TP C#13
Game of Tron

1 Consignes de rendu

À la fin de ce TP, vous devrez soumettre une archive respectant l'architecture suivante :

```
rendu-tp-prenom.nom.zip
    |-- rendu-tp-firstname.lastname/
        |-- AUTHORS
        |-- README
        |-- GameOfTron/
            |-- GameOfTron.sln
            |-- GameOfTron/
                |-- Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer *prenom.nom* par votre propre login.

N'oubliez pas de vérifier les points suivants avant de rendre votre TP :

- Le fichier AUTHORS doit être au format habituel : ** prenom.nom$* où le caractère '$' représente un retour à la ligne.
- Vous devez suivre le sujet **scrupuleusement**, et notamment **respecter les prototypes** des différentes méthodes.
- Pas de dossiers bin et/ou obj dans le projet.
- **Votre code doit compiler !**

2 Déroulement de ce TP

Ce TP porte sur la réalisation d'une intelligence artificielle sur le jeu que vous avez codé avant les vacances : Tron. Ne vous inquiétez pas si vous n'avez pas réussi le TP12, les deux TPs ont seulement un thème commun, ils restent entièrement distincts sur l'implémentation et la notation.

Lisez *tout* le TP avant de toucher au code.

2.1 Fichiers donnés

Une correction du TP12 légèrement modifiée est disponible sur l'intranet des assistants, il sera nécessaire pour tester votre code. Une classe a été ajoutée, héritant de `Input` : `ProgramInput`. Nous vous recommandons fortement d'aller lire cette classe et de comprendre comment elle fonctionne. Pour faire simple, elle exécute un programme et communique avec en passant par l'entrée et la sortie standard (la console). Vous pouvez donc utiliser la sortie d'erreur standard pour des tests.

De manière générale, **vous n'aurez pas besoin de comprendre comment les deux programmes communiquent**. Un début de projet est disponible sur l'intranet, avec le code permettant de communiquer avec le reste du jeu. Mais comme ça peut aider pour le debug, voilà comment la communication entre les deux programmes fonctionne :

```
> [Timeout-ms]\n
> [Width] [Height]\n
> [Self-x] [Self-y]\n
> [Player-number]\n
> [Player1-start-x] [Player1-start-y]\n
> ...

> [Player-number]\n
> [Player1-x] [Player1-y]\n
> ...
< [Output-command]\n
```

Un exemple sur une map de 20x70, avec le joueur que l'on contrôle en (10,10) et l'adversaire en (10, 40), pour un `timeout` de 100ms :

```
> 100\n
> 20 70\n
> 10 10\n
> 2\n
> 10 10\n
> 10 40\n
> 2\n
> 10 10\n
> 10 40\n
< R\n
...
```

Pour utiliser le programme donné, vous devez le lancer en console en donnant l'exécutable de votre IA en paramètre (le *.exe). Un paramètre fait jouer l'IA contre elle-même, deux paramètres les font jouer l'une contre l'autre. Pour jouer vous-mêmes contre l'IA, passez un tiret - en paramètre. Par exemple, dans un shell :

```
./TP12/TP12/bin/Debug/TP12.exe ./TP13/TP13/bin/debug/TP13.exe -
        # Runs the game with your AI as P1 and arrow keys as P2

./TP12/TP12/bin/Debug/TP12.exe ./TP13/TP13/bin/debug/TP13.exe
        # Runs the game with your AI against itself
```

Pour jouer contre l'IA d'un de vos camarades, vous pouvez vous échanger les binaires. Bien sûr, ne vous échangez pas le code.

2.2 Le classement

Un classement en (quasi) temps réel sera mis à disposition sur l'intrachievment [1] des assistants. Pour apparaître dans ce classement, vous avez juste à effectuer un rendu et à battre l'IA de base [2].

Lors d'un duel entre deux IA, vous serez testés avec un certain `timeout`, et dans les deux sens [3]. En cas d'égalité, une belle sera faite *avec un timeout réduit pour les deux joueurs.* Autrement dit, si deux IA ont environ le même niveau, leur capacité à s'adapter à un `timeout` court va faire la différence.

Le classement en lui-même sera à base d'Elo [4]. Le classement sera donc légèrement imprécis et pourra évoluer avec le temps.

2.3 Votre note

Quelques points de votre note seront déterminés par votre classement, mais cela reste une petite partie. Vous serez principalement notés sur la partie obligatoire du sujet. Cependant, elle ne suffit pas pour avoir 20. Selon les bonus choisis, un ou deux bonus sont nécessaires pour une note parfaite.

1. pirates.acdc.epita.it
2. celle donnée dans le projet de base, qui ne fait qu'éviter les murs
3. Chaque IA sera J1 ET J2
4. https ://en.wikipedia.org/wiki/Elo_rating_system

3 Réalisation de votre IA

Votre code sera principalement écrit dans `AI.cs`. Les autres fichiers donnés ne *devraient* pas être modifiés, mais vous êtes libres de le faire si vous pensez que c'est justifié. Vous êtes libres d'ajouter des fichiers pour des bonus.

3.1 Partie obligatoire : le minimax

Cette partie contient le code que vous *devez* implémenter. Votre note dépendra principalement de ces fonctionnalités. Cependant, ne vous arrêtez pas là : une partie obligatoire parfaite ne garantit pas tous les points.

Vous n'êtes pas *obligés* d'utiliser cette section dans le programme que vous allez soumettre pour le classement, mais elle doit quand même apparaître dans le code, et non commentée.

3.1.1 Description de l'algorithme

Le minimax sera le cœur de votre programme. Il s'agit d'un algorithme simple, presque hardcodé dans sa version la plus simple, permettant de réaliser une IA pour un jeu en tour par tour déterministe. Le principe : choisir l'action qui nous avantage le plus pendant notre tour, en estimant que notre adversaire va faire de même. C'est fortement récursif : chaque coup sera évalué en fonction du score des coups qui en découlent. Un parcours minimax est très similaire à un parcours d'arbre : on utilisera donc souvent le vocabulaire associé au cours de ce TP.

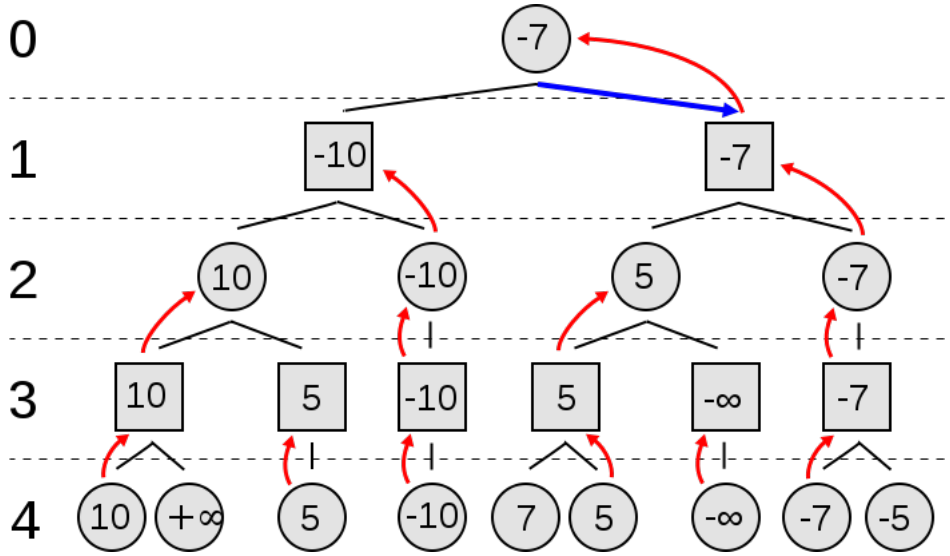
Votre méthode renverra le score de l'état actuel, ainsi que la meilleure action qui en découle. Si le jeu s'est terminé, renvoyez le score directement : $+\infty$ en cas de victoire et $-\infty$ pour une défaite. L'action dans ce cas importe peu. Vous allez à chaque étape sauvegarder l'état de la map et des joueurs, pour effectuer vos tests dessus sans perdre de données. Pour chaque action possible, vous allez appliquer l'action sur la map, et lancer un appel récursif en précisant que le joueur effectuant la prochaine action a changé. Conservez le meilleur score et l'action associée, en faisant attention au joueur : vous allez chercher à maximiser le score, votre adversaire cherche à le minimiser.

Ça reste assez abstrait et peut être difficile à comprendre. N'ayez pas peur, c'est un algorithme simple comparé à ce que vous avez déjà codé. Rien de tel qu'un peu de pseudocode pour bien comprendre un algorithme ! Attention, il n'est pas complet et devra être fortement modifié dans la suite du TP.

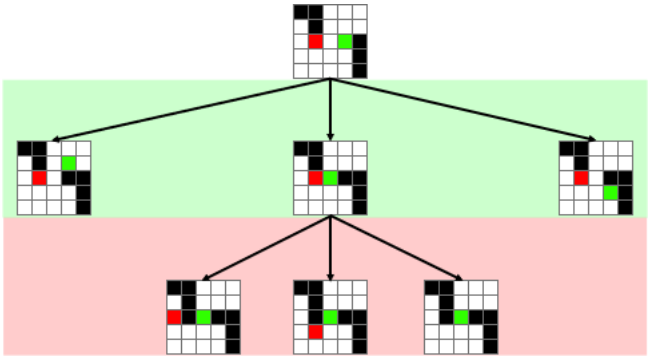
Algorithm 1: Minimax

```
1  function Minimax (maximizing);
   Input  : maximizing whether we maximize this player's score
   Output: Best possible score
2  begin
3      if end then
4          return +∞ or −∞;
5      else
6          if maximizing then
7              bestValue := −∞;
8              forall action do
9                  run(action);
10                 v := Max(bestValue, Minimax(false));
11             end
12         else
13             bestValue := +∞;
14             forall action do
15                 run(action);
16                 v := Min(bestValue, Minimax(true));
17             end
18         end
19         return bestValue;
20     end
21 end
```

Une autre manière de visualiser ça, avec un arbre. Les ronds représentent le joueur exécutant l'algorithme (max), les carrés l'adversaire (min). Tous les tests sont effectués après les appels récursifs : lisez l'arbre du bas vers le haut.



Un autre exemple sur le jeu : chaque branchement correspond à une action possible. Si les couleurs rendent mal sur le sujet papier, allez le chercher sur l'intra.



Vous devez suivre le prototype suivant. Vous comprendrez tous les arguments en lisant la suite du sujet, ce que vous allez évidemment faire avant de commencer à coder (n'est-ce pas ?)

```
1 private Tuple<float, Map.Action>
2 Minimax(Map map, int maxDepth, float alpha, float beta, bool ownTurn);
```

Vous devrez essayer de factoriser votre code au maximum. Évitez de juste copier-coller votre code d'un cas à l'autre : vous risquez de vous retrouver avec une fonction de 200 lignes après vos ajouts. Essayez au maximum d'écrire une seule fois les lignes qui apparaissent plusieurs fois à l'identique dans l'algorithme.

3.1.2 Fonction d'évaluation

Le minimax basique tel qu'on l'a présenté, c'est bien sympa, mais c'est bien trop lent. Vous avez sans doute remarqué que la complexité est exponentielle : il n'est pas question de tester toutes les possibilités comme cela. Vous devez obligatoirement limiter la profondeur de vos récursions, et utiliser une heuristique [5] pour estimer la valeur d'un état.

Votre heuristique sera la fonction `private float Evaluation()`. Vous êtes entièrement libres sur le contenu de cette fonction, mais vous devez renvoyer une valeur plus élevée sur des situations avantageuses pour vous.

Quelques suggestions sur les paramètres à évaluer : la surface accessible par rapport à la surface libre totale, la position des joueurs par rapport aux murs et aux autres joueurs, etc. La fonction doit être rapide. Ne vous limitez pas à votre propre joueur, regardez également l'état des autres. Vous serez testés en comparant les scores de différentes situations simples.

3.1.3 alpha-beta pruning

On va commencer à rendre notre Minimax un minimum intelligent, au lieu de faire une rechercher de graphe bête et brutale. L'`alpha-beta pruning` est une modification de l'algorithme `minimax`, qui a pour but d'éviter de parcourir des branches inutiles de l'arbre.

Dans quel cas peut-on estimer qu'une branche ne vaut pas la peine d'être explorée ? Prenons un exemple. On commence par tester un déplacement vers le haut. On explore cette branche en entier, et on se rend compte que le résultat est neutre. On reste dans une position à peu près équivalente à la position précédente. Maintenant, on continue de tester les différentes possibilités : vers le bas. On teste chacune des actions de l'adversaire, mais dès la première on réalise qu'il peut instantanément nous faire perdre. On peut directement dire que nous ne devons pas nous déplacer vers le bas : dans tous les cas, se déplacer vers le haut sera mieux. Ça n'a pas d'importance de savoir de combien le déplacement bas est pire que le haut, et on n'a pas besoin de tester toutes les autres actions de l'adversaire. Une seule a suffi pour se rendre compte que l'action est mauvaise.

D'un point de vue algorithmique, qu'est-ce que ça représente ? Les deux joueurs ont des valeurs extrêmes, la meilleure valeur qu'il peut atteindre, qui représente le seuil à ne pas dépasser. Nommons ces valeurs, de manière complètement arbitraire, α pour le joueur à maximiser et β pour celui à minimiser. α contient la valeur minimale de score à atteindre, et β la valeur maximale. Si un état sort de ces limites, on ne prend même pas la peine de continuer à explorer ses fils.

Au niveau du code, vous devez utiliser les paramètres `alpha` et `beta` du prototype de `Minimax`. Vos résultats ne doivent pas changer du tout, seul le temps de calcul sera modifié (accéléré, normalement). Vous serez testés sur cette partie, même si votre résultat ne change pas.

5. Méthode de calcul donnant rapidement une approximation [6]
6. Merci Wikipedia

3.2 Améliorations possibles

À partir d'ici, vous devez chercher à améliorer votre IA. Plusieurs méthodes s'offrent à vous. On vous a préparé une petite liste de pistes à suivre. Vous pouvez bien sûr implémenter vos propres améliorations, mais *vous devez dans ce cas le détailler dans le README.* Décrivez de toute façon vos bonus dans le README, mais soyez particulièrement descriptifs s'ils ne sont pas dans cette liste. Nous vous recommandons fortement de lire *toute* cette section, même si vous n'avez pas l'intention de tout implémenter. Juste parce que ces algos sont intéressants.

Notez que les algorithmes et méthodes présentés ici n'apportent pas forcément d'amélioration significative sur les performance de votre IA. Nous avons choisis des méthodes que nous estimons être intéressantes, sans vraiment se soucier de son efficacité sur Tron. Choisissez bien.

3.2.1 Iterative deepening

Comme vous devez le savoir, le `timeout` de votre programme est variable. Nous allons même appeler votre programme avec différentes valeurs si la situation nous semble appropriée. Mais alors, comment s'assurer de ne pas dépasser ce `timeout`, tout en profitant au maximum du temps accordé ? En utilisant un nombre de récursions fixe et très faible ? En limitant les récursions à coup de divisions sur le `timeout` ? Et si on lance nos tests sur un PC affreusement lent ?

Une des solutions est l'*iterative deepening.* Une méthode à première vue inefficace, mais qui remplit bien les objectifs. Vous commencez à lancer votre minimax avec une profondeur très faible, qui est sûre de marcher : à tout hasard, 1. Ça a marché ? Bien, enregistrez le résultat et réessayez avec 2. Ça marche encore, super, on remplace l'ancien résultat. Vient un point où l'on a plus le temps : on stoppe tout, et on renvoie le dernier résultat récupéré. On se retrouve donc à faire plusieurs fois les mêmes calculs, mais on s'assure d'utiliser le temps donné sans dépasser. Des optimisations peuvent encore être faites, cf. les deux prochaines parties.

À vous de trouver une bonne méthode pour interrompre votre minimax sans risquer de dépasser le `timeout`.

3.2.2 Ordre des actions à évaluer

Dans un Tron, vous avez le choix à chaque noeud entre 4 actions. C'est peu, mais ça ne veut pas dire que vous devez les évaluer dans n'importe quel ordre. Vous vous souvenez de l'`alpha-beta pruning` ? "J'évalue pas cette branche, j'ai déjà trouvé mieux". Comment améliore-t-on ça ? Vous l'avez sans doute deviné, en évaluant les meilleures branches en premier.

Pour ce faire, une nouvelle possibilité de coder des petites heuristiques. À vous de tester et trouver ce qui marche. Vous gagnerez en performance du moment que vous testez les meilleurs coups plus souvent au début qu'à la fin de votre boucle. Préférez cependant la vitesse de calcul plutôt que la précision.

3.2.3 Réutilisation des résultats précédents

De manière générale, lors d'un tour les deux joueurs suivent les meilleures actions que vous avez calculées lors du `minimax`. Vous avez donc normalement déjà eu une petite idée de ce qui se passe ensuite. Vous pouvez réutiliser ces résultats, en particulier sur l'ordre des actions à effectuer (cf. section précédente). Vous pouvez également améliorer votre `iterative deepening` de la même manière, en réutilisant les résultats d'une itération à l'autre (cf. 3.2.1).

Sur un jeu plus complexe comme les échecs, on aurait pu se retrouver à parcourir deux branches identiques issues des chemins différents. Il y a des techniques sympa pour conserver les états parcourus juste assez pour ne pas les faire deux fois, à base de hachage. C'est inutile ici, mais c'est toujours intéressant.

3.2.4 Optimisation de la mémoire

Vous avez probablement codé votre `minimax` en copiant toute la map et tous les joueurs entre chaque action, pour pouvoir revenir en arrière.

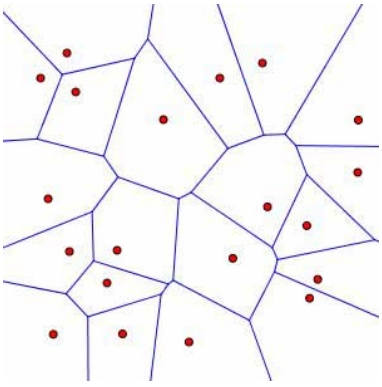
C'est lent. C'est sale.

Vous pouvez faire mieux. Vous pouvez par exemple enregistrer entre chaque étape, non pas l'entièreté de la map, mais seulement l'action à effectuer pour revenir dans l'état initial. Moins de copie, meilleure optimisation mémoire, le code est efficace et écologique.

Si vous voulez vous amuser encore plus, vous pouvez faire ça de manière très élégante en passant par des objets.

3.2.5 Diagramme de Vornoï

Un diagramme de Vornoï [7] est un partitionnement de plan en différentes régions basées sur des points appelés gemmes. Chaque gemme est entourée d'une région représentant l'ensemble des points plus proche de la gemme en question que d'une autre. Ce n'est pas un concept facilement décrit en texte, voici donc un exemple :



Dans notre cas, ces régions peuvent représenter les zones contrôlées par chaque joueur. Cela peut permettre de faire des fonctions d'évaluation efficaces.

7. https ://en.wikipedia.org/wiki/Voronoi_diagram

3.2.6 Parallélisme

Le minimax a une propriété intéressante : le résultat de chaque branche ne dépend pas des autres (mis à part pour l'α-β, on va y revenir). Cela offre une possibilité intéressante : calculer chaque branche en même temps. Pour ce faire, vous pouvez utiliser des `threads`[8]. Vous n'en avez pas encore utilisé dans nos TPs, alors : qu'est-ce qu'un `thread` ? C'est une manière d'exécuter du code en parallèle, potentiellement sur plusieurs cœurs en même temps. Vous allez lancer une méthode en arrière-plan, sans interrompre la méthode courante.

C'est relativement dangereux à utiliser : si vous avez plusieurs lectures ou écritures en même temps sur la même ressource, vous pouvez avoir des résultats inattendus. Les ressources pouvant être des variables partagées, des fichiers, etc. Par exemple, si on garde une valeur maximale sur plusieurs `threads`. Un premier `thread` compare le maximum avec sa valeur, voit que la nouvelle valeur est plus grande, et l'écrit dans la variable max. Entre le test et l'écriture, un autre `thread` est venu se glisser pour écrire son propre nouveau maximum. La première valeur écrite est écrasée, qu'elle soit ou non le maximum.

Vous pouvez utiliser le mot-clé `lock`[9] pour éviter ce problème. Il permet de verrouiller un objet, le code dans le bloc qui suit ne sera jamais exécuté sur plus d'un `thread`. Les `mutex`[10] sont une autre solution, plus lourde mais plus flexible. Voici un exemple de code sur les `threads` et `lock`, n'hésitez pas à regarder les exemples sur MSDN si ça ne suffit pas :

```
1 Thread newThread = new Thread(methodeAExecuter);
2 newThread.Start();      // Lance le thread sur methodeAExecuter
3 lock(objetPartagé)
4 {
5         // Manipulation des ressources partagées,
6         // avec la garantie qu'on est le seul thread dessus
7 }
8 newThread.Join()        // Attend que le nouveau thread se termine
9                         // pour s'assurer qu'il ait fini ses calculs
```

Cependant, faites attention : si vous lancez un `thread` sur chaque appel récursif à chaque nœud de l'arbre, vous allez vous retrouver avec des milliers de `threads`. C'est pas une bonne idée : créer un `thread` est coûteux, et les bénéfices diminuent fortement quand on en a beaucoup. Nous vous *imposons* une limite (arbitraire) de 16 `threads`. Il y a aussi le problème de l'α-β, qui ne fonctionne pas avec du parallélisme. Des méthodes propres[11] existent pour ce problème, mais pour l'instant contentez-vous de créer des nouveaux `threads` dans les premiers niveaux de récursions, niveaux où vous n'aurez pas vraiment d'α-β.

8. https ://msdn.microsoft.com/fr-fr/library/system.threading.thread.aspx
9. https ://msdn.microsoft.com/fr-fr/library/c5kehkcz.aspx
10. https ://msdn.microsoft.com/fr-fr/library/system.threading.mutex(v=vs.110).aspx
11. "Young brother wait"

3.2.7 Plus de deux joueurs

Ça me semble assez explicite. On ne vous testera pas directement là dessus, mais c'est toujours intéressant à coder. Vous ne devez faire que quelques modifications mineures sur le minimax, notamment sur le booléen `ownTurn` (qui, comme vous l'avez sans doute deviné, sera légèrement moins booléen). Chaque joueur autre que vous-même cherchera à minimiser la fonction d'évaluation. Vous pouvez aller encore plus loin que ça, mais c'est un bon début.

3.2.8 N'essayez même pas

Réseau de neurones. Bon, là on attaque des trucs sérieux. Donc je le répète : *ne faites pas ça*, à moins d'avoir déjà fait tout le reste et d'avoir pas mal de temps libre devant vous. Vous n'aurez sans doute pas le temps de le faire, mais dans le pire des cas vous aurez appris pas mal de choses. Si vous voulez faire ça, vous êtes à un point où vous n'avez pas vraiment besoin de notre aide, donc je vais me contenter de vous envoyer vers la page wikipédia [12] , vous pouvez aussi faire des bonnes recherches google. Allez faire un tour dessus même si vous ne comptez pas essayer de coder ça, c'est un sujet passionnant.

Dans une IA à base de minimax, un réseau de neurones servirait à remplacer / améliorer votre fonction d'évaluation.

Maintenant, à vous de jouer. Impressionnez nous, on a hâte de se faire battre par des sups.

The Code is the Law.

12. https ://en.wikipedia.org/wiki/Artificial_neural_network