Analýza a návrh riešenia

Slavomír Slovenkai

Univerzita Pavla Jozefa Šafárika v Košiciach

1 Perfektná hra

Hra sa pokladá za vyriešenú, ak jej výsledok (výhra, prehra, remíza) - **herné teoretické hodnoty** - je možné určiť na základe aktuálne pozície hry. Riešenie hry je možné na niekoľkých úrovniach:

1. **ultra-slabé riešenie**: bola získaná herná teoretická hodnota začiatočných pozícií nekonštruktívnym dôkazom, čo neposkytuje stratégiu ako dosiahnuť daný stav
2. **slabé riešenie**: je známa herná teoretická hodnota začiatočných pozícií a stratégia ako ju dosiahnuť (napríklad pomocou databázi herných teoretických hodnôt pre veľkú podmnožinu herných stavov)
3. **silné riešenie**: je známa stratégia, ako dosiahnuť herné teoretické hodnoty začínajúc z ľubovoľnej dosiahnuteľnej pozície.
4. **ultra-silné riešenie**: je známa stratégia umožňujúca dosiahnuť viac ako herné teoretické hodnoty proti oponentovy, ktorý môže spraviť chybu.

V prípade, že je známa stratégia, ktorá vedie k najlepšiemu výsledku pre daného hráča a každá koncová pozícia môže byť kvalifikovaná ako výhra, prehra alebo remíza, hovoríme o **perfektnej hre**.

Hráč, ktorý túto stratégiu využíva sa nazýva **perfektný hráč** a platí, že ak sa nachádza vo výhernej pozícii, nemôže sa dostať do remízovej alebo prehrávajúcej. A ak sa nachádza v remízovej pozícii môže sa dostať len do vyhrávajúcej alebo ostať v remízovej pozícii.

Cieľom je vytvoriť algoritmus pre perfektného hráča ovládaného počítačom, tak aby dosiahol čo najviac výhier.

2 Nine Mens Morris

Jednou z vyriešených hier je aj hra **Nine Mens Morris**. Je to hra dvoch hráčov na hracej ploche, ktorá pozostáva z 24 pozícií, pričom niektoré z nich sú spojené čiaramy. Obaja hráči majú na začiatku hry po 9 figúrok mimo hracej plochy. Hra pozostáva z dvoch fáz:

1. **úvodná fáza (ukladacia)**: hráči postupne ukladajú figúrky na voľné pozície (jedna figúrka v jednom ťahu). Táto fáza končí, keď hráči položili všetky svoje figúrky.
2. **neskorá fáza (ťahanie)**: hráči môžu pohybovať figúrkami, len po vyznačených čiarách. Výnimkou je, keď má hráč iba tri figúrky na hracej ploche, môže s ich presúvať na ľubovoľné voľné hracie políčko.

Špeciálnou kombináciou pozícií nazývame **mlyn**. Ten nastáva, keď má jeden hráč tri figúrky uložené na priamej čiare spájajúcej tri políčka. Moment, keď hráč vytvorí mlyn, hovoríme, že hráč uzavrel mlyn. Keď sa tak stane, vyberá jednu figúrku protihráča z hracej plochy. Pričom platí, že ak protihráč má nejakú figúrky mimo mlyna, musí vyberať z nich.

Hráč hru prehráva, ak nemôže vykonať ťah alebo má menej ako tri figúrky na hracej ploche a nemá žiadnu na položenie.

2.1 Metódy a postupy simulácie perfektného hráča

Minmax algoritmus slúži na vybranie najlepšieho ťahu z danej pozície pomocou prehľadávanie stromu hry, ktorého vrcholy sú herné pozície a hrany reprezentujú možný prechod medzi nimi. Jeho cieľom je minimalizovať straty hráča a maximalizovať straty protihráča. Toto prehľadávanie vyžaduje ohodnotenie herných pozícií, ktoré určí, ako vhodná je vybraná pozícia pre hráča, ktorý je na ťahu.

```
function MINMAX(node, depth, maximizingPlayer)
    if depth = 0 || node is a terminal node then
        return the heuristic value of node
    end if
    if maximizingPlayer then
        v ← −∞
        for each child of node do
            v ← max(v, MINMAX(child, depth − 1, false))
            bestValue ← max(bestValue, v)
        end for
        return bestValue
    else
        v ← ∞
        for each child of node do
            v ← min(v, MINMAX(child, depth − 1, true))
            bestValue ← max(bestValue, v)
```

```
        end for
        return bestValue
    end if
end function
```

Alpha-Beta prehľadávanie je rozšírený minmax algoritmus. Narozdiel od neho neprehľadáva vo vetvách, v ktorých už nemôže dosiahnuť lepší výsledok, ako je aktuálne najlepší možný.

```
function ALPHABETA(node, depth, α, β, maximizingPlayer)
    if depth = 0 || node is a terminal node then
        return the heuristic value of node
    end if
    if maximizingPlayer then
        v ← −∞
        for each child of node do
            v ← max(v, ALPHABETA(child, depth − 1, α, β, false))
            α ← max(α, v)
            if β ≤ α then
                break
            end if
        end for
        return v
    else
        v ← ∞
        for each child of node do
            v ← min(v, ALPHABETA(child, depth − 1, α, β, true))
            β ← max(β, v)
            if β ≤ α then
                break
            end if
        end for
        return v
    end if
end function
```

Ich ohodnotenie môže byť napríklad na základe funkcie, ktorá zhodnotí danú pozíciu, v ktorej sa hráč nachádza, alebo pomocou herných teoretických hodnôt, poprípade kombináciou spomenutých metód.

Retrográdna analýza je štandardnou metódou, ktorá sa používa pre silné riešenia. Jej úlohou je na základe databázy koncových stavov vypočítať databázu herných teoretických hodnôt pre všetky stavy v hre. Túto databázu môže využiť herný program pre určenie ďalšieho ťahu. Aby sa zabránilo možnému zacykleniu, sú popri herných teoretických hodnotách výhry a prehry uložené aj minimálne počty ťahov do konca hry (hĺbka do výhry). V prípade využitia tejto databázy sa používajú rôzne postupy, ako sa rozhodovať. Jeden z nich využíva klasifikáciu remízových stavov.

Retrográdna analýza pre výpočet databázy prezentovaná v článku [2] v neupravenej verzii pozostáva z dvoch častí. Prvou je inicializácia a druhou je spracovanie prioritného radu.

```
function INITIALISATION
    Q ← priority queue
    for all p ∈ positions do
        if p is a win end-state then
            R[p] ← value(win in 0)
            push p into Q
        else if p is a loss end-state then
            R[p] ← value(loss in 0)
            push p into Q
        else if p is a draw end-state then
            R[p] ← count(0)
        else
            R[p] ← count(number of possible moves in p)
        end if
    end for
    return Q
end function
function ProcessingQueue(Q)
    while Q is not empty do
        pop position e from Q
        for all predcessors p ∈ e do
            if R[p] is count then
                if R[e] is win then
                    decrement R[p].count
                    if R[p].count = 0 then
                        R[p] ← value(loss in 1 + R[e].value)
                        push p into Q
                    end if
                else
                    R[p] ← value(win in 1 + R[e].value)
                    push p into Q
                end if
            end if
        end for
    end while
end function
```

Všetky pozície, ktoré budú po skončení algoritmu neoznačené ako výherné alebo prehrávajúce, sú považované za remízové.

Iné lokálne heuristiky využívajú informácie z databázy herných stavov, ktoré sú najviac pár ťahov vzdialené od aktuálnej pozície. Napríklad sa rozhoduje na základe **pomeru počtu optimálnych ťahov k počtu možných ťahov** z danej pozície.

Spomínané metódy môžu byť rekurzívne skombinované s pravdepodobnosťou vykonania optimálneho ťahu pozdĺž zvolenej cesty, ako je napríklad algoritmus využívajúc **náročnosť pozície**.

Ďalšími možnými spôsobmi je napríklad naučiť sa model oponenta, a vykonávať ťahy podľa toho, ako hral v minulosti alebo ohodnotiť herné pozície pomocou počtu figúrok v hre.

2.2 Náročnosť pozície

Následný algoritmus navrhnutý v [1] využíva databázu herných teoretických hodnôt s počtom ťahov do konca hry pre výhru a prehru.

Ak sa perfektný hráč nachádza vo vyhrávajúcej pozícii, vyberá si vyhrávajúci ťah na najmenej krokov. Keď sa nachádza v prehrávajúcej pozícii, vyberá si ťah na najviac krokov. Keďže remízové pozície neobsahujú informáciu o počte ťahov do konca hry, je potrebné sa rozhodnúť na základe iného kritéria.

V tomto modely je rozhodovanie perfektného hráča v prípade remízového stavu založené na náročnosti pozície pre oponenta. Ak je známa pravdepodobnosť vykonania perfetkného ťahu oponentom po i-tom ťahu perfektného hráča, tak perfektný hráč vyberá taký ťah, kde bude táto hodnota minimálna.

V článku boli popísané dve modely protihráča. **Náhodný hráč**, ktorého pravdepodobnosť vykonania ťahu je pre všetky možnosti rovnaká a **hráč s výberom na základe vzdialenosti**, u ktorého sa predpokladá, že vie predpovedať niekoľko ťahov dopredu.

Príklad ohodnotenia pozície pre náhodného hráča, kde R je remíza, P je prehra a perfektný ťah je neprehrávajúci ťah:

1. Možné ťahy: R, R. Počet perfektných ťahov: 2. Ohodnotenie: 1
2. Možné ťahy: R, P. Počet perfektných ťahov: 1. Ohodnotenie: 0.5
3. Možné ťahy: R, P, P. Počet perfektných ťahov: 1. Ohodnotenie: 0.33

V prípade **výberu ťahu na základe vzdialenosti** sa pre simulovanie odhadu hráča používa horné (U) a dolné (L) ohraničenie. Pravdepodobnosť, že hráč bude hrať ťah, ktorý je prehrávajúci na L a menej ťahov je určená ako nulová. Pravdepodobnosť, že oponent vykoná pohyb s počtom krokov väčším ako je U ťahov je považovaná akoby vykonával remízový ťah.
Príklady uvažujúc L = 2 a U = 12:

1. Možné ťahy: R, R. Ohodnotenie: 1
2. Možné ťahy: R, P2. Ohodnotenie: 1
3. Možné ťahy: R, P6. Ohodnotenie: 0.7
4. Možné ťahy: R, P12. Ohodnotenie: 0.5

Algorimtus pre určenie ohodnotenia i-tej pozície s v nasledovníkmi, kde d_{ij} je 0, ak to nie je remízový ťah, D je počet ťahov do konca a U, L sú horné a dolné ohraničenie:

for $j = 1$ to v **do**
  if $d_{ij} = 0$ **then**

```
        if D ≥ U then
            sum ← sum + 1
        else if D ≥ L then
            sum ← sum + (D − L)/(U − L)
        end if
    else
        sum ← sum + 1
        perfektne ← perfektne + 1
    end if
end for
ohodnotenie_i ← perfektne/sum
```

Dosiahnuté výsledky ukazujú výhodu tohto prístupu, na rozdiel od výberu iba na základe hodnôt z databázy, avšak nepreukázali viditeľný rozdiel medzi Náhodným hráčom a Výberom na základe vzdialenosti.

2.3 Klasifikácia remíz pomocou podpriestorov

V článku [2] bol vytvorený algoritmus pre ultra silné riešenie hry Nine Mens Morris, pomocou ohodnotenia remízových stavov a upravenej retrográdnej analýzy, pre výpočet databázy z koncových stavov, ktorá obsahuje aj počet krokov do konca hry pre výherné a prehrávajúce pozície. Pri rozhodovaní sa v prípade výhernej alebo prehrávajúcej pozície postupuje štandardne. Článok rieši problém rozhodovania sa v prípade remízových stavov, tak že ich ohodnotí na základe toho, ako sú stabilné. Teda podľa toho aká je šanca, že sa opustí remízový stav.

Najprv si potrebujeme objasniť niektoré pojmy. Pod **podpriestorom** sa rozumie štvorica čísel: počet figúrok na položenie pre prvého a pre druhého hráča (2 samostatné čísla), počet figúrok na ploche pre prvého a druhého hráča (2 samostatné čísla). **Pracovnou jednotkou** sa nazýva taká dvojica podpriestorov, ktorá medzi sebou vytvára cyklus. To znamená, že z podpriestoru α sa vieme dostať do podpriestoru β a naopak. Príkladom sú podpriestory (0, 0, 8, 9) a (0, 0, 9, 8).

Základnou myšlienkou algoritmu je, že ak nie je možný výherný ťah, zvolí sa stabilná remíza. Remíza je stabilná, ak existuje optimálna postupnosť ťahov, ktoré neopustia pracovnú jednotku.

Pre priradenie hodnôt stabilným stavom boli použitá heuristika, ktorá vypočítala hodnotu podpriestoru a hodnota stabilného stavu bola daná hodnotou podpriestoru, v ktorom sa nachádza.
V článku boli porovnané dve prístupy:

1. Ohodnotenie stabilných stavov na základe rozdielu figúrok hráčov.
2. Využitie databázy herných teoretických hodnôt a pomer výhier, prehier a remíz.

Ohodnotenie podpriestoru s v druhom prístupe bolo určené nasledovným vzťahom

$$val_s = (W_s + L_{-s} + D_s/2 + D_{-s}/2)/(T_s + T_{-s}) \tag{1}$$

kde $-s$ je druhý podpriestor v pracovnej jednotke, W_x, L_x, D_x sú počty výhier, prehier, remíz a T_x je počet všetkých stavov v podpriestore x.

Každý stav obsahuje dva kľúče. Prvý je herná teoretická hodnota a druhý je počet ťahov do konca (DTW). Vyššie uvedená heuristika sa využíva pre určenie o koľko je lepší (horší) daný podpriestor, ako ten v ktorom sme skončíli optimálnou hrou. Tieto hodnoty sú uložené ako prvé kľúče stavov.

Na určenie DTW pre remízy je využitý nasledujúci princíp: maximalizuje sa rozdiel mezdi počtom ťahov v podpriestoroch, ktoré sú lepšie ako konečný a počtom ťahov v podpriestoroch, ktoré sú horšie ako konečný podpriestor.

V uvedenom článku porovnali silné a utlra-silné riešenie proti ľudským hráčom a s ultra silným riešením dosiahli viac výhier. Avšak napriek očakávaniam, heuristika využívajúca počet figúriek hráčov dosiahla lepšie výsledky ako heuristika využívajúca pomer výhier, prehier a remíz.

2.4 Ohodnotenie na základe typu ťahu

V článku [3] používajú na rozhodovanie minmax algoritmus a pozície ohodnocujú na základe typu ťahu, ktorý je možné spraviť. Kritériá zahŕňajú položenie alebo posunutie figúrky, vytváranie potenciálneho mlynu, blokovanie mlynu oponenta a vytvorenie mlynu daným ťahom.

Uvedené výsledky ukazujú, že táto stratégia dosiahla 7 000 výhier a 3 000 prehier z 10 000 odohraných herných partií. Navyše v hre nebolo zahrnuté pravidlo skákania pri počte troch figúrok a pri rozhodovaní využívali ťahy len do hĺbky jedna.

2.5 Neurónove siete

V práci [4] využívajú neúronove siete aby rozlíšili platné a neplatné ťahy. Celkovo používajú tri neurónové siete, pričom každá z nich predpovedá jednu časť z pohybu, ktorý je daný troma zložkami a to: kam položiť figúrku, odkiaľ vyhodiť figúrku, odkiaľ vybrať figúrku pre pohyb. Všetky tri siete sú reziduálne, pri ktorých uvádzajú ľahšie riešenie optimalizačných problémov. Tieto siete sa odlišujú optimalizovaním funkcie H pomocou jej reziduálu

$$F(x) = H(x) - x \tag{2}$$

z ktorého je možné získať späť originálnu funkciu, ktorá je

$$H(x) = F(x) + x \tag{3}$$

Výsledky ukázali, že systém je schopný odlíšiť rozdiel medzi platným a neplatným ťahom. Navyše experimentálne výsledky ukázali, že neurónové siete sú schopné navrhnúť platné ťahy ako najlepšie možnosti.

3 Návrh riešenia

Rozhodovanie ťahu v prípade, že sa hráč riadený počítačom nachádza vo výhernej alebo prehrávajúcej pozícii, je vyriešené výberom najlepšieho možného ťahu. V prípade výhernej pozície vyberáme pre nás výherný ťah na najmenej krokov a v prípade prehrávajúcej pozície sa volí pre nás prehrávajúci ťah na čo najviac krokov. Budeme sa najmä zaoberať výberom ťahu, ak sa nachádzame v remízovej pozícii.

Návrh nášho riešenia optimalizácie algoritmu dvoch hráčov aplikovaného na hru Nine Mens Morris bude vychádzať z predpokladu, že ľudský hráč je omylný a úlohou nášho algoritmu bude nájsť pre počítač také ťahy, ktoré budú viesť k väčšej šanci na výhru v porovnaní s algoritmom náhodného hráča prezentovaného v článku [1].

3.1 Analýza existujúcich prístupov

Väčšina vyššie prezentovaných algoritmov uvažuje o perfektnej hre pre počítačom riadeného hráča a vyberajú také ťahy, ktoré z dlhodobého hľadiska vytvárajú náročné pozície pre ľudského hráča, keďže ten väčšinou robí chyby a tento bod chceme vo svojom postupe tiež zohľadniť.

Niektoré z algoritmov využívajú pre svoje rozhodovanie pravdepodobnosť, akou protihráč vykoná daný ťah, ktorá sa však odvíja len od toho, či je protihráč schopný odhaliť to, že po vykonaní daného ťahu prehrá. Pre určenie, či danú skutočnosť protihráč dokáže odhaliť je využitá len minimálny počet krokov k prehre.

Algoritmy prezentované v [1,3] využívajú pre výber najlepšieho ťahu z aktuálnej pozície alpha-beta prehľadávajúci algoritmus, ktorý sa rozhoduje podľa poslednej úrovne ťahov a pozícií, na ktorú sa pri prehľadávaní pozerá, teda neberie do úvahy ostatné vetvy, ktoré tiež môžu viesť k prehre, na základe predpokladu, že protihráč sa snaží vykonať najlepší ťah.

Na druhej strane algoritmus prezentovaný v [2] berie do úvahy pre výpočty množstvo prehrávajúcich, vyhrávajúcich a remízových ťahov, z ktorých bude mať protihráč na výber a vôbec sa nepozerá na skutočnosť, aká je pravdepodobnosť, že nad daným ťahom bude protihráč vôbec uvažovať alebo ním potiahne.

Ohodnotenie pozície na základe typu vykonaného ťahu, ako bolo prezentované v článku [3] z časti zahŕňa istú mieru pravdepodobnosti, že protihráč daný ťah vykoná, ale nerozlišuje, či hráč dokáže odhaliť možnú prehru po vykonaní ťahu.

3.2 Analýza hry

Ako bolo už v tomto článku spomenuté, budeme pracovať s tým, že nie všetky možné ťahy hráča majú rovnakú pravdepodobnosť, že budú vykonané a preto by sa im pri výbere najlepšieho ťahu nemala prikladať veľká váha.

Napríklad, ak má protihráč na výber z niekoľkých ťahov a len jeden z nich je možnosť zabrániť oponentovi vo vytvorení mlynu na jeden ťah, tak ho rozumný hráč takmer vždy vykoná, keďže nechce prísť o vlastnú figúrku. V takomto prípade je pravdepodobnosť vykonania ostatných ťahov veľmi nízka a dá sa povedať, že aj takmer nulová.

To znamená, že aj keby sa hráč nachádzal z pohľadu možných prehier, do ktorých sa môže z aktuálnej pozície dostať, v ťažkej hernej situácii, z jeho pohľadu sa jeho rozhodovanie zúži na vykonanie ťahu, ktorým zablokuje súperovi mlyn.

Opačná ale podobná situácia nastáva, keď hráč má možnosť vytvoriť na jeden ťah mlyn a nie je ohrozený protihráčom, ktorý by mohol vytvoriť mlyn po jeho ťahu. Vtedy je tiež veľká pravdepodobnosť vykonania daného ťahu a ignorovania ostatných možností, čo opäť vedie k zjednodušeniu situácie.

K ťahu, pri ktorom hráč vytvorí mlyn sa počíta aj vybratie súperovej figúrky. Predpokladajúc pravidlo hry, ktoré nedovoľuje výber protihráčovej figúrky, ktorá je v mlyne, ak existuje figúrka mimo mlynu, bude hráč pravdepodobne preferovať tie figúrky, ktoré umožnia protihráčovi vytvoriť mlyn o dve ťahy a nemá ich medzi tým ako zablokovať. Myslíme si, že ako ďalšie figúrky bude uvažovať tie, ktoré mu bránia vo vytvorení mlynu skôr, ako to môže spraviť súper.

Teda pri výbere figúrky na vyhodenie z hracej plochy hráč môže ignorovať určité ťahy a to tiež mení náročnosť daného herného stavu, v ktorom sa hráč nachádza.

3.3 Návrh

V prvej časti, chceme vytvoriť hodnotiacu funkciu, ktorá ohodnotí všetky možné ťahy vychádzajúce z aktuálnej hernej pozície na základe viacerých parametrov inšpirujúc sa aj existujúcimi prístupmi.

Túto funkciu plánujeme aplikovať iba v prípade, že sa perfektný hráč nachádza v remízovej pozícii, čiže má na výber iba z remízových možných ťahov.

Vstupom funkcie by mal byť aktuálna pozícia, v ktorej sa hráč nachádza a ako výstup očakávame ohodnotenie každého možného (teda platného) ťahu, ktorý sa dá z aktuálnej pozície vykonať. Pričom by malo platiť, že tieto ohodnotenia sú z uzatvoreného intervalu [0, 1] a ich súčet dáva hodnotu 1.

Od hodnotiacej funkcie teda chceme približnú pravdepodobnosť vykonania nasledujúceho ťahu protihráčom, ak by sme si zvolili aktuálny remízový ťah. Pri jej výpočte plánujeme zohľadniť nasledujúce kritériá:

1. počet ťahov, z ktorých má hráč v aktuálnej pozícii na výber (dôležité ak sa pozeráme na poslednú úroveň prehľadávania)
2. počet ťahov do prehry (ak to je prehrávajúci stav)
3. zablokovanie mlynu
4. umožnenie vytvorenia mlynu (hráč pohybom zruší blokovanie mlynu)

5. vytvorenie mlynu
6. znemožnenie opakovania mlynu (protihráčovi je znemožnené pohnúť figúrkou z mlynu a tak ho následne znovu vytvoriť na ďalší ťah)
7. umožnenie opakovania mlynu protihráčovi
8. opakovanie mlynu (pohnutie figúrkou von z mlynu a následne jej vrátenie)

V prípade vytvorenia mlynu, budeme zohľadňovať akú figúrku súperovi vyberie:

1. protihráč môže vytvoriť mlyn (pravdepodobne najviac na 2 ťahy)
2. figúrka blokuje vytvorenie mlynu
3. figúrka blokuje pohyb z iného mlynu
4. figúrka môže na ďalší ťah zablokovať znovu otvorenie mlynu
5. počet figúrok, z ktorých vyberá

Výsledné hodnoty chceme použiť pre modifikáciu pravdepodobnosti výhry (respektíve prehry) po zvolení daného ťahu perfektným hráčom.

Napríklad:
Nech f je vyššie popísaná funkcia a pozeráme sa na remízovú pozíciu, z ktorej vedú pre protihráča tri možné ťahy. Jeden z nich je remíza (R) a dve prehry (P1, P2) z pohľadu protihráča. Pre zjednodušenie predpokladajme, že máme vypočítanú pravdepodobnosť výhry po vykonaní remízového ťahu v(R) = 0.3.

Nech ťah R predstavuje zablokovanie mlynu súpera s ohodnotením f(R) = 0.7, ťah P1 je prehra na 20 ťahov a má ohodnotenie f(P1) = 0.2 a ťahom P2 umožníme protihráčovi uzatvoriť ďalší mlyn s ohodnotením f(P2) = 0.1 a je to prehra na 11 ťahov. Výslednú pravdepodobnosť výhry v aktuálnom remízovom stave určíme tak, že každú pravdepodobnosť potiahnutia, vynásobíme pravdepodobnosťou výhry, ktorá je v prípade prehrávajúceho ťahu 1.

Teda výsledok pre tento príklad by vyzeral tak, že pravdepodobnosť výhry v prípade remízového stavu by bola f(R)*v(R) = 0.21 a celková pravdepodobnosť výhry v aktuálnom remízovom stave by bola daná súčtom 0.21 + 0.1 + 0.2 = 0.51.

V druhej časti plánujeme na základe vytvorenej hodnotiacej funkcie vybudovať databázu najlepších ťahov pomocou retrográdnej analýzy a porovnať výsledný algoritmus s existujúcim na základe výberu náhodného hráča.

3.4 Zistenie aktuálneho stavu hry

Na to, aby sme mohli určiť v akom stave sa aktuálne hra nachádza, teda či je vyhrávajúca, prehravajúca alebo remizujúca, budeme využívať existujúcu databázu herných stavov, ktorá obsahuje okrem herných teoretických hodnôt aj minimálny počet ťahov do ukončenia hry v prípade vyhrávajúcej a prehrávajúcej pozíciie.

Databáza je rozdelená do dvoch hlavných častí. Prvá obsahuje informácie o pozíciach v otváracej fáze hry, teda keď sa figúrky na ukladajú na hernú plochu a druhá pozície z druhej fázy hry. Samotná databáza je uložená vo forme binárnych

súborov, pričom každý súbor má označenie tvaru maticaXYz, kde X predstavuje počet figúrok na ploche pre prvého hráča, Y počet figúrok na hracej ploche pre druhého hráča a z je buď 'b' ak sa hra nachádza v ukladacej fáze alebo 'm' ak sa hra nachádza v druhej fáze.

Každý takýto súbor obsahuje maticu l x k celých nezáporných čísel (persnejšie bytov), ktoré kódujú stav hry. Pre účel tohto kódovania boli určené nasledujúce konštanty:

1. WIN 130
2. LOSS 125
3. DRAW 127
4. UNREACHABLE 128

Za výhru su považované všetky hodnoty väčšie alebo rovné ako WIN. Opačne za prehru sú považované všetky hodnoty menšie alebo rovné ako LOSS. Hodnota remízy je daná jednoznačne konštantou DRAW. Avšak nie všetky herné pozície sú počas hrania reálne dosiahnuteľné. V tom prípade je jej ohodnotenie UNREACHABLE.

Na to aby sme vedeli určiť aj počet ťahov do konca hry, v prípade výhry alebo prehry, použjeme nasledujúci jednoduchú funkciu, ktorá dostane na vstup herný stav a vráti počet ťahov na základe jeho hodnoty.

```
function TURNSTOEND(state)
    if state ≥WIN then
        return ((state − WIN) ∗ 2) − 1
    end if
    if state ≤LOSS then
        return ((LOSS − state) ∗ 2)
    end if
    return 0
end function
```

Rozdiel hernej hodnoty od príslušnej konštanty určoval počet dvojťahov (teda ťah prvého a ťah druhého hráča bol považovaný ako jeden) preto bolo potrebné túto hodnotu vynásobiť dvoma.

Na to, aby sme vedeli získať hrený stav z matice, potrebujeme najprv vedieť, do ktorej matice sa máme pozerať. Na základe fázy hry vieme jednoducho rozlíšiť medzi maticami s koncovkou b a m. Aby bol zredukovaný počet potrebných dát na uloženie, máme len matice z pohľadu jedného hráča. To znamená, že pri výbere matice budeme za prvého hráča považovať toho, ktorý je momentálne na ťahu a jeho protihráča za druhého.

Ako druhú informáciu potrebujeme vedieť riadok a stĺpec, na ktorý sa máme v matici pýtať. Pre tento účel potrebujeme využiť ďalšie matice, ktoré boli pripojené k samotnej databáze:

1. QuickSmall matica tvaru 1 x m slúži na určenie riadka
2. Quick2 matica tvaru 1 x n uchováva číslo stĺpcov
3. Perm24 a Perm12 tvaru m x n slúžia na zistenie permutácií
4. Tran12 matica tvaru m x n pre binárne posunutie

Okrem toho sú ako vstup do funkcie potrebné dve binárne čísla, ktoré kódujú rozloženie figúrok na ploche pre prvého a druhého hráča. Tento binárny kód je definovaný ako suma prvkov $g * 2^{field-1}$, kde $field$ je z množiny $\{1, 2, \ldots, 24\}$ a g je 1 práve vtedy, keď má hráč položenú figúrku na hernom políčku označeného číslom $field$, inak 0.

Binárne číslo prvého hráča kóduje prvok v matici QuickSmall a na základe neho vieme získať dve podstatné informácie. Bity 9 až 32 uchovávajú číslo riadka, na ktorý sa potrebujeme pozerať. Keďže herná plocha hry Nine Mens Morris obsahuje niekoľko symetrií, počet hodnôt v databáze je pomocou nich zredukovaný a z toho dôvodu je potrebné zistiť do ktorej simetrii alebo permutácie patrí aktuálne rozloženie figúrok. Teda druhá informácia získaná na základe tejto hodnoty je číslo permutácie, ktoré by malo byť v rozmedzí od 1 po 16.

Aby bolo možné určiť prislúchajúci stĺpec, je potrebné zisitť kód pre prislúchajúcu pozíciu v matici Quick2. Tá je daná počtom miest, ktoré neobsadzujú figúrky prvého hráča. Okrem toho je táto pozícia v matici ovplyvnená permutáciou rozloženia bielych a čiernych figúrok.

Binárne číslo prvého hráča je rozdelené na dve časti o veľkosti 12 bitov, ktoré spolu s číslom permutácie kódujú pozície v maticiach Perm12 a Perm24 a následne získaný binárny zápis permutácie figúrok prvého hráča pomocou spomínaných matíc. Podobne sa získa binárny zápis permutácie figúrok druhého hráča.

Následne sa získajú prvé a druhé časti permutácií pre prvého a druhého hráča, ktoré kódujú pozície v matici Tran12, čo slúži na upravenie pozície pre maticu Quick2.

```
function getLineAndCollumn(bbinar1, bbinar2)
    value ← QuickSmall[bbinar1]
    numberOfPerm ← (value ≫ 24)&31
    line ← value & bits9_32
    if numberOfPerm > 16 or numberOfPerm < 1 then
        numberOfPerm ← 1
    end if
    whitePart1 ← bbinar1 ≫ 12
    whitePart2 ← bbinar1 & bits21_32
    binar1Perm ← Perm24[numberOfPerm][whitePart1] | Perm12[numberOfPerm][whitePart2]
    countOfOnes ← countOfOnes[whitePart1] + countOfOnes[whitePart2]
    if countOfOnes = 0 then
        sum ← 0
    else
        sum ← power2[24] - power2[25-countOfOnes]
    end if
    blackPart1 ← bbinar2 ≫ 12
    blackPart2 ← bbinar2 & bits21_32
```

```
    blackPerm ← Perm24[numberOfPerm][blackPart1] | Perm12[numberOfPerm][blackPart2]
    blackPart2Perm ← blackPerm & bits21_32
    whitePart2 ← binar1Perm & bits21_32
    moveNumbers ← countOfOnes[whitePart2]
    blackPart1Perm ← blackPerm ≫ 12
    whitePart1 ← binar1Perm ≫ 12
    tran1 ← Tran12[blackPart2Perm][whitePart2]
    tran2 ← Tran12[blackPart1Perm][whitePart1]
    sum ← sum | tran1 | tran2 ≪ (12 - moveNumbers)
    collumn ← Quick2[sum]
    return [line, collumn]
end function
```

Keďže prehľadávanie súboru s maticou vo forme riadok, stĺpec by bolo náročné, je potrebné na základe získaných informácií vypočítať poradie prvku v matici. Pre tento účel potrebujeme poznať riadok, stĺpec a počet stĺpcov v matici. Prvé dve údaje poznáme pomocu predchádzajúcej funckie. Počet stĺpcov v matici sa dá zistiť z faktu, že pre každú maticu, na ktorú sa budeme v tomto kroku pozerať platí, že počet jej stĺpcov sa dá vyjadriť ako kombinačné číslo pomocou počtu figúrok prvého a druhého hráča.

```
function getIndexOfElement(line, column, first, second)
    numbreOfColumns ← C(24−first, second)
    index ← ((line − 1) ∗ numberOfColumns) + (column − 1)
    return index + 1
end function
```

Teda výsledná funkcia pre zistenie herného stavu vyzerá nasledovne. Na vstup dostane binárne kódy hráčov a počty ich uložených figúrok. Výstupom je herný stav, ktorý kóduje hernú teoretickú hodnotu a počet ťahov do konca hry.

```
function getState(binar1, binar2, first, second)
    [line, column] ← getLineAndCollumn(binar1, binar2)
    index ← getIndexOfElement(line, column, first, second)
    return get element of matrix first x second at position index
end function
```

3.5 Zistenie možných ťahov

Na to aby bolo možné použiť minmax alebo alpha-beta prehľadávanie, je potrebné zistiť aké sú možné ťahy z aktuálnej hernej pozície. Ťah je definovaný ako usporiadaná trojica (from, to, remove), kde každá zložka označuje jedno z políčok na hracej ploche, alebo je nepoužitá. Pričom prvá zložka *from* určuje ktorou figúrkou sa bude ťahať. Ak je prvá zložka nevyužitá, figúrka sa bude ukladať. Druhá zložka *to* určuje políčko hracej plochy, na ktoré sa figúrka presunie alebo položí (závisí od prvej zložky). A nakoniec tretia zložka *remove* je definovaná práve vtedy, keď hráč vytvoril mlyn a určuje, z ktorého políčka má odstrániť figúrku.

Poznamenajme, že chceme generovať iba platné ťahy a teda to znamená, že *from* nemôže ukazovať na políčko, na ktorom nie je figúrka hráča, ktorý vykonáva daný ťah. Ďalej *to* musí ukazovať na prázdne herné políčko a v prípade, že hra nie je v ukladacej fáze a hráč, ktorý je na ťahu nemôže skákať, tak políčka *from* a *to* musia byť spojené čiarou na hernom pláne, teda musí byť medzi nimi povolený prechod. Nakoniec *remove* nemôže ukazovať na políčko s figúrkou hráča, ktorý ťah vykonáva. A v prípade, že hráč vytvoril mlyn a protihráč má nejaké figúrky, ktoré netvoria mlyn, tak *remove* musí ukazovať na jednu z nich.

Nižšie uvedená funkcia *getPossibleMoves* dostane na vstupe počet figúrok, ktoré má hráč na ťahu uložené na hracej ploche a identifikátor hráča, ktorý je na ťahu. Ako výstup vráti množinu možných ťahov na základe aktuálneho stavu na hracej ploche *board*, ktorá má 25 políčok a každé obsahuje kód hráča, ktorý ma na ňom práve uloženú figúrku, alebo má označenie voľného miesta.

Najprv je potrebné overiť, či má protihráč všetky figúrky v mlynoch, alebo má nejaké voľné, aby bolo možné sa pri vyhadzovaní rozhodnúť podľa pravidiel. Ak sa hra nachádza v ukladacej fáze, funkcia prechádza cez voľné políčka na hracej ploche a postupne na ne skúša položiť figúrku.

Ak by daným uložením figúrky bol vytvorený mlyn, tak následne prechádza cez políčka s figúrkami protihráča a skúša ich postupne vyhadzovať. Pri tomto pokuse o vyhodenie, najprv overí, či daná figúrka nie je náhodou v mlyne. Ak áno a protihráč má figúrky mimo mlyna, tak ju nevyhodí a pokračuje v hľadaní. V prípade, že nájde figúrku, ktorú môže vyhodiť, vytvorí nový ťah $[nepouitpozcia, to, remove]$ a pridá sa do zoznamu možných ťahov. Pričom *to* je pozícia položenej figúrky a *remove* je pozícia figúrky, ktorá sa vyhadzuje.

Ak daným uložením figúrky by mlyn nevznikol, tak sa vytvorí ťah, v ktorom bude použitá len zložka *to* a bude označovať políčko uloženia figúrky. Tiež sa tento ťah pridá do množiny možných ťahov.

Ak sa hra nenachádza v ukladacej fáze, tak funckia prechádza tými políčkami hracej plochy, na ktorých sú figúrky hráča, ktorý je práve na ťahu. Pre každé nájdené platné políčko, hľadá voľné políčko na hracej ploche. V prípade, že hráč nemôže skákať, teda má viac ako tri figúrky na hracej ploche, tak sa skontroluje, či je táto dvojica políčok spojená a teda či je medzi nimi platný pohyb figúrky.

Ďale pre túto dvojicu pokračuje analogicky ako v prípade ukladania figúrky s tým rozdielom, že ak nájde platný ťah, tak prvá zložka ťahu, ktorý bude pridaný medzi možné ťahy bude mať označenie políčka, z ktorého je figúrka premiestňovaná a druhá zložka bude číslo políčka, na ktoré je figúrka umiestnená. Na konci funkcia vráti zoznam všetkých možných a zároveň platných ťahov pre hráča, ktorý je na rade.

```
function getPossibleMoves(placed, playerOnMove)
    hasOponentAllInMill ← true
    for p from 0 to 24 do
```

```
    if board[p] = NO_FIGURE then
        continue
    end if
    if board[p] = playerOnMove then
        continue
    end if
    if HASMILL(p) then
        continue
    end if
    hasOponentAllInMill ← false
end for
moves ← emptyset
if PHASE_PLACING then
    for p from 0 to 24 do
        if board[p] ≠ NO_FIGURE then
            continue
        end if
        if WILLHAVEMILL(p, playerOnMove) then
            for x from 0 to 24 do
                if board[x] = NO_FIGURE then
                    continue
                end if
                if board[x] = playerOnMove then
                    continue
                end if
                if HASMILL(x) and !hasOponentAllInMill then
                    continue
                end if
                move ← [NOT_USED, p, x]
                push move into moves
            end for
        else
            move ← [NOT_USED, p, NOT_USED]
            push move into moves
        end if
    end for
else if PHASE_MOVING then
    playerCanJump ← placed ≤ 3
    for p from 0 to 24 do
        if board[p] ≠ playerOnMove then
            continue
        end if
        for k from 0 to 24 do
            if board[k] ≠ NO_FIGURE then
                continue
```

```
                end if
                if !playerCanJump and !canBeMoved(p, k) then
                    continue
                end if
                if willHaveMill(p, k, playerOnMove) then
                    for x from 0 to 24 do
                        if board[x] = NO_FIGURE then
                            continue
                        end if
                        if board[x] = playerOnMove then
                            continue
                        end if
                        if hasMill(x) and !hasOponentAllInMill then
                            continue
                        end if
                        move ← [p, k, x]
                        push move into moves
                    end for
                else
                    move ← [p, k, NOT_USED]
                    push move into moves
                end if
            end for
        end for
    end if
    return moves
end function
```

Literatúra

1. LINCKE, Thomas. Perfect Play using Nine Mens Morris as an example, Diploma Thesis Department of Computer Science ETH Zürich
2. GÉVAY, Gábor E.; DANNER, Gábor. Calculating Ultrastrong and Extended Solutions for Nine Men's Morris, Morabaraba, and Lasker Morris. IEEE Transactions on Computational Intelligence and AI in Games
3. BOYD, Martin; HIRUNTHANAKORN, Christopher. Analyzing Nine Men's Morris For a Optimal Strategy. 2012.
4. CHESANI, Federico, et al. Can Deep Networks Learn to Play by the Rules? A Case Study on Nine Men's Morris. IEEE Transactions on Games, 2018.