Gymnázium Dr. J. Pekaře Mladá Boleslav

Maturitní práce

METODY UMĚLÉ INTELIGENCE VE STOLNÍCH HRÁCH

Předmět: Informační a komunikační technologie

Datum a rok odevzdání: **20. 2. 2017**

Jméno a příjmení: **Michal TÖPFER**

Ročník: **8.O**

Počet stran: **32**

Počet příloh: **0**

Prohlášení

Prohlašuji, že jsem tuto maturitní práci vypracoval samostatně pod vedením Františka Tumajera. V přiložené bibliografii jsem uvedl všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

……

Poděkování

Na tomto místě bych rád poděkoval svému strýci Pavlu Töpferovi za pomoc při vymýšlení tématu této práce.

Obsah

1 Úvod

Jedním z nejrychleji rostoucích oborů dnešní informatiky je umělá inteligence. Jedná se o oblast, do které se investuje velké množství peněz, a téměř každý měsíc se internetem šíří zprávy o nových úkolech, které je umělá inteligence schopna splnit.

Zajímavým oborem, ve kterém můžeme umělou inteligenci uplatnit, je programování počítačových hráčů do deskových her, například do šachů. Cílem této práce je představit jednotlivé používané metody a ukázat, ve kterých hrách se tyto metody využívají. Zaměřím se především na takzvanou minimaxovou metodu a sám ji využiji při programování umělé inteligence pro hru piškvorky.

V první části nejprve obecně popíšu, jak minimaxová metoda funguje, a následně rozeberu její implementaci pro hru tic-tac-toe (varianta piškvorek na ploše 3x3) a poté i pro klasické piškvorky, kde využiji ještě alfa-beta prořezávání. Součástí této části práce je i naprogramování algoritmu minimax pro piškvorky.

V druhé části se budu věnovat dalším používaným metodám v různých hrách. Konkrétně se zmíním o metodách Monte Carlo tree search a o strojovém učení včetně použití umělých neuronových sítí. Také se podívám na to, jak si lidstvo stojí proti počítačům ve hrách šachy a go.

Jedním z cílů mé práce je i implementace minimaxové metody pro piškvorky v jazyce C#. Výsledný program je přiložen k práci a je ho možné také stáhnout z webové stránky https://github.com/Mnaukal/piskvorky-minimax, na které je k dispozici i kompletní zdrojový kód.

2 Základní pojmy

2.1 Umělá inteligence

Umělá inteligence je jedním z oborů informatiky. Velmi často je používán anglický název *Artificial Intelligence* (zkratka AI). Zabývá se tvorbou zařízení (a počítačových programů), které vykazují „inteligentní chování“. Inteligentní chování je obtížné přesně definovat, ale nejčastěji je jím myšleno chování podobné lidskému.

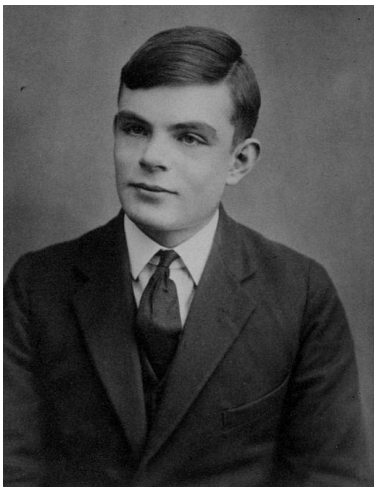
> *Umělá inteligence je věda o vytváření strojů nebo systémů, které budou při řešení určitého úkolu užívat takového postupu, který – kdyby ho dělal člověk – bychom považovali za projev jeho inteligence.*
>
> *Marvin Minsky, 1967*

Výzkum umělé inteligence je rozdělen do několika specializovaných oblastí, které lze jen těžko názorově spojit. Můžeme se zabývat řešením konkrétních technických problémů, použitím různých nástrojů, nebo se snažit vytvořit konkrétní aplikace. Důležitou součástí umělé inteligence je snaha napodobit lidské chování a myšlení.

Mezi zajímavé aplikace patří třeba rozpoznávání tvarů v obrazech, samořídící automobily, robotika, odezírání ze rtů nebo třeba protivníci v počítačových i deskových hrách. Za zmínku také stojí zpracování přirozeného jazyka (mluveného i psaného) a jeho následný automatický překlad.

2.1.1 Turingův test

Americký informatik Alan Turing (1912-1954) vymyslel v roce 1950 test, jehož cílem je prověřit, jestli se umělá inteligence chová opravdu inteligentně. Test probíhá tak, že testující člověk pokládá otázky v přirozené řeči. V oddělené místnosti na tyto otázky odpovídá buď člověk, nebo testovaná umělá inteligence. Pokud testující nedokáže rozpoznat, jestli komunikuje se strojem, nebo s člověkem, pak tato umělá inteligence uspěla v Turingově testu.



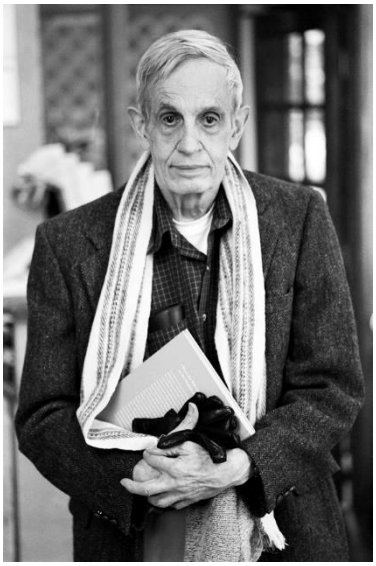
Obrázek 1: Alan Turing

2.2 Teorie her

Teorie her je jednou z disciplín aplikované matematiky. Jejím cílem je analyzovat široké spektrum konfliktních rozhodovacích situací a hledat co nejlepší strategie

pro účastníky těchto konfliktů. Má mnoho uplatnění v různých oblastech lidské činnosti – od ekonomie, přes politologii až k sociologii a biologii.

Využití je samozřejmě také při hraní her. Hra formálně obsahuje hráče a jejich možné tahy. Může se stát, že nějaký z hráčů má tzv. výherní strategii, což znamená, že ať jeho soupeř hraje jakkoli, pokaždé může vyhrát. Podobně existuje také neprohrávající strategie. Já v této práci využiji teorii her k definování her, ke kterým budu hledat strategie.



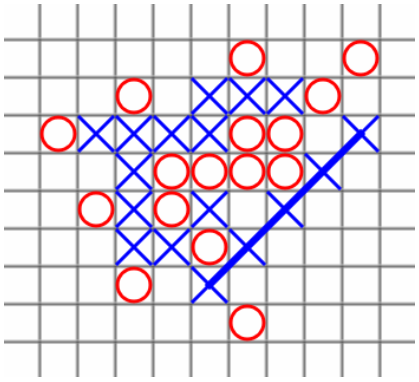
Obrázek 2: John Forbes Nash Jr.

Disciplína jako taková vznikla v roce 1944 vydáním publikace *Theory of Games and Economic Behavior* Johna von Neumanna a Oskara Morgensterna, nicméně nejdůležitější pro tento obor je pravděpodobně přínos Johna Nashe, kterému byla dokonce udělena Nobelova cena za ekonomii právě za objevy v teorii her.

Hry můžeme rozlišovat podle několika kritérií. Jedním z nich je dělení na hry s nulovým součtem a hry s nenulovým součtem. Nulový součet znamená, že vítěz vyhrává na úkor ostatních (když jeden vyhraje, druhý prohraje). V reálném světě ale většina her nemá nulový součet[1], tedy výhra jednoho nemusí nutně znamenat úplnou prohru druhého. Dále můžeme hry rozlišovat podle úplnosti informace. Ve hrách s úplnými informacemi má každý hráč k dispozici stejné informace jako všichni ostatní. Naopak hrou s neúplnými informacemi je např. poker (nevíme, jaké karty mají ostatní hráči) nebo vězňovo dilema.

2.3 Piškvorky

Nyní je důležité definovat si hru piškvorky, kterou se v této práci budu zabývat. V angličtině (mezinárodně) se často setkáme s označením *Gomoku*, které původně označuje hru s drobnými změnami v pravidlech[2]. Piškvorky se nejčastěji hrají na čtverečkovaném papíře a hráči používají symboly křížek a kolečko.



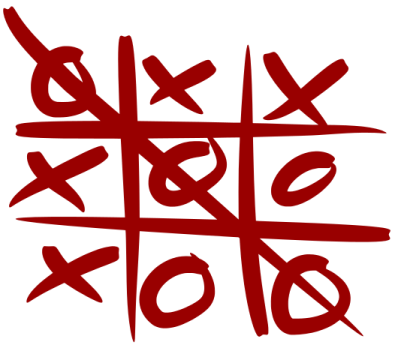
Obrázek 3: Piškvorky

[1] Příkladem hry s nenulovým součtem je *vězňovo dilema* (cs.wikipedia.org/wiki/Vězňovo_dilema)
[2] Viz cs.wikipedia.org/wiki/Piškvorky#Gomoku

Piškvorky budeme hrát na omezené čtvercové hrací ploše. Každý hráč má svou značku (křížek/kolečko). První hráč umístí svou značku na libovolné políčko hrací plochy. Dále se hráči střídají v umísťování značek na volná políčka. Hra končí ve chvíli, kdy se jednomu z hráčů podaří umístit 5 značek do jedné řady bezprostředně za sebe (vodorovně, svisle nebo diagonálně). Tento hráč se stává vítězem.

2.3.1 Tic-tac-toe

Tic-tac-toe je varianta piškvorek, která se hraje na ploše velikosti 3x3. Pravidla jsou téměř stejná, pouze pro výhru stačí umístit 3 svoje značky do řady. Výhodou této hry je její jednoduchost, takže je velmi dobře použitelná právě pro ukázky algoritmů z umělé inteligence.



Obrázek 4: Tic-tac-toe

2.4 Stavová reprezentace úlohy

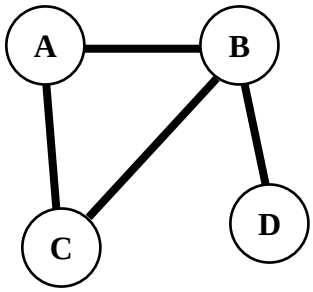
Pro zjednodušení úloh v oblasti umělé inteligence se používá tzv. *stavový prostor*. Úloha se tím převede na hledání přechodů mezi jednotlivými stavy (konkrétní rozložení křížků a koleček na hrací ploše). Nejčastěji chceme najít ten nejvýhodnější možný přechod.

2.4.1 Graf

Graf je základním objektem oblasti diskrétní matematiky zvané teorie grafů. Z formálního hlediska se jedná o uspořádanou dvojici množin V a E, kde V je množina *vrcholů* grafu a E je množina *hran*. Každá hrana je dvouprvkovou podmnožinou množiny V.

Trochu méně formálně se na graf můžeme dívat jako na množinu vrcholů (bodů, uzlů), které jsou hranami spojeny. Při analýze her většinou nechceme, aby hrana vedla z vrcholu zpět do něj (takové hrany se nazývají *smyčky*), ani aby vedlo více hran mezi stejnými vrcholy (pak by se jednalo o *multigraf*).

Vpravo je jednoduchý příklad grafu s množinou vrcholů $V = \{A, B, C, D\}$ a hranami $E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}\}$.

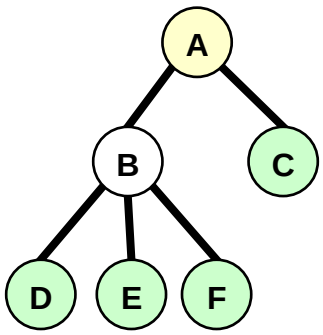


Obrázek 5: Příklad grafu

Grafy často používáme i v reálném životě, ale ani si to neuvědomujeme. Příkladem mohou být třeba plán sítě metra, dálniční síť nebo organizační struktura společnosti.

Grafy můžeme ještě dále dělit. Pokud graf neobsahuje žádné cykly (cestu přes různé hrany zpět do stejného vrcholu – v příkladu výše existuje cyklus A-B-C-A), říkáme mu *strom*. Strom má jeden kořen (ten se zpravidla kreslí nahoru) a z něj vedou hrany do jeho

následovníků (synů). Z nich potom mohou vést další hrany do jejich následovníků a tak dále. Přitom platí, že žádný vrchol není následovníkem dvou ani více jiných vrcholů zároveň. Vrcholy, které už žádné následovníky nemají, se nazývají *listy*.

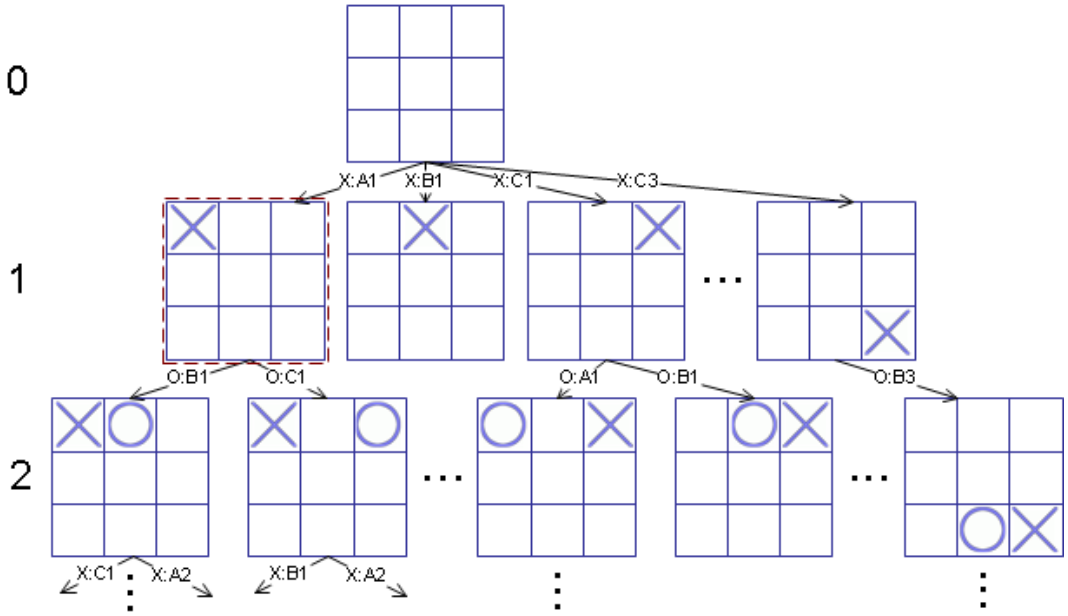


Obrázek 6: Příklad stromu s kořenem A a listy C, D, E a F

Hrany grafů mohou také být ohodnocené nějakými číselnými hodnotami (například vzdálenosti měst) nebo orientované (můžeme je projít jen jedním směrem).

2.4.2 Stavový prostor

Stavový prostor je graf, jehož vrcholy jsou stavy hry. Stavem hry u piškvorek rozumíme rozmístění křížků, koleček a volných polí na hrací ploše. Hranami stavového prostoru (tedy přechody mezi stavy) jsou možné tahy – některý z hráčů umístí svoji značku na hrací plochu. Můžeme si všimnout, že hráči nemohou odebírat své značky, takže stavový prostor piškvorek je stromem.



Obrázek 7: Ukázka grafu hry piškvorky – vrcholy jsou stavy hrací plochy, hrany jsou tahy hráčů

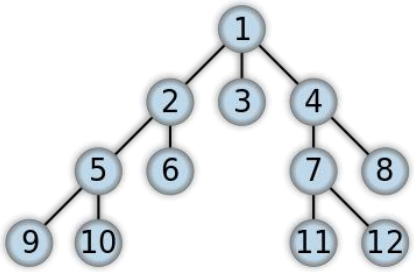
2.4.3 Prohledávání grafu

Abychom mohli vybrat správný tah, musíme nějakým způsobem projít stavový prostor a podívat se na jednotlivé možnosti. Základními typy prohledávání, které projdou celý graf, jsou *prohledávání do hloubky (DFS)*[3] a *prohledávání do šířky (BFS)*[4]. Oba v zásadě fungují podobně, a přesto se zásadně liší v pořadí procházení vrcholů. Musíme si udržovat seznam vrcholů, které chceme navštívit. V každém kroku algoritmu odebereme

[3] z anglického *Depth-first search*
[4] z anglického *Breadth-first search*

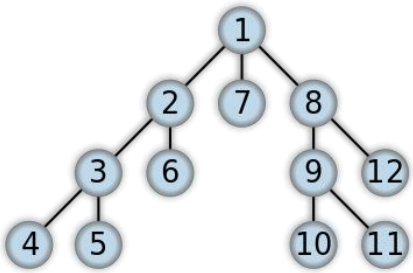
jeden vrchol z tohoto seznamu a můžeme do něj nějaké přidat. Konkrétně přidáváme všechny vrcholy, do kterých vedou hrany z právě odebraného vrcholu a které jsme ještě neprošli.

Při prohledávání do šířky tento seznam realizujeme pomocí fronty (FIFO)[5], což znamená, že vrcholy jsou z fronty odebírány ve stejném pořadí, v jakém jsou vloženy. Toto se projeví tak, že algoritmus prochází jednotlivé stavy „po vrstvách“. Z toho vyplývá, že pokud existují nějaké konečné stavy (hra v nich končí), najdeme nejprve ty z nich, které jsou umístěné nejméně hluboko (dostaneme se k nim za nejméně tahů).



Obrázek 8: Pořadí procházení vrcholů při prohledávání do šířky

Prohledávání do hloubky využívá zásobník (FILO)[6], což způsobí, že nejdříve přidaný vrchol bude odebrán jako poslední a naopak poslední přidaný prvek bude zpracován jako první. V algoritmu se to projeví tak, že nejprve projdeme celou větev grafu až k nejhlouběji umístěnému vrcholu a pak přejdeme na další větev. Hlavní výhodou prohledávání do hloubky je jeho snadné naprogramovaná pomocí rekurzivní funkce (funkce, která volá sama sebe). Druhou výhodou je, že rychleji najde nějaké řešení (ne nutně nejlepší). Při hledání nejlepšího řešení musíme projít celý strom, takže jsou oba prohledávací algoritmy srovnatelné.



Obrázek 9: Pořadí procházení vrcholů při prohledávání do hloubky

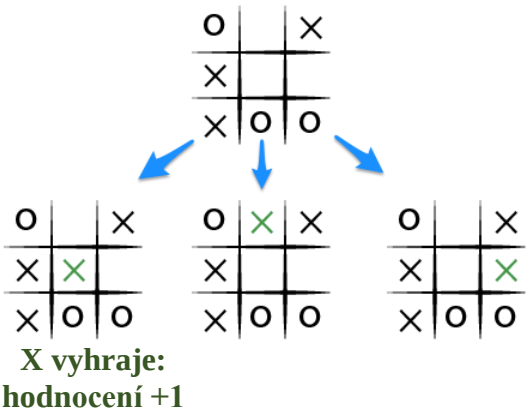
[5] *First In, First Out;* fronta se anglicky nazývá *queue*
[6] *First In, Last Out;* zásobník se anglicky nazývá *stack*

3 Minimaxová metoda

Minimax je jedním z algoritmů používaných v umělé inteligenci. Cílem tohoto algoritmu je v dané chvíli vybrat nejlepší možný tah (takový, který směřuje k výhře). Tahy můžeme v zásadě vybírat dvěma způsoby: buď náhodně (což je docela hloupé), nebo nějakým kvalifikovanějším způsobem. K tomu však potřebujeme nástroj pro hodnocení stavů hry.

3.1 Hodnocení stavů

Na hodnocení stavů se používá tzv. *hodnotící funkce*, která každý stav ohodnotí nějakým číslem. Nejjednodušší hodnotící funkcí může být to, že stav, ve kterém vyhrajeme, ohodnotíme číslem 1, prohru hodnotou -1 a remízu nulou. Piškvorky (podobně jako většina her pro dva hráče) jsou hra s nulovým součtem (zero-sum game). To prakticky znamená poměrně banální skutečnost, že žádný z hráčů nemůže udělat takový tah, který by byl výhodný pro oba. Tedy stav s hodnotou 1 pro nás bude mít hodnotu -1 pro našeho protihráče.

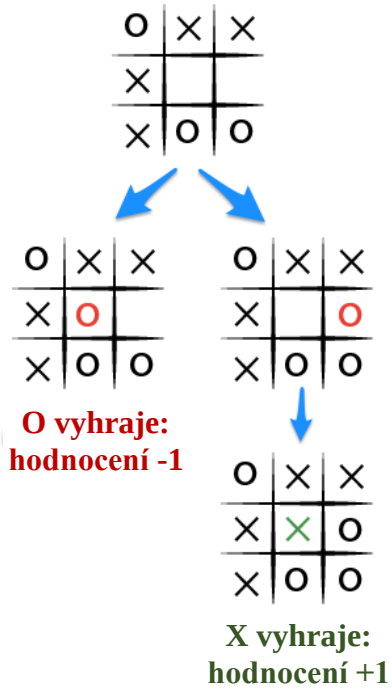


Obrázek 10: Výběr našeho tahu

Naším cílem je samozřejmě zahrát takový tah, který nám přinese nejvyšší zisk a nejvíce nás přiblíží k výhře. Na obrázku nahoře je příklad hry. Hrajeme za hráče se značkou X a jsme na tahu. Samozřejmě si vybereme levý tah, protože vede k naší výhře.

3.2 Jak bude hrát soupeř?

Počítač při hledání tahu nevidí do budoucnosti a neví, jaký další tah udělá člověk hrající proti němu. Proto předpokládáme, že se soupeř bude snažit vyhrát, a tedy bude táhnout tak, aby to pro nás bylo nevýhodné. Na příkladu vpravo vidíme, že pokud protihráč neudělá chybu, může si vybrat tah, který povede k jeho výhře, tedy našemu skóre -1.



Obrázek 11: Tah soupeře

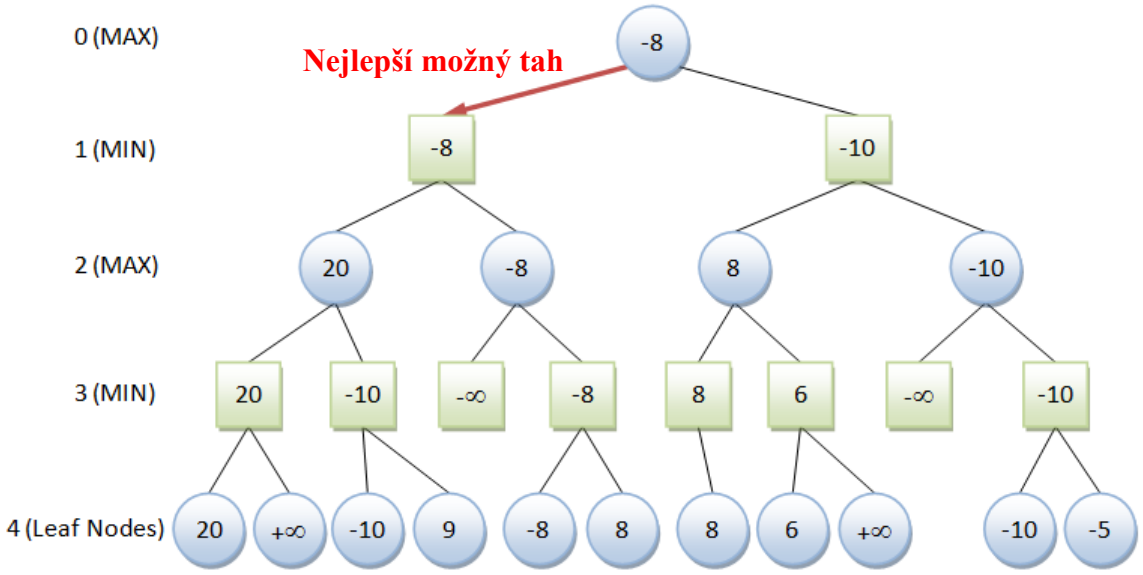
3.3 Průběh minimaxu

Minimax je prohledávání grafu do hloubky, které se snaží najít takový tah, aby *maximalizoval* skóre, které může počítač získat. Naopak protihráč pravděpodobně udělá takový tah, který přiblíží k výhře jeho, tedy *minimalizuje* skóre, které získá počítač. Od střídání minimalizace a maximalizace skóre vznikl také název algoritmu.

Předpokládejme, že hrajeme za hráče, který je zrovna na tahu. Průběh funkce minimax bude zhruba následující:

- Pokud hra skončila, vrať skóre z našeho pohledu (+1 naše výhra, 0 remíza, -1 prohra)
- Jinak vytvoř seznam všech možných tahů
- Pro každý z těchto stavů spusť minimax (rekurzivně[7]) a ulož si hodnotu, kterou ti vrátí
- Pokud jsem na tahu já, vrať MAXIMUM z uložených hodnot
- Pokud je na tahu protivník, vrať MINIMUM z uložených hodnot

Na příkladu dole můžeme ve čtvrté vrstvě vidět koncové stavy (s nějakou složitější hodnotící funkcí). Sudé vrstvy jsou naše tahy (hledáme maximum ze synů – vrcholů o vrstvu níže), liché jsou tahy protihráče (předpokládáme pro nás nejhorší variantu – minimum). Algoritmus nejprve dojde do listů grafu (*leaf nodes*) a následně vyhodnocuje hodnoty směrem nahoru.



Obrázek 12: Příklad prohledávání stromu minimaxem

[7] Rekurze znamená, že funkce volá sama sebe během svého průběhu

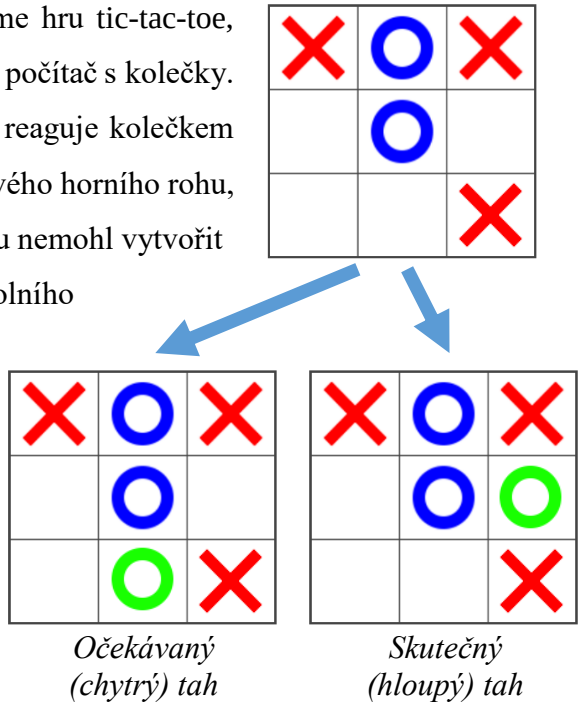
3.3.1 Pseudokód

Níže je uveden pseudokód (téměř funkční kód v C#) průběhu minimaxu. K této funkci je ještě potřeba mít definovanou funkci `Ohodnoceni()`, která je naší hodnotící funkcí, a také funkce pro hledání všech možných tahů a umístění (resp. odebrání) tahu na hrací plochu. Jejich implementace je závislá na pravidlech konkrétní hry.

```
int MiniMax()
{
  StavHry hodnoceni = Ohodnoceni();// aktuální ohodnocení hrací plochy

  if (hodnoceni.Dohrano == false) // nedohráno
  {
    List<int> hodnoceníTahů = new List<int>();
    List<Tah> tahy = new List<Tah>();

    // vyzkoušej všechny možné tahy
    foreach(Tah tah in MožnéTahy())
    {
      UmístiTah(tah); // umístí (vyzkouší) tah na hrací plochu
      hodnoceníTahů.Add(MiniMax()); // provede vnořený minimax s novým tahem
      tahy.Add(tah);                // a uloží jeho hodnocení
      OdeberTah(tah); // odebere tah z hrací plochy, aby bylo možné vyhodnotit další tahy
    }

    if (tahy.Count == 0) //neexistuje žádný možný tah
      return 0;

    //najdi maximum/minimum
    if (naTahu == NaTahu.pocitac) // jsme na tahu -> hledáme maximum
    {
      int maximum = -nekonečno;
      for (int i = 0; i < tahy.Count; i++) // projdi všechny tahy
      {
        if (hodnoceníTahů[i] > maximum) // hodnocení procházeného tahu je větší než dosavadní
        {                               // maximum
          maximum = hodnoceníTahů[i]; // je to nové maximum
          vybranyTah = tahy[i];     // a nový nejlepší (vybraný) tah
        }
      }
      return maximum; // funkce MiniMax vrátí nalezené maximum
    }
    else // na tahu je protihráč -> hledáme minimum
    {
      int minimum = nekonečno;
      for (int i = 0; i < tahy.Count; i++) // projdi všechny tahy
      {
        if (hodnoceníTahů[i] < minimum) // hodnocení procházeného tahu je menší než dosavadní
        {                               // minimum
          minimum = hodnoceníTahů[i]; // je to nové minimum
        }
      }
      return minimum; // funkce MiniMax vrátí nalezené minimum
    }
  }
  else // hra je dohraná
  {
    return hodnoceni.Ohodnoceni; // vrátí aktuální ohodnocení plochy
  }
}
```

3.4 Hloubka

Nyní už máme algoritmus, který vždy dokáže najít nejvýhodnější tah, který směřuje k výhře. Ani tento algoritmus ale není úplně dokonalý. Někdy hraje takové tahy, které se lidem zdají nesmyslné. Uvedu příklad. Hrajeme hru tic-tac-toe, začíná hráč, který má křížky, a proti němu hraje počítač s kolečky. Hráč zahraje do levého horního rohu, počítač reaguje kolečkem do středu plochy. Hráč zahraje další tah do pravého horního rohu, počítač ho „zablokuje“, aby v následujícím tahu nemohl vytvořit trojici v horní řadě. Hráč zahraje do levého dolního rohu. Aktuální situace je na obrázku vpravo.



Obrázek 13: Hloubka

V tuto chvíli by lidský hráč s kolečky určitě zahrál do spodního středového políčka a tím rovnou vyhrál hru. Ale náš program bude hrát jinak. Konkrétně do pravého středního políčka, což na první pohled působí trochu hloupě.

Když si ale situaci trochu promyslíme, zjistíme, že tento tah nebyl vůbec hloupý. Ať bude hrát protivník v tuto chvíli jakkoli, počítač ve svém následujícím tahu vždy zvítězí. Pro počítač se oba tahy zdají být stejně výhodné, protože v obou případech vždy vyhraje. To, který z nich si vybere je dáno konkrétní implementací, v mém případě vybral ten „hloupý“, protože k němu prostě došel při výpočtu dříve (prochází z levého horního rohu po řádcích).

Řešením této situace je malinko poupravit algoritmus, aby dřívější výhru (za méně tahů) považoval za lepší. Docílíme toho tím, že budeme při prohledávání zkoumat hloubku. Hloubkou je myšleno, kolikátý vnořený minimax zrovna počítáme. Budeme ji předávat jako parametr funkce `MiniMax`, který vždy při vnořeném volání zvýšíme o 1. Při vracení hodnoty pak hloubku odečítáme. Dále jsou ukázány úpravy ve zdrojovém kódu.

```
int MiniMax(int hloubka)
...
      hodnoceníTahů.Add(MiniMax(hloubka + 1)); // provede vnořený minimax s novým tahem
...
      return maximum - hloubka; // funkce MiniMax vrátí nalezené maximum
...
      return minimum + hloubka; // funkce MiniMax vrátí nalezené minimum
...
    return hodnoceni.Ohodnoceni - hloubka; // vrátí aktuální ohodnocení plochy
```

Tento algoritmus už je dostatečně dobrý na to, aby dokázal analyzovat celou hru tic-tac-toe až do konce a vybrat ten nejlepší tah. Problém nastane, když se pustíme do „větších“ her, jako jsou třeba piškvorky. Stavový prostor piškvorek (i na velmi omezené ploše 10x10) je řádově větší než stavový prostor tic-tac-toe. Není proto možné ho projít celý. K tomu opět využijeme hloubku, a to tak, že v určité hloubce už neprohledáváme další možné stavy a jen vrátíme aktuální ohodnocení. Tato úprava vyžaduje, abychom měli lepší hodnotící funkci (protože naše současná nedokáže ohodnotit stav, který není koncový), ale o tom více v další části.

3.5 Složitost

U algoritmů se běžně určuje tzv. *složitost*[8]. Jedná se o odhad závislosti doby běhu programu a potřebného rozsahu paměti na velikosti vstupu. Časová složitost je závislost času výpočtu na velikosti vstupu. Podobně paměťová složitost je závislost potřebné paměti na velikosti vstupu. Nejčastěji se udává tzv. asymptotická složitost, což v podstatě znamená, že se bere v úvahu jen nejrychleji rostoucí člen závislosti, protože složitost nás zajímá především pro velká čísla. Používá se zápis pomocí velkého O. Například složitost $O(n^2)$ znamená, že doba běhu programu je závislá na druhé mocnině velikosti vstupu (například počtu čísel na vstupu).

Časová složitost minimaxu je exponenciální, tedy $O(b^d)$, kde b je větvící faktor (tedy kolik možných tahů existuje) a d je hloubka prohledávání. Pro piškvorky je b řádově rovno velikosti hrací plochy. Paměťová složitost je relativně malá, protože nám stačí pamatovat si vždy jen aktuální procházený stav.

3.6 Prořezávání

Když zkusíme tento algoritmus spustit na piškvorky, dostaneme se do problémů. Už při hloubce prohledávání 3 trvá běh tohoto algoritmu na ploše 10x10 asi 8 sekund. To nevypadá jako příliš mnoho, ale musíme si uvědomit, že složitost tohoto algoritmu je exponenciální vzhledem k hloubce prohledávání. To znamená, že při hloubce o 1 větší poběží algoritmus v tomto konkrétním případě asi 100krát delší dobu. To už je prakticky nerealizovatelné.

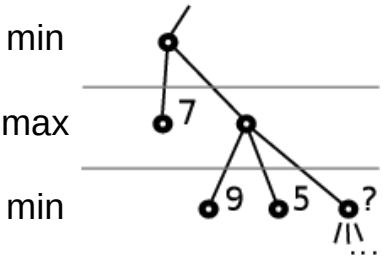
Musíme tedy náš algoritmus nějak zrychlit. Nejčastější metodou zrychlování prohledávacích algoritmů je prořezávání (někdy též ořezávání). Základní myšlenka je

[8] Více o složitosti naleznete například v kuchařce Korespondenčního semináře z programování MFF UK (http://ksp.mff.cuni.cz/kucharky/slozitost/)

taková, že některé části stavového prostoru nemusíme prohledávat, protože z nich stejně nedostaneme lepší řešení, než to, které už máme. Tyto větve algoritmus „odřízne“ a už je neprohledává. Výhodou prořezávání oproti některým jiným způsobům, jak zrychlit běh programu, je to, že jeho aplikováním neztratíme optimální řešení.

3.6.1 Alfa-beta prořezávání

Nejčastěji používaným prořezáváním v minimaxové metodě je alfa-beta prořezávání (anglicky *alfa-beta pruning*). Ukážeme si ho na příkladu. Někde při prohledávání stavového prostoru jsme narazili na situaci zobrazenou na obrázku vpravo. Potřebujeme pro optimální rozhodnutí znát hodnotu stavu s otazníkem (a kvůli tomu prohledávat celý prostor pod ním)? Nepotřebujeme. Pokud bude hodnota stavu s otazníkem větší než 5, bude minimum na této úrovni 5. V opačném případě bude minimem hodnota stavu s otazníkem. V obou případech je tedy minimum na této úrovni nejvýš 5. Hodnota 5 je ale menší než číslo 7, které máme na nadřazené maximalizační úrovni. To znamená, že i kdyby minimum na této minimalizační úrovni bylo menší než 5, stejně na nadřazené maximalizační úrovni zvolíme 7, protože je to pro nás výhodnější.

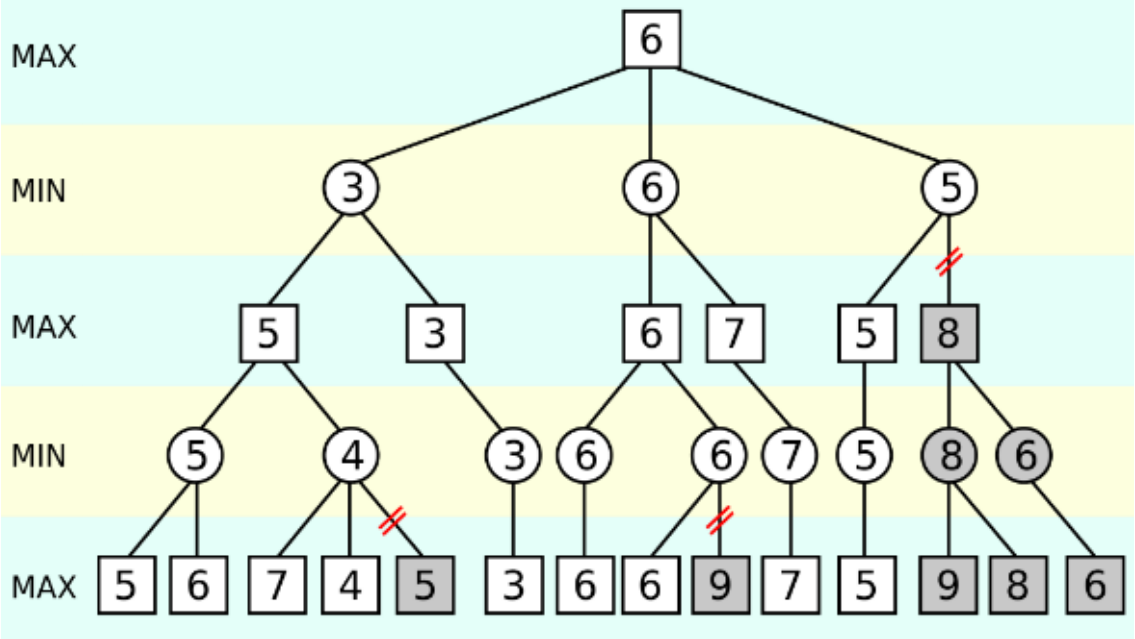


Obrázek 14: Příklad prořezávání

Do minimaxové metody si zavedeme 2 nové proměnné. Alfa představuje maximální dosažitelné (zatím nalezené) hodnocení maximalizujícího hráče (nejvyšší hodnotu). Beta bude naopak nejlepší dosažitelné skóre minimalizujícího hráče (nejmenší hodnota). Na maximalizující úrovni spočítáme alfa jako maximum z hodnocení potomků a alfa z nadřazené úrovně. Na minimalizační úrovni se alfa nemění. Podobně beta na minimalizační úrovni spočítáme jako minimum z potomků a beta na nadřazené úrovni. A na maximalizační úrovni se beta nemění.

K odříznutí části stromu může dojít, když je splněna podmínka alfa ≥ beta. Na maximalizační úrovni lze tuto podmínku interpretovat tak, že jsme právě dosáhli tahu (s hodnocením alfa), který je pro soupeře více nevýhodný (je větší než beta). Soupeř bude táhnout tak, aby se do aktuální větve stromu nedostal, a tedy nemá smysl ji dále prozkoumávat.

Je důležité poznamenat, že pokud budeme mít smůlu, alfa-beta prořezávání průběh algoritmu vůbec nezrychlí, protože nedojde k odříznutí žádné větve. Naopak největšího zrychlení dosáhneme ve chvíli, kdy nejlepší tahy vyzkoušíme jako první. V takové situaci se dostaneme do zhruba dvakrát větší hloubky prohledávání za stejný čas. Pro uspořádání tahů, aby se ty nejnadějnější prozkoumávaly jako první, se používají různé heuristiky (odhady řešení).



Obrázek 15: Příklad alfa-beta prořezávání

4 Implementace

Tato část práce popisuje mou konkrétní implementaci minimaxové metody. Je více technická a komentuje obsah jednotlivých souborů zdrojového kódu. Minimaxovou metodu jsem programoval v prostředí Microsoft Visual Studio 2015 v jazyce *C#* (za použití WPF). Kompletní zdrojový kód je k dispozici na webové stránce https://github.com/Mnaukal/piskvorky-minimax, aktuální verze spustitelného programu je ke stažení na https://github.com/Mnaukal/piskvorky-minimax/releases.

4.1 Tic-tac-toe

Pro tic-tac-toe jsem zvolil jednoduchou hodnotící funkci (`Ohodnoceni()`), která jen testuje všechny možné trojice na ploše a plochu ohodnotí 10 body, pokud počítač vyhraje (má všechna 3 políčka některé trojice), a -10 body, pokud počítač prohraje. Nevýhodou této funkce je, že dokáže ohodnotit jen dohrané hry, ale v případě tak „malé“ hry jako je tic-tac-toe to nevadí. Tento program je v souboru *Window-TicTacToe.xaml.cs*. Podobně jako v pseudokódu uvedeném výše je zde minimalizační a maximalizační fáze implementována ve stejné funkci.

V souboru *Window-TicTacToe-hloubka.xaml.cs*, jsem přidal hloubku pro označení dřívějších výher jako lepších. Soubor *Window-TicTacToe-MINMAX,hloubka.xaml.cs* obsahuje podobný kód, pouze je minimax rozdělený do dvou funkcí, jedné minimalizační a druhé maximalizační. Zkušební soubor *Window-TicTacToe-hloubka,lokalni.xaml.cs* testuje použití jiné hodnotící funkce, která počítá ohodnocení jen podle okolí posledního umístěného políčka. Taková hodnotící funkce je ale nepoužitelná pro hry, které nedokážeme dohrát do konce.

4.2 Piškvorky

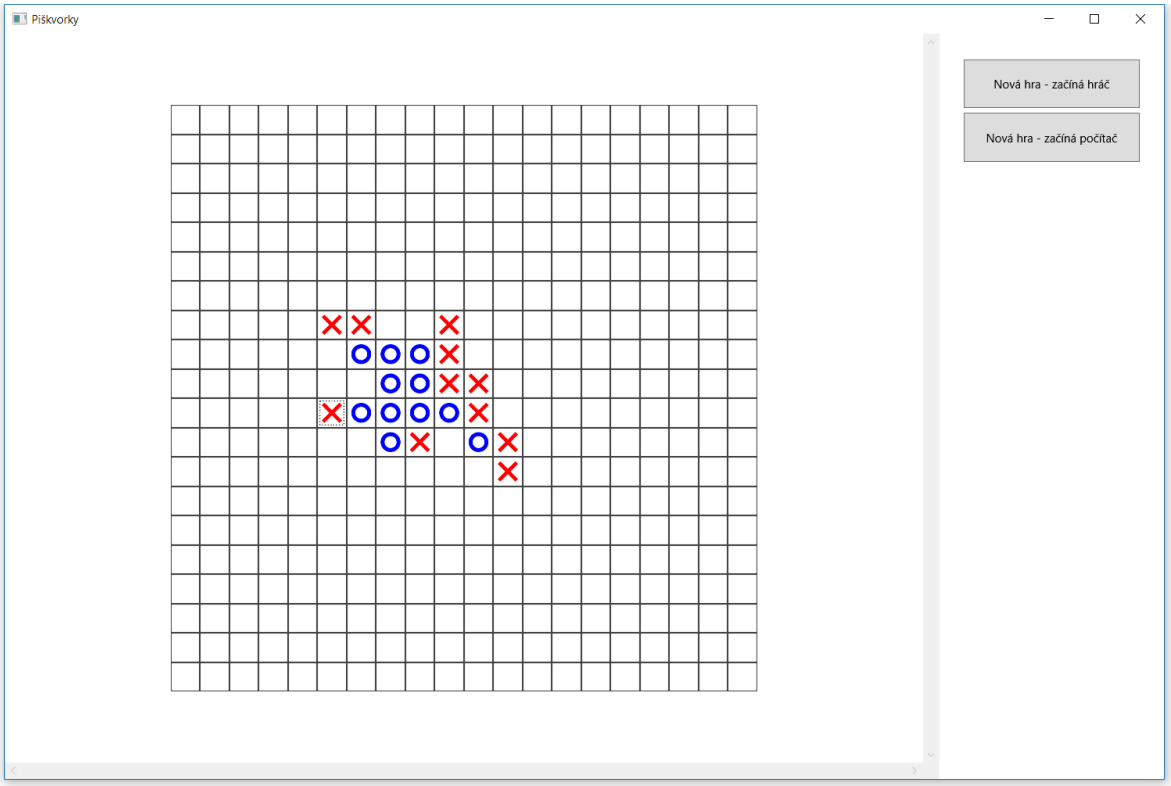
Soubory *Window-Piskvorky.xaml.cs* a *Window-Piskvorky-MINMAX.xaml.cs* implementují minimaxovou metodu pro piškvorky na větší ploše (velikost je nastavitelná). Jako hodnotící funkci používají součet hodnot jednotlivých políček. Hodnotu každého políčka počítám jako mocninu čísla 10 na exponent, který je délkou souvislé řady znaků přes toto políčko v každém směru. Pokud značky patří počítači, je hodnocení kladné, pokud patří protivníkovi, je hodnocení záporné. Nejedná se o příliš dokonalou hodnotící funkci, protože její běh trvá celkem dlouho (musíme projít všechna políčka) a také počítá body i za ukončené řetězce značek, které už nemohou vést k vítězství.

Minimaxovou metodu jsem v dalším kroku rozšířil o alfa-beta prořezávání v souboru *Window-Piskvorky-AlfaBeta.xaml.cs*. Tento soubor ještě obsahuje stejnou hodnotící funkci jako předchozí soubory.

Další vylepšení a optimalizace jsem provedl v souboru *Window-Piskvorky-AlfaBeta-optimalizace.xaml.cs*. Konkrétně jsem použil novou hodnotící funkci, která každému políčku v každém směru připočítá 5 na délku řady, pokud má jeden otevřený konec, a 7 na délku řady, pokud má otevřené oba konce. Řady délky 1 (tedy 1 samostatná značka) a řady uzavřené z obou konců jsou hodnoceny 0.

Druhou provedenou optimalizací je, že při vyhledávání množných tahů neprocházím celou hrací plochu, ale jen políčka v okolí již zahraných (konkrétně ve vzdálenosti maximálně 2 políčka od obdélníku ohraničeného nejkrajnějšími zahranými značkami). Tyto možné tahy procházím od prostředka (od tahů nejblíže těm již zahraným), protože je pravděpodobné, že budou mít lepší hodnocení a ořezávání zrychlí běh programu.

Takto optimalizovaný program už je schopen vyhledávat tahy v řádu sekund na ploše velikosti 20x20 při hloubce 3. Vygenerované tahy jsou většinou celkem dobré a počítač takto pravděpodobně dokáže porazit většinu lidí.



Obrázek 16: Uživatelské rozhraní programu

5 Další metody umělé inteligence

Kromě algoritmu minimax a jeho různých vylepšení existuje ještě mnoho dalších metod a algoritmů, které se v umělé inteligenci používají. V této kapitole se zmíním o několika zajímavých z nich.

5.1 Monte Carlo tree search

Monte Carlo tree search[9] je jedním z heuristických vyhledávacích algoritmů. Když se potřebujeme rozhodnout, jaký tah uděláme, podíváme se nejdříve, jaké máme možnosti. Pro každou z těchto možností následně náhodně nasimulujeme několik her až do konce a zjistíme, kdo častěji vyhraje. Z možných tahů pak vybereme ten, který má největší pravděpodobnost výhry.

Výhodou Monte Carlo tree search je, že nepotřebujeme žádnou přesnou hodnotící funkci. Stačí pouze naimplementovat herní mechaniky a potom simulovat hry až do konce, čímž zjistíme, kdo vyhraje. Metodu Monte Carlo je možné použít také na hry s určitým prvkem náhody.

5.2 Strojové učení

Strojové učení je podoblastí umělé inteligence, která se zabývá algoritmy se schopností učit se. Učením je zde myšlena taková změna stavu, která vede k přizpůsobení okolnímu prostředí a zvýšení úspěšnosti programu. Rozlišujeme klasifikační (když máme správně přiřadit možnost z výběru – např. dopravní značky) a regresní úlohy (odhadujeme číselnou hodnotu výstupu podle vstupu – např. hmotnost člověka podle jeho výšky a obvodu pasu). Můžeme rozlišovat učení s učitelem a bez učitele.

V učení s učitelem máme k dispozici nějakou označenou množinu trénovacích dat. Například pokud programujeme algoritmus na rozpoznávání dopravních značek, dostaneme k dispozici sadu obrázků s popisy, jakou který obrázek obsahuje značku. Snažíme se vytvořit algoritmus, který se naučí na základě těchto obrázků rozpoznávat i ostatní.

Učení bez učitele se snaží najít nějaké pravidelnosti, ale nemá zvenku zadáno, čím se takové pravidelnosti budou vyznačovat. Učením bez učitele můžeme hledat nějaké významné vlastnosti, například detekovat anomálie.

[9] Více o Monte Carlo metodě se můžete dočíst v maturitní práci Davida Novotného (8.O 2016/2017)

Kromě trénovacích dat jsou u strojového učení ještě důležité množiny dat testovacích a validačních, které slouží k určení chyby modelu. K určování přesnosti našeho algoritmu se nejčastěji používá střední kvadratická odchylka, což je průměr z druhých mocnin rozdílu našeho výsledku a správného výsledku pro každou z hodnot. Tato metoda se ale nehodí pro klasifikační úlohy, kde se používá tzv. *accuracy*, což je v podstatě procento správných odpovědí.

5.2.1 Lineární regrese – gradientová metoda

V lineární regresi se snažíme najít takzvaný lineární model, což v podstatě znamená, že výstup vytvoříme tak, že každou složku ze vstupu vynásobíme nějakým číslem. Právě tato čísla jsou hledaným lineárním modelem. Zkoušení všech možných různých kombinací čísel by trvalo velmi dlouho, a tak se využívají různé metody, jak je nezkoušet všechny a tím minimalizovat čas na učení.

Jednou z nich je gradientová metoda. Začneme v nějakém bodě. Nyní se podíváme, kterým směrem se nám chybová funkce nejvíc zmenšuje. K tomu nám pomůže tzv. *gradient,* což je vektor, který říká, kterým směrem se chybová funkce nejvíc zvětšuje. My se ale chceme posunout přesně opačným směrem než je gradient. To uděláme tak, že odečteme nějaký násobek gradientu od naší současné pozice. Tento postup budeme opakovat, dokud nebude gradient dostatečně malý (tím najdeme lokální minimum chybové funkce).

Zbývá doplnit, jak se gradient počítá. Pokud nic nevíme o chybové funkci, můžeme místo gradientu zkusit posunout každou ze složek lineárního modelu o nějakou hodnotu a vybrat, které posunutí nám nejvíce sníží chybovou funkci. U střední kvadratické odchylky se dá gradient počítat explicitně. Přesný vzorec je možno najít například v podkapitole Gradientová metoda[10] seriálu 28. ročníku Korespondenčního semináře z programování MFF UK.

5.2.2 Algoritmus K nejbližších sousedů

Algoritmus K nejbližších sousedů je založený na jednoduché myšlence. Když chceme spočítat výstup pro nějaké vstupní hodnoty, podíváme se do trénovacích dat a najdeme K vzorků, které jsou nejpodobnější tomuto vstupu. U nich známe správné výstupy,

[10] http://ksp.mff.cuni.cz/tasks/28/tasks4.html#task8

takže výstup zadaného vzorku spočítáme jako jejich průměr (u regresních úloh) nebo nejčastější kategorii (u kategorizačních úloh).

Nejpodobnější vzorky můžeme vybírat například pomocí Euklidovské vzdálenosti (druhá mocnina rozdílu každé složky vstupních vektorů[11] porovnávaných záznamů), ale je nutné data předem normalizovat (přeškálovat do rozsahu 0 až 1), aby všechny složky vstupu měly stejný význam. U kategorizačních úloh, můžeme podobnost definovat pomocí Hammingovy vzdálenosti, což je počet složek vektorů, které se liší.

5.3 Neuronové sítě

Umělé neuronové sítě jsou v poslední době velmi používaným výpočetním modelem umělé inteligence. Jsou velmi úspěšné při zpracování obrazu a zvuku a také přirozeného jazyka. Tento model navazuje na strojové učení.

Jsou inspirovány skutečnými neurony v lidském mozku. Lidské neurony mají vstupní kanály zvané dendrity a jeden výstup (axon), který se větví a rozesílá informaci do dalších neuronů. Neuron sbírá signály od svých vstupů a ve chvíli, kdy se v něm nahromadí dostatečné množství potenciálu, vyšle signál do výstupu. Spojením mezi neurony se říká synapse. Některé synapse jsou excitační (zvyšují potenciál v cílovém neuronu), jiné inhibiční (snižují potenciál – brání vyslání dalšího signálu).

5.3.1 Umělý neuron

V počítači se pro výpočet potenciálu používají čísla. Umělý neuron je jednotka, která má n číselných vstupů $x_1, \ldots x_n$ a jeden výstup y. Celkový potenciál se spočítá tak, že se každý ze vstupů vynásobí nějakým číslem tzv. *vahou* (inhibiční vstupy mají zápornou váhu) a pak se vstupy sečtou. Tento součet se potom předá jako proměnná *aktivační funkci*, čímž vznikne výstup.

Je možné použít různé aktivační funkce. Většinou jsou definované na celém R, omezené (často -1 až 1 nebo 0 až 1) a neklesající. Pokud použijeme funkci signum (vrací 1 pro kladná čísla, 0 pro 0 a -1 pro záporná), nazývá se tento umělý neuron *perceptron*.

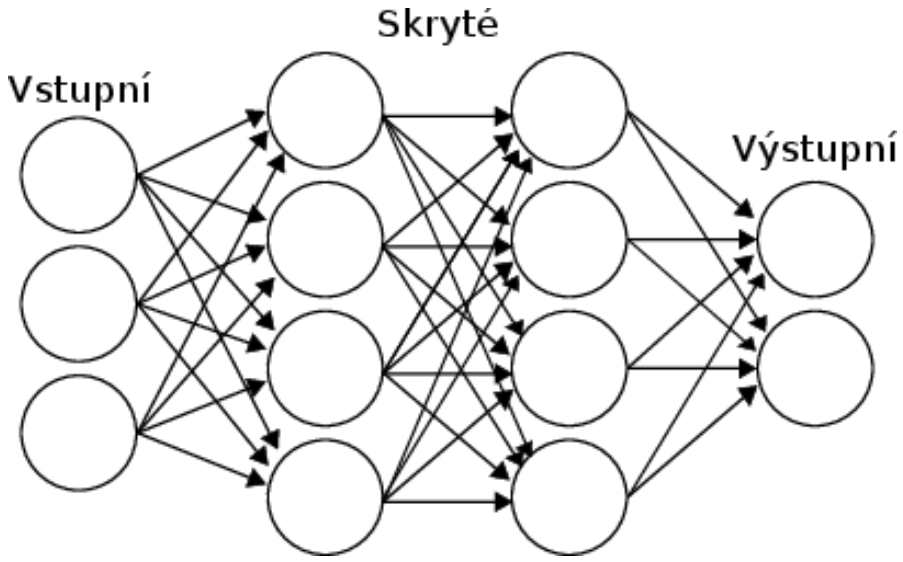
Perceptrony se často používají ke třídění do dvou kategorií. Pro učení umělého neuronu si nejprve upravíme testovací data. Všechny vstupní hodnoty u jedné z kategorií vynásobíme mínus jedničkou a tím docílíme toho, že všechny takto upravené vstupy budou

[11] vstupní vektor je vektor, jehož každá složka je jednou ze zadaných hodnot ve vstupu – u odhadu hmotnosti by se jednalo o dvojrozměrný vektor se složkami *výška* a *obvod pasu*

patřit do jedné kategorie. Cílem našeho perceptronu je pak vracet 1 pro všechna trénovací data. Postupně se podíváme na všechny testovací vstupy a pro každý si spočítáme výstup. Pokud je správný, nemusíme nic dělat. V případě špatného výstupu přičteme ke každé váze násobek příslušné složky vstupu a jdeme na další vzorek. Takto pokračujeme, dokud nenajdeme perceptron, který správně klasifikuje všechny vzorky, nebo dokud nám nedojde čas. V tomto případě je nejlepším perceptronem ten, který správně klasifikoval nejvíce vzorků za sebou.

5.3.2 Dopředné vrstevnaté neuronové sítě

Problémem umělých neuronů je, že dokáží klasifikovat jen do dvou skupin. Tento problém ale můžeme vyřešit tím, že z neuronů vytvoříme síť. Nejjednodušším způsobem, jak zapojit neurony do sítě, je rozdělit je do vrstev. Výstup každého neuronu z jedné vrstvy napojíme na vstup každého neuronu z následující vrstvy. Každý z těchto vstupů má svoji váhu.



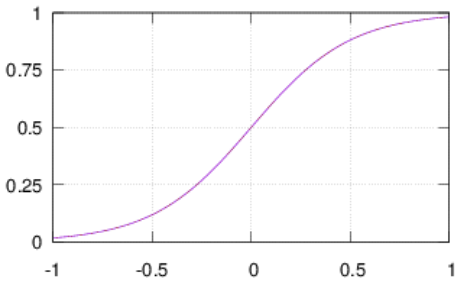
Obrázek 17: Schéma neuronové sítě

První vrstva se nazývá *vstupní*. Nepřijímá signály od jiných neuronů, ale přímo vstupní data, která bez úpravy předává dál. Podobně poslední vrstva se nazývá *výstupní*, protože z ní čteme výstup sítě. Ostatní vrstvy jsou pro okolní svět *skryté*.

Naučení sítě znamená nalezení takových vah, aby síť prováděla zadaný úkol. Tedy aby byl pro vstupy z trénovacích dat výstup sítě co nejpodobnější požadovanému výstupu. Opět musíme nějakým způsobem měřit chybu sítě. Většinou se odchylka každého neuronu umocní na druhou, aby se zdůraznily větší chyby a zanedbaly ty menší.

Je vhodné, aby aktivační funkce neuronů byla spojitá a rostoucí. Místo funkce signum se tedy používá takzvaná *sigmoida*, která je definována následujícím vzorcem:

$$f(p) = \frac{1}{1 + e^{-\lambda * p}}$$



Obrázek 18: Sigmoida pro λ= 4

λ je v této funkci konstanta, jejímž nastavením se mění strmost funkce. Písmenkem *p* značíme potenciál daného neuronu, *e* je Eulerovo číslo.

Algoritmus pro výpočet výstupu neuronové sítě je celkem přímočarý. Postupně spočítáme výstupy jednotlivých neuronů v každé vrstvě a tyto výstupy potom použijeme jako vstupy neuronů další vrstvy.

5.3.3 Zpětná propagace

Nejzákladnější metodou, která se používá k učení sítě, je zpětná propagace. Nejdříve náhodně nastavíme všechny váhy sítě. Při učení si postupně náhodně vybíráme vstupy z trénovacích dat, necháme síť spočítat výsledek a ten pak porovnáme se správným výsledkem. Potom upravíme váhy sítě, aby se její výsledek přiblížil požadovanému.

Tyto úpravy probíhají postupně od výstupní vrstvy, přes všechny skryté, až ke vstupní vrstvě (proto *zpětná* propagace). Změna každé váhy závisí na chybě neuronů, mezi kterými vede, a jejich výstupech. Podrobnější informace a vzorce pro úpravu vah se opět můžete dočíst v seriálu 28. ročníku KSPčka[12].

[12] http://ksp.mff.cuni.cz/tasks/28/tasks5.html#task8

6 Použití umělé inteligence v dalších hrách

Poslední kapitola této práce se týká použití metod umělé inteligence v dalších stolních hrách. Konkrétně se bude jednat o šachy a staročínskou hru go.

6.1 Umělá inteligence pro šachy

V dnešní době existuje mnoho šachových programů spustitelných na běžných počítačích nebo chytrých telefonech, které dokážou člověka porazit. Mezi některé z nich patří třeba šachové programy *Stockfish*, *Crafty*, *Fruit* a *GNU Chess*, které si každý může zdarma stáhnout z internetu. Ty nejlepší z nich, jako *Shredder* nebo *Fritz*, jsou dokonce schopné porazit šachové velmistry.

Historie specializovaných šachových programů i počítačů se začala psát někdy v polovině 70. let 20. století. Od té doby se šachové programy postupně zlepšují až do současnosti. Jedním z nejdůležitějších milníků tohoto odvětví a umělé inteligence vůbec byla první porážka šachového velmistra počítačem.

6.1.1 Superpočítač IBM Deep Blue

V roce 1985 by sestrojen a naprogramován počítač *ChipTest*, který měl speciální šachový čip. Byl schopen analyzovat asi 50 000 tahů za sekundu. V dalších generacích se podařilo počítač, který byl přejmenován na *Deep Thought*, vylepšit, aby zvládal přibližně 750 000 tahů za sekundu. Následně se jeho vývojový tým spojil s IBM a vytvořili počítač *Deep Blue*.

První utkání se šachovým velmistrem Garry Kasparovem se odehrálo v roce 1996. Počítač vyhrál první partii a tím se zapsal do dějin jako první program, který porazil šachového velmistra. Celý zápas nakonec skončil v poměru 4:2 pro Kasparova. Další vylepšení počítač Deep Blue posunula dokonce mezi superpočítače. Dosahoval výkonu 11,38 GFLOPS, což už ale dnes zvládne většina mobilních telefonů.

Obrázek 20: Deep Blue

Obrázek 19: Garry Kasparov

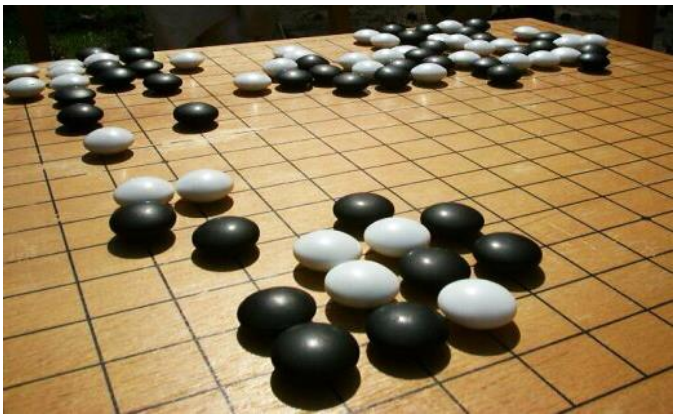
Další souboj se konal v květnu 1997. V šesti partiích se počítači *Deep Blue* podařilo vyhrát v poměru 3½:2½ (remíza se počítá za půl bodu pro každého). Inženýři mezi jednotlivými hrami stále vylepšovali algoritmy, takže Garry Kasparov se později vymlouval, že hra byla příliš podobná hře živého šachisty a měl tedy podezření, že tahy byly během hry ovlivňovány rozhodováním člověka. To se však nikdy nepotvrdilo.

6.1.2 Metody používané v šachách

Většina šachových programů je založena na algoritmu minimax. Tento algoritmus je ale příliš pomalý, takže se používají různá vylepšení. Nejzákladnějším z nich je alfa-beta ořezávání, které bylo popsáno dříve v této práci. Další používaná vylepšení jsou například NegaScout, MTD(f) nebo použití nějaké heuristiky.

6.2 Umělá inteligence pro go

Další velmi zajímavou hrou, která dlouho odolávala metodám umělé inteligence, je *go*. Je to desková hra, která původně pochází z Číny (tam se nazývá *Wej-čchi*). Je velmi populární v celé východní Asii. Hraje se obvykle na desce 19×19 průsečíků, které se říká *goban*. Cílem je mít na konci hry větší skóre, které je tvořeno zajatými kameny a obklíčenými průsečíky. Podrobnější pravidla najdete například na Wikipedii[13].



Obrázek 21: Rozehraná partie go

Go je velmi komplexní hra, která vyžaduje strategické a kreativní myšlení a také značné množství intuice. Z tohoto důvodu je pro počítače velmi těžká. Konkrétními problémy, se kterými se počítačové programy pro go musí vypořádat, jsou poměrně velká hrací plocha (361 průsečíků) a také velké množství možných tahů (téměř vždy je možné udělat tah na jakékoli políčko). Dalším problémem je, že je velice obtížné vytvořit hodnotící funkci, která by správně ohodnocovala všechny stavy hry.

Až do roku 2015 hrála většina počítačových programů go hůře, než amatérský hráč. Profesionální hráči dokonce porážely počítače i s hendikepem 25 kamenů. Nedávný rozvoj Monte Carlo tree search a strojového učení umožnil i vylepšování počítačových hráčů go.

[13] https://cs.wikipedia.org/wiki/Pravidla_hry_go

Podařilo se dosáhnout dobrých výsledků pro malé hrací plochy (9x9) a později i pro větší. V roce 2010 vyhrál program *MogoTW* na hrací ploše 19x19 proti Cătălin Țăranovi (profesionální hráč páté úrovně – nejvyšší je devátá) s handicapem 7 kamenů. Dalšímu programu, který se jmenuje Zen, se podařilo vyhrát v roce 2012 nad Masaki Takemiyou (profesionál úrovně 9) s handicapem 4 kamenů.

6.2.1 AlphaGo

Veliký obrat ve hře go přinesl Google. Vytvořil program *AlphaGo*, který je založený na jeho systému neuronových sítí vyvíjených v *Google DeepMind*. Tomuto programu se podařilo v říjnu 2015 porazit evropského šampióna v go Fan Huiho v pěti z pěti zápasů. To byl první případ, kdy počítač porazil profesionálního hráče bez handicapu.

Obrázek 22: Logo AlphaGo

Druhým a možná ještě důležitějším zápasem byl souboj AlphaGo se světovým mistrem Lee Se-dolem (profesionál 9. úrovně). Tato událost, která proběhla v březnu 2016, byla velmi medializovaná a spousta zpravodajských serverů psala o každém ze zápasů. První tři zápasy vyhrála umělá inteligence, ve čtvrtém uspěl člověk, ale AlphaGo potvrdila pátým zápasem svou výhru 4:1. Podle velmistra byla hra proti stroji nepříjemná v tom, že nemohl sledovat fyzické reakce svého protihráče, které by prozradily nervozitu. Právě podle toho se hráči go často řídí.

Celý systém AlphaGo je velmi komplikovaný a snaží napodobit hru člověka. Používá metodu Monte Carlo tree search, kterou vybírá nejlepší tahy. K tomu využívá hlubokou neuronovou síť, která se učí zkoumáním obrovského množství partií mezi lidmi i počítači. V nich se snaží objevit strategie, které vedou k výhře.

7 Závěr

V této práci jsem popsal různé metody umělé inteligence používané při programování počítačových hráčů do deskových her. Podrobně jsem rozebral minimaxovou metodu s využitím alfa-beta prořezávání, která se dá použít nejen pro piškvorky, ale také pro šachy. Dále jsem zmínil metody Monte Carlo tree search a strojové učení včetně neuronových sítí, díky kterým byl nedávno poražen světový velmistr ve hře go.

Naprogramoval jsem minimaxovou metodu ve hrách tic-tac-toe a piškvorky. Výsledný program pro tic-tac-toe je neporazitelný, hra vždy skončí remízou nebo prohrou člověka. Pro piškvorky jsem byl více limitován výkonem počítačů a velkým množstvím dat, která je potřeba prohledat pro kvalitní rozhodnutí. Vytvořený program nicméně hraje piškvorky na velmi vysoké úrovni a není vůbec jednoduché ho porazit. Je možné ho stáhnout ze stránky https://github.com/Mnaukal/piskvorky-minimax/releases.

Umělá inteligence je velmi důležitým oborem současné informatiky. Jejím cílem není jen programování počítačových hráčů do her, ale i řada praktických aplikací. Výhodou her jsou jejich jasná pravidla, takže se na nich algoritmy dobře testují. Reálné využití algoritmů umělé inteligence je potom v mnoha oblastech, jako je například analýza dat uživatele na internetu kvůli poskytování cílené reklamy, automatické překlady psané i mluvené řeči, nebo třeba rozpoznávání předmětů na obrázcích a samořídící automobily.

8 Bibliografie

1. **Barták, Roman.** Hry (minimax, alfa-beta prohledávání). *Umělá inteligence I.*

2. **Setnička, Jiří a kolektiv.** *Korespondenční seminář z programování XXVIII. ročník.* Praha : MatfyzPress, 2016. ISBN 978-80-7378-330-3.

3. **Popelka, Ondřej.** Piškvorky. *Umělá inteligence.* [Online] [Citace: 28. 12. 2016.] https://akela.mendelu.cz/~xpopelka/cs/ui/piskvorky/.

4. —. Stavová reprezentace úlohy. *Umělá inteligence.* [Online] [Citace: 28. 12. 2016.] https://akela.mendelu.cz/~xpopelka/cs/ui/stavy/.

5. —. Grafy a prohledávání grafu. *Umělá inteligence.* [Online] [Citace: 28. 12. 2016.] https://akela.mendelu.cz/~xpopelka/cs/ui/graf/.

6. —. Minimaxová metoda. *Umělá inteligence.* [Online] [Citace: 28. 12. 2016.] https://akela.mendelu.cz/~xpopelka/cs/ui/minmax/.

7. —. Prořezávání. *Umělá inteligence.* [Online] [Citace: 28. 12. 2016.] https://akela.mendelu.cz/~xpopelka/cs/ui/prorezavani/.

8. **Töpfer, Pavel.** *Algoritmy a programovací techniky.* Praha : PROMETHEUS, 1995. 80-85849-83-6.

9. **Javůrek, Karel.** Deep Blue a 15 let od porážky nejlepšího člověka v šachu. *VMT.cz.* [Online] [Citace: 18. 12. 2016.] http://vtm.e15.cz/deep-blue-a-15-let-od-porazky-nejlepsiho-cloveka-v-sachu.

10. **Zuna, Pavel.** Den, kdy počítač porazil mistra světa v šachu (10. únor 1996). *Stream.* [Online] Seznam.cz. [Citace: 18. 12. 2016.] https://www.stream.cz/slavnedny/10004926-den-kdy-pocitac-porazil-mistra-sveta-v-sachu-10-unor.

11. **Čížek, Jakub.** Před 64 lety začal počítač soupeřit s člověkem. Nyní jej opět pokořil jako kdysi Kasparova. *Živě.cz.* [Online] [Citace: 18. 12. 2016.] http://www.zive.cz/clanky/pred-64-lety-zacal-pocitac-souperit-s-clovekem-nyni-jej-opet-pokoril-jako-kdysi-kasparova/sc-3-a-181218/default.aspx.

12. **Schön, Otakar.** Umělá inteligence AlphaGo od Googlu porazila v prvním zápase velmistra hry Go. *IHNED.cz.* [Online] Hospodářské noviny. [Citace: 18. 12. 2016.] http://tech.ihned.cz/c1-65199220-umela-inteligence-alphago-od-googlu-porazila-v-prvnim-zapase-velmistra-hry-go.

13. **Přispěvatelé Wikipedie.** Umělá inteligence. *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 10. 12. 2016.] https://cs.wikipedia.org/wiki/Um%C4%9Bl%C3%A1_inteligence.

14. —. Turingův test. *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 10. 12. 2016.] https://cs.wikipedia.org/wiki/Turing%C5%AFv_test.

15. —. Teorie her. *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 10. 12. 2016.] https://cs.wikipedia.org/wiki/Teorie_her.

16. —. Piškvorky. *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 10. 12. 2016.] https://cs.wikipedia.org/wiki/Piškvorky.

17. —. Monte Carlo tree search. *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 10. 12, 2016.] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.

18. —. Heuristika. *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 30. 12. 2016.] https://cs.wikipedia.org/wiki/Heuristika.

19. —. Go (hra). *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 18. 12. 2016.] https://cs.wikipedia.org/wiki/Go_(hra).

20. —. AlphaGo. *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 18. 12. 2016.] https://en.wikipedia.org/wiki/AlphaGo.

21. —. Alfa-beta ořezávání. *Wikipedie: Otevřená encyklopedie.* [Online] [Citace: 30. 12. 2016.] https://cs.wikipedia.org/wiki/Alfa-beta_o%C5%99ez%C3%A1v%C3%A1n%C3%AD.

22. **Wikipedia contributors.** Breadth-first search. *Wikipedia, The Free Encyclopedia.* [Online] [Citace: 28. 12. 2016.] https://en.wikipedia.org/wiki/Breadth-first_search.

23. —. Computer Go. *Wikipedia, The Free Encyclopedia.* [Online] [Citace: 18. 12. 2016.] https://en.wikipedia.org/wiki/Computer_Go.

24. —. Computer chess. *Wikipedia, The Free Encyclopedia.* [Online] [Citace: 18. 12. 2016.] https://en.wikipedia.org/wiki/Computer_chess.

25. —. Depth-first search. *Wikipedia, The Free Encyclopedia.* [Online] [Citace: 28. 12. 2016.] https://en.wikipedia.org/wiki/Depth-first_search.

26. **Laramée, François Dominic.** Chess Programming Part I: Getting Started. *GameDev.net.* [Online] [Citace: 18. 12. 2016.] http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-i-getting-started-r1014.

27. **Fox, Jason.** Tic Tac Toe: Understanding The Minimax Algorithm. *Never Stop Building.* [Online] 13. 12. 2013. [Citace: 29. 12. 2016.] http://neverstopbuilding.com/minimax.

28. **Wikipedia contributors.** Tic-tac-toe. *Wikipedia, The Free Encyclopedia.* [Online] [Citace: 10. 12. 2016.] https://en.wikipedia.org/wiki/Tic-tac-toe.

29. **Hock-Chuan, Chua.** Tic-tac-toe AI. *Java Game Programming Case Study.* [Online] [Citace: 28. 12, 2016.] https://www.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html.

30. **Böhm, Martin a kolektiv.** *Korespondenční seminář z programování XXIV. ročník.* Praha : MatfyzPress, 2012. ISBN 978-80-7378-227-6.

9 Seznam obrázků: