Compte Rendu INFO1b : 7Colors

Samuel Ogulluk samuel.ogulluk@ens-rennes.fr [1]

[1]ENS Rennes, Département de Mécatronique

4 mai 2025

Table des matières

Introduction **2**

1 Le jeu **2**
1.1 Les files . . . 2
1.2 L'exploration de graphes . . . 2
1.3 La vitesse d'execution . . . 3

2 L'interface graphique **3**

3 Les Agents **4**
3.1 L'Agent Stochastique . . . 4
3.2 L'Agent Glouton . . . 4
3.3 L'Agent Hégémonique . . . 4
3.4 L'heuristique . . . 5
3.5 La recherche en profondeur . . . 6
3.6 Les Agents Frontières . . . 8

4 Analyse des différents Agents **9**

Introduction

L'objet de ce DM est de créer et de faire un agent capable de jouer au 7colors, un jeu de stratégie dans lequel l'objectif est de capturer le plus de territoire sur une grille initialisée aléatoirement. L'ensemble des codes sont disponibles sur Github.

1 Le jeu

1.1 Les files

Afin d'avoir une efficacité algorithmique améliorée et pour avoir de meilleures performances, j'ai voulu éviter l'utilisation d'algorithmes récursif en utilisant de la programmation dynamique. Ainsi, j'ai créé diverses fonctions pour créer et gérer des files (dans le fichier queue.c). Afin d'augmenter l'efficacité des files et tout particulièrement pour augmenter l'efficacité de la fonction :" isinQueue" permettant de dire si un élément est dans la file, j'ai ajouté en parralèle de cette file, un tableau remplit de valeurs ordonnées (en utilisant une injection de $\phi : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$) ainsi, les files crées étaient de complexité (temporelle) :

action	Complexité temporelle (O), (n est le nombre de données dans la file)
insérer	log(n)
isinqueue	log(n)
retirer	log(n)
copier	n
reset	1

Table 1 — *Caption*

Toutes les données stockées dans la file sont dans la Heap pour des raisons d'allocations dynamique et pour limiter l'usage mémoire au maximum, quitte à perdre légerement en performance étant donnée le nombre très élevé de files qui seront crées par la suite (surtout pour les agents d'exploration en profondeur). L'injection utilisée était dans un premier temps : $\phi : (a, b) \mapsto (a + b)(a + b + 1) + b$, Cependant, par manque d'efficacité et surtout du fait des valeurs potentiellement très élevée de cette injection, j'ai opté pour utiliser $\phi : (a, b) \mapsto (a \times 2^{16}) + (b\%2^{16})$ qui a l'avantage d'être rapidement calculable (via des opérations sur les bits directement) et d'avoir des valeurs qui sont plus facilement controlables (qui peuvent tenir sur un entier en 4 octets). De plus, l'intérêt d'utiliser des files et de la programmation dynamique était pour l'analyse de complexité, ainsi que pour le debogage et l'absence de redondance dans les calculs effectués.

1.2 L'exploration de graphes

A l'aide des files créées, j'ai pu ensuite créer la fonction GR0_getnetwork qui prend en entrée les coordonnées d'un carré et qui permet d'obtenir une file avec tous les carrés de même couleur rattaché à ce dernier, on a un sous réseau (que j'appelerai noeud par la suite) du réseau que compose la carte. On appelera par la suite cette fonction la fonction "Territoire" T(joueur, pas de temps). A partir de ce noeud, on peut définir différentes fonctions, ajouter des vérifications des voisins de ce noeud pour obtenir les coups jouables par le joueur (GR0_get_move_available) en partant de sa racine (le coin dans lequel le joueur commence) et en explorant les noeuds voisins.

On appelera par la suite cette fonction "Coup" C(joueur, pas de temps) et qui renvoie un tableau de 7 files dans lesquelles sont stockées les mouvements possibles .

L'usage de la programmation dynamique permet ici de se ramener d'un algorithme en force brute en $O(n^4)$ (ou n^3 selon l'algorithme exact utilisé) vers un algorithme en $O(n \log_2(n))$ et donc de considérablement accélerer le déroulement du programme ce qui sera particulièrement utile pour la suite.

1.3 La vitesse d'execution

La vitesse d'execution peut varier d'un ordinateur à un autre, les résultats suivants servent davantage d'ordre de grandeur, notamment en terme de complexité algorithmique. La figure suivante montre le temps d'execution de 10000 affrontements entre des agents parmis ceux qui seront étudiés et décrit plus tard. On observe une croissance importante du temps d'execution

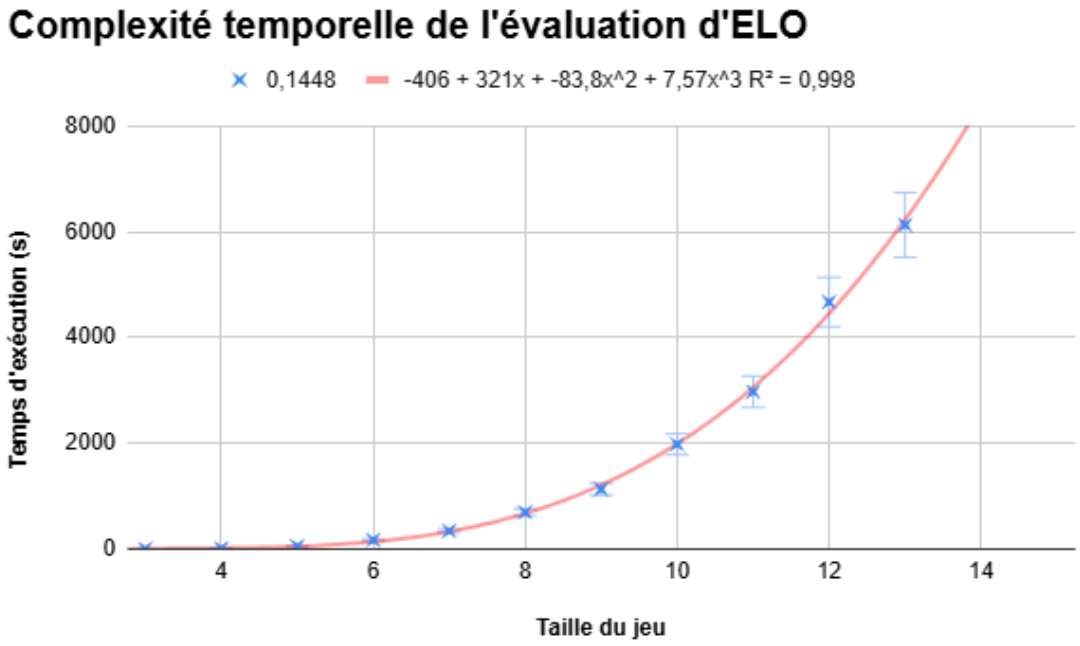

FIGURE 1 — *Complexité temporelle*

en fonction de la taille de la grille. Ainsi, pour une seule partie, le temps moyen total peut aller de moins de 10μs pour une grille de taille 3, à 0.1s pour une grille de taille 15. La complexité semble suivre un modèle en $O(n^4)$.

2 L'interface graphique

Une fois toutes ces fonctions d'exploration de graphe et de fonctionnement du jeu fonctionnelles, on s'intéresse à l'aspect graphique et à l'interface de jeu.

L'interface a été mise sous deux formes, une première forme pour jouer dans le terminal, pour tester le bon fonctionnement des algorithmes et les comparaisons entre les Agents, ainsi qu'une version graphique à l'aide de la librairie SDL2 (et de SDL2ttf).

La première version a été faite en utilisant la couleur du fond dans le terminal, en encodant chaque couleur puis en y faisant appel.

En m'inspirant de ce qui est fait dans le monde des échecs, j'ai ajouté une barre d'estimation de victoire à gauche de la grille. Cette barre évolue en fonction de la partie et montre qui est en train de gagner selon l'algorithme minmax8 qui sera vu plus tard.

De plus, un bouton "indice" est disponible afin de mettre en avant les coups "idéaux" donnés par l'Agent Minmax8.

Les agents peuvent etre sélectionnés dans un menu au démarrage du jeu. Additionnelement, ces derniers peuvent aussi être selectionnés via une ligne de commande le fonctionnement du jeu (selon les modalités définies dans le sujet).

3 Les Agents

3.1 L'Agent Stochastique

Le tout premier agent créé est un agent jouant au hasard parmi les coups qui lui sont possibles. Pour ce faire, j'ai commencé par extraire les mouvements possibles (encodé sur un entier en 8 bits où un 1 représente une couleur jouable et un 0 représente une couleur non jouable), puis en prenant un de ces 1 au hasard, l'algorithme fonctionne correctement.

3.2 L'Agent Glouton

Vient ensuite l'algorithme glouton, dont l'heuristique est la suivante, soit $p \in \{0,1\}, t \in \mathbb{N}$: $\mathcal{H}_g(p,t) = ||T(p,t+1)||$.

Ici l'heuristique a été légerement changée pour avoir l'algorithme suivant : $\mathcal{C}_g(p,t) = \text{argmax}_{i\in[[0,6]]}(||C(p,t)[i]||$

Cependant, ces heuristiques sont en réalité équivalents, puisque choisir la couleur maximisant le nombre de cases acquise revient à maximiser le nombre de cases acquise au tour suivant.

3.3 L'Agent Hégémonique

L'algorithme hégémonique vient ensuite. Son heuristique quant à lui est la suivante :

$$\mathcal{H}_h(p,t) = \left(\sum_{i\in[0,7]} ||C(p,t+1)[i]|| \right)$$

Autrement dit, il vient essayer de maximiser le nombre de cases qui sont à sa frontière à chaque tour.

Cet algorithme a cependant un défaut majeur en l'état, lorsqu'il se retrouve dans une position où ses frontières sont obligées de diminuer, il prend de moins bons coups puisqu'il essaie de minimiser la perte de frontière alors qu'ils doit justement essayer de capturer le plus de cases. Ce problème peut être palié en lui couplant un algorithme glouton dans ces cas là. On a ainsi un agent mixte.

3.4 L'heuristique

Ensuite, j'ai voulu ajouter certains comportements à mes agents, ainsi, j'ai développé le masque suivant :

$$\mathcal{H}(x,y) = \delta \frac{\tanh\left(\frac{2x}{s}\right) + \tanh\left(\frac{-2y}{s}\right) + \exp\left(-\frac{x^2+y^2}{(sa)^2}\right)}{2\tanh(1)}$$

où $\delta \in \{-1, 1\}$ en fonction du joueur qui joue, s est la taille de la map, a=0.3 pour contrôler à quel point on donne une priorité au milieu, le 2tanh(1) sert à pseudo normaliser (restreindre entre -1 et 1) . L'idée étant que plus on est proche du centre, plus on a du controle de territoire et de

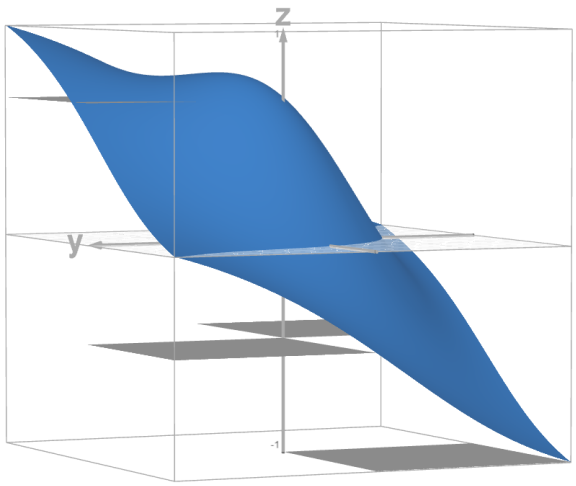

FIGURE 2 — *Masque des récompenses*

potentielles cases futures, et plus on loin de sa zone de départ, plus on empiète sur le territoire adverse. Ainsi, pour modéliser ces comportement, on somme une tangente hyperbolique pour représenter la récompense liée à l'éloignement, à une gaussienne pour représenter l'avantage lié au centre.

Cependant, afin d'avoir une heuristique rapide à calculer, j'ai utilisé les approximations suivantes : On utilise l'approximation suivante pour la tangente :

$$\tanh(\frac{2x}{s}) \underset{0}{\sim} \frac{2x}{s} \frac{27 + \frac{2x}{s}^2}{27 + 9\left(\frac{2x}{s}\right)^2}$$

Tandis que pour l'exponentielle, on utilise l'approximation suivante :

$$e^x \underset{]-10,0[}{\sim} \frac{1}{1 - x(1 - 0.5 \times x)}$$

Ces approximations permettent d'obtenir des résultats rapidement et avec une précision correcte, sachant que le masque n'a pas vocation a être précis. Un développement en profondeur de cette idée pourrait aboutir à un développement d'un algorithme de deep-Q Learning qui donnerait une Q-table donnant les récompenses associées à chaque coup joués et donc à un masque reflétant l'expérience d'un agent.

3.5 La recherche en profondeur

En parralèle de ce masque, l'exploration d'un graphe implique souvent une recherche en profondeur et tout particulièrement l'algorithme minimax. Ce dernier fonctionne selon le principe suivant :

Algorithm 1: Algorithme Minimax avec élagage alpha-bêta

```
1  Fonction Minimax_AlphaBeta(noeud, profondeur, alpha, beta, estMaximisant) :
2      if profondeur = 0 ou noeud est terminal then
3          return ÉVALUATION(noeud)
4      if estMaximisant then
5          val ← −∞
6          for chaque enfant dans GÉNÉRER_COUPS(noeud) do
7              val ← max(val, Minimax_AlphaBeta(enfant, profondeur − 1, alpha, beta,
                faux))
8              alpha ← max(alpha, val)
9              if beta ≤ alpha then
10                 break                                   // Élagage beta
11         return val
12     else
13         val ← +∞
14         for chaque enfant dans GÉNÉRER_COUPS(noeud) do
15             val ← min(val, Minimax_AlphaBeta(enfant, profondeur − 1, alpha, beta,
                vrai))
16             beta ← min(beta, val)
17             if beta ≤ alpha then
18                 break                                   // Élagage alpha
19         return val
```

Cet algorithme est récursif (je ne l'ai pas adapté en programmation dynamique par manque de temps, cependant les gains de cette transition ne serait absolument pas négligeables), l'élagage alpha-bêta permet d'éviter une exploration inutile de certains coup (notamment en cas de victoire sur l'un des noeuds, une exploration plus en profondeur ne serait pas utile). Une fois cet algorithme développé, j'ai souhaité connaitre l'importance de la profondeur d'exploration. Ceci est représenté sur la figure suivante à l'aide du système ELO :

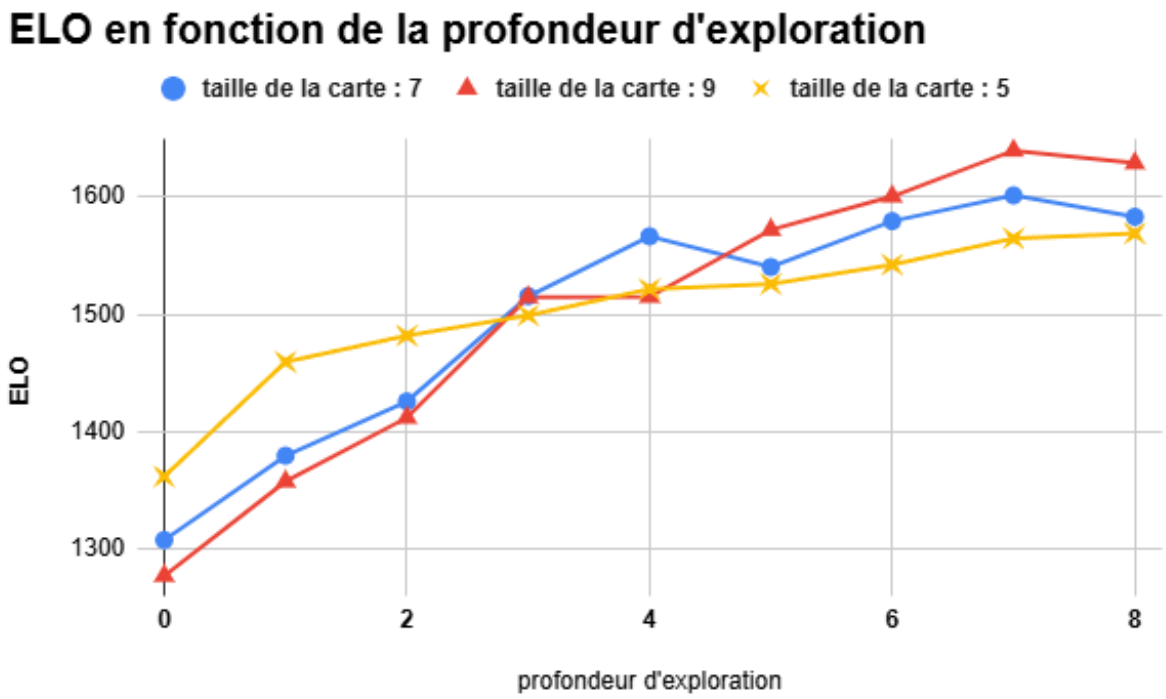

FIGURE 3 — *Variation de l'ELO en fonction de la profondeur d'exploration et de la taille de la carte*

On observe l'influence de l'exploration en profondeur sur l'amélioration de l'agent. Il est à noté que sur ce graphe, le degré 0 de la profondeur d'exploration est en fait un algorithme glouton servant de base de comparaison. Ainsi, les deux premiers points illustrent l'influence du masque choisi, qui a lui seul parvient à hausser le niveau de 80 points elo, c'est à dire que l'algorithme glouton avec heuristique a 60% de probabilité de l'emporter face à l'algorithme glouton simple. De plus, on constate la concavité de la courbe. Ainsi, explorer en profondeur trop loin n'est pas toujours intéressant, tout particulièrement d'un point de vue computationnel, ici, l'implémentation utilisé fonctionne dans des temps raisonnalbesau moins jusqu'à 14 de profondeur, je me suis cependant restreint à 8 ici pour une raison de temps (au delà de 10, les temps pour chaque coup deviennent trop long pour que 10000 parties soient raisonnables) De plus, comme la figure 4 le montre, le temps de réflexion d'un algorithme en profondeur évolue au cours de la partie, il est faible au début, lorsque le nombre de coup est restreint, puis croit exponentiellement lorsque l'espace des coups possibles augmente. Un compromis est donc à trouver afin de toujours respecter le délai imposé. Cet équilibre semble ici se trouver entre 4 et 7 en fonction des performances souhaitées.

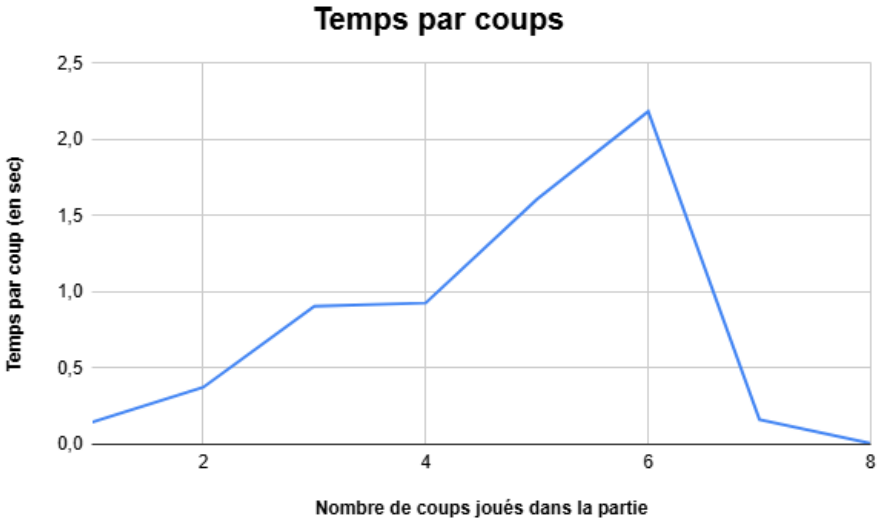

FIGURE 4 — *Temps par coups*

Enfin, on constate que plus la carte est grande, plus la profondeur de l'exploration est importante, avec une différence d'ELO de 50 points entre la profondeur 8 sur une carte de taille 5 ou 9.

3.6 Les Agents Frontières

Ensuite, l'Agent Frontière5 a été créé en utilisant l'heuristique suivante :

$$\mathcal{H}_{F5} = ||C(1,t)|| - ||C(2,t)||$$

Cette heuristique permet de prendre en compte la volonté de croissance des frontières du joueur 1, tout en essayant de minimiser les frontières du joueur adverse, et donc essayer de palier au défaut de l'algorithme hégémonique. Cette heuristique est ici mise en couple avec un Minimax de profondeur 5.

Un deuxième algorithme peut etre créé à partir de cette idée, en modifiant l'heuristique choisie et en appliquant l'heuristique choisie précedemment aux coups possibles, on obtient une heuristique améliorée pour Frontière5 :

$$\mathcal{H}_{F5h} = \sum_{(x,y)\in C(1,t)} \mathcal{H}(x,y) - \sum_{(x,y)\in C(2,t)} \mathcal{H}(x,y)$$

J'ai également utilisé OpenMP pour parraléliser les affrontements et la recherche en profondeur, ce qui a fait augmenté d'un facteur 100 la vitesse d'execution de la boucle pour calculer l'ELO. Cette accéleration pourrait encore être accentuée en utilisant un GPU pour l'éxecution des fonctions simples et indépendantes comme :"GR0_get_adjacent_cases" qui vérifie les cases alentours d'une case donnée.

4 Analyse des différents Agents

Afin de comparer les agents, j'ai utilisé un classement ELO. Ce système fonctionne en opposant les algorithmes entre eux et en mettant à jour leurs points respectifs en fonction de leurs défaites ou victoires selon la regle suivante : Soit deux agents i et j, à chaque tour,

$$\begin{cases} E^i_{n+1} = E^i_n + K_i[W_i - p(D)] \\ E^j_{n+1} = E^j_n + K_j[W_j - p(D)] \end{cases}$$

où $D = E^i_n - E^j_n$ et $p(D) = \frac{1}{1+10^{\frac{-D}{400}}}$ est la probabilité de victoire d'un agent face à l'autre en fonction de leur différence d'ELO, et W_i ou W_j valent 1 si le joueur a gagné ou 0 sinon. Enfin, K permet de gérer à quel point chaque affrontement modifie l'ELO des deux joueurs (K est élevé au début et diminue avec les affrontement pour avoir une convergence de l'ELO). A noter que le classement ELO commence pour tous les agents à 1500. Le classement ELO des différents algorithmes utilisés donne : De plus, en fonction de la taille de la carte, la fonction

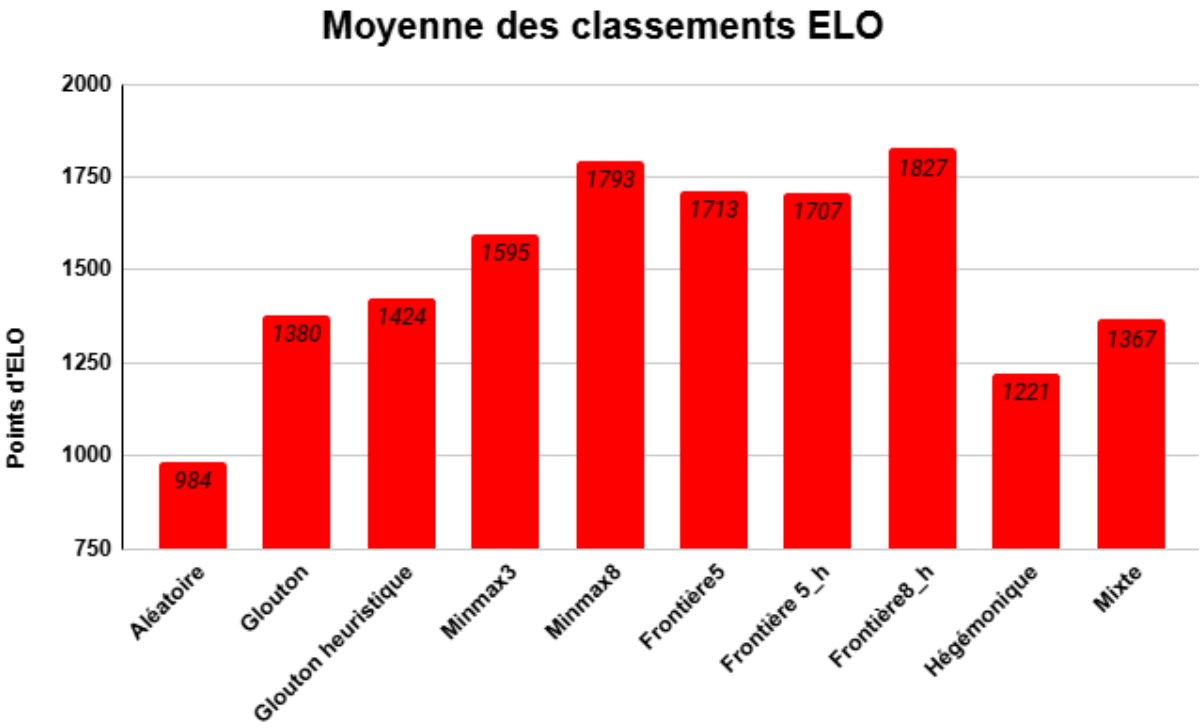

FIGURE 5 — *Comparaison des ELO des différents Agents*

GR0_general_ranking permet d'obtenir la figure 6 en moyennant 3 classement elo pour chaque carte avec des tailles allant de 3 à 15. (cette fonction est longue à faire tourner, des résultats moins précis peuvent être obtenus en diminuant

— le nombre d'agents (jouer sur ce facteur influe néanmoins sur la valeur donnée mais pas sur les probabilités exprimées sous forme d'ELO)

— le nombre de tailles de carte sur lesquelles ont teste l'elo des Agents
— le nombre de parties pour chaque classement ELO (il vaut mieux le garder au dessus des 500 pour obtenir un résultat avec une bonne précision)
— le nombre de classements que l'on fait pour chaque taille (on vient ensuite moyenner les classements)

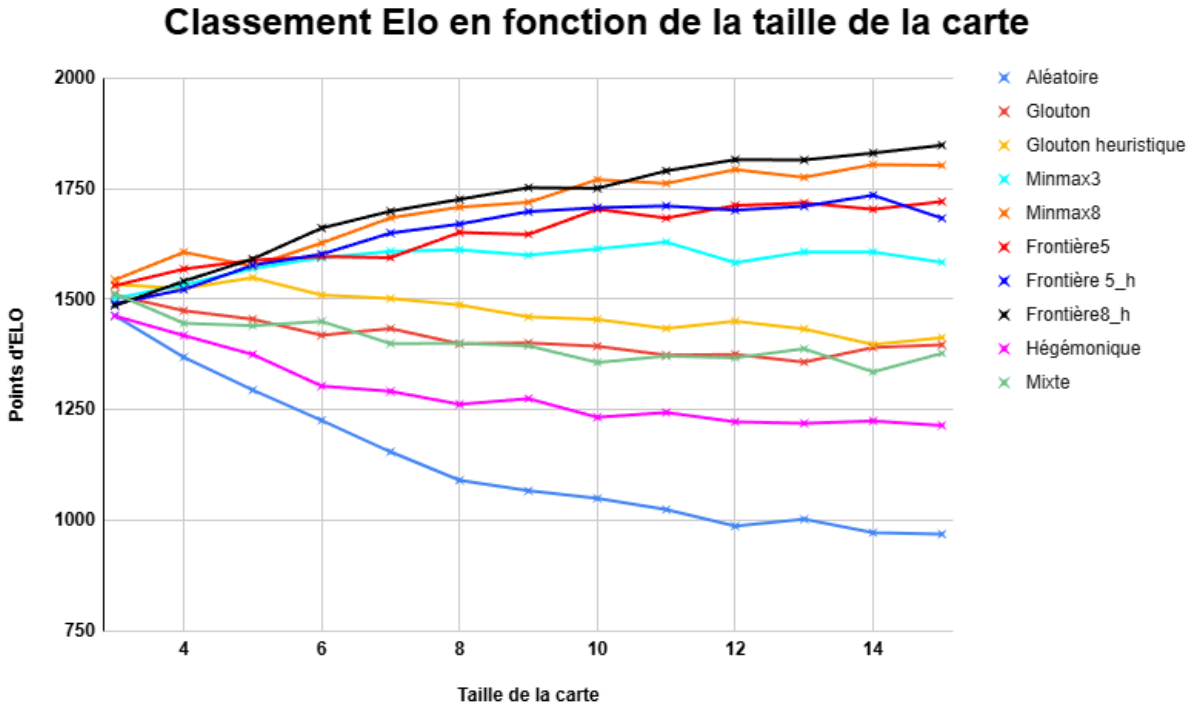

FIGURE 6 — *Comparaison des ELO en fonction de la taille de a carte*

On constate immédiatement une nette domination des algorithmes en recherche en profondeur. Ces algorithmes ont en moyenne 300 points d'ELO par rapport aux autres algorithmes. De plus, si l'algorithme stochastique a des bons résultats aux petites échelles (du fait de la répartition des couleurs sur la carte ayant une importance plus élevée), cet algorithme est dépassé lorsque la taille augmente et aurait 99% de probabilité de perdre face à un minmax8 ou un frontière5 + heuristique pour des grandeurs 14 ou plus.

L'algorithme frontière5 et frontière5 avec le masque performe plutot bien du fait de son exploration en profondeur. Par ailleurs, en poussant avec un Frontière8 et l'heuristique, on obtient le meilleur agent proposé ici. Cependant, cet agent dépasse souvent la limite de temps autorisé pour des cartes allant au delà d'une taille de 8.

Enfin, j'ai fait de sorte que d'autres agents soient jouables et d'ouvrir la fonction servant à classer les agents à d'autres agents d'autres personnes. Pour ce faire, il suffit de mettre le nom du fichier dans lequel est l'autre agent dans l'entete du fichier console.h, et d'ajouter un pointeur vers cette fonction à la liste dans GR0_elo_ranking. Il est à noté qu'un classement ELO rigoureux sur plusieurs tailles de cartes prend du temps de calcul et beaucoup de parties, ainsi, la création de la figure 6 via la fonction GR0_general_ranking a pris 12h de calculs sur mon ordinateur à 100% d'utilisation CPU en permanence, pour un total de 136000 parties jouées. Ces nombres

peuvent facilement être diminués en influant sur les facteurs vus précedemment et être ramenés à des ordres de grandeurs bien plus faibles, tout en restant relativement corrects.

Conclusion

Au cours de ce projet, j'ai pu mettre en pratique les cours de C appris sur ce semestre, ainsi que diverses autres ressources et idées que j'ai pu développer ces dernières années, afin d'avoir des agents pour jouer au 7colors, ainsi que pour avoir une interface graphique fonctionnelle et intuitive pour ce jeu. Des pistes possibles de continuation du projet pourraient être :

- Menu pour changer de thème
- Menu pour changer de police
- si on veut afficher le territoire de chaque joueur
- taille de la grille fixe, seule la taille des carreaux change dans l'affichage SDL
- mettre des sons
- mettre des animations
- mode sombre/ clair
- affichage de l'agent contre lequel on joue
- utilisation de Deep-Q Learning ou de Proximal Policy Optimisation conjointement à des réseaux de neurones en graphes pour avoir des agents potentiellement encore meilleurs.

Je souhaiterais enfin vous remercier pour votre enseignement que j'ai tout particulièrement apprécié et pour la découverte du langage C que j'affectionne désormais énormement (à quelques segfault près).

PS : OpenMP génère 2 fausses pertes de mémoire repérées par valgrind. Ces fuites de mémoire ne sont pas réelle et leur est causé par cette librairie. De plus, l'utilisation de SDL et particulièrement l'ouverture d'une fenêtre SDL empêche l'utilisation de Valgrind en saturant de messages d'erreurs. Enfin, le programme contient un certain nombre de fuites de mémoire d'après Valgrind pour lesquelles je ne parvient pas à trouver les origines (et pour lesquelles –leak-check=full ne m'affiche rien).