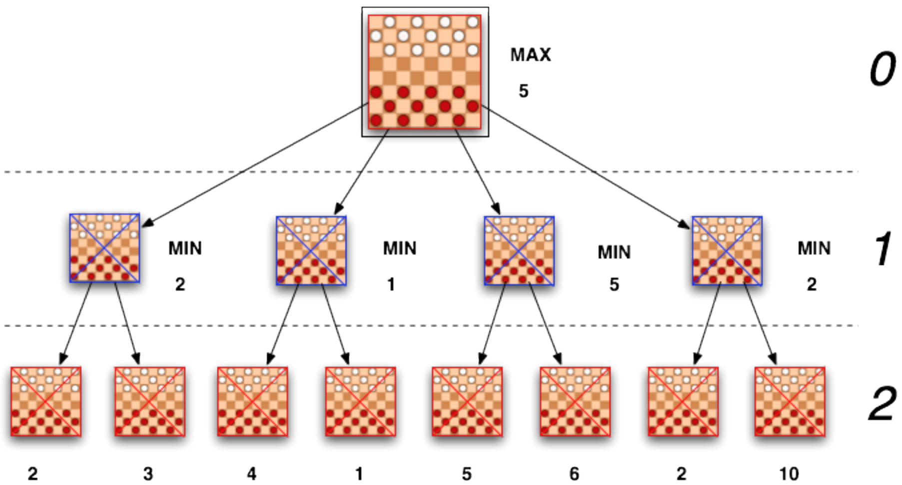
Proiectarea algoritmilor - Seria CD

Laborator 5

Minimax

Mihai Nan

mihai.nan.cti@gmail.com



Facultatea de Automatică şi Calculatoare

Universitatea Politehnica din Bucureşti

Anul universitar 2016 - 2017

1 Descriere

Inteligența Artificială poate fi definită ca simularea inteligenței umane procesată de mașini, în special, de sisteme de calculatoare. Acest domeniu a fost , în general, caracterizat de cercetări complexe în laboratoare, iar, în ultima perioadă, a devenit parte a tehnologiei în aplicațiile comerciale.

Începuturile inteligenței artificiale pot fi văzute imediat după al Doilea Război Mondial, în primele programe care rezolvau puzzle-uri sau care jucau anumite jocuri. Au existat două motive pentru care jocurile au fost printre primele domenii de aplicare a inteligenței artificiale: pentru că performanța programului este ușor de măsurat, apoi, deoarece regulile sunt, în general, simple și puține la număr, deci pot fi ușor descrise și folosite.

1.1 Descrierea formală a unui joc

Faptul că în cazul jocurilor mai apare și un adversar face ca problema de alegere a unei acțiuni să fie mai complicată decât o problema de căutare clasică, deoarece adversarul este cel care aduce o incertitudine, jucătorul curent neștiind decizia următoare a adversarului. Astfel, faptul că mutările adversarului sunt imprevizibile ne face să specificăm o mutare pentru fiecare posibil răspuns al adversarului.

În continuare, vom considera cazul general al unui joc de două persoane, pe care le vom numi `Max` și `Min`, cu informație perfectă.

Pornind de la aceste considerente, un joc poate fi definit formal ca o problemă de căutare cu următoarele componente:

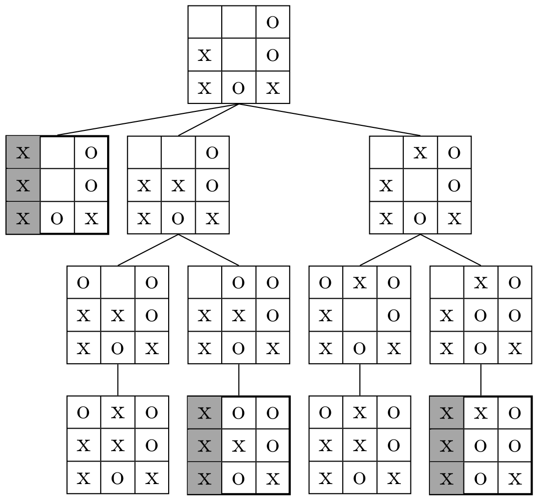
- **starea inițială** - include pozițiile de pe tablă și cine este cel care urmează să efectueze mutarea;
- **mulțimea acțiunilor posibile** - mutările admise pe care le poate face un jucător la o anumită rundă;
- **stare terminală** - stările în care jocul se încheie (cu victoria unui jucător sau cu remiză);
- **funcție de utilitate** - care întoarce o valoare numerică pentru rezultatul jocului.

Problema care se pune pentru rezolvarea problemei de căutare este găsirea unei strategii care să îl ducă pe `Max` la o stare terminală în care el este câștigătorul, indiferent de ce mutări face `Min`.

1.2 Arborele de joc

Arborele de joc este o modalitate de reprezentare a mutărilor posibile ce se pot realiza pentru un anumit joc. Rădăcina arborelui este starea inițială a jocului, iar nodurile arborelui reprezintă stări intermediare în joc. Arcele sunt mutările efectuate în joc și, astfel, putem spune că trecerea dintr-o stare intermediară (nod în arbore) în altă stare intermediară se face prin intermediul unei mutări (arc în arbore). Frunzele arborelui sunt stări finale, configurații din care jucătorul aflat la mutare nu mai poate muta. Cu alte cuvinte, frunzele corespund pozițiilor de victorie, pierdere sau remiză.

În continuare, se va prezenta un exemplu pentru celebrul joc `X` și `0`, având ca rădăcină o configurație diferită de cea inițială, reprezentând, astfel, un subarbore al întregului arbore de joc.



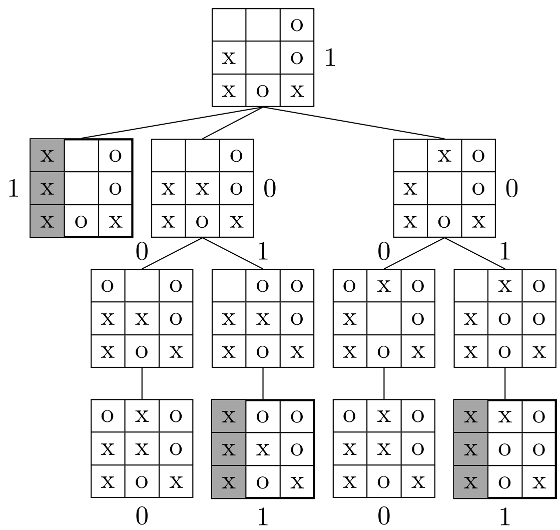
1.3 Algoritmul Minimax

În cazul jocurilor cu doi jucători, în care oponenți își modifică pe rând poziția de joc sau starea, algoritmul **Minimax** este cel mai uzitat, împreună cu variantele sale îmbunătățite. În linii mari, acest algoritm folosește o funcție care decide cât de bună este o poziție, prin atribuirea unor scoruri. Algoritmul se bazează pe existența a doi jucători cu strategii diferite: jucătorul ***Max*** este cel care va încerca în permanență să-și maximizeze câștigul, în timp ce jucătorul ***Min*** dorește să minimizeze câștigul jucătorului ***Max*** la fiecare mutare. Având în vedere că în cazul jocurilor ce au un factor de ramificare mare arborele de căutare ar conține foarte multe noduri, algoritmul ajungând să fie aproape imposibil de aplicat, datorită timpului mare necesar analizării tuturor pozițiilor disponibile, în vederea selectării celei mai potrivite, s-a încercat optimizarea acestuia. Astfel, au apărut diverse variante echivalente optimizate cum ar fi **Negascout**, **Negamax**, **Alpha-Beta**.

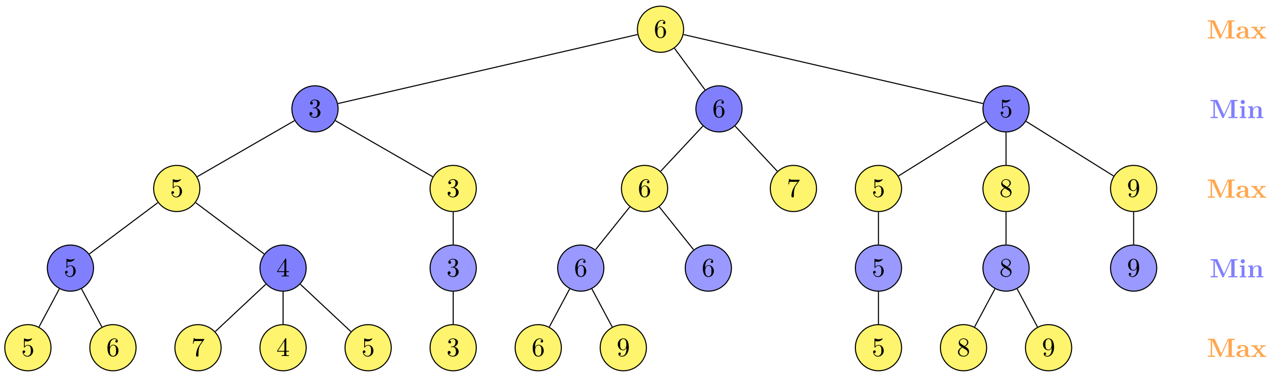
Algoritmul **Minimax** realizează o căutare în adâncime în arborele de joc, unde nodurile reprezintă stări ale jocului, iar arcele definesc acțiuni posibile ce se pot realiza dintr-o stare de joc. Astfel, configurația inițială a jocului este rădăcina arborelui de joc, iar frunzele reprezintă stări finale pentru joc, pentru care se poate aplica o funcție de utilitate în vederea determinării unei valori, care este propagată spre nivelurile superioare ale arborelui.

Pentru o înțelegere mai bună a acestui algoritm, vom relua exemplul anterior, untilizând o funcție de utilitate care atribuie valoarea 1 unei stări finale dacă jucătorul `X` a câștigat, -1 dacă jucătorul `0` a câștigat și 0 pentru remiză.

În continuare, se va prezenta un exemplu pentru celebrul joc `X` și `0`, având ca și rădăcină o configurație diferită de cea inițială, reprezentând, astfel, un subarbore al întregului arbore de joc. În acest caz, funcția de utilitate atribuie valoarea 1 unei stări finale dacă jucătorul `X` a câștigat, -1 dacă jucătorul `0` a câștigat și 0 pentru remiză.



Pentru acest algoritm, arborele de căutare constă în alternarea nivelelor pe care un jucător încearcă să-și maximizeze câștigul cu nivelele pe care adversarul își minimizează câștigul. Primul jucător va încerca să-și maximizeze câștigul, astfel, jucătorul `Max` este cel care mută primul, iar al doilea jucător va încerca să minimizeze câștigul primului jucător, jucătorul `Min` reprezentând adversarul.



În concluzie, algoritmul **Minimax** determină o strategie optimă pentru `MAX`, iar acesta constă în următorii pași:

1. Generează tot arborele de joc, până la stările terminale.
2. Aplică funcția de utilitate pentru fiecare stare terminală pentru a îi determina valoarea.
3. Folosește utilitatea stărilor terminale pentru a determina utilitatea stărilor de la un nivel superior din arborele de căutare.
4. Continuă evaluarea utilităților nodurilor pe niveluri mergând până la rădăcină.
5. Când se ajunge la rădăcină, `Max` alege nodul de pe nivelul inferior cu valoarea cea mai mare.

1.4 Alpha - Beta Pruning

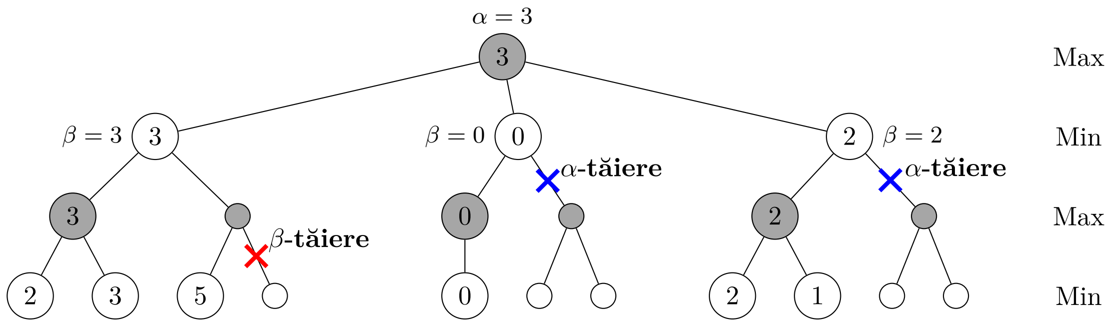
Alpha-Beta pruning reprezintă o îmbunătățire semnificativă a Minimax-ului, deoarece această tehnică elimină întregi subarbori corespunzători unei anumite mutări dacă mutarea respectivă este mai slabă, din punct de vedere calitativ, decât cea mai bună mutare curentă. Această tehnică de optimizare folosește două valori: `alpha` și `beta` care se actualizează în timpul execuției algoritmului; care constituie o fereastră folosită pentru filtrarea mutărilor posibile.

- α - mai este numit și plafonul de minim, fiind cel care stabilește faptul că o mutare nu poate avea o valoare mai mică decât acest prag;
- β - numit și plafonul de maxim, se folosește pentru a verifica dacă o mutare este prea bună pentru a putea fi luată în calcul.

Ținând cont de principiul algoritmului Minimax, se poate concluziona că plafonul minim al unui jucător reprezintă plafonul maxim al celuilalt și invers. Această fereastră, definita de cele două plafoane, este inițializată cu intervalul $[-\infty, +\infty]$ și se tot micșorează pe parcursul rulării algoritmului, determinând tăierea unui număr din ce în ce mai mare de mutări din arborele de joc. Aceste tăieri sunt de două tipuri:

- **α-tăieri** - În cazul în care există, pentru un nod **Min**, o acțiune ce are asociată o valoare $v \leq \alpha$, atunci putem renunța la expandarea subarborelui său, deoarece **Max** poate atinge deja un câștig mai mare, α, dintr-un subarbore precedent.
- **β-tăieri** - În cazul în care există, pentru un nod de tip **Max**, o acțiune ce are asociată o valoare $v \geq \beta$, atunci putem renunța la expandarea subarborelui său, deoarece **Min** a limitat deja câștigul lui **Max** la β.

1.4.1 Exemplu



1.4.2 Pseudocod

Pornind de la descrierea anterioară a metodei, se poate contura următorul algoritm[1]:

Algorithm 1 Alpha-Beta Pruning

```
function ALPHA-BETA-SEARCH(state)
    v ← MAX-VALUE(state, −∞, +∞)
    for action ∈ ACTIONS(state) do
        if GETVALUE(action) = v then
            return action
        end if
    end for
end function

function MAX-VALUE(state, α, β)
    if TERMINAL-TEST(state) then
        return UTILITY(state)
    end if
    v ← −∞
    for action ∈ ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(state, action), α, β))
        if v ≥ β then
            return v
        end if
        α ← MAX(α, v)
    end for
end function

function MIN-VALUE(state, α, β)
    if TERMINAL-TEST(state) then
        return UTILITY(state)
    end if
    v ← +∞
    for action ∈ ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(state, action), α, β))
        if v ≤ β then
            return v
        end if
        β ← MIN(β, v)
    end for
end function
```

[1] Variantă propusă în ***Artificial Intelligence: A Modern Approach***

2 Probleme propuse

2.1 Problema 1

Implementați în clasa **AICore** metoda **minimax** care va întoarce o pereche formată din utilitatea unei stări și o acțiune.

Algorithm 2 Minimax

```
function MINIMAX(board, player, maxDepth, currentDepth)
    if board.isGameOver() or currentDepth = maxDepth then
        return board.evaluate(),None
    end if
    bestMove ← None
    if board.currentPlayer() = player then
        bestScore ← −∞
    else
        bestScore ← ∞
    end if
    for move in board.getMoves() do
        newBoard ← board.makeMove(move)
        recursedScore, currentMove ← MINIMAX(newBoard, player, maxDepth, currentDepth + 1)
        currentScore ← −recursedScore
        if board.currentPlauer() = player then
            if currentScore > bestScore then
                (bestScore, bestMove) ← (currentScore, move)
            end if
        else
            if currentScore < bestScore then
                (bestScore, bestMove) ← (currentScore, move)
            end if
        end if
    end for
    return bestScore, bestMove
end function
```

2.2 Problema 2

Implementați în clasa **AICore** metode **negamax** care va întoarce același lucru, aplicând, de această dată algoritmul **Negamax**.

Algorithm 3 Negamax

```
procedure NEGAMAX(board, maxDepth, currentDepth)
    if isGameOver(board) or curentDepth == maxDepth then
        return (evaluate(board), None)
    end if
    bestMove ← None
    bestScore ← −INFINITY
    for move ∈ getMoves(board) do
        newBoard ← makeMove(board, move)
        (recursedScore, currentMove) ← NEGAMAX(newBoard, maxDepth, currentDepth + 1)
        currentScore ← −recursedScore
        if currentScore > bestScore then
            bestScore ← currentScore
            bestMove ← move
        end if
    end for
    return (bestScore, bestMove)
end procedure
```

2.3 Problema 3

Implementați metoda **abNegamax** în clasa **AICore**.