendre des décisions aux vues des besoins du bot. Par exemple, si la faim est considérée comme le but le plus vitale pour le bot, le bot décidera de manger afin de ne pas mourir. Ces comportements sont développés dans le répertoire `lib/TAEB/AI/Behavoiral/Behavior`.

La partie prise de décision est paramétrable ; c'est-à-dire qu'il est possible de dire au bot d'aller par exemple au plus bas dans le donjon ou de faire un maximum de point. Ces paramètres influent sur les priorités de chaques besoins. Ils sont définissables dans le fichier *config.yml* du répertoire etc.

Ce bot va donc nous aider à faire le lien entre besoin et stratégie pour le réaliser. Nous verrons également comment prendre en compte les données fournies dans le fichier de configuration. Cependant, le fait qu'il soit développé en Perl nous pose de sérieux problèmes de compréhension des parties techniques du code.

2.5 Les réseaux bayesiens

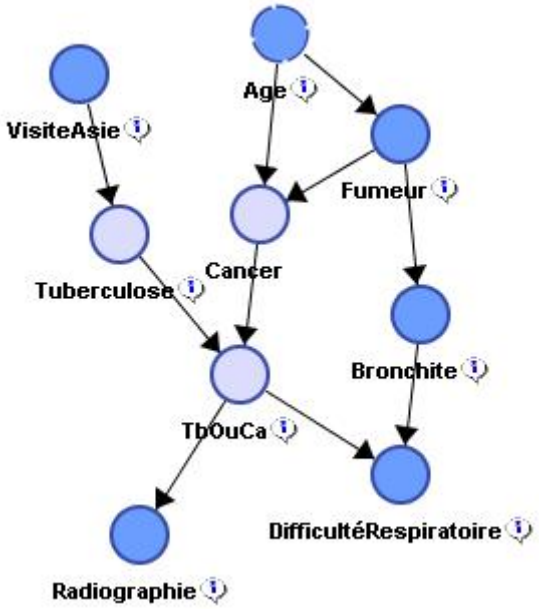

FIG. 2.2 – Ce schéma d'un réseaux bayesien représente le diagnostic d'une maladie des poumons, des causes aux symptômes.

Le nom réseau bayesien provient du révérend Bayes et plus précisément de son Théorème, mais il n'y a pas toujours de rapprochement entre un modèle de type "réseau bayesien" et la "statistique bayesienne". C'est pourquoi le terme *diagramme d'influence*, qui est plus général et dénué de connotation est parfois utilisé.

Les réseaux bayesiens sont à le fois des modèles de représentation des connaissances et des "Machines à calculer" les probabilités conditionnelles.

Ce type de réseau peut se voir comme un graphe causal acyclique où les noeuds représentent des variables aléatoires, et les liens des relations de cause à effet entre les variables. Les flèches représentent des relations entre variables qui sont soit déterministes, soit probabilistes (cf : [NWL+07]). Dans notre cas, les valeurs des sommets représenteront les besoins du bot (chiffrés), et les liens des actions possibles préprogrammées pour agir sur ces besoins. Avec cette représentation sous forme de graphiques nous avons une modélisation qualitative (l'événement A implique l'événement B) mais aussi quantitative (si la variable A est modifiée, nous avons par exemple 60% de chance pour que la variable B soit également modifiée). Grace à ce type de graphe, nous représenterons l'intelligence artificielle du bot.

Les réseaux bayesiens sont surtout utilisés pour le diagnostic, l'analyse de risques, la détection des spams et le data mining. C'est une description qualitative et quantitative des dépendances entre les variables.

2.6 Les algorithmes de déplacement

2.6.1 L'algorithme A* (A-Star)

Cet algorithme est actuellement le plus utilisé pour les jeux-vidéos car il offre de nombreux avantages. En effet, le temps nécessaire pour calculer un plus court chemin est très interressant pour des applications en temps réel telles que les jeux (cf : Chapitre 7 [eGS04]). En plus de cela, il est possible d'affecter des poids spécifiques à des terrains particuliers, comme par exemple une case piège aura un poids défavorable pour son franchissement ce qui provoquera son évitement par le bot.

Fonctionnement

Le plus souvent, l'algorithme A-Star est utilisé pour trouver un chemin dans un graphe, et les paramètres en entrés sont le nœud de départ, le nœud d'arrivée et bien sûr le graphe.

L'algorithme comprend deux listes, une appelée liste ouverte qui contient les nœuds à analyser, et une appelée liste fermée qui contient les nœuds déjà analysés.

Pour commencer, l'algorithme place le nœud de départ dans la liste ouverte. Ensuite, il le traite en ajoutant tous les nœuds adjacents dans la liste, et en indiquant leur parent ainsi que leur poids (distance parcourue depuis le nœud de départ). Lorsque tous les nœuds adjacents ont été traités, le nœud de départ est retiré de la liste ouverte et placé dans la liste fermée.

Viens maintenant une boucle de recherche. Un nœud ayant le plus petit poids parmi ceux de la liste ouverte est alors choisi. Il faut maintenant en étudier les cases adjacentes. Pour chaque case adjacente :

- si elle est dans la liste fermée, on l'ignore ;
- si elle est dans la liste ouverte, on recalcule son poids et s'il est infèrieur on remplace le père par le nœud courant ;
- si elle n'est dans aucune liste, on la place dans la liste ouverte et on calcule son poids.

L'algorithme s'arrête lorsque l'on ajoute le nœud d'arrivée à la liste ouverte ou lorsque la liste ouverte est vide. Il suffit ensuite de remonter les parents pour obtenir le chemin.

2.6.2 L'algorithme de Bresenham

Grace à cet algorithme, il est possible de calculer le chemin le plus direct d'un point à un autre. Il est en général utilisé pour tracer des segments sur un écran. L'algorithme prend en paramêtre deux pixels ou deux cases de coordonnées (x1, y1) et (x2, y2), et l'objectif est de calculer quels sont les pixels qui vont devoir être noircis ou les cases à emprunter pour obtenir une approximation du segment souhaité. Cet algorithme garantit le tracé d'un segment continu en un temps de calcul relativement faible.

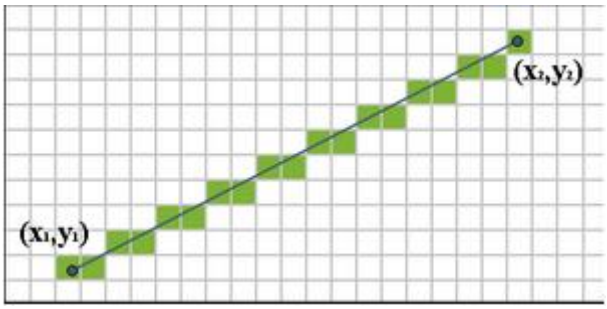

FIG. 2.3 – Résultat de calcul de l'algorithme de Brésenham

Fonctionnement

L'algorithme de base prend en entrée les coordonnées du points de départ (x1, y1) et celles du point d'arrivée (x2, y2) sous la forme de quatre entiers. Le but est de trouver, pour toutes colonnes x entre x1 et x2, la ligne y correspondante qui soit au plus proche du segment. Ainsi, nous obtenons les coordonnées (x, y) à noircir ou la case à parcourir. Nous garantissons donc que le chemin entre les deux points est une suite de cases voisines.

Nous paramètrerons les cases de façon à ne pas rester bloqué face à un mur. Si la sitiation se produit, la première trajectoire à calculer sera celle qui mène vers la porte puis nous suivrons si besoin un couloir.

Chapitre 3

Introduction au projet

Le but de notre projet est de créer un bot capable de jouer au rogue-like NetHack. Pour ce faire, nous avons récupéré le code existant de NetHack pour y intégrer nos modules.

3.1 Introduction à l'architecture de NetHack

Le code existant de NetHack est principalement divisé en deux parties distinctes, le moteur du jeu (le noyau) et l'interface du jeu. Du fait que NetHack ait été développé pour plusieurs systèmes, le code existant comprend plusieurs interfaces, et lors de la compilation du jeu, seuls les fichiers nécessaires sont compilés. Ensuite, au cours d'une partie, le noyau et l'interface s'envoient des informations : l'interface envoyant les ordres que le joueur donne au héro, et le noyau les mises à jours correspondantes à ces ordres. Par exemple, si le joueur donne l'ordre de se déplacer, le noyau renverra à l'interface la ou les cases de terrain nouvellement découvertes. A chaque nouvelle partie, le donjon est généré aléatoirement, avec parfois des niveaux des précédentes parties.

Une autre chose est que le "main" est dans l'interface de NetHack, c'est-à-dire que c'est l'interface qui lance le jeu.

3.2 Intégration de notre bot

Notre bot se placera donc à la place d'une interface, c'est-à-dire qu'il récupèrera les mêmes données qu'une interface, les analysera et effectuera ensuite une nouvelle action suivant la configuration dans laquelle il se trouvera. Les données qu'il enverra seront récupérées par le noyau et le jeu fonctionnera ainsi de la même façon que s'il y avait un utilisateur. De plus, comme il sera à la place d'une interface, notre bot aura accès aux mêmes informations qu'un joueur, ce qui implique donc qu'il ne pourra pas tricher.

3.3 Les modules

NetHack est un jeu assez complet et difficile qui offre de nombreuses possibilités au héro. C'est pour cela que nous avons pour but de développer plusieurs modules, qui correspondront chacun à un objectif spécifique du bot. Il y aura par exemple un module de déplacement, que l'on validera en regardant les performances du bot en exploration.

Ces modules sont des classes qui seront utilisées par notre intelligence artificielle, et qui lui permettront de faire ses choix en fonction de l'environnement et de la situation dans laquelle se trouvera le bot.

3.4 Les tests

Pour pouvoir tester les capacités de notre bot à atteindre un objectif précis, nous allons créer des donjons spécifiques par module. En effet, si l'on veut par exemple tester la capacité de notre bot à explorer

le plus de terrain possible, les parties seront lancées sur un donjon qui ne contient pas de monstres et où le bot ne peut pas mourir de faim. De plus, puisque notre bot n'aura pas accès à certaines informations, un module pour les statistiques englobera notre bot et le noyau du jeu. Ce module pourra récupérer les informations telles que le nombre de portes cachées dans un donjon (données auxquelles le bot n'aura pas accès) et ainsi faire une étude suivant le nombre de portes trouvées par rapport au nombre de portes totales existantes.

Les tests seront effectués régulièrement tout au long du développement, et surtout lors d'ajout de nouvelles fonctionnalités où les tests unitaires nous permettront de nous assurer que tout le travail précédent n'a pas été mis à mal.

Notre projet étant basé sur une étude statistique, puisqu'il faut prouver au client que notre bot est performant, nous ne pouvons pas nous passer de tests et c'est pour cette raison que nous créerons des tests spécifiques à chaque module, afin de pouvoir quantifier les performances de nos algorithmes et les améliorer.

Chapitre 4

Cahier des charges

4.1 Besoins non fonctionnels

4.1.1 Langage de programmation

Tout d'abord, le choix du langage de programmation s'est porté sur C. Ce choix a été fait car le code existant de NetHack est en C, ce qui permettra de facilement nous intégrer au code. Avec ce langage, nous pourrons réutiliser des algorithmes existants, en particulier pour gérer les déplacements et l'intelligence artificielle.

Risques

Ce langage nous impose de mettre au point un certain nombre de conventions de développement puisqu'aucune norme ne nous contraint. Cependant, dans un soucis de lisibilité et de cohérence, nous devons créer un code uniforme.

Parades

Avant de commencer à coder notre bot, nous allons élaborer notre standard de développement en prenant soin de s'approcher le plus possible de ce qui existe déjà. Par exemple, nous donnerons à nos variables des noms, en anglais, rappelant leur utilisation.

4.1.2 Portabilité

Le développement se fera pour la plate-forme Linux, et une portabilité du code sera mise en place (tests sous Windows). Nous développons notre logiciel pour la plate-forme Linux car cela nous a été demandé par le client, et nous faisons la portabilité sous Windows pour toucher un maximum d'utilisateurs. De plus, le client nous a demandé un code portable. Le choix des bibliothèques que nous allons utilisées sera fait en fonction de leur portabilité.

Risques

Le risque majeur est une perte de temps importante dûe à la recherche de bibliothèques portables sur les deux plate-formes. Nous devrons, au besoin, écrire deux codes différents pour répondre à ce critère ce qui peut jouer en notre défaveur.

Parades

Régulièrement, nous testerons le programme sous les deux plate-formes de façon à ne pas avoir à refaire une partie importante du code. Ainsi, la perte de temps sera minimale puisque nous résoudrons ces problèmes au fur et à mesure.

4.1.3 Environnement de développement

L'outil de développement utilisé sera Visual Studio car les sources de NetHack sont prêtes pour cette IDE, et Visual Studio nous permet de travailler efficacement car nous sommes habitués à cet outil de développement.

Risques

Visual Studio étant un outil spécifique à Windows, nous ne devons pas perdre de vue la contrainte de portabilité imposée par le client.

Parades

Cet outil nous servira essentiellement pour la lecture du code car nous sommes plus à l'aise avec cet environnement mais nous utiliserons également emacs que nous connaissons bien sous Linux.

4.1.4 Algorithmes utilisés

Les algorithmes utilisés devront être efficaces en temps et en poursuite des objectifs du bot. Ils devront être rapides en exécution et efficaces afin que le bot soit performant et que l'on puisse faire des jeux de tests importants. L'efficacité des algorithmes est à définir suivant un compromis entre qualité et rapidité.

Risques

Cette étude de compromis entre qualité et rapidité est très subjective, elle risque donc d'imposer au sein du groupe un grand nombre de débats. Nous devrons faire varier plusieurs paramètres et tester plusieurs méthodes avant d'aboutir à une version optimale des algorithmes. De plus, étant novice dans le domaine de l'intelligence artificielle pour la plupart d'entre nous, un important travail de recherche sera à fournir avant de débuter la phase de programmation.

Parades

Il nous est demandé de créer un module de calcul de statistiques qui nous permettra de comparer les algorithmes de façon précise. En fonction des besoins spécifiques du bot, il peut être intéressant d'utiliser des algorithmes différents qui répondrons le mieux aux stratégies employées.

4.1.5 Intelligence artificielle

L'IA développée sera non évolutive. Nous allons créer un bot capable de jouer à NetHack à la place d'un utilisateur, c'est-à-dire prendre la meilleure décision dans une configuration de jeu donnée. Il aura accès aux mêmes informations qu'un joueur et ne pourra donc pas tricher.

Risques

Le risque le plus important est lié à la collecte et à l'analyse des données. Cette étape doit être rapide pour ne pas ralentir l'exécution du jeu mais elle doit aussi prendre en compte un grand nombre d'informations pour être efficace.

Parades

Nous porterons un soin particulier à la mise en place de structures adaptées. Ainsi, l'enregistrement et la lecture des données se fera sans peine.

4.1.6 Tests de comparaison

Enfin, des tests de comparaison vont nous permettre de visualiser les évolutions de notre bot au cours des différentes implémentations de nos algorithmes. Ce besoin va induire les besoins fonctionnels de rejouer un donjon, de tracer un bot et de remontrer une partie.

Risques

Pour faire des tests comparatifs, il faut pouvoir récupérer des données qui seront stockées soit dans le noyau du jeu, soit dans l'interface. L'accès à ces informations peut nous poser quelques problèmes puisqu'il faut savoir les trouver et les interpréter.

Parades

Au cours d'une partie, nous pouvons enregistrer l'état actuel de la carte découverte, le nombre de points de vie et toutes les informations qui nous seront utiles. Il suffit de savoir lire ce fichier de sauvegarde pour les récupérer.

4.2 Besoins fonctionnels

L'ordre d'apparition des besoins fonctionnels est fait en fonction de leur importance attribuée par le client pour ce projet.

4.2.1 Intéraction avec le jeu pour lancer un bot

Il faut pouvoir s'interfacer entre le noyau de NetHack et son interface pour exécuter le bot de la même façon qu'un joueur. On doit donc être en mesure de transmettre les commandes envoyées par le noyau au bot et à l'interface mais aussi celles envoyées par le bot et l'interface au noyau.

Risques

Le risque majeur de cette étape est de ne pas prévoir d'envoyer une réponse attendue par le noyau et donc de provoquer des situations d'inter-blocage.

Parades

Pour résoudre ce problème, nous avons prévu une réponse type à renvoyer à toutes les questions que l'on ne saurait pas interpréter. Même si ce n'est pas toujours optimal, le bot ne sera jamais bloqué.

4.2.2 Evaluation des performances du bot

Pour valider un bot, nous devrons tester ses performances et le comparer avec ceux implémentant des algorithmes différents. Ce travail nous permettra de créer un bot au fonctionnement optimal par rapport à ses exigences : un algorithmes peut être très efficace dans un certain cas de figure et mauvais dans un autre.

Risques

Considérer qu'un bot est performant est difficile puisqu'il faut faire varier un grand nombre de critères. De plus, un algorithme répondant à 60% de nos attentes peut être considérer comme correct alors qu'il peut être améliorer et un autre qui serait efficace sur 50% des cas va nous demander beaucoup de temps alors qu'il ne peut pas être optimiser.

Parades

Nous mettrons en place un module de calcul de statistiques qui nous permettra d'estimer notre bot sur des données chiffrées. Les choix d'algorithmes à utiliser seront établis préalablement pour ne pas perdre trop de temps à la recherche de meilleures performances.

4.2.3 Rejouer un donjon

Un objectif est aussi de pouvoir rejouer un donjon déjà joué. Les donjons de NetHack sont générés aléatoirement et le client nous a demandé de pouvoir rejouer un niveau déjà joué pour pouvoir faire des tests de comparaison.

Risques

Il faut donc être en mesure de récupérer toutes les données aléatoires ou de savoir modifier le générateur de nombres aléatoires pour le relancer à l'identique.

Parades

Le module Tracer enregistrera toutes les données relatives au donjon et nous pourrons ainsi lancer un donjon personnalisé qui correspondra en réalité à notre sauvegarde.

4.2.4 Caractéristiques du bot

Le déplacement

Le bot doit être capable de se déplacer, c'est-à-dire avancer de case en case mais aussi identifier une porte lorsqu'il en rencontre une et être capable de l'ouvrir.

Risques Le risque principal est que nous ne soyons pas capable de faire bouger notre bot, à savoir utiliser les bonnes commandes aux bons endroits. Un autre risque est que notre bot ne soit pas en mesure de reconnaître qu'une case est en réalité un mur ou une porte.

Parades Pour contrer ce risques, nous avons codé une structure case contenant les caractéristiques de la case comme le fait que ce soit une porte ou un escalier. Nous paramètrerons les algorithmes de déplacement en tenant compte de cette spécificité ce qui nous permettra de ne pas rester bloquer face à un mur par exemple.

Nous avons aussi étudier le code existant de NetHack pour comprendre quelles étaient les commandes de déplacement, ou les envoyer et comment les envoyer.

L'exploration

Le bot doit être capable d'explorer la pièce dans laquelle il se trouve, et ensuite d'explorer le niveau puis d'emprunter les escaliers pour continuer son exploration.

Risques Le risque principal est que le bot ne soit pas capable de faire la différence entre les différents éléments du niveau. En effet, s'il ne fait pas la différence entre un mur et une porte, jamais il ne pourra ouvrir les portes et donc explorer un niveau.

Parades Pour éviter ce problème, nous avons là aussi étudié le code existant de NetHack, et utilisé notre structure efficacement afin de stocker le type de chaque case et s'en souvenir.

La recherche de portes

Notre bot doit être capable de chercher les portes secrètes afin de d'explorer un maximum de terrain et de pouvoir se débloquer de certaines situations.

Il n'y avait pas vraiment de risque ici puisqu'à ce niveau là du développement, nous étions capable d'effectuer des commandes ; il nous suffisait donc juste d'utiliser la commande permettant de chercher.

Chapitre 5

Architecture

NetHack possède déjà un code complet et complexe, ce qui nous impose une réponse architecturale adéquate. Pour cela, nous avons dû séparer notre bot en plusieurs sous-blocs. Nous avons un bloc de communication avec NetHack, un autre bloc de gestion du bot (prise de décision, besoins et stratégies) et un bloc indépendant des deux premiers : le module statistique.

Notre bot est constitué des trois parties :

- les données auxquelles le bot a accès dans le module World ;
- les opérations à effectuer dans les modules Movement, Food, Inventory et Fight ;
- les décisions seront prises dans le module Artificial Intelligence.

5.1 Schéma d'architecture

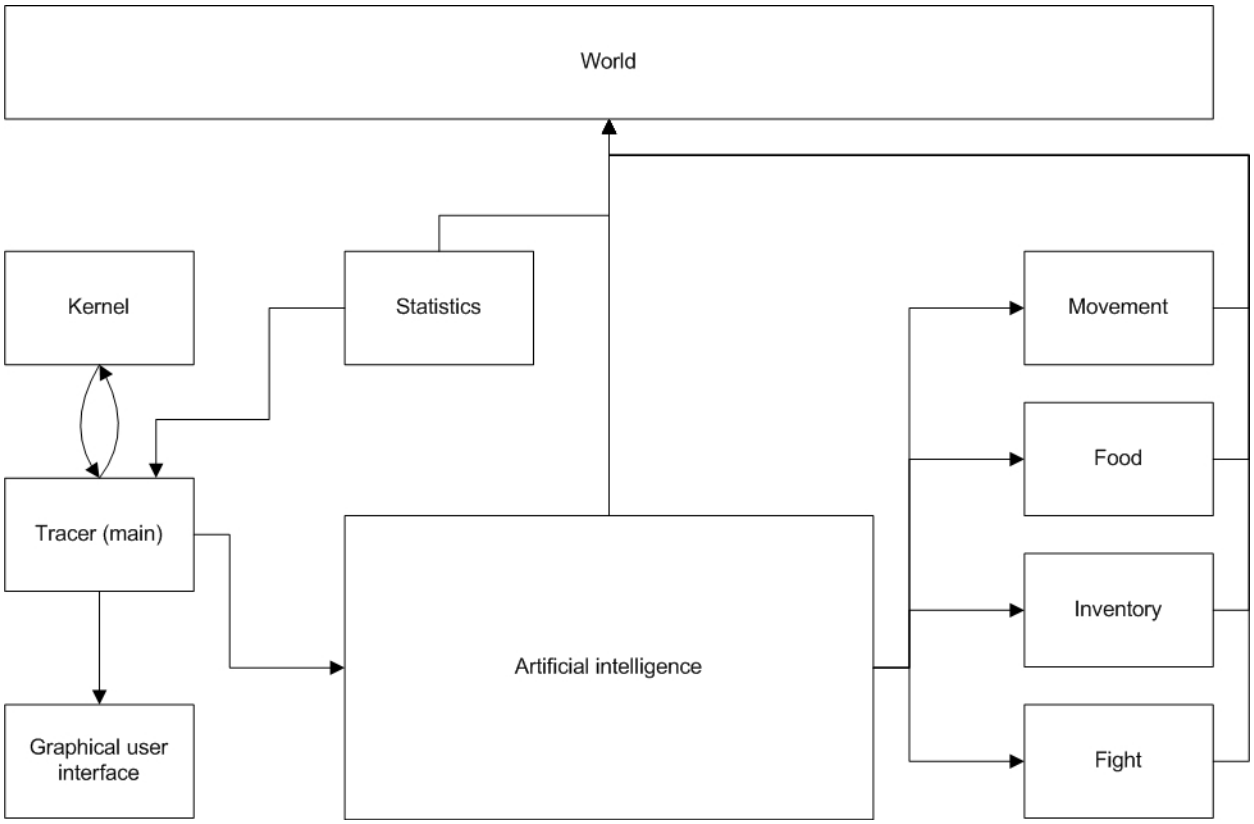

Fig. 5.1 – Architecture

5.2 Description des modules

5.2.1 Module World

Ce module nous servira à stocker toutes les informations dont le bot a accès, ce sont donc les données qu'un joueur voit. Elles sont déjà enregistrées, il suffit donc de les mettre en forme dans des structures appropriées.

Nous devons sauvegarder la carte visitée par le bot. Une carte est un donjon qui se décompose en plusieurs niveaux qui sont formés d'un ensemble de cases. Nous avons donc implémenté plusieurs structures nous permettants de gérer l'arborescence des niveaux.

Tout d'abord voici la structure bot_square qui représente une case d'un niveau avec ses attributs.

```
typedef struct bot_square {
        //type de la case du pointde vue MAP (porte, couloir, mur...)
        int type;
        int weight;
        //type de l'élément sur la case (si il y en a un, -1 sinon)
        //s'il y en a plusieurs, c'est celui qui s'affiche
        //à l'écran qu'on retiend
        int thing;
        //nombre de fois que la case a été fouillée
        int nb_search;
        //indice indiquant que la case doit être fouillé immédiatement
        int mustSearch;
        //entier pour savoir si la case a déjà été découverte
        int visited;
} bot_square;
```

Ensuite, voici la structure nous mermettant de stocker les escaliers, qui nous permettent de nous déplacer entre les niveaux.

```
//structure pour stocker les escaliers
typedef struct bot_stair {
        stairway stair; //structure de NetHack pour un escalier
        struct bot_level *toLev; //niveau vers lequel mène l'escalier
} bot_stair;
```

Nous avons bien entendu une structure représentant un niveau, avec la grille du niveau, une liste des escaliers du niveau et d'autres attributs.

```
typedef struct bot_level {
        bot_square grid[COLNO][ROWNO];
        bot_list *stairs; /*qui contient des bot_stair*/
        d_level level; //structure de NetHack
        int searchLevel;
        int botNbCasesFound;
        int botNbDoorsFound;
        int botNbSDoorsFound;
        int visited;
} bot_level;
```

Et enfin nous avons une structure pour le donjon qui contient le premier niveau dans lequel débute le jeu.

```
typedef struct bot_dungeon {
        bot_level *root;
} bot_dungeon;
```

Le bot doit pouvoir se situer sur la carte, pour cela, nous sauvegarderons sa position dans la structure suivante :

```
struct bot_pos {
    int x; //abscisse de la case sur la grille
    int y; //ordonnée de la case sur la grille
    int lastX; //abscisse au tour précédent
    int lastY; //ordonnée au tour précédent
    int freezed; valeur pour savoir si le bot est bloqué
}bot_pos;
```

Gestion de la mémoire

Afin de pouvoir facilement libérer l'espace mémoire utilisé lors de l'utilisation de notre bot, nous avons mis en place une liste permettant de stocker tous les pointeurs que nous allouons. De cette façon, à la fin du jeu, il nous suffit de parcourir les éléments de la liste un par un et de les désallouer.

5.2.2 Module Tracer

Ce module s'interfacera entre le noyau du jeu et son interface tty. La fonction main sera définie ici, c'est donc ce module qui aura en charge de lancer le bot mais aussi d'interpréter le fichier de configuration et les instructions envoyées au noyau ou par le noyau.

```
struct GameOptions {
    int monster; //booléen précisant la présence ou non de monstres
    char* dungeon; //nom du donjon à charger
    int explore; //booléen indiquant le mode de jeu explorer
    ... //autres options ajoutées en fonction des besoins
};//options récupérées dans le fichier de configuration

char* saveAndExecute(char* command);
void read(char* file);
void write(char* file, char* text);
```

Nous pourons ainsi sauvegarder toutes les actions faites par le bot de façon à permettre de rejouer une partie. Ce traitement sera effectué par la fonction saveAndExecute qui prend en entrée une commande, la transmet au noyau et l'enregistre dans un fichier.

C'est dans ce module que nous implémenterons les fonctions de lecture et d'écriture dans un fichier qui nous servirons aussi dans le module Statistics.

5.2.3 Module Artificial intelligence

Ce module se chargera de gérer les comportements du bot. C'est ici que seront implémentés l'intelligence artificielle et les algorithmes de prise de décision.

```
int needs[MAX_NEEDS];
char bot_decision();
```

Pour cela, il prendra en compte les objectifs fixés par l'utilisateur au démarrage du jeu. Ces objectifs influeront sur l'évolution des différents besoins élémentaires du bot. En fonction de ces besoins, le bot définira ses buts à atteindre à très court terme. C'est à dire que si le besoin est d'explorer mais que son état de faim est critique, son premier besoin sera de manger afin de ne pas mourir. A chaque tour, il faudra qu'il mette à jour la liste de ses besoins.

La fonction bot_decision décide quelle stratégie adopter pour répondre au besoin considéré comme le plus urgent. Elle renvoit donc au traceur la commande à effectuer pour le tour courant (ou la combinaison de commande si cela est nécessaire).

5.2.4 Module Movement

Ce module permettra de gérer les déplacements du bot. Chaque fonction définie ici renverra au module Artificial Intelligence une ou plusieurs commandes à effectuer via une liste de commandes.

```
void bot_explore();
void bot_goDownWithStairs();
void bot_searchHidenWay();
```

Suivant le besoin courant, le bot a le choix entre ces trois stratégies. Chacune de ses stratégies a ses propres effets, mais elles produisent toutes trois des coordonnées d'objectif à atteindre. Avant de se terminer elles font donc toutes appel à l'algorithme A* pour trouver un chemin vers leur objectif.

5.2.5 Module Food

Ce module va permettre au bot de se nourrir et de rechercher de la nourriture si besoin est.

```
char eat();
int foodStockEval();
```

Pour se nourrir, nous implémenterons une fonction eat() qui renverra la commande pour manger ou -1 s'il manque de la nourriture. Pour savoir si le stock de rations est suffisant, nous implémenterons la fonction foodStockEval() qui va regarder la quantité de rations disponible et renverra un entier suivant cette valeur. L'entier renvoyé permettra de définir le niveau de priorité du besoin de rechercher de la nourriture.

5.2.6 Module Inventory

```
char getObject();
char dropObject(Object o);
int evalUtility(Object o);
```

Dans ce module il sera question de l'inventaire. Le bot doit être en mesure de ramasser un objet avec la fonction getObject(). Il faut aussi, lorsqu'il est trop lourd ou qu'il détient des objets inutiles, qu'il puisse les poser avec la fonction dropObject(object o). Après ces opérations, il devra mettre à jour le contenu de son sac.

Le bot doit pouvoir évaluer l'intérêt et l'utilité de chaque objet.

5.2.7 Module Fight

Ce module sert à déterminer la faisabilité d'un combat mais aussi à effectuer le combat.

```
int victoryEval(monstre m);
char attack(int x, int y)
int*  escapeArea();
int restRisk();
char rest();
```

La fonction victoryEval prend en paramètre un monstre et selon les différentes caractéristiques du bot, elle renvoie un indice correspondant aux chances de gagner. La fonction attack va attaquer un monstre et décider par elle même suivant la position passée en paramètre (celle du monstre), de quelle arme utiliser : jet ou mélée. La fonction escapeArea renvoie une coordonnée correspondant à un point éloigné de la zone de combat afin que le bot sème ses poursuivants. La fonction restRisk évalue la nécessité de se reposer combinée à la faisabilité de cette action, c'est-à-dire est-ce qu'il n'y a pas de monstre dans les alentours pour ne pas se faire attaquer pendant le repos. La fonction repos renvoie la commande pour se reposer.

5.2.8 Module Statistics

A chaque niveau visité, le noyau sauvegarde les données relatives au niveau. Nous devons donc les récupérer afin de produire une analyse statistique des comportements du bot. Il est par exemple important de connaître le nombre de portes présentes sur le niveau pour savoir quel pourcentage de portes a été ouvertes. Nous connaitrons aussi le nombre de monstres tués, le pourcentage de map découverte en étudiant les données collectées par le module Tracer.

Notre structure pour les statistiques

Nous avons créé une structure pour pouvoir associer un *bot_level* avec son nombre de cases total, son nombre de portes et portes cachées total. En effet, pour que notre bot ne puisse pas tricher, nous ne pouvions pas mettre ces informations directement dans la structure *bot_level*. C'est pourquoi nous avons créé une nouvelle structure dans botStats.h.

```
typedef struct bot_stats {
        bot_level *level;
        int nbCases;
        int nbDoors;
        int nbSDoors;
}bot_stats;

extern bot_list *botListStats; /*liste contenant des bot_stats*/
```

Pour parcourir facilement les instances de cette structure nous avons aussi créé la liste *botListStats*.

5.3 Les besoins du bot

Les besoins correspondent à des valeurs qui définissent les stratégies à appliquer à un instant donné de la partie. Ces valeurs sont le résultat d'une analyse systématique (à chaque tour) de la situation via des fonctions appropriées.

5.3.1 Besoin d'explorer

Ce besoin se calcule par rapport au nombre de cases découvertes et le nombre de cases inexplorées (noires). Etant donné qu'on ne sait jamais exactement combien de cases sont utilisées par niveau, il faudra tempérer cette valeur en enlevant par exemple des cases noires. Si ce besoin ne peut être satisfait, l'intelligence du bot augmentera le besoin de trouver des portes secrètes et le besoin de descendre, afin de débloquer la situation. Dans la version que nous avons développer, le bot se contente de calculer ce besoin en fonction des cases encore inexplorées. S'il en reste le besoin est fort, sinon il est faible.

5.3.2 Besoin de trouver des portes secrètes

Typiquement, ce besoin augmente lorsque le bot se retrouve face à une situtation de blocage. Par exemple, il a exploré toute la carte mais il considère qu'il reste des zones à découvrir. Ou encore, il a tout exploré et n'a pas trouvé d'escalier pour descendre d'un étage.

5.3.3 Besoin de descendre

Plus un étage sera exploré, plus ce besoin augmentera, jusqu'à arriver au point où l'exploration de l'étage sera considérée comme terminée. Le bot devra alors répondre à ce besoin.

5.4 Stratégies en réponse aux besoins

Nous avons établis une stratégie correspondant à chaque besoin pour pouvoir les diminuer.

5.4.1 Stratégie d'exploration

Cette stratégie consiste à explorer le plus de terrain possible. Le but est donc de se diriger en premier vers les zones non découvertes les plus proches. Pour cela, il faudra emprunter les couloirs ou ouvrir les portes pour aller dans une autre salle. Moins il y aura de terrain inexploré, plus les valeurs des besoins de chercher les portes secrètes et de descendre au niveau inférieur augmenteront.

5.4.2 Stratégie pour trouver les portes secrètes

La stratégie pour chercher des portes consiste à longer les murs tout en cherchant. De plus, lorsqu'un couloir ne mène nul part, il est judicieux de chercher s'il n'y a pas de portes secrètes ou de cases secrètes. Une autre astuce est que si l'on se trouve dans une salle en bas à gauche du niveau, les portes secrètes se trouvent probablement en haut à droite de la pièce. Pour améliorer la recherche de porte secrètes et la rendre plus rapide, il sera judicieux de par exemple se déplacer en diagonale dans les coins ou lieu de parcourir toutes les cases.

5.4.3 Stratégie pour descendre dans un autre niveau

Il suffit de se déplacer vers l'escalier menant au niveau inférieur.

Chapitre 6

Conventions de codage (cf : [Lab])

Afin de garantir la lisibilité du code et de faciliter sa maintenance, nous avons mis en place un certain nombre de conventions et de règles à respecter tout au long du développement de notre projet.

6.1 Nommage et structuration des fichiers

Dans un soucis de cohérence, les noms des fichiers ont un nom "parlant", c'est-à-dire qu'ils reflètent le rôle du fichier. De plus, le nom de tous les fichiers ajoutés pour utiliser le bot sont préfixés par "bot" (à l'exception des fichiers winBot.h et .c pour respecter les conventions de nethack). Il n'y a pas de limitation de taille dans un fichier, mais nous avons tenté de simplifier au maximum le nombre de lignes de codes dans chaque fichier tout en s'assurant que les lignes ne soient pas trop longues (une instruction ou un test par ligne). Ainsi, un fichier sert uniquement à un traitement particulier explicité dans son nom.

6.1.1 Les fichiers source

Chacun de nos fichier source (d'extention ".c")commence par un court commentaire expliquant son utilisation et son intérêt. Viennent ensuite les inclusions de fichiers d'entête suivies des déclarations de constantes ou de macro. Ensuite, nous développons les méthodes nécessaires au traitement.

6.1.2 Les fichiers d'entête

Les fichiers d'entête (d'extention ".h") contiennent en premier lieu, pour éviter les double-inclusions :

```
#ifndef EXAMPLE_H
#define EXAMPLE_H
```

La dernière ligne du fichier est donc :

```
#endif /* EXAMPLE_H */
```

Nous pouvons maintenant inclure d'autres fichiers d'entête et définir les variables globales. Nous trouvons dans ces fichiers, les prototypes des fonctions développées dans les fichiers sources du même nom.

6.2 Variables

Le nom des variables commence par "bot" suivi du nom parlant en anglais commencant par une majuscule. Les noms des structures sont construits de la même façon. Les variables sont caractérisées par leur nom mais aussi par leur type qui peut être un type énuméré :

```
enum botVar {val1, val2, val3 };
```

Une structure est définie de la façon suivante :

```
typedef struct botExemple {
char* botName;
int botId;
} botExemple;
```

6.2.1 Les variables globales

Elles sont déclarées dans le fichier d'entête et précédées du mot clé "external". Au début du fichier source, on définit ces variables afin de pouvoir les utiliser par la suite. Un soin particulier doit être apporté à l'initialisation de ces variables car elles sont utilisées par plusieurs fonctions.

6.2.2 Les constantes

Les constantes sont déclarées au début du fichier de la façon suivante :

```
#define CONSTANTE 20
```

Le nom de la constante doit suffire à expliciter sa fonction, il n'est donc pas nécessaire d'y ajouter un commentaire.

6.3 Commentaires

Un soin particulier est apporté à la rédaction des commentaires de façon à pouvoir comprendre et débuger notre code facilement. Nous devons distinguer différents styles de commentaires :

- les commentaires précédant le développement d'une fonction ou d'un fichier. Ils explicitent leur rôle et donnent éventuellement les valeurs attendues en paramètre. De 3 à 10 lignes, ces commentaires sont encadrés par les caractères "/*" une ligne au dessus et "*/" à la ligne du dessous. Nous trouvons le caractère "*" au début de chaque ligne de commentaire ;
- les commentaires à l'intérieur des fonctions servent à expliquer des passages du code en donnant la raison d'un test ou d'une affectation et en détaillant le contenu, à priori, des variables. Leur taille ne dépace pas, dans la majeur partie des cas, celle d'une ligne et ils sont encadrés par les caractères "/*" et "*/".

Les commentaires sont rédigés en français de façon à faciliter notre lecture du code et à se comprendre plus aisément.

Chapitre 7

Description des tests

7.1 Tests relatifs au module de calcul des statistiques

C'est cette partie du programme qui nous permet de démontrer l'efficacité des bots. Il est donc indispensable de s'assurer de sa fiabilité.

Pour contrôler que la fonction de récupération des données utiles aux statistiques fonctionne correctement, nous génèrons des donjons personnalisés. De cette façon, nous connaissons à l'avance toutes les informations que l'on souhaite retrouver. Pour vérifier que les données reçues sont bien celles attendues, il suffit de faire un affichage sur la sortie standard du jeu.

En ce qui concerne les calculs, nous avons commencé par envisager des cas relativement simples avec de petits niveaux de façon à pouvoir effectuer les mêmes traitements manuellement. Une fois cette étape validée, nous évaluons la capacité à détecter les erreurs de type : la division par 0 ou entrer un décimal négatif quand on attend un entier positif. Nous testons également les calculs avec des nombres trop grands pour déterminer les limites de notre implémentation.

7.2 Tests relatifs aux déplacements

Afin de contrôler que notre bot se déplace correctement, nous créons, pour chaque test (ou presque) des donjons particuliers répondant exactement à ce que nous cherchons à voir. Dans un premier temps, tant que le bot n'est pas capable de se défendre, nous supprimons les monstres.

Pour commencer, il nous faut vérifier que l'interface et le noyau communiquent correctement. Pour cela, nous voulons observer que le bot est capable de reconnaître son environement. En effet, il doit être capable de donner sa position et de définir une position à atteindre.

Il faut à présent pouvoir changer de pièce et ouvrir une porte. Nous commençons par mettre le bot face à une porte afin de vérifier qu'il l'ouvre et la traverse correctement. Nous testons ensuite s'il est capable de chercher une porte et de l'emprunter. De la même façon, nous contrôlons le passage du bot par les escaliers et les trappes.

Il peut être pratique de savoir se déplacer d'un point à un autre en prenant le chemin le plus court. Par exemple, si nous voulons rejoindre une pièce préalablement explorée, le bot doit calculer le chemin à suivre pour traverser un couloir. Cependant, cela implique de mémoriser les cases déjà visitées et nous devons également prendre en compte le temps de calcul de la trajectoire.

Pour commencer, nous calculons le chemin le plus court entre deux points d'une même pièce, puis d'une pièce à une autre. Le bot doit être capable de revenir en arrière afin de franchir un obstable.

Pour parcourir l'intégralité d'un niveau, nous devons considérer le fait que certaines portes sont cachées ; il est donc indispensable que le bot les découvre. Pour cela, nous allons créer un niveau dans lequel toutes les portes sont cachées. De cette façon, le bot n'a aucune autre solution et est dans l'obligation de les chercher pour avancer.

7.3 Tests de portabilité

Tout au long du projet, nous testons que le programme compile et fonctionne de la même manière sous Windows et sous Linux. Nous lui donnons les mêmes données, et il doit fournir des résultats similaires à tous les tests.

Chapitre 8

Plannings

8.0.1 Planning prévisionnel

Tâches / Semaines	6	7	8	9	10	11	12	13
1) Lecture appronfondie du code	X	X	X	X				
2) Mise en place du module de calcul des statistiques			X	X	X	X	X	
3) Implémentation du traceur		1	4	1				
4) Création du fichier de configuration			1					
5) Mise en place des structures dans le module world			1					
6) Implémentation de l'intelligence du bot				2	1	1	2	
7) Implémentation du module move				1	1	1		
8) Tests et améliorations du module de déplacement				1	1	1		
9) Implémentation du module eat				1	1			
10) Tests de la prise en compte de la faim				1	1			
11) Implémentation du module inventory					1	1	1	
12) Tests de gestion de l'inventaire					1	1	1	
13) Implémentation du module fight						1	1	
14) Tests et amélioration des algorithmes d'attaque						1	1	
15) Finalisation du mémoire								4

8.0.2 Planning réel

Tâches / Semaines	6	7	8	9	10	11	12	13
Lecture approfondie du code	X	X	X	X				
Mise en place du module de calcul des statistiques					1	1	1	
Implémentation du traceur			1	1				
Création du fichier de configuration				1				
Mise en place des structures dans le module world				2				
Implémentation de l'intelligence du bot				2	2	2	2	
Implémentation du module move				2	2	2		
Tests et améliorations du module de déplacement				2	3	3	3	
Rédaction du mémoire	X	X	X	X	X	X	X	X

8.1 Lecture approfondie de code

Cette étape est fondamentale pour savoir exactement à quels endroits nous allons pouvoir insérer notre code et pour connaître de façon plus précise le fonctionnement du jeu. Il est indispensable de comprendre ce qui est géré par le noyau du jeu et ce qui est géré par l'interface. Cette frontière doit être clairement établie pour que notre bot n'ait pas accès à certaines informations qu'un joueur classique n'aurait pas. On pourra également étudier le fonctionnement des monstres et s'inspirer de leur intelligence artificielle pour notre bot.

8.2 Mise en place du module de calcul des statistiques

Le module de calcul des statistiques nous permettra de valider les capacités de notre bot. Nous allons le mettre en place dès le début du projet puis nous l'améliorerons au fur et à mesure des besoins. En

effet, c'est grace à ce module que nous allons pouvoir valider et améliorer un bot. Nous le mettrons à jour chaque fois qu'un nouveau module sera créé.

En réalité ce module a été mis en place lorsque nous étions capable de générer des niveaux personnalisés. La première difficulté rencontrée a été de de récupérer les données issues du noyau puisque nous ne savions pas exactement comment elles étaient stockées. Il faut ensuite être en mesure de les utiliser efficacement. Les informations relatives au jeu du bot doivent être actualiser tout au long de la partie.

8.3 Implémentation du traceur

Nous devons être capable de tracer un bot afin de connaître les actions qu'il aura effectué au cours d'une partie, et ainsi voir si les choix qu'il a faits sont bons ou non. De cette façon, nous pourrons améliorer nos algorithmes et élaborer de nouvelles tactiques de jeu. Ce module permet l'exécution du bot puisque c'est lui qui transmet les commandes au noyau de NetHack, il est donc indispensable de commencer par là. De plus, c'est dans ce module que nous tirerons les informations du fichier de configuration nécessaires ua lancement du jeu.

8.4 Création du fichier de configuration

Le fichier de configuration va nous permettre de lancer un bot avec des objectifs prioritaires particuliers comme par exemple : manger, explorer, se battre, ... Ce fichier définissant les traits comportementaux du bot, il est nécessaire qu'il soit utilisable avant de commencer le bot.

La seule priorité implémentée étant l'exploration, le fichier de configuration n'a pas de raison d'être.

8.5 Mise en place des structures dans le module world

Le bot, comme un joueur, a accès à certain nombre d'informations telles que ses points de vie, son niveau, ... Nous devons donc stocker ces données de façon à ceux qu'elles puissent être facilement lues par le bot. Nous les enregistrerons dans des structures adaptées en nous inspirant de la manière dont c'est actuellement fait.

Ce module sert essentiellement au stockage de la carte mais il suffit de le développer au fur et à mesure des besoins du bot.

8.6 Implémentation de l'intelligence du bot

Nous devons maintenant implémenter le fait que le bot a plusieurs choix à sa disposition. En effet, il pourra explorer un donjon mais devra aussi se nourrir lorsque cela devient nécessaire. Jusqu'à présent, les modules étaient séparés donc le bot n'avait qu'un objectif. Ici nous allons essayer de permettre au bot de gérer sa faim, la recherche de nourriture et l'exploration du donjon. C'est dans ce module que nous développerons la capacité du bot à prendre des décisions.

Les seules décisions que le bot a à prendre sont en réalité de choisir la case qu'il veut explorer en priorité. C'est dans ce module que le bot "regarde" les élements qui l'entourent afin de se déplacer vers le plus intéressant.

8.7 Implémentation du module move

Nous devons créer un module de déplacement, qui nous permettra de créer un bot sachant se déplacer dans un donjon. Ceci comprend se déplacer dans les huit directions, contourner les obstacles, ouvrir les portes, ...

Ce module a été relativement difficile à mettre en place puisque c'est lui qui se charge de renvoyer la liste de commandes nécessaires pour aller vers la case pointée par l'intelligence artificielle. Il faut donc prendre en compte la position de l'animal de compagnie mais aussi ne pas s'interrompre si on entend un bruit ou réagir au cas où une porte serait bloquée.

8.8 Tests et améliorations du module de déplacement

Des tests seront réalisés pour savoir si notre bot se déplace astucieusement et s'il ne reste pas bloqué dans certaines situations. Suivant les résutats de ces tests, nous pourrons améliorer nos algorithmes de déplacements. A l'issue de cette étape, nous rendrons au client, M. Le Borgne, une première version du programme puisque les fonctionnalités basiques seront opérationnelles.

Cette étape a pris plus de temps que prévu puisque nous devons créer des niveaux personnalisés afin de tester tous les cas de figure. Nous avons donc résolu des problèmes liés au choix des cases à visiter mais aussi des difficultés dans le déplacement.

8.9 Implémentation du module eat

A ce stade, notre bot devra gérer le fait qu'il a faim. Si de la nourriture est disponible dans son inventaire, il devra être capable de se nourrir. D'un autre côté, si les réserves de nourriture deviennent cirtiques, il devra chercher de la nourriture et la ramasser pour en avoir toujours en quantité suffisante sur lui.

8.10 Tests de la prise en compte de la faim

Les tests réalisés nous permettrons de déterminer d'une part, si le bot sait se nourrir correctement mais également s'il sait gérer ses priorités. En effet, à ce stade de développement du bot, nous pourons contrôler qu'il prent bien les bonnes décisions : qu'il ne e laisse pas mourir de faim si sa priorité est d'explorer le plus de terrain possible.

8.11 Implémentation du module inventory

Nous devons faire en sorte que notre bot prenne en compte les objets présents dans son inventaire et les utilise au besoin. De plus, il faut qu'il soit capable de jeter des objets dont il n'a plus besoin ou lorsqu'il n'a plus de place. De la même façon, il doit être capable de s'équiper de la meilleure arme à sa disposition.

8.12 Tests de gestion de l'inventaire

Ces tests vont nous assurer que le bot ne reste pas bloqué par exemple s'il est trop lourd ou que l'arme en main est bien la meilleure car cela va être déterminant dans capacités d'attaque du bot.

8.13 Implémentation du module fight

Il faudra donner au bot la possibilité de combattre. En effet, jusque là nos tests étaient effectués sans monstre et donc le bot n'était pas tué. A partir de maintenant, il aura la possibilité de se battre ou de fuir si cela est préférable.

8.14 Tests et améliorations des algorithmes d'attaque

Nous testerons si le bot a combattu dans de bonnes conditions, à savoir est-ce qu'il avait assez de points de vie, est-ce qu'il avait une chance de survivre ou non, ... De plus, nous ferons des statistiques pour savoir le nombre de décès par rapport au nombre de combats. De cette façon, nous pourrons améliorer nos algorithmes d'attaque et rendre notre bot plus performant.

8.15 Finalisation du mémoire

Nous nous réservons une semaine pour finaliser le rapport.

Le rapport a été développé et amélioré tout au long du projet. De cette façon, nous pensons avoir expliquer plus clairement le cheminement de notre réflexion sans omettre d'informations importantes.

Chapitre 9

Intégration à NetHack

L'une des étapes cruciales du projet a été l'intégration du bot à NetHack. En effet, celui ci devait s'intégrer en apportant un minimum de modifications au noyau, et devait rester portable. La solution la plus simple a été d'intégrer le bot en tant qu'interface graphique.

En suivant les conseils de la documentation fournie avec NetHack [Tea], nous avons implémenté le bot en tant qu'interface graphique. Il a fallu créer une structure window_procs qui contenait des équivalents pour toutes les fonctions d'interface graphique, mais utilisées par le bot.

Il a ensuite fallu définir toutes ces fonctions, puis rajouter le bot dans le *config.h*. Mais ce n'était que la première étape, pour pouvoir visualiser les résultats, il a fallu lier une interface graphique avec le bot.

Le moyen le plus simple et le plus efficace a été de réutiliser la même façon de faire que le noyau de nethack, en utilisant les window_procs déjà créés. Il a fallu remodifier le *config.h* pour pouvoir choisir quelle interface graphique le bot devrait lancer (le jeu considérant que le bot est l'interface graphique principale).

Chapitre 10

Les changements de niveaux

10.1 Naviguer entre les niveaux

Pour naviguer plus facilement à travers les niveaux du jeu, nous avons mis en place une variable globale *botCurrLev* de type *bot_level* qui pointe sur le niveau où se trouve le héro à tout moment. De cette façon, lorsque le héro change de niveau, nous mettons à jour cette variable. Nous avons aussi mis en place une variable globale *botCurrStair* de type *bot_stair* qui pointe sur le dernier escalier sur lequel le héro est passé, ce qui nous permet de mettre plus facilement à jour le niveau courant.

10.1.1 Mise à jour du dernier escalier

A chaque tour du jeu, nous avons une fonction qui vérifie si le bot se trouve sur un escalier. Pour faire cela, nous regardons le type de la case qui a pour coordonnées la position du bot. Si c'est un escalier, alors nous parcourons la liste d'escaliers du niveau courant et nous faisons pointer *botCurrStair* sur celui où est le bot.

Garder en mémoire le dernier escalier sur lequel le bot est passé nous permet de changer plus facilement le niveau courant lorsque le bot emprunte un escalier.

10.1.2 Mise à jour du niveau courant

Pour nous rendre compte que le héro a changé de niveau, nous comparons deux valeurs : le niveau indiqué par le jeu et le niveau indiqué par notre structure. Si ces deux valeurs sont différentes c'est que le bot a pris un escalier, et il faut alors mettre à jour le niveau courant.

Pour cela, nous avons une fonction qui fais plusieurs choses. Tout d'abord, elle regarde dans la structure pointée par *botCurrStair*. Si le pointeur vers le niveau où mène l'escalier est nul, alors c'est que le niveau dans lequel se trouve le bot n'a jamais été visité. Sinon, on y est déjà allé.

Le niveau n'a jamais été visité

Si le niveau n'a jamais été visité, alors la fonction crée un nouveau niveau de type *bot_level* et l'initialise. En plus de cela, la fonction ajoute le niveau niveau dans la structure pointée par *botCurrStair*, puisque le dernier escalier sur lequel est passé le bot mène vers celui où il se trouve maintenant. Pour terminer, la variable *botCurrLev* va pointer sur le nouveau niveau.

Le niveau a déjà été visité

Si le niveau a déjà été visité, c'est-à-dire si *botCurrStair-¿toLevel* n'est pas nul, alors on fais pointer *botCurrLev* dessus.

Changer de niveau à l'aide d'un escalier ou d'un trou

A l'aide d'un escalier Si le bot a changé de niveau en utilisant un escalier, alors cela veut dire que dans le nouveau niveau, le bot se retrouve directement sur un escalier. La fonction crée donc un nouvel

élément de type *bot_stair* en initialisant entre autre le sens de l'escalier comme étant l'inverse du dernier escalier parcouru. De plus, le nouvel escalier pointe sur le dernier niveau dans lequel le bot était. La variable *botCurrStair* pointe alors sur l'escalier qui vient d'être créé.

A l'aide d'un trou Si le bot est tombé dans un trou, il ne peut pas revenir dans le niveau précédent donc on ne crée pas d'escalier, la seule chose faite est que le trou mène vers le nouveau niveau.

Les trous sont donc considéré comme étant des escaliers à sens unique.

10.2 Les calculs des spécificités des niveaux

Pour chaque niveau, nous calculons le nombre de cases, de portes et de portes cachées que le bot découvre.

Pour ce faire, la variable *visited* de chaque case d'un niveau est initialisée à 0, et lorsque la case est découverte, elle passe à 1, pour que chaque case ne soit prise en compte qu'une seule fois.

10.2.1 Identification du type d'une case

De nombreuses informations sont envoyées par le noyau à l'interface, et toutes ces données sont envoyées sous forme d'entiers calculés appelés *glyph*. Il nous a donc fallu chercher leur signification pour pouvoir filtrer les informations qui nous interresaient. Nous comparons ensuite le *glyph* à des valeurs pour en connaitre le type.

10.2.2 Calcul du nombre de cases découvertes

Pour savoir si c'est une case (et non un bruit entendu par le héro ou encore l'effet d'un sort), on vérifie que le *glyph* est celui d'une case. Nous le testons donc pour savoir s'il est compris entre le *glyph* correspondants à un mur vertical et une trappe. En effet, ces deux valeurs sont les extrémités des *glyph* des éléments d'un niveau.

10.2.3 Calcul du nombre de portes découvertes

Pour savoir si le *glyph* passé correspond à une porte, nous le testons avec les *glyph* correspondants à une porte verticale fermée et celui d'une porte horizontale fermée. En effet, nous ne comptons que les portes fermées puisque les portes ouvertes n'ont aucun intérêt ; elles n'empêcheront pas le bot d'avancer.

10.2.4 Calcul du nombre de portes cachées découvertes

Pour les portes cachées, notre fonction est appelée à chaque tour, avant que le type de la case ne soit stocké dans notre structure. De cette façon, notre fonction compare l'ancien type de la case dans notre structure avec le nouveau : si le type précédent était un mur et que c'est devenu une porte alors on a trouvé une porte cachée.

Chapitre 11

L'exploration

L'élément clef pour un joueur et donc pour le bot, est de bien savoir explorer un donjon. Ainsi il peut espérer trouver un maximum d'objets lui permettant de mener à bien sa quête, un maximum de monstres l'aidant à grimper dans les niveaux d'expérience mais également les nombreux passages secrets que le donjon rescelle. Nous avons donc choisis de séparer, pendant le développement, les algorithmes de déplacement du bot de ceux lui dictant où il devait se rendre.

11.1 Les différentes phases de l'exploration

Le bot procède par étape pour savoir quelles sont les endroits les plus dignes d'intérêts à un moment donné de la partie.

- Tout d'abord il va attribuer une valeur d'importance : un poids, sur toutes les nouvelles cases qu'il découvre. Ceci lui permet dans un premier temps de se souvenir de façon simple qu'à tel endroit, il devra se rendre pour rendre son exploration efficace.
- Ensuite, si la partie décisionnelle décide que l'exploration est une priorité du moment, le bot va chercher un lieu où aller pour améliorer sa connaissance du donjon.
- Enfin, différents algorithme travail sur les données emmagasinées par le bot afin de renvoyer au bot une destination qui lui permettra d'avancer et de découvrir de nouvelles choses.

11.1.1 La mise en place des poids

A chaque nouvelle case découverte, le bot doit définir la pertinence de chacune. Afin d'affecter un poids pertinent à chacune des cases, nous avons mis en place un ensemble de règles fondées sur une réflexion simple : quelles sont les cases importantes à visiter pour explorer un niveau ? De cette réflexion est sorti plusieurs points important :

- Tout d'abord, les cases les plus importantes sont les portes ouvertes ou les passages. En effet il est facile de comprendre qu'à n'importe quel moment du jeu, il est préférable de prendre une porte ouverte qui mène soit dans des salles si on est dans un couloir, soit dans un couloir à priori inexploré (sinon on aurait déjà emprunté la porte) si on est dans une salle.
- Ensuite viennent les cases secondaires, elles représentent le vaste réseau de couloir du donjon et les portes fermées. Les couloirs sont moins importants que les passages car même si ils mènent potentiellement à d'autres salles, qui sont des zones importantes pour le bot car recèlent potentiellement de nombreux trésors, en eux-mêmes ils sont plutôt vides. Les portes fermées malgré leur grand intérêts sont dans cette catégorie de case secondaire car elle demande du temps pour être ouverte, ce qui met en danger le bot. Plus on reste longtemps au même endroit, plus il y a de chance qu'un monstre nous tombe dessus, ou qu'on ait faim, ou quoi que se soit d'autres. Ces cases au même titre que les couloirs ont donc leur place dans les cases dites secondaires, c'est-à-dire qui présente un intérêt mais réduit.
- Afin de signaler au bot ou est-ce qu'il a déjà marché, qu'est-ce qu'il a déjà visité, il a fallut mettre aussi des cases de poids standard. Ce niveau d'intérêt (qui est l'intérêt nul) est atteins par toutes les cases ayant déjà étaient visité. Les cases d'une salle sont dès le départ à ce niveau d'intérêt, car

à partir du moment où le bot rentre dans la salle l'ensemble des cases se révèlent (contrairement au couloir). Il n'est donc pas nécessaire que le bot parcours physiquement toutes les cases d'une salle pour se rendre compte qu'il a tout exploré dans la salle.
- Et enfin, pour tout ce qui est noir à l'écran, pour les murs et tout autre élément infranchissable, une valeur barrière est affectée aux cases concernées. Cette valeur permet tout simplement au bot d'ignorer l'existence même de la case.

Les cases apparaissant parfois avec des monstres ou des objets dessus, il a fallut rajouter dans notre structure un élément permettant de distinguer les éléments architecturaux, des éléments mobiles (comme les objets, les monstres etc.. . .). Ces éléments recouvrant une case de type « carte », nous avons du choisir de mettre une valeur standard afin que le bot ne considère pas que ces autres cases soient des barrières, mais qu'il ne se focalise pas dessus non plus.

11.1.2 La prise de décisions : le calcul du besoin

Pour que le bot décide d'explorer le donjon plutôt que faire une autre action a sa disposition, il doit calculer le besoin correspondant à l'exploration. Ce besoin est volontairement calculé de façon simpliste dans notre algorithme, car étant donné que nous n'avons pas un grand nombre de besoin à prendre en compte, inutile de faire une fonction complexe.

Le bot calcul cette valeur en regardant tout simplement s'il reste des cases à explorer sur la carte. Si c'est le cas, le besoin reste à dix, sinon il passe à zéro (zéro étant la valeur minimum des besoins et dix la valeur maximum). Ainsi le bot a l'obligation de tout explorer avant de passer à autre chose.

Pour étoffer cette fonction en cas de besoin plus nombreux exigeants plus de finesse pour l'évaluation du besoin, nous avons pensé à plusieurs pistes. Par exemple, décrémenter ce besoin lorsque un escalier est trouvé, ou encore trouver le nombre de case en moyenne généré par niveau et décrémenter le besoin par exemple par palier de 10% de cases explorées. Etant donné que le calcul du besoin est complètement indépendant, ce genre de modification est facile à insérer dans notre code.

11.1.3 Les algorithmes d'exploration

L'algorithme principal

Maintenant que nous avons nos poids, que le bot a pris la décision d'explorer plus de terrain, il faut lui permettre de choisir une case parmi toutes celles que nous lui proposons selon le critère de « poids ».

Ayant une multitude de cases avec des poids similaires, nous avons mis en place un système de recherche du maximum en tenant compte de plusieurs critères. En effet, là aussi nous avons du réfléchir à comment rendre l'exploration efficace connaissant les lieux clefs de la carte. Il faut que le bot en choisisse un, et de préférence un bon. Cette étape s'est faites en plusieurs améliorations successives d'un algorithme de base, qui consisté tout bêtement à parcourir la matrice de case en mémoire, et à en ressortir la première valeur observé comme maximale. Cette version marchait très bien mais présenté des aspects bien trop naïf pour qu'elle soit efficace en terme de tour de jeu. Analysons un cas de figure. (fig. 11.1)

Fig. 11.1 – 1er cas de figure

Ici le joueur aurait naturellement tendance à aller sur la case se situant à sa droite. Mais notre algorithme étant, ne raisonnant pas en termes de position du joueur mais plutôt en termes de position absolue des cases dans la matrice, le bot choisira la porte tout à gauche de l'écran. Une fois fait, étant donné que la seule case de plus haut niveau d'importance se situe à droite (le passage), le bot reviendra à ce passage.

Ce balayage de la carte n'est pas souhaitable, car chaque déplacement a un coup, que se soit en termes de nourriture ou de mise en danger du bot.

Il a fallut donc raffiner cet algorithme afin de nous recentrer sur une recherche de la case la plus importante, non pas bêtement en parcourant la matrice du début à la fin, mais en partant de la position du bot. Le schéma que nous avons appliqué est celui de l'escargot, c'est-à-dire qu'on part de la position du bot dans la matrice, et on cherche la case de plus grande importance en faisant des carrés (car c'est une grille) de diamètre de plus en plus grand. (fig. 11.2)

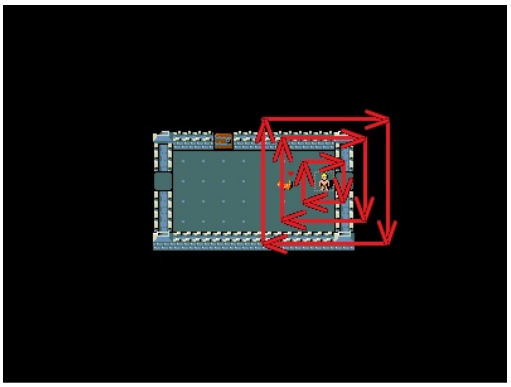

Fig. 11.2 – 2eme cas de figure - 1

Cet algorithme gomme une partie du problème précédent, car nous voyons bien ici que le passage à proximité du bot est repéré en premier, ce qui fait qu'il le prend pour objectif. Mais un autre problème persiste. En effet une fois le passage de droite exploré, même si la recherche s'effectue à partir du bot, il choisira le passage de gauche car le poids du passage est plus fort que le poids du couloir nouvellement trouvée. On se retrouvera dans cette situation. (fig. 11.3)

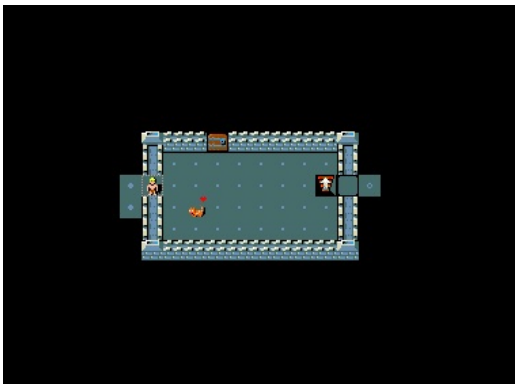

FIG. 11.3 – 2eme cas de figure - 2

Encore une fois, on s'aperçois qu'en termes de tour de jeu cela n'est pas intéressant, il vaut mieux que le bot explore ce qui est proche de lui.

Cette fois-ci au lieu de reprendre tout l'algorithme de parcours de la matrice qui nous paraissait bon pour notre bot, nous avons décider d'ajouter une contrainte au choix du maximum qui dit que si la case se trouve dans un rayon d'action définis dont le bot est le centre, alors la ou les cases concernées on un poids supérieur à la normal. De ce fait, une case couloir qui a un niveau moyennement élevé passe à très élevé comme le son les passages par défaut. En faisant de la sorte, le bot préfèrera un couloir à proximité plutôt qu'une porte éloignée.

Le rayon d'action étant volontairement limité à un, car sinon le bot hésitait trop souvent entre deux chemins. Et donc il arrivait que le bot ne puisse atteindre certaines cases. Par exemple, lorqu'il parcourt un niveau, le bot peut appercevoir des cases qui l'intéressent mais qui ne sont pas connexe avec le reste du niveau. Du coup l'algorithme A* ne peut trouver de chemins y menant. Il a donc fallut ajouter une liste noir temporaire, qui a chaque tour identifie les cases hors de portées du bot, afin qu'il puisse chercher des objectifs accessibles.

L'algorithme complémentaire

A présent notre bot obtient des objectifs pertinents sur la carte, mais il n'empêche qu'à certain moment, étant donné que le parcours des cases de type « salle » n'est pas dans ses priorités, il se peut que des salles soient explorées de manière incomplète. Ceci entraine bien évidemment qu'un certain nombre de passage reste méconnu du bot. (fig. 11.4)

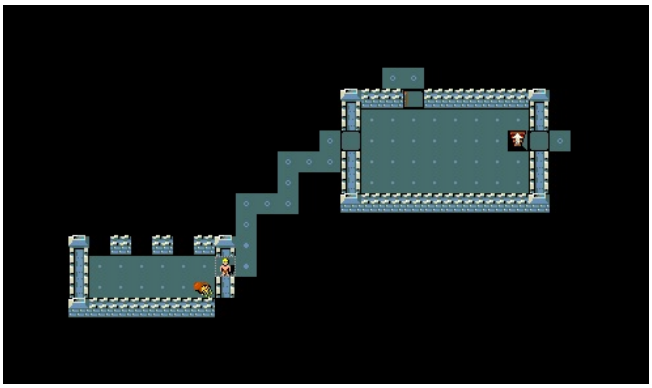

FIG. 11.4 – 3eme cas de figure - 1

Ici le bot en arrive à la conclusion que tout est découvert, les couloirs en haut et à droite étant exploré à fond. Alors que nous voyons bien que ça n'est pas le cas. Un algorithme est donc dédié à cette tâche de détecter les salles « incomplètes », c'est-à-dire auxquelles il manque des portions de murs et/ou des portes.

Cet algorithme comme le précédent à été fait d'amélioration successive. Tout d'abord dans une première version, il prospectait en faisant une forme d'escargot autour du bot, cherchant les murs (car toutes les salles sont entourées de mur), et comptant le nombre de murs voisins de chaque mur (dans murs voisins il est aussi compris les portes fermées, passages et portes ouvertes qui sont des éléments de murs). Si ce nombre est strictement inférieur à trois, cela veut dire qu'il manque un mur. L'algorithme affecte alors un poids maximal à une case où le bot peut marcher (une case de type « salle »en l'occurrence) à proximité de ce mur. Le bot voit alors qu'il y a une case d'intérêt, et y va, ce qui permet de découvrir le reste de la salle ou de la portion cachée comme le montre cet illustré. (fig. 11.5)

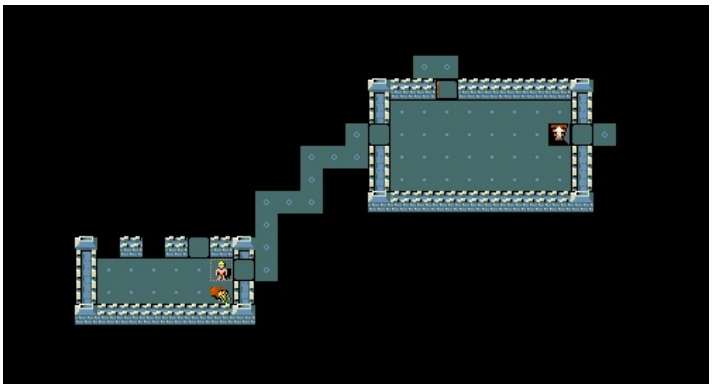

Fig. 11.5 – 3eme cas de figure - 2

Le bot a découvert une porte, l'algorithme principal reprend la main et le bot peut continuer l'exploration du donjon.

Par la suite, la principale amélioration a été de l'intégrer à la recherche de maximum. En fait en plus de ce travail, l'algorithme test en permanence s'il y a des salles incomplètes. Il réutilise donc le parcours en escargot déjà implémenté pour le maximum, il regarde juste si la case en question est un mur, si oui, il compte ses voisins. La réelle nouveauté vient du fait que maintenant nous ne mettons plus un poids de valeur maximale sur une case voisine de cette brèche, mais nous renvoyons, en courcircuitant le système des maximums, la position de la case à explorer. De la sorte, dès que le bot est face à une salle « incomplète », il avance afin de finir de l'explorer desuite. Par la même occasion ça permet au bot d'explorer d'un coup les salles sans lumières, où seuls les murs apparaissent au fil de l'exploration.

11.2 Changement de niveau dans le donjon

Une fois le niveau complètement exploré, il va de soit que le bot puisse changer de niveau et aller explorer le suivant. Il a donc fallut gérer le fait que chaque niveau possède au moins un escalier vers un nouveau niveau.

11.2.1 La prise de décisions : le calcul du besoin

Ici le besoin se calcul simplement. Une fois que le niveau est entièrement exploré, le besoin passe à 10 pour peu qu'un escalier est été découvert.

11.2.2 L'algortihme déclenchant la descente d'un escalier

Une fois que le bot ait choisis de déclencher ce mode, le bot va chercher un escalier, et gràce à l'algorithme A*, il va définir comment y aller en ajoutant les commandes dans la liste de commande. Ensuite, une fois à destination, il va envoyé la commande pour descendre. La simplicité de cet algorithme est du au fait que toute la gestion complexe d'un changement de niveau dans la mémoire de notre bot est prise en charge par d'autres algorithmes en aval de celui-ci.

11.3 L'exploration fine : la recherche de passage secret

Tout donjon dans NetHack renfermant des passages secrets, et en quantité, il nous a parut indispensable d'implémenter un algorithme permettant de les détecter efficacement. Cet aspect de l'exploration est volontairement expliquer en dehors de l'explication de l'exploration car il fait l'objet d'un besoin propre : le besoin de trouver des portes cachés, qui n'est pas forcément indispensable à l'exploration.

11.3.1 La prise de décisions : le calcul du besoin

Tout d'abord, intéressons-nous au déclenchement de cette phase du bot. Le besoin, pour l'instant, se calcul très simplement. Une fois que le bot a exploré tout ce qu'il pouvait explorer et qu'il n'a pas encore trouvé d'escalier descendant, le besoin d'exploration passe à zéro brutalement ainsi que le besoin de descendre, ce qui entraine un passage à dix du besoin de chercher des passages secrets. Cette version suffit amplement pour notre bot actuel, mais nous avons pensé à des améliorations pour rendre cette valeur plus fine. Par exemple, la majeur partie du coin en bas à gauche de la carte est non exploré et aucun couloir n'y mène ; notre algorithme pourrait prendre en compte ce fait, et augmenter le besoin de chercher de façon progressive lorsqu'on se rapproche de cette zone, de sorte qu'il se mette à chercher quand il est au plus près de cette zone noir.

11.3.2 Déroulement de l'algorithme de recherche

Cet algorithme est assez similaire à celui de recherche des salles incomplètes. A la différence près qu'ici on ne compte plus les voisins des murs, mais on affecte aux cases de type « salle » voisines des murs, des niveaux de priorités.

Tout d'abord nous avons ajouté une valeur sur chaque case qui permet de compter le nombre de fois qu'une case a été explorée, cette valeur permet de savoir si un secteur de la carte a été plus fouillé qu'un autre, et dans ce cas réagir en changeant le secteur des fouilles. Enfin nous avons dans chaque niveau un compteur qui indique le niveau de recherche en vigueur dans le niveau. Cet indicateur permet de savoir si les cases du niveau ont été fouillées « suffisamment » au vu des besoins du bot.

Maintenant que notre structure est en place, analysons un cas de figure pour voir comment se déroule notre algorithme. (fig. 11.6)

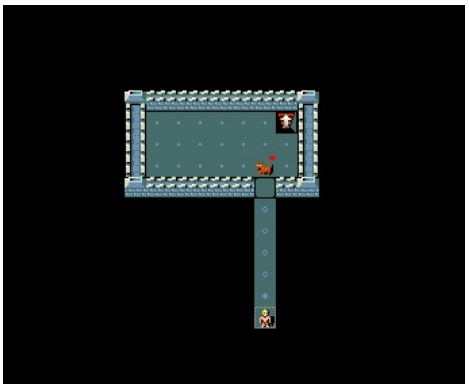

Fig. 11.6 – 4eme cas de figure - 1

Ici nous voyons que le bot a exploré tout ce qu'il pouvait explorer et que visiblement, aucun escalier n'a été trouvé. Le besoin d'exploration passe donc brutalement à zéro ainsi que celui de descendre, ce qui entraine que le besoin de chercher des portes cachées passe à dix. La recherche de passage secret comme celle du maximum s'effectue en faisant un balayage concentrique de plus en plus grand autour du bot. L'algorithme cherche des cases qui n'auraient pas été fouillé suffisamment au vu du niveau de recherche en vigueur dans le niveau. Ici nous sommes à une premières recherche, ce niveau est donc deux (les paliers des niveaux sont paramétrables). Le bot cherche donc une case qui aurait été fouillé moins de deux fois, il va même prendre la case qui a été fouillé le moins de fois possible. Cela lui permet une recherche homogène dans toute la carte. Le bot effectue un l'algorithme A*, et ajoute autant de fois que nécessaire la commande chercher dans la liste de commande. (fig. 11.7)

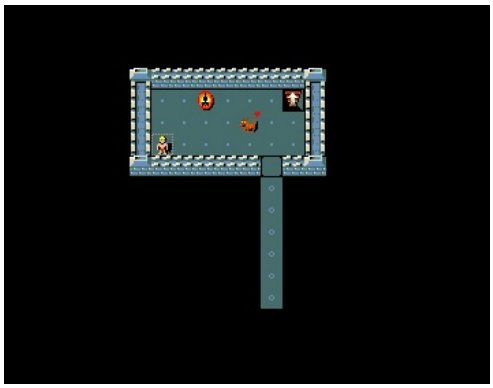

Fig. 11.7 – 4eme cas de figure - 2

Le bot une fois arriver sur la case définis auparavant, va fouiller autant de fois que le nécessitequ'il lui ait demandé dans la liste de commandes. Par exemple ici, c'est la première fois qu'on fouille cette case, et le niveau de recherche est de deux car on est au premier palier, la case va donc être fouillé deux fois de suite afin d'atteindre le premier palier. Ensuite, l'algorithme cherche une autre case à fouiller ayant été moins fouillé que le palier l'exige.

Le bot va continuer comme ça jusqu'à ce qu'il n'y ai plus de case à fouiller. Une fois arriver là, et s'il n'a toujours rien trouvé, le bot va augmenter l'intensité de ses recherches, comme ce que ferait n'importe quel joueur. Pour ça, il va augmenter le niveau de recherche du niveau en multipliant le palier de base par deux (car c'est la deuxième recherche qu'il va effectuer). Ensuite, l'algorithme cherche à nouveau la

case la moins fouillée, la fouille et ainsi desuite. La fouille s'arrêtera lorsque le bot aura trouvé un passage secret menant à un escalier.

Pour affiner cette recherche, nous avons également implémenté un algorithme permettant de trouver les portes cachées en bout de couloir dans certain cas de figures. En fait nous regardons si une case couloir a moins de 3 voisins (en se comptant elle-même), si c'est le cas, cela veut dire qu'on est sur un cul de sac, ce qui correspond potentiellement à une porte cachée. Cet algortihme est limité mais résoud pas mal de cas de portes cachées dans les couloirs.

Maintenant, voyons ce qu'il se passe lorsque le bot trouve quelque chose. (fig. 11.8)

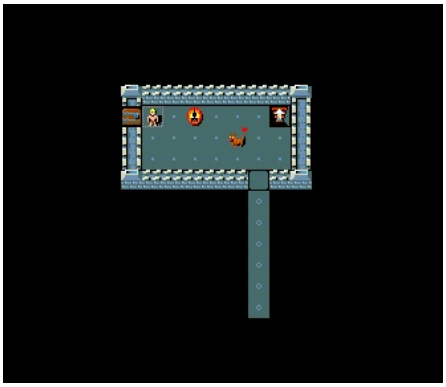

FIG. 11.8 – 4eme cas de figure - 3

Ici, après plusieurs tours de la salle, et plusieurs paliers de recherche franchis, le bot trouve enfin une porte. Cette découverte a pour conséquence d'annuler le besoin de chercher des passages secrets, vu que nous en avons trouvé un, et par la même occasion d'enlever des objectifs du bots les cases à fouiller. Il garde en mémoire uniquement le nombre de fouilles qu'il a effectué dans chaque case. Maintenant le bot passe en mode exploration classique vu qu'un nouveau passage s'offre à lui, jusqu'au moment où il devra à nouveau basculer en mode recherche et ainsi desuite.

11.4 Mouvements aléatoires

Au cours de nos tests, nous sommes aperçus que notre bot n'avancait plus, soit parce qu'il n'avait plus de commande à exécuter, soit parce qu'il essayait d'exécuter une commande que le jeu n'accepte pas. Par exemple, on ne peut pas franchir les portes en diagonales mais nous n'avons pas réussi à résoudre complètement ce problème.

De plus, il arrivait que le bot veuille aller vers une porte fermée qu'il voyait, mais que le chemin jusqu'à la porte n'était pas complètement découvert. Dans ce cas l'algorithme A* ne pouvait pas trouver de chemin, ce qui impliquait que le bot ne bougeait plus.

Nous avons donc mis en place un système de déplacements aléatoires.

11.4.1 Détection d'un arrêt du bot

Pour savoir si le bot est bloqué, nous utilisons les attributs du pointeur *botPosition*, à savoir *lastX*, *lastY* et *freezed*.

A chaque tour, nous vérifions que les coordonnées du bot au tour précédent sont différentes des actuelles ou non pour savoir s'il s'est déplacé.

Le bot n'a pas bougé

Si le bot n'a pas bougé, nous vérifions dans ce cas que ce n'est pas parce qu'il est en train de chercher ou d'essayer de casse une porte. En effet, dans ces deux cas le bot ne change pas de case.

S'il n'est pas dans une de ces deux situations, alors c'est qu'il est peut-être bloqué. Dans ce cas, nous ajoutons "+1" à l'attribut *freezed*. De cette façon, lorsque cette valeur atteint 10 (valeur définie par *BOT_FREEZE* alors nous déclenchons un mouvement aléatoire et nous remettons le compteur à 0.

Le bot a bougé

Dans ce cas, nous remettons juste la valeur *freezed* à 0 puisque si le bot était bloqué, il ne l'est plus.

11.4.2 Déclenchement d'un mouvement aléatoire

Les mouvements aléatoires que nous déclenchons sont les 8 mouvements possibles du bot. Lorsque notre fonction pour envoyer un mouvement aléatoire est appelée, elle fais un random sur 8 valeurs et renvoie un des 8 déplacements, à part si le bot est sur une porte. En effet, comme on ne peut pas franchir une porte en diagonale, si on est sur une porte sur un mur vertical alors un fais un random sur les directions LEFT et RIGHT ; sinon TOP et BOTTOM.

En plus de cela, la fonction appelle la fonction pour vider la liste de commandes puisque le mouvement aléatoire a fais que la liste de commandes n'est plus valide. En effet, le bot n'est plus sur la case qu'il était censé être pour effectuer les commandes donc elles deviennent inutilisables.

Chapitre 12

Les statistiques

Pour pouvoir faire des statistiques sur les performances de notre bot quant au nombre de cases découvertes, le nombre de portes ou de portes cachées, nous devons connaitre le nombre qu'il en découvre, ceci est fais au cours de la partie, mais aussi le nombre total qu'il y en a.

Notre module statistique se charge donc de récupérer le nombre total de ces trois éléments pour chaque niveau visité par le bot. De cette façon, nous pouvons facilement faire une étude à la fin du jeu sur ses performances.

12.1 Récupération des informations

Pour stocker efficacement les informations de chaque niveau et pouvoir les raccrocher au bon, nous avons créé une liste *botListStats* qui contient des éléments de type *bot_stats*.

12.1.1 Identification des données

Pour récupérer facilement les données de chaque niveau, le module statistique va regarder directement dans le noyau du jeu. Nous avons décidé de faire ainsi parce que cela est rapide, et comme le bot n'a pas accès au module statistique, il ne peut pas tricher, le module statistique peut donc allé chercher des informations dans le noyau sans risque.

Pour identifier les cases, les portes et les portes cachées de chaque niveau, nous parcourons l'ensemble des cases du niveau et nous regardons de quelle type elles sont.

Calcul du nombre de cases

Pour le nombre de cases, nous regardons si le type de la case est compris entre les valeurs *VWALL* et *CLOUD*. Ces valeurs appartiennent à NetHack et représentent tous les types de cases qu'un niveau peut contenir. Bien entendu, les portes sont comprises dedans puisque ce sont des éléments d'un niveau.

Calcul du nombre de portes

Pour les portes, nous regardons si le type est égual à *DOOR*, qui est aussi une valeur de NetHack et qui représente une porte. Comme nous ne comptons que les portes fermées ou bloquées (les portes ouvertes n'étant pas importantes), nous regardons aussi la valeur *doormask* de chaque case qui est une porte. Si ce "mask" est *D_CLOSED* ou *D_LOCKED*, alors c'est une porte fermée ou bloquée et nous la comptons donc.

Calcul du nombre de portes cachées

Pour compter le nombre de portes cachées, nous regardons simplement si le type de la case est ègual à la valeur de NetHack *SDOOR*.

12.1.2 Création de la liste

Pour créer les éléments de notre liste, nous utilisons l'entier *visited* de *bot_level*. Lors de la création du niveau, il est initialisé à 0, et lorsque le module statistique ajoute le niveau dans sa liste, *visited* passe à 1. De cette façon, un niveau n'est ajouté que s'il n'a jamais été visité auparavant.

Notre fonction est appelée lorsque le bot change de niveau, juste après l'appel de la fonction qui change le niveau courant. Ainsi, le nouveau niveau est créé et les nombres de casess et de portes est calculés sont ajoutés à la structure.

12.2 Calcul des statistiques

A la fin du jeu, la fonction *bot_doStats* est appelée. Cette fonction s'occupe de créer un fichier nommé *STATS* en mode "ajout" (le fichier est d'abord vidé). La liste du module statistiques est alors parcourue et pour chaque niveau, la fonction écrit :

- le nombre de cases du niveau, le nombre de cases découvertes par le bot ainsi que le pourcentage ;
- le nombre de portes du niveau, le nombre de portes découvertes par le bot ainsi que le pourcentage ;
- le nombre de portes cachées du niveau, le nombre de portes cachées découvertes par le bot ainsi que le pourcentage.

En plus de cela, lorsque tous les niveaux ont été parcourus, la fonction écrit le nombre de niveaux parcourus ainsi que les pourcentages de ces trois données, mais qui représentent cette fois les pourcentages par rapport à la totalité des niveaux parcourus.

Chapitre 13

Les comportements observés du bot

Le bot ayant de multiples facettes via les différents algorithmes qui le composent. Il est intéressant de voir lequel de ses traits de caractère est dominant, ou tout simplement de voir quel est et pourquoi l'algorithme le moins réussi. Nous avons fait ces séries de test avec le bot ayant la dernière version de notre implémentation, ainsi que le fichier de configuration du jeu ayant choisi par défaut la classe barbare et le ramassage automatique ne prenant que les petits objets genre trésor. De ce fait, notre bot peut se mouvoir de façon optimal.

13.1 L'aspect global

Commençons par jeter un œil sur la répartition des fins de jeux observées. (fig. 13.1)

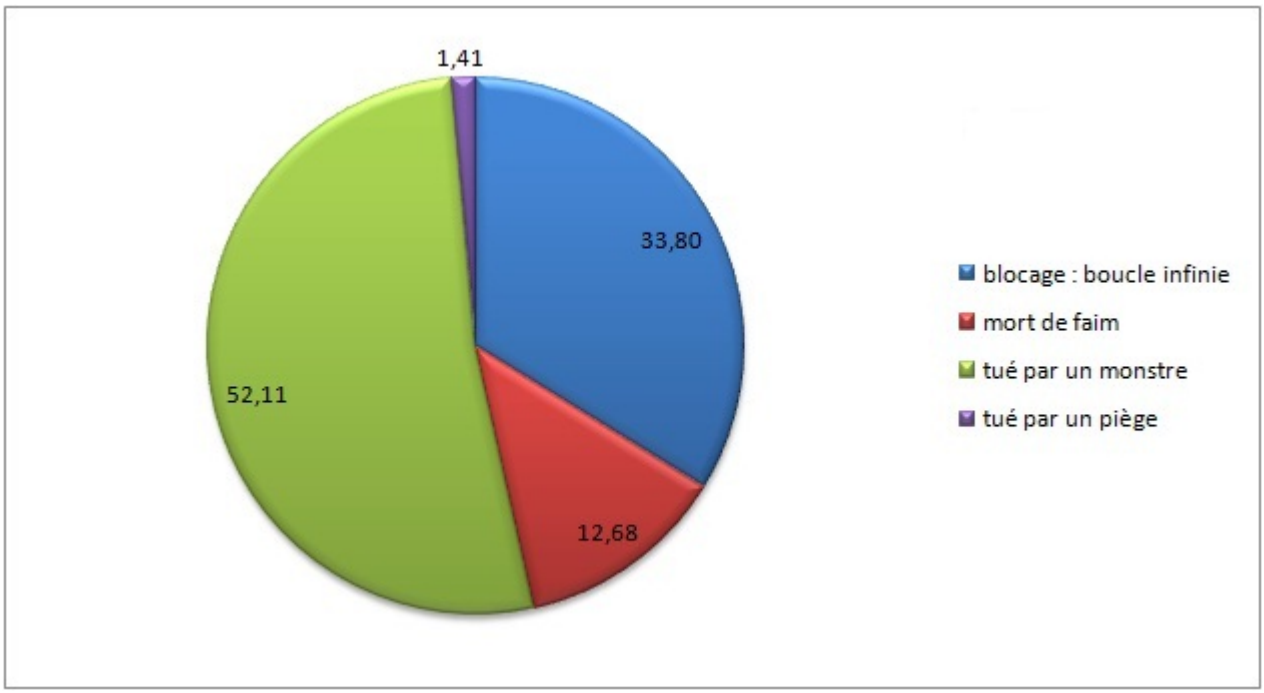

Fig. 13.1 – Répartition des différentes morts

Tout d'abord, et c'est le plus gênant, on se rend compte que près de de 34% des parties ne peuvent avoir d'issus. Ces parties correspondent à des hésitations du bot, ou un chemin inaccessible que le bot essaye de prendre, ou un bug qui rend la prise de décision impossible, ou cyclique. Tout de même, malgré cela,

on peut voir également que le bot ne plante jamais, et ne s'arrête jamais de jouer. Les autres indications que nous apporte ce fromage sont plus sur les priorités des améliorations à apporter. Ici nous voyons très nettement que l'absence d'algorithme de gestion du combat est extrêmement préjudiciable au bot. Dans plus de 50% des cas, ça lui est fatale. Ensuite vient la faim, qui est plus anecdotique car le barbare peut ne pas manger pendant beaucoup plus de tour que les autres classes. Ce qui a dirigé notre choix de classe pour que ces tests soient plus parlants. Ensuite, et c'est beaucoup plus rare, la mort suite à un piège.

13.2 Le détail

Maintenant que nous avons vu comment le bot effectuerse comporter sur un ensemble de parties, intéressons nous à chaque mort spécifique, afin de mieux comprendre pourquoi et comment il en arrive là.

13.2.1 Les techniques d'explorations

Le bot a à sa disposition plusieurs algorithmes pour parcourir un niveau. Ces différents algorithmes amènent dans le meilleur des cas à la mort du bot. C'est cette mort qui est intéressante car elle nous renseigne sur la façon dont est bâtit le donjon et donc sur comment a réagis le bot. Ce graphique nous montre le détail des performances du bot selon comment s'est terminé la partie. (fig. 13.2)

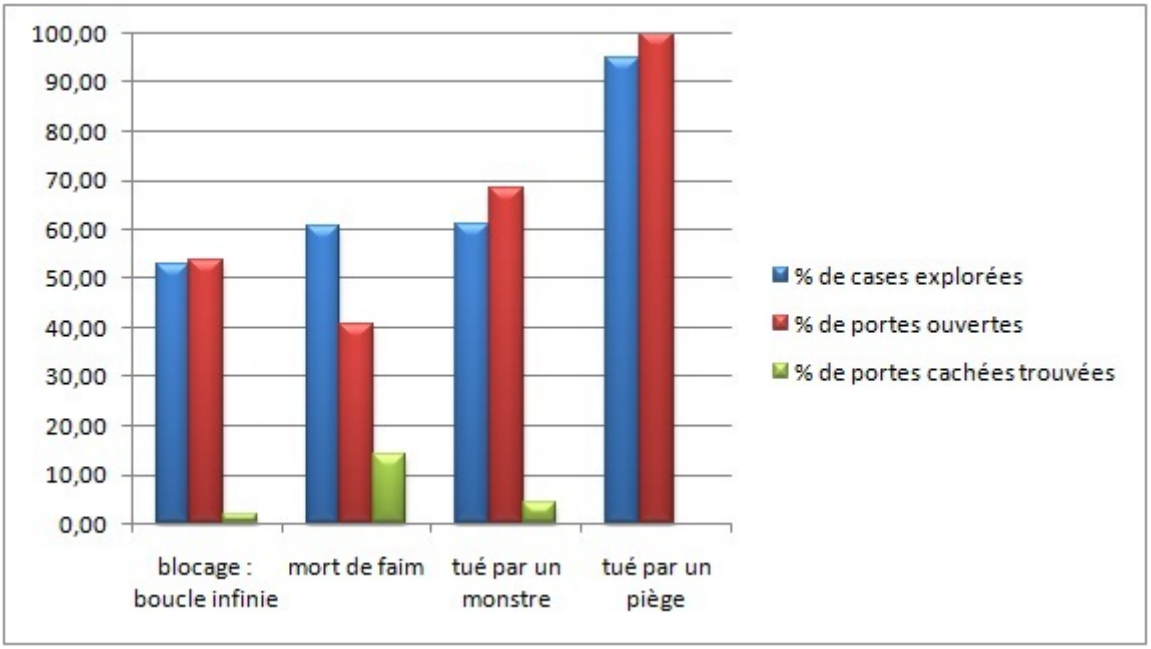

FIG. 13.2 – Performance pour chaque mort

Le blocage du bot représente un arrêt du bot à un état quelconque d'exploration du niveau, c'est pour ça que le pourcentage de cases explorées et le pourcentage de portes ouvertes sont proches de 50%. C'est une sorte de moyennes globales. En revanche lorsque le bot est mort de faim, cela veut dire qu'il a passé beaucoup de temps à chercher les portes cachées. Ce qui explique qu'on ait le taux de portes cachées trouvées le plus élevé. Mais ça veut dire aussi que les niveaux qu'il a parcourut ne contenait pas beaucoup de porte visible à porté, ce qui explique son mauvais score en ouverture de porte. Lorsque le bot est tué par un monstre, on voit clairement que les niveaux étaient idéalement configurés pour le bot. En effet il y avait peut de passages secrets et beaucoup de portes visibles. Il a donc pu facilement explorée beaucoup de terrain et de ce fait rencontré plus de monstres qu'à l'accoutumé. Les dernières barres montrent d'une certaine façon que lorsque le bot meurt dans un piège c'est que certainement qu'il a exploré beaucoup

de terrain, et pour une raison ou une autre, il passe à répétition sur le piège. Étant donné que nous ne gérons pas les pièges, il ne peut les voir et meurt.

Ce que nous pouvons tirer de cela, c'est que le bot aurait certainement à gagner à ne pas descendre trop vite lorsqu'il le peut, car on voit bien que même si ça lui permet de découvrir plus de terrain, il peut éventuellement passer à côté de salles secrètes intéressante. Ce graphique nous montre également la faiblesse de notre algorithme de recherche des portes cachées, car lorsqu'il n'a pas d'autres choix, le bot s'épuise complètement à chercher des passages secrets.

13.2.2 La recherche de portes cachées

Le bot étant capable de chercher des portes cachées, il est également intéressant de voir les résultats de nos algorithmes sur un grand nombre de parties. (fig. 13.3)

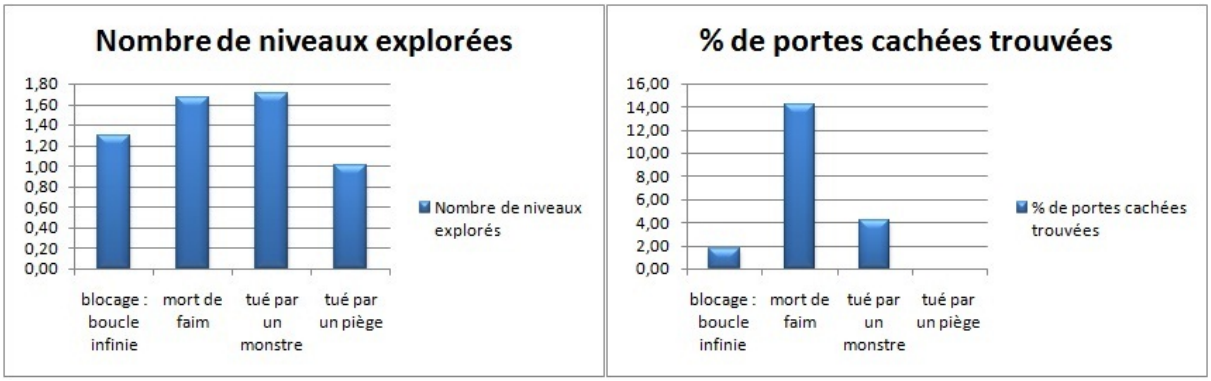

FIG. 13.3 – Passages secrets et escaliers

Il faut se rappeler ici que le bot est codé de façon à ce qu'il va préférer descendre un escalier que chercher des salles secrètes. On voit très clairement le résultat de cette stratégie ici. Le blocage représente toujours une valeur moyenne, et le piège reflète bien le fait que le bot est mort en fouillant le 1er niveau, ce qui explique les scores élevées sur les graphes précédents. Les différences entre les deux autres morts nous révèlent deux choses. Tout d'abord, lorsque le bot est coincé, et n'a d'autres solutions que de chercher des passages secrets, il y arrive plutôt bien, ce qui lui permet même de trouver un escalier. Cette constatation permet de relativiser la conclusion précédente. Et ensuite, nous voyons encore une fois que ce qui fait que les monstres tuent le bot, c'est que le bot s'aventure trop vite, trop loin dans les profondeurs du donjon là où les monstres sont plus forts.

13.3 Conclusion

Cette série de tests nous a appris plusieurs choses sur notre bot. Tout d'abord, il est indispensable de cibler et de corriger les bugs restant, car beaucoup trop de parties ne peuvent se finir normalement. Ensuite il faudra, si nous voulons renforcer notre bot, lui donner plus de possibilités d'action comme par exemple se battre, ou encore pouvoir se nourrir. Mais dans l'état actuel des choses, il est également envisageable d'optimisé au maximum le bot afin d'améliorer ces statistiques. Par exemple, et ce graphique

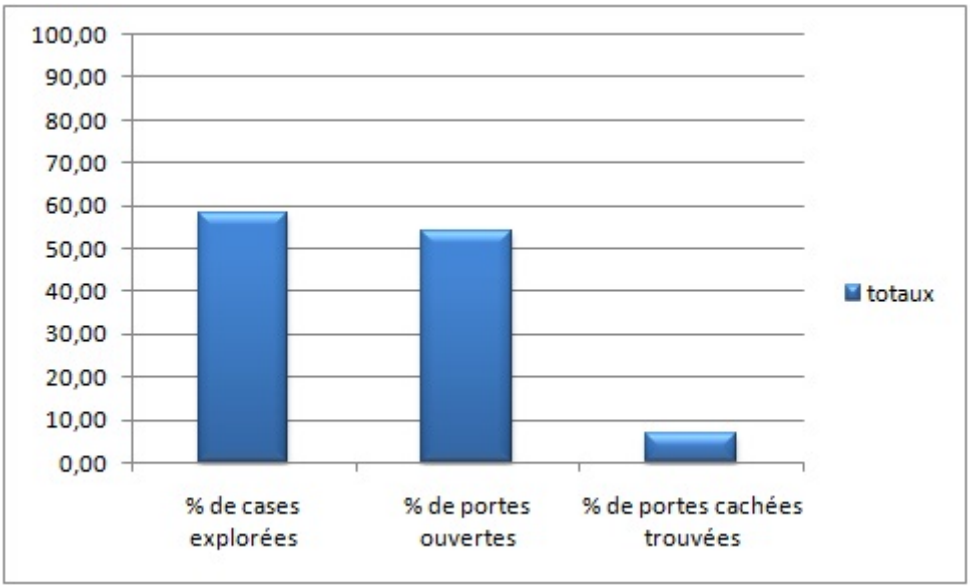

Fig. 13.4 – Moyennes sur toutes les morts

de la moyenne globale des statistiques le montre bien (fig. 13.4), il serait judicieux de changer les conditions de descente d'un étage. En prenant par exemple en considérations le niveau d'expérience actuel du bot, et calculer ses chances de survies dans un milieu plus dangereux que le niveau actuel. Cela permettrait par la même occasion de mieux fouiller un niveau, car si le bot considère que descendre est encore trop dangereux pour lui, il pourrait fouiller le niveau afin de trouver des passages secrets.

Chapitre 14

Améliorations possibles

Malgré nos efforts, nous n'avons pas pû faire tout ce que nous pensions. En effet, plusieurs phases du projet nous ont posé problème et nous avons pris du retard que nous n'avons malheureusement pas réussi à rattraper.

C'est pourquoi nous allons ici parler des différentes améliorations que nous jugeons utiles d'apporter, ou qui pourraient être interressantes. Ces améliorations n'ont pas été faites car certaines nous demandaient trop de travail supplémentaire que nous ne pouvions fournir en raison du temps dont nous disposions, et d'autres n'ont tout simplement pas été menées à bien.

14.1 Caractéristiques du bot

14.1.1 Survivre

La survie est probablement le besoin le plus important puisque c'est ce qui va permettre au bot d'explorer le plus de terrain possible et de tuer le plus grand nombre de monstres. En effet, s'il n'est pas capable de se nourrir ou qu'il ne sait pas quand le faire, il mourra et idem si il ne sait pas chercher de la nourriture et s'en faire un stock.

Pour la même raison, il doit être capable de se battre face à un monstre et aussi d'évaluer ses chances pour ne pas se faire tuer par le premier venu.

Se nourrir

La faim est gérée dans le jeu grâce à une variable qui se décrémente dans le noyau. Suivant la valeur de cette variable, le joueur est informé de l'état de le faim du héro. La faim est représentée par 7 états que le joueur connait : SATIATED, NOT_HUNGRY, HUNGRY, WEAK, FAINTING, FAINTED et STARVED.

Trouver de la nourriture

Une amélioration possible serait de gérer le stock de nourriture du bot. Lorsque le stock de nourriture atteint un certaint seuil, alors le bot doit mettre en attente ce qu'il était en train de faire et se concentrer sur la recherche de nourriture.

Combattre

Il serait aussi interressant que le bot soit capable de combattre, ce qui signifie choisir une arme, s'en équiper et l'utiliser contre les monstres.

Risques Le risque principal réside dans les choix faits par le bot : s'il considère qu'il est temps de se nourrir, il pourra se faire tuer par un monstre proche ou, à l'inverse, s'il choisit de combattre, il ne faut pas qu'il meurt de faim.

Parades Un poids sera attribué à chaque besoin du bot. Ils seront mis à jour à chaque tour en incrémentant ou en décrémentant le poids d'un besoin en fonction de la situation. Prenant en compte ainsi tous les paramètres, le bot mettra en place la stratégie la mieux adaptée.

Avec notre implémentation actuelle, le bot ne sait pas reconnaitre un monstre donc il combat dans le sens où il "fonce" dedans sans s'arrêter. Hors lorsque l'on veut aller sur la case d'un monstre, alors cela lui inflige des coups. Nous pouvons donc considérer que notre bot se bat d'une certaine façon.

Se soigner

Le bot pourrait aussi se soigner. Pour cela, il faut regarder les points de vie du bot et décider de se soigner à partir d'un certain seuil par exemple.

14.1.2 Dormir

De la même façon que pour la faim, on peut décider de dormir lorsque le niveau de fatigue atteint un certain seuil. Il faut cependant regarder qu'il n'y a pas de monstres aux alentours pour ne pas être tué pendant le sommeil.

14.1.3 Gestion de l'inventaire

Il serait aussi interressant que le bot soit capable d'identifier les objets au sol et de les ramasser.

De plus, il devra être capable de lister un certain type d'objet dans l'inventaire (liste des armes, nombre de pièces, ...).

Il devra s'équiper d'une arme (la meilleure), être capable d'échanger des objets avec des PNJ[1] et être capable de jeter des objets pour en stocker de meilleurs.

De plus le bot devra considérer le poids de son inventaire afin de ne pas rester bloqué ce qui est une difficulté suplémentaire à prendre en compte.

Il faudra donc mettre en place des structures adaptées de façon à pouvoir accéder à toutes les informations rapidement. L'intelligence articielle du bot sera relativement complexe et riche pour prendre en compte toutes ces données et les considérer indépendemment des besoins ou des spécificités du bot.

14.1.4 Modification des comportements du bot

Nous avions pensé mettre en place un fichier de configuration du bot qui aurait permis à l'utilisateur de rentrer certaines options avant le lancement du jeu. Par exemple, il aurait pû spécifier que le bot ne fasse que de l'exploration, ou au contraire qu'il essait de descendre au plus bas dans le donjon.

Ceci implique donc de lire ce fichier de configuration, de savoir l'interpréter et de pouvoir agir en conséquence. Le risque principal est de configurer le bot de façon contradictoire. Si, par exemple, on fixe comme première priorité au bot de se nourrir mais que la nécessité de trouver de la nourriture est considérée comme peu importante, il y aura des conflits au niveau de la prise de décision puisque le bot voudra manger mais pas remplir son inventaire en fonction de ce besoin.

De plus, il faut essayer de faire en sorte que le fichier soit suffisament intuitif pour que l'utilisateur n'ait pas besoin d'avoir de notion particulière pour l'éditer.

14.1.5 Remontrer une partie

Il aurait aussi été interressant d'être capable de remontrer une partie, c'est-à-dire de rejouer le même niveau et revoir les choix fais par le bot. Ceci n'a malheureusement pas été fait par manque de temps, puisque nous ne savons pas comment le faire pour le moment.

Au début d'une partie, il faut donc être capable d'enregistrer le donjon complet pour pouvoir le relancer par la suite.

Ensuite, au cours d'une partie jouée par le bot, il faut donc enregistrer tous ses comportements ce qui est très couteux en temps puisqu'à chaque action, il faudra écrire dans un fichier les choix effectués.

[1] Personnage Non Joueur

14.2 Les changements de niveau

En ce qui concerne les changements de niveau, plusieurs améliorations peuvent être apportées.

14.2.1 La gestion des trous

Le premier problème

Pour éviter que notre programme ne s'arrête à cause d'une erreur, nous avons géré les trous de la même façon qu'un escalier, à la différence près qu'ils sont à sens unique. Le problème qui se pose et que nous n'avons pas eu le temps de résoudre est que lorsque le bot tombe dans un trou, un nouveau niveau est créé. Ensuite, le bot peut se déplacer dans ce niveau mais s'il emprunte un escalier qui mène vers le niveau précédent, alors le niveau sera de nouveau créé car notre structure ne permet pas de savoir que nous l'avons déjà visité.

Voici un petit schéma permettant de mieux comprendre la situation (14.1) :

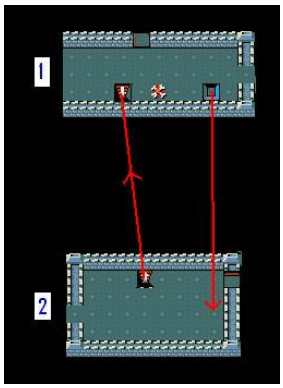

FIG. 14.1 – Exemple pour les problèmes avec les trappes

Nous voyons sur ce petit schéma qui si le bot tombe dans le trou du niveau 1, alors le niveau 2 va être créé et les initialisations vont être faites. Mais ensuite, si le bot emprunte l'escalier du niveau 2, alors le niveau 1 va de nouveau être créé comme étant un nouveau niveau. Ceci n'aurait pas eu lieu si le bot avait emprunté l'escalier du niveau 1 puis l'escalier du niveau 2 parce qu'avec les escaliers, nous savons où nous allons et d'où nous venons ; ce qui n'est pas le cas avec les trous.

Ceci n'est pas un problème dans le sens ou cela n'empêche pas le programme de fonctionner, mais dans le sens ou certains niveaux sont ajoutés plusieurs fois, ce qui fausse bien entendu les statistiques. Cependant, cette situation est plutôt rare, c'est pourquoi nous n'avons pas essayé à tout prix de trouver une solution.

Solution au premier problème

Nous avons quand même cherché une solution au problème mais sans trop de succès. En effet, ce qui nous paraît évident c'est que notre structure n'est pas adaptée pour gérer cette situation et il faudrait donc en trouver une autre.

Nous supposons que l'arborescence des niveaux est déjà gérée dans NetHack, mais nous n'avons pas trouvé comment exactement et cela ne nous parraissait pas aussi important que certains autres points, ce qui explique que nous n'ayons pas plus cherché.

De plus, nous n'aurions pas eu le temps de mettre au point une autre structure, de la mettre en place et d'effectuer tous les tests nécessaires pour s'assurer que tout le reste fonctionnait encore.

Le second problème

Une autre amélioration à apporter quant au déplacement entre les niveaux concerne le fait que dans certains cas, au lieu d'utiliser la position de notre bot à l'aide de notre variable *botPosition*, nous allons

lire la position du bot directement dans le noyau. En effet, notre variable *botPosition* n'est pas toujours à jour au bon moment. Il arrive que nous fassions des traitements alors que notre variable n'a pas encore été actualisée.

Cette fois-ci, c'est un vrai problème puisque si nous utilisons notre variable alors qu'elle n'est pas à jour, une erreur survient et le jeu "plante".

Solution au second problème

Nous nous sommes rendu compte de ce problème bien trop tard pour pouvoir espérer le résoudre. En effet, nous avons essayé de faire la mise à jour de notre variable à différents moments, sans succès.

Si nous le considérons comme un problème c'est parce que nous avions décidé de ne pas toucher au noyau, il serait donc bon de faire en sorte d'uiliser notre variable globale.

14.3 Les mouvements aléatoires

Pour ce qui est des mouvements aléatoires, nous n'avons pas géré le fait que le bot pouvait être bloqué dans le sens où il effectue un cycle de commandes. Ceci n'est pas facile à détecter et nous n'avons pas eu le temps d'y réfléchir. Nos algorithmes actuels ne permettent pas de détecter un cycle et il serait interressant de les détecter puisque cela éviterait certains bloquages.

Une autre possibilité serait de regarder la direction dans laquelle le bot est censé aller et de retourner la bonne direction.

Chapitre 15

Conclusion

Nous connaissions peu le jeu de rôle NetHack, notre premier travail dans la réalisation du projet a donc été de jouer à ce jeu afin de mieux comprendre ce que nous devions implémenter. Cette étape a été très enrichissante car nous avons pris conscience de la difficulté de ce jeu. En effet, il nous a fallu du temps avant d'être à l'aise avec les commandes mais aussi pour comprendre tous les paramètres à prendre en compte lors d'une partie. Nous avons été agréablement surpris de voir la solidarité qui existe entre les joueurs que l'on peut trouver sur le net, nous avons eu ainsi connaissance de grand nombre de stratégies ou d'astuces utiles pour tenter de gagner NetHack.

Faute de temps, nous n'avons malheureusement pu remplir intégralement le cahier des charges. Nous nous sommes mis trop tard dans la lecture du code ainsi que son expérimentation primaire. Nous avons était également bloqué par des retards lors de phases clef du projet comme la mise en place du traceur et le retour de commandes au noyau. Cela nous a tout de même appris toutes l'importances de mettre en place au plus tôt des parades efficaces face aux risques potentiels d'un projet. Notre expérience a été également renforcé en ce qui concerne l'architecture du projet. En effet, dès le début nous nous sommes concentrés à penser le projet de façon efficace et détaillé. Et au final nous avons jamais eu a remettre en cause les fondements de notre architecture. Ce qui, tout de même, nous a permis d'éviter nombre de problèmes et de retards.

Ce projet nous a permis de prendre conscience de points clefs qu'il est indispensable de mettre au point rapidement lorsqu'on travail avec plusieurs personnes : par exemple le partage des tâches, la préparation du travail, la prévision de problèmes éventuels et leurs parades associés, l'importance de la communication entre chaque membre de l'équipe et bien sûr la responsabilité qu'incombe à chacun de remplir au mieux ses objectifs afin de rendre le travail final le meilleur possible.

Bibliographie

[CC] Jean-Christophe Collet and David Cohrs. `http://nethack.sourceforge.net/docs/nh343/lev_comp.txt`. Dernière visulisation le 05 Avril 2009.

[eGS04] David M.Bourg et Glenn Seeman. *AI for Game Developers*. O'reilly Media Inc, 2004.

[Ella] Rob Ellwood. `http://members.shaw.ca/rob.ellwood/HowToReadTheSource.txt`. Dernière visualisation le 12 Janvier 2009.

[Ellb] Rob Ellwood. `http://members.shaw.ca/rob.ellwood/MakingNewLevels.txt`. Dernière visualisation le 05 Avril 2009.

[FSF] Inc. Free Software Foundation. `http://sartak.org/code/TAEB`. Dernière visualisation le 05 Avril 2009.

[Har] John Harris. `http://www.gamesetwatch.com/2007/02/18-week/`. Dernière visualisation le 05 Avril 2009.

[Jac08] Scott Jacobs, editor. *Game Programming Gems 7*. Course Technology CENGAGE Learning, 2008.

[JN03] Stuart J.Russell and Peter Norvig. *Artificial Intelligence : A Modern Approach*. Person Education International, 2eme edition, 2003.

[Lab] Bell Labs. `http://www.psgd.org/paul/docs/cstyle/cstyle.htm`. Dernière visualisation le 05 Avril 2009.

[LKM] Kenneth Lorber, Kensington, and Maryland. `http://www.nethack.org/`. Dernière visualisation le 05 Avril 2009.

[Mar] German Martin. `http://members.shaw.ca/rob.ellwood/sources.txt`. Dernière visualisation le 05 Avril 2009.

[ML] Shawn M. Moore and Jesse Luehrs. `http://search.cpan.org/~sartak/NetHack-Item-0.05/lib/NetHack/Item.pm`. Dernière visualisation le 05 Avril 2009.

[NWL+07] Patrick Naïm, Pierre-Henri Wuillemin, Philippe Leray, Olivier Pourret, and Anna Becker. *Réseaux bayesiens*. Eyrolles, 3eme edition, 2007.

[Ray] Eric S. Raymond. `http://www.nethack.org/v343/Guidebook.html`. Dernière visualisation le 05 Avril 2009.

[Ste] M. Stephenson. `http://nethack.sourceforge.net/docs/nh343/dgn_comp.txt`. Dernière visualisation le 05 Avril 2009.

[Tea] NetHack Dev Team. doc/window.doc. Support des différentes interfaces graphiques dans NetHack.