Vysoké učení technické v Brně

Fakulta informačních technologií

Umělá inteligence pro hru Dicewars

Dokumentace k projektu z předmětu

Umělá inteligence a strojové učení

Klára Formánková (xforma14)

Adam Grünwald (xgrunw00)

Jakub Smejkal (xsmejk28)

2021/2022

Úvod

Cílem projektu bylo vytvořit umělou inteligenci (AI) pro strategickou hru Dicewars. V implementaci AI bylo třeba využít prohledávání stavového prostoru a strojové učení.

V dokumentaci je popsán způsob, jakým jsme dospěli k výsledné implementaci AI. Dokumentace se nejprve věnuje počátkům implementace, dále samotné implementaci agenta a prohledávání stavového prostoru a v poslední kapitole i vyhodnocení úspěšnosti.

Postup implementace

Ze začátku projektu jsme testovali základní dostupné funkce na seznamování agenta se světem (herní plochou). Pomocí těchto funkcí jsme vytvořili jednoduchého agenta inspirovaného ‘random’ agentem, který je dostupný v repozitáři. Tento náhodný agent se ukázal jako překvapivě schopný. I takto jednoduchý agent bez použití metod strojového učení a umělé inteligence dokázal plnohodnotně hrát a v turnajích získat i několik výher proti předpřipraveným agentům.

Dalším krokem po seznámení se s funkcemi byla implementace agenta, který nějakým způsobem zvažuje další možné tahy ve hře a na základě nich se snaží odehrát svůj aktuální tah co nejlépe. Tato implementace je popsána v další kapitole.

Implementace agenta a prohledávání stavového prostoru

Celý agent je řízen pomocí jednoduchého třístavového konečného automatu. Tah začíná ve stavu *‘transfer’*. Jakmile jsou provedeny všechny možné a vhodné transfery, nebo je vyčerpán maximální možný počet transferů v kole hráče (po odečtení 1 transferu, který je uschován pro poslední fázi), přejde automat do stavu *‘attack’*. Ve stavu *‘attack’*, vybírá agent pomocí algoritmu *MaxN* nejlepší možné útoky a postupně je provádí. Po provedení útoku je obnoven stav *‘transfer’*, protože došlo ke změně hracího pole a mohou se tak objevit nové výhodné přesuny. Pokud agent žádný přesun neprovede, automat přechází do stavu *‘attack’* a opět se pokouší najít vhodný útok. Pokud nedojde k provedení ani žádného útoku, automat se přepne do stavu *‘escape'*, kde se pokusí udělat přesun kostek pryč z hraniční oblasti, o kterou během hraní soupeřů pravděpodobně přijde. Pokud není možné nebo vhodné provést žádný další přesun v *‘escape’* fázi, je tah ukončen.

Transfery

Přesuny na hranice provádíme pomocí vytvoření stromu možných přechodů až do hloubky: `self.max_transfers - nb_transfers_this_turn`.

Pro každou hraniční oblast (seřazeno od nejslabší) se vytvoří strom všech možných cest do hloubky `'n'` (počet zbývajících přechodů za tah). Z tohoto stromu je vybrána největší možná hodnota při nejmenším možném počtu přesunů a tato operace je postupně provedena po krocích od nejhlubšího listu po kořen stromu.

Touto strategií je dosaženo posílení nejslabších hranic jak nejlépe je to jde, za nejmenší počet přesunů, před následným útočením nebo ukončením tahu.

Hranice se vyřazují z možné cesty ve stromě, aby nedocházelo k oslabování hranic posilováním jiných. Dále jsou hranice vyřazeny z prohledávání, pokud mají 7 a více kostek. V tuto chvíli jsou hranice brány jako dostatečné silné a není potřeba je posilovat.

Vytváření stromu probíhá prozkoumáním sousedů kořene (hranice) a následným rekurzivním prohledáváním sousedů těchto sousedů. Jako možní sousedé nejsou považována pole, která nepatří stejnému hráči a jiná hraniční pole. Pro zamezení cyklů je každý prohledávaný soused zařazen do pole, které obsahuje všechny prohledané oblasti před dosažením konkrétního pole.

Prvek stromu je tvořen ve tvaru `[nazev_pole, pocet_kostek]`.

Pro vytvoření stromu se využívá knihovna `treelib`.

Příklad posloupnosti transferů

Na tomto příkladu je vidět strom vytvořený pro pole `[33, 6]`. Pro toto pole bude ve stromě vybrána cesta `[19, 8]` ➜ `[30, 1]` ➜ `[33, 6]`, díky tomuto získáme za 2 tahy 8 kostek v kořeni stromu (zkoumané hranici). Tyto 2 tahy jsou vidět v grafu níže (červeně jsou zobrazeny uzly, ze kterých se bude transferovat, zeleně ty, do kterých se bude transferovat).

```
[33, 6]
└── [30, 1]
    ├── [19, 8]
    │   └── [27, 5]
    └── [32, 8]
        ├── [19, 8]
        │   └── [27, 5]
        └── [27, 5]
```

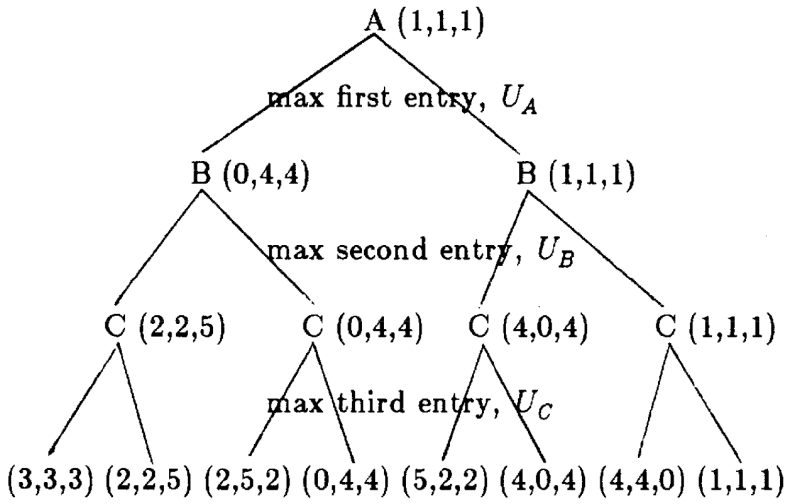
```
[33, 6]
└── [30, 8]
    ├── [19, 8]
    │   └── [27, 5]
    └── [32, 1]
        ├── [19, 8]
        │   └── [27, 5]
        └── [27, 5]
```

MaxN pro útoky

Útoky agenta se řídí algoritmem *MaxN*[1], tedy modifikací algoritmu *ExpectiMiniMax* pro více než 2 hráče. Základem algoritmu je strom, kde každý uzel reprezentuje stav hry a každá hrana vedoucí z uzlu určuje možný tah hráče. Uzly jsou strukturovány do několika úrovní, přičemž každá úroveň zahrnuje stav hry a možnosti tahu pro jednoho konkrétního hráče. Příklad takového stromu pro MaxN

[1] Zdroj pro pochopení algoritmu *MaxN* - https://www.aaai.org/Papers/AAAI/1986/AAAI86-025.pdf

je na Obrázku 1. Terminální uzly obsahují ohodnocení konkrétního stavu hry, do kterého se lze z kořene stromu dostat. Ohodnocení stavu hry je ve formě n-tice, kde *n* je určeno počtem hráčů, proto jsou na Obrázku 1 terminální uzly ohodnoceny trojicemi čísel. Jednotlivá čísla n-tice určují, jak výhodný je daný stav hry pro každého hráče. Čím je číslo větší, tím lepší je stav hry pro hráče. Ohodnocení stavu hry pro konkrétního hráče je v n-tici dáno pořadím hráčů. Ve stromě v příkladu lze tedy ohodnocení pro hráče najít v n-tici takto: (A, B, C). Cílem každého hráče je maximalizovat svoje číslo ohodnocující stav hry, a zároveň minimalizovat čísla ostatních hráčů. Každý hráč se proto při výběru nejlepšího tahu podívá na svoje možné tahy a vybere z nich ten, který pro něj ukazuje největší šanci na výhru, tzn. vybere co největší číslo v n-tici pro sebe a co nejnižší součet čísel ostatních hráčů. Pokud je na tahu hráč C z příkladu, vybere ohodnocení stavu hry, které maximalizuje vztah C - (A + B).



Obrázek 1: příklad MaxN stromu (zdroj: https://www.aaai.org/Papers/AAAI/1986/AAAI86-025.pdf)

Algoritmus *MaxN* je implementován funkcí `maxN(player, depth, board)`. Parametry funkce tvoří:

- `player`: hráč, který je právě na tahu,
- `depth`: hloubka prohledávání, která je dána počtem hráčů ve hře (tzn. algoritmus uvažuje následující tah každého hráče ve hře),
- `board`: aktuální stav herní plochy.

Algoritmus pracuje tak, že pokud byla dosažena maximální hloubka prohledávání, je volána funkce ohodnocující stav hry (funkce je popsána v podkapitole Ohodnocovací funkce), jinak dojde k naplnění počáteční n-tice nulami. Dále je pro každý uvažovaný útok hráče (získání popisuje podkapitola Získání uvažovaných útoků) rekurzivně volána funkce `maxN(next_player, depth+1, board_after_move)`, jejíž výsledek je následně porovnán s aktuálním nejlepším ohodnocením stavu hry. Pokud je výsledek lepší, je nastaven jako nový aktuální nejlepší tah a ten je i funkcí vrácen.

Funkce tedy rekurzivně dojde až k terminálnímu uzlu, získá n-tici ohodnocení a tu vrátí předchůdci terminálního uzlu. Předchůdce terminálního uzlu vybere ze získaných n-tic tu, která je pro něj jako pro hráče nejlepší a tu vrátí svému předchůdci. Tento předchůdce si opět vybere pro něj nejlepší n-tici a takto "probublávají" n-tice (vždy nejlepší tahy pro konkrétního hráče) až ke kořeni. Kořen

reprezentující našeho agenta se rozhodne, který ze získaných tahů (n-tic) je pro něj nejlepší a tento tah také odehraje.

Na rychlost funkce `maxN` má největší vliv parametr `depth` ovlivňující hloubku prohledávání. Pro potřeby projektu (hra 4 hráčů) je parametr nastaven na konstantu 4. Experimentálně jsme zjistili, že při prohledávání do hloubky 4 zbývá v tahu dostatek času na provedení přesunů i všech vhodných útoků. Času je v tahu dostatek i na hlubší prohledávání, to ale při experimentech nevedlo k výraznému zlepšení winrate agenta (řádově desetiny procenta). Zároveň je prohledáváním do hloubky 4 zajištěno prohledání následujících tahů všech hráčů ve hře, takže nemůže dojít k tomu, že by agent při rozhodování se, kam zaútočí, zapomněl prověřit možné tahy některého ze soupeřů, a tím se okradl o významné informace.

Získání uvažovaných útoků

Pro získání seznamu uvažovaných útoků hráče pro algoritmus MaxN jsou ze všech možných útoků hráče vybrány pouze takové útoky, které splňují následující:

- na poli, odkud je veden útok, je více než jedna kostka,
- pravděpodobnost zisku soupeřova pole je větší nebo rovna 0,5 a
- pravděpodobnost udržení pole do dalšího tahu je větší nebo rovna 0,3.

Výjimkou jsou pole, na kterých je 8 kostek. Taková pole jsou přidána do seznamu uvažovaných útoků hráče vždy.

Ohodnocovací funkce

Ohodnocovací funkce zjišťuje momentální stav herního pole pro zkoumaného hráče a následně tento stav ohodnocuje pomocí následujícího vzorce:

```
eval = player_areas_weight * number_of_player_areas +
biggest_region_weight * size_of_biggest_region - weak_borders_weight *
number_of_weak_borders + weak_enemies_weight * number_of_weak_enemies
```

Hodnoty v tomto vzorci jsou získány z prohledávání pole hráče. Slabé hranice jsou definovány jako hranice s nevýhodou v počtu kostek -2 a menší, slabí nepřátelé jsou definováni jako pole, kde má hranice výhodu o 1 kostku nad nepřítelem nebo pole kde hranice i nepřítel mají 8 kostek.

Získávání hodnot pro ohodnocovací funkci

Pro získání váhových hodnot pro ohodnocovací funkci jsme vytvořili bash skript, který spouští několik turnajů po 100 hrách čtyř hráčů s náhodnými hodnotami všech klíčových proměnných v AI, které jsou zahrnuty v ohodnocovací funkci tahu. Po ukončení turnaje je vypsána procentuální hodnota vítězství v turnaji.
Pomocí tohoto skriptu jsme získali optimální hodnoty, které vedly k maximalizaci úspěšnosti hry, a které jsou použity ve finální implementaci.

```
player_areas_weight = 0.5
biggest_region_weight = 0.55
weak_borders_weight = 0.3
weak_enemies_weight = 0.4
```

Dále pomocí tohoto skriptu získáváme rozhodovací hodnoty pro uvažování tahu a to pravděpodobnost úspěchu útoku a pravděpodobnost, že pole udržíme do dalšího tahu.

```
hold_prob = 0.3
succ_prob = 0.5
```

Escape

Cílem fáze *escape* je zachránit některé kostky z hraniční oblasti, o kterou agent při hraní soupeřů pravděpodobně přijde. Výběr oblastí odkud a kam se budou přesouvat kostky probíhá tak, že ze všech hraničních oblastí agenta jsou vybrány ty, kde je pravděpodobnost udržení pole menší než 0,3 a zároveň je možné kostky z tohoto pole přesunout na jiné (je zde dostatečná kapacita). Pokud existuje více vhodných oblastí, kam lze kostky přesunout, je vybrána oblast s nejvíce kostkami. Následně jsou takto vybrané hraniční oblasti seřazeny podle pravděpodobnosti udržení oblasti do dalšího kola a kostky z oblasti s nejnižší pravděpodobností jsou přesunuty.

Významné kroky implementace

Implementaci agenta provádělo několik významných kroků, které podstatně zvýšily úspěšnost agenta v turnajích, a tato kapitola je věnována právě jim.

Implementace výhodnějších transferů

Ze začátku jsme neuvažovali všechny možné cesty do hranice, kterou jsme chtěli posilovat. Tento přístup vedl pouze k náhodné úspěšnosti, kdy sousedé posilované hranice měli vyšší počet kostek. Po zvážení jsme implementovali postup, který pomocí stromové struktury uvažuje všechny možné transfery až do hloubky `'n'`, tímto jsme docílili možnosti posílení hranice i ze vzdálenějších polí a hlavně posilování potřebných hranic za co nejmenší cenu a s největší možnou úspěšností.

Oprava nefunkčních transferů

Implementace přístupem popsaným výše byla výhodná, ale hrací pole se mohlo dostat do stavu, kdy se agent zasekl a přestal hrát (nemohli jsme se dostat ze stavu *transfer* do stavu *attack*). Po debugování a opravě několika chyb jsme dosáhli mnohem větší úspěšnosti ve hrách, jelikož agent hrál celou hru a nemohlo dojít k nesmyslnému zaseknutí.

Implementace algoritmu MaxN

Náš původní plán útoků byl pouze vybrat nejlepší možný útok pro našeho agenta na základě počtů kostek hranic a nepřátel, a ten provést. Po implementaci algoritmu MaxN, který je popsán výš, kde prohledáváme všechny možné tahy a uvažujeme pouze ty vhodné, jsme dosáhli větší úspěšnosti útoků a tím pádem i vyšší úspěšnosti ve hrách.

Přidání fáze *escape*

K přidání fáze *escape* došlo až úplně ke konci implementace projektu, kdy jsme se v turnajích 4 hráčů stále umisťovali na třetí pozici a za prvními jsme zaostávali většinou o winrate 10%. Rozhodli jsme se proto implementovat tuto ústupnou fázi a k našemu překvapení jsme zjistili, že nejvýhodnější je vyhradit si na ústup z hranic pouze jeden přesun.

Ačkoliv je logika této fáze velmi jednoduchá, pomohla vylepšit winrate našeho agenta přibližně o 8%, což sice v turnajích 4 hráčů znamená většinou stále třetí příčku (občas i dělenou druhou), ale bezpochyby jsme se díky tomu dokázali velmi těsně přiblížit nejlepším poskytnutým AI.

Vyhodnocení úspěšnosti umělé inteligence

V začátcích implementace jsme využívali hry 2 hráčů, kdy jsme sami zkoušeli hrát proti našemu agentovi a zkoumali, jestli je vůbec schopen dělat základní tahy. Následně jsme dlouhou dobu zkoušeli, jak si náš agent vede v turnajích dvou hráčů (většinou proti '*dt.rand'*), abychom mohli vylepšovat logiku agenta. Až ve chvíli, kdy se nám s jistotou dařilo nad '*dt.rand'* vyhrávat, jsme se pustili do her více hráčů a hraní proti propracovanějším agentům. Pro debugování a opravu kódu jsme využívali hru 2 hráčů, kdy jeden hráč byl člověk a druhý naše AI. Při vytváření algoritmu pro transfery, jsme využili *visual-debugger.py* pro zjištění, zda se strom vytváří korektně pro daná pole. Běžné debugování jiných částí kódu jsme většinou prováděli pomocí výpisů na standardní výstup.

V pozdějších fázích implementace jsme pro hodnocení agenta využili turnaje 4 hráčů o několika stovkách až tisících her. Turnaje byly prováděny proti AI zmíněným v zadání jako ty, proti kterým se bude hodnotit.

Mnohokrát jsme zkoušeli i turnaje proti jiným AI. Náš agent se často umisťoval na předních příčkách. Proti AI ze zadání se náš agent umisťoval mezi prvními třemi, obvykle na třetím místě za *'kb.stei_adt'* a *'kb.stei_at'*, ale procentuální úspěšnost byla velmi podobná s těmito AI. Na obrázku je vidět výstup z turnaje 4 hráčů pro 5000 her.

```
.             .     % winrate [ .    / .     ] dt.stei   kb.sdc_pre_at kb.stei_adt kb.stei_at kb.stei_dt xsmejk29
kb.stei_adt   42.62 % winrate [ 5687 / 13344 ] 45.8/8028 41.9/7996     42.6/13344  38.2/7996  46.5/8032  40.7/7980
kb.stei_at    40.52 % winrate [ 5397 / 13320 ] 44.1/7992 41.0/8000     36.0/7996   40.5/13320 43.1/7988  38.4/7984
xsmejk29      32.37 % winrate [ 4312 / 13320 ] 34.8/8000 34.5/8000     28.9/7980   29.7/7984  34.0/7996  32.4/13320
kb.sdc_pre_at 20.42 % winrate [ 2722 / 13328 ] 22.3/7992 20.4/13328    18.9/7996   18.8/8000  21.7/7996  20.4/8000
kb.stei_dt    8.25  % winrate [ 1101 / 13344 ] 10.1/8020 8.5/7996      7.3/8032    7.6/7988   8.3/13344  7.8/7996
dt.stei       5.52  % winrate [ 737  / 13344 ] 5.5/13344 6.3/7992      4.5/8028    4.6/7992   7.2/8020   5.1/8000
```