Vysoké učení technické v Brně
Fakulta informačních technologií

SUI — Umělá inteligence a strojové učení

Umělá inteligence pro Válku kostek

Josef Kolář (xkolar71)
Dominik Harmim (xharmi00)
Petr Kapoun (xkapou04)
Jindřich Šesták (xsesta05)

29. listopadu 2020

Obsah

1 Úvod

Cílem projektu bylo vytvořit *umělou inteligenci (AI)* pro hru **Válka kostek**. Umělá inteligence musí *prohledávat stavový prostor* hry a musí využívat *strojové učení*. Projekt je postaven na bakalářské práci [3][1].

Dokumentace je dále strukturována následovně. Kapitola 2 pojednává o základní implementaci umělé inteligence založené na prohledávání stavového prostoru hry. Kapitola 3 popisuje rozšíření implementace o prvky strojového učení. Vyhodnocení výsledné umělé inteligence a srovnání použitých přístupů je popsáno v kapitole 4. Kapitola 5 vysvětluje obsah odevzdávaného archivu. Konečně kapitola 6 shrnuje výsledky tohoto projektu.

2 Implementace umělé inteligence prohledáváním stavového prostoru

Po analýze prostředí a implementace hry bylo zjištěno, že prostředí je *plně pozorovatelné* a *stochastické*. Na základě těchto vlastností bylo nejdříve uvažováno o implementaci umělé inteligence jako *agenta* (agent reprezentuje jednoho hráče), který bude implementovat *prohledávání stavového prostoru* algoritmem $ExpectiMiniMax$ popsaným v původní bakalářské práci [3]. Tento algoritmus vychází z algoritmu $MiniMax$. Je však rozšířen o *nedeterministické* akce, proto je vhodný pro toto stochastické prostředí.

Algoritmus $MiniMax$ i $ExpectiMiniMax$ je však ve své standardní variantě definován pouze pro hru dvou hráčů. V tomto projektu je nutné uvažovat hru čtyřech (a obecně n) hráčů. Existuje zobecnění těchto algoritmů pro hru n hráčů. Jedná se o tzv. Max^n algoritmus, který lze opět rozšířit o nedeterminismus. Dále bude hovořeno pouze o Max^n algoritmu, ale ve výsledné implementaci tohoto projektu byl patřičně rozšířen i o nedeterministické akce. Algoritmus Max^n byl poprvé představen v článku *An Algorithmic Solution of N-Person Games* [1], kde je podrobně popsán spolu s problematikou her pro více hráčů. Článek *Comparison of Algorithms for Multi-player Games* [2] pak srovnává algoritmus Max^n s alternativními algoritmy pro hry více hráčů.

2.1 Algoritmus Max^n

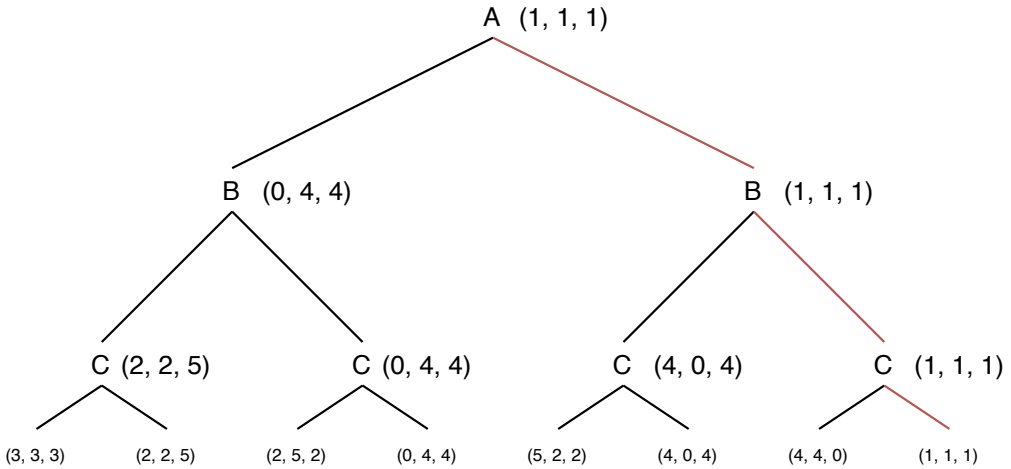
Algoritmus Max^n je přesně definován ve výše uvedených článcích [1, 2], proto bude v této kapitole už jen neformálně popsán a znázorněn.

V klasickém algoritmu $MiniMax$ se střídají tahy dvou hráčů. Hráč na tahu – MAX – se snaží vybírat tahy vedoucí do stavů s maximálním ohodnocením, tj. snaží se maximalizovat svoji *objektivní funkci*. Protihráč – MIN – se naopak snaží minimalizovat objektivní funkci hráče MAX. Tyto dvě hodnoty lze reprezentovat jednou hodnotu vytvořením jejich sumy. Je také možné dívat se na tento problém tak, že každý z hráčů se snažím maximalizovat svoji vlastní objektivní funkci. Potom by se dalo při prohledávání stavového prostoru tohle reprezentovat jako dvojice maximálních hodnot objektivní funkce pro každého z hráčů.

Algoritmus Max^n potom funguje stejně jako $MiniMax$, kde se ale při prohledávání uvažuje n-tice, kde n je počet hráčů. Každý prvek této n-tice reprezentuje maximalizovanou hodnotu objektivní funkce daného hráče a každý z hráčů se snaží svoji objektivní funkci při svém tahu maximalizovat. Algoritmus je ilustrován na obrázku 1. Uvažuje se hra pro 3 hráče — A, B, C. Tito hráči se v uvedeném pořadí střídají při hraní svých tahů. Každý z hráčů může provést dva možné tahy. Po třech tazích hra skončí (obecně by po tahu hráče C hrál opět hráč A, ale zde jsou tahy hráče C uvažovány jako koncové, po nichž hra končí). Stav hry je tady popsán trojicí – $(obj(\mathrm{A}), obj(\mathrm{B}), obj(\mathrm{C}))$ – kde $obj(\mathrm{X})$ je hodnota objektivní funkce hráče X. Na obrázku je vidět expandovaný graf hry pro dané hráče. Listové uzly jsou ohodnoceny objektivní funkcí a nelistové uzly jsou ohodnoceny podle maximalizace objektivní funkce

[1] Projekt je konkrétně založen na upravené verzi z původní bakalářské práce — https://github.com/ibenes/dicewars.

aktuálního hráče na tahu. Je zřejmé, že hráč A učiní tah „vpravo“, protože stav, do kterého tento tah vede, je ohodnocen pro něho nejlepší objektivní funkcí — $(1, 1, 1)$. Cesta do koncového stavu s tímto ohodnocením je vyznačena červenými hranami.



Obrázek 1: Příklad *prohledávání stavového prostoru* hry pro 3 hráče algoritmem Max^n

2.2 Konkrétní implementace a použití algoritmu Max^n

Tato kapitola popisuje výslednou implementaci umělé inteligence prohledáváním stavového prostoru algoritmem Max^n pro tento projekt. Dále popisuje rozšíření tohoto algoritmu o nedeterministické akce a konkrétní použití v projektu.

Implementace prohledávání stavového prostoru se nachází ve třídě `dicewars.ai.xkolar71.AI` (kromě implementace heuristické funkce pro ohodnocování stavů hry, která je implementována ve třídě `dicewars.ai.xkolar72_orig.AI`, vzhledem k tomu, že heuristická funkce byla nakonec nahrazena strojově učeným modelem, viz kapitola 3).

Definice 2.1 ($maxN$)**.** Byla definována *rekursivní* funkce $maxN$, která implementuje algoritmus Max^n pro účely tohoto projektu. Funkce v každém tahu daného hráče zkoumá následující tahy všech ostatních hráčů v patřičném pořadí a vrací nejlepší spočítaný tah. Funkce konkrétně uvažuje n iterací prohledávání tahů všech následujících hráčů. Parametr n byl experimentálně zvolen na hodnotu 1, tj. uvažuje se nanejvýše jedenkrát sekvence tahů každého z následujících hráčů. Při vyšších hodnotách tohoto parametru už byly naměřeny horší výsledky, což je způsobené tím, že prostředí hry je *silně nedeterministické* a jen obtížně lze přesněji predikovat výsledky akcí dále do budoucnosti. Při zkoumání tahů hráče následujícího v pořadí se generuje m útoků. Následně z každého útoku se generuje $m-1$ útoků, z každého z nich $m-2$ útoků atd. až do hloubky l. Parametry m a l byly experimentálně nastaveny na hodnotu 5, opět s ohledem na stochastické prostředí. Útoky, které se generují v daném tahu, jsou vybrány jako ty možné útoky s *největší pravděpodobností* na výhru funkcí $possibleTurns$, viz definice 2.2. Tímto byl algoritmus Max^n rozšířen o nedeterminismus. Uvažované útoky jsou vždy simulovány na kopii aktuální konfigurace hry. Funkce $maxN$ potom v listových uzel (definovaných parametry n a l) ohodnotí daný stav *heuristickou funkcí* h, viz definice 2.3. Konečně $maxN$ maximalizuje hodnotu funkce h pro každého z hráčů a vrací tah do uzlu s nejlepším ohodnocením.

Definice 2.2 ($possibleTurns$)**.** Tato funkce vrací všechny tahy, které je možné provést seřazené podle *pravděpodobnosti* p. Tahy, které je možné provést, jsou všechny možné útoky (tj. útoky z území, na nichž jsou alespoň 2 kostky a sousedí s územím protihráče), jejichž pravděpodobnost p je větší než 0,4 (bylo zvoleno experimentálně) a nebo je počet kostek na útočícím území roven 8 (největší možný

počet kostek na území). Pravděpodobnost p je spočtena následovně $p = P_{A\to D} \cdot P_D \cdot P_L$, kde $P_{A\to D}$ je pravděpodobnost úspěšného útoku z území A na území D, P_D je pravděpodobnost udržení území D po úspěšném útoku při tazích následujících hráčů a P_L je 2 (zvoleno experimentálně) v případě, že území A leží v největším regionu daného hráče, tj. je proveden útok z největšího území hráče, v opačném případě je $P_L = 1$. Výpočty těchto pravděpodobností jsou blíže diskutovány v původní bakalářské práci [3].

Definice 2.3 (h)**.** Hodnota této *heuristické funkce* udává ohodnocení stavu hry pro každého hráče ve smyslu maximalizace jeho *objektivní funkce*. Hodnota h pro hráče i je spočtena následovně $h(i) = D(i) + \sum_r^{R(i)} 5r + 50 \max(R(i))$, kde $D(i)$ je počet kostek hráče i, $R(i)$ jsou velikosti regionů hráče i a číselné konstanty byly zvoleny experimentálně.

Na nejvyšší úrovni je tah hráče určen tak, že pokud je zbývající časové omezení tahu větší než x sekund a zároveň je počet tahů v aktuální sérii tahů daného hráče menší než y, tak se nalezne nejlepší tah funkcí $maxN$. V opačném případě se vybere nejlepší možný útok s ohledem na pravděpodobnost výhry funkcí $possibleTurns$. Pokud žádný takový útok není možný, tah se ukončí. Parametry x a y byly experimentálně nastaveny na hodnoty 1,0 a 5.

3 Strojově učený model pro heuristickou funkci

Model substituuje *heuristickou funkci* a pro každého hráče vyhodnocuje v rámci aktuální konfigurace hry jeho šanci na výhru. Vstupem je tedy serializace aktuální konfigurace hry (stavu desky a stavu hráčů) do XX celých čísel a výstupem je vektor 4 hodnot v intervalu $< 0; 1 >$, který udává odhad modelu na výhru jednotlivých hráčů; 0 značí, že model si je jistý s nevýhrou hráče s konkrétním indexem (rep. jménem).

3.1 Serializace hry

Pro trénování modelu jsme navrhli serializaci hry do vektoru XX celých čísel, který odpovídá konkrétnímu stavu hry, tedy desky a hráčů na ní hrajících. Výsledný vektor má následující strukturu:

- 435 prvků z matice sousednosti (trojúhelník z 30x30 matice bez diagonály)
- 30 vlastníků polí
- 30 počtů kostek
- 4 velikosti největších regionů

zapsat matematicky

Po každém ukončeném tahu hráče je uschována serializace hry. Po konci hry je ke všem serializacím přiložen index vítěze konkrétní posloupnosti tahů a tento datový balíček je binárně uložen na disk. Pro tento účel byla upravena metoda `run` třídy `Game`, která se nachází v modulu `dicewars.server.game`. Pro tuto serializaci se dále používají funkce z vytvořeného modulu `dicewars.ml`.

3.2 Trénování modelu neuronové sítě

Všechny skripty zmíněné v této sekci se nachází v adresáři `ml-scripts`.

Čtyřikrát původní AI s Max^n algoritmem, souborový systém se soubory s jednotlivými konfiguracemi. Pro finální natrénování modelu bylo použito 201 291 různých konfigurací her s odpovídajícími indexy vítězů. Pro generování konfigurací byla hra spuštěna vytvořeným skriptem `dump-data-to-learn.py`.

Pomocí skriptu `transform-to-numpy.py` je tato datová sada ze souborů transformována na pole typu `numpy` o dimenzích $(N, 1 + 499) - N$ značí celkový počet konfigurací a $1 + 499$ je index vítěze

a kompletní serializace hry. Takto zkonstruované pole je uloženo jako jeden samostatný soubor pro další zpracování.

Implementace dalšího zpracování datové sady je pak v souboru `shuffle-datasets.py`, který ji načte, náhodně zamíchá podle první dimenze (tedy pořadí různých konfigurací) a následně rozdělí na *trénovací*, *validační* a *testovací* data. To dělá v poměru 70 %, 20 % a zbylých 10 % pro testovací data. Takto rozdělená sada je následně uložena do třech samostatných souborů.

Trénovací a validační část datové sady je následně použita pro natrénování modelu (skript `train-model.py`) – tím je v tomto případě *lineární neuronová síť* s následující architekturou:

bližší popis?

```
_________________________________________________
Layer (type)           Output Shape        Param #
=================================================
input_1 (InputLayer)   [(None, 499)]       0
_________________________________________________
dense (Dense)          (None, 64)          32000
_________________________________________________
dropout (Dropout)      (None, 64)          0
_________________________________________________
dense_1 (Dense)        (None, 32)          2080
_________________________________________________
dense_2 (Dense)        (None, 4)           132
=================================================
Total params: 34,212
Trainable params: 34,212
Non-trainable params: 0
_________________________________________________
```

Nutno zmínit, že v případě předpokládaných výsledků pro každou konfiguraci dochází ke *kategorizaci* – transformaci celočíselného indexu vítěze na vektor o počtu hráčů s hodnotou 1 na místě indexu vítěze a hodnotami 0 na ostatních pozicích.

Pro samotné natrénování je použit algoritmus `Adam` pro optimalizátor s *learning rate* na hodnotě $0,01$. Jako *loss* pak trénování používá instanci `CategoricalCrossentropy` dodanou přímo knihovnou `keras`. Jako velikost trénovací i validační dávky jsme experimentálně stanovili hodnotu 32 a počet epoch jsme opět experimentálně nastavili na 15.

Pro ověření správnosti modelu jsme připravili skript `check-test-accuracy.py`, který určí přesnost odhadů nad částí datové sady určenou pro testování (tedy 10 % z původní sady). Natrénovaný model z testovací datové sady o velikosti 20 130 správně určil celkem 19 804 vzorků, což značí úspěšnost cca 98.3 %.

rychlost predictu

3.3 Integrace modelu

Pro použití natrénovaného modelu v AI je upravena heuristická funkce, která nyní na základě předané konfigurace hry vyhodnocuje jednotlivé hráče, resp. jejich šanci na vítězství.

batch heuristic

4 Vyhodnocení implementace umělé inteligence

TODO: Zde pospat vyhodnocení. Uvést nějaké srovnání toho, jak se změnila úspěšnost po nasazení strojového učení.

5 Obsah odevzdaného archivu

Odevzdaný archiv `xkolar71.zip` obsahuje následující soubory:

- `dicewars/ai/xkolar71_orig.py`: Původní implementace umělé inteligence *bez strojového učení* popsaná v kapitole 2.
- `dicewars/ai/xkolar71/__init__.py` a `dicewars/ai/xkolar71/ai.py`: Balíček, který obsahuje implementaci umělé inteligence, která *používá strojové učení*. Viz kapitola 3.
- `dicewars/ai/xkolar71/model.h5`: Natrénovaný model založený na *neuronové síti*, který se používá pro modelování *heuristické funkce* pro ohodnocování stavů hry.
- `dicewars/ai/xkolar71_2.py`, `dicewars/ai/xkolar71_3.py`, `dicewars/ai/xkolar71_4.py`: Další instance původní implementace umělé inteligence `xkolar71_orig` vytvořené pro účely trénování modelu.
- `dicewars/server/game.py`: Modifikovaný soubor s třídou, která řídí hru. Byla provedena úprava pro generování konfigurací hry za účelem trénování modelu. Viz kapitola 3.1.
- `dicewars/ml/__init__.py` a `dicewars/ml/game.py`: Balíček obsahující funkce sloužící k serializaci konfigurací hry. Viz kapitola 3.1.
- `ml-scripts/dump-data-to-learn.py`: Skript pro generování konfigurací hry. Viz kapitola 3.2.
- `ml-scripts/transform-to-numpy.py`: Skript pro převod vygenerovaných konfigurací hry na pole typu `numpy`. Viz kapitola 3.2.
- `ml-scripts/shuffle-datasets.py`: Skript pro zamíchání konfigurací hry a jejich následné rozdělení na *trénovací*, *validační* a *testovací* datové sady. Viz kapitola 3.2.
- `ml-scripts/train-model.py`: Skript pro vytvoření a trénování modelu založeném na neuronové síti. Viz kapitola 3.2.
- `ml-scripts/check-test-accuracy.py`: Skript pro ověření správnosti modelu na testovacích datech. Viz kapitola 3.2.
- `requirements.txt`: Aktualizovaný seznam knihoven, na kterých je projekt závislý.
- `doc/xkolar71.pdf`: Tato dokumentace k projektu.

6 Závěr

TODO

Reference

[1] Luckhardt, C. A. a Irani, K. B. An Algorithmic Solution of N-Person Games. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*. Philadelphia, Pennsylvania: AAAI Press, srpen 1986. S. 158 — 162. AAAI'86.

[2] Sturtevant, N. A Comparison of Algorithms for Multi-player Games. In Schaeffer, J., Müller, M. a Björnsson, Y. (ed.). *Computers and Games*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. S. 108 – 122. ISBN 978-3-540-40031-8.

[3] Tureček, D. *Umělá inteligence pro deskovou hru*. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2018. Bakalářská práce. Vedoucí práce Ing. Karel Beneš.