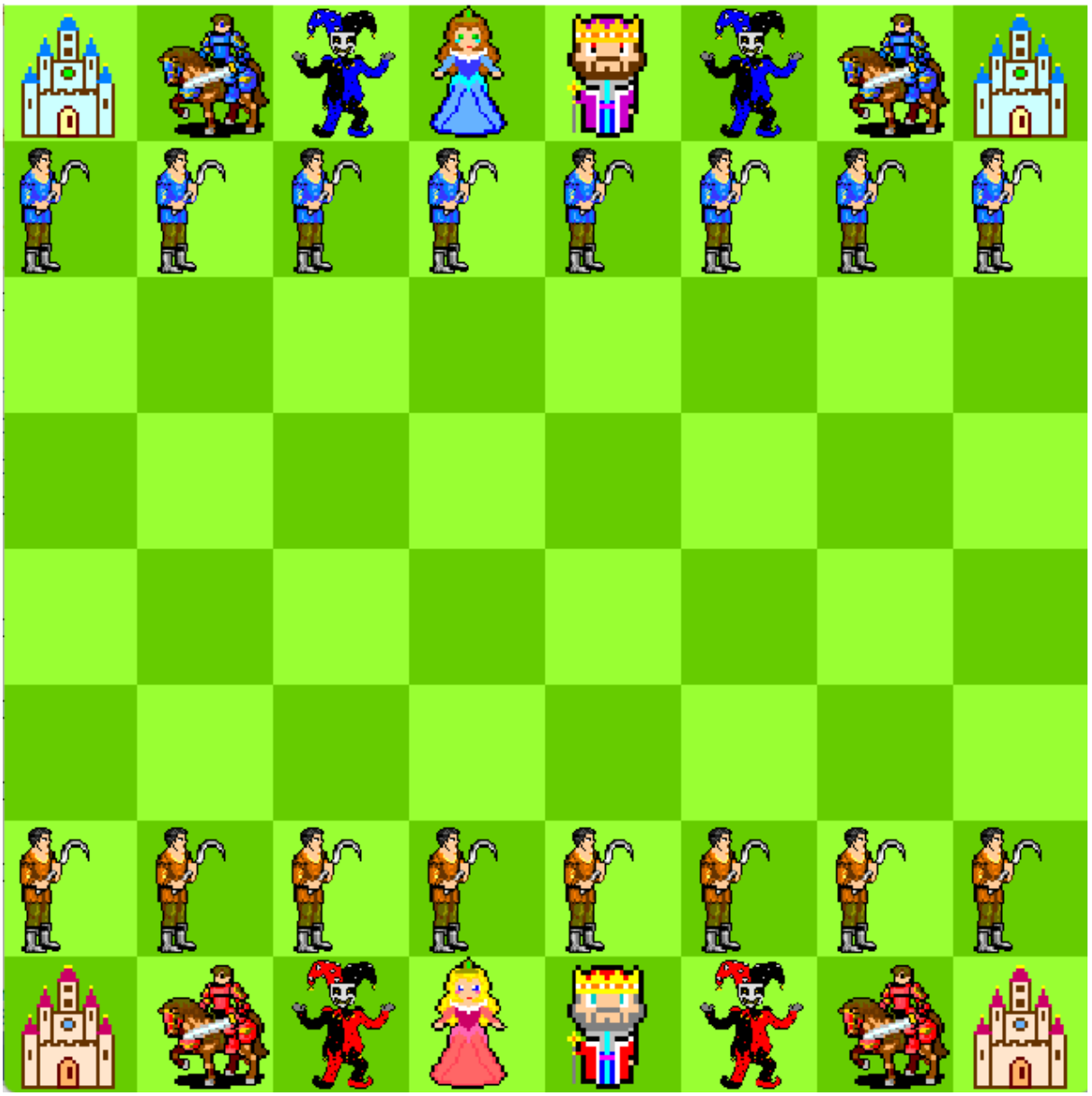
Projet E3 : Jeu d’échecs



Timothée ROYER - Clément LAVIE - Hugo VAUBOURG - Mathurchan JEYAKANTHAN

Tuteur: Lilian BUZER

Plan

Objectif

Le but de ce projet est de créer un jeu d'échecs où l'on peut jouer contre une IA, qui a pour but de calculer et choisir les meilleurs coups possibles afin de nous battre.

Introduction

Étant depuis longtemps intéressés par les jeux de stratégie, nous avons donc décidé de créer notre propre jeu. Pour cela, nous avons commencé par étudier quelques algorithmes tels que l'algorithme du plus court chemin ou le miniMax ,qui sont référence dans l'apprentissage algorithmique. Après avoir fini cet entraînement, nous nous sommes lancés dans la création d'un jeu d'échecs pour pousser plus loin l'un des algorithmes vu: le MiniMax.

Apprentissage d'algorithme

Dans un premier temps, afin d'apprendre et comprendre le fonctionnement des algorithmes nécessaires à la conception de notre IA, nous nous sommes intéressés à plusieurs types d'algorithmes sur Python tels que ceux qui sont présents dans les jeux Pacman, Morpion ou Puissance 4.

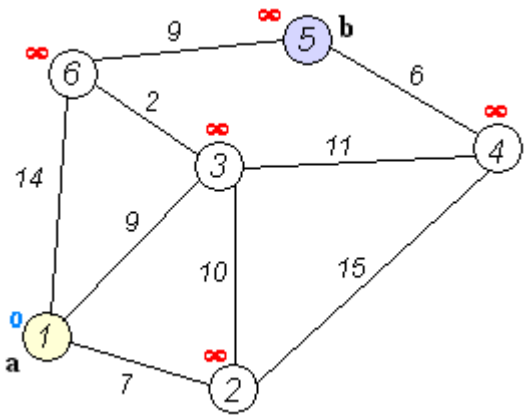
A) Pacman

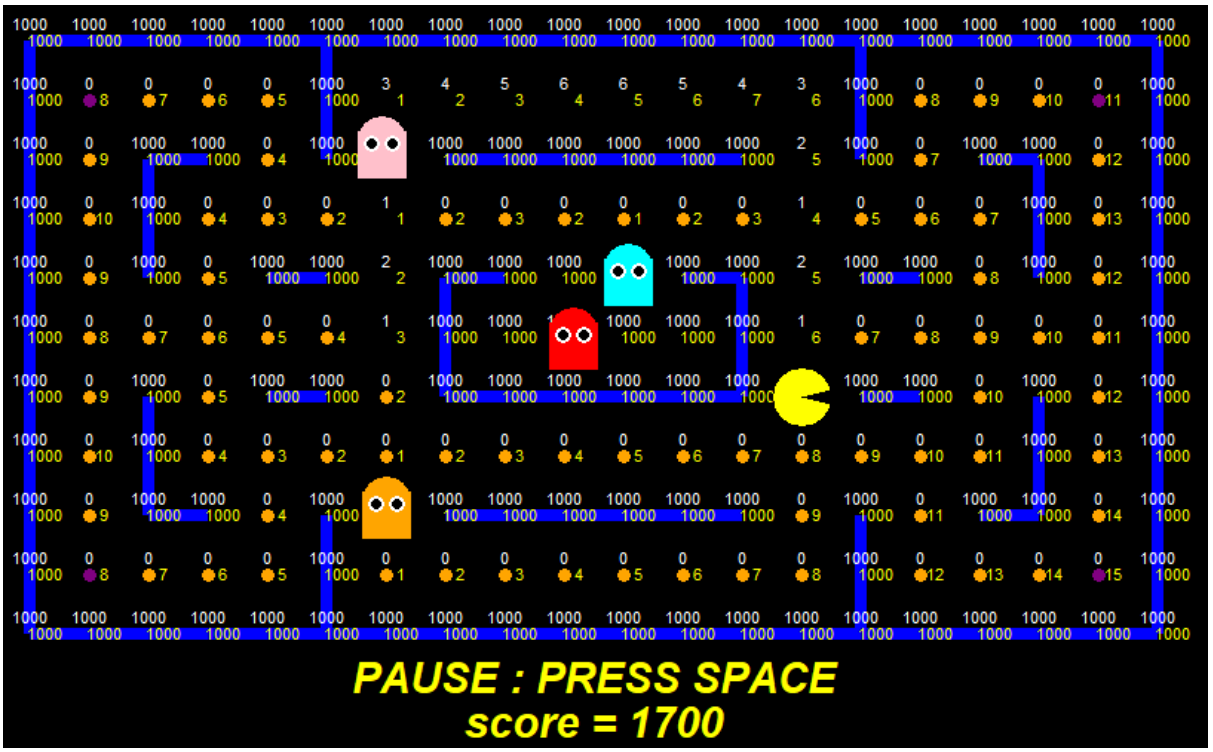
Dans le jeu pacman, nous avons créé une IA qui contrôle Pacman.

Elle cherche donc à manger tous les Pac Gommes, sans être mangée par les fantômes. Pour ce faire, nous avons utilisé l'algorithme du plus court chemin. Le principe de cet algorithme est de donner un poids infini à toutes les cases où on peut aller. Puis, on calcule le poids entre la case de départ et les autres cases du plateau.Si le poids d'une case est supérieur à (la distance entre cette case et celle de départ + le poids de la case de départ) alors on donne à cette case cette dernière valeur.On va ensuite considérer comme nouvelle case de départ, la case ayant le plus petit poids dans les cases qui n'ont pas encore été choisies comme case de départ. On va répéter ce processus jusqu'à ce que l'ensemble des cases du plateau soient choisies comme cases de départ.

Par exemple, sur le schéma ci-dessous, on donne au départ un poids infini à tous les sommets hormis notre sommet d'origine 1(valeur en rouge). Puis, on calcul les poids des distances entre notre sommet 1 et les autres sommets. Pour les sommets 6,3 et 2, ce poids va changer. En effet, par exemple pour le sommet 3, le poids du sommet 3 est infini, ce qui est plus grand que le poids du sommet 1 (qui est de 0 car c'est notre sommet d'origine) plus la distance entre le sommet 1 et 3 qui est de 9. Le nouveau poids du sommet 3 va donc être de 0+9 =9. De la même manière, les poids des sommets 6 et 2 vont alors être

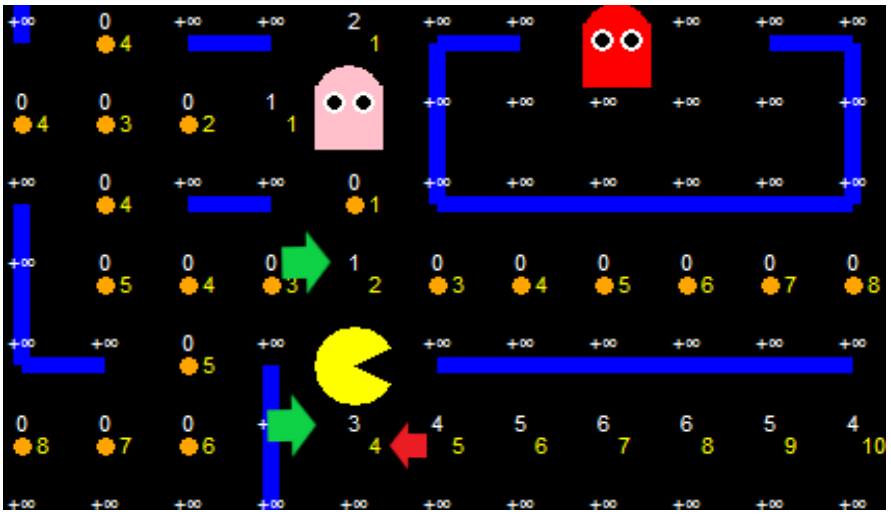
respectivement quatorze et sept. Par contre , les poids des sommets 5 et 4 vont être inchangés car on ne connaît pas la distance entre le sommet 5 et 1 ou entre le sommet 4 et 1. On définit ensuite le sommet 2 comme nouveau sommet . On remarque alors que le poids du sommet 3 reste inchangé car celui-ci est de neuf ce qui est inférieure à la distance entre 3 et (2 + le poids du sommet 2). Par contre, le sommet 4 est modifié car son poids est infini ce qui est supérieur à la distance entre 4 et 2 qui est de 15 plus le poids de 2 qui est de sept. Le nouveau poids de 4 va donc être 7+15=22.





Grâce à l'algorithme vu précédemment ,nous avons généré deux cartes de distance, une pour la distance avec les fantômes et l'autre pour la distance avec le prochain pac-gomme . On regarde alors les emplacements jouables et on se dirige vers la valeur de la carte des

pac-gommes la plus petite. Cependant, on regarde aussi la carte de distance avec les fantômes et, si la distance entre le pacman et un des fantômes est trop petite, pacman se dirigera vers la valeur la plus élevée de la carte de distance afin de s'éloigner des fantômes.



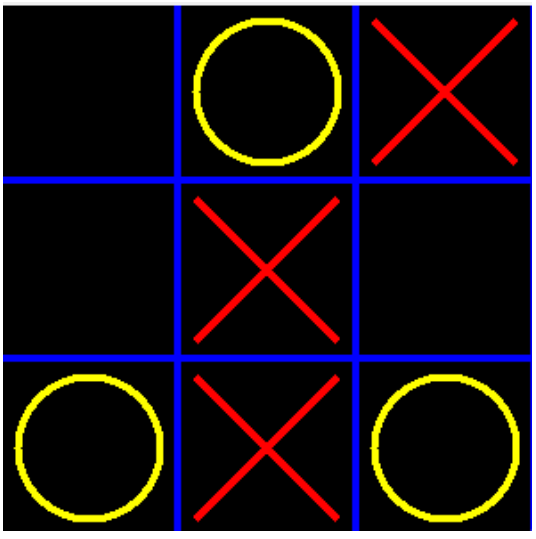
Sur cette image, nous avons indiqué avec des flèches vertes, les valeurs correspondantes à la carte des distances de pacman avec les pac-gommes. On remarque alors que pacman va se diriger vers le nord car la valeur est plus faible.

Les valeurs dorées représentent la carte des distances entre les fantômes et pacman. Les fantômes n'étant pas assez proche de pacman, celui-ci va donc les ignorer et chercher à s'approcher des pac-gommes au lieu de fuir les fantômes.

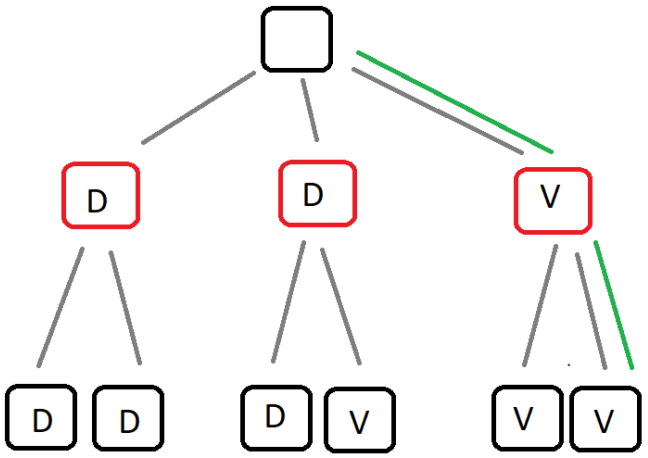
L'ia du fantôme est basique. Lorsque le fantôme arrive à une intersection, on génère un nombre aléatoire pour choisir le chemin qu'il va prendre. Sinon lorsque le fantôme est dans un couloir ou dans un coin, il continue sa route normalement.

B) Morpion et Puissance 4

1. Morpion



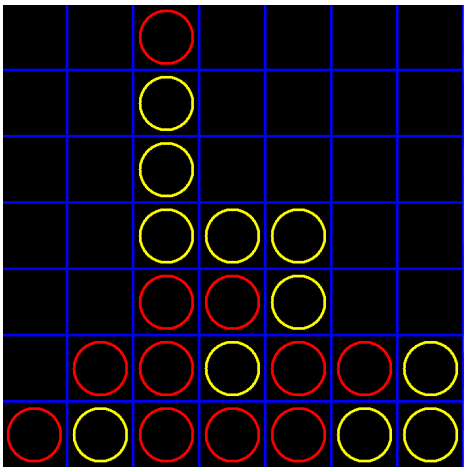
On a conçu une IA capable de jouer au morpion contre un joueur et gagner si possible.
Pour cela, on a utilisé l'algorithme **MiniMax**, son fonctionnement est simple . L'algorithme va simuler tous les coups possibles et prendre celui qui aura le meilleur résultat possible en fonction du joueur. L'algorithme va simuler les coups de l'IA mais aussi du joueur pour lui permettre de prendre le meilleur chemin.



Prenons un exemple simple où une partie de morpion touche à sa fin, le premier carré noir représente le coup que l'IA va jouer. L'algorithme va alors simuler tous les chemins possibles et renvoyer le résultat . On remarque que le premier embranchement va l'amener à une défaite à coup sûr et le second aussi si l'adversaire joue bien, l'IA va donc choisir le troisième embranchement qui est le chemin le plus favorable car elle est obligée de gagner.

2. Puissance 4

Le principe de l'algorithme pour le jeu du puissance 4 est essentiellement le même que celui du Morpion à la différence qu'il y aura beaucoup plus de chemin possible. L'ordinateur ne pourra donc pas calculer toutes les possibilités à temps. On impose un nombre de profondeur qui va permettre de limiter le nombre de chemins que l'algorithme va parcourir.



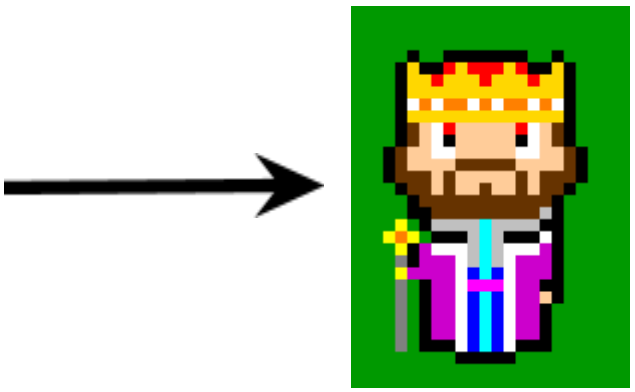
Jeu d’échecs

A) La base du programme

Contrairement aux autres algorithmes que nous avons codés en python, le jeu d'échecs a été codé en C++. En effet, le morpion , le puissance 4 et le pacman étant des entraînements, ils ne nécessitent pas beaucoup de ressource. Il est donc plus facile de les coder en Python. Cependant, pour les échecs, nous avions besoin que le programme soit le plus rapide possible, ce qui est le cas avec le C++ qui est un langage de bas niveau.

La partie graphique est faite avec la librairie G2D. Celle-ci nous permet de créer toutes les textures souhaitées en transformant des string en texture. En effet, chaque lettre ou symbole représente une couleur particulière que l’on définit dans la librairie. G2D va donc nous permettre de convertir une string en succession de pixels de couleurs, donnant ainsi la possibilité de créer nos propres dessins et textures.

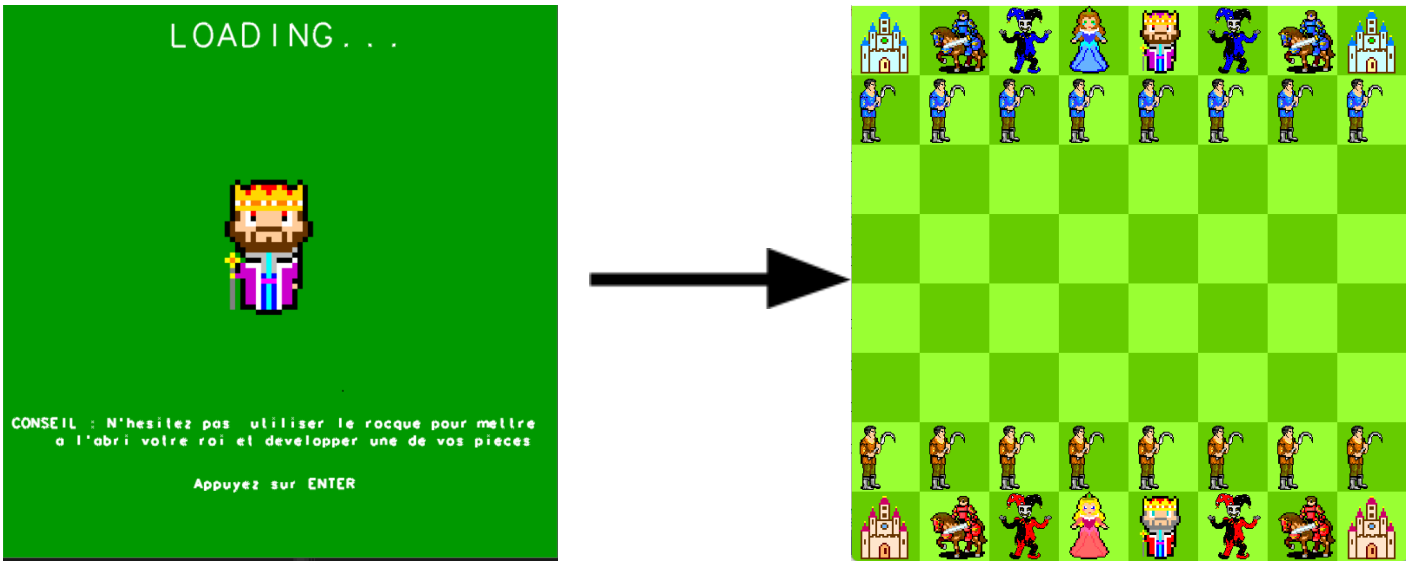
```
string Roi =
    "[  K  KKKKKKK  K  ]"
    "[ KOKKRORRRORKKOK ]"
    "[ KOOROOOROOOROOK ]"
    "[ KOOOOOOOOOOOOK ]"
    "[ KWAWAAWAWAAWAWK ]"
    "[ KOOOOOOOOOOOOK ]"
    "[ KJTWRTTTTTRWTJK ]"
    "[ KJTWKTTTTTKWTJK ]"
    "[KJTTWWTTTTTWWTTJK]"
    "[KJTTTTJJJJJTTTTJK]"
    "[KJJJTJTTTTTJTJJJK]"
    "[ KJJTJTTJTTJTJJK ]"
    "[  KJJJJJJJJJJJK  ]"
    "[  KKJJJJJJJJJSK  ]"
    "[ Y KSSSSCSSSSKK  ]"
    "[YAY KKKSCSKKKWK  ]"
    "[ YKPWWWSCSWWWPK  ]"
    "[ GKPPPWSCSWPPPK  ]"
    "[ YPPPPWBCBWPPPK  ]"
    "[ G KPPWMMMWPPKK  ]"
    "[ G KPPWBCBWPPTK  ]"
    "[ G KPPWBCBWPPK   ]"
    "[ G KPPWBCBWPPK   ]"
    "[ G KPPWBCBWPPK   ]"
    "[ G KKKWBCBWKKK   ]"
    "[      KKKKK      ]";
```



Il existe aussi des fonctions interne à G2D permettant de dessiner des carrés, des cercles ou des phrases sur l'écran.

Quant à la gestion des différents menus, nous avons utilisé des constantes, chacunes associées à un écran. Lors du lancement, une fonction principale est appelée qui redirige sur la fonction liée à l'écran courant. Cela fonctionne pour la partie graphique et pour la partie interne au programme. Selon certaines actions du joueur, la valeur de l'écran courant change, ce qui change l'écran. On a créé 7 écrans, allant de l'écran d'accueil à l’écran de victoire.

Toutes les données relatives à une pièce comme sa position, sa couleur, sa texture et son état sont stockées dans des objets. De plus, un plateau est généré, l'information principale de ce plateau est une string. Cette string représente toutes les cases du plateau et à chaque case est associée un nombre, en fonction de la couleur de la pièce s'y situant. On mettra donc un '0' si la case est vide, un '1' s'il y a une pièce blanche sur la case, et '2' si c'est une pièce noire. Ce plateau va donc changer tout au long de la partie et va permettre de vérifier la légalité d'un coup.Les pièces sont régénérées à leurs états initiaux à chaque début de partie.

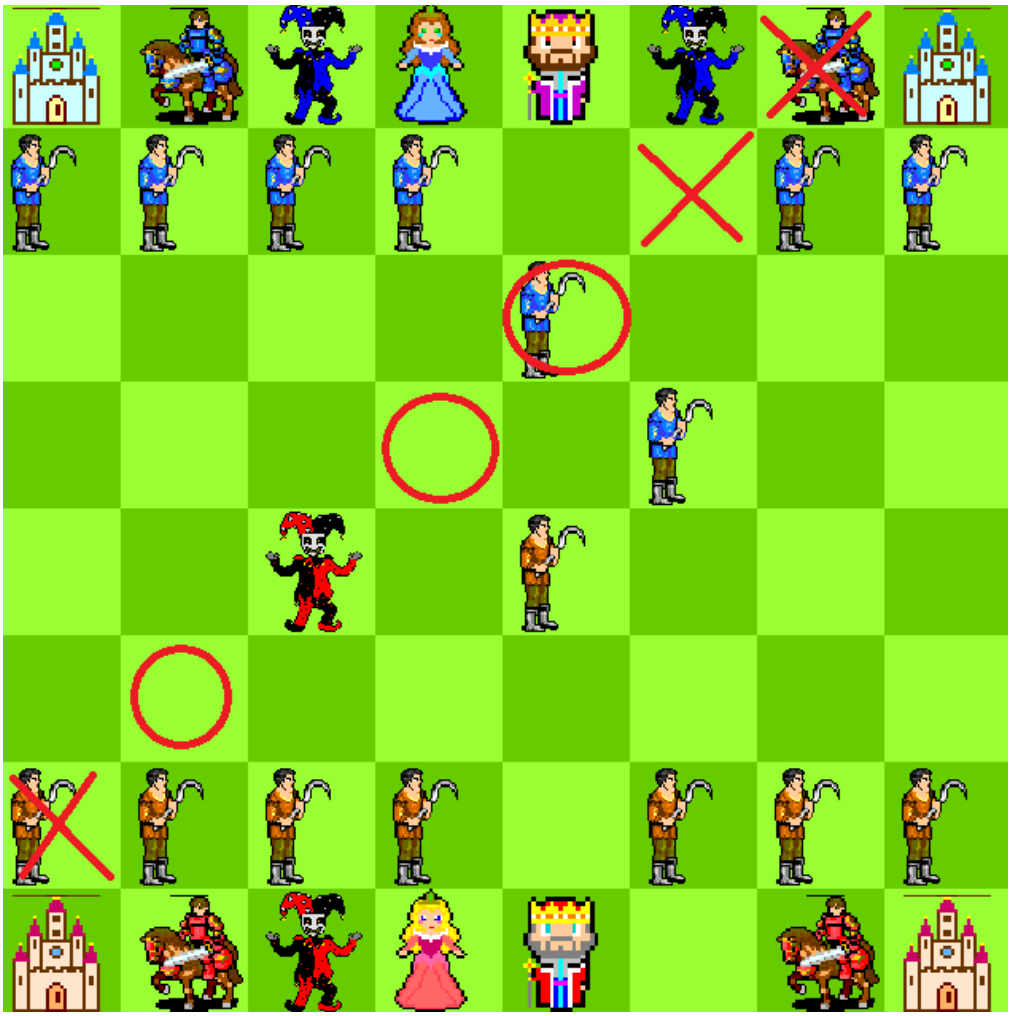


Passage de l'écran de chargement à l'écran de jeu

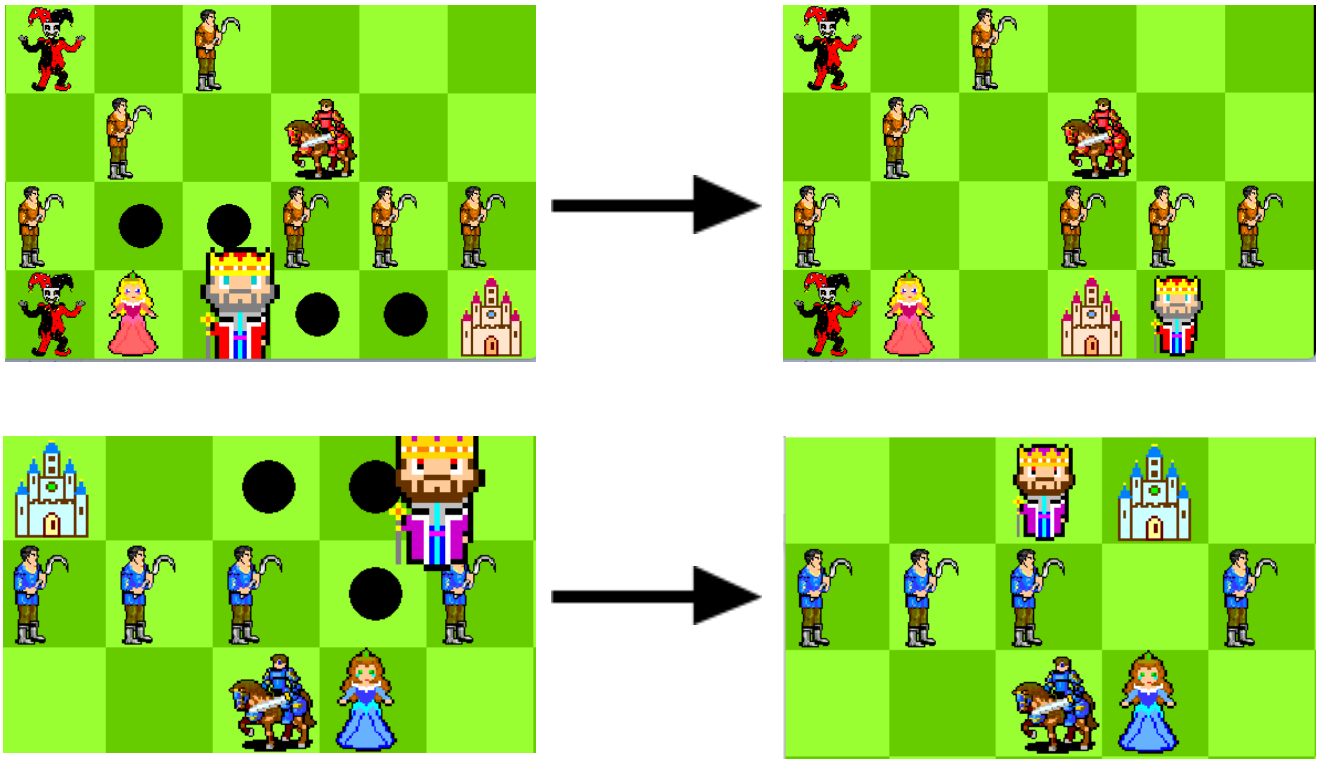
Ecran de Victoire pour les blancs/PAT/Victoire pour les noirs

B) Déplacement

Nous avons créé une fonction booléenne qui prend en paramètre une pièce du jeu d'échecs et une position sur le plateau. La fonction va ensuite récupérer le type de la pièce (tour, pion, cavalier…) et en fonction de celui-ci , va vérifier si le déplacement est autorisé. Par exemple, pour le déplacement d'un fou, l'algorithme va vérifier si le déplacement est bien en diagonale, puis va le tester case par case. Pour cela, la fonction va récupérer le plateau avec la position des cases occupées. Si la case est un '0', alors la case est vide et donc la pièce d'échec peut se déplacer sur cette case. Sinon si la case est un '1' (il existe une pièce blanche dans la case) ou '2' (il existe une pièce noir dans la case) alors si la piece que l'on souhaite déplacer est de la meme couleur, le coup est interdit. Si la pièce d'echecs est de couleur opposée, le coup est autorisé seulement si l'on souhaite s'arrêter sur cette case. Si le déplacement est autorisé sur l'ensemble de la simulation , alors la fonction va retourner true, indiquant que le coup respecte bien les règles des échecs. Sinon, la fonction retournera false et le coup sera refusé.



On a ajouté la détection du roque dans notre fonction de déplacement. Pour le faire fonctionner, on a ajouté un attribut à chaque pièce qui vérifie si celle-ci n'a jamais été déplacée. Pour que le roque soit valide, on vérifie si la tour et le roi n'ont jamais été bougés.Si c'est le cas, on vérifie ensuite que le roque ne met pas en danger le roi.Enfin si le roque est valide, on renvoie true et notre fonction qui s'occupe de déplacer nos pièces, va appliquer le déplacement des deux pièces.

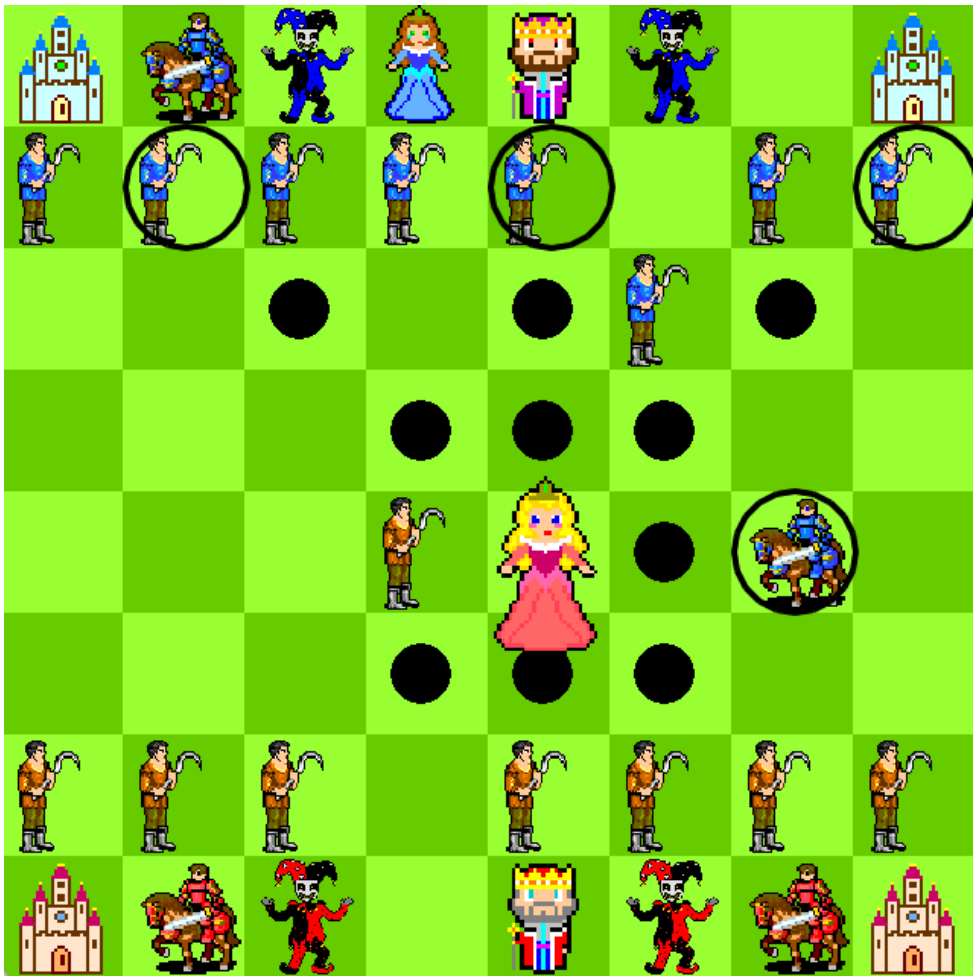


Cette fonction est par la suite utilisée dans de nombreuses parties du programme, que cela soit pour les tests de l'IA ou pour les déplacements du joueur.

Pour la gestion des déplacements du joueur, nous avons opté pour un système de glisser-déposer, c'est-à-dire que l'on peut cliquer sur une pièce, puis la déposer dans la case où on souhaite la jouer. On a donc utilisé l'interface graphique Glut qui va nous permettre de détecter les clics de souris et la position de celle-ci en temps réel. Glut permet aussi de détecter les touches du clavier . Lorsque le joueur va sélectionner une pièce avec sa souris, on va garder en mémoire la pièce sélectionnée si celle-ci est de la couleur du joueur. Ensuite, on va traquer la souris grâce à Glut, afin de pouvoir faire déplacer le sprite de la pièce sélectionnée avec la souris. Enfin, lorsque le joueur relâche le clic de la souris, on va récupérer les coordonnées de la souris et lancer la fonction qui vérifie si le coup est autorisé. Si cette dernière renvoie true, on modifie les coordonnées de la pièce avec ses nouvelles coordonnées.

Afin de rendre le jeu plus compréhensible pour des joueurs débutants, nous avons décidé de créer une fonction permettant d'indiquer au joueur les cases où il peut jouer. Pour ce faire, nous avons fabriqué une fonction qui prend en paramètre la pièce que l'on souhaite déplacer. La fonction va ensuite simuler le déplacement de la pièce sur l'ensemble des cases du plateau et enregistrer dans un vecteur tous les coups autorisés.

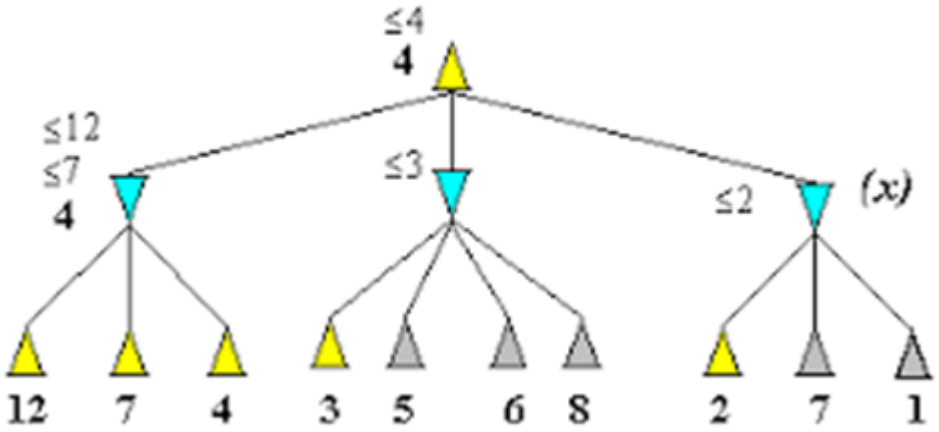
Notre fonction qui s'occupe des affichages du jeu va ensuite récupérer ce vecteur et dessiner des indications permettant au joueur de savoir où il peut poser sa pièce et quelles sont les pièces qu'il peut manger. Si notre joueur est en échec, les coups ne permettant pas de protéger le roi ne seront donc ni proposés, ni autorisés.



Pour vérifier l’issue d’une partie, on utilise une fonction qui génère la liste des coups possibles du joueur actuel. Si cette liste est vide, on regarde alors si l’autre joueur le met en position d'échecs. Si c’est le cas, ce dernier est alors déclaré vainqueur. Si l’adversaire ne le met pas en échec alors la partie est considérée comme PAT. On affiche alors le résultat sur l'écran de jeu avant de renvoyer l'écran de victoire correspondant.

C) L’algorithme

Pour faire jouer l’IA aux échecs, nous avons codé l’algorithme minimax avec élagage alpha-beta. Cet algorithme consiste à explorer l’arbre des possibilités de chaque coup disponible. L’algorithme fait jouer chacun leur tour les blancs, puis les noirs et ainsi de suite, jusqu’à arriver en bout d’arbre. A ce moment, on évalue le résultat de l’échiquier, puis, à tour de rôle, l’algorithme fait remonter l'arbre pour qu’à chaque nœud, le joueur prenne le résultat qui lui est le plus favorable. Retourner en haut de l'arbre, l'algorithme sait maintenant quel coup jouer pour gagner. A cela, on ajoute l’élagage alpha-beta, pour réduire le nombre de branches à regarder. L’élagage consiste à créer un intervalle, afin d’omettre certaines branches, qui sont déjà jugées avec un trop bon score ou un trop mauvais score. Pour cela, à chaque fois qu’un score est donné , il devient une des bornes de l’intervalle alpha-beta. Ensuite, lors de l’exploration de l’arbre si on obtient un score qui est en dehors de l’intervalle, on supprime la branche correspondante ainsi que toutes les branches secondaires.



Ceci est un schéma qui représente l'alpha-beta. On peut apercevoir qu’après la première branche, le coup sera de minimum quatre, alors que dans la deuxième branche, à la première feuille, le score est de trois. On peut alors directement arrêter de calculer pour les autres feuilles, car de toute façon, le coût sera inférieur à trois ce qui est déjà moins bien que son premier choix. Et il se passe la même chose pour la dernière branche.

Avant de commencer à coder en C++, nous avons créé une esquisse de l’algorithme alpha-beta en python. Cet algorithme a été conçu grâce à la librairie chess qui contenait déjà toutes les fonctions nécessaires que ce soit pour la liste des coups légaux ou pour la gestion de fin de partie.

Dans notre jeu d'échecs, nous avons recréé cet algorithme grâce à une fonction, qui, tant qu'elle n'est pas arrivée en fin d'arbre, fait appliquer l'algorithme à la liste des coups possibles. Cette liste est générée grâce à une seconde fonction, qui pour chaque pièce regarde les coups autorisés et les stocke. Lorsque le programme arrive au bout d'une branche, on appelle une fonction qui renvoie le score de l'échiquier . Ce score est obtenu en donnant une valeur à chaque pièce, afin de pousser l'IA à garder ses pièces en vie et à manger les pièces de l'adversaire .A cela, on ajoute un bonus si l'ia occupe les case centrales ,qui sont primordiales aux échecs, couplé à un augmentation de point pour chaque case jouable afin de l'inciter à sortir ses pièces. Enfin, lorsque son adversaire a moins de 12 pièces, on additionne le nombre de points a une fonction inverse ,qui prend le nombre de coups possibles de l'adversaire en antécédent et qui cherche à bloquer celui-ci, facilitant ainsi l'échec et mat. Après le calcul des points, on applique l'élagage alpha-beta et on remonte l'algorithme jusqu'au début où on peut enfin appliquer le coup final. Pour changer l'IA afin que celle-ci joue aussi bien les blanc que les noir, on a créé une condition qui nous donne la couleur de l'Ia. En fonction de celle-ci, on inverse ou non notre algorithme de calcul de points, afin d'optimiser son coup .

Cependant, malgré l'élagage alpha-beta, le nombre de possibilités reste trop élevé pour être utilisé comme tel. On a alors introduit un système de profondeur afin d'arriver plus rapidement à la fin de l'arbre. Cette profondeur est décrémentée à chaque appel de la fonction réflexive.Et lorsqu'on arrive à la profondeur maximum, on fait remonter prématurément le score de l'échiquier, contrairement au programme ne possédant pas l'étalage.

A chaque changement de position des pièces pendant l'algorithme, la string du plateau est changée temporairement pour pouvoir appeler la fonction vérifiant la légalité d'un coup. Afin de changer plus facilement le plateau dans son état d'origine, chaque coup est stocké en temps qu'un objet contenant la position finale de la position de début l'index de la pièce et son dernier paramètre lui permet de savoir si la pièce est un pion transformer afin qu'il revienne à son état initial. Il suffit donc de donner à la pièce la position de début du coup pour qu'elle retourne où elle était. Lors du retour en arrière, on réactualise le plateau pour ne pas changer les informations de celui-ci avant l'appel de l'IA.

Conclusion

Ce projet nous a ainsi permis de découvrir des algorithmes incontournables à la programmation d'IA pour des jeux vidéo. De plus, nous avons pu apprendre et comprendre mieux le principe d'optimisation d'un programme.En effet, dans un premier temps, nous avions décidé de recréer chaque image de nos pièces à chaque itération de notre code, provoquant de nombreuses pertes de mémoire. En modifiant le programme pour créer chaque image qu'une seule fois lors de la phase d'initialisation, nous avons pu supprimer ce problème de mémoire et ainsi faire fonctionner notre jeu de façon plus optimale.

Pour aller plus loin et ainsi permettre une plus grande capacité de calcul à notre IA, il pourrait être intéressant de remplacer notre algorithme par un réseau de neurones. Celui-ci étant bien plus performant et rapide. A plus petite mesure, on pourrait utiliser une descente du gradient afin d'avoir un score plus significatif. En effet, celle-ci permettrait d'optimiser les poids des différents composants du score.

Source

https://github.com/wrobpierre/Cpp-Echec

https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta
Source Image:
https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra