Pierre GOUTH

Emilien LAMBERT

Rapport projet IA

1. Affichez à chaque coup de l’ordinateur, le nombre de simulations réalisées pour calculer ce coup et une estimation de la probabilité de victoire pour l’ordinateur.

Voir exécution du programme.

2. Testez différentes limites de temps pour l’ordinateur et comparez les résultats obtenus. A partir de quel temps de calcul l’ordinateur vous bat à tous les coups ?

A partir de 1 seconde l’ordinateur nous bat à tous les coups. En effet, nous avons essayé de faire jouer l’IA contre elle-même, pour qu’elle est un adversaire jouant des coups optimaux, et nous avons remarqué qu’entre 1 seconde ou 2, les choix sont identiques.

3. Implémentez l’amélioration des simulations consistant à toujours choisir un coup gagnant lorsque cela est possible. Comparez la qualité de jeu de cette nouvelle version avec la précédente et expliquez à quoi cela est dû

Insérer ici observation après ajout de l’amélioration :

Avec opti et -O3 : Nombre de simu total effectue : 314867 proba de gagner : 0.475629

Sans opti et -O3 : Nombre de simu total effectue : 501120 proba de gagner : 0.453676

Cela est dû au fait qu’on réduit la profondeur de l’arbre (de 1 au minimum).

Ceci s’explique par le fait que nous élaguons l’arbre en se basant que le fait que, jouer un coup gagnant directement sera toujours ce que fera l’IA de toute façon et que donc parcourir l’arbre à partir des autres nœuds est inutile.

Plus précisément, à partir d’une profondeur de 6, potentiellement, les premières possibilités d’avoir un nœud gagnant apparaissent. Il serait donc bête de simuler des coups menant à un match nul, soit une branche de 6\*6 nœuds, dans le pire cas.

4. Si vous travaillez en C, quelle est l’utilité ici de compiler avec gcc -O3 plutôt qu’avec les options par défaut ? Donnez des exemples illustratifs.

|  |  |  |  |
| --- | --- | --- | --- |
| Différentes options : | With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.  -O turns on the following optimization flags: | -O2 turns on all optimization flags specified by -O. It also turns on the following optimization flags: | -O3  Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the following optimization flags: |
| Flags ajoutés | -fauto-inc-dec  -fbranch-count-reg  -fcombine-stack-adjustments  -fcompare-elim  -fcprop-registers  -fdce  -fdefer-pop  -fdelayed-branch  -fdse  -fforward-propagate  -fguess-branch-probability  -fif-conversion  -fif-conversion2  -finline-functions-called-once  -fipa-profile  -fipa-pure-const  -fipa-reference  -fipa-reference-addressable  -fmerge-constants  -fmove-loop-invariants  -fomit-frame-pointer  -freorder-blocks  -fshrink-wrap  -fshrink-wrap-separate  -fsplit-wide-types  -fssa-backprop  -fssa-phiopt  -ftree-bit-ccp  -ftree-ccp  -ftree-ch  -ftree-coalesce-vars  -ftree-copy-prop  -ftree-dce  -ftree-dominator-opts  -ftree-dse  -ftree-forwprop  -ftree-fre  -ftree-phiprop  -ftree-pta  -ftree-scev-cprop  -ftree-sink  -ftree-slsr  -ftree-sra  -ftree-ter  -funit-at-a-time | -falign-functions -falign-jumps  -falign-labels -falign-loops  -fcaller-saves  -fcode-hoisting  -fcrossjumping  -fcse-follow-jumps -fcse-skip-blocks  -fdelete-null-pointer-checks  -fdevirtualize -fdevirtualize-speculatively  -fexpensive-optimizations  -ffinite-loops  -fgcse -fgcse-lm  -fhoist-adjacent-loads  -finline-functions  -finline-small-functions  -findirect-inlining  -fipa-bit-cp -fipa-cp -fipa-icf  -fipa-ra -fipa-sra -fipa-vrp  -fisolate-erroneous-paths-dereference  -flra-remat  -foptimize-sibling-calls  -foptimize-strlen  -fpartial-inlining  -fpeephole2  -freorder-blocks-algorithm=stc  -freorder-blocks-and-partition -freorder-functions  -frerun-cse-after-loop  -fschedule-insns -fschedule-insns2  -fsched-interblock -fsched-spec  -fstore-merging  -fstrict-aliasing  -fthread-jumps  -ftree-builtin-call-dce  -ftree-pre  -ftree-switch-conversion -ftree-tail-merge | -fgcse-after-reload  -fipa-cp-clone  -floop-interchange  -floop-unroll-and-jam  -fpeel-loops  -fpredictive-commoning  -fsplit-paths  -ftree-loop-distribute-patterns  -ftree-loop-distribution  -ftree-loop-vectorize  -ftree-partial-pre  -ftree-slp-vectorize  -funswitch-loops  -fvect-cost-model  -fversion-loops-for-strides |

En ajoutant l’option -O3, on remarque que le nombre de nœuds visités est doublé.

Insérer ici exemple illustratif

Avec -O3 :

Sans -O3 :

5. Comparez les critères "max" et "robuste" pour choisir le coup à jouer en fin d’algorithme. Conduisent-ils souvent à des coups différents ? Lequel paraît donner la meilleure performance ?

=> robuste : maximise le nombre de simulations

=> max : maximise le ratio victoire/nb\_simu

=> à tester

6. Donnez une estimation du temps de calcul nécessaire pour jouer le premier coup avec l’algorithme Minimax (sans alpha-beta ni limitation de profondeur).

Si nous implémentions uniquement l’algorithme Minimax, sans optimisation (alpha-beta, limitation de profondeur), on se doute que le temps avant jouer le premier coup serait énorme.

En effet, d’après le fonctionnement de l’algorithme qui est de développer tout l’arbre pour pouvoir choisir quel coup jouer, on peut estimer le nombre de nœud à environ 741 (7 coups possibles pour les 42 cases du plateau, -1 car en fin de jeu on complète des colonnes et donc le nombre de coup possible diminue).

Si un parcours de nœud se fait en une opération, on devrait donc effectuer 4,45 x1032 opérations. Etant donné un ordinateur 3GHz (3 x 109 opérations par seconde) et 1 année comportant 3 x 107 secondes, nous pouvons effectuer 9 x 1016 opérations en 1 an.

Il nous faudrait donc (4,45 x 1032) / (9 x 1016) = 9 \* 1016 années.