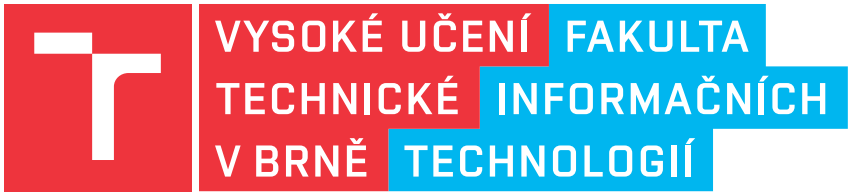


Umělá inteligence pro Válku kostek

SUI – Umělá inteligence a strojové učení

Jan Krejčí (xkrejc70)
Matěj Kudera (xkuder04)
Matěj Sojka (xsojka04)

Prosinec 2021

Obsah

1 Úvod

Cílem projektu bylo vytvoření umělé inteligence (dále jen AI) pro pozměněnou hru Dicewars[1], která se z pohledu AI vystihuje tím, že je pro každého hráče viditelné celé prostředí hry a jednotlivé akce v této hře jsou stochastické. Výsledná AI má dle zadání používat prohledávání stavového prostoru a prvky strojového učení, pomocí kterých rozhoduje o svých útocích a přesunech během hraní této hry.

Popis tvorby výsledné AI je rozdělen do následujících kapitol. Kapitola 2 pojednává o použitém prohledávání stavového prostoru. Kapitola 3 se zabývá tím, jak v tomto projektu bylo využito strojové učení. V kapitole 4 je blíže popsaná strategie, která se jevila jako nejefektivnější, a jsou zde ukázány výsledky strategií, které jsme postupně testovali. Dále je v kapitole 5 popsáno zhodnocení projektu. A nakonec je v kapitole 6 uveden seznam souborů, které byly během tvorby projektu vytvořeny a ke každému je uveden krátký popis.

2 Prohledávání stavového prostoru

Prohledávání stavového prostoru jsme v našem projektu primárně využili pro hledání efektivního útoku, který má naše AI provést při aktuálním stavu hry. Toto prohledávání jsme začali stavět na vzoru algoritmu Minimax[2], který byl doporučený zadáním projektu. Problémem je zde však, že ten algoritmus je v základní implementaci koncipovaný jen pro deterministická prostředí a pro hry dvou hráčů, kde ale ve hře Dicewars může hrát typicky N hráčů. Bylo tedy nutné tento algoritmus postupně upravovat až do podoby nutné pro danou hru.

2.1 Expectiminimax

První úpravou algoritmu bylo rozšíření pro stochastická prostředí. Běžně algoritmus generuje nové stavy dle dostupných akcí z původního stavu a to buď pro hráče min nebo max. Je zde tedy patrné, že je vždy z aktuálního stavu při použití jedné akce vytvořen jen jeden nový stav. V této hře je ale možné, že hráč buď políčku zabere a nebo nezabere (dáno stochasticky hodem kostek) a musí se počítat s oběmi těmito možnostmi. Algoritmus byl tedy rozšířen na verzi Expectiminimax[3], ve které se dovoluje vygenerovat více stavů jednou akcí a výsledné ohodnocení kvality této cesty je vypočítáno jako:

$$w = \sum_{i=1}^{N} p_i + w_i$$

Kde w je výsledné ohodnocení této cesty, N je počet možných nových stavů při aplikování jedné akce, p_i je pravděpodobnost tohoto výsledku akce a w_i je vrácené ohodnocení v tomto podstromě.

Ve hře Dicewars jsou při útoky možné dva výsledky, a to že se podaří pole obsadit a nebo ne. Při vracení ohodnocení cesty se tedy bude počítat suma pravděpodobnosti výhry vynásobené s ohodnocením výherního podstromu a pravděpodobností prohry vynásobení s ohodnocením podstromu s prohrou. Zde už jsme dopředu tušili, že by mohl být problém s tím, že se bude generovat moc stavů, takže jsme za účelem prořezání stromu odstranili generování cest s prohrou pokud je šance na výhru útoku větší než 95%.

2.2 MaxN

Dále bylo potřeba vyřešit problém, že algoritmus Expectiminimax není uzpůsoben pro hry více než dvou hráčů, protože v základu jen bere tahy s maximálním ohodnocením pro hráče co algoritmus zavolal a bere tahy s minimálním ohodnocením pro nepřátelského hráče (minimální hodnota pro nás znamená že pro druhého hráče je

[1] https://github.com/ibenes/dicewars
[2] https://en.wikipedia.org/wiki/Minimax
[3] https://en.wikipedia.org/wiki/Expectiminimax

to nejlepší možný tah). Algoritmus Expectiminimax jsme tedy rozšířili o logiku algoritmu MaxN[4]. Kde algoritmus MaxN postupně pro jednotlivé vrstvy stromu cyklicky bere jednotlivé hráče ve hře v pořadí jak aktuálně hrají hru a vybírá pro ně nejlepší tah co mohou udělat dle ohodnocení co dorazilo od listů stromu. Ohodnocení je zde potřeba provést pro všechny hráče, aby si mohl algoritmus vybírat maximální hodnotu dle konkrétního hráče na tahu.

Pro naše AI jsme toto rozšíření implementovali tak, že výsledné stavy dostupné v listech stromu jsme ohodnotili pro každého hráče zvlášť hodnotící funkcí jemu příslušnou (bližší popis této funkce se nachází v další sekci), dohromady tak vznikl vektor ohodnocení, ve kterém každá hodnota patří jednomu hráči. Při rekurzivním navracení této hodnoty z listové úrovně bylo dále potřeba vyřešit vynásobení pravděpodobnostmi konkrétní cesty, zde se mezi sebou vynásobily a sečetly hodnoty příslušné pro konkrétního hráče a tím vznikl nový hodnotící vektor.

2.3 Ohodnocení stavu

Algoritmy pro vrácení nejlepšího tahu za pomoci generování stavového prostoru vždy potřebují nějakou hodnotící funkci která přiřazuje pro konkrétního hráče a stav desky ohodnocení, které říká jak moc je tento stav pro tohoto hráče prospěšný k tomu, aby se mu podařilo danou hru vyhrát. Za tímto účelem jsme si tedy vytvořili funkci `evaluate_board`, která pro specifikovaného hráče a aktuální stav desky vypočítá jak moc je tento stav pro tohoto hráče dobrý. Hodnotící funkce počítá hodnocení hráče z následujících parametrů:

- Poměr vlastněného počtu polí hráčem vůči počtu všech polí na desce
- Poměr velikosti největší souvislé plochy vlastněné hráčem vůči ploše celého jeho území
- Poměr počtu vnitřních území hráče vůči ploše celého území hráče
- Poměr kostek v hráčově polích ve vrstvě o konkrétní vzdálenosti od hranice vůči maximálnímu počtu kostek v této vrstvě

Z těchto základních parametrů je prováděno hodnocení pro každého hráče. První tři parametry jsou jen obyčejná čísla v rozmezí 0 - 1. Poslední parametr je zde už trochu složitější, je to struktura, která obsahuje pro každou nalezenou vrstvu v hráčově oblasti poměr součtu kostek vůči maximálnímu možnému součtu kostek v této vrstvě. Bude zde tedy N hodnot, kde každá bude zase v rozmezí 0 - 1. Pro účely výsledného jednočíselného je potřeba těchto N hodnot spojit. Zde jsme se rozhodli využít priority, které uvádí, že je pro hráče lepší mít kostky co nejblíže k hranici, tedy poměr z vrstvy na hranici bude mít největší dopad na výsledné ohodnocení a se zvyšující se vzdáleností od hranice se bude priorita poměru postupně snižovat. Pořád je zde ale důležité, aby výsledná hodnota byla také v poměru 0 - 1 a šlo s ní nakládat stejně jako s ostatními parametry. Za tímto účelem byla tedy využita nekonečná součtová řada, která se sčítá do 1 a hodnoty z této řady byly použity jako priority, kterými se vynásobily poměry kostek v je jednotlivých vrstvách. Vzorec této řady je:

$$\sum_{i=1}^{\infty}(\frac{1}{2})^i = 1$$

Pro malý počet vrstev byly místo této řady použity dopočítané koeficienty, které se také sečetly do 1, ale i tak zachovaly vždy dvojnásobnou prioritu předchozí vrstvy.

Po zisku všech čtyřech parametrů kde každý je v rozmezí 0 - 1, jsme ještě implementovali globální priority, které určují jak moc je který parametr pro výsledné ohodnocení důležitý.

[4] https://www.aaai.org/Papers/AAAI/1986/AAAI86-025.pdf

Zde jsme si určili, jak moc je podle nás daný parametr pro výsledné ohodnocení důležitý a zvolili jsme následující priority:

- Poměr vlastněného počtu polí hráčem vůči počtu všech polí na desce = 1
- Poměr velikosti největší souvislé plochy vlastněné hráčem vůči ploše celého jeho území = 10
- Poměr počtu vnitřních území hráče vůči ploše celého území hráče = 4
- Výsledný poměr kostek ve vrstvách dle součtové řady = 8

Těmito prioritami jsme vynásobili příslušné parametry a výsledek vydělili součtem priorit aby znova vyšlo hodnota v rozmezí 0 - 1. Toto výsledné číslo je pak bráno jako ohodnocení aktuálního stavu desky pro zadaného hráče.

2.4 Časové omezení a hloubka stromu

Dalším důležitým problémem, který jsme při tvorbě našeho AI museli řešit je časové omezení na jedno kole hráče. Toto omezení je řešeno hrou tak, že při začátku kola je každému hráči přidělen čas během kterého může provádět své tahy a po jeho uplynutí je puštěn na řadu další hráč. Zde je tedy jasné, že budeme muset kontrolovat čas, který nám zbývá do konce hry a rozhodovat, zda chceme ještě počítat další možný tah a nebo už pro dané kolo skončíme. Tento problém jsme se rozhodli řešit tak, že budeme při rekurzivním generování rozhodovacího stromu kontrolovat čas, který nám zbývá a pokud je zbývající čas menší než 3 sekundy tak veškeré zanořování ukončíme a vrátíme nejlepší útok, který se našel ve stromě který se stihl vygenerovat. Tímto bude zajištěno, že AI bude stíhat dokončení potřebných výpočtů a může se rozhodnou ještě pro nějakou akci.

Jak už bylo zmíněno, tak se prohledávání stavového prostoru většinou provádí jen do nějaké omezené hloubky, hlavně u her, které mají velký počet stavů k vygenerování. Hra Dicewars nejen obsahuje velký stavový prostor, který by jsme nebyli v omezeném čase na jedno kolo schopni projít, ale také se ve hře děje spoustu věcí náhodně (přidělení nových kostek po tahu nebo šance na obsazení pole). Díky těmto náhodným jevům se každou další generovanou hloubkou stromu vzdalujeme od reálného stavu co nastane a tyto výsledky nám už tedy moc nepomohou. Z tohoto důvodu jsme se tedy rozhodli otestovat, jak moc dobré výsledky bude která hloubka generování poskytovat a podle toho vybereme tu s nejlepšími dosaženými výsledky (testy jsou popsány v další sekci).

Nakonec jsme ještě řešili možnost zavedení prořezávání generovaných stavů v každé vrstvě zanoření. Původní implementace algoritmu zde totiž z každého stavu generuje nové stavy pomocí všech dostupných útoků v daném stavu. Toto však při velkém množství validních útoků vede také na značné narůstání generovaného stromu a tím se i zvyšuje čas potřebný na vypočítání nejlepšího útoku. Z těchto důvodů jsme se tedy zkoušeli zavést mechanizmus který bude rozhodovat zda má cenu konkrétní útok generovat. Bližší informace k tomuto mechanizmu, jsou popsány v části 3.1.1.

2.5 Vyhodnocení nejlepší hloubky

Pro otestování nejlepší hloubky zanoření jsme se rozhodli provést několik turnajů proti oponentům stanoveních zadáním a v každém turnaji vyzkoušet jinou hloubku zanoření. Z těchto turnajů bude následně vybrána jako nejlepší úroveň zanoření ta, která poskytne nejlepší výsledky.

Hloubka	Procento výher
1	32.00%
2	37.00%
3	37.50%
4	34.00%
5	40.00%
6	41.50%
7	34.00%
8	39.50%
9	39.50%
10	33.50%

Tabulka 1: Procento výher z 200 her pro různé hloubky zanoření (testy provedeny s finálním prořezáváním a strategií).

Rozhodli jsme se otestovat hloubky zanoření 1 - 10. Jak lze ale z tabulky vidět tak výsledky byly ovlivněni né zrovna velkým počtem odehraných her, a je tedy možné že některé výsledky mohou být trochu zkreslené. Avšak je zde vyděl jakýsi trend který poukazuje na to že malá hloubka má menší úspěšnost a potom zase příliš velká hloubka má menší úspěšnost. Takováto výsledek dává logicky smysl protože malá hloubka generování předpovídá výsledek jen z pár tahů hry a zase moc velké hloubky už mají tak zkreslené stavy náhodnými jevy že už předpověď není moc použitelná. Velká hloubka generování má také problém s tím že moc dlouho trvá a stihne se tedy spočítat jen malí počet útoků. Pro naše AI jsme se tedy nakonec rozhodli použít hloubku zanoření rovnu 6.

3 Strojové učení

V této části rozebíráme jednotlivé přístupy, jakými jsme se pokoušeli implementovat strojové učení do našeho řešení, popisujeme jaké metody jsme použili a také jaký měli vliv na řešení.

3.1 Ohodnocení stavu pomocí regrese

Původním plánem pro použití strojového učení v tomto projektu, bylo jeho využití pro predikci ohodnocení stavu za použití regrese. Pro trénování jsme použili výše zmíněnou evaluační funkci(2.3). Spustili jsme hru a při každém stavu, ve kterém se náš bot nacházel, jsme si zaznamenali hodnotu příznaků a hodnotu evaluační funkce(250k záznamů). Poté jsme se pokoušeli natrénovat regresi za pomoci algoritmu Random Forest z knihovny *scikit-learn*[5]. Původní záměr byl poskytnout k trénování pouze data, kdy došlo ke zvýšení hodnoty evaluační funkce v dalším kroku, ale toto nefungovalo. Dále jsme se pokusili kromě stavů, které vedly ke zlepšení ohodnocení, poskytnout i stavy, které vedly ke zhoršení, kterým jsme ovšem přiřadili evaluační hodnotu stavu, do kterého vedly. Toto bohužel také nevedlo ke zlepšení při hraní hry. Pokoušeli jsme se dále manipulovat s trénovacími daty, ale nakonec jsme usoudili, že se pokusíme v rámci projektu radši o jinou aplikaci strojového učení.

3.1.1 Prořezávání pomocí klasifikace

V rámci prořezávání jsme použili funkce `probability_of_holding_area` a `probability_of_successful_attack` poskytnuté v rámci zadání. Pro ohodnocení tahu jsme použili hodnotu součinu těchto dvou funkcí.

[5] https://scikit-learn.org/stable/

Jako příznaky pro učení jsme zvolili následující informace o tahu:

- počet hráčů
- počet kostek, který má útočící políčko
- počet kostek, který má bránící se políčko
- počet kostek, který má útočník na hranici
- průměrný počet kostek, který má útočník na hranici
- průměrný počet kostek, který má obránce na hranici
- zda útok pochází z největší oblasti vlastněné hráčem
- počet útočníkových políček, které sousedí s políčkem, na které se útočí
- počet políček, které sousedí s políčkem, na které se útočí, a nejsou útočníka
- poměr počtu oblastí vlastněných útočníkem vůči počtu oblastí v celé hře
- směrnice vektoru vyjadřujícího aktuální distribuci kostek útočníka na hrací desce vůči hranicím

Následně jsme bota nechali hrát dva turnaje o pěti hráčích a padesáti hrách a při každé evaluaci tahu jsme si příznaky a ohodnocení zaznamenali do souboru. Tímto způsobem jsme získali 935k záznamů pro trénování. Sérií pokusů jsme došli k závěru, že budeme provádět binární klasifikaci, a že data rozdělíme podle hodnoty ohodnocení tahu následovně. Pro další rozgenerování budou určeny trénovací data, kde evaluační funkce je větší než 0.9 a do třídy pro nerozgenerování budou patřit data, která jsou ohodnoceny 0.6 a méně.

3.1.2 Implementace

Pro tvorbu modelu jsme se rozhodli použít knihovnu *scikit-learn*, konkrétně implementaci klasifikátoru *Random Forest*[6] a implementaci klasifikátoru pomocí metody podpůrných vektorů[7]. Pro předzpracování dat jsme se ještě rozhodli vyzkoušet metodu *StandardScaler*[8] a ověřit, jaké bude mít její využití dopad na výsledek.

Vytvořený model se načítá pomocí nástroje *Pickle* v konstruktoru třídy `Mattack`, aby toto načítání probíhalo pouze jednou za celou dobu trvání hry. Samotný model se využívá v souboru `xkuder04/utils/utils.py` v metodě `resonable_attacks_for_player`. Všechny metody pro tvorbu a práci s modely jsou implementovány v souboru `xkuder04/utils/model_utils.py`, data pro trénování jsou k dispozici ve složce `xkuder04/utils/model_data` a výsledné metody jsou ve složce `xkuder04/models`.

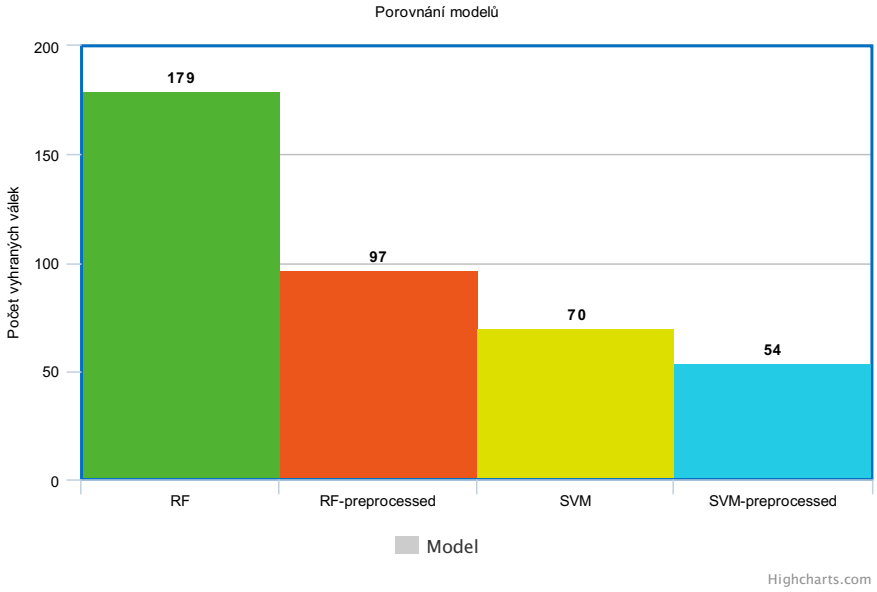
3.1.3 Vyhodnocení

Za účelem porovnat jednotlivé modely jsme se rozhodli je postavit ve hře proti sobě. Spustili jsme tedy 400 bitev po 4 hráčích mezi našimi čtyřmi modely, abychom zjistili, zda nějaká bude mít lepší výsledky než ostatní. Z 1 je zřetelné, že modely *Random Forest* skončily podstatně lépe v porovnání s modely *SVM* a že v obou případech předzpracování příznaků pomocí *StandardScaler* nevedlo ke zlepšení výsledků. Z tohoto důvodu jsme se rozhodli do našeho finálního bota vložit model *Random Forest*, který byl natrénovaný na nepředzpracovaných datech.

[6] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[7] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC

[8] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html



Obrázek 1: Výsledek 400 bitev ve 4 hráčích mezi našimi boty, kdy každý využíval jiný model.

.	.	% winrate [. / .]	dt.stei	kb.sdc_pre_at	kb.stei_adt	kb.stei_at	kb.stei_dt
xkuder04.xkuder04							
kb.stei_adt	45.16	% winrate [56 / 124]	50.0/56	43.4/76	45.2/124	35.7/56	51.7/60	45.2/124
kb.stei_at	37.93	% winrate [44 / 116]	40.6/64	26.8/56	35.7/56	37.9/116	48.2/56	37.9/116
xkuder04.xkuder04	28.00	% winrate [56 / 200]	28.3/120	29.8/124	25.0/124	27.6/116	29.3/116	28.0/200
kb.sdc_pre_at	26.61	% winrate [33 / 124]	36.7/60	26.6/124	21.1/76	21.4/56	28.6/56	26.6/124
dt.stei	5.00	% winrate [6 / 120]	5.0/120	8.3/60	1.8/56	4.7/64	5.0/60	5.0/120
kb.stei_dt	4.31	% winrate [5 / 116]	6.7/60	1.8/56	0.0/60	8.9/56	4.3/116	4.3/116

Obrázek 2: Výsledek turnaje bez prořezávání.

.	.	% winrate [. / .]	dt.stei	kb.sdc_pre_at	kb.stei_adt	kb.stei_at	kb.stei_dt
xkuder04.xkuder04							
xkuder04.xkuder04	38.50	% winrate [77 / 200]	41.4/116	36.4/140	35.3/116	37.5/112	42.2/116	38.5/200
kb.stei_adt	34.48	% winrate [40 / 116]	35.7/56	35.3/68	34.5/116	28.8/52	37.5/56	34.5/116
kb.stei_at	33.93	% winrate [38 / 112]	31.8/44	33.3/72	32.7/52	33.9/112	37.5/56	33.9/112
kb.sdc_pre_at	19.29	% winrate [27 / 140]	22.4/76	19.3/140	17.6/68	15.3/72	21.9/64	19.3/140
dt.stei	10.34	% winrate [12 / 116]	10.3/116	13.2/76	5.4/56	13.6/44	8.9/56	10.3/116
kb.stei_dt	5.17	% winrate [6 / 116]	8.9/56	6.2/64	5.4/56	0.0/56	5.2/116	5.2/116

Obrázek 3: Výsledek turnaje s prořezáváním pomocí *Random Forest*.

Poslední věcí bylo provedení padesáti turnajů pro implementaci bez prořezávání a pro implementaci s prořezáváním, aby bylo vidět, zda prořezávání mělo na výsledek nějaký vliv. Na obrázcích 2 a 3 je jasně vidět vliv, jaký prořezávání mělo na výsledek. Implementace bez prořezávání neměla nejspíš dostatek času na provedení nejvhodnější akce a proto její výsledek není tak kvalitní.

4 Strategie

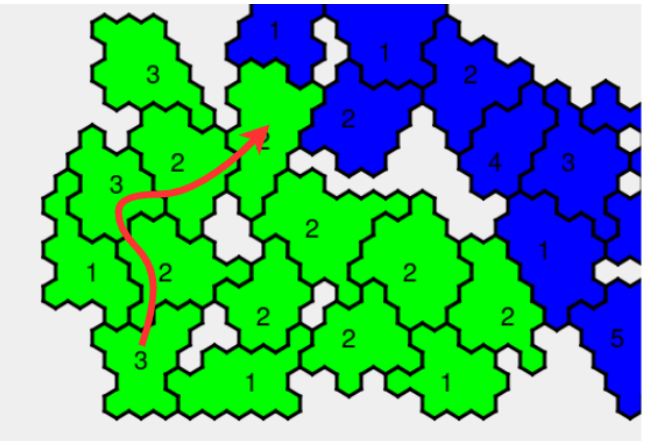
Pro získání nejlepšího možného výsledku ve hře bylo zapotřebí co nejlépe pracovat s přesuny kostek na vlastním území. Počet těchto přesunů byl omezen na 6 za jeden tah. Bylo tak nutné analyzovat různé herní strategie a porovnat je mezi sebou. V této sekci je popsána realizace přesunů kostek a útoků, průběh analýzy různých

strategií a výběr té nejoptimálnější. Testování probíhalo především formou turnaje čtyř hráčů s poskytnutími AI (*kb.stei_at*, *dt.stei*, *kb.sdc_pre_at*, *kb_stei_dt* a *kb_stei_adt*), která jsou i součástí hodnocení.

4.1 Přesun kostek

První jednoduchý přesun kostek byl realizován pomocí funkcí pro přesun přímo na hranice ze sousedních území a pro přesun z větší hloubky na sousední území hranic. Kritériem bylo vždy nalezení největšího možného přesunu kostek mezi dvěma úrovněmi. To se však ukázalo jako neúčinné v momentě, kdy potřebujeme, aby na sebe jednotlivé přesuny z větší hloubky vlastního území navazovaly a my tak na hranici dostali maximální možný počet kostek za co nejméně přesunů.

To jsme vyřešili pomocí implementace v souboru `transfer_tree.py`. Výsledné metodě se specifikuje vrstva, do které máme zájem transportovat kostky a počet kroků, který jsme ochotni pro tento transport poskytnout. Výsledek výpočtu by měla být cesta, která v co nejmenším počtu tahů dodá co nejvíce kostek ze vzdálenějších oblastí, jako například na obrázku 4. Pro vyhledávání používáme prohledávání do šířky (chceme šetřit tahy), kde ukončujeme prohledávání v momentě, kdy nám dojdou uzly k prohledávání nebo když najdeme řešení, které naplní uzel v cílové vrstvě.



Obrázek 4: Příklad transferu kostek z větší hloubky.

4.2 Uvažované strategie

Nejdříve jsme vyzkoušeli využít veškeré přesuny ihned na začátku tahu a přesunout tak kostky co nejblíže hranicím za pomoci prvních metod popsaných v předchozí podsekci 4.1, abychom co nejvíce podpořili následný útok. Útok zde byl hledán stromovým prohledáváním a pokud toto hledání nic nenašlo ale existovaly možné útoky (avšak byly horší než co by stromové očekávání odpovědělo jako dobrý útok), tak se provedl útok s největší pravděpodobností na dobití nepřátelského pole. Tyto neefektivní útoky zde byly použity pro posunutí stavu hry když by se nepřátelské AI rozhodly jen bránit, potom by hra nikdy neskončila. Tento velmi triviální způsob pro začátek fungoval obstojně a se základním prohledáváním stavového prostoru a jednoduchou evaluační funkcí bylo naše AI schopno porážet dvě nejslabší AI.

Při pozorování jsme zjistili, že naše AI často ukončuje tah se slabými hranicemi, ačkoli by je mohla jedním či dvěma přesuny značně posílit. Tento nedostatek měl fatální následky v podobě snadného vniknutí soupeře do vnitřních oblastí. Zkusili jsme tak na závěr tahu ponechat přesuny na finální posílení hranic. Při testování nám nejlépe vyšel poměr 2:1, tedy provedení maximálně 4 přesunů před útoky a minimálně 2 přesunů pro finální posílení hranic či oblastí sousedících s hranicemi. Tato strategie však značné zlepšení výsledků nepřinesla.

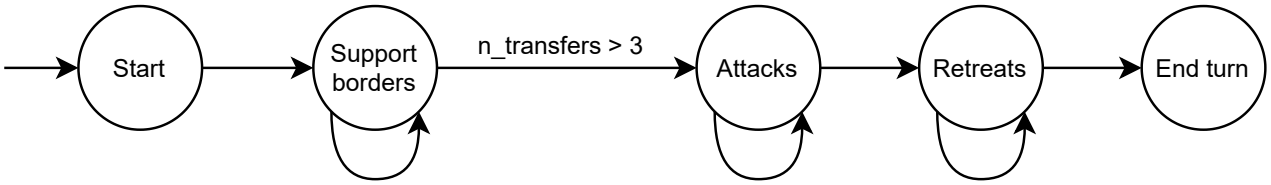
Dalším krokem bylo nahrazení těchto primitivních metod pro přesuny kostek sofistikovanějším řešením popsaným ve druhé části předešlé podsekce 4.1. Na začátku tahu jsme opět podpořili hranice s maximálně

4 přesuny, aby opět zbyly nejméně 2 přesuny pro finální posílení hranic. Zde byla fáze útoku upravena tak že pokud nebyl nalezen útok stromovím prohledáváním, tak se strategie snaží použít nejlepší možný útok dle pravděpodobnosti zabrání nepřátelského pole. Tyto útoky byly ale ořezány aby minimální pravděpodobnost byla větší než 60%. Pokud však mají všechny útoky menší pravděpodobnost na výhru, tak se z pravděpodobností 30% provede nejlepší útok co se našel (vždy bude mít ale menší šanci na výhru než 60%). Tato strategie značně zlepšila celkové výsledky a naše AI bylo schopno porážet tři nejslabší AI (*dt.stei*, *kb.sdc_pre_at* a *kb_stei_dt*).

Poslední testovanou strategií bylo nahrazení finálního posílení hranic za záchranu co nejvíce kostek z oblastí s největší pravděpodobností dobytí soupeřem. Tento ústup je realizován v metodě `retreat_forces()` v souboru `xkuder04/xkuder04.py`. Samotné vyhodnocení nejvíce ohrožených oblastí a na základě toho zvolení vhodných přesunů je implementováno v metodě `retreat_transfers()` v souboru `xkuder04/utils/transfer_utils.py`. Útok se zde dostal jen lehkou úpravu a to že jsme zamezily maximální počet útoků na konkrétní zvolené číslo. Omezením počtu útoků jsme zde chtěly dosáhnout aby AI zbytečně neprovádělo útoky založené jen na pravděpodobnosti výhry a náhodě a spíš si počkalo jestli se stav v dalším kole nezlepší. Společně s úpravou strategie ohledně útočení dosáhlo AI prozatím nejlepších výsledků. Podrobnější popis jednotlivých částí výsledné strategie je v následující podsekci.

4.3 Výsledná strategie

Výsledná strategie je kombinací strategií popsaných v předchozí podsekci 4.2, které při testování vykazovaly nejlepší výsledky. Logika je implementována v souboru `xkuder04/xkuder04.py`. I díky této strategii dokázala naše AI porážet všechny zbývající AI, jak bylo popsáno v podsekci 3.1.3. Pro lepší představu je vyobrazena diagramem na obrázku 5.



Obrázek 5: Výsledná strategie přesunů kostek a útoků

Výsledná strategie z obrázku 5 se skládá z následujících částí:

- *Supoort borders* označuje počáteční přesun kostek na hranice. To je realizováno pomocí již zmíněného způsobu vícenásobného přesunu kostek popsaného ve druhé části předešlé podsekce 4.1, kterému je umožněno provést maximální počet dovolených přesunů. Díky tomu dojde k přesunu kostek i z niternějších oblastí. Pokud je počet přesunů menší než 4, je nalezeno a uskutečněno další nejoptimálnější posílení hranic. Jinak se přechází na útoky.

- *Attacks* provádí sérii nejefektivnějších útoků na základě výsledků z prohledávání a ohodnocení stavového prostoru popsaného v sekci 2. Útok je proveden pouze pokud nebyl vyčerpán čas a maximální počet útoků v aktuálním tahu. Pokud se stromovým hledáním nic nenajde tak se berou útoky s pravděpodobnost výhry vyšší než prahová hodnota 0.6, která nám při testování vyšla jako nejlepší možná. Pokud žádný útok nemá šanci na výhru vyšší než 60% tak se z pravděpodobností 30% použije nejlepší nalezený útok dle pravděpodobnosti výhry (menší však než 60%)

- *Retreats* značí stahování kostek z nebezpečných oblastí, detailněji popsáno v podsekci 4.2. K dispozici jsou přesuny, které nebyly použity pro počáteční posílení hranic.

5 Závěr

V tomto projektu se nám povedlo udělat AI pro hru Dicewars, která používá prohledávání stavového prostoru pro nalezení nejlepšího možného útoku a pro nalezení nejlepšího přesunu kostek na hranici, dále jsme vytvořili klasifikátor, který slouží pro prořezávání generovaných stavů při hledání nejlepšího útoku a nakonec jsme tyto funkčnosti zabalili do strategie, která řeší logiku hraní našeho AI v jednotlivých kolech. Jednotlivé části AI jsme postupně ladili na turnajích proti poskytnutým oponentům, dokud jsme se nedostali na uspokojivé výsledky.

Celkově jsme spokojení s dosaženými výsledky, ale je jasné, že by naše AI mohla být dále vylepšováno pro dosažení ještě lepších výsledků. Možností by bylo například zapojení posilovaného učení pro prořezávání stromu možných útoků, případně by mohlo být spojeno i s vybíráním nejlepších možných přesunů. Dále by bylo možné vytvořil složitější hodnotící funkci, která by mohla započítávat do svého ohodnocení i to, jak moc jsou poškozeni ostatní hráči daným tahem (třeba rozbití největšího souvislého území).

6 Odevzdané soubory

Součástí odevzdaného archivu jsou následující soubory:

- `xkuder04/xkuder04.py` – hlavní skript obsluhující kompletní tah AI. To zahrnuje všechny části strategie popsané v sekci 5, tedy přesuny, útoky a ukončení tahu. .
- `xkuder04/Mplayer.py` – skript pro udržení a snadné získání informací o hráči v aktuálním stavu hry.
- `xkuder04/Mattack.py` – script obsahující funkce pro prohledávání stavového prostoru, popsáno v podsekci 2.
- `xkuder04/utils/debug.py` – správa debug výpisů.
- `xkuder04/utils/models_util.py` – skript obsahující funkce pro správu modelů, popsáno v podsekci 3.1.2.
- `xkuder04/utils/transfer_tree.py` – skript obsahující bfs prohledávání pro nalezení nejoptimálnějších přesunů kostek, popsáno v podsekci 4.1.
- `xkuder04/utils/transfer_utils.py` – Skript obsahující zbylé funkce pro přesuny kostek, které byly popsány v podsekci 4.1.
- `xkuder04/utils/utils.py` – skript obsahující funkce pro ohodnocení jednotlivých útoků.
- `xkuder04/model_data/` – složka obsahující data pro trénování, popsáno v podsekci 3.1.2.
- `xkuder04.pdf` – dokumentace.