Jozef Waldhauser

Úloha 9

Minimax pre tic-tac-toe

Obsah

[Predslov autora 2](#_Toc199108254)

[Úloha 9 – zadanie 3](#_Toc199108255)

[Úloha 9 – zdroje 3](#_Toc199108256)

[Teória a analýza 4](#_Toc199108257)

[Design Algoritmu 6](#_Toc199108258)

[Podporné funkcie 6](#_Toc199108259)

[Minimax algoritmus 8](#_Toc199108260)

[Herné funkcie 10](#_Toc199108261)

[Testovacie funkcie 12](#_Toc199108262)

[Main Funkcia 14](#_Toc199108263)

[Podrobnejšia analýza 15](#_Toc199108264)

[Dátové štruktúry 15](#_Toc199108265)

[Performance algoritmu 15](#_Toc199108266)

[Záver 16](#_Toc199108267)

# Predslov autora

Všimol som si že v dokumentácií predošlej úlohy som nezahrnul zdrojový kód, tak som to tentokrát doplnil do designu algoritmu. Na úlohu by som bol rád viac času, tak sa ospravedlňujem ak nie je všetko podľa Vašich predstáv.

Prajem príjemné opravovanie -

s úctou,



# Úloha 9 – zadanie

1. Naimplementujte funkci ttt\_move(game, myplayer, otherplayer), která vezme na vstupu seznam seznamů (matici 3x3) herního plánu tic-tac-toe obsahující 0, 1 nebo 2 = (prázdné políčko, křížek nebo kolečko). Parametr myplayer určuje číslo hráče, který je na tahu - 1 (hraje křížky) nebo 2 (hraje kolečky). Parametr otherplayer je číslo druhého hráče. Funkce vrací nový herní plán rozšířený o tah hráče myplayer.
2. Můžete využít předimplementované funkce v příslušném jupyter notebooku

# Úloha 9 – zdroje

GitHub copilot – Claude Sonnet 3.7 & claude.ai Claude Sonnet 4.0 – pomoc pri kóde

Perplexity ai – časová estimácia behu algoritmu

<https://www.youtube.com/watch?v=trKjYdBASyQ>

<https://www.youtube.com/watch?v=l-hh51ncgDI>

<https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning>

<https://en.wikipedia.org/wiki/Minimax#Minimax_algorithm_with_alternate_moves>

# Teória a analýza

Hra tic-tac-toe alebo „piškvorky“ sa hrajú na matici 3x3, hrajú ju dvaja hráči.  
Cieľom hry je mať 3 za sebou idúce znaky, či už po riadku, stĺpci alebo diagonále.

Minimax algoritmus je deterministický algoritmus používaný *originálne pre* game theory pre rozhodovanie v hrách pre dvoch hráčov s nulovým súčtom   
(zero-sum games: zisk jedného hráča je stratou pre druhého hráč1 +10b 🡪 hrač2 -10b).  
Minimax ale našiel aj využitie ako výberové pravidlo v umelej inteligencii, ktoré minimalizuje straty pre najhorší prípad.

Algoritmus pracuje na princípe, že jeden hráč sa snaží maximalizovať svoje skóre „**max**uje“ zatiaľ čo druhý hráč sa snaží minimalizovať skóre prvého hráča – **mini**malizuje.

* **Deterministický:** Pri rovnakej pozícii vždy vráti rovnaký výsledok
* **Optimálny:** Garantuje najlepší možný výsledok za predpokladu optimálnej hry oboch hráčov
* **Predvída/Simuluje inteligenciu:** Algoritmus skúma jeden ťah, potom všetky možné odpovede súpera, potom svoje ďalšie možnosti, a tak ďalej až do konca hry.
* **Rekurzívny:** Viac menej celá funkcionalita minimaxu spočíva v tom, že rekurzívne volá sám seba a tak „nachádza“ nové rozvetvenia možných skončení hry.

Minimax algoritmus pracuje so „stromom“ hry, kde:

* **Uzly (nodes)** reprezentujú stavy hry
* **Hrany (edges)** reprezentujú možné ťahy
* **Listy (leaves)** reprezentujú koncové stavy hry

Tento „strom“ je vytvorený rekurziou a predstavovaním si najlepších možných ťahov (pre algoritmus najhorších). Každé volanie minimax reprezentuje jeden uzol v strome.

Môžeme to rozvinúť:

**Koreň stromu** = aktuálna pozícia, kde AI rozhoduje o svojom ťahu

**Vetvy** = všetky možné ťahy z danej pozície

**Listy** = koncové pozície hry, kde už niekto vyhral alebo je remíza

**Vnútorné uzly** striedavo reprezentujú:

* **MAX uzly** (ťahy AI) - hľadajú najvyššie skóre pre AI
* **MIN uzly** (ťahy človeka) - hľadajú najnižšie skóre pre AI

1. Rekurzia sa dostane až k listom stromu (koniec hry)
2. Listy vrátia svoje hodnoty (+10, -10, 0)
3. Vnútorné uzly vyberú MAX alebo MIN z hodnôt svojich detí
4. Táto informácia sa „dostane“ až ku koreňu
5. Koreň vyberie ťah s najlepšou hodnotou

Každá úroveň stromu buď **minimalizuje** alebo **maximalizuje** hodnoty zo svojich potomkov, podľa toho ktorý hráč je na rade.

# Design Algoritmu

Implementácia hry tic-tac-toe („piškvorky“) s AI súperom využívajúcim minimax algoritmus s alpha-beta pruning.

Implementácia umožňuje hrať proti neporaziteľnému AI alebo testovať jeho výkonnosť.

**Konštanty:**

* **BOARD\_SIZE**: Veľkosť hracej plochy (3x3)
* **EMPTY**: Reprezentácia prázdneho políčka (0)
* **PLAYER\_X**: Reprezentácia hráča ľudského (1)
* **PLAYER\_O**: Reprezentácia hráča AI (2)

## Podporné funkcie

**Function: print\_board**

* **Vstup**: board (2D array)
* **Účel**: Vizualizuje aktuálny stav hracej plochy
* **Implementácia**:
  + Mapuje číselné hodnoty na symboly (0 = ‚ ‘ 1 = X, 2 = O)
  + Formátuje a vypíše hernú plochu s označením súradníc

|  |
| --- |
| def print\_board(board):      # Vypise stav hry      symbols = {0: ' ', 1: 'X', 2: 'O'}      print("\n  0   1   2")      for i in range(3):          print(f"{i} {symbols[board[i][0]]} | {symbols[board[i][1]]} | {symbols[board[i][2]]} ")          if i < 2:              print("  -----------") |

**Function: check\_winner**

* **Vstup**: board (2D array)
* **Výstup**: 0 (nikto), 1 (X), 2 (O)
* **Účel**: Kontroluje všetky možné výherné kombinácie
* **Kontroluje**:
  + Všetky riadky (horizontálne víťazstvo)
  + Všetky stĺpce (vertikálne víťazstvo)
  + Obe diagonály (diagonálne víťazstvo)

|  |
| --- |
| def check\_winner(board):      # kontroluje, ci niekto vyhral vystup: 1 (X), 2 (O), alebo 0 (nikto)      # Kontrola riadkov      for row in board:          if row[0] == row[1] == row[2] != 0:              return row[0]        # kontrola stlpcov      for col in range(3):          if board[0][col] == board[1][col] == board[2][col] != 0:              return board[0][col]        # Kontrola diagonal      if board[0][0] == board[1][1] == board[2][2] != 0:          return board[0][0]      if board[0][2] == board[1][1] == board[2][0] != 0:          return board[0][2]        return 0 |

**Function: is\_board\_full**

* **Vstup**: board (2D array)
* **Výstup**: Boolean
* **Účel**: Kontroluje či je herná plocha úplne zaplnená (remíza)

|  |
| --- |
| def is\_board\_full(board):      # kontroluje, ci je doska plna      for row in board:          if 0 in row:              return False      return True |

**Function: get\_available\_moves**

* **Vstup**: board (2D array)
* **Výstup**: Zoznam tuples [(row, col), ...]
* **Účel**: Vráti všetky dostupné pozície pre ťah
* **Implementácia**: Iteruje cez hernú plochu a hľadá prázdne políčka

|  |
| --- |
| def get\_available\_moves(board):      # vrati zoznam dostupnych tahov ako (row, col) tuples      moves = []      for i in range(3):          for j in range(3):              if board[i][j] == 0:                  moves.append((i, j))      return moves |

**Function: is\_game\_over**

* **Vstup**: board (2D array)
* **Výstup**: Boolean/Integer
* **Účel**: Kombinovaná kontrola ukončenia hry
* **Kontroluje**: Víťaza alebo plnú plochu

|  |
| --- |
| def is\_game\_over(board):      # kontroluje, ci je hra ukoncena (vitaz alebo remiza)      has\_winner = check\_winner(board) != 0      is\_full = is\_board\_full(board)        if has\_winner:          return has\_winner      if is\_full:          return is\_full      return False |

## Minimax algoritmus

**Function: evaluate\_board**

* **Vstup**: board (2D array)
* **Výstup**: Integer skóre (-10, 0, +10)
* **Účel**: Ohodnotenie pozície z pohľadu AI
* **Hodnoty**:
  + +10: AI vyhralo
  + -10: Človek vyhral
  + 0: Remíza alebo hra pokračuje

|  |
| --- |
| def evaluate\_board(board):      winner = check\_winner(board)      if winner == PLAYER\_O:  # AI vyhralo          return 10      elif winner == PLAYER\_X:  # clovek vyhral          return -10      else:          return 0 |

**Function: minimax**

* **Vstup**: board, depth, is\_maximizing, alpha, beta
* **Výstup**: Najlepšie skóre pre pozíciu
* **Účel**: Hlavný minimax algoritmus s alpha-beta pruning
* **Implementácia**:
  + **Rekurzívne volanie**: Prehľadáva strom možných ťahov
  + **Maximalizujúci hráč** (AI): Hľadá najvyššie skóre
  + **Minimalizujúci hráč** (Človek): Hľadá najnižšie skóre
  + **Alpha-beta pruning**: Eliminuje nepotrebné vetvy stromu
  + **Depth adjustment**: Preferuje skoršie víťazstvá/prehry

|  |
| --- |
| def minimax(board, depth, is\_maximizing, alpha=float('-inf'), beta=float('inf')):      # kontrola ci niekto nevyhral      if is\_game\_over(board):          score = evaluate\_board(board)            # prisposobenie skore podla hlbky, preferujeme skorsie vyhry          if score > 0:  # AI vyhrava              adjusted\_score = score - depth              return adjusted\_score          elif score < 0:  # clovek vyhrava              adjusted\_score = score + depth              return adjusted\_score          else:  # remiza              return 0        if is\_maximizing:          # AI sa snazi maximalizovat skore          max\_eval = float('-inf')          available\_moves = get\_available\_moves(board)            for move in available\_moves:              row, col = move                # urobime tah              board[row][col] = PLAYER\_O                # rekurzivne volanie              eval\_score = minimax(board, depth + 1, False, alpha, beta)                # vratime tah spat              board[row][col] = EMPTY                # aktualizujeme najlepsie skore              max\_eval = max(max\_eval, eval\_score)              alpha = max(alpha, eval\_score)                # alpha-beta pruning (orezavanie)              if beta <= alpha:                  break            return max\_eval      else:          # clovek sa snazi minimalizovat skore          min\_eval = float('inf')          available\_moves = get\_available\_moves(board)            for move in available\_moves:              row, col = move                # urobime tah              board[row][col] = PLAYER\_X                # rekurzivne volanie              eval\_score = minimax(board, depth + 1, True, alpha, beta)                # Vratime tah spat              board[row][col] = EMPTY                # aktualizujeme najlepsie skore              min\_eval = min(min\_eval, eval\_score)              beta = min(beta, eval\_score)                # apha-beta pruning (orezavanie)              if beta <= alpha:                  break            return min\_eval |

**Function: get\_best\_move**

* **Vstup**: board (2D array)
* **Výstup**: (row, col) tuple
* **Účel**: Nájde optimálny ťah pre AI
* **Implementácia**:
  + Testuje všetky dostupné ťahy
  + Pre každý ťah volá minimax algoritmus
  + Vráti ťah s najvyšším skóre

|  |
| --- |
| def get\_best\_move(board):      best\_move = None      best\_value = float('-inf')        for row, col in get\_available\_moves(board):          # urobime tah          board[row][col] = PLAYER\_O          # vyhodnotime pomocou minimax          move\_value = minimax(board, 0, False)          # vratime tah          board[row][col] = EMPTY            # ak je tento tah lepsi, zapamatajme si ho          if move\_value > best\_value:              best\_value = move\_value              best\_move = (row, col)        return best\_move |

## Herné funkcie

**Function: make\_ai\_move**

* **Vstup**: board (2D array)
* **Výstup**: Aktualizovaný board
* **Účel**: Vykoná ťah AI na hernej ploche
* **Implementácia**:
  + Zavolá get\_best\_move pre nájdenie optimálneho ťahu
  + Umiesni 'O' na vybraté políčko
  + Vypíše informáciu o ťahu

|  |
| --- |
| def make\_ai\_move(board):      # urobi tah AI      move = get\_best\_move(board)      if move:          row, col = move          board[row][col] = PLAYER\_O          print(f"AI hra na poziciu ({row}, {col})")      return board |

**Function: make\_human\_move**

* **Vstup**: board (2D array)
* **Výstup**: Aktualizovaný board
* **Účel**: Spracuje ťah ľudského hráča
* **Validácia**:
  + Kontroluje rozsah súradníc (0-2)
  + Overuje či je políčko prázdne
  + Spracováva chybné vstupy a výnimky
* **Loop**: Opakuje až do platného ťahu

|  |
| --- |
| def make\_human\_move(board):      # spracuje tah cloveka      while True:          try:              print("\nTvoj tah (X):")              row = int(input("Zadaj riadok (0-2): "))              col = int(input("Zadaj stlpec (0-2): "))                if row < 0 or row > 2 or col < 0 or col > 2:                  print("Neplatne suradnice! Zadaj cisla 0-2.")                  continue                if board[row][col] != EMPTY:                  print("Policko je obsadene! Vyber ine.")                  continue                board[row][col] = PLAYER\_X              break            except ValueError:              print("Neplatny vstup! Zadaj cisla.")          except KeyboardInterrupt:              print("\nKoniec hry.")              exit()        return board |

**Function: play\_game**

* **Vstup**: Žiadny
* **Účel**: Hlavná herná slučka
* **Implementácia**:
  + Inicializuje prázdnu hernú plochu
  + Umožní výber kto začína
  + **Herná slučka**:
    - Zobrazí hernú plochu
    - Vykoná ťah aktuálneho hráča
    - Skontroluje víťaza alebo remízu
    - Prepne hráča
  + Ukončí hru s oznámením výsledku

## Testovacie funkcie

**Function: test\_minimax**

* **Účel**: Testuje minimax na predefinovaných pozíciách
* **Test scenáre**:
  + **Test 1**: AI má možnosť vyhrať v jednom ťahu
  + **Test 2**: AI musí blokovať víťazný ťah protivníka
* **Implementácia**: Vytvorí testovacie herné plochy a overí optimálnosť návrhov

|  |
| --- |
| def test\_minimax():      # testuje minimax algoritmus na predom definovanych poziciach      print("Testovanie minimax algoritmu...")        # Test 1: AI ma moznost vyhrat v jednom tahu      test\_board1 = [          [1, 2, 0],          [1, 2, 0],          [0, 0, 0]      ]      print("\nTest 1 - AI moze vyhrat:")      print\_board(test\_board1)      move = get\_best\_move(test\_board1)      print(f"Najlepsi tah: {move}")        # Test 2: AI musi blokovat vitazstvo protivnika      test\_board2 = [          [1, 1, 0],          [2, 0, 0],          [0, 0, 0]      ]      print("\nTest 2 - AI musi blokovat:")      print\_board(test\_board2)      move = get\_best\_move(test\_board2)      print(f"Najlepsi tah: {move}") |

**Function: benchmark\_performance**

* **Účel**: Testuje výkonnosť AI proti náhodnému hráčovi
* **Implementácia**:
  + Simuluje 100 hier AI vs náhodný hráč
  + Počíta víťazstvá AI a remízy
  + Vypočítava percentuálnu úspešnosť
  + **Očakávaný výsledok**: AI by malo vyhrať alebo remizovať všetky hry

|  |
| --- |
| def benchmark\_performance():      # testuje vykonnost minimaxu proti nahodnym vyberom      print("\nBenchmark - AI vs nahodný hráč (100 hier)...")        ai\_wins = 0      draws = 0        for game in range(100):          board = [[EMPTY for \_ in range(3)] for \_ in range(3)]          current\_player = PLAYER\_X if random.random() < 0.5 else PLAYER\_O            while True:              if current\_player == PLAYER\_O:                  # AI tah                  move = get\_best\_move(board)                  if move:                      board[move[0]][move[1]] = PLAYER\_O              else:                  # nahodny tah                  available = get\_available\_moves(board)                  if available:                      move = random.choice(available)                      board[move[0]][move[1]] = PLAYER\_X                winner = check\_winner(board)              if winner == PLAYER\_O:                  ai\_wins += 1                  break              elif winner == PLAYER\_X:                  break              elif is\_board\_full(board):                  draws += 1                  break                current\_player = PLAYER\_O if current\_player == PLAYER\_X else PLAYER\_X        print(f"Výsledky: AI vyhralo {ai\_wins}/100, remizy: {draws}")      print(f"Úspešnosť AI: {(ai\_wins + draws)/100\*100:.1f}%") |

## Main Funkcia

**Hlavné menu:**

1. **Hrať hru**: Spustí interaktívnu hru človek vs AI
2. **Testovať minimax**: Spustí testy optimálnosti algoritmu
3. **Benchmark výkonnosti**: Spustí test výkonnosti proti náhodnému hráčovi
4. **Ukončiť**: Ukončí program

|  |
| --- |
| if \_\_name\_\_ == "\_\_main\_\_":      while True:          print("\n" + "="\*50)          print("MINIMAX TIC-TAC-TOE")          print("="\*50)          print("1 - Hrať hru")          print("2 - Testovať minimax")          print("3 - Benchmark výkonnosti")          print("4 - Ukončiť")            try:              choice = int(input("\nVyber možnosť (1-4): "))                if choice == 1:                  play\_game()              elif choice == 2:                  test\_minimax()              elif choice == 3:                  benchmark\_performance()              elif choice == 4:                  print("Ďakujem za hru!")                  break              else:                  print("Neplatná voľba!") |

**Error handling:**

* Spracováva neplatné vstupy
* Zachytáva KeyboardInterrupt pre korektné ukončenie
* Validuje používateľské voľby

|  |
| --- |
| except ValueError:              print("Zadaj platné číslo!")          except KeyboardInterrupt:              print("\nKoniec programu.")              break |

# Podrobnejšia analýza

## Dátové štruktúry

**Reprezentácia hracej plochy:** 2D pole (matica) veľkosti 3x3, kde hodnoty predstavujú prázdne políčko (0), hráča X (1) a AI O (2).

**Reprezentácia ťahu:** Tuple (riadok, stĺpec) označujúci pozíciu na hr. ploche.

**Zoznam dostupných ťahov:** Dynamicky generovaný zoznam všetkých prázdnych políčok na hracej ploche.

**Funkcie validácie:** Kontrola platnosti ťahov, kontrola výhry alebo remízy, kontrola plnosti hracej plochy.

**Minimax s Alpha-Beta Pruning:** *(zdroj pre časovú komplexitu claude a perplexity ai)*

**Časová komplexita**: O() v najhoršom prípade, kde b = branching factor, d = hĺbka

**Priemerná komplexita**: O() s efektívnym pruning

**Pre tic-tac-toe**: Maximálne 9! = ~362 900 pozícií, ale pruning drasticky znižuje počet

**Časová komplexita celého algoritmu**: pri najhoršom O(9!), najlepšom O(94.5)

**Optimalizácie:**

**Alpha-beta pruning**: Eliminuje až 50% uzlov

**Depth adjustment**: Preferuje skoršie víťazstvá

**Early termination**: Ukončenie pri nájdení víťaza

**Alpha-beta pruning** výrazne znižuje počet vyhodnotených uzlov stromu. V ideálnom prípade môže znížiť časovú zložitosť až na O(), teda približne na druhú odmocninu pôvodného počtu stavov.

**Depth adjustment**: Hodnotenie víťazstva alebo prehry je upravené podľa hĺbky (depth) v strome, aby algoritmus preferoval skoršie víťazstvá a neskoršie prehry.

**Early termination**: Ak je nájdený víťaz, algoritmus ukončí ďalšie vyhodnocovanie v danej vetve.

## Performance algoritmu

Benchmark - AI vs náhodný hráč po sto hrách

|  |  |
| --- | --- |
| Beh | Výsledok |
| 1 | AI vyhralo 92/100, remizy: 8 Úspešnosť AI: 100.0% |
| 2 | AI vyhralo 94/100, remizy: 6 Úspešnosť AI: 100.0% |
| 3 | AI vyhralo 91/100, remizy: 9 Úspešnosť AI: 100.0% |
| 4 | AI vyhralo 89/100, remizy: 11 Úspešnosť AI: 100.0% |
| 5 | AI vyhralo 93/100, remizy: 7 Úspešnosť AI: 100.0% |

AI nikdy neprehráva, čo značí že vie dobre brániť, minimax algoritmus je implementovaný správne.

# Záver

Riešenie úlohy 9 mi umožnilo pochopiť teórií minimax algoritmu a pochopiť praktickej aplikácií minimax algoritmu s optimalizáciou pomocou alpha-beta pruning.   
Taktiež som zistil ako môže byť použitý v rámci AI.

Počas implementácie som sa snažil dbať nielen na funkčnosť, ale aj na čitateľnosť a modularitu kódu. Myslím že celkovo algoritmus funguje výborne, keďže sa mu (neraz) podarilo poraziť aj mňa, ak začínalo AI ako prvé.

Verím, že výsledné riešenie spĺňa očakávania zadania, a že implementácia a dokumentácia je dostačujúca. Ďakujem za možnosť vypracovať túto úlohu.