



ENSEIRB-MATMECA : FILLIÈRE INFORMATIQUE

Projet S6 : Penguins

Auteurs :

Sylvain CASSIAU
Quentin HARSCOËT
Victor SAINT GUILHEM
Étienne THIERY

Enseignant :

Frédéric HERBRETEAU

Groupe 1327

10 mars 2016

Table des matières

Introduction	2
1 Étude du sujet	3
1.1 Modélisation du jeu	3
1.2 Définition de l'interface	4
1.3 Interface finale	4
2 Serveur	6
2.1 Structure de données	6
2.2 Utilisation de Banquise	6
2.3 Fonctionnement de Banquise	7
2.3.1 Initialisation du serveur	7
2.3.2 Initialisation des stratégies	9
2.3.3 Tours de placement	10
2.3.4 Tours de jeu	11
2.3.5 Clôture du jeu	12
3 Stratégies	13
3.1 Contexte des stratégies	13
3.2 Stratégie aléatoire	13
3.3 Stratégie à un coup	13
3.4 Stratégie évoluée	14
4 Interface graphique	16
4.1 Généralités	16
4.2 Organisation du code	16
4.2.1 Structures de données	16
4.2.2 Fonctionnement	16
5 Tests	17
A RÈGLES DU JEU	18
A.1 Le plateau	18
A.2 Les joueurs	18
A.3 La partie	18
A.3.1 Le début de la partie	18
A.3.2 Le déroulement de la partie	18
A.3.3 La fin de la partie	18

Introduction

Ce projet a pour but l'implémentation en langage C du jeu de plateau *Hey, That's My Fish!*, traduit *Pingouins* en français et renommé *Penguins* pour ce projet.

Le but de ce jeu pour chacun des joueurs est de cumuler, grâce à ses pingouins, le plus de poissons possible. Ces poissons sont disposés sur les tuiles (originellement hexagonales) qui forment le plateau. À chaque tour, les joueurs déplacent un de leurs pingouins en ligne droite, dans n'importe quelle direction et sur n'importe quelle distance, faisant ainsi disparaître leur tuile de départ (après bien sûr avoir collecté les poissons qui s'y trouvaient). De fait, ce déplacement se rapproche de celui de la tour aux échecs. Le règles du jeu sont détaillées en annexe A.

Étant imposé le fait de pouvoir interchanger les stratégies entre équipes, une modélisation commune du jeu a dû être mise en place. Ce travail a aboutit en la rédaction d'une interface commune. Cette interface ayant été notre principale contrainte, le cheminement pour y parvenir sera décrit en première partie. Nous décrirons ensuite l'algorithme du serveur et celle des différentes stratégies implémentées. Enfin, une interface graphique ayant été mise en place, elle sera décrite dans une dernière partie.

1 Étude du sujet

1.1 Modélisation du jeu

Dans le cadre du projet, il était imposé de pouvoir échanger des stratégies entre groupes, et de les faire s'affronter. Cette contrainte a mené à une séparation naturelle du projet en deux parties : un serveur et des stratégies.

Le serveur centralise les ordres émis par les différentes stratégies, les traite (vérifie qu'un coup est valide par exemple) et fait avancer la partie. Les stratégies, elles, analysent la partie en cours, calculent un coup et le communiquent au serveur. Ainsi, globalement, une partie se déroule selon le schéma présenté en figure 1.

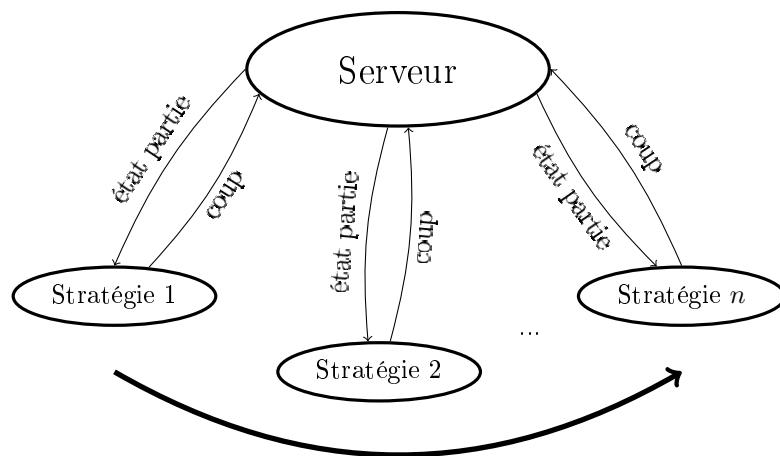


FIGURE 1 – Interaction entre stratégie et serveur

Du côté serveur comme du côté stratégies, des structures de données ont été mises en place pour modéliser les joueurs, et le plateau lui-même. Bien que propres au serveur d'une part et à chaque stratégie d'autre part, ces structures se sont logiquement avérées très semblables. En effet, nous avons toujours modélisé le problème de détermination d'un coup valide et intéressant, par un problème d'algorithmique de graphe. Ainsi on retrouve dans le serveur, et dans les stratégies, une structure de graphe dont les noeuds sont des structures représentant les tuiles, et les arêtes donnent la relation de voisinage entre elles. Cette structure est enrichie de plus ou moins d'informations, mais reprend les grandes lignes de la structure de graphe du serveur, dont voici la description.

Structure graph du serveur
Entier : nombre de tuiles
Tableau de structure tile
Entier : nombre initial de joueurs
Entier : nombre initial de pingouins par joueur
Tableau à deux dimensions d'entier : position des pingouins

Le plateau de jeu est constitué d'un ensemble de tuiles sur lesquelles sont placés les pingouins des joueurs. Ainsi la structure graph contient un tableau de structures tile, dont l'ordre est constant. Elle contient également un tableau à deux dimensions qui indique pour chaque pingouin de chaque joueur le numéro de la tuile sur laquelle il est positionné.

Structure tile du serveur

Entier : nombre de poissons
Entier : identifiant du pingouin présent sur la tuile
Entier : nombre de tuiles adjacentes
Tableau d'entiers : numéros des tuiles adjacentes

Chaque structure `tile` comporte des entiers indiquant le nombre de poissons présents sur la tuile et l'identifiant du pingouin présent (ou à un code `EMPTY` s'il n'y en a pas). Cette redondance avec le tableau de positions de la structure `graph` permet un accès en temps constant au lieu de linéaire. Enfin, pour parcourir le graphe, chaque structure `tile` contient le tableau des numéros des tuiles voisines.

Abordons maintenant le problème de la communication entre le serveur et les stratégies malgré des structures de données potentiellement différentes.

1.2 Définition de l'interface

La description du projet stipulant que les stratégies doivent être chargées dynamiquement par le serveur et interchangeables entre les équipes d'un même encadrant, le système de communication entre stratégie et serveur (que nous appelons *interface*) a fait l'objet de discussions et de mise en commun d'idées.

Notre équipe a dans un premier temps proposé un ensemble de fonctions que le serveur devrait fournir, et qui aurait permis aux stratégies de récupérer toutes les données nécessaires quand elles le souhaitaient. Cette structure aurait imposé une gestion du plateau du côté du serveur, mais pas nécessairement du côté de la stratégie, puisque l'ensemble des fonctions fournies par le serveur avait pour but de garantir un accès aux données pour la stratégie.

Nous avions également proposé que chaque stratégie implémente une fonction `init()` et une fonction `play()`. La fonction `init()` devait se contenter de préparer la stratégie. La fonction `play()`, quant à elle, devait être appelée à chaque fois que la stratégie devait jouer, et retourner le coup à jouer. Parallèlement à cette proposition, l'idée de fournir à `play()` la liste des coups joués depuis la dernière fois permettait de limiter le nombre d'appels aux fonctions du serveur en laissant la stratégie gérer ses propres données concernant l'état du plateau de jeu.

Une première version de l'interface commune a donc utilisé ces deux idées en parallèle, avant de ne garder que la deuxième, diminuant ainsi le nombre de fonctions imposées pour le serveur. Il a donc également fallu communiquer l'intégralité du plateau initial à la fonction `init()`.

1.3 Interface finale

Après cette première phase d'analyse du problème, nous sommes d'un commun accord parvenus à une solution.

Les moyens de communication entre les stratégies et le serveur se limitent à quatre fonctions, chacune utile à un moment précis de la partie. Pour permettre le début de la partie, serveur et stratégies doivent tous deux connaître l'état initial du plateau : c'est le rôle de la fonction `init()`. Cette fonction, appelée pour chaque stratégie par le serveur se chargera d'allouer la mémoire nécessaire et d'initialiser les structures de données de la stratégie. Pour cela, le serveur passe en arguments toutes les informations structurelles de la partie (nombre de tuiles et répartition des poissons, liens entre les tuiles, nombre de joueurs et de pingouins). Les liens entre les tuiles sont représentés par un tableau de structure `edge` :

Structure edge

Entier : tuile de départ
Entier : tuile de destination

Cette structure contient simplement les entiers nécessaires pour indiquer quelles tuiles cette arrête lie

Pour le déroulement de la partie, il a été convenu de distinguer le début de partie (placement des pingouins) du reste. Dans un premier temps, le serveur fera jouer les stratégies via leur fonction `place()`. Ensuite, ce sera la fonction `play()` de chaque stratégie qu'il appellera pour récupérer leurs coups. Ces deux fonctions ont le même fonctionnement si ce n'est qu'elles ne seront pas appelées durant la même phase de jeu : elles prennent toutes deux en entrée l'évolution de la partie depuis le dernier coup de la stratégie (la liste des coups des autres stratégies) et fournissent en sortie un coup.

Enfin, il a été ajouté une fonction `done()` dans le but de libérer la mémoire en fin de partie.

Aussi, pour les fonctions de jeu, il a été nécessaire de créer une structure pour représenter les coups, structure qui est précisée ci-contre.

Structure move

Entier : joueur dont c'est le coup
Entier : tuile de départ
Entier : tuile de destination

On remarque que ce choix de l'interface permet au serveur et aux stratégies une totale liberté quant au choix de la façon de représenter le plateau lui-même. Cependant, parce qu'aucune raison n'a poussé à choisir une représentation différente de celle donnée en 1.1, c'est celle-ci qui a été communément choisie.

2 Serveur

Cette partie présente le fonctionnement de la partie serveur du projet. Celle-ci correspond aux modules `server` et `main`, et permet de faire jouer les différentes stratégies entre elles. Dans cette section on s'attache également à vérifier que l'implémentation proposée répond correctement au problème défini dans le sujet et le fichier d'interface.

2.1 Structure de données

Une fois l'interface serveur client définie, plusieurs structures de données ont du être créées pour représenter une partie du jeu. Dans un soucis de simplicité, leur organisation se rapproche de celle qu'on pourrait retrouver sur la table lors d'une partie du jeu de plateau. Deux de ses structures, la structure `graph` et la structure `tile`, ont déjà été présentées.

Structure `server`

Pointeur vers une structure <code>graph</code>
Entier : nombre de stratégies
Tableau de structures <code>strategy</code>
Tableau d'entiers : scores

La structure générale du serveur est composée du plateau de jeu (une structure `graph`), autour duquel se placent les joueurs représentés par un tableau de stratégies, et d'un tableau de scores. La taille de ces tableaux est égale au nombre de joueurs.

Structure `strategy`

Pointeur vers une bibliothèque
Pointeur vers une fonction <code>init()</code>
Pointeur vers une fonction <code>place()</code>
Pointeur vers une fonction <code>play()</code>
Pointeur vers une fonction <code>done()</code>

Chaque stratégie est constituée de 4 fonctions. Afin de pouvoir les appeler au cours de la partie, chaque structure `strategy` contient donc 4 pointeurs de fonctions. }A ceux-ci on a ajouté un pointeur vers la bibliothèque de la stratégie, une "handle" retournée lors de son chargement dynamique par `dlopen`, à conserver pour libérer la mémoire en fin de partie.

Avec ce choix de structure de données, toutes les informations sont accessibles en temps constant. Ainsi, les accesseurs et mutateurs développés pour ces structures font pour la plupart une seule ligne de code, et servent plutôt à clarifier le code qu'à le raccourcir.

Afin de simplifier le code, et d'éviter de devoir passer un pointeur vers la structure `server` à chaque appel de fonction, une variable globale de type `server` est déclarée en début de module.

2.2 Utilisation de Banquise

Dans la version physique du jeu, pour des raisons pratiques, le nombre de joueurs est limité à 4, tout comme le nombre de pingouins par joueur. De plus, la banquise est toujours un pavage hexagonal de taille 7 par 8. Pour ce projet, le serveur de jeu, libre de ces contraintes physiques, devait permettre de jouer des parties sur des pavages de formes variées, avec un nombre de joueurs et un nombre de pingouins par joueur limités uniquement par les capacités de la machine.

Pour répondre à ce besoin, la fonction principale du serveur attend plusieurs paramètres dont dépend le déroulement de la partie : le nombre de joueurs, le nombre de pingouins placés par chaque joueur, la forme des tuiles du plateau, la façon dont elles sont disposées, leur nombre et les stratégies utilisées par les joueurs. Ces valeurs sont à passer en arguments à l'exécutable `Banquise`, qui permet de lancer une partie, avec la syntaxe suivante :

```
./banquise <nb de pingouins par joueur> <motif> <largeur du plateau> <hauteur du plateau> <stratégie-1> <stratégie-2> ... <stratégie-n>
```

Le motif est à choisir parmi 3 types de pavages réguliers ayant été implémentés : "full-tetra" (pavage carré), "full-hexa" (pavage hexagonal, comme dans le jeu original), "octo-tetra" (pavage d'octogones et de carrés). La largeur et la hauteur du plateau se mesurent en nombre de tuiles, et leur produit donne le nombre de tuiles total du plateau. Le nombre de stratégies passées en arguments détermine le nombre de joueurs.

2.3 Fonctionnement de Banquise

Une partie est constituée de 5 phases consécutives, auxquelles correspondent les 5 fonctions principales du module server. L'exécutable Banquise utilise successivement ces 5 fonctions, dont nous allons maintenant détailler le fonctionnement.

2.3.1 Initialisation du serveur

Cette phase est prise en charge par la fonction `server_init()`, cette dernière effectue successivement :

- Une vérification des arguments par la fonction `correct_main_args()` (il faut au moins deux stratégies, une configuration de tuiles connue, et un plateau assez grand pour le nombre de pingouins par joueur).
- L'allocation de la structure `server` globale
- L'allocation de son champ tableau de stratégies, puis son initialisation via chargement dynamique des bibliothèques des stratégies par la fonction `server_load_strats()`
- L'allocation et l'initialisation à 0 de son champ tableau des scores.
- L'allocation et l'initialisation de sa structure `graph` par la fonction `graph_init()`

Chargement dynamique des stratégies

Il était attendu que le serveur soit capable de charger dynamiquement pour chaque joueur une bibliothèque contenant les 4 fonctions de sa stratégie. L'intérêt de cette caractéristique du serveur réside dans la séparation entre le serveur et les stratégies en termes de compilation, et l'interchangeabilité résultante. Pourvu que le serveur et les stratégies respectent bien les spécifications du fichier d'interface, il est possible de demander au serveur de lancer une partie entre n'importe quelles stratégies sans avoir à recompiler ou re-linker celles-ci et le serveur. Si on avait utilisé des bibliothèques statiques pour les stratégies, à chaque fois que l'on aurait souhaité faire jouer une nouvelle stratégie, il aurait fallu re-linker les fichiers objets du serveur et des stratégies.

Pour pouvoir charger dynamiquement les stratégies, la fonction `server_load_strats()` prend en argument un tableau de chaînes de caractères, contenant les chemins vers les bibliothèques. Ceux-ci sont extraits des arguments fournis par l'utilisateur à Banquise par `server_init()`. Le module `d1fcn` est alors utilisé pour charger les stratégies. Pour chacune d'entre elles :

- La fonction `dlopen()` ouvre la bibliothèque, et la rend disponible via une "handle", une poignée de type pointeur polymorphe.
- Puis en fournissant cette poignée et le nom d'un symbole présent dans la bibliothèque à la fonction `dlsym()`, on obtient l'adresse de ce dernier. Sont ainsi obtenus 4 pointeurs de fonctions vers les fonctions `init()`, `place()`, `play()`, et `done()` de la stratégie. Ces derniers sont stockés, ainsi que la poignée, dans le tableau de structures `strategy`.

- À chaque appel à une fonction de `d1fcn`, son bon fonctionnement est vérifié en comparant la valeur de retour de `d1error()` avec le pointeur `NULL`. En cas d'erreur, les ressources sont libérées, un message d'erreur affiché, et la partie annulée.
- En fin de partie, la fonction `dlclose()` permettra de refermer la bibliothèque.

Génération de pavages

La structure de graphe utilisée devait d'après le sujet être suffisamment générale pour représenter n'importe quel pavage, même des pavages apériodiques comme les pavages de Penrose. C'est le cas, mais un facteur limitant est la génération de ces pavages. En effet il est complexe de mettre en place une méthode générique permettant de générer n'importe quel pavage. Pour ce projet, il a été décidé de se limiter à des pavages de polygones à nombre pair de côtés pour simplifier le calcul de trajectoire des pingouins. Nous avons encore réduit le champ des pavages possibles en se restreignant aux pavages de polygones réguliers, qui sont au nombre de 4. Finalement, a été mise au point une méthode numérique relativement simple permettant de générer 3 de ces 4 pavages. Ces derniers sont : le pavage purement carré (motif "full-tetra"), le pavage purement hexagonal (motif "full-hexa") et le pavage alterné carré/octogones (motif "octo-tetra"). On retrouve ces trois pavages en figure 2.

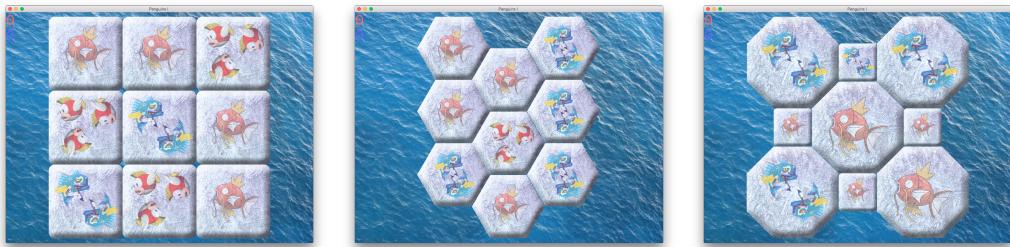


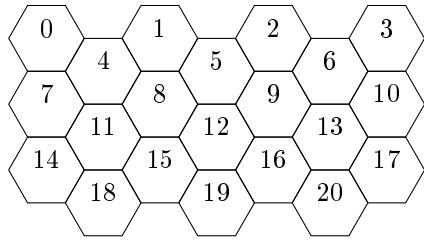
FIGURE 2 – Représentation des trois pavages sur un plateau de taille 3×3

La fonction `graph_init()` prend en arguments une chaîne de caractères correspondant au motif, deux entiers indiquant la largeur et la hauteur du plateau en nombre de tuiles, et deux entiers indiquant le nombre de joueurs, et de pingouins par joueur. Selon le motif, la largeur et la hauteur du plateau sont passées à une des trois fonctions d'initialisation de pavage `init_full_tetra()`, `init_full_hexa()` et `init_octo_tetra()`. Puis, `graph_init()` initialise les nombres de joueurs et de pingouins avec les valeurs des arguments. Enfin elle alloue et initialise le tableau des positions de pingouins à 0.

Les trois fonctions de pavage suivent le même principe. On a déterminé pour chaque motif une méthode de numérotation des tuiles, qui permet d'identifier une relation numérique entre les tuiles voisines. Par exemple pour le pavage carré, les cases sont numérotées ligne par ligne de haut en bas, puis colonne par colonne de gauche à droite (dans le sens de la lecture). Ainsi, sous réserve que ces voisins existent, le voisin gauche de la tuile x est numéroté $x - 1$, le voisin droit $x + 1$, le voisin supérieur $x - \text{largeur}$ et le voisin inférieur $x + \text{largeur}$.

Pour le pavage hexagonal, on a du différencier le cas où la largeur est paire, et les cas où elle est impaire. La figure ci-dessous illustre la numérotation du cas de largeur impaire sur un pavage de taille 7×3 . Avec celle-ci, les numéros des voisins sont donnés par la règle présente sur la figure.

Grâce à ces relations numériques, la création des pavages peut s'effectuer en 4 étapes :



$$\begin{array}{c}
 x - \frac{\text{largeur}}{2} - 1 & x - \frac{\text{largeur}}{2} \\
 \text{---} & \text{---} \\
 x + \frac{\text{largeur}}{2} & x + \frac{\text{largeur}}{2} + 1
 \end{array}$$

x x

FIGURE 3 – Numérotation des pavages hexagonaux à largeur impaire

- Allocation du tableau de tuiles de taille $\text{largeur} \times \text{hauteur}$.
- Pour chaque tuile : initialisation du nombre de poisson aléatoirement grâce à la fonction `random_fishes()` (qui retourne un entier entre 1 et 3 aléatoirement suivant des proportions définies), initialisation du pingouin présent à `EMPTY`, du nombre de voisins à la valeur correcte. Allocation du tableau des voisins et initialisation par calcul grâce aux relations numériques.
- Découpage de la banquise : aux extrémités, certaines tuiles ont à ce stade certains voisins qu'elles ne devraient pas avoir. Sur ces tuiles extrêmes, identifiées par des calculs sur les numéros, mise de certains voisins à `NO_TILE`.

2.3.2 Initialisation des stratégies

Cette phase est prise en charge par la fonction `strategies_init()`. Les fonctions `init()` des stratégies ont besoin d'un certain nombre d'informations décrivant les conditions de jeu de manière unique. Si certaines sont accessibles directement en lisant la structure `server` globale, d'autres doivent être générées au préalable. C'est le cas d'un tableau de structures `edge`, contenant toutes les arêtes du graphe, ainsi qu'un tableau d'entiers indiquant le nombre de poissons présents sur chaque tuile. Ces 2 tableaux sont alloués et initialisés par la fonction `create_init_data()`. Après avoir appelé la fonction d'initialisation de chaque stratégie, les deux tableaux sont libérés par la fonction `free_init_data()`.

La fonction `create_init_data()` effectue un simple parcours des tuiles par ordre d'indice. La taille du tableau d'arêtes, égale à la somme des nombres de voisins de toutes les tuiles, est difficilement déterminable avant le parcours pour les pavages contenant différents types de polygones. C'est pourquoi ce tableau est ré-alloué au fur et à mesure du parcours.

Grâce aux deux premières fonctions principales, le serveur joue son rôle d'hôte des clients, et d'initiateur d'une partie. Pour faire un parallèle avec le jeu de plateau, l'appel à ces deux fonctions permet de disposer les tuiles et pingouins correctement pour former la banquise, de tracer un tableau des scores vierges, et de réunir les joueurs autour du jeu. Les deux fonctions suivantes vont alors simuler la partie à proprement parler.

2.3.3 Tours de placement

Cette phase est prise en charge par la fonction `place_penguins()`. Celle-ci est appelée autant de fois qu'il y a de placements de pingouins. Pour simplifier le code principal (dans `main.c`), nous avons fait le choix d'utiliser deux variables statiques pour tenir le compte des tours et du numéro du joueur dont c'est le tour. Tant que les tours de placements ne sont pas terminés :

- La fonction récupère le placement du joueur dont c'est le tour via sa fonction `place()`.
- Puis, elle appelle la fonction `check_n_place()` pour vérifier sa validité et le répercuter sur la représentation du jeu du serveur.
- Puis, elle met à jour un tableau de structure `move` représentant les derniers coups, nécessaire à l'appel des fonctions `place()`. Cette mise à jour est effectuée par la fonction `update_previous_moves()`.
- Enfin, elle passe au joueur suivant en mettant à jour le numéro du joueur dont c'est le tour.

Validation des coups

Une fonctionnalité fondamentale du serveur est la vérification des coups proposés par les joueurs. La fonction `check_n_place()` prend en argument un numéro de joueur, la structure `move` contenant le placement proposé par ce joueur, le numéro du tour, et vérifie la validité du placement. Pour cela elle vérifie que :

- la tuile de destination existe, c'est à dire que son numéro est compris entre 0 (compris) et le nombre de tuiles (non compris) ;
- la tuile de destination n'a pas été retirée, et est vide ;
- le numéro de joueur indiqué dans la structure `move` est bien celui du joueur dont c'est le tour
- le nombre de poissons sur la tuile de destination est bien égal à 1, comme le stipule les règles du jeu.

Ensuite, si toutes ces conditions sont remplies, elle répercute le coup sur la structure `server` globale. Ceci implique une mise à jour du tableau de positions des pingouins et du champ `pinguin` de la tuile destination.

Mise à jour du tableau des coups précédents

Initialement le tableau des coups précédents est vide. Il est cependant alloué directement avec une taille égale au nombre de joueurs car un entier est passé aux fonctions de placement des stratégies pour indiquer sa taille. Durant les premiers tours, sa taille est incrémentée à chaque placement, jusqu'à atteindre le nombre de joueurs. Puis il conserve sa taille tant qu'aucun joueur n'a fini de jouer, c'est à dire au moins jusqu'à la fin des tours de placements. En effet, lors de la vérification des arguments passés par l'utilisateur, le serveur vérifie que le plateau est suffisamment grand pour que les joueurs puissent placer tous leurs pingouins.

À chaque placement, la fonction `update_previous_moves()` prend un pointeur sur le tableau des coups précédents, le dernier placement effectué, et met à jour son contenu. Pendant le premier placement de chaque joueur, cela consiste simplement à mettre le placement dans la case ayant le numéro du joueur. Après ce premier tour de placements, à chaque coup, les placements déjà présents dans le tableau sont décalés d'une place (éliminant donc le plus ancien) et le nouveau placement est ajouté dans la case ayant pour numéro le nombre de joueurs encore en jeu moins 1. À ce stade cette case est toujours la dernière du tableau. Le serveur s'assure ainsi que le contenu

du tableau des placements précédents est bien celui indiqué dans l'interface : tous les coups joués depuis le dernier placement du joueur, celui-ci inclus.

2.3.4 Tours de jeu

Cette phase est prise en charge par la fonction `play_game()`. Celle-ci est appelée tant qu'il reste des déplacements possibles. Comme pour les tours de placements, plusieurs variables sont statiques de façon à ce que la fonction puisse en quelque sorte reprendre la où elle en était d'un appel à l'autre. Une de ces variables est un tableau des joueurs toujours en jeu. A chaque tour :

- `play_game()` vérifie que le joueur fasse partie des joueurs encore en jeu. Sinon, elle passe au suivant.
- Puis elle appelle la fonction `is_blocked()` avec en paramètre le numéro du joueur dont c'est le tour. Celle-ci retourne un booléen indiquant s'il reste au moins un coup valide à ce joueur ou pas.
- S'il lui reste un coup valide, le fonctionnement est similaire à celui de `place_penguins()`. La fonction de jeu de la stratégie est appelée. La validité du coup qu'elle retourne est vérifié par `check_n_play()` et le cas échéant, celui-ci est répercuté sur la structure `server` globale. Le tableau des coups précédents est mis à jour via `update_previous_moves()` et la fonction passe au joueur suivant.
- S'il ne lui reste aucun coup valide, `play_game()` passe à `update_previous_moves()` un coup dont les tuiles sources et destination sont `NO_TILE`, puis retire le joueur du tableau des joueurs encore en jeu.

Vérification de la possibilité de jouer d'un joueur

Pour déterminer si un joueur a encore des coups valides possibles, la fonction `is_blocked()` parcourt, pour chaque pingouin du joueur, les tuiles voisines de celle sur laquelle se situe le pingouin. Dès qu'une tuile vide est trouvée, la fonction renvoie faux : le joueur peut déplacer le pingouin en question sur cette tuile. Si à la fin du parcours, aucune tuile vide n'a été trouvée, elle renvoie vrai : tous les pingouins du joueur sont isolés, et donc celui-ci a fini de jouer.

Validation des coups 2

Tous comme avec les placements, le serveur doit impérativement vérifier la validité des coups de déplacement retournés par les fonctions `play()` des stratégies. La fonction `check_n_play()` prend en argument un numéro de joueur, la structure `move` contenant le placement proposé par ce joueur, et vérifie la validité du placement. Pour cela elle vérifie que :

- la tuile source existe, et contient un pingouin du joueur ;
- la tuile de destination existe, n'a pas été retirée, et est vide ;
- le numéro de joueur indiqué dans la structure `move` est bien celui du joueur dont c'est le tour
- la tuile destination est accessible depuis la tuile source. La fonction `is_reachable()` se charge de cette vérification.

Ensuite, si toutes ces conditions sont remplies, elle répercute le coup sur la structure `server` globale. Ceci implique une mise à jour du tableau de positions des pingouins, du champ pingouin des tuiles sources et destinations. Pour finir `check_n_play()` déconnecte la tuile source du graphe avec la fonction `disconnect_tile()`, et met à jour le score du joueur.

La fonction `is_reachable()` prend en arguments les numéros d'une tuile source et d'une tuile destination. Depuis la tuile source, dans chaque direction, la fonction parcourt le graphe en ligne droite jusqu'à tomber sur un obstacle (tuile retirée ou tuile non vide). Si durant ce parcours elle passe sur la tuile destination elle renvoie vraie : la tuile destination est accessible en un coup depuis la tuile source. Sinon, elle renvoie faux car ce n'est pas le cas.

La fonction `disconnect_tile()` prend en argument le numéro d'une tuile à déconnecter de ses voisins dans le graphe. Pour cela, pour chaque tuile adjacente à la tuile à déconnecter, elle retire cette dernière du tableau des voisins. Ensuite, elle retire tous les voisins du tableau des voisins de la tuile à déconnecter.

Mise à jour du tableau des coups précédents 2

Une subtilité de la fonction `update_previous_moves()` permet de l'utiliser aussi pour mettre à jour le tableau des derniers déplacements de pingouins. Comme expliqué plus haut, celle-ci prend entre autres arguments le nombre de joueurs encore en jeu, et place le nouveau mouvement dans la case du tableau dont le numéro est le nombre de joueurs encore en jeu moins 1. Ainsi au fur que des joueurs sont "éliminés", la partie du tableau réellement utilisée rétrécit. Lorsqu'un joueur n'a plus aucun coup valide, `play_game()` appelle `update_previous_moves()` avec un coup portant le numéro du joueur et dont les tuiles sources et destination sont `NO_TILE`. Ce coup est placé dans la dernière case encore utilisée, puis celle-ci devient inutilisée car la taille du tableau est décrémentée. Ainsi on pourrait éventuellement lire le tableau de la fin vers le début à la fin de la partie pour donner l'ordre d'élimination des joueurs.

Ces deux fonctions permettent donc de répondre à un des problèmes soulevés par le sujet, à savoir, réussir à faire jouer les stratégies entre elles, à communiquer les coups de l'une aux autres, et inversement. Elles permettent aussi d'arbitrer la partie, puisqu'elles vérifient la validité des coups et tiennent les scores à jour.

2.3.5 Clôture du jeu

Cette ultime phase est prise en charge par la fonction `end_game()`. Elle doit d'une part annoncer les scores finaux, et d'autre part libérer correctement toutes les ressources allouées pour simuler une partie. Pour cela :

- La fonction retire chaque pingouin de la tuile sur laquelle il est resté coincé (de la même manière que lors d'un déplacement), et ajoute les poissons de la tuile au score du joueur auquel appartient le pingouin.
- Elle affiche les scores finaux, et utilise un algorithme classique de recherche de maximum dans le tableau des scores pour annoncer le vainqueur. Les égalités sont prises en charge.
- Elle clôture la partie pour les stratégies grâce à la fonction `strategies_done()`, qui appelle la fonction `done()` de chaque stratégie. Ces fonctions permettent notamment aux stratégies de libérer les ressources mémoires qu'elles utilisaient.
- Enfin, elle libère la mémoire utilisée par les structures de données du serveur, grâce à la fonction `free_srv()`, qui appelle elle-même en cascade les fonctions `free_graph()`, `free_strats()` et `free_scores()`.

3 Stratégies

3.1 Contexte des stratégies

Il est expliqué en section 1 que le plateau est représentée de la même façon pour les stratégies que pour le serveur. En revanche, les deux sont traités de façon indépendante dans le code, et aucun de ces deux parties ne peut accéder à ou modifier le graphe de l'autre.

Ainsi le choix a été fait pour les stratégies de reprendre la structure de graphe indiquée par la fonction `init()`. De plus, la plupart des fonctions utiles à une stratégie se sont révélées utiles à toutes les autres. Pour ces raisons, il a été choisi de mettre en commun ces fonctions ainsi que les structures de représentation du jeu dans des fichiers `strat.h` et `graph.h`.

Cela a en outre permis d'avoir une implémentation des stratégies qui se réduit au strict nécessaire *i.e.* aux fonctions `place()` et `play()`.

3.2 Stratégie aléatoire

Dans un premier temps, pour tester le serveur, nous avons développé une simple stratégie aléatoire, se déplaçant de voisin en voisin. Pour cela il a tout de même fallu gérer les changements communiqués par le serveur à chaque tour afin d'être sûr de retourner des coups valides.

Les spécifications des fonctions de jeu garantissant l'existence d'un coup à retourner, cette stratégie procède en tirant au hasard un pingouin parmi ceux du joueur, puis une tuile parmi les voisins de ce pingouin. Si cette tuile est libre, elle retourne le mouvement correspondant. Dans le cas contraire, elle recommence au tirage du pingouin. Ceci n'est donc pas optimisé pour les situations avec beaucoup de pingouins dont la plupart sont bloqués.

Une première évolution de cette stratégie a été de rajouter, une fois le voisin choisi, un tirage aléatoire de la longueur de la ligne droite sur laquelle se déplacer, en ayant calculé en amont la longueur maximale valide dans cette direction.

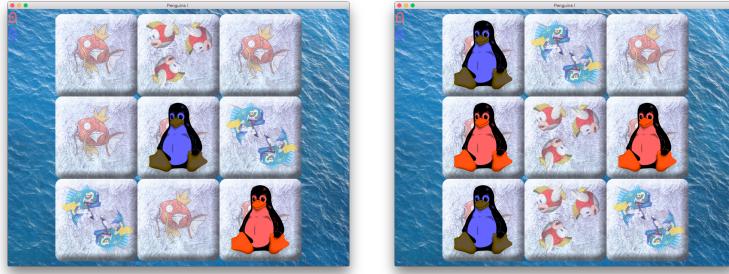
Pour se déplacer en ligne droite, on utilise la position des voisins dans le tableau listant ceux de chaque tuile. En arrivant de la tuile numérotée $t_{origine}$ sur la tuile $t_{courant}$, on recherche l'indice i tel que dans le tableau des voisins de $t_{courant}$ (noté ici v) on ait $v[i] = t_{origine}$. La tuile suivante dans la ligne droite est alors $v[(i + n/2)\%n]$ où n est le nombre de voisins de la tuile $t_{courant}$. On remarque ici l'intérêt que le nombre de voisins soit pair dans la définition d'une ligne droite.

3.3 Stratégie à un coup

Une heuristique de base consiste à privilégier les tuiles avec le plus grand nombre de poissons. La stratégie `strat_line` est donc une évolution de la stratégie aléatoire en ce sens. Elle liste, parmi les tuiles accessibles en un coup, celles avec le plus grand nombre de poissons possible. Elle tire ensuite aléatoirement la tuile de destination parmi celles-ci.

Le placement initial de cette stratégie a été modifié pour que le premier coup de la partie soit le plus avantageux possible. Les règles imposant de placer les pingouins sur des tuiles à un seul poisson, la stratégie sélectionne de telles tuiles voisines de tuiles avec le plus grand nombre de poissons possible.

On remarque bien, en figure 4, que la `strat_line` place ses pingouins sur les tuiles voisines de trois poissons, et à défaut celles voisines de deux poissons. Cela permet de jouer un second coup directement avantageux.

FIGURE 4 – Placement de `strat_rand` et de `strat_line`

3.4 Stratégie évoluée

Un outil d'algorithmique de graphe : la coupe minimale

En jouant de nombreuses parties du jeu de plateau dont s'inspire ce projet, on a pu se rendre compte qu'une autre stratégie consiste non pas à prendre les quelques tuiles contenant un grand nombre de poissons, mais plutôt à s'isoler sur une grande partie de la banquise pour pouvoir la vider de tous ses poissons. Pour cela, il est nécessaire de savoir repérer lorsqu'une scission du graphe est en train de se produire.

Le module `ford_fulkerson` fournit un outil pour repérer de telles situations. Il définit la structure `network`, similaire à la structure `graph`, mais enrichie de façon à pouvoir représenter un réseau (un graphe orienté avec une source, une destination, et des capacités sur chaque arc). Il définit également une structure `cut`, qui permet de représenter une coupe, c'est à dire un ensemble d'arcs sans lesquels le graphe ne serait plus connexe. La fonction `min_cut()` retourne une coupe minimale, c'est à dire une coupe dont le cardinal des arrêtes est minimal. Pour cela, elle commence par déterminer un flot maximal du réseau grâce à l'algorithme de Ford-Fulkerson¹. Ensuite, elle utilise un algorithme de remplissage par diffusion² pour déterminer la composante connexe de la source dans le graphe résiduel. Les arcs dont une des extrémités appartient à cette composante mais pas l'autre forment alors la coupe minimale.

Un autre outil : la recherche de chemin Hamiltonien

Une fois isolé sur une partie de graphe, un pingouin n'a plus de concurrence pour récupérer les poissons à sa portée. Il convient alors de planifier un chemin permettant de récupérer les plus de poissons possibles. L'idéal serait de déterminer un chemin Hamiltonien sur cette partie de graphe. Malheureusement, il n'est pas toujours possible de parcourir l'ensemble des tuiles de la partie.

La fonction `connex_size()` de la stratégie `strat_eat` calcule le nombre de tuiles de la composante connexe par un parcours linéaire en nombre de tuiles. Ce nombre de tuiles sert de borne supérieure pour la taille des chemins *pseudo*-Hamiltonien que l'on recherche par la suite.

Cette recherche, effectuée par la fonction `eat_them_all()`, procède en parcourant le graphe en profondeur, en gardant en mémoire le chemin parcouru et le meilleur chemin trouvé jusqu'à lors. Lorsqu'elle trouve un chemin plus long que le meilleur, elle le remplace. Quand ce meilleur

1. plus d'informations sur cet algorithme sur la page wikipédia correspondante

2. plus d'informations sur cet algorithme sur la page wikipédia correspondante

chemin atteint la longueur maximale calculée par `connex_size()`, la recherche stoppe et le chemin est utilisé pour jouer.

Cet algorithme énumère donc les chemins possibles pour trouver le meilleur. Il a une complexité exponentielle, et est impraticable sur les graphes à plus de 10 tuiles. Pour le rendre utilisable, nous avons essayé plusieurs heuristiques :

- Remplacer systématiquement la borne supérieure de la taille pour la recherche par un entier de l'ordre de 10. L'algorithme termine alors rapidement, mais les chemins obtenus ne permettent pratiquement jamais de se rapprocher d'un chemin Hamiltonien. A la place, les chemins isolent le pingouins sur une partie du graphe encore plus petite.
- Considérer un chemin comme satisfaisant s'il couvre une certaine portion des tuiles de la partie de graphe (90% par exemple). Cela fait terminer la recherche plus vite, mais n'est pas non plus exploitable sur de trop grands graphes.
- Ajout d'une limite en temps de calcul pour l'exploration. Lorsque le temps prévu est révolu, on utilise le meilleur chemin trouvé. Celui-ci n'est pas forcément optimal, mais il se montre plutôt efficace sur des parties de graphe comprenant jusqu'à une centaine de tuiles.

D'autres variations auraient pu être :

- Maximiser le nombre de poissons accumulés à la place du nombre de tuiles parcourues. Le maximum théorique à calculer devient ici le nombre de poissons présents sur la partie de graphe. Cette heuristique est plus proche du but du jeu de Penguins, mais nous avons manqué de temps pour l'implémenter.
- Remplacer l'heuristique déterministe de l'exploration exhaustive par une heuristique probabiliste de type Monte-Carlo en recherchant aléatoirement des chemins en temps limité. Il faudrait *a priori* plus de temps pour trouver un chemin optimal si la partie de graphe est grande, mais il est sûrement possible de s'en rapprocher assez vite.

Dissekator

Afin de déterminer un ensemble de noeuds de cardinal minimal à retirer pour séparer deux sommets du graphe, la stratégie dissekator utilise la fonction `v_min_cut()`. Celle-ci se sert du module `ford_fulkerson`. Elle effectue d'abord une transformation permettant de transformer ce problème en une recherche de coupe minimale dans un réseau, grâce à la fonction `graph_to_network()`, qui prend en argument un graphe et retourne un pointeur vers un réseau sur lequel appliquer la fonction `min_cut()`. Le détail de cette transformation est documenté de façon détaillée dans les commentaires du code. Puis elle traduit la coupe minimale retournée par la fonction `min_cut()` en l'ensemble de sommet recherché grâce à la fonction `cut_to_v_cut()`.

Cette fonctionnalité permettant de lire en avance les coupes du graphe, une stratégie a été imaginée pour en tirer partie. Celle-ci consiste à se placer, juste avec la coupe définitive d'un bloc du graphe, sur le point de rupture, pour bloquer l'adversaire et profiter des poissons en fin de partie via un chemin hamiltonien.

La recherche des coupes a été la première étape à l'implémentation de cette stratégie. Seulement, elle nécessite le choix de deux tuiles (tuiles que la coupe sépare) ; il a donc fallu trouver un moyen de choisir ces deux tuiles. Le choix a été fait de prendre les tuiles sur lesquels se trouvent un pingouin allié et un pingouin ennemi. Si leur séparation est imminente et que le pingouin allié peut atteindre la tuile de rupture avant celle-ci, ce couple de pingouin est retenu. En revanche, la question du plus court chemin du pingouin à la tuile de rupture se pose. L'implémentation d'un algorithme type Dijkstra a été discutée, mais elle n'a pas aboutie par manque de temps.

4 Interface graphique

L'interface graphique a été réalisée à l'aide de la bibliothèque `SDL2`³.

4.1 Généralités

La principale motivation ayant encouragé la création d'une interface graphique est la facilité de lecture de jeu qu'elle procure, par rapport à une sortie texte en console. L'interface permet donc de voir ce qu'il se passe et comment réagissent les stratégies, ce qui facilite le travail des développeurs de stratégies, tout en ayant un attrait visuel.

Le code de l'interface graphique est conçu pour être plus ou moins indépendant vis à vis du reste du code. En effet, l'ensemble des fonctions relatives à l'interface sont contenues dans le module `display`, et ces fonctions ne sont utilisées que dans `main.c`. Ainsi, il est possible de désactiver l'interface simplement en n'exécutant pas quelques lignes, ce qui est fait dans notre cas au niveau du pré-processeur grâce à un `#ifndef`, et d'une règle dans le `Makefile`

4.2 Organisation du code

4.2.1 Structures de données

Le code de l'interface définit plusieurs structures de données :

- `gameGraphics` contient les différents sprites, le fond d'écran, la définition des couleurs utilisée par les joueurs, ainsi que l'ensemble des tuiles et des pingouins, contenues dans des `graphicsElement`
- `graphicElement` contient les informations sur les éléments graphiques, c'est-à-dire les coordonnées, les dimensions, le sprite et la couleur utilisé par l'élément.

On dispose également de variables globales contenant la fenêtre, le renderer ou la police d'écritures, toutes fournis par la `SDL`.

4.2.2 Fonctionnement

Le code fonctionne grâce à quelques fonctions principales :

- `display_init` et `graphics_init`, qui initialisent respectivement la `SDL` et les variables globales associées, et les structures de données décrites précédemment.
- `display_free` et `graphics_free`, qui libèrent la mémoire allouée par leurs homologues d'initialisation.
- `handleEvents` gère toutes les interactions utilisateurs. Notre interface servant juste à visualiser une partie jouées par des programmes, les interactions se limitent à des actions basiques telles que fermer la fenêtre, la redimensionner, et gérer le déroulement de la partie (passer au coup suivant, jouer toute la partie d'un coup...)
- `synchronize` permet d'actualiser les données de la vue avec celle contenu dans le serveur, c'est-à-dire la position des pingouins, quelle tuiles ont été supprimées...
- `draw` dessine les sprites dans la fenêtre stockée en mémoire.
- `updateScreen` actualise l'écran réel avec la représentation mémoire de la fenêtre.

L'intérêt d'une telle séparation est qu'elle permet de bien délimiter les différentes opérations, mais également de garder au maximum la séparation entre le serveur et l'interface, en centralisant toutes les interactions entre les deux dans la fonction `synchronize`

3. <https://www.libsdl.org/>

5 Tests

Dans un soucis de robustesse, les structures utilisées par nos programmes sont manipulées à l'aide de fonctions testées et comprenant des vérifications de conformité des arguments à l'aide de la bibliothèque `assert`. Ces tests sont exécutables depuis le makefile via une règle spécifique.

Pour les fonctions du serveur, nous avons choisi d'effectuer rigoureusement des tests unitaires, de façon à s'assurer séparément du bon fonctionnement de chacune d'entre elle. Ces tests peuvent être lancés via une règle spécifique du Makefile également (cf le README du projet). Le lancement de nombreuses parties avec les stratégies aléatoires a ensuite en quelque sorte constitué un test d'intégration.

Enfin, le serveur a été testé à l'aide de l'outil Valgrind, de façon à certifier qu'avec le mode de compilation `nographics` (qui n'utilise pas la librairie SDL), il n'y a aucune fuite mémoire ou accès incorrect, autres que ceux éventuellement effectués par les stratégies.

Conclusion

Ce projet a été l'occasion de mettre à la fois en pratique des techniques de programmation avancées, avec les chargements dynamique de bibliothèques par exemple, mais aussi de travailler l'algorithmique théorique, surtout l'algorithmique de graphes, afin de modéliser les problèmes correctement. De plus, ce projet nous a également permis de travailler dans des équipes plus nombreuses que lors du premier semestre, ce qui a mis à l'épreuve nos capacités à se coordonner, et à des outils de travail en groupe tels que subversion.

A RÈGLES DU JEU

A.1 Le plateau

Le jeu *Penguins* se joue sur un plateau à deux dimensions, composés de tuiles symbolisant des morceaux de banquise. Ces tuiles contiennent chacune un certain nombre de poissons (allant de 1 à 3). Aucune tuile n'est vide. De plus, la moitié au moins des tuiles doit ne contenir qu'un seul poisson.

Cette banquise forme un pavage régulier du plan. Dans cette optique, plusieurs pavages sont envisageables (cf 2.3.1). Le plan ainsi constitué forme un rectangle (pas nécessairement aussi haut que large).

A.2 Les joueurs

Les joueurs sont a priori en nombre illimité (tant qu'il reste des tuiles valides pour les pingouins). Ils possèdent chacun un ou plusieurs pingouins qu'ils peuvent placer puis faire bouger sur le plateau.

Leur nombre de pingouins est aussi, a priori, illimité, tant que le nombre total de pingouins n'excède pas le nombre de tuiles à 1 poisson. Il est fixé au début de la partie.

A.3 La partie

A.3.1 Le début de la partie

Au début, chaque joueur pose tour à tour ses pingouins, un par un. Il doit impérativement les poser sur des tuiles contenant seulement 1 poisson. Deux pingouins différents ne peuvent être placés sur la même tuile.

A.3.2 Le déroulement de la partie

Tour après tour, les joueurs doivent déplacer l'un de leur pingouin sur une autre case ; un seul mouvement par joueur est permis, et un seul mouvement pour ce pingouin. Un joueur ne peut pas passer son tour.

Ce déplacement obéit à des lois très précises : un pingouin ne se déplace qu'en ligne droite, c'est à dire d'un côté d'un polygone au côté strictement opposé (raison pour laquelle les polygones sont choisis réguliers et avec un nombre pair de côtés). En revanche, la longueur du déplacement importe peut tant que le pingouin ne traverse pas l'eau (les tuiles doivent être connexes), qu'il ne traverse pas une tuile occupée par un pingouin et qu'il ne sorte pas du plateau.

Lorsqu'un pingouin quitte une tuile, il ajoute au score du joueur auquel il appartient le nombre de poissons qui s'y trouvaient, et fait fondre la tuile, qui devient inaccessible (on ne peut plus ni y aller, ni la traverser).

Si un pingouin en vient à s'isoler complètement (plus de tuile connexe), il ne peut plus être déplacé par le joueur.

A.3.3 La fin de la partie

Si tous les pingouins d'un joueur en viennent à s'isoler, il est exclu de la partie.

La partie se termine quand tous les joueurs sont exclus (le dernier joueur pourra donc jouer tout seul). Ce moment atteint, les pingouins sont retirés en ajoutant au score les poissons de la tuile sur laquelle ils étaient immobilisés jusque là. Le joueur avec le plus de poissons gagne.