Obsah

1. Úvod 2
2. Teoretické pozadie 2
2.1 História šachu 2
2.2 Základné pravidlá 2
2.3 Minimax algoritmus 3
2.4 Alfa-beta orezávanie 3
3. Analýza problému 3
3.1 Špecifikácia obmedzení 3
3.2 Problémy pri modelovaní 4
4. Návrh riešenia 5
4.1 Architektúra aplikácie v NetLogo 5
5. Implementácia 6
5.1 Procedúra setup 6
5.2 Procedúra make-piece 7
5.4 Procedúra best-move 8
5.5 Procedúra minimax (s alfa-beta orezávaním) 10
5.7 Procedúra evaluate 12
5.7 Procedúra position-key a record-position 13
6. Testovanie a experimenty 14
9. Záver 16

Modelovanie komplexných systémov a multiagentové simulácie

Bc.Pavol Lukačka

Semestrálna práca

„Pawns and Kings Chess“

1. Úvod

Hra šach je jednou z najznámejších stolových hier na svete, v ktorej dvaja protihráči ovládajú svoje figúry na 64-polovej šachovnici. V klasickom šachu každý hráč má 16 figúr, vrátane kráľa a ôsmich pešiakov en.wikipedia.org. Cieľom hry je dať šach a mat súperovmu kráľovi. Šach podporuje rozvoj strategického myslenia a je často používaným príkladom pri výučbe umelej inteligencie a algoritmického rozhodovania. Motiváciou tohto projektu je vytvorenie zjednodušenej verzie šachovej partie s obmedzeným počtom figúr (kráľmi a pešiakmi) v prostredí NetLogo. NetLogo umožňuje vizualizovať dynamické modely a simulácie viacagentových systémov, čo je vhodné aj pre modelovanie šachových partií. Cieľom dokumentácie je podrobne opísať tvorbu projektu „Pawns and Kings Chess“: jeho teoretické východiská, analýzu problému, návrh riešenia, implementáciu a testovanie. V práci tiež porovnáme čistý minimax s alfa-beta pruningom (orezávaním) a načrtneme možné vylepšenia.

2. Teoretické pozadie

2.1 História šachu

Šachy majú bohatú históriu, ktorá siahá do staroveku. Ich predchodcom bola indická hra **čaturanga** zo 6. storočia, odkiaľ sa šíril do Perzie a odtiaľ do moslimského sveta. Následne sa prostredníctvom Španielska a Sicílie dostal do Európy. Moderné pravidlá šachu sa ustálili približne v 15. storočí v Európe. Je zaujímavé, že práve v 19. a 20. storočí sa rozvinuli aj teórie a počítačové analýzy šachu. V roku 1997 počítač Deep Blue porazil v zápase svetového šampióna Garryho Kasparova, čo značne zintenzívnilo vývoj počítačových šachov en.wikipedia.org. Odvtedy sa algoritmy pre šachovú hru (napr. Stockfish, Komodo) stali veľmi silnými. Práve štúdium šachových algoritmov (minimax, alfa-beta atď.) je dôležité pre pochopenie rozhodovacích procedúr v hre, a preto sme sa rozhodli vytvoriť zjednodušenú šachovú hru s kráľmi a pešiakmi v prostredí NetLogo.

2.2 Základné pravidlá

Šach sa hrá na štvorcovej šachovnici 8×8 polí, ktoré sú striedavo svetlé a tmavé. V klasickom šachu každý hráč ovláda 16 figúr šiestich typov (kráľ, kráľovná, veže, strelci, kone a pešiaci) en.wikipedia.org. V tomto projekte však používame obmedzenú verziu: na každej strane je iba jeden **kráľ** a osem **pešiakov**. Základné pohyby týchto figúr sú nasledovné:

- **Kráľ** sa môže pohnúť o jedno políčko v ľubovoľnom smere (horizontálne, vertikálne aj diagonálne), ak pole nie je obsadené vlastnou figurou. Kráľ sa nemôže pohnúť mimo šachovnice.
- **Pešiak** sa pohybuje vpredu (smerom k súperovej strane). Môže sa pohybovať o jedno políčko priamo vpredu, ak je toto políčko voľné. Ak sa pešiak ešte nepohol (t. j. je na počiatočnej

pozícii), môže sa posunúť o dve políčka vpredu, ak sú obe polia pred ním voľné. Pešiaci **neposkakujú cez figúry** a nikdy sa nemôžu pohnúť dozadu. Pešiaci sa líšia od ostatných figúr v spôsobe braní: môžu zobrať nepriateľskú figúru, ktorá stojí na jednom z dvoch diagonálnych políčok pred nimi en.wikipedia.org.

Okrem pohybov je dôležitý aj cieľ hry: v bežnom šachu je cieľom dať mat súperovmu kráľovi – to znamená donútiť kráľa ísť na napadnuté pole, kde nie je možné uniknúť . V našej zjednodušenej verzii je cieľom **chytiť kráľa** – hra končí okamžite, keď jeden hráč zachytí súperovho kráľa, pretože kráľ už nemôže byť pohybovaný (čiže akoby padol šachmat).

2.3 Minimax algoritmus

Minimax je algoritmus využívaný v teorii hier a vyhľadávaní v hernom strome na rozhodovanie o najlepšom ťahu geeksforgeeks.org. Predpokladá, že obaja hráči hrajú optimálne. Algoritmus rekurzívne prehľadá všetky možné ťahy (a protiťahy) do určitej hĺbky (popisu možných výsledkov hry) a vyhodnotí konečné pozície pomocou hodnotiacej funkcie (evaluačnej funkcie). Potom sa výsledné hodnoty vrátia smerom hore stromom: pri ťahu nášho hráča (maximalizujúceho hráča) vyberáme ťah s maximálnou hodnotou, a pri ťahu protivníka (minimalizujúceho hráča) vyberáme ťah s minimálnou hodnotou. Takto sa algoritmus rozhodne pre optimálny ťah pri danom stave. Minimax sa často používa v hrách ako Tic-Tac-Toe, šach, dáma a ďalších, kde sú dvaja hráči ťahujúci striedavo geeksforgeeks.org. Hlavnou nevýhodou klasického minimaxu je exponenciálna časová náročnosť (počet uzlov rastie veľmi rýchlo s hĺbkou prehľadávania), čo nás motivuje použiť aj **alfa-beta orezávanie** pre zrýchlenie (ďalej v bode 2.4).

2.4 Alfa-beta orezávanie

Alfa-beta orezávanie (angl. *alpha-beta pruning*) je vylepšenie minimax algoritmu, ktoré umožňuje výrazne znížiť počet hodnotených stavov bez zmeny výsledného ťahu en.wikipedia.org geeksforgeeks.org. Pri prehľadávaní stromu si algoritmus udržiava dve hodnoty: **alfa** (najlepšia dosiahnuteľná hodnota pre maximalizujúceho hráča) a **beta** (najlepšia pre minimalizujúceho). Keď sa pri prehľadávaní objaví ťah, ktorý je horší ako už nájdená alternatíva (t. j. alfa ≥ beta), algoritmus preruší skúmanie zvyšných ťahov tohto uzla (prune). Týmto spôsobom sa ignorujú vetvy stromu, ktoré už nemôžu zlepšiť výsledok. Výhodou je dramatické zrýchlenie. Alfa-beta stále vracia rovnaký optimálny ťah ako čistý minimax, len menší počet podproblémov sa skutočne vyhodnotí. Vo výsledku prevažná časť hracieho stromu môže byť orezaná, čo umožňuje hľadať hlbšie v rovnako dlhom čase.

3. Analýza problému

3.1 Špecifikácia obmedzení

Projekt „Pawns and Kings Chess“ obsahuje niekoľko významných obmedzení a zjednodušení oproti plnému šachu. Najdôležitejšie sú:

- **Len králi a pešiaci:** Na každej strane je iba jeden kráľ a osem pešiakov. Žiadne iné figúry (kráľovné, veže, strelci, kone) sa nepoužívajú. To zásadne znižuje počet možných ťahov a štandardné i špeciálne situácie.
- **Bez promócie:** Pešiaci sa pri dosiahnutí koncového radu nemenia na inú figúru. Bežne by pešiak dosiahol postupom posledný rad a premenil sa na kráľovnú alebo inú figúru, ale tu je takáto možnosť vypnutá. Pešiaci po dosiahnutí základne protivníka ostávajú pešiakmi.

- **Bez en passant:** Špeciálny ťah „en passant“ nie je implementovaný. To znamená, že ak by čierny pešiak išiel z druhej rady o dva ťahy a biely pešiak mal možnosť eliminovať ho diagonálne, túto možnosť nemá. (Ak by sa zaoberali situácie en passant, museli by sme kontrolovať krátkodobú históriu ťahov.)
- **Zachytenie kráľa ako cieľ:** Hra končí okamžite, keď jeden hráč zachytí súperovho kráľa. Normálny šachmat (kde kráľ nie je priamo zobrať) sa tu nekoná; ak je kráľ v situácii, že ho súper môže vziať, takýto ťah vedie k ukončeniu hry.

Tieto obmedzenia zjednodušujú model a zároveň vylučujú mnohé komplikované situácie (napr. figúry s komplikovanými ťahmi či výhrou materiálu cez promócie). Projekt sa tak zameriava na koncentrovaný problém minimaxového vyhľadávania v relatívne obmedzenom priestore stavov.

3.2 Problémy pri modelovaní

Pri tvorbe modelu šachovej hry v NetLogo sme identifikovali niekoľko výziev:

- **Reprezentácia pozície:** Treba efektívne reprezentovať aktuálnu pozíciu na šachovnici tak, aby bolo možné generovať pohyby a zároveň potenciálne detegovať opakovanie pozícií. V prostredí NetLogo môžeme každému poli reprezentovať pozíciu (napr. pomocou súradníc patchov) a figúrky (tortle-ov) umiestniť na tieto polia.
- **Striedanie hráčov:** Je potrebné riadiť, ktorý hráč je na ťahu („biela“ alebo „čierna“). To sa spravuje globálnou premennou (napr. side-to-move).
- **Generovanie legálnych ťahov:** Treba implementovať procedúry na generovanie všetkých legálnych ťahov pre danú stranu. Aj keď je hra zjednodušená, počet legálnych ťahov v každom kroku môže byť desiatky, čo ovplyvní čas vyhľadávania. Navyše sa musia brať do úvahy obmedzenia (žiadne vlastné zachytenie, kontrola hraníc dosky, atď.).
- **Backtracking ťahov:** Pri simulácii ťahov a prehľadávaní musíme mať možnosť vrátiť pozíciu naspäť (undo move). Každý ťah obsahuje informácie o premiestnení figúry a prípadnom zobraťi susednej figúry. Preto ukladáme históriu ťahov (ako zásobník), z ktorého vieme pri „undo-move“ obnoviť predchádzajúci stav.
- **Vyhodnocovanie konečného stavu:** V zjednodušenej hre s kráľom a pešiakmi existuje jednoznačné koncové kritérium (zachytenie kráľa), ale nie je počiatočne definovaná „váha“ pozície okrem tejto udalosti. Musíme preto navrhnúť evaluačnú funkciu, ktorá v ne-koncových pozíciách poskytne číselné ohodnotenie „výhody“ jedného alebo druhého hráča (viac o tom v bode 5. Implementácia).

Tieto problémy určujú náš návrh riešenia a implementačnú stratégiu v nasledujúcich kapitolách.

4. Návrh riešenia

4.1 Architektúra aplikácie v NetLogo

Aplikácia je vytvorená v prostredí NetLogo, ktoré umožňuje modelovať svet rozdelený na polia (patches) a agenty (turtles). V našom modeli použijeme polia na reprezentáciu šachovnice (8×8) a tortly ako figúrky (kráľov a pešiakov). Práca je postavená na niekoľkých častiach:

- **Globálne premenné:** Definujú základné údaje, napríklad side-to-move (strana na ťahu), move-number (počet ťahov), nodes-searched (počet preskúmaných uzlov za ťah), prípadne premenné pre výstup do grafov (napr. last-eval – posledné vyhodnotenie, pawn-balance – rozdiel počtu pešiakov).
- **Turtles a ich vlastnosti:** Každá figúrka je turtle so zodpovedajúcimi vlastnosťami (turtles-own), napr. typ (kráľ alebo pešiak) a farba (biela/čierna). Kráľ a pešiaci majú nastavené tvary („shape“) podľa potreby – importované z knižnice shapes.
- **Dátové štruktúry:** Používame:
    - *Tabuľky* (s využitím NetLogo extension table), v ktorých môžeme uchovávať napríklad vyhodnotenia pozícií alebo počítať, koľkokrát sa daná pozícia objavila. Kľúčom by mohol byť reťazec alebo vlastný identifikátor pozície.
    - *Zásobník ťahov* (napríklad zoznam move-history), do ktorého pri každom vykonaní ťahu vložíme záznam o ťahu (t.j. aká figúrka kam sa pohla a či bola nejaká figúrka zničená). Tento zásobník umožní funkciu **undo-move**, ktorá vráti posledný ťah.
- **Správa opakovaných pozícií:** Pri vyhľadávaní budeme sledovať, či sa v hĺbke vyhľadávania nenájde už predtým navštívená pozícia. Môžeme použiť tabuľku s kľúčmi pozícií. Ak sa pozícia opakuje (v rôznej vetve), môžeme minimalizovať opakovanú prácu – buď vrátiť predchádzajúce hodnoty, alebo z nej skončiť (víťazstvom/dosiahnuť remízu). Takto sa znižuje počet redundantných výpočtov a zvyšuje sa efektivita.

Na rozhraní používame tlačidlá a monitore (ako je vidieť na obrázku úvodného stavu v Nasledujúcej kapitole). Medzi hlavné tlačidlá patria **Setup** (inicilizácia partie), **Run whole game** (spustenie celej simulácie) a **Step** (vykonanie jedného ťahu). Grafy (Plots) zobrazujú priebežné informácie: hodnotenie pozície, rozdiel materiálu, počet prehľadaných uzlov či vzdialenosť kráľov (metrika analyzovaná v experimentoch).

Celá architektúra je riadená z procedúr NetLogo, ktoré nazývame zo **Setup** tlačidla. Program najprv vykreslí šachovnicu, potom rozmiestni figúrky, nakoniec je pripravený k simulácii ťahov prostredníctvom volania procedúr pre generovanie ťahov a vyhodnotenie najlepšieho ťahu.

5. Implementácia

5.1 Procedúra setup

Procedúra setup zabezpečuje počiatočné nastavenie hry. Vyčistí svet, nafarbí šachovnicu a umiestni figúrky do počiatočných pozícií:

```
to setup
  clear-all
  resize-world 0 7 0 7           ;; 8 × 8 šachovnica
  set shape-map table:from-list [
    ["pawn" "chess pawn"]     ;; vstavané tvary
    ["king" "chess king"]    ;; prípadne "crown" / "circle"
  ]
  ask patches [                  ;; vytvor šachovnicový vzor
    set pcolor ifelse-value ((pxcor + pycor) mod 2 = 0) [113] [33]
  ]
  set move-stack []
  set total-nodes 0
  set last-eval evaluate
  ;; --- evidencia opakovaní -----------------------------------
  set pos-counts table:make
  set repetition-limit 3         ;; zmeň na 2 pre dvojité opakovanie
  record-position                ;; ulož začiatočnú pozíciu

  ;; --- príprava grafov ---------------------------------------
  set-current-plot "Evaluation"       clear-plot
  set-current-plot-pen "eval"
  set-current-plot "Nodes/ply"       clear-plot
  set-current-plot-pen "nodes"
  set-current-plot "Material diff"   clear-plot
  set-current-plot-pen "pawns"
  set-current-plot "King distance"   clear-plot
  set-current-plot-pen "distance"

  set turn "white"
  make-start-position
  reset-ticks


to make-start-position             ;; klasická úvodná zostava
  ;; pešiaci
  foreach range 8 [ i ->
    ask patch i 1 [ make-piece "pawn" "white" ]
    ask patch i 6 [ make-piece "pawn" "black" ]
  ]
  ;; králi
  ask patch 4 0 [ make-piece "king" "white" ]
  ask patch 4 7 [ make-piece "king" "black" ]
end
```

5.2 Procedúra make-piece

Táto procedúra robí jednoznačnú vec: vytvorí jedného nového turtle-a a nastaví mu breed (typ figúrky), color a shape (tvar, napr. "king" alebo "pawn")

```
to make-piece [kind col]            ;; vytvor figúrku na aktuálnom patchi
  let shp table:get shape-map kind
  sprout-pieces 1 [
    set shape shp
    set size 1
    set ptype kind
    set side  col
    set color ifelse-value col = "white" [white] [black]
  ]
end
```

5.3 Procedúra legal-moves

Procedúra (resp. reporter) legal-moves vráti zoznam všetkých legálnych ťahov aktuálnej strany:

- Pre každý turtle figúrky (iba tých, ktorých farba zodpovedá premennej side-to-move) sa skontroluje typ – či je to kráľ alebo pešiak. Podľa toho generujeme ťahy.
- **Králove ťahy:** Kráľ môže ísť na susedné políčka (dx = -1,0,1; dy = -1,0,1, okrem (0,0)). Kontrolujeme, či cieľové pole je v rámci dosky a nie je obsadené vlastnou figúrou (jednoducho, aby nenastalo samo-zobratie):
- **Pešiakove ťahy:** Pešiak sa pohybuje inak podľa farby:
    - Ak je biely pešiak (color = white), pohybuje sa nahor (zvyšovanie y súradnice). Skontrolujeme políčko vpredu o jedno (y+1). Ak je voľné, pridáme tento ťah. Ak je pešiak ešte na svojom počiatočnom riadku (y = -6 v našom nastavení) a voľné sú obe políčka vpredu (o jedno aj dve), pridáme aj dvojitý ťah (y+2).
    - Počiatočný pohyb pre dva ťahy je povolený, pretože typický šachov smer je taký (aj keď bez en passant).
    - Pri bielom pešiakovi kontrolujeme dve diagonálne polia vpredu doľava (x-1, y+1) a doprava (x+1, y+1). Ak na nich stojí čierna figúrka (nepriateľský kráľ alebo pešiak), môžeme tam skočiť a „zobrať“ ju. Podobne pre čierneho pešiaka, len smer je naopak (nastúpa na y-1).
    - (En passant nie je implementované, takže ignorujeme špeciálny prípad, keď by to bolo možné.)

Pri generovaní ťahov teda vynecháme tie, ktoré by posunuli figúrku mimo šachovnice, alebo by odstránili vlastnú figúrku. Nakoniec legal-moves zoberie všetky také dvojice [figúrka novýX novýY] a vracia ich ako zoznam (list) možných ťahov.

```
to-report legal-moves [col]
  let mv []
  ask pieces with [side = col] [
    let local []
    ;; pešiak ------------------------------------------------
    if ptype = "pawn" [
      let dir ifelse-value side = "white" [1] [-1]
      let fwd patch-step 0 dir
      if (fwd != nobody) and not any? pieces-on fwd [
        set local lput (list self fwd) local
      ]
      foreach [-1 1] [d-x ->
        let tgt patch-step d-x dir
        if (tgt != nobody) and any? (pieces-on tgt) with
           [side != [side] of myself] [
          set local lput (list self tgt) local
        ]
      ]
    ]
    ;; kráľ ---------------------------------------------------
    if ptype = "king" [
      foreach [[1 0] [-1 0] [0 1] [0 -1]
               [1 1] [1 -1] [-1 1] [-1 -1]] [d ->
        let tgt patch-step (item 0 d) (item 1 d)
        if (tgt != nobody) and
           (not any? pieces-on tgt or
            [side] of one-of pieces-on tgt != side) [
          set local lput (list self tgt) local
        ]
      ]
    ]
    set mv sentence mv local
  ]
  report mv
end
```

5.4 Procedúra best-move

Procedúra best-move určí najlepší ťah pre súčasnú stranu na ťahu, pomocou algoritmu minimax v kombinácii s alfa-beta, postupuje takto:

1. Zavolá sa legal-moves pre aktuálnu stranu, čím sa získa zoznam všetkých ťahov (kombinácia figúrka + cieľové súradnice).
2. Pre každý taký ťah:
    - **Simulujeme ťah**: vykonáme ho (t.j. premiestnime figúrku, prípadne odstránime nepriateľskú figurku).
    - Zavoláme rekurzívne minimax-s-pruning so zníženou hĺbkou a s prehodenou roľou maximizer/minimizer.

- **Vrátime ťah späť**: pomocou undo-move (viď nižšie) obnovíme pôvodný stav.
- Porovnáme získané ohodnotenie s doterajším „najlepším“. Ak ide o ťah maximalizujúceho hráča (biela), budeme hľadať najväčšiu hodnotu; pre minimalizujúceho (čierna) najmenšiu.

3. Výsledkom je ťah, ktorý maximalizuje (resp. minimalizuje) ohodnotenie pozície.

```
to-report best-move [col depth alpha beta]
 let best []
 let moves legal-moves col
 if empty? moves [ report best ]

 ifelse col = "white"
 [                                    ;; MAX hráč
   let value -999
   let i 0
   while [i < length moves and beta > alpha] [
     let m item i moves
     set i i + 1
     do-move m
     let s minimax (opposite col) (depth - 1) alpha beta
     undo-move
     if s > value [ set value s set best m ]
     set alpha max (list alpha value)
   ]
 ]
 [                                    ;; MIN hráč
   let value 999
   let i 0
   while [i < length moves and beta > alpha] [
     let m item i moves
     set i i + 1
     do-move m
     let s minimax (opposite col) (depth - 1) alpha beta
     undo-move
     if s < value [ set value s set best m ]
     set beta min (list beta value)
   ]
 ]
 report best
end
```

5.5 Procedúra minimax (s alfa-beta orezávaním)

Implementujeme minimax rekurzívne so zbieraním alfa a beta.

Na začiatku kontrolujeme ukončujúce podmienky: ak depth = 0 (dosiahnutá hĺbka) alebo ak je hra ukončená (kráľ chytený, tzv. *game-over?*), zavoláme evaluate, čo vráti ohodnotenie pozície.

- Pre maximalizujúcu fázu (biely hráč) iterujeme cez všetky možné ťahy bieleho, vykonáme ťah, a voláme ďalej minimax pre opačnú fázu (tentokrát minimalizujúcu) a hĺbku o 1. Po návrate undo-move obnoví pozíciu.
- Pre minimalizujúcu (čierny) je postup symetrický.
- Alfa-beta rez pracuje tak, že ak sa hodnota beta stane menšou alebo rovnou alpha, prerušíme ďalšie prehľadávanie v tejto vetve.

```
to-report minimax [col depth alpha beta]
 set nodes-searched nodes-searched + 1
 if depth <= 0 [ report evaluate ]
 let moves legal-moves col
 if empty? moves [ report evaluate ]

 ifelse col = "white"
 [                                  ;; MAX
   let val -999
   let i 0
   while [i < length moves and beta > alpha] [
     let m item i moves
     set i i + 1
     do-move m
     let s minimax (opposite col) (depth - 1) alpha beta
     undo-move
     if s > val [ set val s ]
     set alpha max (list alpha val)
   ]
   report val
 ]
 [                                  ;; MIN
   let val 999
   let i 0
   while [i < length moves and beta > alpha] [
     let m item i moves
     set i i + 1
     do-move m
     let s minimax (opposite col) (depth - 1) alpha beta
     undo-move
     if s < val [ set val s ]
     set beta min (list beta val)
   ]
   report val
 ]
end
```

Pri simulovaní ťahov v algoritme minimax potrebujeme vedieť nielen vykonať ťah, ale aj vrátiť stav hry späť. To zabezpečujú dve procedúry: **do-move** (vykonanie ťahu) a **undo-move** (vrátenie ťahu).

do-move spraví ťah nasledovne:

- Zistí, ktorá figúrka sa má pohnúť (mover) a kam (tgt).
- Zapamätá si pôvodné miesto figúrky (orig).
- Ak je na cieľovom poli súperova figúrka, zaznamená jej typ a stranu a odstráni ju (branie).
- Figúrku presunie na nové miesto.
- Všetky informácie o ťahu (figúrka, pôvodné miesto, cieľ, typ a strana zobratej figúrky) uloží do zásobníka move-stack, aby sa ťah dal neskôr vrátiť.

undo-move vráti posledný ťah takto:

- Zo zásobníka move-stack si načíta posledný záznam o ťahu.
- Figúrku vráti na jej pôvodné miesto.
- Ak bola pri ťahu zobratá figúrka, na cieľovom mieste ju znovu vytvorí.

```
to do-move [m]                  ;; m = [piece tgt]
  let mover first m
  let tgt   last  m
  let orig  [patch-here] of mover

  ;; branie?
  let cap-kind nobody
  let cap-side nobody
  if any? pieces-on tgt [
    let victim one-of pieces-on tgt
    set cap-kind [ptype] of victim
    set cap-side [side]  of victim
    ask victim [ die ]
  ]

  ask mover [ move-to tgt ]
  set move-stack fput (list mover orig tgt cap-kind cap-side) move-stack
end


to undo-move
  let rec first move-stack
  set move-stack but-first move-stack
  let mover     item 0 rec
  let orig      item 1 rec
  let tgt       item 2 rec
  let cap-kind  item 3 rec
  let cap-side  item 4 rec
  ask mover [ move-to orig ]
  if cap-kind != nobody [
    ask tgt [ make-piece cap-kind cap-side ]
  ]
end
```

5.7 Procedúra evaluate

Hodnotiaca funkcia evaluate priraďuje číselnú hodnotu aktuálnej pozícii z pohľadu jedného hráča (napr. záporné pre čierne, kladné pre biele). Pre našu hru je jednoduchá metrika napríklad:

- **Materiál:** Rozdiel počtu pešiakov bieleho a čierneho (každý pešiak môže mať váhu napr. 1 bod). V ideálnom prípade začne biely so ziskom 0, pretože oba majú 8 pešiakov. Keď biely zoberie čierneho pešiaka, zvýši sa jeho skóre.
- **Rozdiel kráľ – cieľ:** Hlavný cieľ je chytiť kráľa, takže ak je kráľ súperov zničený, získame extra vysoké skóre (napr. +1000 pre výhru bieleho, alebo -1000 pre stratu bieleho).
- **Vzdialenosť kráľov:** Môžeme zobrať aj vzdialenosť medzi oboma kráľmi. Menšia vzdialenosť je výhodnejšia pre hráča, ktorý má na ťahu (bližší súboj). Napríklad, nech dist je Manhattanova alebo euklidovská vzdialenosť medzi kráľmi. Môžeme pridať hodnotu ako 10 - dist, aby bližší králi (dist=0 až 10) dali lepšiu situáciu.

Takto implementovaná evaluácia postupne počas hry rastie alebo klesá. Napríklad keď biely zoberie čierneho pešiaka, score stúpne; ak králi sú blízko, hodnota stúpne (pridáme 10-dist, kde dist môže byť malý), čo môže stimulovať bieleho hráča k útoku kráľa. Toto ohodnotenie nie je dokonalé, ale postačuje na riadenie simulácie a porovnanie variantov algoritmu.

```
to-report evaluate
  let score 0
  ask pieces [
    set score score +
      (ifelse-value ptype = "pawn" [1] [100]) *
      (ifelse-value side = "white" [1] [-1])
  ]
  report score
end


to-report opposite [col]
  report ifelse-value col = "white" ["black"] ["white"]
end

; ----------------------  ANALÝZA STAVU -------------------------

to-report pawn-balance
  report (count pieces with [ptype = "pawn" and side = "white"]) -
         (count pieces with [ptype = "pawn" and side = "black"])
end


to-report king-distance
  let k1 one-of pieces with [ptype = "king" and side = "white"]
  let k2 one-of pieces with [ptype = "king" and side = "black"]
  if (k1 = nobody) or (k2 = nobody) [ report 0 ]
  report max (list abs ([pxcor] of k1 - [pxcor] of k2)
                   abs ([pycor] of k1 - [pycor] of k2))
end
```

5.7 Procedúra position-key a record-position

V šachových hrách je bežné, že rovnaká pozícia na šachovnici môže vzniknúť viackrát. Preto existuje pravidlo, že ak sa tá istá pozícia zopakuje trikrát (alebo viackrát), môže sa vyhlásiť remíza. V našom modeli sledujeme opakovanie pozícií pomocou dvoch procedúr: **position-key** a **record-position**.

Procedúra position-key vytvára **jedinečný textový reťazec**, ktorý popisuje aktuálnu situáciu na šachovnici. Tento reťazec slúži ako "kľúč" na identifikáciu pozície v tabuľke.

Ako to funguje:

- Pre každú figúrku na šachovnici si zapíšeme jej typ (ptype), farbu (side) a súradnice) do zoznamu.
- Tento zoznam **zoradíme** (sort), aby bol poriadok vždy rovnaký bez ohľadu na to, v akom poradí sú figúrky v NetLogo.
- Na začiatok reťazca pridáme aj informáciu o tom, kto je na ťahu (turn).
- Nakoniec všetko spojíme dohromady do jedného dlhého reťazca, ktorý vraciame ako kľúč.

Procedúra record-position slúži na **zaznamenanie aktuálnej pozície** do tabuľky a sledovanie, koľkokrát sa táto pozícia už objavila.

Ako to funguje:

- Získame kľúč aktuálnej pozície volaním position-key.
- Skontrolujeme, či sa táto pozícia už nachádza v tabuľke pos-counts.
    - Ak áno, zvýšime počet jej výskytov o 1.
    - Ak nie, uložíme ju do tabuľky s počtom 1.
- Ak sa pozícia zopakuje minimálne toľkokrát, koľko je nastavené hra sa automaticky ukončí

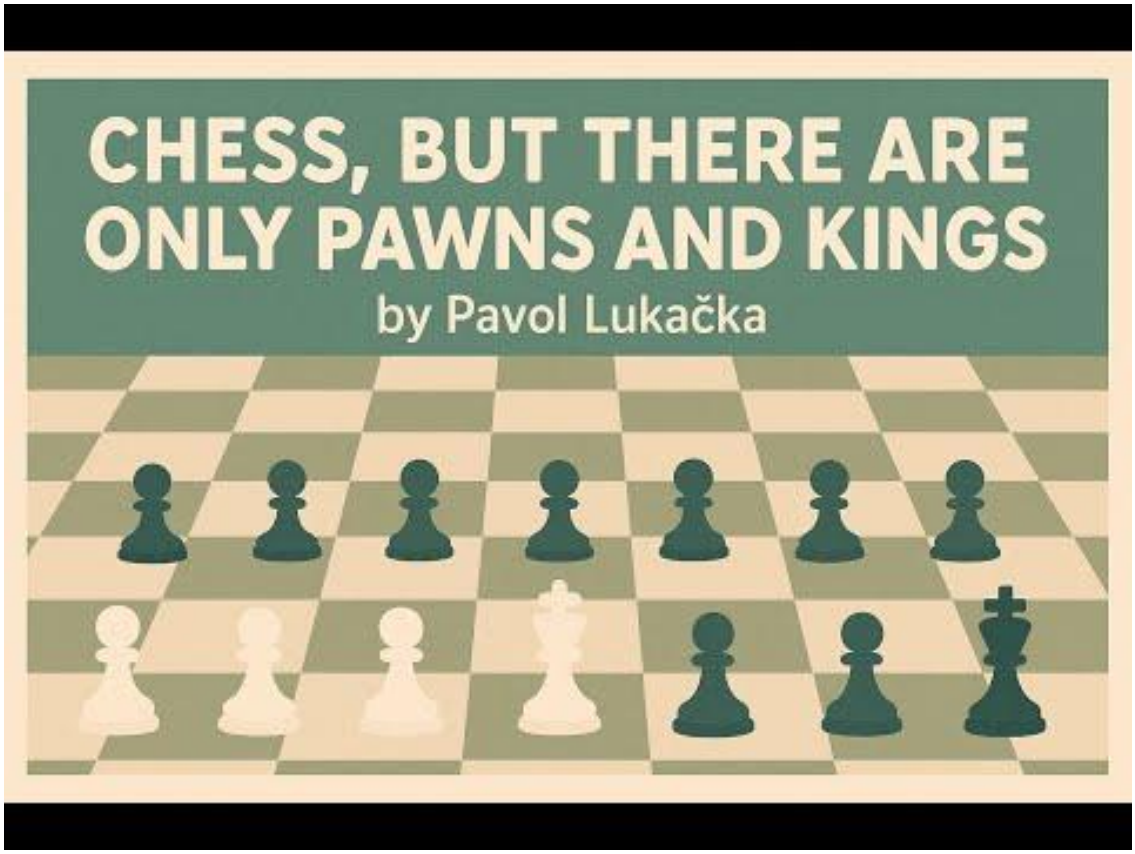
```
to-report position-key            ;; reťazec jednoznačne opisujúci pozíciu
  let lines []
  ask pieces [
    set lines lput (word ptype " " side " " pxcor " " pycor) lines
  ]
  set lines sort lines            ;; deterministický poriadok

  ;; spoj všetko do jedného kľúča
  let k turn                      ;; začni farbou na ťahu
  foreach lines [ l ->
    set k (word k "|" l)
  ]
  report k
end


to record-position                ;; aktualizuj tabuľku opakovaní
  let k position-key
  let n 1
  if table:has-key? pos-counts k [
    set n table:get pos-counts k + 1
  ]
  table:put pos-counts k n
  if n >= repetition-limit [
    user-message (word "Remíza – pozícia sa opakovala " n "×.")
    stop
  ]
end
```

6. Testovanie a experimenty

Pri testovaní sme sledovali:

- **Evaluácia pozície** (graf *Evaluation*): Zobrazuje skóre last-eval po každom ťahu (klé čas je na osi ticks). Keď biely získa výhodu (viac pešiakov či lepšia pozícia), graf stúpa do kladných hodnôt, a naopak. Po hladkom priebehu, ak biely postupne získa materiál, rastie zelená či modrá krivka nad nulu.
- **Rozdiel materiálu** (graf *Material diff*): Graf pawn-balance zobrazuje rozdiel v počte pešiakov (napr. biely minus čierny). Keď biely zoberie čierny pešiak, tento rozdiel sa zväčší o 1. Takto sme videli, že biely, ktorý začal ako maximalizátor, postupne buduje materiálnu výhodu (graf rastie).
- **Počet preskúmaných uzlov** (graf *Nodes/ply*): Sleduje nodes-searched na osiach s časom. V experimentoch vidno, že pri ťahoch, kde nastáva zložité rozhodnutie (napr. viaceré voľby ťahov), počet uzlov prudko narastie. Vplyvom alfa-beta rezania sa však počet uzlov významne znižuje (to ukazujú porovnávacie testy).
- **Vzdialenosť kráľov** (graf *King distance*): Sleduje aktuálnu Manhattanovu vzdialenosť medzi bielym a čiernym kráľom. V priebehu hry zvyčajne klesá, pretože hráči králi sa približujú k boju (a nebezpečenstvu). Keď sa vzdialenosť priblíži k 0 (králi susedia), je to indikátor konca hry.

Ukážky priebehu hry



8. Diskusia

Výsledky ukazujú, že navrhnutý algoritmus funguje podľa očakávaní. S implementovaným minimaxom (s alfa-beta) dokáže heuristicky vyhodnocovať pozície a voliť rozumné ťahy. Medzi hlavné zistenia patrí:

- **Fungujúce hľadanie:** Algoritmus úspešne identifikuje víťazné stratégie (napr. postupné získavanie pešiakov) a vedie k zachyteniu kráľa.
- **Účinnosť orezávania:** Alfa-beta významne znížila počet prehľadaných uzlov a umožňuje hlbšie hľadanie.
- **Správa opakovaných pozícií:** Aj keď v jednoduchých testoch sa vzácne vyskytovala presná repetícia (hraním na malej doske), implementovaná detekcia opakovaných stavov by zamedzila nekonečným cyklom a prípadne umožnila remízu. Tabuľka opakovaných pozícií je dobrým základom pre zložitejšie rozšírenia.

  Možné vylepšenia a rozšírenia projektu:

- **Promócia pešiakov:** Pridať možnosť, že keď pešiak dosiahne posledný rad, stane sa kráľovnou (alebo inou figúrou). To dramaticky mení strategiu, pretože výmena pešiaka za pešiaka by získala materiál so silnejšou figúrou.
- **En passant:** Implementovať špeciálny ťah en passant, ktorý si vyžaduje sledovanie predchádzajúceho ťahu a špeciálnych podmienok. Pridala by sa historická zložitosť do generovania ťahov.
- **Časové obmedzenie:** NetLogo nemá zabudovaný časový limit, ale mohli by sme simulovať náročnejšie (vyššie hĺbky) s obmedzením času, aby sme pozorovali, ako algoritmus prechádza do iteratívneho prehľadávania (poste ke kritérium).
- **Vylepšenie hodnotiacich funkcií:** Hodnotenie by mohlo zohľadniť lepšie pozíciu pešiakov (napr. blízkosť k promócii, kontrola stredu dosky) či možnú hrozbu (koordináciu kráľov). Lepšia heuristika by mohla viesť k silnejšiemu hľadaniu.
- **Rozšírenie o ďalšie figúry:** Umožniť klasický šach s kompletnou sadou figúr, prípadne ďalšia úroveň zložitosti, by znamenalo ďalší stupeň náročnosti ako projekt.

Ďalšie experimenty by tiež mohli sledovať rôzne hĺbky prehľadávania a ich vplyv na hru. Celkovo projekt ukázal princípy herného stromového vyhľadávania a poskytuje priestor na ďalší výskum v oblasti hĺbkového prehľadávania v prostredí NetLogo.

9. Záver

Predložená práca predstavila návrh, implementáciu a testovanie zjednodušeného šachového modelu „Pawns and Kings Chess“ v prostredí NetLogo. Po uvážení základných pravidiel šachu (určenie pohybov kráľa a pešiaka) a algoritmov minimax a alfa-beta sme definovali obmedzenia projektu (len králi a pešiaci, bez promócie a en passant). Ďalej sme navrhli architektúru aplikácie s globálnymi premennými, dátovými štruktúrami (tabuľky, zásobník ťahov) a procedúrami (generovanie ťahov, vyhodnocovanie, výber najlepšieho ťahu, undo). Testovanie preukázalo, že algoritmus správne vyhodnocuje pozície a dokáže rozdielom v pešiakoch pritlačiť súpera, pričom alfa-beta orezávanie zvyšuje efektívnosť oproti čistému minimaxu.