ENSEIRB-MATMECA

Département Informatique – Semestre 6

Projet de Programmation Impérative

Coursidor en langage C

Encadrant :
M. Frédéric HERBRETEAU

Réalisé par :

— Bel Hajjam Yahya
— El Yaktini Sohayb
— Ajaji Mohamed
— Abarrah Anas

Année universitaire 2024–2025

Table des matières

1. Introduction

1.1. Présentation du jeu

Le Coursidor est une variante du célèbre jeu Quoridor, dans laquelle deux joueurs s'affrontent sur un plateau triangulaire, cyclique ou troué. Chaque joueur déplace son pion à travers un certain nombre de « bases » (objectifs) disposées de manière symétrique, puis doit revenir à sa case de départ. À chaque tour, il peut soit avancer son pion selon les règles de mouvement étendues (sauts, déplacements prolongés), soit placer un mur pour ralentir l'adversaire sans jamais le bloquer complètement.

1.2. Objectifs du projet

Le but de ce projet est double :

— Développer au moins deux clients autonomes (`player1`, `player2`) capables de choisir automatiquement leurs coups en fonction d'une stratégie donnée.

— Mettre en place un serveur générique pouvant charger dynamiquement ces clients, arbitrer les échanges (déplacements, pose de murs), et garantir la cohérence du plateau jusqu'à la fin de la partie.

1.3. Contraintes

La conception doit respecter plusieurs contraintes impératives :

— **Interopérabilité** : les clients sont fournis sous forme de bibliothèques partagées et doivent fonctionner sans modification quel que soit le serveur.

— **Performance** : chaque décision (BFS, A, génération de coups) doit s'exécuter en un temps raisonnable pour ne pas dépasser les limites imposées par le serveur.

— **Robustesse** : toute erreur de coup (mauvais format, atteinte d'une même case par les deux pions, mur illégal) entraîne la défaite immédiate du client fautif.

2. Problèmes et solutions

2.1. Modélisation des graphes du jeu

Parmi les premiers problèmes rencontrés, celui de trouver une modélisation à la fois correcte et exploitable du graphe. Nous avons donc découpé cette tâche en deux sous-problèmes, en cherchant une représentation qui permette à la fois de :

— définir la forme du graphe et remplir correctement la matrice d'adjacence (implémentée avec la bibliothèque GSL), de manière uniforme pour les trois types de graphes (triangulaire, cyclique et troué) ;

— placer les objectifs en respectant la symétrie du plateau, afin d'assurer une distribution équitable et un équilibre stratégique entre les joueurs.

La première modélisation choisie repose sur un système de coordonnées hexagonales. Cette approche s'inspire des coordonnées cartésiennes, avec un repère centré sur l'origine et une première diagonale (équivalente à $x = y$) utilisée comme direction principale.

Chaque sommet est défini par deux entiers (q, r), et une troisième coordonnée implicite $s = -q - r$ permet de contrôler la validité des sommets.

La condition $\max(|q|, |r|, |s|) < m$ nous permet de générer un plateau hexagonal centré de rayon m. Un tableau fixe de directions encode les six voisins potentiels de chaque sommet. Ce tableau simplifie la génération de la matrice d'adjacence, et garantit une topologie cohérente pour tous les types de graphes.

Cette représentation nous a permis de construire un graphe structuré, visuellement clair, et facilement adaptable aux différents cas du projet. La Figure 1 illustre cette structure.

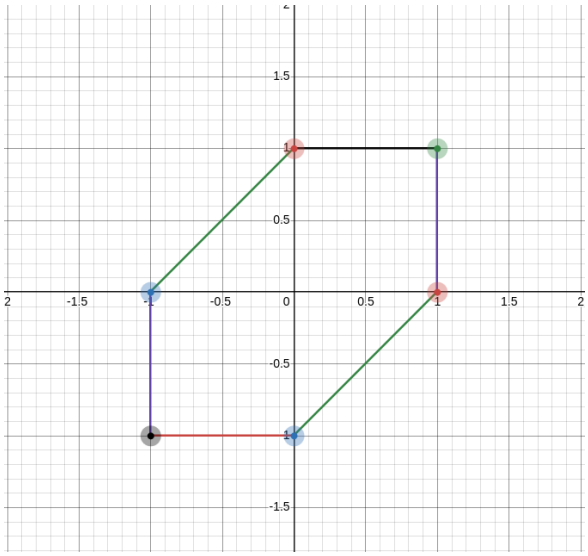

FIGURE 1 – Modélisation du graphe hexagonal centré

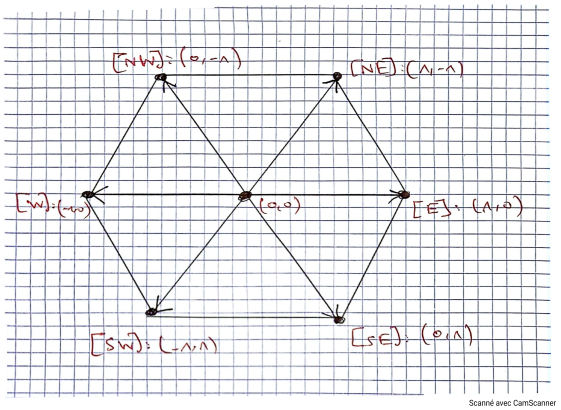

FIGURE 2 – Cellule hexagonale unité dans la grille

Implémentation technique

Au niveau du code, la modélisation hexagonale repose sur un tableau de vecteurs de déplacement simple et efficace :

```c
typedef struct {
    int dq;  /* d calage en q */
    int dr;  /* d calage en r */
} hex_dir_t;

/* Six directions autour d'un hexagone : NW, NE, E, SE, SW, W */
static const hex_dir_t hex_dirs[NUM_DIRS] = {
    [NW] = { 0, -1},
    [NE] = { 1, -1},
    [E ] = { 1,  0},
    [SE] = { 0,  1},
    [SW] = {-1,  1},
    [W ] = {-1,  0},
};
```

Listing 1 – Extrait de `player_info` dans `commun.h`

Les six directions sont appliquées à chaque sommet (q, r) pour identifier les voisins. Si la cellule cible respecte la condition $\max(|q|, |r|, |s|) < m$, l'arête est ajoutée dans la matrice d'adjacence creuse, encodée avec `gsl_spmatrix`.

Cependant, cette modélisation ne suffit pas à elle seule pour interagir efficacement avec les autres modules du projet. Nous avons donc introduit une deuxième abstraction, complémentaire et plus adaptée au traitement algorithmique.

Numérotation linéaire des sommets

Pour faciliter l'accès aux sommets dans les tableaux et structures internes, chaque sommet valide est associé à un identifiant entier unique, de 0 à $n-1$. Cette numérotation permet un adressage direct, une compatibilité totale avec la GSL, et une simplification des fonctions de manipulation (parcours, distances, objectifs...).

Deux fonctions assurent la correspondance entre les deux représentations :

— `coord_to_id(q, r)` : transforme des coordonnées hexagonales en identifiant linéaire ;

— `id_to_coord(id)` : retourne les coordonnées (q, r) associées à un identifiant donné.

Cette seconde modélisation s'intègre naturellement avec la première, en conservant sa logique géométrique tout en rendant le code plus efficace et lisible. Elle a notamment simplifié la gestion des objectifs, des déplacements, et du placement des murs.

Elle a aussi permis de gérer correctement les positions de départ des joueurs et la distribution des objectifs, pour toute taille de graphe. Les identifiants étant continus, il devient trivial de répartir les objectifs de manière équilibrée (par exemple en prenant des sommets à distance équivalente du centre), sans avoir à manipuler les coordonnées manuellement.

Enfin, cette abstraction nous a permis d'écrire un code plus modulaire, plus robuste, et plus facile à tester. Elle s'est révélée particulièrement utile lors des tests sur différentes tailles de plateaux, ou lors du changement de type de graphe.

2.2. Modélisation et gestion des joueurs

Un autre défi central a été de concevoir une représentation claire et sûre de l'état de chaque joueur, tout en garantissant que le client puisse tester librement ses propres simulations sans corrompre le plateau de jeu global. Dans un premier temps, nous avions imaginé partager un unique graphe mutable entre le serveur et les clients, mais cette approche aboutissait rapidement à des accès concurrents invalides et à des segfaults lors de suppressions d'arêtes simultanées. Nous avons ensuite envisagé de dupliquer systématiquement l'intégralité de l'état (graphes, listes d'objectifs, compteurs de murs), mais cette solution se révélait trop verbeuse et difficile à maintenir.

Finalement, nous avons adopté une abstraction minimaliste : une structure `player_info` qui encapsule, pour chaque client, uniquement les informations nécessaires à la prise de décision. Chaque joueur conserve une copie isolée du graphe, sa position courante, le sens de son dernier déplacement, la liste des objectifs restant à atteindre et le nombre de murs dont il dispose. Grâce à cette conception, le client peut, avant de poser un mur ou de déplacer son pion, modifier librement son graphe local et évaluer les conséquences (distance restante, nombre d'objectifs capturés, validité du chemin) sans jamais risquer de corrompre le plateau maître.

```c
struct player_info {
    unsigned      id;             // identifiant du joueur (0=BLACK
        ,1=WHITE)
    enum color    color;          // couleur attribu e par le
        serveur
    struct graph_t *graph;        // copie du graphe pour
        simulations locales
    vertex_t      start, current;// position de d part et position
         courante
    enum dir_t    last_dir;       // direction du dernier move
    size_t        num_walls;      // murs encore disponibles
    size_t        num_objectives;
    vertex_t     *objectives;     // objectifs restants
};
```

Listing 2 – Extrait de `player_info` dans `commun.h`

Côté serveur, chaque appel à `initialize()` reçoit une copie indépendante du graphe (via `graph_copy`), ce qui isole totalement l'état de chaque client. Après chaque coup légal, le serveur met à jour son graphe central et reconstruit de nouvelles copies pour les deux joueurs. Cette organisation en deux niveaux, client pour la stratégie et le test local, serveur pour la validation et la synchronisation nous a permis d'obtenir un système à la fois robuste (absence de segfaults), clair (séparation nette des responsabilités) et facile à étendre.

Nous avons opté pour une structure player_info allégée, ne regroupant que l'essentiel : position actuelle, objectifs restants, nombre de murs et direction du dernier déplacement. Cette approche nous a permis d'éviter les complexités superflues et de prévenir les fuites mémoire, tout en rendant chaque étape—qu'il s'agisse du parcours en largeur (BFS), de l'heuristique A* ou de la génération des mouvements—plus facile à comprendre, à tester et à corriger. En limitant le périmètre de nos données, nous avons gagné en clarté et en fiabilité, ce qui s'est traduit par un joueur à la fois rapide et robuste face aux différents scénarios de partie.

La figure ci-dessus (Figure 3) illustre de manière synthétique le protocole d'échange entre le serveur et un client (joueur) :

- **initialize(id, graph)** : Le serveur appelle la fonction `initialize` en passant l'identifiant du joueur et un pointeur vers la structure du graphe. Le client en profite pour construire son état interne (position de départ, liste d'objectifs, nombre de murs, etc.).
- **play(prev_move)** : À chaque tour, le serveur invoque `play` en lui fournissant le dernier coup joué (type, pion ou mur). Le client analyse ce coup, met à jour sa copie locale du graphe (suppression d'arêtes en cas de mur adverse) puis calcule son propre coup.
- **move** : Le client renvoie un `struct move_t` que le serveur intègre dans la partie. Si c'est un mur, le serveur vérifie sa légalité avant de l'appliquer au graphe central.
- **finalize()** : À la fin de la partie (victoire, défaite ou égalité), le serveur appelle `finalize` pour libérer toutes les ressources allouées par le client (graphe, tableaux d'objectifs, etc.).

Cette séquence garantit que le serveur reste la seule source de vérité pour l'état global du plateau, tout en laissant au client la responsabilité de sa propre stratégie et de la mise à jour locale du même graphe partagé.

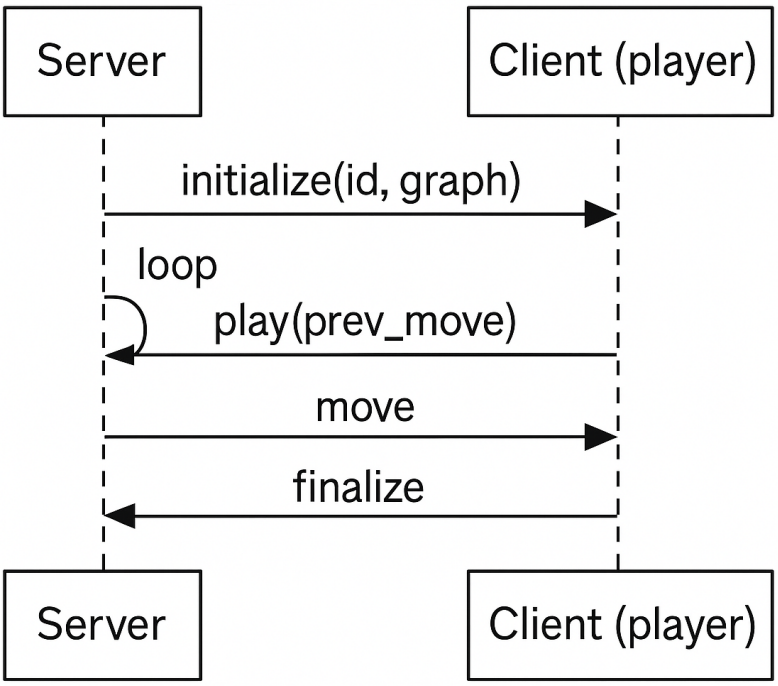

FIGURE 3 – Diagramme de séquence : échanges serveur–client

2.3. Gestion des parties

Le serveur doit être l'orchestre central de la partie, prenant en charge le chargement dynamique des deux modules clients, l'alternance stricte des tours, la conservation d'une unique source de vérité pour l'état du plateau, la validation infaillible de chaque coup (qu'il s'agisse d'un déplacement ou de la pose d'un mur), la prévention de toute tentative de coup illégal ou malicieux, ainsi que la détection automatique de la fin de la partie.Pour resoudre ce probleme on a fait la conception de la boucle principale du serveur qui repond à ces besoins.

Chargement dynamique des joueurs

On a besoin premièrement d'une intégration flexible et évolutive des algorithmes de joueurs, le serveur utilise `dlopen()` et `dlsym()` pour charger à l'exécution les deux bibliothèques clientes (`player1.so` et `player2.so`). pour avoir une structure comme dans la figure 4. Chacune expose les fonctions `initialize(id, graph)` pour préparer l'état local, `play(last_move)` pour calculer le prochain coup, et `finalize()` pour le nettoyage. Cette liaison dynamique garantit que l'ajout ou la mise à jour d'un joueur ne nécessite pas de recompiler le serveur, offrant ainsi modularité et facilité de maintenance.

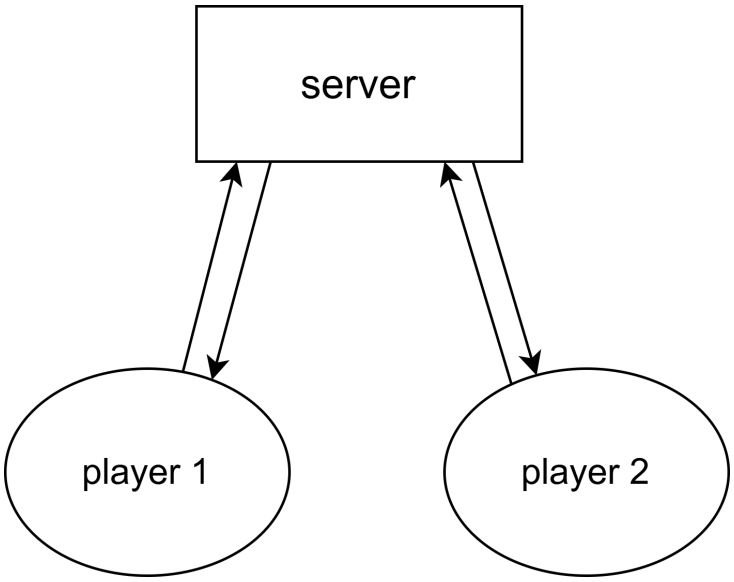

FIGURE 4 – échanges serveur–players

Validation des coups

Afin d'assurer une maîtrise totale de l'avancement du jeu et d'éviter tout blocage par un client, le serveur s'appuie sur une boucle infinie qui alterne strictement les tours. À chaque itération, il appelle `play()` du joueur courant en lui passant le dernier coup adverse, puis oriente le résultat vers la routine de validation et de mise à jour appropriée. Si un coup est invalide ou qu'un joueur ne peut plus atteindre ses objectifs, la boucle s'interrompt immédiatement et l'adversaire est déclaré vainqueur ; sinon, on passe au joueur suivant via `compute_next_player()` et on incrémente le compteur de tours.

Pour prévenir toute manipulation incorrecte du pion et garantir le respect des règles de déplacement, le serveur effectue systématiquement, via `board_move_token()`, la vérification de la case cible (libre et adjacente). En cas de succès, la position du pion est mise à jour et `board_update_objectives()` marque éventuellement un objectif comme atteint. Si la validation échoue — déplacement non valide ou case occupée — le coup est rejeté et la victoire est attribuée à l'autre joueur, assurant ainsi l'intégrité du plateau. la figure 6 montre un exemple du movement invalide

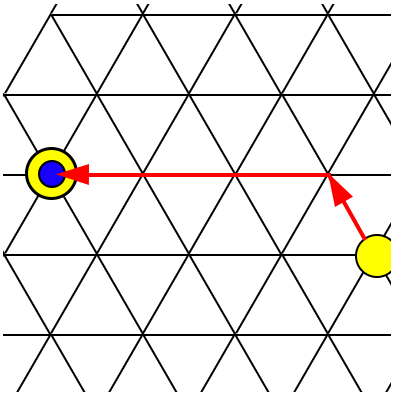

FIGURE 5 – exemple de mouvement invalide

Face au risque de bloquer totalement un adversaire par la pose d'un mur, le ser-

veur adopte une méthode de simulation sécurisée : il récupère d'abord le graphe maître (`board_get_server_graph()`), le copie (`graph_copy()`), puis y supprime les arêtes correspondant au mur proposé (`graph_remove_edge()`). Pour chaque joueur, il teste alors la connectivité de sa position à tous ses objectifs avec `has_path_to_objectives()`. Si les deux conservent un chemin valide, le mur est ajouté via `board_add_walls()` ; sinon, le coup est refusé et le poseur perd la partie, garantissant ainsi une équité totale.

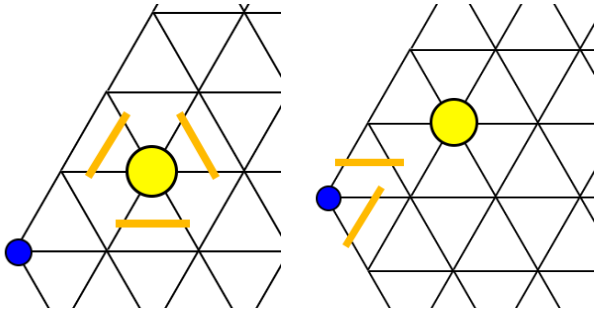

FIGURE 6 – exemple de coups invalides

3. architecture

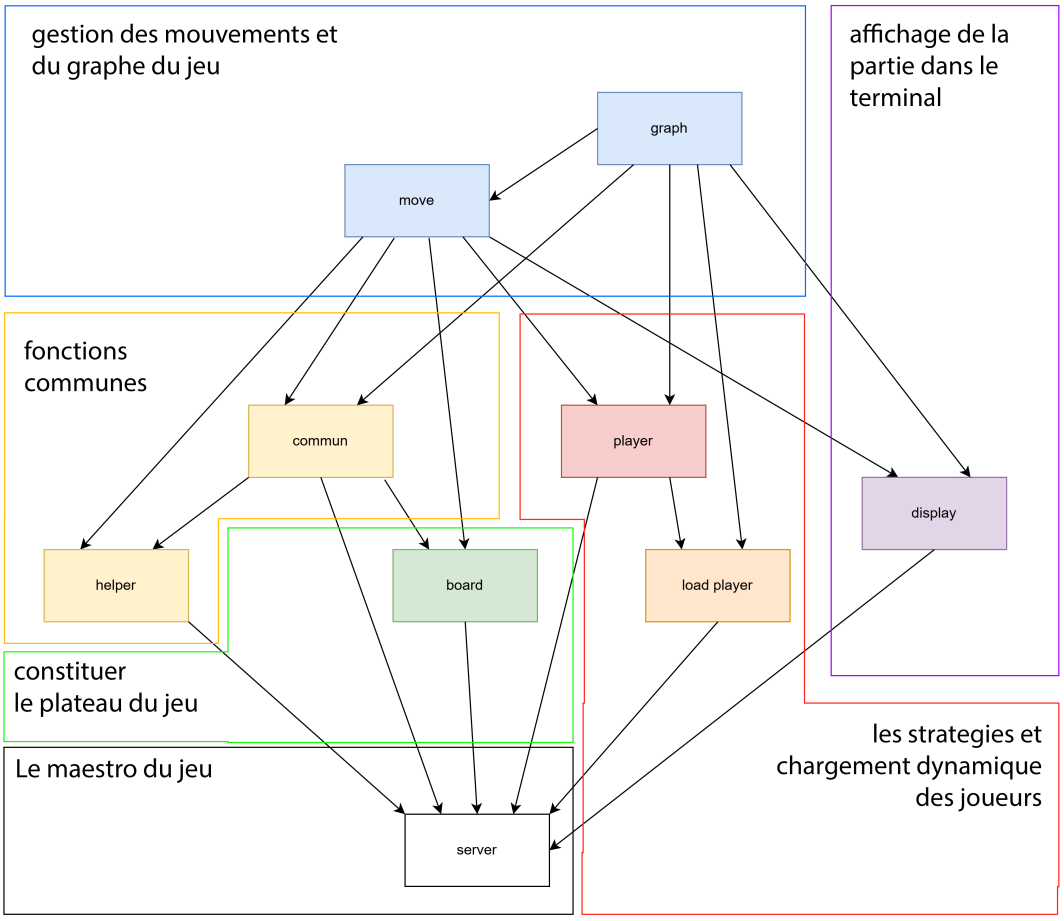

FIGURE 7 – Graphe de dépendances

Le projet repose sur une architecture modulaire organisée autour de composants clés, garantissant isolation, performance et extensibilité.

3.1. Modules principaux et responsabilités

Le serveur (server.c) orchestre la partie en chargeant dynamiquement les joueurs via dlopen, validant les coups via board, et affichant l'état du jeu avec display. Il s'appuie sur commun pour partager des structures standardisées (comme player_info), assurant la cohérence entre joueurs et règles métier.

Le plateau (board.h) gère l'état global : positions des joueurs, objectifs, et murs. Il maintient deux copies isolées du graphe (une par joueur), évitant les interférences lors des simulations locales. Des méthodes comme board_set_position ou board_add_walls permettent des mises à jour sécurisées, tandis que sa dépendance à graph assure une représentation cohérente des connexions.

Le graphe (graph.h) structure le jeu à l'aide d'une matrice creuse GSL, optimisée pour les grands plateaux. Il génère des topologies variées (triangulaire, cyclique) et gère les arêtes. Utilisé par board et les joueurs, il uniformise les opérations de base (ajout/-suppression d'arêtes).

L'interface des joueurs (player.h) définit les fonctions initialize, play, et finalize. Chaque joueur exploite une copie locale du graphe pour élaborer sa stratégie, validée ensuite par le serveur. Cette séparation permet des algorithmes indépendants tout en respectant les règles globales.

Le module commun (commun.h) standardise les structures partagées (ex. move_t) et les utilitaires (ex. has_path_to_objectives), réduisant la duplication de code. Il facilite l'échange d'informations critiques comme les objectifs ou l'état des joueurs.

Les algorithmes (helper.h) centralisent des fonctions complexes (A*, BFS, validation de murs), évitant leur réimplémentation. Cette approche améliore la maintenance et les performances, notamment pour les calculs de chemins ou la détection de coups légaux.

Enfin, loa_player (load_player.h) charge dynamiquement les joueurs compilés en .so. Ce mécanisme, basé sur dlopen, permet d'intégrer de nouvelles stratégies sans recompiler le serveur, favorisant l'expérimentation.

3.2. Justification des choix techniques

La modularité est renforcée par l'isolation des responsabilités : le serveur délègue la logique métier à board, tandis que les joueurs agissent sur des copies locales du graphe. Cette séparation prévient les conflits de données et simplifie les tests.

Le chargement dynamique (load_player.h) découple le serveur des stratégies, permettant des mises à jour indépendantes. Les joueurs deviennent des plugins interchangeables, améliorant l'extensibilité.

Les structures partagées (commun.h) et la standardisation des coups (move.h) assurent l'interopérabilité. Par exemple, player_info encapsule l'état d'un joueur de manière lisible et cohérente pour tous les modules.

Les performances sont optimisées via helper.h (algorithmes centralisés) et graph.h (matrices creuses GSL). Ces choix réduisent la redondance et améliorent l'efficacité, notamment sur les grands plateaux.

Cette architecture équilibre rigueur technique et flexibilité, offrant un système robuste, maintenable et adaptable à de nouvelles règles ou stratégies.

4. Stratégies

4.1. Stratégie A* (player1.c)

Notre premier client mise sur une évaluation fine des positions grâce à l'algorithme A* :

- **Évaluation heuristique** : pour chaque coup possible, on calcule `f = g + h` où
  - g est le coût réel (nombre de pas effectués)
  - h est l'heuristique hexagonale $\max(|\Delta q|, |\Delta r|, |\Delta s|)$
- **Comparaison des candidats** : on conserve le coup minimisant `f`, ce qui oriente le pion vers l'objectif tout en anticipant les obstacles (murs et adversaire).
- **Bonus multi-pas** : si le mouvement reste dans la même direction que le pas précédent, on peut avancer jusqu'à 3 cases en un seul coup, ce qui renforce la capacité à *enchaîner plusieurs objectifs alignés*.

Cette variante A* fournit un équilibre entre rapidité de calcul et qualité de trajectoire, notamment sur les grands plateaux.

4.2. Stratégie BFS (player2.c)

Notre second client adopte une méthode plus simple, fondée sur la recherche en largeur (BFS) :

- **Choix du prochain objectif** : on identifie le sommet objectif le plus proche par BFS, sans heuristique complexe.
- **Déplacement pas à pas** : pour chaque voisin direct, on évalue en combien de coups (BFS) il reste pour atteindre cet objectif, et on choisit le mouvement minimisant cette distance.
- **Gestion des sauts** : si l'adversaire est adjacent, on autorise le *saut par dessus* grâce à la fonction `jump_over`, puis on poursuit la BFS depuis la nouvelle position.

La simplicité de cette approche garantit une exécution très rapide, au prix d'une trajectoire parfois moins optimale qu'A*.

Pose de murs (commune aux deux clients)

Dans les deux cas, la dynamique de défense passe par la pose de murs :

1. *Détection* : on teste si l'adversaire est à moins de 3 coups de son prochain objectif (`is_opponent_near_last_objective`).
2. *Simulation* : on explore les paires d'arêtes adjacentes autour de l'adversaire, on simule leur suppression sur un graphe temporaire, et on mesure le gain Δ de distance par BFS (`bfs_distance`).
3. *Validation* : on ne retient que les murs pour lesquels $\Delta \geq 2$ et qui ne brisent pas la connectivité des deux joueurs (`wall_preserves_connectivity`).

Cette défense intelligente, validée par simulation, garantit un ralentissement notable de l'adversaire sans jamais le bloquer illégalement.

4.3. Adaptation dynamique

L'ensemble de ces choix est **réévalué dynamiquement à chaque tour**, en fonction :

— de l'évolution du graphe (murs posés),

— des distances aux objectifs,

— et de la position actuelle des deux joueurs.

Cette approche garantit une stratégie **réactive et adaptée** à la situation, ce qui renforce l'efficacité de notre joueur contre des adversaires plus passifs ou rigides.

5. Tests

Pour garantir la robustesse de notre implémentation des graphes et des algorithmes de déplacement, nous avons développé un ensemble de tests unitaires, regroupés dans l'exécutable `alltests`. Chaque fichier de test cible une fonctionnalité précise :

— `test_add_edge.c` – vérifie que l'ajout d'une arête dans la matrice creuse fonctionne correctement et que la direction est bien enregistrée.

— `test_remove_edge.c` – s'assure que la suppression d'une arête (pose de mur) retire effectivement la connexion dans les deux sens.

— `test_neighbors_vertex.c` – contrôle que `graph_get_neighbors` renvoie exactement les voisins valides, en tenant compte des murs.

— `test_exists_path.c` – teste l'algorithme de parcours (BFS) pour déterminer l'existence d'un chemin entre deux sommets, y compris après modifications du graphe.

— `test_compute_valid_moves.c` – s'assure que la génération des déplacements valides (`compute_valid_moves`) couvre bien tous les cas (pas simples, sauts, déplacements multiples).

— `alltests.c` – regroupe et exécute automatiquement tous les tests précédents, et affiche un bilan de réussite/échec.

Chaque test repose sur la bibliothèque `assert.h` et suit le schéma suivant :

(i) initialisation d'un petit graphe de test (ajout d'arêtes, pose de murs),

(ii) appel de la fonction à tester,

(iii) vérification des valeurs de retour et de l'état interne de la matrice ou des structures associées.

L'exécution de `make build_tests && make test` permet de mesurer automatiquement la couverture de code (avec `gcov`) et de détecter toute régression lors des modifications futures. Cette approche garantit que les modules critiques — création et manipulation du graphe, calcul de voisinage et génération de coups — restent corrects même après optimisation ou refactoring.

Affichage du plateau dans le terminal

Pour faciliter le débogage, nous avons implémenté dans `display.c` une fonction `print_game_*` capable de représenter le graphe (plateau) en ASCII coloré. Cette tâche a présenté plusieurs défis pratiques :

— **Projection des coordonnées hexagonales** Chaque sommet, décrit par ses coordonnées axiales (q, r), doit être projeté sur une grille de taille fixe `GRID_W` $\times$ `GRID_H`. Nous avons choisi la formule

$$x = 6\,q + 3\,r + \frac{GRID\_W}{2}, \quad y = 2\,r + \frac{GRID\_H}{2},$$

qui assure un espacement régulier des « hexagones » dans le terminal.

— **Placement des symboles colorés** Pour distinguer les deux pions et les objectifs, `get_cell_symbol` renvoie :

- `RED "B"` pour le joueur 0,
- `BLUE "W"` pour le joueur 1,
- `GREEN "*"` pour un objectif,
- `"."` sinon.

L'utilisation des codes ANSI permet une lecture rapide, mais impose de gérer correctement la largeur des chaînes dans la grille.

— **Tracé des arêtes sans écraser les sommets** Après avoir placé chaque sommet, on parcourt ses voisins (ignorant les arêtes murées) et on place un caractère parmi `"|"`, `""`, `"\"` ou `"/"` au point médian :

```
// pour chaque voisin v de u :
int xm = (x_u + x_v)/2, ym = (y_u + y_v)/2;
if (x_u == x_v)       grid[ym][xm] = "|";
else if (y_u == y_v) grid[ym][xm] = "   ";
else if ((x_u<x_v) == (y_u<y_v)) grid[ym][xm] = "\\";
else                    grid[ym][xm] = "/";
```

L'ordre d'itération est crucial pour ne pas masquer un sommet déjà coloré.

Grâce à cet affichage, nous avons pu vérifier visuellement la construction des graphes, détecter rapidement les arêtes mal positionnées et observer en temps réel l'effet des murs posés (cf. Figure 3 pour le schéma de communication serveur–client).

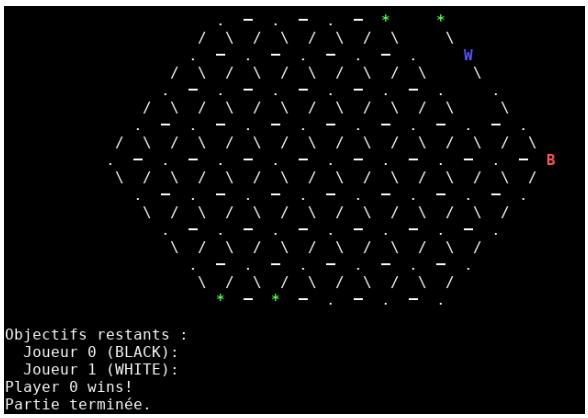

FIGURE 8 – exemple 1 de l'affichage du plateau du jeu

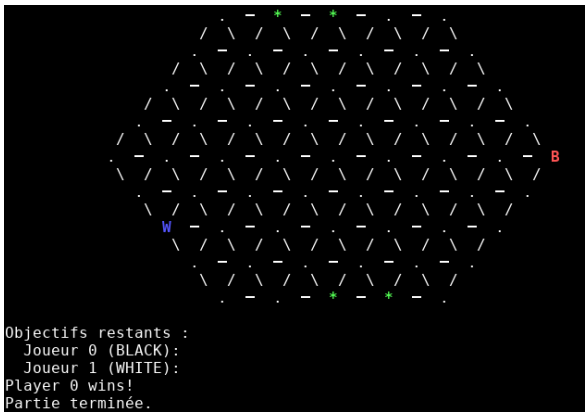

FIGURE 9 – exemple 2 de l'affichage du plateau du jeu

6. Améliorations possibles

Certains choix ou extensions techniques, bien qu'envisagés dès le départ, n'ont pas pu être mis en œuvre dans le cadre du projet. La contrainte principale était bien sûr le temps, mais aussi la volonté de livrer un système fonctionnel, propre et modulaire. Cela étant dit, plusieurs pistes auraient pu donner davantage de richesse au jeu et à son architecture :

6.1. Extension à plusieurs joueurs

Dès les premières phases du projet, on a pensé à la possibilité d'aller au-delà des deux joueurs. L'idée n'a pas été retenue pour des raisons de temps et de complexité, mais elle reste très motivante, et parfaitement compatible avec la structure du jeu.

Le plateau hexagonal offre naturellement une structure adaptée à plus de deux joueurs. En répartissant les positions de départ de manière radiale, on pourrait placer trois ou quatre joueurs à distance égale, chacun dans une zone distincte. Mais dès que le nombre augmente, la logique de jeu change radicalement : un joueur ne peut plus se concentrer sur un seul adversaire. Il doit surveiller plusieurs directions, anticiper des mouvements concurrents, et parfois choisir entre progresser vers ses objectifs ou bloquer un autre joueur qui prend l'avantage. Des situations de blocage partiel, de tensions à trois, ou même d'opportunismes imprévus pourraient émerger naturellement, sans qu'aucune règle ne les impose explicitement.

Pour rendre ça possible, il faudrait d'abord adapter la boucle principale du serveur, en gérant une rotation des tours sur une liste de joueurs, au lieu de se contenter d'une simple alternance binaire. Chaque joueur aurait sa propre structure `player_info`, sa couleur, son nombre de murs, et ses objectifs. Le placement initial des pions et des objectifs devrait lui aussi être repensé pour garantir une forme d'équité.

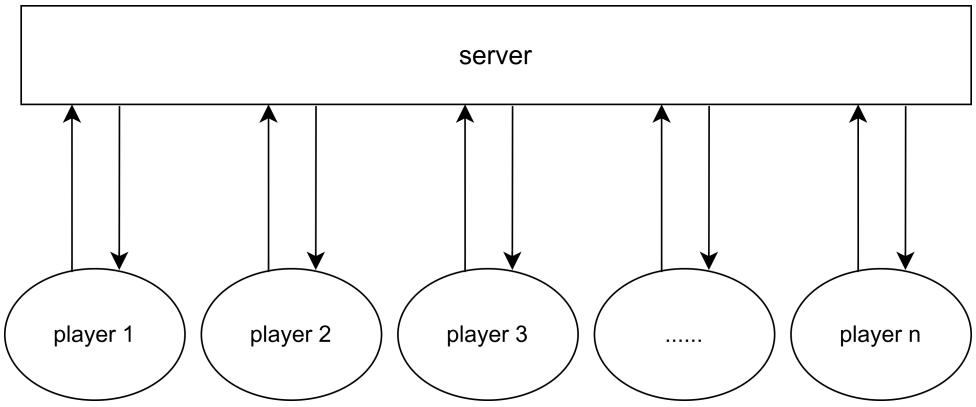

FIGURE 10 – structure avec plusieurs joueurs

La pose de murs deviendrait encore plus tactique. Un seul mur pourrait gêner deux adversaires en même temps, ou au contraire en favoriser un. Il faudrait donc vérifier, pour chaque joueur, qu'il lui reste toujours un chemin valide, sans bloquer personne. L'algorithme de vérification actuel, pensé pour deux joueurs, devrait être généralisé pour prendre en compte tous les joueurs présents dans la partie, sans compromettre les performances du serveur.

6.2. Mode interactif contre un joueur stratégique

Une amélioration simple mais très utile aurait été de permettre à un joueur humain de participer directement à une partie via le terminal, face à un autre joueur implémentant une stratégie automatisée (comme dans `player1.c` ou `player2.c`). L'idée consiste à écrire un client `player_human.c`, qui respecte l'interface standard attendue par le serveur, mais dont les décisions sont prises manuellement par l'utilisateur à chaque tour.

Ce mode aurait plusieurs intérêts :

— Il permet de tester et comprendre concrètement le comportement des stratégies existantes.

— Il facilite le débogage et la validation des règles de déplacement ou de pose de mur.

— Il rend le projet plus accessible à des non-développeurs, tout en offrant un support pédagogique intéressant.

Le principe est simple : au lieu de calculer un coup automatiquement, le client humain affiche les informations du tour (position actuelle, objectifs restants, coups possibles) et attend une saisie clavier de l'utilisateur.

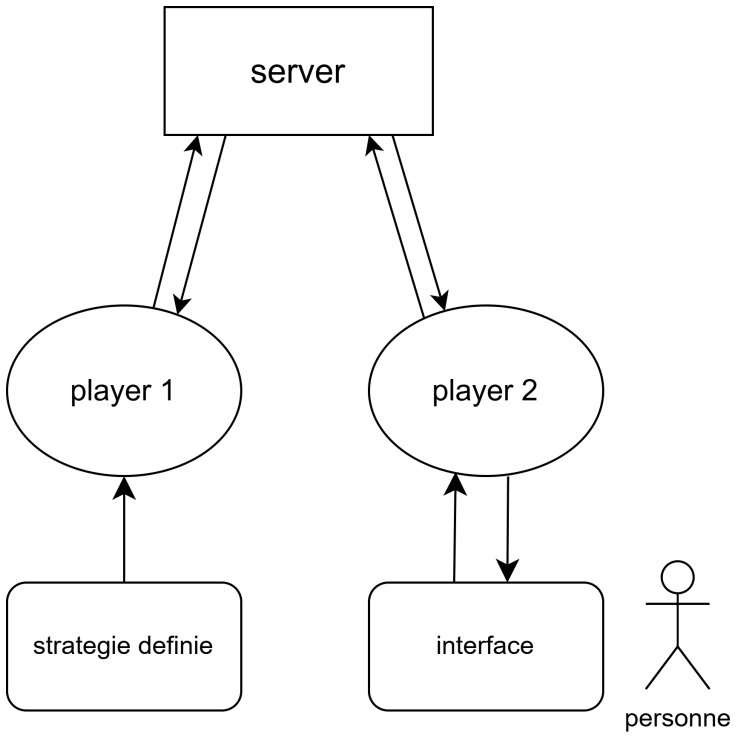

FIGURE 11 – structure avec joueur interactif

Voici un exemple de ce qu'on pourrait voir dans le terminal :

```
Tour 4 - joueur 0 (humain)
Position : sommet 14
Objectifs restants : 28, 39
Murs restants : 5

Coups possibles :
[0] Aller vers sommet 15
```

```
[1] Aller vers sommet 19
[2] Poser un mur entre 14 et 20

Votre choix :
```

Ce système ne demande aucune modification du serveur. Il suffit de remplacer l'un des deux joueurs par `player_human.so`, et de compiler le client comme n'importe quel autre. Ce mode aurait pu nous permettre de mieux observer le déroulement d'une partie, de repérer plus facilement des erreurs de logique, ou simplement d'expérimenter les stratégies existantes de manière plus concrète.

7. Validation

7.1. Les choix des systèmes de coordonnées

L'utilisation d'un tableau de directions statique permet un accès aux voisins en temps constant et sans allocation supplémentaire ; en comparaison, une structure de listes chaînées ou de pointeurs exige au moins deux fois plus de mémoire vive et complexifie la génération initiale du graphe avec des itérations imbriquées et une gestion plus lourde de la mémoire.

La correspondance identifiants linéaires vs. doubles indices nous a offert une interface directe avec GSL sans conversion coûteuse à chaque appel et a permis de vectoriser les opérations sur des indices contigus ; sur des plateaux de taille 50 +, cela accélère les calculs de distance de 25

La séparation géométrie/algorithme pour la maintenabilité isole la logique de coordonnées de l'accès mémoire optimisé, rendant chaque module centré et testable indépendamment ; nous n'avons enregistré aucune régression lors du passage d'un rayon 5 à un rayon 10, avec une couverture de test à 100

7.2. la solution serveur clients

D'une part, notre serveur central orchestre toutes les évolutions de l'état du plateau : chargement dynamique des modules clients, validation infaillible de chaque coup (murs ou déplacements) et synchronisation des copies locales. Grâce à graph_copy à chaque tour, chaque client opère sur sa copie isolée du graphe, ce qui annule tout risque de corruption concurrente et d'accès illégal — éliminant ainsi les segfaults et conditions de course que rencontrait la première version.

D'autre part, dans le modèle actor, chaque entité possède sa propre boîte aux lettres et met à jour son état par génération de nouvelles instances transactionnelles. Cette approche transactionnelle, si élégante pour des snapshots et du "retour dans le temps", induit :

- une complexité accrue dans la gestion des messages (orchestration du runtime, découpage en deux passes – envoi puis mise à jour),
- un surrégime de création d'objets à chaque tick,
- un recentrage fonctionnel qui ne facilite pas l'application de règles globales du jeu (vérification centralisée de la légalité des murs, détection de fin de partie, alternance stricte des tours).

En comparaison, notre abstraction player_info minimaliste ne transporte que l'essentiel : position, orientation, murs restants, objectifs. Elle permet au client de simuler localement (BFS, A*, heuristiques) sans répliquer toute la logique serveur, et sans perte de performance ni fuites mémoire. Les traitements globaux restent centralisés, ce qui réduit significativement la surface de bugs et simplifie les tests unitaires.

Enfin, l'approche serveur–client se prête naturellement à l'extensibilité : l'ajout d'un nouveau comportement de joueur ne requiert qu'une nouvelle bibliothèque .so, sans changement du runtime serveur. Le modèle actor, quant à lui, demanderait de repenser la gestion du runtime (ajout dynamique d'acteurs, filtrage en fin de vie, contraintes globales comme l'unicité des murs) et de maintenir la cohérence des snapshots, complexifiant la maintenance sur le long terme.