ENSEIRB-MATMECA

Projet du semestre 6
Informatique

Penguins

Auteurs :
Alexis Chan
Florian Fichant
Sébastien Gruchet
Florian Le Vern

Responsable :
M. Emmanuel Jeannot

Mai 2015

Sommaire

Introduction

Ce projet consiste à implémenter en langage C le jeu de 'Hey that's my fish'/'Pinguins'. Celui-ci oppose un certain nombre de pingouins par joueur sur un plateau constitué de tuiles, chacune d'elles offrant une récompense au joueur qui se tient dessus. A chaque déplacement de joueur, la case où il se tenait est retirée du plateau. Quand un joueur ne peut plus se déplacer, il est automatiquement retiré de la partie et son score est fixé par le nombre de poissons indiqués sur les tuiles qu'il a occupées. Le gagnant est ainsi le joueur avec le plus de points.

L'implémentation du jeu doit permettre de réaliser des parties avec un nombre quelconque de joueurs, et se doit d'être le plus générique possible au niveau du graphe représentant le plateau de jeu.

Afin de réaliser ce projet, il est demandé plusieurs choses :

— Une interface commune à tous les groupes de projets
— La mise en place d'une structure de données pour les graphes résultant du déroulement du jeu
— La réalisation d'un serveur de jeu, qui puisse accueillir des clients de jeu et gérer le déroulement d'une partie
— La réalisation d'au moins deux stratégies clientes, pouvant être chargées dynamiquement sous forme de librairies

1 Analyse du projet et représentation des données

Tout d'abord, l'étude préliminaire du sujet, en concertation avec les autres groupes, a permis de dresser une liste des besoins d'une stratégie ou du serveur. La plupart de ces fonctions ne servent qu'à rendre facilement accessibles les données du serveur et sont ainsi réalisées de manière assez directe, avec un champ dans la structure concernée qui stocke ces informations. Par exemple, le nombre de joueurs, la liste des pingouins d'un joueur et leur position, le score de chaque joueur...

Pour la communication d'une stratégie vers le serveur, trois fonctions sont nécessaires :
- `char * get_client_name();` qui permet à une stratégie d'indiquer son nom au serveur
- `void place_pawn(int * dst);` qui permet à une stratégie d'indiquer son prochain coup au serveur
- `void play(int * src, int * dst);` qui permet au serveur de donner la main à une stratégie pour décider de son coup

Pour la communication du serveur vers une stratégie, certaines fonctions permettent d'obtenir des informations globales sur l'état du jeu comme `int who_am_i();` qui permet à une stratégie de s'identifier par rapport aux autres, `int nb_sides(int tile);` qui permet de récupérer le nombre actuel de voisins d'une tuile, ou `int nb_fish(int tile);` qui permet de récupérer le nombre de poissons sur une tuile précise.
D'autres s'intéressent plus particulièrement aux interactions des joueurs et des tuiles : comme par exemple `int nb_neighbours(int tile);` pour obtenir le nombre de voisins d'une certaine tuile, `int tile_neighbours(int tile, int ** neighbours);` pour obtenir la liste de ces mêmes voisins, ou encore `int tile_ahead(int previous, int current);` pour savoir si il est possible de se déplacer en ligne droite à partir d'une tuile.

Etant donné que le plateau de jeu doit constamment être mis à jour en relation avec les différents coups des joueurs, ainsi que rendre facilement accessible ses données, la structure de données privilégiée pour réaliser le graphe semble être une matrice d'adjacence, qui gère les différentes relations entre les tuiles (les liens entre elles) et donc la possibilité ou non de passer de l'une à l'autre.
Toutes les structures tuiles sont mémorisées dans un tableau au début de la partie et la matrice illustre les relations entre elles.

Ainsi, la structure pour le graphe ressemble à ceci :

```
struct graph{
    //first is the actual tile
    //second is the previous tile
    int matrix[MAX_TILE][MAX_TILE];
};
```

La case matrix[i][j] de la matrice contient l'id de la tuile se trouvant "en face" de la tuile j par rapport à la tuile i. La complexité en temps d'accès à la donnée `tile_ahead()` est donc de $\Theta(1)$, c'est l'idéal pour que le client puisse accéder rapidement aux données

nécessaires à l'élaboration de sa stratégie. En contrepartie, la complexité en espace est de $\Theta(MAX\_TILE^2 + MAX\_TILE \times sizeof(struct\ tile))$.

Pour limiter la complexité en espace, une solution qui a été envisagée est le format `Yale Sparse Matrix` qui à une matrice creuse associe une série de tableaux, ce qui diminue la complexité en espace contre une augmentation de la complexité en temps pour certaines opérations.

Le format `Yale Sparse Matrix` est une série de tableaux dans lequel le minimum d'information d'une matrice est stocké, pour permettre un gain de place potentiel. Ici, la matrice d'adjacence peut être représenté par trois tableaux :

- le premier tableau rassemble l'ensemble des données non nulles dans l'ordre où elles apparaissent dans la matrice de gauche à droite et de haut en bas.
- le second tableau (de taille n, le nombre de tuiles total) représente pour chaque case, le nombre d'éléments de la matrice présent strictement avant la première case de la ligne correspondante, en d'autres termes, si les lignes 0, 1 et 2 contiennent respectivement 2, 3, et 6 éléments, le tableau sera ainsi rempli : $[0, 2, 5, 11]$. Cette représentation permet de coder les lignes correspondant aux données.
- le troisième tableau représente pour chaque donnée sa colonne dans la matrice.

De plus, pour améliorer potentiellement la rapidité de recherche, nous avons choisi de rajouter un quatrième tableau, qui représente le nombre d'éléments manquants avant un élément dans sa ligne. Par exemple, pour l'élément (2,3), à l'indice correspondant à sa position dans le premier tableau, le quatrième tableau indiquera 3 si (2,0), (2,1) et (2,2) ne sont pas présents, 2 si seuls deux d'entre eux ne le sont pas, etc.. Ceci permet d'accélérer la recherche d'éléments contre une augmentation faible relativement à la taille totale prise par l'ensemble des tableaux.

L'utilisation d'une matrice donne pour chaque opération, une complexité en temps suivante :

- ajout d'une tuile et de ses voisins : $\Theta(1)$
- retrait d'une tuile et de ses voisins : $\Theta(1)$
- obtention de la tuile opposée à une autre : $\Theta(1)$
- obtention de la liste des voisins d'une tuile : $\Theta(MAX\_TILE)$, MAX_TILE^2 étant le nombre de tuiles du plateau

L'utilisation du format `Yale Sparse Matrix` permet quant à lui d'obtenir pour chaque opération la complexité en temps suivante :

- ajout d'une tuile et de ses voisins :Θ(n), n étant le nombre de tuiles du plateau
- retrait d'une tuile et de ses voisins : Θ(n), n étant le nombre de tuiles du plateau
- obtention de la tuile opposée à une autre : Θ(log m), m étant le nombre de valeurs non nulles de la matrice équivalente
- obtention de la liste des voisins d'une tuile : $\Theta(1)$

On peut voir que l'on échange une construction du graphe plus lente pour une meilleure capacité de recherche des voisins et une faible perte de vitesse pour obtenir une tuile opposée à une autre. Pourtant, utiliser cette méthode permet d'économiser beaucoup en espace : la complexité est en effet en Θ(n + 3 * m) où n est le nombre de tuiles du plateau et m est le

nombre de valeurs non nulles de la matrice équivalente, soit 2 pour une liaison entre deux tuiles. Pour un plateau avec des tuiles à peu de côtés, cela revient à augmenter l'espace nécessaire, mais dès que l'on dépasse un certain nombre de côtés, étant donné que l'on travaille a priori sur des graphes planaires, le nombre de valeurs nulles de la matrice explose et le format `Yale Sparse Matrice` est alors bien plus intéressant.
Pour un plateau de 100 tuiles carrées, la taille de la matrice au format `Yale Sparse Matrix` est de plus de 2800 octets , tandis que la taille de la matrice standard est de 800 octets.
Pour un plateau de 95 tuiles hexagonales(10 lignes alternées de 10 et 9 tuiles chacune), cela représente en début de partie une taille d'environ 12000 octets contre plus de 72000 octets pour la matrice standard, sachant que ce nombre va diminuer au fil de la partie.

2 Le serveur

2.1 Représentation des données

Le serveur de jeu doit posséder une structure complète qui définit l'état du plateau.
Les variables constantes tout au long de la partie, comme le nombre de joueurs, lo nombre de pions par joueur, ou le nombre de tuiles, sont initialisées dans une structure globale au moment de l'initialisait du plateau de jeu. Celui-la est définit de la manière suivante :

```
struct board {
    int ** pawns_positions;
    int current_player;
    struct tile * tiles;
    struct graph * ahead;
    struct score * scores;
};
```

Les différents champs de la structure permettent de localiser les pions des joueurs, de connaitre l'id de la stratégie en train de jouer, de stocker toutes les tuiles du plateau pour y avoir accès en temps constant, de connaitre les relations entre les tuiles (laquelle est en face de laquelle), et de stocker les scores courants des joueurs.

Remarque : on notera que de manière générale, la complexité en espace a été volontair
mise en retrait au profit de la complexité en temps.

2.2 Actions du serveur

Pour que plusieurs stratégies puissent s'affronter le serveur doit disposer de fonctions qui puissent modifier l'état du jeu défini par la structure ci-dessus :

- `struct board * board_init(char *path_to_map, int nb_player, int nb_pawns);` qui initialise le plateau de jeu en fonction de la carte du plateau donnée en .txt
- `int check_place_pawn(struct board *, int);` qui vérifie la validité du coup `place_pawn()`
- `void place(struct board *, int);` qui modifie l'état du jeu en plaçant un un pion du joueur courant sur la tuile indiquée
- `int is_alive(struct board *, int);` qui vérifie si un joueur a encore un pion en jeu
- `int check_play(struct board *, int src, int dst);` qui vérifie la validité du coup `play()`
- `void move(struct board *, int src, int dst);` qui modifie l'état du jeu en plaçant le pion de la tuile source sur la tuile de destination et en ajoutant les points au joueur courant
- `int win(struct board *);` qui indique le joueur gagnant en fonction des scores (ou égalité ou pas de joueur gagnant)
- `void remove_player(struct board *, int player);` qui élimine un joueur de la partie

— `void add_points(struct board *, int player, int points);` qui ajoute le nombre de points indiqués au joueur indiqué et lui ajoute 1 à son compteur de tuiles ramassées
— `void free_board(struct board *);` qui libère la mémoire allouée lors de l'initialisation du plateau de jeu

Grâce à ces fonction le serveur peu lancer le déroulement de la partie selon le scénario suivant :

```
Les stratégies sont chargées dynamiquement.
Tant qu'il reste des pions a placer
Si le joueur courant a encore un pion a placer
Demander l'id de la tuile sur laquelle il veut jouer
Si le coup est valide
Jouer le coup
Sinon
Eliminer le joueur
Passer au joueur suivant
Tant qu'il reste des pions en jeu
Si le joueur courant possède encore des pions en jeu
Demander le coup qu'il souhaite jouer
Si le coup est valide
Jouer le coup
Sinon
Eliminer le joueur
Passer au joueur suivant
Afficher le gagnant
Libérer la mémoire
```

2.3 Chargement des librairies dynamiques

Pour charger les différentes stratégies passées en paramètre sur la ligne d'execution du programme, la librairie `<dlfcn.h>` est utilisée. Un client est chargé dynamiquement grâce à la fonction `dlopen()` et des pointeurs vers ses trois fonctions obligatoires sont stockés dans une structure client. Les fonctions des stratégies sont appelées selon le déroulement vu ci-avant.

2.4 Gestion des plateaux de jeux

Un jeu de plateau n'en est pas un sans le plateau qui fait son nom. C'est donc un point important du serveur d'exploiter et de traiter les données du plateau. Le serveur a besoin de gérer les différents plateaux de jeu, ils peuvent être composés de tuiles de différentes formes : carrées, pentagonales, hexagonales, octogonales voire même former un pavage de Penrose. Le nombre de tuiles par plateau n'est pas défini, la forme générale du plateau non plus. Il est donc nécessaire d'avoir une structure de stockage des plateaux de jeu suffisament malléable pour convenir ces représentations toutes plus diverses et variées les unes que les autres. Nous avons choisi de stocker les plateaux sous forme de fichier texte contenant toutes les métadonnées

nécessaires à son traitement par le serveur. Donnons un exemple du début du fichier d'un de nos plateaux :

```
37
0 6 2 3 | 25 25
6|-2
1|-2
1 6 3 2 | 25 75
0|2
6|12
7|-2
```

La première ligne correspond au nombre total de tuiles du plateau, pour que le serveur puisse toutes les traiter et les ajouter à son graphe de représentation du plateau. La seconde ligne est le prélude à la description d'une tuile, respectivement dans l'ordre ils correspondent à : Identifiant de la tuile, son nombre de côtés, son nombre de relations avec d'autres tuiles, son nombre de poissons, sa coordonnée selon un axe pour la bibliothèque graphique, sa coordonées selon l'autre axe pour la bibliothèque graphique. Cette seconde ligne est suivie par les relations de la tuile sus-citée avec les autres tuiles, que nous appelons paires. C'est de cette facon que nous connaissons toutes les tuiles et leurs relations et qu'on peut permettre la mise en place de n'importe quel type de carte.

Nous avons mis en place un logiciel de génération de carte, *genMap*, qui en fonction des paramètres qui lui sont donnés, crée un fichier "map.txt". Ces paramètres sont : le nombre de tuiles totales du plateau, le nombre de cotés de chaque tuile, le nombre de tuile par coté du plateau, le mode de génération du plateau, avec par défaut un mode aléatoire.

Cette gestion des cartes ne peut se faire sans un parseur pour pouvoir traiter les données enregistrées sur chaque fichier. Ce parseur est appelé par la fonction *parseMap* qui prend pour paramètre le nom du fichier plateau de jeu et retourne une structure plateau contenant les infirmations issues du fichiers. C'est en effet la première opération réalisée sur le traitement du plateau, il est donc normal qu'elle retourne cette structure "board". On réalise ainsi la génération du graphe du serveur qui lui permet de gérer les différentes relations issues du plateau, donc la réalisation des différentes fonctions qui lui incombent.

3 L'interface graphique

Pour pouvoir observer le comportement des stratégies et vérifier celui du serveur (notamment la validité des coups), il est agréable d'avoir une interface graphique pour le jeu.
La librairie `SDL2` (complétée de `SDL_ttf`) a été utilisée.
Le principe de la SDL est de placer des surfaces ou des textures sur le plan de l'écran. Ainsi la partie la plus complexe dans le calcul de l'interface graphique réside dans l'affichage automatisé des tuiles et leur placement.

La SDL ne permettant que de remplir des rectangles, la solution choisie pour générer les tuiles est la suivante :
Pour un nombre n de cotés, on peut trouver les coordonnées des sommets d'un polygone régulier à n cotés grâce aux formules trigonométriques : $x_i = R \times cos(\frac{2 \times \pi \times i}{n})$ et $y_i = R \times sin(\frac{2 \times \pi \times i}{n})$.
On peut tracer les lignes entre ces points grâce à une fonction de l'API `SDL2`.
Afin de remplir le polygone, on circule d'un point à un autre en distinguant les pixels qui sont dans le polygone de ceux qui sont à l'extérieur.
On colorie les pixels situés à l'intérieur du polygone.

`Remarque : pour optimiser l'affichage, il aurait été possible de stocker les textures de forme des tuiles nécessaires au plateaux dans un tableau alloué dynamiquement. Cela aurait permis de ne pas re calculer la forme de chaque tuile lorsque elles sont identiques.`

Ensuite, pour placer les tuiles, la première approche fut de calculer par une opération mathématique les coordonnées des centres des tuiles. En partant d'une tuile, il est possible d'utiliser un TAD file pour dessiner tous ses voisins et ainsi de suite : c'est un parcours en largeur du graphe. Cependant le problème des tuiles en face les unes des autres et les calculs compliqués de symétrie par rapport à chaque arête de la tuile du milieu impliquent des complexités en temps et en espace qui peuvent être évités. En effet, la méthode utilisée est de définir directement les coordonnées des centres des tuiles dans le fichier texte des maps. Finalement l'interface graphique est moins automatiques qu'elle aurait pu l'être mais on trouve tout de même un gain en complexité.

4 La stratégie

L'étape suivante dans ce projet était l'élaboration d'une stratégie. Dans un premier temps la création d'une copie du plateau de jeu en interne a été enviagée. En effet, celle-ci aurait permis de posséder toutes les données en internet, et donc limiter limiter le nombre d'informations à demander au serveur à chaque tour. Cependant, pour des raisons d'efficacité en temps, mais surtout en mémoire, cette solution a été abandonnée, au profit de stratégie basées sur une communication importante avec le serveur, afin d'obtenir les informations nécessaires à la prise de décision.

Dans l'optique d'établir une startégie viable, une première stratégie a été mise en place, basé sur des déplacements aléatoires. Cette première stratégie, bien que non compétitive, a permis une première approche de l'acquisition et du stockage des données importantes, qui aura été réutilisée dans une seconde stratégie plus performante.

4.1 Copie interne du plateau

Dans un premier temps, les stratégies se basaient sur une copie interne totale du plateau de jeu, qui leur permettait d'avoir accès à l'ensemble des informations (nombre de poissons par tuiles, nombre de voisins d'une tuile, alignement des tuiles...) sans avoir à faire appel aux fonctions du serveur (`int nb_fish(int tile);`, `int nb_neighbours(int tile);`, `int tile_ahead(int previous, int current);`...).

Ainsi, au premier tour de jeu, une série d'appel au serveur serait effectuée, afin d'initialiser la copie interne. Dès lors, à chaque tour de jeu une simple série d'appels à la fonction `int pawn_positions(int player, int** positions);` afin de connaître la position de tous les pions des joueurs suffit à mettre à jour la copie du plateau à l'état actuel du jeu.

En effet, la nouvelle position des pions de chaque joueur permet de connaître les déplacements qui ont été effectués. Dès lors il est aisé d'en déduire quels sont les pions qui ont été déplacés (en se basant sur l'ancienne position des pions qui est toujours connue). De même, la déduction des tuiles qui ont été enlevées est immédiate.

Ainsi, au premier abord la construction d'une copie interne du plateau de jeu semblait apporter de nombreux avantages, notamment en terme de temps, puisque, mis à part au premier tour où de nombreux appels au serveur sont nécessaires (afin d'acquérir l'état total du plateau de jeu), seul quelques appels sont ensuite nécessaires à chaque tour afin de visiualiser la totalité du plateau de jeu.

Cependant, cette méthode s'est révélée poser de nombreux inconvénients :

— en terme d'espace

— en terme de complexité

En effet, cette méthode présentait deux inconvénients majeurs en terme d'espace :

— Tout d'abord la consommation d'espace est nettement plus importante qu'en basant une stratégie sur des demandes au serveur. En effet, on effectue une copie totale des informations du serveur, ce qui nécessite un tableau pour gérer la position de chaque joueur, un tableau afin de connaître le nombre de poissons sur chaque tuile, mais surtout, une matrice afin de coder l'alignement des tuiles (de sorte que `aligment[i][j]` contienne

la valeur de `tile_ahead(i,j)`, matrice de taille `n^2` par rapport aux nombre de tuiles, et en grande partie creuse. Pour limiter l'importance de ce problème nous avons mis en place un type abstrait de données plus adapté à la gestion de matrices creuses.
— De plus, cette méthode créait également un problème de gestion de la mémoire. En effet, afin de stocker les données représentant le plateau de jeu, un certain nombre de tableaux était nécessaire, au sein d'une variable globale, qui restait allouée tout au long de la partie (puisqu'on la modifiait à chaque pour représenter l'évolution du jeu. Il était alors malaisé de désalouer ces tableaux, car le serveur fait appel aux procédures `void play(int *src, int *dst);` `void plac_pawn(int *dst);`, qui ne peuvent désalouer ces tableaux puisque la désallocation ne s'effectue qu'après la partie entière jouée.

Cette méthode créait également des difficultés en terme de complexité des stratégies. En effet, il devenait nécessaire de gérer les mouvements des joueurs et la disparition des tuiles à chaque tour de jeu, là où une stratégie basée sur des demandes au serveur ne s'en soucie pas. De plus, le risque d'erreur était d'autant plus important, puisque l'on ne demande au serveur que la position des joueurs, et que l'on gère en interne la disparition des tuiles, toute erreur peut entraîner une mauvaise représentation du plateau, et ainsi des mouvements illégaux ou non optimaux.

En conclusion la création d'une copie interne du plateau de jeu apparaît comme une mauvaise idée car cela force la stratégie à effectuer des calculs qui sont déjà effectués par le serveur, et à stocker un certain nombre d'informations déjà connues (et obtenables facilement), ce qui entraîne une consommation de mémoire et un risque d'erreur inutile.

4.2 Stratégie aléatoire

La première stratégie réalisée est une stratégie aléatoire. Afin de réaliser cette stratégie, il a été décidé (suite aux observations précédentes) de ne pas utiliser une copie du plateau, mais plutot de faire des appels aux serveurs pour obtenir les information nécessaires.

Cette stratégie demeure plutot basique, et nous a permis de tester le bon fonctionnement de nos serveurs et générateur de plateau, tout en appronfondissant notre vision du plateau, et des necessités pour obtenir les informations. Ainsi cette stratégie est composée de deux étapes (correspondant aux deux étapes de jeu : positionnement des pions et déplacement), fonctionnant de la sorte :

— le positionnement des pions se fait en créant un tableau de toutes les cases possédant un unique poisson et non occupées, puis en sélectionne une aléatoirement.
— le déplacement des pions se fait en choisissant aléatoirement un pion, puis en le déplaçant aléatoirement sur une des cases accessibles (c'est à dire ses voisins, et les cases alignées non occupées).

Cette stratégie aléatoire n'est pas homogène (chaque pion possède la même probabilité d'être chosi, indépendanmment du nombre de chemin qui s'offre à ce pion). Ainsi, une stratégie aléatoire homogène demande au préalable le calcul de tous les chemins possibles, là où dans cette stratégie seul les chemins issus du pion déjà choisis sont calculés. Le calcul de la totalité des coups possible compliquerait légèrement l'implémentation d'une telle stratégie (puisqu'il faudrait stocker le pion a l'origine de chaque destination correspondant à un coup), mais celà

reste tout à fait possible. L'apport étant minimal (la stratégie aléatoire n'étant de doute façon pas compétitive), le choix a été fait de ne pas l'implémenter, mais de conserver cette notion qui va caractériser un déplacement possible :

- le pion à l'origine du mouvement
- la direction du mouvement (le `neighbour` que l'on choisit
- le nombre de déplacement dans cette direction (le nombre de `tile_ahead()` sur lequel on se déplace dans cette direction

afin de s'en servir dans d'autres stratégies plus avancées, où il devient plus intéressant de pousser la simulation des coups potentiels.

Cette stratégie aléatoire est d'une complexité relativement faible, puisque d'un point de vue espace elle crée deux tableaux de tailles linéaires en le nombre de tuiles (qui représentent les voisins, ainsi que les tuiles accessibles depuis la tuile du pingouin choisi aléatoirement) dans le pire des cas, qui sont ensuite désaloués à chaque tour, elle est donc de complexité en espace linéaire en le nombre de tuiles. En complexité en temps elle est également de complexité linéaire puisqu'elle va réaliser au plus n appels à `tile_ahead` (pour obtenir le tableau des coups possible), qui fonctionne en temps constant.

Ainsi la complexité, tant en temps qu'en espace, de cette première stratégie est relativement faible (comparé à l'implémentation d'une copie interne, qui utilisait au moins une matrice de taille quadratique en le nombre de tuiles.

4.3 Stratégie non aléatoire : intelligente

Il est important qu'on aie une stratégie non-aléatoire, c'est à dire intelligente pour que nous puissions montrer une possibilité d'intelligence artificielle. Nous sommes partis sur une stratégie de pondération de tous les chemins possibles par tous nos pions, ce qui est proche d'une stratégie de minmax. Tout comme la stratégie aléatoire, nous avons exploité toutes les fonctions qui nous sont offertes par le serveur, comme par exemple le nombre de voisins accessibles ou encore la tuile accessible en face. On se repose ici sur une structure *path* :

```
struct path {
  int idTileSrc;
  int firstIdTilePath;
  int idTileMax;
  float mass;
};
```

Cette structure est composée de la tuile source, de la première tuile composant le chemin, de l'identifiant de la tuile avec le plus de poissons et enfin, la masse du chemin. Cette masse du chemin est la composante principale de cete stratégie, qui nous permet de pondérer le potentiel de chaque chemin et donc du score qu'il est possible d'avoir. Cette structure permet d'avoir le meilleur chemin possible à l'issue de toutes les fonctions de la stratégie qui sont les suivantes :

```
place_pawns
getOneFishers
play
findBestPathet rien ee
```

`ponderPath`
`prediction`

4.3.1 place pawns et getOneFishers

Ces deux fonctions permettent l'initialisation du plateau en début de partie, donc le placement des pions sur les tuiles avec un seul poisson. La fonction *place pawns* est une fonction essentielle et requise par le serveur pour que la stratégie puisse être jouée dans le groupe. Il était prévu que *place pawns* pose les pions de la stratégie sur les tuiles les plus prometteuses en terme de tours prédits, néanmoins, sur la version du serveur subversion, il n'y a pas d'intelligence quant au placement initial de ces pions, car des difficultés sont apparues au moment de l'implémentation de la prédiction, sur laquelle une des sections suivante est dédiée.

Pour connaître toutes les tuiles avec un unique poisson, une fonction *getOneFishers* a été écrite, qui parcourt le plateau dans son intégralité et renvoie un tableau contenant l'identificateur de toutes les tuiles contenant un unique poisson.

4.3.2 play

C'est la seconde fonction requise par le serveur, c'est elle qui régit chaque tour de jeu de la stratégie. Elle fait appel à la fonction implémentée par le serveur *pawns positions* qui renvoie la positions de tous nos pions, puis enchaine sur l'appel de *findBestPath* qui permet d'avoir le meilleur chemin en fonction de la position de tous nos pions. Enfin, elle envoie au serveur le pion qu'on souhaite déplacer, donc sa coordonnée d'origine et sa destination.

4.3.3 findBestPath

Fonction essentielle pour la stratégie souhaitée, elle permet de ne garder que le chemin le plus prometteur pour la suite du jeu via la pondération. Elle pondère à l'aide de la fonction *ponderPath* tous les chemins accessibles depuis la tuile indiquée en argument, ne renvoyant que le chemin avec le plus de potentiel.

4.3.4 ponderPath

Seconde fonction essentielle de la stratégie, elle regarde le potentiel d'un chemin en faisant la moyenne de tous les poissons du chemin en pondérant celle-ci d'autant plus s'il y a des tuiles avec un nombre important de poisson.

4.3.5 prediction

Cette fonction aurait été la troisième de nos fonctions essentielles si son implémentation était un peu plus poussée. Elle prédit les meilleurs coups suivants possibles sur les n prochains tours avec n le nombre de tours qui est donnée en paramètre. Elle renvoie donc le nombre potentiel de poissons collectables par un pion.

Cette stratégie est améliorable par l'implémentation de la fonction prédiction à plusieurs étapes : dans *place pawns* pour avoir un placement optimal des pions au début de la partie, dans *play* pour pouvoir déplacer un des pions de la stratégie de façon à préparer les tours suivants. On pourrait également modifier la façon de pondérer pour avoir une pesée différente

des chemins. Elle peut également tendre vers une stratégie MinMax pour avoir le coup le plus optimal, car nous avons ici uniquement la partie Max de l'algorithme sus-cité.

5 Résultats et perspectives

5.1 Tests

Pour s'assurer du bon fonctionnement des fonctions, nous avons réalisé un ensemble de tests pour chacune d'elles, ainsi que vérifier avec l'aide de valgrind de potentielles erreurs mémoires dans plusieurs cas pour repérer des dépassements de tableaux et des oublis de désallocation mémoire.

5.2 Améliorations

Dans l'optique d'améliorer les stratégies déjà en place, une stratégie basée sur le min-max a été envisagée. Cependant, de nombreux obstacles sont apparus, et cette solution n'a pas été retenue. En effet, bien que très performante, la mise en place d'un algorithme min-max s'est révélé très complexe, ce qui fait qu'une stratégie basée sur des choix différents à été choisie.

La mise en place d'un algorithme min-max présente un premier incovénient du à la complexité de sa mise en place de façon générique. En effet, parmi les critères du projet, il est stipulé que le jeu soit jouable avec un nombre de joueur indéfini. Ainsi, si il est relativement aisé d'établir le gain maximum et la perte minimum dans le cas d'un jeu à deux joueurs, l'ajout de joueurs supplémentaires rend le problème autrement plus complexe, puisqu'il faut alors :

- calculer les mouvements de chaque joueur (on ne joue plus un tour sur deux, mais un sur trois ou quatre etc...)
- établir un ordre de priorité complexe (on ne compare plus le gain entre nous et un autre joueur, mais le gain entre nous et de nombreux autres joueurs, ce qui rend extrement compliqué l'établissement d'un ordre de priorité.

Ainsi, en raison de ces complications (notamment l'établissement d'un ordre de priorité global, s'adaptant à un nombre variant de joueurs), l'implémentation d'un stratégie min-max a été abandonné dans un premier temps, mais jamais complétement oubliée, car elle demeure une excellente stratégie une fois mise en place.

En plus de ça, la version utilisant le format `Yale Sparse Matrix` n'a pas été réalisée à temps de manière optimale, et n'est pas donc présente dans les dossiers sources du projet, alors qu'elle aurait présentée des avantages non négligeables à la représentation du graphe et aurait contribuée à son amélioration.

Conclusion

Pour conclure, le projet 'Hey that's my fish'/'Pinguins' s'est révélé très formateur sur de nombreux points. Tout d'abord, ce projet a été l'occasion de consolider les acquis du premier semestre en matière de programmation en langage C, ainsi que de mettre en oeuvre les acquis supplémentaire du semestre 6. En outre, ce projet nous a également donné l'occasion de travailler notre capacité à coder de manière aussi générique que possible (notamment la gestion du graphe, qui pourra se révéler utile à l'avenir). De plus ce projet nous a également permis de nous intéresser à de nombreux aspects qui n'avaient pas été abordé au premier semestre, notamment l'aspect serveur, ainsi que la réalisation d'une interface graphique.

En somme ce projet s'est révélé très formateur, et extrêmement complet ce qui nous a permis de fortement consolider, et travailler nos points faibles, en langage C, le tout dans un cadre relativement ludique (puisqu'il s'agit au final de l'implémentation d'un jeu), qui s'est révélé propice à l'expérimentation.