Laborator 06: Minimax

Responsabili:

- Darius Neațu
- Ștefan Popa

Obiective laborator

- Insusirea unor cunostinte de baza despre **teoria jocurilor** precum si despre **jocurile de tip joc de suma zero (suma nula, zero-sum games)**
- Insusirea abilitatii de rezolvare a problemelor ce presupun cunoasterea si exploatarea conceptului de joc de suma zero (zero-sum game);
- Insusirea unor cunostinte elementare despre algoritmii necesari rezolvarii unor probleme de joc de suma zero (zero-sum game).

Precizari initiale

Un curs foarte bine explicat este pe canalul de YouTube de la MIT. Va sfatuim sa vizionati integral Search: Games, Minimax, and Alpha-Beta inainte sa parcurgeti materialul de pe ocw.

Importanţă – aplicaţii practice

Algoritmul **Minimax** si variantele sale imbunatatite (**Negamax**, **Alpha-Beta** etc.) sunt folosite in diverse domenii precum teoria jocurilor (Game Theory), teoria jocurilor combinatorice (Combinatorial Game Theory – CGT), teoria deciziei (Decision Theory) si statistica.

Astfel, diferite variante ale algoritmului sunt necesare in proiectarea si implementarea de aplicatii legate de inteligenta artificiala, economie, dar si in domenii precum stiinte politice sau biologie.

Descrierea problemei și a rezolvărilor

Algoritmii Minimax permit abordarea unor probleme ce tin de teoria jocurilor combinatorice. CGT este o ramura a matematicii ce se ocupa cu studierea jocurilor in doi (two-player games), in care participantii isi modifica rand pe rand pozitiile in diferite moduri, prestabilite de regulile jocului, pentru a indeplini una sau mai multe conditii de castig.

Exemple de astfel de jocuri sunt: sah, go, dame (checkers), X si O (tic-tac-toe) etc.

CGT nu studiaza jocuri ce presupun implicarea unui element aleator (sansa) in derularea jocului precum poker, blackjack, zaruri etc. Astfel decizia abordarii unor probleme rezolvabile prin metode de tip Minimax se datoreaza in principal simplitatii atat conceptuale, cat si raportat la implementarea propriu-zisa.

Minimax

Strategia pe care se bazeaza ideea algoritmului este ca jucatorii implicati adopta urmatoarele strategii:

- Jucatorul 1 (**maxi**) va incerca mereu sa-si **maximizeze** propriul castig prin mutarea pe care o are de facut;
- Jucatorul 2 (**mini**) va incerca mereu sa **minimizeze** castigul jucatorului 1 la fiecare mutare.

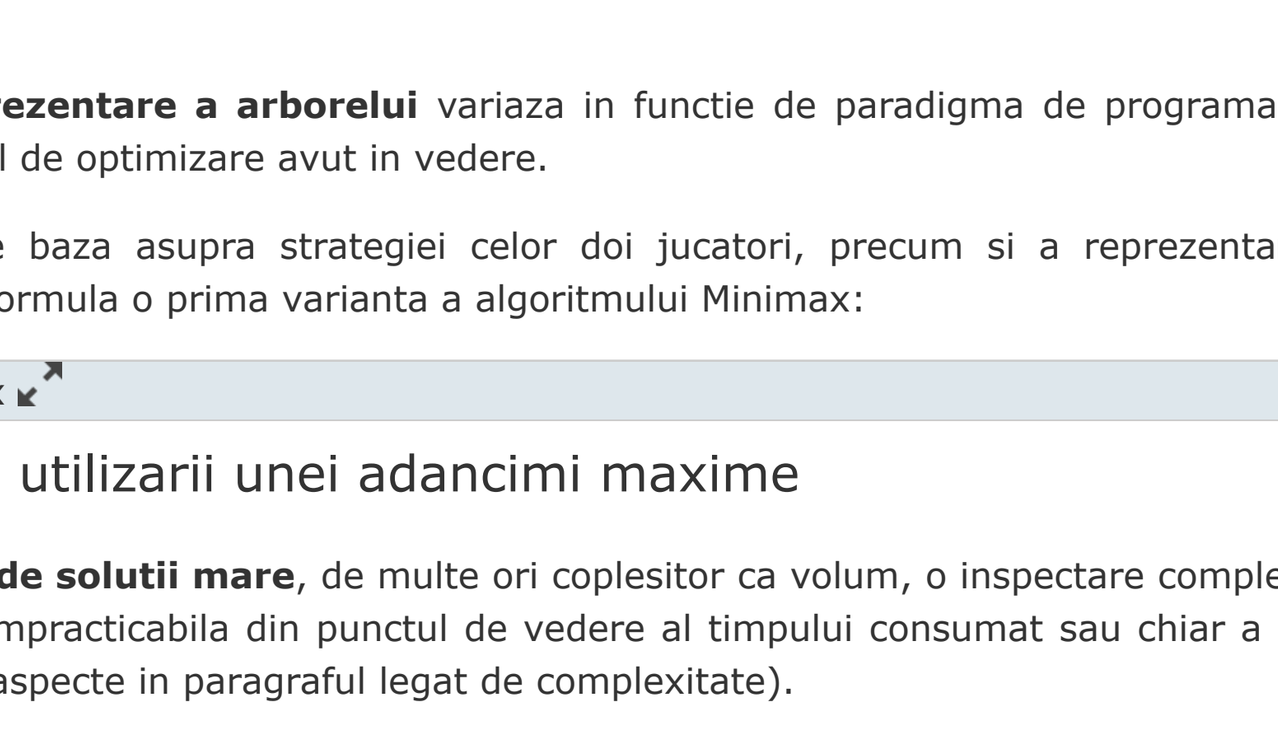
De ce merge o astfel de abordare? Dupa cum se preciza la inceput, discutia se axeaza pe jocuri de suma zero (zero-sum game). Acest lucru garanteaza, printre altele, ca orice castig al Jucatorului 1 este egal cu modulul sumei pierdute de Jucatorul 2. Cu alte cuvinte cat pierde Jucator 2, atat castiga Jucator 1. Invers, cat pierde Jucator 1, atat castiga Jucator 2. Sau

$$Win_{Player_1} = |Loss_{Player_2}|$$

$$|Loss_{Player_1}| = Win_{Player_2}$$

Reprezentarea spatiului solutiilor

In general spatiul solutiilor pentru un joc in doi de tip zero-sum se reprezinta ca un **arbore**, fiecarui nod fiindu-i asociata o stare a jocului in desfasurare (game state). Pentru exemplul nostru de X si O putem considera urmatorul arbore (partial) de solutii, ce corespunde primelor mutari ale lui X, respectiv O:



Metodele de reprezentare a arborelui variaza in functie de paradigma de programare aleasa, de limbaj, precum si de gradul de optimizare avut in vedere.

Avand notiunile de baza asupra strategiei celor doi jucatori, precum si a reprezentarii spatiului solutiilor problemei, putem formula o prima varianta a algoritmului Minimax:

Pseudocod Minimax

Argumentarea utilizarii unei adancimi maxime

Datorita **spatiului de solutii mare**, de multe ori coplesitor ca volum, o inspectare completa a acestuia nu este fezabila si devine impracticabila din punctul de vedere al timpului consumat sau chiar a memoriei alocate (se vor discuta aceste aspecte in paragraful legat de complexitate).

Astfel, de cele mai multe ori este preferata o abordare care parcurge arborele numai pana la o anumita **adancime maxima („depth”)**. Aceasta abordare permite examinarea arborelui destul de mult pentru a putea lua decizii minimalist coerente in desfasurarea jocului.

Totusi, **dezavantajul major** este ca pe termen lung se poate dovedi ca decizia luata la adancimea depth nu este global favorabila jucatorului in cauza.

De asemenea, se observa recursivitatea indirecta. Prin conventie acceptam ca **inceputul algoritmului** sa fie cu functia maxi. Astfel, se analizeaza succesiv diferite stari ale jocului din punctul de vedere al celor doi jucatori pana la adancimea depth. Rezultatul intors este scorul final al miscarii celei mai bune.

Negamax

Negamax este o varianta a minimax, ce se bazeaza pe urmatoarea observatie: intr-un joc cu suma zero castigul unui jucator este egal cu modulul sumei pierdute de celalalt jucator si invers.

Intr-adevar putem spune ca jucatorul mini incearca de fapt sa maximizeze in modul suma pierduta de maxi. Astfel putem formula urmatoarea implementare ce profita de observatia de mai sus.

Nota: putem exprima aceasta observatie si pe baza formulei **max(a, b) = -min(-a, -b)**.

Pseudocod Negamax

Se observa direct avantajele acestei formulari fata de Minimax-ul standard prezentat anterior:

- **claritatea** si **eleganta** sporita a codului
- usurinta in **intretinerea** si **extinderea** functionalitatii

Din punctul de vedere al complexitatii temporale, Negamax nu difera absolut deloc de Minimax (ambele examineaza acelasi numar de stari in arborele de solutii).

Putem concluziona ca este de **preferat** o implementare ce foloseste **negamax** fata de una bazata pe minimax in rezolvarea unor probleme ce tin de aceasta tehnica.

Alpha-beta pruning

Pana acum s-a discutat despre algoritmii Minimax / Negamax. Acestia sunt algoritmi exhaustivi (**exhausting search algorithms**). Cu alte cuvinte, ei gasesc solutia optima examinand intreg spatiul de solutii al problemei. Acest mod de abordare este extrem de ineficient in ceea ce priveste efortul de calcul necesar, mai ales considerand ca extrem de multe stari de joc inutile sunt explorate (este vorba de acele stari care nu pot fi atinse datorita incalcarii principiului de maximizare a castigului la fiecare runda).

O imbunatatire substantiala a minimax/negamax este **Alpha-beta pruning** (**taiere alfa-beta**). Acest algoritm incearca sa optimizeze mini/negamax profitand de o observatie importanta: **pe parcursul examinarii arborelui de solutii se pot elimina intregi subarbori, corespunzatori unei miscari m, daca pe parcursul analizei gasim ca miscarea m este mai slaba calitativ decat cea mai buna miscare curenta.**

Astfel, consideram ca pornim cu o prima miscare M1. Dupa ce analizam aceasta miscare in totalitate si ii atribuim un scor, continuam sa analizam miscarea M2. Daca in analiza ulterioara gasim ca adversarul are cel putin o miscare care transforma M2 intr-o miscare mai slaba decat M1 atunci orice alte variante ce corespund miscarii M2 (subarbori) nu mai trebuie analizate.

De ce?

O observatie foarte importanta se poate face analizand **modul de functionare** al acestui algoritm: este extrem de importanta **ordonarea miscarilor dupa valoarea castigului**.

In **cazul ideal** in care cea mai buna miscare a jucatorului curent este analizata prima, toate celelalte miscari, fiind mai slabe, vor fi eliminate din cautare timpuriu.

In **cazul cel mai defavorabil** insa, in care miscarile sunt ordonate crescator dupa castigul furnizat, Alpha-beta are aceeasi compelxitate cu Mini/Nega-max, neobtinandu-se nicio imbunatatire.

In **medie** se constata o imbunatatire vizibila a algoritmului Alpha-beta fata de Mini/Nega-max.

Exemplu Grafic

Un video cu un exemplu detaliat si foarte bine explicat se gaseste in tutorialul recomandat de pe YouTube (de la minutul 21:30 la 30:30).

Implementare

In continuare prezentam o implementare conceptuala a Alpha-beta, atat pentru Minimax, cat si pentru Negamax:

Pseudocod Minimax with Alpha-beta
Pseudocod Negamax with Alpha-beta

Din nou remarcam claritatea si coerenta sporita a variantei negamax!

Complexitate

In continuare prezentam complexitatile asociate algoritmilor prezentati anterior. Pentru aceasta vom introduce cateva notiuni:

- **branch factor** : **b** = **numarul mediu de ramificari** pe care le are un nod neterminal (care nu e frunza) din **arborele de solutii**
- **depth** : **d** = **adancimea maxima** pana la care se face cautarea in arborele de solutii
  - orice nod de adancime d va fi considerat terminal

Un arbore cu un branching factor **b**, care va fi examinat pana la un nivel **d** va furniza b^d noduri frunze ce vor trebui procesate (ex. calculam scorul pentru acele noduri).

Explicatie

- **minimax/negamax**
  - Un algoritm **mini/negamax** clasic, care analizeaza toate starile posibile, va avea complexitatea $O(b^d)$ - deci exponentiala.
- **alpha-beta**
  - Cat de bun este insa alpha-beta fata de un mini/nega-max naiv? Dupa cum s-a mentionat anterior, in functie de ordonarea miscarilor ce vor fi evaluate putem avea un caz cel mai favorabil si un caz cel mai defavorabil.
  - **best case** : miscarile sunt ordonate descrescator dupa castig (deci ordonate optim), rezulta o complexitate
    - $O(b*1*b*1*b*1...de\ d\ ori...b*1)$ pentru d par
    - $O(b*1*b*1*b*1...de\ d\ ori...b)$ pentru d impar
    - restrangand ambele expresii rezulta o complexitate $O(b^{\frac{d}{2}}) = O(\sqrt{b^d})$
    - prin urmare, intr-un caz ideal, algoritmul alpha-beta poate explora de 2 ori mai putine nivele in arborele de solutii fata de un algoritm mini/nega-max naiv.
  - **worst case**: miscarile sunt ordonate crescator dupa castigul furnizat unui jucator, astfel fiind necesara o examinare a tuturor nodurilor pentru gasirea celei mai bune miscari.
    - in consecinta complexitatea devine egala cu cea a unui algoritm mini/negamax naiv.

Concluzii si observatii

Alpha-beta NU ofera o alta solutie fata de Minimax! Este doar o optimizare pusa deasupra algoritmului Minimax care ne permite sa exploram mai multe stari in acelasi timp sau pentru acelasi numar de stari sa optinem un timp de doua ori mai mic.

Exemple

Dintre cele mai importante jocuri in care putem aplica direct strategia minimax, mentionam:

- X și 0
  - joc foarte simplu/usor (spatiul starilor este mic).
  - Prin urmare tot arborele de solutii poate fi generat si explorat intr-un timp foarte scurt.
- sah
  - joc foarte greu (spatiul starilor este foarte mare)
  - minimax/negamax simplu poate merge pana la $d = 7$ (nu reusea da bata campionul mondial la sah - campion uman)
  - alpha-beta poate merge pana la $d = 14$
  - **Deep Blue** a fost implementarea unui bot cu minimax si alpha-beta care a batut in 1997 campionul mondial la sah (Gary Kasparov).
- Ultimate tic-tac-toe
  - varianta mult mai grea de X si 0 (spatiul starilor foarte mare)
  - s-a dat la proiect PA 2016 :D
  - implemantarile se pot testa aici
- Nim
- Reversi

Alte exempe de jocuri interesante:

- Go
  - solutiile se bazeaza pe Monte Carlo Tree Search (nu pe minimax)
  - AlphaGo este botul cel mai bun pe tabla de 19×19

Nim

Fiind date 3 multimi de bile, fiecare jucator trebuie sa extraga la fiecare mutare 1, 2 sau 3 bile din oricare multime.

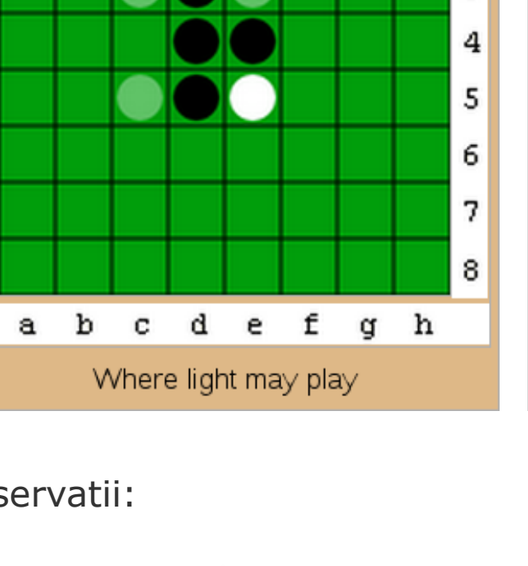
Cel care este fortat sa aleaga ultima bila, pierde.

Reversi game

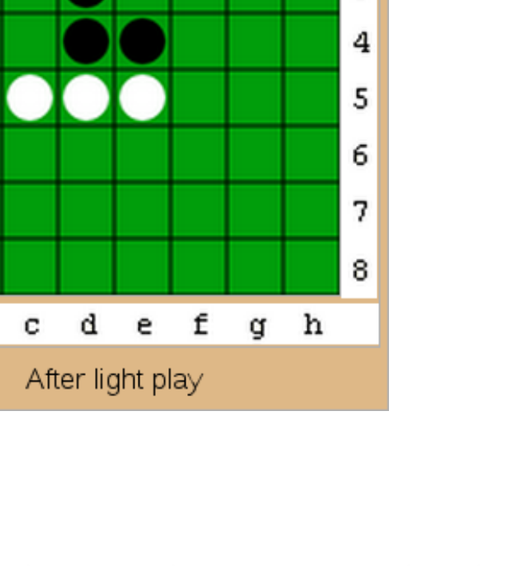
Tabla de joc consista dintr-un grid 6×6. Piesele pot fi reprezentate de monede, fiecarui jucator fiindu-i asociata o fata diferita a monezii. Jucatorii muta alternativ, dupa regula urmatoare:

- Pozitia (x, y) in care este plasata piesa trebuie sa fie libera
- Trebuie sa existe o alta pozitie (x', y'), pe aceeasi linie, coloana sau diagonala a jucatorului aflat la mutare si toate pozitiile dintre (x, y) si (x', y') trebuie sa fie ocupate de piese ale adversarului
- Piesele adversarului dintre (x, y) si (x', y') vor fi capturate, intorcandu-se monedele pe cealalta fata.

Mai jos, de la stanga spre dreapta: pozitia initiala, posibilitatile de mutare ale primului jucator, tabla dupa prima mutare, posibilitatile de mutare ale celuilalt jucator, tabla dupa a doua mutare.

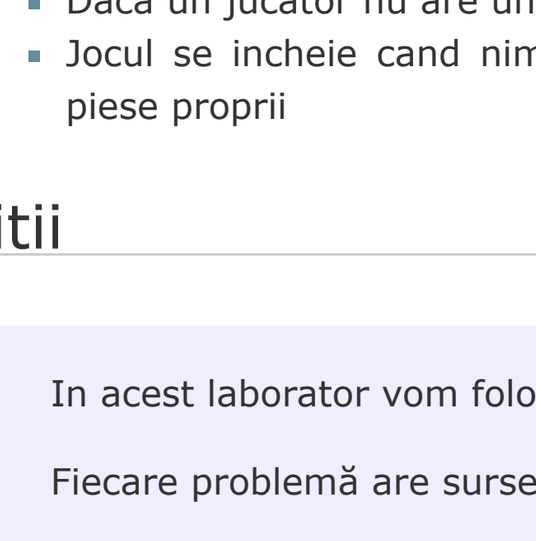


Starting position

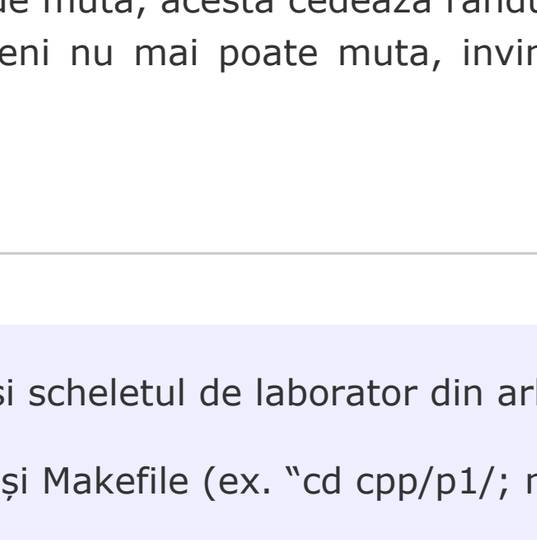


Where dark may play

After dark play



Where light may play



After light play

Observatii:

- Jucatorul poate acapara piese ale adversarului in mai multe directii simultan
- Daca un jucator nu are unde muta, acesta cedeaza randul, adversarul efectuand a doua mutare la rand
- Jocul se incheie cand nimeni nu mai poate muta, invingatorul fiind acela care detine cele mai multe piese proprii

Exercitii

In acest laborator vom folosi scheletul de laborator din arhiva skel-lab06.zip.

Fiecare problemă are surse și Makefile (ex. “cd cpp/p1/; make; make run”).

Vom implementa algoritmi pentru jocurile **Nim** si **Reversi**.

Minimax Nim

Se doreste implementarea algoritmulului minimax sau negamax pentru Nim.

Recomandam implementarea variantei Negamax.

Minimax Reversi

Se doreste implementarea algoritmulului minimax sau negamax pentru Reversi.

Recomandam implementarea variantei Negamax.

Bonus

Extindeti algoritmul implementat anterior pentru jocul **Reversi** intr-un algoritm de tip alpha-beta pruning. Cum puteti sa comparati cei doi algoritmi implementati pentru Reversi?

Referinte

[1] http://en.wikipedia.org/wiki/Minimax

[2] http://en.wikipedia.org/wiki/Negamax

[3] http://en.wikipedia.org/wiki/Alpha-beta_pruning

[4] http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/

Table of Contents