MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Katarína Dančejová

Analýza platných ťahov v hre Scrabble

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: RNDr. Daniel Zeman, Ph.D.
Studijní program: Informatika
Studijní obor: Programování a softwarové systémy

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Ďakujem môjmu školiteľovi RNDr. Danielovi Zemanovi, Ph.D., za vypísanie tejto zaujímavej témy, odbornú pomoc, užitočné rady a pripomienky.

Ďakujem českému online Scrabble slovníku SCRABBLEX za to, že mi odporučil výborný zdroj pre vytvorenie českého slovníka pre môj program.

Taktiež ďakujem mojej rodine za jej trpezlivosť a ohľaduplnosť počas písania mojej práce; môjmu Danielovi za to, že som sa naňho mohla kedykoľvek obrátiť s mnohými otázkami, za podporu a pomoc pri formulovaní myšlienok. Mojej študijnej skupinke ďakujem za cenné rady a spätnú väzbu.

Název práce: Analýza platných ťahov v hre Scrabble

Autor: Katarína Dančejová

Ústav: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: RNDr. Daniel Zeman, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Cieľom práce je vytvoriť program, ktorý slúži na analýzu ťahov v spoločenskej hre Scrabble. Slovník jazyka je uložený v dátovej štruktúre GADDAG, ktorá umožňuje rýchle vyhľadávanie slov v slovníku. Ťahy sú vyhodnocované na základe heuristík pre strednú hru a koncovku. Tieto heuristiky berú v úvahu aj strategické aspekty pozície a snažia sa o dosiahnutie čo najvyššieho počtu bodov na konci hry. To má za následok zvýšenie úrovne analýzy ťahov natoľko, že je program hodnotný aj pre skúsenejšieho hráča. Simuláciami je porovnávaná výkonnosť umelej inteligencie používajúcej heuristiku s umelými inteligenciami, ktoré heuristiku nepoužívajú. Umelá inteligencia s heuristikou dosahuje v partiách výrazne lepšie výsledky, či už hru začína prvá alebo druhá v poradí. Na analýzu ťahov v Scrabble je vytvorené grafické používateľské rozhranie, ktoré navrhuje najlepšie ťahy podľa preukázane výhodnej heuristiky.

Klíčová slova: scrabble desková hra optimální tah

Title: Analysis of valid moves in Scrabble

Author: Katarína Dančejová

Institute: Institute of Formal and Applied Linguistics

Supervisor: RNDr. Daniel Zeman, Ph.D., Institute of Formal and Applied Linguistics

Abstract: The bachelor thesis aims to create a program for the analysis of moves in Scrabble. The language dictionary is stored as a GADDAG structure, which enables fast searching of the words of the dictionary. The heuristics for the middlegame and the endgame are applied. These heuristics' goal is to reach the best score at the end of the game, so they also consider the strategic aspects of a position. Thus the program can be helpful even for advanced players. The simulations are made to compare the artificial intelligence using heuristics against artificial intelligence not using heuristics. Artificial intelligence with heuristics has significantly better performance in the games, whether it was starting or not. We create a graphical user interface that recommends the best moves based on the provingly effective heuristics.

Keywords: scrabble board game optimal move

Obsah

Úvod **3**

1 Pravidlá hry **4**
1.1 Začiatok hry 4
1.2 Priebeh hry 4
1.3 Ukladanie kameňov 4
1.4 Bodovanie 5
1.5 Koniec hry 6
1.6 Povolené slová 6

2 Dátové štruktúry na ukladanie slovníka **7**
2.1 Trie 7
2.2 Štruktúra DAWG 8
2.2.1 Generovanie ťahov 8
2.3 Štruktúra GADDAG 8
2.3.1 Generovanie ťahov 10

3 Heuristika **11**
3.1 Stredná hra 11
3.2 Koncovka 12

4 Implementácia **14**
4.1 Úprava pythonovskej knižnice GADDAG 14
4.1.1 Neúspešný pokus o podporu znakov s diakritikou 14
4.1.2 Funkčná podpora znakov s diakritikou 15
4.1.3 Pridané funkcie 17
4.2 Vytvorenie slovníka 18
4.3 Generovanie ťahov 18
4.4 Heuristika 20
4.4.1 Stredná hra 20
4.4.2 Koncovka 21
4.4.3 Simulácie 22
4.5 Možné vylepšenia 24

5 Programátorská dokumentácia **26**
5.1 GADDAG knižnica 26
5.1.1 DLL knižnica v jazyku C++ 26
5.1.2 Pythonovská časť knižnice 28
5.2 Hracie kamene, hracia doska, zásobníky 30
5.2.1 Hracie kamene 30
5.2.2 Hracia doska 32
5.2.3 Zásobníky 33
5.3 Umelá inteligencia hráča 34
5.3.1 Reprezentácia stavu hry 34
5.3.2 Umelá inteligencia 35
5.4 Grafické používateľské rozhranie 38

5.4.1 Inicializačné metódy 39
5.4.2 Aktualizačné metódy 39
5.4.3 Metódy spravujúce signály objektov 40
5.4.4 Metódy na prepojenie front-endu a back-endu 40

6 Používateľská dokumentácia **42**
6.1 Minimálne požiadavky 42
6.2 Inštalácia 42
6.3 Ovládanie 42
6.3.1 Panel s menu 42
6.3.2 Zoznam kameňov vo vrecúšku 44
6.3.3 Zásobník 44
6.3.4 Analyzér 44
6.3.5 Ostatné 44

Záver **45**

Zoznam použitej literatúry **46**

A Prílohy **47**
A.1 Zdrojový kód 47
A.2 Obsah vrecúška v českej verzii Scrabble 47
A.3 Záznam zápasu SMART vs. NORMAL 47

Úvod

Scrabble® je stolná hra pre 2–4 hráčov, v ktorej hráči prikladaním písmen skladajú na hracej doske slová. Všetky hracie kamene s písmenami majú určité bodové ohodnotenie a na hracej doske sú rozmiestnené prémiové políčka, ktoré môžu hráčovi zisk bodov znásobiť. Vyhráva hráč, ktorý po vyčerpaní všetkých písmen v balíčku získa najviac bodov (Slovenský spolok SCRABBLE).

Jadrom programu, ktorý umožňuje analyzovať partie v tejto hre, je umelá inteligencia hráča. Tá sa musí vysporiadať s niekoľkými vecami. Jednou z nich je to, že na kontrolu platnosti slov, ktoré chce hráč zahrať, je potrebné udržiavať si veľmi rozsiahly slovník slov daného jazyka. Preto musí byť slovník uložený vo vhodnej dátovej štruktúre, ktorá umožňuje rýchle vyhľadávanie slov v slovníku.

Začiatočník v tejto hre zahrá slovo, ktoré zabezpečí v danom kole najlepší bodový zisk. Profesionálni hráči však vedia, že pre získanie najvyššieho možného počtu bodov na konci hry je nutné strategizovať a niekedy si vyberať aj ťahy, ktoré nám v kole prinesú síce menší bodový zisk, ale aj umožnia v neskorších kolách zahrať vysoko ohodnotené slová, ktoré túto dočasnú nevýhodu nakoniec vykompenzujú. Aby sa naša umelá inteligencia myslením priblížila profesionálnemu hráčovi, musí pre výber najlepšieho ťahu uplatniť určitú heuristiku. Tá by mala vybrať najlepší ťah nielen z bodového, ale aj zo strategického hľadiska.

Cieľom tejto práce je vytvoriť program, ktorý sa dokáže jednoducho prispôsobiť rôznym jazykovým variantom hry Scrabble, dokáže zahrať ťah, za ktorý by sa nehanbil ani skúsený hráč, a zvládne pozíciu úspešne zanalyzovať v rozumnom čase.

1. Pravidlá hry

Pravidlá hry v tejto kapitole sú definované na webových stránkach Českej asociácie Scrabble (skrátene ČAS) a Slovenského spolku SCRABBLE (Česká asociace Scrabble; Slovenský spolok SCRABBLE). Základné pravidlá sú zhodné v oboch jazykoch, odlišnosti sú v slovníkoch a prameňoch používaných na overovanie platnosti slov.

Organizované súťaže sa hrajú len vo dvojiciach a na čas, v tejto práci sa budeme venovať iba hre dvoch hráčov.

1.1 Začiatok hry

Na začiatku hry je vo vrecúšku 100 hracích kameňov. Na každom kameni je písmeno, ktoré kameň reprezentuje a jeho bodová hodnota. Každý z hráčov si z vrecúška náhodne vytiahne jeden kameň. Poradie v hre je potom dané lexikografickým usporiadaním písmen na vytiahnutých kameňoch. Ak je vytiahnutý žolík, hráč, ktorý ho vytiahol, ťahá znova. Po určení poradia hráčov sú kamene vrátené naspäť do vrecúška a obsah vrecúška je opäť zamiešaný. Podľa poradia si hráči následne ťahajú z vrecúška po 7 kameňov a uložia si ich na svoj zásobník tak, aby ich ostatní hráči nevideli.

Začínajúci hráč uloží prvé slovo použitím aspoň 2 kameňov zo zásobníka tak, aby jeden z nich ležal na stredovom políčku dosky, označeným hviezdičkou. Pokladanie musí byť v súlade s pravidlami uvedenými v sekcii 1.3. Okrem tejto podmienky sa prvý ťah riadi všeobecnými pravidlami pre vykladanie slov. Po ukončení ťahu nasleduje ďalší hráč.

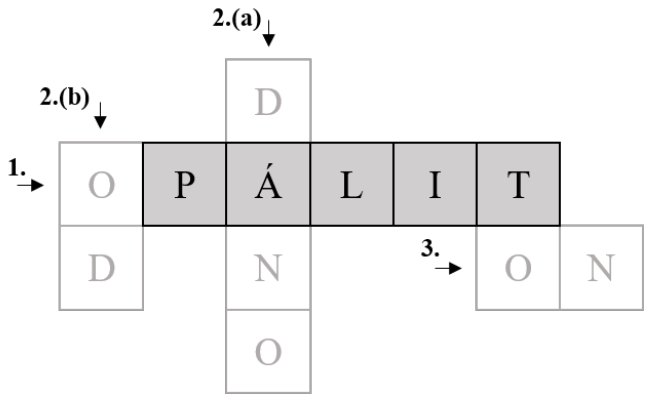
1.2 Priebeh hry

Hráč, ktorý je na ťahu, vyberie zo zásobníka aspoň jeden kameň a položí ho na hraciu dosku (podrobnejšie v sekcii 1.3). Hráč si následne sčíta body za všetky vytvorené alebo obmenené slová (viď sekcia 1.4). Potom si vylosuje z vrecúška toľko kameňov, aby ich mal na zásobníku 7, ak je to možné. Obsah vrecúška na začiatku hry je v prílohe A.2.

Okrem toho hráč môže namiesto uloženia slova na hraciu dosku kolo preskočiť (*pass*) alebo si vymeniť ľubovoľný počet kameňov za iné, náhodne ťahané z vrecúška. Tieto možnosti však náš program nepodporuje.

1.3 Ukladanie kameňov

Aspoň jeden z kameňov vyloženého slova musí susediť (vertikálne alebo horizontálne) s nejakým skôr vyloženým kameňom na hracej doske. Všetky novovyložené kamene musia ležať v jednom riadku alebo stĺpci a nesmie medzi nimi byť medzera. Novovyložené slovo musí byť čitateľné zľava doprava alebo zhora nadol a musí byť povolené (viď sekcia 1.6). Na všetkých miestach, kde susedí so skôr vyloženými kameňmi, musia tiež vzniknúť povolené slová čitateľné jedným z povolených smerov. Žiadny z už vyložených kameňov nesmie byť presúvaný.



Obr. 1.1: Spôsoby pridania nového slova na hraciu dosku. Vytvorené slová sú: 1. OPÁLIT; 2.(a) DÁNO; 2.(b) OPÁLIT a OD; 3. TO a ON.

Príklad so všetkými spôsobmi, ako je možné vytvoriť nové slovo, je vidieť na obr. 1.1, kde sa snažíme vytvoriť slovo/slová pri už existujúcom slove PÁLIT. Nové slová teda môžu vzniknúť nasledovnými spôsobmi:

1. Pridaním jedného alebo viacerých kameňov k slovu, ktoré už je na hracej doske (t. j. pridaním prefixu alebo suffixu).

2. Umiestnením slova kolmo k už existujúcemu slovu na hracej doske. Takéto slovo buď

   (a) využíva jeden z kameňov kolmého slova, alebo

   (b) pridaním kameňov pred alebo za tento kameň vytvorí platné slovo.

   Nové slovo takto môže premostiť aj niekoľko kolmých slov.

3. Položením celého slova rovnobežne vedľa už existujúceho slova na doske tak, že tesne susediace kamene tiež vytvárajú platné slová.

Prázdny kameň (žolíka) je možné použiť namiesto ktoréhokoľvek písmena. Hráč musí oznámiť, aké písmeno žolík zastupuje a toto písmeno bude zastupovať až do konca hry.

1.4 Bodovanie

Za každý kameň všetkých novovytvorených alebo obmenených slov dostane hráč toľko bodov, koľko je na nich uvedené (žolík má hodnotu 0). Kamene, ktoré ležia vo viacerých slovách súčasne, sa započítavajú v každom slove. Okrem toho na počet bodov vplývajú tzv. „prémiové políčka“, na hracej doske sú farebne odlíšené nasledovne:

- svetlomodré políčko zdvojnásobuje hodnotu kameňa, ktorý je na ňom položený;

- tmavomodré políčko strojnásobuje hodnotu kameňa, ktorý je na ňom položený;

- ružové políčko zdvojnásobuje hodnotu celého slova, ktoré ním prechádza;
- červené políčko strojnásobuje hodnotu celého slova, ktoré ním prechádza.

Prémie si započíta len hráč, ktorý na prémiové políčko položil kameň. Hráč, ktorý vo svojom ťahu využije kameň ležiaci na prémiovom políčku, si už započítava iba základnú hodnotu kameňa. V prípade, že novovyloženým slovom hráč pokryje písmenné aj slovné prémie, najprv aplikuje písmenné prémie a potom aj slovné. To o. i. znamená, že ak nové slovo prechádza dvomi červenými políčkami, je možné získať až 9-násobok pôvodnej hodnoty slova.

Hráč, ktorý v jednom ťahu umiestni na hraciu dosku všetkých 7 kameňov zo zásobníka, dostáva zvláštnu prémiu 50 bodov – ide o tzv. „bingo“. Táto prémia sa započítava až po započítaní písmenných a slovných prémií.

Na konci hry sa hráčom zo skóre odpočíta hodnota kameňov, ktoré mu ostali v zásobníku. Ak má na konci hry hráč prázdny zásobník, pripočítava sa k jeho skóre súčet kameňov v súperovom zásobníku.

1.5 Koniec hry

Hra končí, keď hráč využil všetky svoje kamene a vrecúško je prázdne; alebo ak sa všetci hráči 2 kolá po sebe vzdajú ťahu. K bodom hráčov sa nakoniec pripočítajú, resp. odpočítajú body z nevyužitých písmen na zásobníku, viď 1.4. Hráč s najvyšším počtom bodov vyhráva.

1.6 Povolené slová

Vo všeobecnosti sa pre každú jazykovú variantu hry Scrabble používa zoznam povolených slovníkov, ktoré definujú, aké slová je možné vytvárať. Slová sa typicky smú skloňovať.

Pre českú variantu hry sa oficiálne používajú slovníky ČAS (Česká asociace Scrabble), kde patria NOVEX, BLEX 2001, BLEX PLUS 2003 a BLEX PLUS 2007. Slová, ktorých platnosť nie je možné posúdiť použitím slovníkov ČAS, sa posudzuje podľa tzv. prameňov, kde patria napríklad Slovník spisovného jazyka českého I–VIII, Slovník spisovné češtiny pro školu a veřejnost, Pravidla českého pravopisu a. i.

Pre slovenskú variantu hry sa pre posudzovanie platnosti slov používajú slovníky KSSJ (Krátky slovník slovenského jazyka), PSP (Pravidlá slovenského pravopisu), SSSJ (Slovník súčasného slovenského jazyka) či elektronický slovník El Siňo pre slová dĺžky 2–9 písmen (Slovenský spolok SCRABBLE).

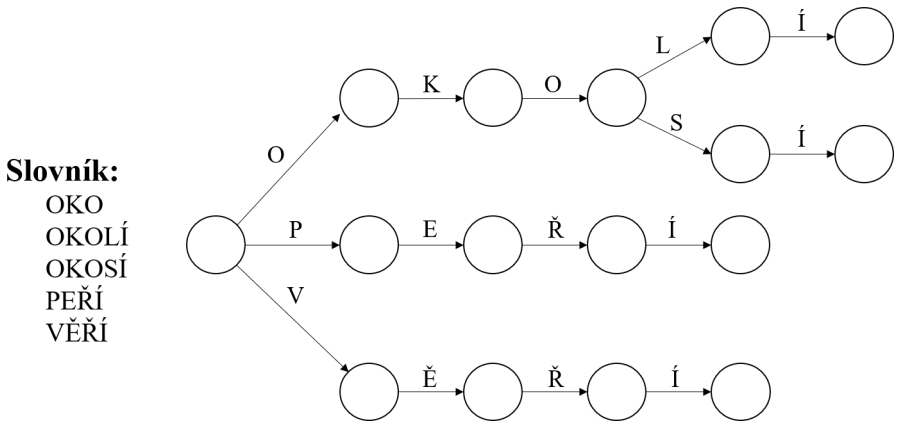
2. Dátové štruktúry na ukladanie slovníka

Aby sme vedeli hľadať možné ťahy v Scrabble a posudzovať platnosť slov, je potrebné mať k dispozícii slovník platných slov. Takýto slovník pre Scrabble zvyčajne neobsahuje všetky platné slová daného jazyka, pretože vytváranie slov dlhších ako 9 písmen nie je veľmi zvyčajné. Napriek tomu je slov kratších ako 10 písmen stále veľmi veľké množstvo, a preto je nevyhnutné zvoliť si vhodnú dátovú štruktúru na uloženie slovníka.

Pri výbere dátovej štruktúry môžeme buď hľadieť na to, koľko zaberie v pamäti miesta, alebo na to, ako rýchlo vieme v štruktúre vyhľadávať. V súčasnej dobe už nie je taký veľký problém s kapacitou pamäte, a preto sa môžeme sústrediť na hľadanie štruktúry pre čo najrýchlejšie vyhľadávanie slov.

2.1 Trie

Trie (názov odvodený od anglického „retrieval“ – získanie, vyhľadávanie) je dátová štruktúra určená na ukladanie dvojíc kľúč-hodnota, kde je kľúč väčšinou znakový reťazec (Mareš, 2021). Ide o orientovaný stromový graf s koreňom (obr. 2.1). Hrany zvyčajne reprezentujú znaky (písmená) a kľúč vrcholu je získaný zreťazením znakov hrán cesty z vrcholu do koreňa. To znamená, že kľúč vrcholu je zároveň prefixom kľúčov všetkých jeho potomkov.



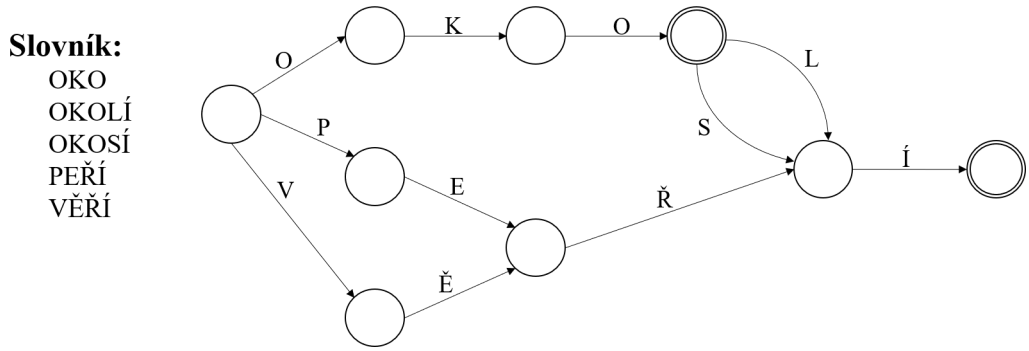
Obr. 2.1: Trie pre zadaný slovník.

Časová zložitosť vyhľadávania je $O(m)$, kde m je dĺžka reťazca. Pre porovnanie, vyhľadávanie v binárnom vyhľadávacom strome je v čase $O(log(v))$, kde v je počet vrcholov štruktúry. Dĺžka reťazca m je pre väčšie slovníky výrazne kratšia ako počet vrcholov v binárnom strome v. Z hľadiska časovej zložitosti sa pre veľké dátové štruktúry preto trie javí ako lepšia voľba.

2.2 Štruktúra DAWG

Pre ukladanie slovníka nie je potrebné mať vo vrcholoch uložené hodnoty, preto pre tieto účely môžeme trie brať ako deterministický konečný automat, kde vrcholy budú stavy automatu a vrcholy, v ktorých slovo končí, budú konečnými stavmi automatu. Automat potom rozpoznáva jazyk tvorený z množiny všetkých slov v slovníku. Túto reprezentáciu štruktúry využíva dátová štruktúra DAWG (Appel a Jacobson, 1988).

Je možné vytvoriť veľa rôznych automatov rozpoznávajúcich daný jazyk, no jeden z nich bude mať najmenší možný počet stavov, nazvime ho „minimálny automat“. Je zrejmé, že jeho grafom nebude strom, pretože v strome do každého vrcholu vedie práve jedna hrana. Redukciou však vieme automat, ktorého grafom je trie, upraviť na minimálny automat. Štruktúra DAWG (skratka pre „Directed Acyclic Word Graph“ – orientovaný acyklický slovný graf) pracuje s minimálnym automatom, t. z. umožňuje, aby do jedného vrcholu viedlo viacero hrán, čo v porovnaní s trie výrazne znižuje pamäťovú zložitosť (obr. 2.2).



Obr. 2.2: Štruktúra DAWG pre zadaný slovník.

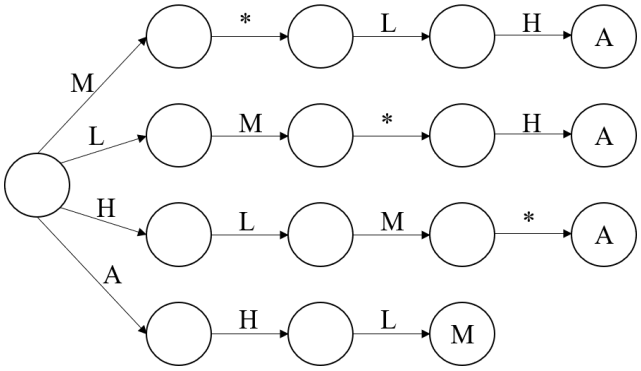
2.2.1 Generovanie ťahov

Appel a Jacobson (1988) predstavili k dátovej štruktúre DAWG aj algoritmus umožňujúci hľadanie všetkých platných ťahov pre Scrabble. Pre všetky prázdne políčka vedľa (horizontálne alebo vertikálne) políčka s kameňom vygeneroval algoritmus rekurzívne najprv všetky možné prefixy (t. j. aspoň jedno slovo v slovníku ho obsahuje) a pre nájdené prefixy vygeneroval všetky možné suffixy. To však znamená, že sa vygeneruje veľmi veľa prefixov, ktoré určite nie je možné využiť - buď lebo sa prefix nedá doplniť použitím zvyšných kameňov v zásobníku alebo ho nie je možné rozšíriť pomocou kameňov na hracej doske. Takéto redundantné generovanie prefixov zaberá určitý čas a samotný algoritmus je nedeterministický, a preto autori štruktúry na zvýšenie efektivity vyhľadávania slov a generovania všetkých platných ťahov prišli s myšlienkou použiť obojsmerný DAWG.

2.3 Štruktúra GADDAG

Vyššie spomenutý obojsmerný DAWG predstavil vo svojej publikácii Gordon (1994) s cieľom čiastočne eliminovať nedeterminizmus algoritmu na generovanie

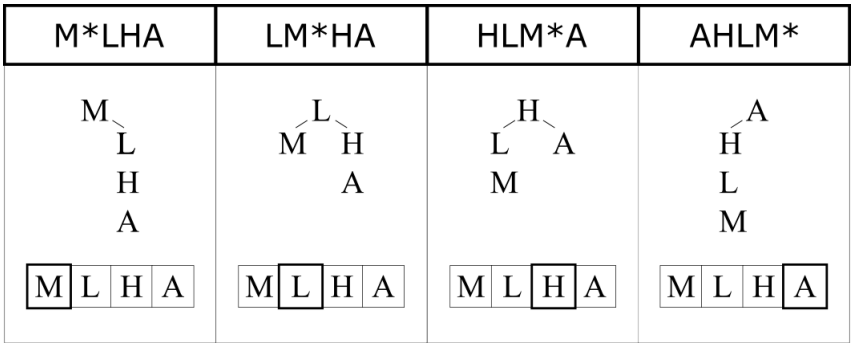
ťahov pre štruktúru DAWG a zvýšiť jeho rýchlosť. Dátovou štruktúrou bude automat pre jazyk $L = \{Rev(x) * y | xy$ je slovo a x je neprázdny reťazec$\}$, kde $Rev(x)$ je reťazec x odzadu a $*$ je oddeľovač (obr. 2.3). Každé slovo slovníka má teda v tejto štruktúre toľko reprezentácií, koľko má písmen. Nech n je priemerná dĺžka slova v slovníku, potom bude GADDAG štruktúra pre daný slovník n-krát väčšia ako DAWG štruktúra pre ten istý slovník. Keďže ide o obrátený DAG spojený oddeľovačom s ďalším DAG, je táto štruktúra pomenovaná GADDAG.



Obr. 2.3: Neminimalizovaná štruktúra GADDAG pre slovo „MLHA“.

Takáto reprezentácia slov má svoj dôvod. Nech sa políčko nazýva „kotva“ vtedy, keď je obsadené nejakým kameňom alebo horizontálne alebo vertikálne susedí s obsadeným políčkom. Potom vieme pridaním kameňov okolo kotvy (v prípade prázdnej kotvy pridaním kameňa aj na kotvu) vytvoriť nové slovo s platným umiestnením.

Kotva použitá na tvorbu nového slova vo všeobecnosti teda môže byť ľubovoľné písmeno tohto nového slova (obr. 2.4). Namiesto toho, aby sme každému takémuto písmenu hľadali všetky možné prefixy algoritmicky, GADDAG umožňuje hľadať prefixy priamo v dátovej štruktúre, čo výrazne zrýchľuje vyhľadávanie slov. Pr veľký slovník americkej angličtiny je GADDAG oproti štruktúre DAWG 5-krát väčší, no priemerne je vyhľadávanie 2-krát rýchlejšie (Gordon, 1994), čo je pri generovaní ťahov v reálnom čase (obzvlášť pri uplatňovaní komplexných heuristík) kľúčové. Preto budeme pre reprezentáciu nášho slovníka používať túto štruktúru.



Obr. 2.4: Všetky možné kotvy v slove „MLHA“. Zvýraznený kameň je kotva.

Automat GADDAG štruktúry je tiež možné minimalizovať. Aby bol automat čo najmenší, pri konštrukcii platia tieto pravidlá:

1. Keď je y v $Rev(x) * y$ prázdne, oddeľovač $*$ sa vynecháva.
2. Stav automatu špecifikuje hrany, ktoré z neho vedú a ich príslušné písmená.
3. Hrana špecifikuje
   (a) stav, do ktorého smeruje;
   (b) písmená, ktoré keď budú nasledovať, vytvoria platné slovo (v obr. 2.3 sú tieto písmená v cieľovom stave hrany).

Množinu písmen hrany, ktoré ukončujú platné slovo, nazvime sadou písmen tejto hrany (*letterset*). Tým, že sa sady písmen viažu k hranám a nie stavom, sa vyhneme tomu, aby boli niektoré stavy konečné. Tento prístup zabezpečuje lepšiu minimalizáciu štruktúry, pretože to umožňuje, aby mala hrana e smerujúca do stavu v neprázdnu sadu písmen a hrana f smerujúca do v prázdnu sadu písmen – nedochádza teda k obmedzovaniu stavu v na konečný, čo by znemožňovalo smerovanie hrany f do tohto stavu.

2.3.1 Generovanie ťahov

Algoritmus pre štruktúru DAWG hľadal prefixy rozdielne od suffixov. Algoritmus pre GADDAG však umožňuje rovnaký prístup k prefixom aj suffixom. Už nájdený podreťazec sa rozširuje postupne po písmenách buď zľava alebo sprava. Podreťazec sa rozširuje (začnúc od kotvy) konkatenáciou po písmenách najprv zľava a keď pri prechádzaní štruktúry narazíme na hranu so znakom $*$, už nájdený podreťazec sa stane prefixom a algoritmus pre túto možnosť bude generovať suffixy konkatenáciou písmen sprava.

Tento algoritmus je stále nedeterministický a počas generovania ťahov naráža aj na slepé uličky. V porovnaní s algoritmom pre DAWG však narazí na menej slepých uličiek a prejde menším množstvom hrán pred tým, ako zistí, že nemá zmysel pokračovať ďalej. Na hracej doske tiež zvykne byť často niekoľko kotiev vedľa seba (napríklad pri rozširovaní slova predponou). V takýchto prípadoch GADDAG algoritmus zabezpečuje, aby sa zbytočne negenerovali tie isté slová pre každú z kotiev zvlášť, čo je ďalšou výhodou oproti algoritmu pre DAWG.

3. Heuristika

V Scrabble víťazí hráč, ktorý má na konci hry najvyšší počet bodov. Aby sa dosiahol najvyšší možný počet bodov na konci hry, nie je vždy najlepše zahrať v ťahu slovo s najvyšším počtom bodov. Profesionálny hráč Scrabble musí myslieť strategicky a zvažovať aj iné veci, napr. či po zahraní ťahu nie je súperovi umožnené využitie prémiových polí na zahranie ťahu za veľký počet bodov alebo ako použiteľné kamene v zásobníku ostanú do ďalšieho kola. Aby sa umelá inteligencia na hranie Scrabble vyrovnala aj profesionálom, je preto nutné pri voľbe ťahu uplatňovať aj určitú heuristiku zohľadňujúcu strategické faktory.

Optimálna stratégia sa počas hry mení. Na začiatku, keď je vo vrecúšku niekoľko kameňov, ide o hru s neúplnou informáciou, keďže nevieme, aké kamene má v zásobníku súper. Takýto stav hry nazvime stredná hra. Naopak, na konci hry, keď už vo vrecúšku žiadne kamene nie sú, vieme určiť obsah zásobníka súpera a pôjde teda už o hru s úplnou informáciou. Tento stav hry nazvime koncovka.

V tejto kapitole si predstavíme heuristiky pre strednú hru a koncovku.

3.1 Stredná hra

Základná vec, podľa ktorej možno hodnotiť ťah, je získaný počet bodov. Ramírez a kol. (2009) navrhuje heuristiku, v ktorej sa berie ohľad aj na to, aké príležitosti ťah poskytne súperovi, a aké príležitosti nám poskytnú zvyšné kamene, ktoré nám po zahraní ťahu ostali v zásobníku. Heuristika pre každý ťah vyzerá nasledovne:

$$v = j + e - d,$$

kde j je počet bodov, ktorý ťahom získame; e je očakávaná hodnota binga; d je defenzívnosť ťahu.

Defenzívnosť ťahu d je nezáporné číslo, pričom:

- $d = 20$, ak ťah umožňuje súperovi potenciálne zahrať ťah cez dve prémiové políčka „3-krát hodnota slova“;
- $d = 10$, ak je súperovi umožnený ťah cez kombináciu políčok „3-krát hodnota slova“ s „2-krát hodnota slova“ alebo „3-krát hodnota písmena“ s „2-krát hodnota slova“;
- $d = 5$, ak ťah umožňuje súperovi zahrať ťah cez dve políčka „2-krát hodnota slova“;
- $d = 0$ v ostatných prípadoch.

Premenná e vyjadruje, s akou pravdepodobnosťou budeme mať v ďalšom ťahu po potiahnutí nových kameňov šancu zahrať 7-písmenové slovo, a teda dostať prémiu 50 bodov – tzv. bingo. Pre objasnenie premennej e si zadefinujme septet ako lexikograficky usporiadanú sedmicu znakov (písmen), z ktorých je možné vyskladať 7-písmenové slovo. Napríklad slovo BROSKEV sa dá vyskladať zo septetu

{B,E,K,O,R,S,V}, podobne aj zo septetu {B,K,O,R,S,V,.}, kde „.“ značí žolíka. Na výpočet hodnoty e sa potom použije vzorec

$$e = \sum_{i=1}^{n} p_i(50 + k\sigma_i),$$

kde n je počet všetkých septetov, ktoré vieme vytvoriť (záleží to teda od počtu 7-písmenových slov v slovníku); p_i je pravdepodobnosť, že budeme mať v ďalšom kole na zásobníku i-tý septet a σ_i je súčet hodnôt písmen i-teho septetu. Hodnotu k autori určili experimentálne a je rovná 2,5.

Na príklade si ukážeme, ako sa počíta pravdepodobnosť p_i. Predpokladajme, že sme so zásobníkom {A,D,I,K,K,O,R} zahrali ťah použitím písmen {A,D,I,K} a na zásobníku nám ostali písmená {K,O,R}. Keďže nevieme, aké písmená má na zásobníku súper, zadefinujme si „rozšírené vrecúško“, ktoré obsahuje kamene na súperovom zásobníku a kamene vo vrecúšku. Obsah rozšíreného vrecúška teda poznáme, pretože ide o obsah plného vrecúška, z ktorého sú zobraté písmená uložené na hracej doske a v našom zásobníku. Nech septet {B,E,K,O,R,S,V} je i-ty septet. Pravdepodobnosť p_i pre zostatok {K,O,R} na našom zásobníku potom bude pravdepodobnosť, že si z rozšíreného vrecúška vytiahneme písmená B,E,S a V. Predpokladajme, že v našom rozšírenom vrecúšku je 85 písmen, z toho dvakrát B, štyrikrát E, raz S a trikrát V. Výsledná pravdepodobnosť p_i potom bude $p_i = C(2,1) \cdot C(4,1) \cdot C(1,1) \cdot C(3,1)/C(85,4) \doteq 1{,}1 \cdot 10^{-5}$, kde $C(m,n)$ je kombinačné číslo.

3.2 Koncovka

V koncovke ide o hru s úplnou informáciou. Na vyhodnotenie najlepšieho ťahu preto môžeme využiť algoritmy prehľadávajúce stromy stavov hry. V Scrabble je však v každom ťahu príliš veľa možností, čo zahrať, a preto by bol strom so všetkými možnými scenármi hry neúnosne veľký. Pre prehľadávanie stavov pre Scrabble je preto vhodné zvoliť usporiadané prehľadávanie (angl. best-first search), ktoré si na prehľadávanie vyberá ten najsľubnejše vyzerajúci uzol.

Berliner (1981) predstavuje tzv. B* algoritmus, ktorý je jedným z algoritmov na usporiadané prehľadávanie stavov. Každý vrchol obsahuje najoptimistickejší a najpesimistickejší odhad toho, ako hra môže skončiť a všetky možné výsledky sa s určitou pravdepodobnosťou nachádzajú v tomto intervale.

Zisťovanie tohto intervalu pre vrchol prebieha tak, že algoritmus preskúma všetkých potomkov do určitej hĺbky a podľa skóre v listoch určí najoptimistickejší a najpesimistickejší scenár. Ak tento interval vrcholu mení interval jeho rodiča, aktualizujú sa všetky intervaly vrcholov od rodiča až po koreň. Prehľadávanie sľubných vrcholov prebieha až kým nevieme s určitosťou povedať, že je jeden z potomkov koreňa jednoznačne najlepšia voľba. Takáto situácia nastane, ak je najpesimistickejší odhad nejakého z potomkov koreňa aspoň taký dobrý ako najoptimistickejšie odhady všetkých ostatných potomkov.

Narozdiel od ostatných algoritmov na usporiadané prehľadávanie, B* algoritmus umožňuje meniť stratégiu prehľadávania počas behu algoritmu. Na výber sú dve možné stratégie:

1. budeme sa snažiť čo najviac zvýšiť najpesimistickejší odhad najsľubnejšieho vrcholu – tzv. PROVEBEST stratégia;

2. budeme sa snažiť čo najviac znížiť najoptimistickejšie odhady ostatných kandidátov – tzv. DISPROVEREST stratégia.

V určitých situáciách sa nájde riešenie rýchlejšie použitím druhej stratégie. Stále, keď sa algoritmus vráti do koreňa (t. j. po aktualizácii intervalov), môže si zvoliť, ktorú stratégiu pri ďalšom prehľadávaní použije. Podľa zvolenej stratégie algoritmus buď navštívi najsľubnejšieho alebo druhého najsľubnejšieho potomka vrcholu, dostane sa k listu a tomuto listu vypočíta jeho najoptimistickejší a najpesimistickejší odhad. Podľa zisteného intervalu potom následne aktualizuje intervaly všetkých predkov až po koreň, kde algoritmus buď vráti jednoznačne najlepší uzol, alebo bude pokračovať v prehľadávaní stavového stromu.

Spôsobov, ako vyberať vhodnú stratégiu, je mnoho. Palay (1982) porovnáva pravdepodobnosť, že bude úspešná prvá stratégia s pravdepodobnosťou, že bude úspešná druhá stratégia na základe vlastností intervalov následníkov koreňa. Berliner a McConnell (1996) sa voľbe stratégií vyhli úplne a vytvorili si algoritmus, ktorý porovnáva pravdepodobnosti, že sa daným uzlom dá dosiahnuť želaná hodnota, ktorá je odvodená z intervalov uzlov. Maven (umelá inteligencia na hranie Scrabble, ktorá je od r. 1995 súčasťou Scrabble CD-ROM spoločnosti Hasbro) na koncovky taktiež používa variant B* algoritmu. Kvôli nepraktickosti pracovania s pravdepodobnosťami však Maven stratégie jednoducho strieda (Sheppard, 2002, str. 46, 107). V tejto práci využijeme prístup Mavenu a budeme stratégie striedať stále, keď sa algoritmus vráti do koreňa.

4. Implementácia

4.1 Úprava pythonovskej knižnice GADDAG

Pôvodná pythonovská GADDAG knižnica (Bass, 2020) podporovala len anglickú abecedu, čomu boli veľkosťou prispôsobené aj dátové typy, t. j. ak sa podporuje 27 znakov (26 písmen anglickej abecedy + 1 špeciálny znak – oddeľovač) a potrebujeme premennú, v ktorej je každý znak reprezentovaný jedným bitom, postačí, ak bude premenná 32-bitový integer. Pre českú (42 písmen) alebo slovenskú (46 písmen) abecedu však takýto dátový typ nepostačuje, a preto sme boli nútení túto knižnicu modifikovať. Ďalšie technické informácie ku knižnici, funkciám, premenným a triedam, ktoré sú v tejto sekcii spomenuté, sú v kapitole 5 Programátorská dokumentácia.

4.1.1 Neúspešný pokus o podporu znakov s diakritikou

Keďže pôvodná knižnica podporovala iba anglickú abecedu, bolo potrebné ju upraviť tak, aby dokázala pracovať so znakmi s diakritkou.

V DLL knižnici je možné rozdeliť využívanie dátového typu `char` na dve kategórie:

1. funkcie zabezpečujúce komunikáciu DLL knižnice s pythonovskými skriptami, čo sú napríklad odpovede na zadané dotazy (`gdg_starts_with(gdg,char prefix)`) alebo pridanie nového slova (`gdg_add_word(gdg,const char *word)`);

2. funkcie volané len vo vnútri DLL knižnice, napr. `gdg_ch_to_idx(char ch)`.

Dátový typ `char` umožňuje používať 256 rôznych znakov. Keďže väčšina z týchto znakov nie je alfabetická, za normálnych okolností sa v slovníku nevyskytnú. Tieto nevyužívané ASCII znaky (napr. číslice) však vieme využiť tak, že na ne určitým spôsobom namapujeme znaky s diakritikou. Ak zabezpečíme, že sa toto mapovanie udeje pred tým, ako sa zavolajú funkcie druhej kategórie, môžeme v nich dátovy typ `char` ponechať. Znaky s diakritikou v týchto funkciách budú vystupovať napríklad ako ASCII znaky „0“, „<“ či „@“.

Prvá kategória funkcií môže dostávať na vstupe slová s diakritikou alebo sú takéto slová požadované na ich výstupe, a preto sme považovali za potrebné, aby tieto funkcie znaky s diakritikou priamo podporovali. Na problémových miestach týchto funkcií sme namiesto reťazcov typu `char *` začali používať reťazce `wchar_t *`, tzv. *wide character*. Tento dátovy typ má veľkosť 2–4 B, takže podporuje aj znaky s diakritikou podľa toho, aký jazyk máme nastavený v `locale`.

Naším plánom bolo nahradiť typom `wchar_t *` reťazce, ktorými používateľ zadáva dotazy alebo na dotazy dostáva odpovede a tieto špeciálne znaky potom nejak namapovať na inak nepoužívané znaky typu `char`. Menovite ide o funkcie `gdg_starts_with(gdg, prefix)`, `gdg_ends_with(gdg, suffix)`, `gdg_contains(gdg, sub)`, `gdg_has(gdg, word)` a výsledky týchto dotazov v štruktúre `Result_Struct`.

Vytvorili sme si konverzné funkcie:

```
const char * unicode_to_ascii(wchar_t * wch);
wchar_t * ascii_to_unicode(const char * ch).
```

Ak bol vo `wchar_t` znak anglickej abecedy, namapoval sa na prislúchajúci `char`, ostatné znaky boli fixne mapované na znaky 48 („0“) až 64 („@“) – podporovalo to teda maximálne 17 špeciálnych znakov.

Dúfajúc, že sa nám túto a iné menšie zmeny (upravené načítavanie zo súboru, ukladanie komprimovanej štruktúry) podarí urobiť jednoduchšie využitím širšej škály dostupných funkcií, zo skriptu v jazyku C sme vytvorili skript jazyka C++.

Zoznam podporovaných špeciálnych znakov abecedy (t. j. znakov s diakritikou) bol udržiavaný v globálnej premennej `wchar_t * special_alphabet`, podľa ktorej bolo umožnené mapovanie znakov `wchar_t` na `char` podľa konkrétnej abecedy. Premennej `const uint32_t GDG_MAX_CHARS` sme odobrali modifikátor `const`, aby ju bolo možné nastaviť podľa aktuálne používanej abecedy.

Nová abeceda sa načítavala z textového súboru, kde v prvom riadku bol názov `locale` a v druhom riadku zoznam špecálnych znakov (malé aj veľké písmená). Podľa obsahu súboru sa potom nastavil správne `locale`, `GDG_MAX_CHARS` a `special_letterset`.

Špeciálne znaky museli podporovať aj premenné v `GADDAG_Struct`, ktorá vyzerala nasledovne:

```
typedef struct GADDAG_Struct GADDAG;
struct GADDAG_Struct {
   uint32_t cap;
   uint32_t num_words;
   uint32_t num_nodes;
   uint32_t num_edges;
   uint32_t *edges;
   uint32_t *letter_sets;
};
```

Premenným `edges` a `letter_sets` sme zmenili typ na `uint64_t *`, aby sa v bitovej reprezentácii čísla v poli zmestili aj špeciálne znaky – s takýmto typom bolo možné podporovať abecedy do dĺžky 63 znakov (+ oddeľovač).

Tento pokus o rozšírenie knižnice zlyhal na type `wchar_t`. Veľkosť tohto dátového typu je definovaná inak na operačnom systéme Windows (fixne $4\,B$) a inak na operačnom systéme Linux (2 alebo $4\,B$). Keďže je naším cieľom vytvoriť platformovo nezávislý program, predstavuje to veľký problém. Taktiež sme problém konverzie špeciálnych znakov na `char` riešili ťažkopádne v zbytočne nízkej úrovni knižnice (typu `wchar_t` sme sa mohli úplne vyhnúť, ak by sme konverzie na `char` robili v pythonovskej časti GADDAG knižnice a nie až v DLL knižnici v jazyku C++).

4.1.2 Funkčná podpora znakov s diakritikou

Keďže sme identifikovali nedostatky prvého pokusu o podporu špeciálnych znakov, mohli sme namiesto toho naimplementovať jednoduchšie a platformovo nezávislé rozšírenie knižnice GADDAG, ktoré umožňuje meniť jazyk jednoducho

aj za behu programu. Prácu so znakmi s diakritikou sme presunuli do pythonovskej triedy GADDAG.

Zmena reprezentácie štruktúry

Štruktúru GADDAG_Struct sme tentoraz upravili trochu inak. Pôvodne bola premenná GDG_MAX_CHARS globálna a samotná štruktúra v takomto stave:

```
const uint32_t GDG_CAP = 100;
const uint32_t GDG_MAX_CHARS = 27;

typedef struct GADDAG_Struct GADDAG;
struct GADDAG_Struct {
   uint32_t cap;
   uint32_t num_words;
   uint32_t num_nodes;
   uint32_t num_edges;
   uint32_t *edges;
   uint32_t *letter_sets;
};
```

Jediná premenná, ktorá v tejto štruktúre priamo ovplyvňuje maximálnu podporovanú veľkosť abecedy, je uint32_t *letter_sets. Jednotlivé bity letter_sets reprezentujú znaky abecedy (+ oddeľovač). Aby štruktúra podporovala aj dlhšie abecedy (do 63 znakov), zmenili sme typ tejto premennej na uint64_t * a podľa toho sme upravili aj funkcie, ktoré s touto premennou pracovali. Pole edges naopak obsahuje čísla vrcholov, do ktorých vedú jednotlivé hrany, a preto nebolo potrebné meniť jeho typ.

Keďže naším cieľom je podporovať rôzne abecedy a umožniť ich zmenu za behu programu, zbavili sme sa konštantnej premennej GDG_MAX_CHARS a použili sme ju ako ďalšiu premennú v GADDAG_Struct, ktorá je v konštruktore pythonovského *wrappera* okolo celej DLL knižnice predaná ako parameter.

Od GDG_MAX_CHARS závisia konverzné funkcie medzi znakom a indexom poľa edges:

```
static int gdg_ch_to_idx(char ch);
static char gdg_idx_to_ch(uint8_t idx);
```

Keďže maximálny počet znakov už nie je konštanta, týmto funkciám bol doplnený ďalší argument – int max_chars, aby sa mohli prispôsobovať veľkosti abecedy.

Konverzia znakov pred volaním DLL knižnice

Pythonovský typ str je sekvencia znakov kódovaná v UNICODE. To pre nás znamená, že Python implicitne podporuje aj znaky s diakritikou; narozdiel od prístupu s typom wchar_t v skripte jazyka C++ nie je nutné ani v závislosti od vstupnej abecedy meniť nastavenie locale. Naším cieľom je teda slovo zakódované v UNICODE prekonvertovať na slovo v ASCII ešte pred tým, ako ho predáme funkciám DLL knižnice.

Aby sme vedeli, aké písmená s diakritikou budú platné v abecede nášho GADDAG-u, naša trieda GADDAG pri inicializovaní načíta abecedu špeciálnych

(s diakritikou) znakov zo súboru, implicitne zo súboru *alphabet.txt*. Napríklad, pre český jazyk súbor obsahuje reťazec *„ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ“*. Podľa obsahu súboru si trieda inicializuje atribúty `special_letterset` a `MAX_CHARS`, podľa ktorej následne vytvorí príslušnú inštanciu štruktúry z DLL knižnice. Funkcie knižnice `int gdg_ch_to_idx(char ch)` a `char gdg_idx_to_ch(uint8_t idx)` budú upravené tak, aby vedeli pracovať s indexami od 0 do `MAX_CHARS`.

Konverzné funkcie, ktoré sme si pôvodne vytvorili priamo v DLL knižnici na kódovanie špeciálnych znakov na ASCII znaky, sme presunuli do triedy `GADDAG` – po novom ide o funkciu `ascii_to_unicode(ascii_word)` a funkciu `unicode_to_ascii(unicode_word)`.

Pre kódovanie z UNICODE do ASCII budú znaky s diakritikou prekonvertované na nealfabetické znaky typu `char`, a znaky bez diakritiky ostanú zachované (ich formát sa len zmení na `char`). Všetky znaky sú *case-insensitive*. Premenná `char ch = 'A'` má číselnú hodnotu 65. Nech je počet špeciálnych znakov našej abecedy n ($0 < n < 38$), potom budú tieto znaky príslušnou funkciou zakódované do ASCII znakov s hodnotami od $(65 - n)$ do 64. Česká abeceda špeciálnych znakov „ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ“ je teda prekonvertovaná na znaky *„23456789:;<=>?@“* a slovo *„AKČNĚJŠÍ“* typu UNICODE bude prekonvertované na slovo *„AK3N6J;7“* typu ASCII.

Textové reťazce, na ktoré sa budeme pýtať v dotazoch na našu štruktúru, budú v príslušných funkciách triedy `GADDAG` najprv zmenené na *case-insensitive*, potom prekonvertované našou konverznou funkciou do ASCII a následne zadané ako argument príslušnej funkcie DLL knižnice. Naopak, slová, ktoré sú odpoveďou na nami zadaný dotaz, budú najprv vybrané zo štruktúry `Result *`, prekonvertované z ASCII do ich UNICODE verzie, pridané do typu `List` a nakoniec volajúcou funkciou vrátené.

4.1.3 Pridané funkcie

DLL knižnica už podporuje aj prácu s dlhšími abecedami, a preto už spĺňa jej základnú úlohu. Ďalšie funkcie potrebné na generovanie ťahov a výpočty pre heuristiku sme pridali ako metódy do našej pythonovskej triedy `GADDAG`.

Na generovanie platných ťahov (viď sekcia 4.3) a hľadanie *lettersetov* pre *bezpísmenové kotvy* sme pridali metódy, ktoré pre zadaný podreťazec nájdu všetky písmená, ktorými je možné ho rozšíriť (zľava alebo sprava) tak, aby stále išlo o podreťazec nejakého slova v slovníku. Ide o funkcie `left_letterset(subword)` a `right_letterset(subword)`. Práca s týmito funkciami o. i. zrýchľuje algoritmus na generovanie ťahov, pretože sa v každej iterácii zaoberá len písmenami, ktorými je naozaj možné rozšíriť zatiaľ nájdený podreťazec.

Na vypočítanie heuristiky pre strednú hru potrebujeme získať zo slovníka zoznam všetkých unikátnych *septetov*, t. j. zoznam všetkých neusporiadaných sedmíc písmen, ktorých permutáciou je možné získať platné slovo v slovníku. Pre tento účel sme do triedy `GADDAG` pridali metódu `find_septets()`, ktorá rekurzívne prejde našou GADDAG štruktúrou a vráti zoznam všetkých septetov.

4.2 Vytvorenie slovníka

Na vytvorenie GADDAG štruktúry s platnými slovami je najprv potrebné mať slovník. Ten je definovaný vo formáte textového dokumentu obsahujúceho všetky slová, ktoré budú považované za prípustné, t. j. pôjde len o podmnožinu všetkých prípustných slov daného jazyka. V našej práci sme vytvorili GADDAG štruktúru pre český slovník, v ktorom je 326 054 slov kratších ako 10 písmen. Použitý slovník je voľne dostupný na webovej stránke autora Toal a kol.

Vytvorená bola aj GADDAG štruktúra pre slovenský slovník so 100 000 slovami kratšími ako 10 písmen. V GUI programu je teda možné hrať aj slovenskú verziu Scrabble. Tento slovník obsahuje veľa skratiek, ktoré v hre Scrabble nie sú povolené. Keďže slovenský slovník slúži len na ukážku jazykovej flexibility programu, nepovažujeme to za problém.

4.3 Generovanie ťahov

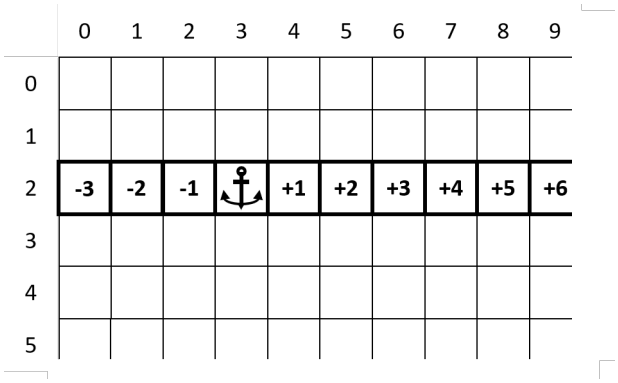
Gordon (1994) okrem GADDAG štruktúry predstavil aj algoritmus, ktorý pomocou nej generuje všetky platné ťahy v danej pozícii. Algoritmus sa skladá z dvoch rekurzívne volaných funkcií, ktoré sme si trochu upravili (viď 4.5):

```
Generuj(pozicia, zasobnik, slovo):
  Ak je na tomto políčku písmeno L:
    Spracuj(pozicia, zasobnik, L, slovo)
  Inak ak na zásobníku ostali nejaké písmená:
    Pre každé písmeno L na zásobníku, ktoré je na políčku povolené:
      Spracuj(pozicia, zasobnik - L, L, slovo)
    Ak zásobník obsahuje žolíka:
      Pre každé písmeno L, na ktoré sa dá zmeniť žolík:
        Spracuj(pozicia, zasobnik - žolík, L, slovo)

Spracuj(pozicia, zasobnik, L, slovo):
  Ak pozicia <= 0:      // slovo rozširujeme smerom doľava
    slovo := L + slovo
    Ak je slovo platné a bezprostredne vľavo nie je žiadne písmeno:
      Zaznamenaj(slovo)
    Ak existuje políčko naľavo:
      Generuj(pozicia-1,zasobnik,slovo)
    Ak existuje políčko napravo:
      Generuj(1, zasobnik, slovo)
  Inak ak pozicia > 0:    // slovo rozširujeme smerom doprava
    slovo := slovo + L
    Ak je slovo platné a bezprostredne vpravo nie je žiadne písmeno:
      Zaznamenaj(slovo)
    Ak existuje políčko napravo:
      Generuj(pozicia+1,zasobnik,slovo)
```

Dôležitou vlastnosťou algoritmu je to, že slová generuje v 1D priestore namiesto 2D priestoru hracej dosky. Postupovať preto budeme tak, že sa pred generovaním ťahov rozhodneme, či chceme hľadať ťahy v horizontálnom alebo vertikálnom smere. Relatívna súradnica políčka `pozicia` má hodnotu 0 na políčku, z ktorého začíname generovať slová, má záporné hodnoty pre políčka naľavo, resp. nad začiatočným políčkom a kladné hodnoty pre políčka napravo, resp. pod začiatočným políčkom. Na obr. 4.1 je možné vidieť relatívne súradnice pre generovanie

slov v horizontálnom smere so začínajúcim políčkom (kotvou) so súradnicami [2,3]. Na generovanie ťahov v danom smere a s danou kotvou slúžia inštancie triedy `Validator`, ktoré umožňujú hľadanie slov v relatívnych 1D súradniciach. Pre každú kotvu sa podľa jej pozície a orientácie vytvorí vlastná inštancia triedy `Validator`.



Obr. 4.1: Relatívne súradnice políčok pre kotvu na pozícii [2,3] v horizontálnom smere.

`Generuj(pozicia,zasobnik,slovo)` rozhodne, aké písmeno ponúkne na rozšírenie už nájdeného podreťazca funkcii `Spracuj(...s)` podľa obsahu zásobníka a podľa toho, či už je na políčku položený nejaký kameň. Naša implementácia tejto funkcie sa volá `generate_opts(int pos,Letter[] rack,str word)` v triede `PlayerAI`. Generovanie ťahov sa po vytvorení požadovanej inštancie `Validator` zavolá ako `generate_opts(0,zasobnik,"")`, kde `zasobnik` je zásobník hráča na ťahu.

Funkcia `Spracuj(pozicia,zasobnik,L,slovo)` pridáva písmeno `L` k slovu `slovo` zľava alebo sprava podľa toho, v akom smere sa pohybujeme. Najprv sa pridáva písmeno stále zľava. Keď `pozicia` nadobudne hodnotu 1, smer sa zmení a písmená sa budú pridávať sprava. Stále sa teda najprv vygeneruje prefix a k nemu sa následne generujú možné suffixy. V našej implementácii ide o funkciu `process_opts(int pos,Letter[] rack,str L,str word)` v triede `PlayerAI`.

Ak vznikne po pridaní písmena platné slovo, zaznamená sa funkciou `Spracuj(slovo)`. Túto úlohu má v implementácii na starosti funkcia `record_word(str word,int pos,Letter[] rack,int state)` triedy `Validator`, kde argument `state` umožňuje rozlišovanie medzi strednou hrou a koncovkou, ktorá na zaznamenanie samotného slova nemá žiadny vplyv a využíva sa na prípadné počítanie časti heuristiky pre strednú hru.

Funkcia `record_word(...)` pred zaznamenaním slova ešte skontroluje:

- či bezprostredne pred alebo za slovom neleží nejaký kameň (ktorý do slova nebol zarátaný);
- či položením kameňov na hraciu dosku nevzniknú aj slová v smere kolmom na zaznamenávané slovo, a ak áno, či sú tieto slová platné.

Ak sú všetky novovzniknuté slová v smere kolmom na zaznamenávané slovo platné, slovo je možné zaznamenať ako platný ťah – vytvorí sa inštancia triedy `Move`, ktorá o. i. obsahuje zoznam všetkých novovzniknutých slov.

Všetky nájdené platné ťahy sa zaznamenávajú do atribútu `found_moves` triedy `Validator` a pred vytvorením novej inštancie triedy `Validator` sa pridajú do zoznamu všetkých nájdených ťahov v triede `PlayerAI`.

4.4 Heuristika

4.4.1 Stredná hra

Vzorec na výpočet heuristiky pre strednú hru je

$$v = j + e - d,$$

kde j je počet bodov za ťah; e je očakávaná hodnota binga a d je defenzívnosť ťahu (viac v kapitole 3).

Hodnota j je počítaná už pri generovaní ťahov funkciou `generate_moves()` v triede `PlayerAI`.

Na výpočet defenzívnosti d je nutná kontrola okolia slova, ktoré sme našli, a preto defenzívnosť počíta pri zaznamenávaní ťahu trieda `Validator`, ktorá je na to prispôsobená. Platnosť ťahov a ich následné zaznamenávanie má na starosti funkcia `Validator.record_word(...,bool state)`. Ak sa ťah našiel počas strednej hry, t. j. `state = 1`, pred pridaním do zoznamu nájdených ťahov sa mu ešte vypočíta jeho defenzívnosť. Pre každý riadok/stĺpec kolmý na hlavné slovo sa prejdú všetky prázdne políčka (pred prvým obsadeným políčkom) do maximálnej vzdialenosti 7 a vypočíta sa maximálny možný multiplikátor. Políčko „3-krát hodnota slova“ má hodnotu multiplikátoru 3, políčko „2-krát hodnota slova“ má hodnotu 2, políčko „3-krát hodnota písmena“ má hodnotu 2,5 a ostatné políčka majú hodnotu 1. Maximálny multiplikátor sa počíta ako súčin hodnôt skúmaných políčok osobitne pre každý stĺpec/riadok. Statický atribút `Move.defensiveness_dict` je slovník, ktorý potom namapuje zistené hodnoty multiplikátorov na hodnoty parametru d pre našu heuristiku (pravidlá pre mapovanie sú spomenuté v kap. 3).

Na získanie hodnoty e potrebujeme mať najprv zoznam *septetov*. Keďže je množina septetov závislá iba na používanom slovníku v štruktúre `GADDAG`, táto množina sa vygeneruje pri načítavaní slovníka funkciou `GADDAG.load(str path)`, resp. `load(str path)` definovaných v skripte *gaddag.py*.

Okrem septetov na výpočet ešte potrebujeme zvyšné kamene, ktoré po zahraní ťahu zostanú na zásobníku. Zoznam týchto kameňov je jedným z atribútov inštancie triedy `Move`. Veľa zostatkových zásobníkov sa pri generovaní všetkých platných ťahov bude opakovať, preto nebudeme počítať hodnotu e pre každý ťah zvlášť, ale vytvoríme si slovník všetkých unikátnych zostatkových zásobníkov z množiny všetkých nájdených ťahov. Nový zásobník sa ako kľúč do slovníka `PlayerAI.remaining_racks_dict` pridáva počas nájdenia nového ťahu vo funkcii `PlayerAI.process_opts(...)`. Vo funkcii `find_best_middlegame_move()`, ktorá pracuje s vygenerovanými ťahmi, sa zavolá funkcia pre heuristiku `compute_next_bonus_probability()`, ktorá každému kľúču slovníka vypočíta hodnotu e (obe funkcie sú metódy triedy `PlayerAI`). Následne sa podľa hodnôt v slovníku priradí každému ťahu `Move` príslušná hodnota e podľa jeho zostatkového zásobníka.

Po vypočítaní hodnoty e pre všetky ťahy už máme vypočítané hodnoty všetkých premenných v heuristike, po zavolaní `Move.calculate_final_points()` na každý ťah vo funkcii `find_best_middlegame_move()` bude ťahu vypočítaná výsledná hodnota v, podľa ktorej sa zoradí zoznam ťahov podľa v.

4.4.2 Koncovka

Ná nájdenie najlepšieho ťahu v koncovke použijeme B* algoritmus. Ide o prehľadávanie stromu možných pozícií hry. Na reprezentáciu uzla stromu máme triedu `PlayerAI.Node`, ktorá reprezentuje stav hry, skóre získané po ceste od koreňa a interval, ktorého hranice tvoria najoptimistickejší a najpesimistickejší odhad toho, koľko bodov je možné získať. Hrany medzi uzlami budú ohodnotené počtom bodov za ťah, ktorým sa z jedného uzlu dostaneme do druhého.

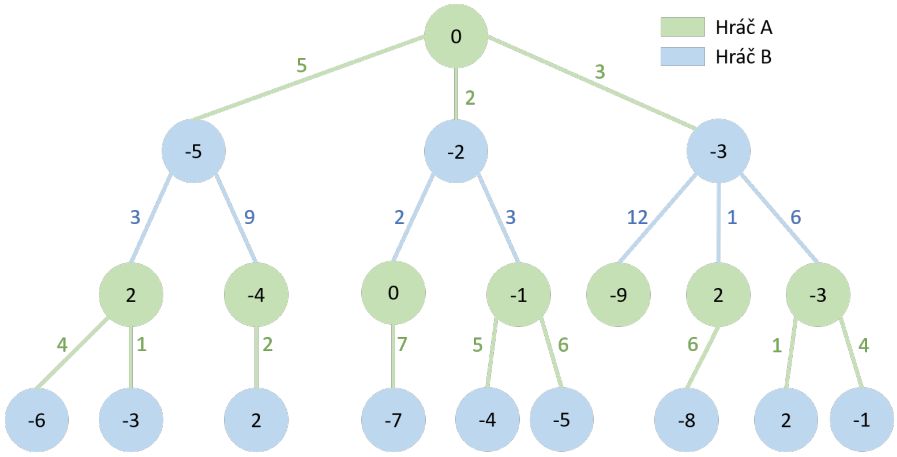
Využijeme algoritmus, ktorý predstavil Palay (1982). Algoritmus je implementovaný ako tzv. *negamax*, t. j. hodnota pozície pre jedného hráča je jej negáciou pre druhého hráča (obr. 4.2). Algoritmus vyzerá nasledovne:

```
Bstar-z-korena(start_poz):
  nájdi a vyhodnoť potomkov start_poz
  naj_uzol := najlepší uzol z potomkov
  Pokiaľ pesimistický odhad naj_uzol < max. optimistický odhad iných potomkov:
    zvoľ si stratégiu, buď PROVEBEST alebo DISPROVEREST
    aktualny_uzol := ďalší prehľadávaný uzol zvolený podľa stratégie
    Bstar-z-uzlu(aktualny_uzol)
  Vráť naj_uzol

Bstar-z-uzlu(aktualny_uzol):
  Ak aktualny_uzol nemá nájdených potomkov:
    nájdi a vyhodnoť potomkov pre aktualny_uzol
  nova_optim := -(najnižšia pesimistická hodnota medzi potomkami)
  nova_pesim := -(najvyššia optimistická hodnota medzi potomkami)
  Pokiaľ nova_optim == optimistický odhad v aktualny_uzol a
   nova_pessim == pesimistický odhad v aktualny_uzol:
    dalsi_uzol := ďalší prehľadávaný uzol zvolený podľa stratégie
    Bstar-z-uzlu(dalsi_uzol)
    nova_optim := -(najnižší pesimistický odhad v potomkoch aktualny_uzol)
    nova_pesim := -(najvyšší optimistický odhad v potomkoch aktualny_uzol)
  optimistický odhad v aktualny_uzol := nova_optim
  pesimistický odhad v aktualny_uzol := nova_pessim
```

Potomkovia vrcholu stromu sú pozície, do ktorých sa hra dostane zahraním jednotlivých možných ťahov hráča na ťahu. Prvá funkcia začína od koreňa, t. j. aktuálnej pozície. Kým nie je nájdený jednoznačne najlepší ťah, t. j. pesimistický odhad najlepšieho uzlu je aspoň taký dobrý ako optimistické odhady ostatných uzlov, funkcia v každej iterácii cyklu volá `Bstar-z-uzlu(aktualny_uzol)`. Túto funkciu máme implementovanú ako `b_star_top_level(Node starting_node)` v triede `PlayerAI`. Aby sa funkcia náhodou nezacyklila do nekonečna, v našej implementácii tejto funkcie máme nastavený maximálny počet iterácií cyklu. Ak z cyklu vyjdeme kvôli dosiahnutiu maximálneho počtu iterácií, za najlepší ťah sa zvolí ten, ktorý má najlepší optimistický odhad.

Na nájdenie a vyhodnocovanie potomkov vrcholu slúži funkcia `evaluate_successors(Node parent)`. Pre každý možnný ťah funkcia nasimuluje jeho zahranie a zistí, aké ťahy v novej pozícii môže zahrať súper. Podľa



Obr. 4.2: Príklad stromu pre negamax.

bodov, ktoré bude mať hráč na ťahu po odohratí jeho ťahu a ťahu súpera, sa zo všetkých možností vyberie najvyšší a najnižší možný dosiahnutý počet bodov, ktoré hráč môže získať – ide o najoptimistickejší a najpesimistickejší odhad výsledku, ktorý sa zaznamená do uzlu `parent`.

Funkcia `Bstar-z-uzlu(aktualny_uzol)` má za úlohu aktualizovať najoptimistickejší a najpesimistickejší odhad uzlu `aktualny_uzol` a rekurzívne sa volá na potomkov vrcholu, až kým nebude možné tento interval aktualizovať. Ak je možné interval aktualizovať, vynáraním z rekurzie sa aktualizujú intervaly rodičov vrcholu, až kým sa nedostaneme ku koreňu. Táto funkcia je implementovaná ako `b_star_lower_level(Node cur_node, bool provebest_strategy)`, kde sa volí ďalší skúmaný vrchol podľa stratégie zvolenej pri poslednej návšteve koreňa.

4.4.3 Simulácie

Pre overenie toho, že s použitím heuristiky umelá inteligencia dosiahne vyššie celkové skóre v porovnaní s hraním ťahov za najvyšší počet bodov, sme simulovali zápasy v hre Scrabble medzi dvomi umelými inteligenciami.

Umelá inteligencia mala 3 možné úrovne:

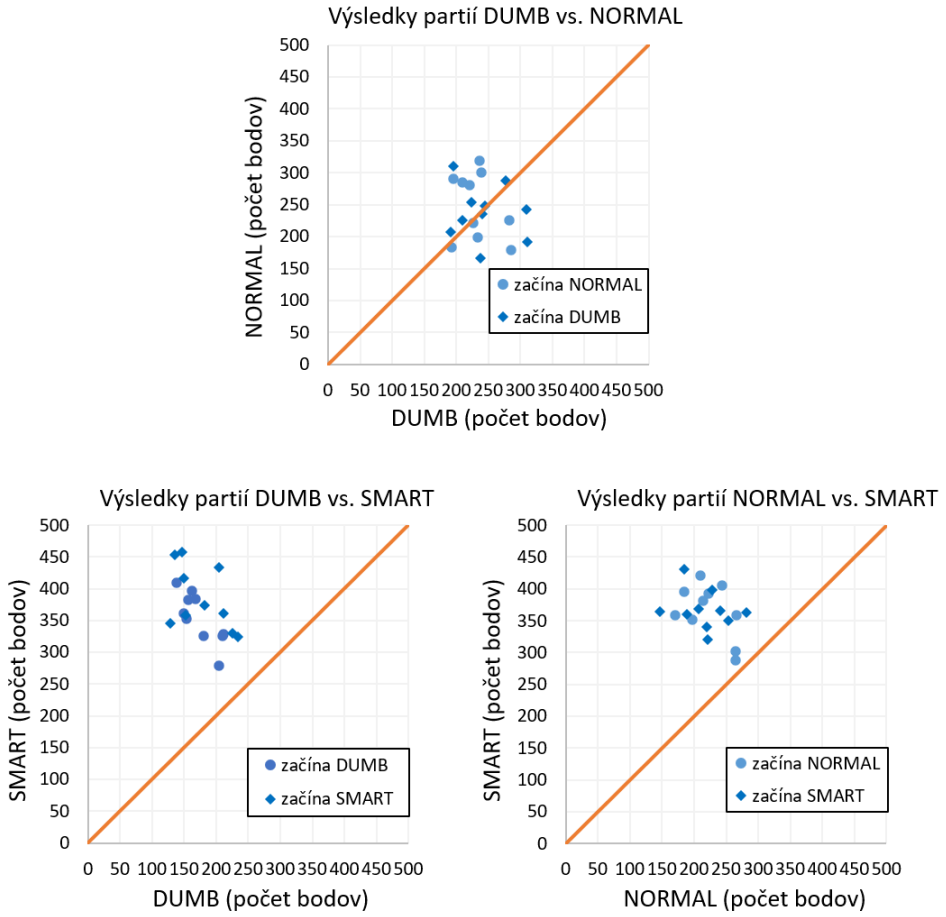
- „DUMB“ – zahrá náhodný platný ťah,
- „NORMAL“ – zahrá ťah za najvyšší počet bodov,
- „SMART“ – zahrá najlepší ťah podľa heuristiky.

Simulácie celých partií boli robené pre tieto dvojice (prvá menovaná úroveň začína):

- DUMB proti DUMB,
- DUMB proti NORMAL a NORMAL proti DUMB,
- DUMB proti SMART a SMART proti DUMB,
- NORMAL proti SMART a SMART proti NORMAL.

Obsah zásobníkov na začiatku hry bol pre každú simuláciu rovnaký, t. j. bez ohľadu na nastavenú inteligenciu mal začínajúci hráč na začiatku hry stále rovnaké kamene, podobne aj nezačínajúci hráč.

Pre každú dvojicu inteligencií bolo simulovaných 10 zápasov. Výsledky simulácií celých partií sú zhrnuté v tabuľke 4.3.

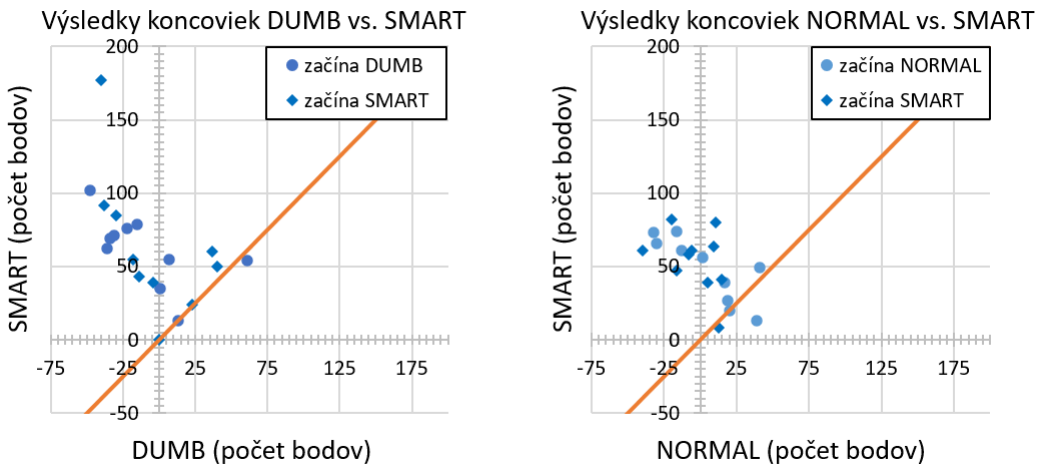


Obr. 4.3: Výsledky zápasov medzi jednotlivými úrovňami umelej inteligencie.

Z výsledkov simulácií vidíme, že umelá inteligencia, ktorá používa našu heuristiku, stabilne vykazuje výborné výsledky v partiách proti hráčom hrajúcim bez nej.

Porovnávanie výkonov hráčov len v koncovke je už náročnejšie. Prvá možnosť je, že zo záznamov simulácií pre celú hru si vyberieme nejakú pozíciu, od ktorej začína koncovka a počnúc od tejto pozície by sme sledovali, ako bude vplývať inteligencia hráčov na počet bodov získaný v koncovke. Tento prístup sme však nevyužili, pretože stratégia SMART hráča priamo ovplyvňuje obsah jeho zásobníka v koncovke.

SMART hráč sa vyhýba ťahom, ktoré by mu v zásobníku nechali ťažko kombinovateľné kamene, ale hráči DUMB a NORMAL pri výbere ťahu neberú ohľad na to, aké kamene im v zásobníku po zahraní ťahu zostanú. Dať v koncovke NORMAL hráčovi zásobník z hry rozohratej SMART hráčom by dávalo NORMAL hráčovi výhodu, ktorú by za normálnych okolností nemal. Koncovkové pozície z hier NORMAL hráča proti DUMB hráčovi zase zanechávali veľmi nevyrovnané obsahy zásobníkov súperov, a preto sme tieto pozície tiež nepovažovali za vhodné.



Obr. 4.4: Výsledky koncoviek medzi jednotlivými úrovňami umelej inteligencie. Výsledky sú označené podľa toho, ktorý z hráčov začínal zápas.

Rozhodli sme sa preto získať dáta z koncoviek priamo zo simulácií celých hier: sledovali sme skóre, aké hráči získajú od začiatku koncovky po koniec hry. Keďže je v tomto momente zaujímavý iba výkon SMART hráča proti súperom, simulácie hier NORMAL hráča proti DUMB hráčovi sme vynechali. Výsledky je možné vidieť na obr. 4.4. Z grafov je možné vyčítať, že výsledky sú v drvivej väčšine prípadov opäť v prospech heuristiky.

V našom programe tieto heuristiky nevyužívame len na výber najlepšieho ťahu, ale aj na usporiadanie možných ťahov podľa toho, aké sú dobré. Heuristika pre strednú hru vie zoradiť podľa tohto kritéria všetky ťahy, no heuristika pre koncovku vie identifikovať len potenciálne najlepší ťah. V koncovke bude program za najlepší ťah považovať ten, ktorý vybral algoritmus na koncovkovú heuristiku, ostatné ťahy budú usporiadané podľa počtu bodov získaných za ich zahratie.

Vychádzajúc z výsledkov simulácií považujeme implementované heuristiky za funkčné a predpokladáme, že ich využitím náš program ponúka dobrý nástroj na analýzu ťahov aj skúsenejším hráčom Scrabble.

4.5 Možné vylepšenia

Nízkoúrovňové generovanie ťahov

Pri generovaní ťahov funkciou `generate_opts(...)` zisťujeme množinu písmen, ktorými je možné podreťazec rozšíriť, pomocou funkcií triedy `GADDAG` `right_letterset(str subword)` a `left_letterset(str subword)`. Pôvodný algoritmus na generovanie ťahov (Gordon, 1994) je viac nízkoúrovňový: argumentmi funkcií sú aj vrcholy GADDAG štruktúry, v ktorých sa momentálne nachádzame a namiesto zisťovania množiny rozširujúcich písmen zisťujeme, či existujú z vrcholu hrany pre jednotlivé písmená v zásobníku. Zmenením našich generovacích funkcií na viac nízkoúrovňové po vzore pôvodného algoritmu by sme program zrýchlili.

Pridanie heuristiky na tzv. predkoncovku

Program MAVEN (Sheppard, 2002) používa navyše inú stratégiu aj na tzv. *predkoncovku* – stav hry, keď ostáva vo vrecúšku už len málo kameňov. Tesne pred koncovkou je ideálne voliť si ťahy tak, aby sme v koncovke začínali ako prví, t. j. po zahraní ťahu si chceme doťahovať tak, aby si posledné kamene vo vrecúšku vytiahol súper. Zvyšuje sa tým šanca, že sa v koncovke zbavíme kameňov skôr ako súper a tým získame body navyše. Pridaním stratégie pre predkoncovku po vzore MAVENu by náš program mohol dosiahnuť ešte lepšie výsledky.

Optimalizácia zálohovania stavu hry

Zálohovanie stavu hry `Game` je kľúčové pre algoritmus používaný na koncovkovú heuristiku. Aby sme sa vedeli pohybovať spätne stavovým stromom, musíme sa vedieť vrátiť do stavu hry pred potiahnutím posledného ťahu. Momentálne náš program zálohuje stavy hry kopírovaním celých inštancií triedy `Game`. Keďže sa zálohovanie volá často, zaberá to netriviálny čas. Tento čas je možné zrýchliť, ak by sme stav hry zálohovali inkrementálne, t. j. ak by sme si pamätali posledný zahraný ťah a podľa tohto údaju zmenili stav hry v rámci jednej inštancie triedy `Game`.

Zlepšenie heuristiky pre koncovku

V našej heuristike pre koncovku sme používali B* algoritmus, v ktorom je možné striedať prehľadávacie stratégie. Využili sme prístup MAVENu a stratégie jednoducho striedali. Na voľbu stratégie by sme mohli v budúcnosti použiť algoritmy založené na počítaní pravdepodobností (Palay, 1982; Berliner a McConnell, 1996), pomocou ktorých by sme vedeli nájsť najlepší ťah rýchlejšie.

Pri počítaní intervalov vrcholov sme prehľadávali hodnoty potomkov len do druhej vrstvy. Čím viac vrstiev by sme pri určovaní intervalov preskúmali, tým presnejšie by sme určili jeho krajné hodnoty. Mohlo by nás to však stáť aj viac času, preto by bolo vhodné určiť ideálnu maximálnu hĺbku prehľadávania.

Komprimované ukladanie slovníka

Načítavanie slovníka z textového súboru do binárneho súboru s príponou *.gdg* je možné využiť, ak chce používateľ v do nášho programu pridať nový jazyk. Ukladanie binárneho súboru je momentálne možné bez kompresie a hotový ZIP súbor, ktorý je možné v GUI načítať, je nutné vytvoriť ručne. Umožniť automatické vytvorenie ZIP súboru vo formáte podporovanom GUI by ešte viac zjednodušilo prácu s programom.

5. Programátorská dokumentácia

Program je písaný v jazykoch Python a C/C++.

Na reprezentáciu slovníka sa využíva GADDAG knižnica pre jazyk Python (Bass, 2020) verejne dostupná na PyPI (Python Package Index), ktorá bola rozšírená tak, aby podporovala aj písmená s diakritikou a vyhovovala potrebám nášho programu. Použitá štruktúra na uloženie slovníka je *case-insensitive*, t. j. nerozlišuje sa medzi malými a veľkými písmenami uložených slov.

V prvej podkapitole si ukážeme základné štruktúry a funkcie upravenej GADDAG knižnice a v ostatných podkapitolách sa budeme venovať zvyšku programu na analýzu ťahov Scrabble, ktoré upravenú knižnicu používajú.

5.1 GADDAG knižnica

Základom GADDAG knižnice je DLL knižnica písaná v jazyku C++, okolo ktorej je v Pythone vytvorený tzv. *wrapper*, ku ktorému sme pridali pár dodatočných funkcií. Celá knižnica je uložená v priečinku *working_pygaddag*.

5.1.1 DLL knižnica v jazyku C++

DLL knižnica písaná v jazyku C++, ktorá tvorí jadro celej GADDAG knižnice, sa nachádza v priečinku *working_pygaddag/src/cGADDAG*.

Štruktúra GADDAG_Struct

Základným prvkom DLL knižnice je štruktúra `GADDAG_Struct`:

```
typedef struct GADDAG_Struct GADDAG;
struct GADDAG_Struct {
       uint32_t MAX_CHARS;
       uint32_t cap;
       uint32_t num_words;
       uint32_t num_nodes;
       uint32_t num_edges;
       uint32_t* edges;
       uint64_t* letter_sets;
   };

static int gdg_ch_to_idx(char ch);
static char gdg_idx_to_ch(uint8_t idx);
```

Premenná `MAX_CHARS` udáva, koľko rôznych znakov štruktúra rozlišuje (vrátane oddeľovača). Premenné `num_words`, `num_nodes` a `num_edges` v sebe majú uložený počet slov, vrcholov a hrán v štruktúre.

Premenné `edges` a `lettersets` sú polia. Keďže sa v nich potrebujeme určitým spôsobom orientovať podľa písmena, ktoré momentálne skúmame, máme pomocnú funkciu `int ch_to_idx(ch, int max_chars)`, ktorá nám vráti číslo

reprezentujúce v štruktúre dané písmeno – index. Do druhého parametra sa predáva hodnota `MAX_CHARS`. Špeciálny znak (oddeľovač) má index 0, ASCII znaky majú indexy od 1 do 26 v abecednom poradí a špeciálne znaky majú indexy od 27 do `MAX_CHARS`-1. Funkcia `char idx_to_ch(idx)` naopak vráti podľa indexu písmena samotné písmeno. Oddeľovač je reprezentovaný znakom „+“.

Vrcholy v štruktúre sú číslované od 1 (v poradí, v akom postupne vznikli). Pre všetky písmená c, ktoré vo vrchole v ukončujú slovo $w.c$, má bit integeru `letter_sets[v]` na pozícii `char_to_idx(c)` hodnotu 1, na všetkých ostatných 0. Usporiadanie bitov je LSB. Ak je v `letter_sets[7]` napr. číslo 17 181 966 624 (prvých 36 bitov v LSB je $0000\,0100\,1000\,0000\,0000\,0100\,0000\,0000\,0010_2$), znamená to, že vo vrchole 7 končia nejaké slová písmenami E, H, U a Ó (jeho ASCII znakom je „9“, viď konverzné funkcie v 5.1.2).

Informácie o hranách medzi vrcholmi sú uložené v dvojrozmernom poli `edges`. Každému vrcholu prináleží jeden riadok dĺžky `GDG_MAX_CHARS`. Číslo uložené v i -tom stĺpci v-tého riadku je číslo vrcholu, do ktorého vedie z vrcholu v hrana s písmenom `c = idx_to_ch(i)` . Ak takáto hrana neexistuje, na pozícii $edges_{vi}$ je 0.

V jazyku Python by sme sa k prvku poľa $edges_{vi}$ dostali cez `edges[v][i]`, v jazyku C++ sa k tomuto prvku dostaneme cez `edges[v*GDG_MAX_CHARS+i]`.

Premenná `cap` určuje počet vrcholov, pre ktoré sa na začiatku miesto v poli `edges` alokuje. Stále, ak bude potrebné spravovať viac vrcholov ako `cap`, funkcia `bool gdg_grow(GADDAG *gdg, uint32_t new_cap)` sa pokúsi pre štruktúru naalokovať viac miesta. Funkcia vráti `true`, ak realokácia prebehla úspešne, inak vráti `false`.

Vytvorenie, odstránenie a rozširovanie štruktúry GADDAG_Struct

```
GADDAG* gdg_create(void);
void gdg_destroy(GADDAG *gdg);
int gdg_add_word(GADDAG *gdg, const char *word);
```

Funkcia `gdg_create()` vytvorí prázdnu štruktúru GADDAG a funkcia `gdg_destroy(gdg)` ju bezpečne odstráni (t. j. uvoľní po nej aj miesto v pamäti alokované touto štruktúrou).

Pomocou funkcie `gdg_add_word(gdg, word)` sa do štruktúry pridá slovo *word*, t. j. do GADDAG štruktúry sa pridajú všetky varianty ukotvenia slova *word*. Napríklad, funkcia `gdg_add_word(gdg, "PERO")` pridá do `gdg` reťazce „P+ERO“, „EP+RO“, „REP+O“ a „OREP“. Ak pridanie slova prebehlo úspešne, funkcia vráti 0. Ak funkcia vráti 1, nastala chyba, lebo sa pokúšame pridať slovo obsahujúce nepodporovaný znak. Ak vráti 2, slovo nebolo možné pridať kvôli nedostatku pamäte.

Načítavanie a ukladanie štruktúry GADDAG_Struct

```
off_t gdg_save(const GADDAG *gdg, const char *path);
GADDAG* gdg_load(const char *path);
```

Štruktúru GADDAG je možné do programu načítať z binárneho súboru pomocou funkcie `gdg_load(path)` a ukladať pomocou funkcie `gdg_save(gdg, path)`. Pre český slovník s 326 054 slovami bola veľkosť binárneho súboru štruktúry 149 MB.

Dotazy

K obsahu slovníka je možné dostať sa zavolaním nejakej z dotazovacích funkcií. Štruktúra, ktorá vracia odpoveď na dotaz, je `Result_Struct`.

```
typedef struct Result_Struct Result;

struct Result_Struct {
   const char* str;
   Result *next;
   Result *prev;
};

bool gdg_has(const GADDAG *gdg, const char *word);
Result* gdg_starts_with(const GADDAG *gdg, const char *prefix);
Result* gdg_contains(const GADDAG *gdg, const char *sub);
Result* gdg_ends_with(const GADDAG *gdg, const char *suffix);
```

Funkcia `gdg_has(gdg, word)` vráti `true`, ak štruktúra `gdg` obsahuje slovo *word*, inak vráti `false`.

Štruktúra `Result_Struct` má formu spojového zoznamu obsahujúceho všetky slová, ktoré spĺňajú kritérium zadané ako parameter dotazovacej funkcie. Neskôr wrapper knižnice v jazyku Python spracuje tento spojový zoznam na klasický pythonovský `List`.

Nech naša GADDAG štruktúra `gdg` obsahuje slová RAJA, JAR, JAHODA, POHODA a JABLKO.

Funkcia `gdg_starts_with(gdg, prefix)` vráti všetky slová začínajúce reťazcom *prefix*, t. j. `gdg_starts_with(gdg, "JA")` nám vráti slová JAR a JABLKO.

Funkcia `gdg_ends_with(gdg, suffix)` vráti všetky slová, ktoré sa končia na *suffix*, t. j. funkcia `gdg_ends_with(gdg, "ODA")` vráti slová JAHODA a POHODA.

Funkcia `gdg_contains(gdg, sub)` vracia všetky slová, ktoré obsahujú *sub* ako podreťazec, t. j. funkcia `gdg_contains(gdg,"A")` vráti všetky slová, ktoré `gdg` obsahuje.

5.1.2 Pythonovská časť knižnice

Pythonovská časť knižnice sa skladá z troch súborov: *cgaddag.py*, *node.py* a *gaddag.py*.

V súbore *cgaddag.py* je pomocou knižnice `ctypes` vytvorený tzv. *wrapper* okolo našej DLL knižnice. Zabezpečuje kompatiblitu typov jazyka C++ a Pythonu a umožňuje v Pythone volať funkcie z našej DLL knižnice. Definuje sa tu inštancia GADDAG štruktúry *cgaddag*.

Súbor *node.py* obsahuje triedu `Node`, ktorej inštancie reprezentujú uzly štruktúry. Táto trieda obsahuje funkcie, ktoré využitím skriptu v *cgaddag.py* volajú príslušné funkcie z DLL knižnice.

Trieda GADDAG

Súbor *gaddag.py* obsahuje triedu `GADDAG`, ktorá využitím inštancie *cgaddag* a triedy `Node` umožňuje volanie všetkých dôležitých funkcií z našej DLL knižnice a vytvára tak funkčnú pythonovskú inštanciu GADDAG štruktúry.

Základné atribúty triedy `GADDAG` sú:

- `str special_letterset`,
- `int MAX_CHARS`,
- `GADDAG_Struct* _gdg`,
- `set septets`.

Aby sme vedeli, aké písmená s diakritikou budú platné v abecede nášho GADDAG-u, trieda `GADDAG` pri inicializovaní načíta abecedu špeciálnych (s diakritikou) znakov zo súboru, implicitne zo súboru *alphabet.txt*. Napríklad, pre český jazyk súbor obsahuje reťazec „ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ“. Podľa obsahu súboru si trieda inicializuje atribúty `special_letterset` a `MAX_CHARS`, podľa ktorej následne vytvorí príslušnú inštanciu štruktúry z DLL knižnice.

Atribút `_gdg` obsahuje `GADDAG_Struct *` objekt z DLL knižnice, pomocou ktorého je možné pracovať so slovníkom.

Do triedy boli pridané pomocné metódy `int ch_to_idx(str ch)` a `str idx_to_ch(int idx)`, ktoré sú totožné s rovnomennými funkciami v DLL knižnici.

Trieda obsahuje metódy `save(str path)` a `load(str path)`, ktoré štruktúru uložia a načítajú. Okrem toho sa v skripte *gaddag.py* nachádza aj funkcia `GADDAG load(str path)` mimo triedy `GADDAG`, ktorá tiež umožňuje načítanie štruktúry.

Konverzné funkcie na kódovanie špeciálnych znakov na ASCII znaky sú funkcie `ascii_to_unicode(ascii_word)` a `unicode_to_ascii(unicode_word)`. Pre kódovanie z UNICODE do ASCII budú znaky s diakritikou prekonvertované na nealfabetické znaky typu `char` a znaky bez diakritiky ostanú zachované (ich formát sa len zmení na `char`). Premenná `char ch = 'A'` má číselnú hodnotu 65. Nech je počet špeciálnych znakov našej abecedy n, potom budú tieto znaky príslušnou funkciou zakódované do ASCII znakov s hodnotami od $65 - n$ do 64. Česká abeceda špeciálnych znakov *„ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ“* je teda prekonvertovaná na znaky *„23456789:;<=>?@“* a slovo „AKČNĚJŠÍ“ typu UNICODE bude prekonvertované na slovo *„AK3N6J;7“* typu ASCII. Tieto konverzné funkcie využíva metóda `add_word(str word)` a dotazovacie metódy.

Dotazovacie metódy triedy sú `str[] contains(str subword)`, `str[] starts_with(str prefix)`, `str[] ends_with(str suffix)` a `bool has_word(str word)`. Všetky textové reťazce, na ktoré sa budeme v dotazovacích metódach pýtať, budú najprv zmenené na *uppercase*, potom prekonvertované našou konverznou funkciou do ASCII a následne zadané ako argument príslušnej funkcie DLL knižnice. Naopak slová, ktoré sú odpoveďou na nami zadaný

dotaz, budú najprv vybrané zo štruktúry `Result *`, prekonvertované z ASCII do ich UNICODE verzie, pridané do typu `List` a nakoniec volajúcou funkciou vrátené.

Na generovanie platných ťahov a hľadanie *lettersetov* pre *bezpísmenové kotvy* bude potrebné pridať funkcie, ktoré pre zadaný podreťazec nájdu všetky písmená, ktorými je možné ho rozšíriť (zľava alebo sprava) tak, aby stále išlo o podreťazec nejakého slova v slovníku. Pre príklady znovu použijeme štruktúru `GADDAG gdg` so slovami RAJA, JAR, JAHODA, POHODA a JABLKO. Trieda `GADDAG` obsahuje pre tieto účely funkcie:

- `right_letterset(self,subword)` – vráti zoznam všetkých písmen l, ktorými je možné rozšíriť `subword` sprava tak, že výsledný podreťazec *subword.l* je podreťazcom slova v slovníku. Napríklad, `gdg.right_letterset("a")` vráti `["B","H","J","R"]` (pripomňme si, že naša štruktúra nerozlišuje medzi veľkými a malými písmenami).

- `left_letterset(self,subword)` – vráti zoznam všetkých písmen l, ktorými je možné rozšíriť `subword` zľava tak, že výsledný podreťazec *l.subword* je podreťazcom slova v slovníku. Napríklad, `gdg.left_letterset("A")` vráti `["D","J","R"]`.

Pomocné metódy pre tieto funkcie sú:

- `str[] get_nonending_letters(int node)` – vráti zoznam písmen, pre ktoré z vrcholu niekam vedie hrana,

- `str[] get_ending_letters(int node)` – vráti zoznam písmen, ktoré sú v `_gdg.letter_sets[node]`.

Metóda `change_lang(GADDAG new_gdg)` nastaví inštancii `gdg` atribúty objektu `new_gdg`. Táto situácia nastáva vtedy, keď sa zo ZIP súboru načítava GADDAG štruktúra pre iný ako predvolený jazyk hry.

Trieda `GADDAG` má jedinú používanú inštanciu definovanú v tomto skripte, a tou je `gdg`.

Na vypočítanie heuristiky pre platné ťahy v strednej hre budeme potrebovať získať zo slovníka zoznam všetkých unikátnych *septetov*, t. j. zoznam všetkých neusporiadaných sedmíc písmen, ktorých permutáciou je možné získať platné slovo v slovníku. Na to máme vytvorenú metódu `find_septets()`, ktorá rekurzívne prejde našou GADDAG štruktúrou a vráti zoznam týchto *septetov*.

5.2 Hracie kamene, hracia doska, zásobníky

5.2.1 Hracie kamene

Trieda Letter

Hrací kameň je reprezentovaný inštanciou triedy `Letter`, definovanej v skripte *letters.py*. V atribúte `str letter` je uložené písmeno (znak) a atribút `int value` obsahuje jeho bodovú hodnotu. Trieda obsahuje aj override funkcie `__eq__(self)`, ktorá zabezpečuje, aby sa kamene s rovnakými znakmi a bodovou hodnotou považovali za rovnaké.

Trieda Bag

Vrecúško s kameňmi je reprezentované triedou `Bag`. Základným atribútom je slovník, ktorého kľúče sú inštancie triedy `Letter` a kľúčmi je ich počet vo vrecúšku. Obsah vrecúška sa pri inicializácii načíta zo súboru, implicitne z *letters_in_a_bag.csv*.

Jedným z hlavných atribútov triedy je `remaining_letters`. Je to zoznam písmen (vo formáte `str`), ktoré sa vo vrecúšku nachádzajú. Umožňuje to prehľadnejšiu manipuláciu s obsahom vrecúška. Druhý hlavný atribút je slovník `bag`. Kľúče sú inštancie triedy `Letter` a hodnota prislúchajúca kľúču je počet písmen vo vrecúšku reprezentovaný ako `int`.

Funkcie v tejto triede:

- `reset()` – dá obsah vrecúška do stavu, v akom bol na začiatku.
- `Letter get_letter(str letter)` – vráti z vrecúška inštanciu triedy `Letter` s písmenom `letter`.
- `Letter take_letter(str letter)` – vráti inštanciu triedy `Letter` a odstráni ju z vrecúška.
- `add_letter_back_to_bag(Letter letter)` – vráti kameň naspäť do vrecúška.
- `change_lang(Bag new_bag)` – správne nastaví obsah vrecúška v globálnom objekte `bag` pri zmene jazyka hry, pre ktorý bolo vytvorené vrecúško `new_bag`.

Trieda `Bag` má stále len jednu používanú inštanciu – singleton `bag` (nemýliť si s atribútom `bag` triedy `Bag`), ktorý je definovaný v skripte *letters.py*. Ostatné triedy a procedúry k tomuto singletonu pristupujú importovaním tejto inštancie použitím `from letters import bag`.

Načítanie jazyka zo ZIP súboru

V *letters.py* je funkcia `load_zip(str archive, str gdg_file="")`, ktorá zo ZIP archívu načíta všetky potrebné počiatočné údaje o jazyku hry. Archív musí obsahovať súbory *.gdg*, *alphabet.txt* a *letters_in_a_bag.csv*. Súbor s príponou *.gdg* je možné z textového súboru so zoznamom slov vygenerovať skriptom *dictionary_loader.py* v priečinku *working_pygaddag* a reprezentuje GADDAG štruktúru pre slovník daného jazyka. Súbor *alphabet.txt* obsahuje zoznam všetkých znakov s diakritikou v abecede daného jazyka, najprv sú vymenované malé písmená, následne sú vymenované im prislúchajúce veľké písmená. Tento súbor je nevyhnutný na správnu reprezentáciu slovníka triedou `GADDAG`. Súbor *letters_in_a_bag.csv* obsahuje zoznam kameňov, ktoré sú na začiatku hry vo vrecúšku. Každý kameň je na samostatnom riadku v jednotnom formáte *písmeno,hodnota,počet*. Táto funkcia umožňuje načítavať rôzne jazyky Scrabble aj počas behu programu. Funkcia vracia `tuple (GADDAG new_gdg,Bag new_bag)`.

5.2.2 Hracia doska

Trieda Square

Trieda `Square` reprezentuje jednotlivé políčka na hracej doske. Jej atribúty sú:

- `Letter letter` – ak je na políčku položené písmeno, atribút obsahuje inštanciu triedy `Letter`, inak obsahuje `None`.
- `int column, row` – súradnice políčka na hracej doske. `row` značí riadok, `column` značí stĺpec.
- `int letter_bonus` – ak prémiové políčko násobí hodnotu písmena, atribút obsahuje tento násobok. Ostatné políčka tam majú hodnotu 1.
- `int word_bonus` – ak prémiové políčko násobí hodnotu slova, `word_bonus` obsahuje tento násobok. Ostatné políčka tam majú hodnotu 1.
- `bool is_center` – ak je políčko stredové, je to `True`, inak `False`.

Metóda `str get_square_bonus()` vracia `str` obsahujúci modifikátory políčka, napr. „*“, „3W“ alebo „2L“.

Hracia doska sa načítava z csv súboru. V tomto súbore je každé políčko v špecifickom formáte. Ak ide o stredové políčko, reťazec začína hviezdičkou („*“). Ak ide o prémiové políčko násobiace hodnotu písmena, prvý znak je číslica (multiplikátor) a nasledujúci znak je „L“. Ak ide o prémiové políčko násobiace hodnotu písmena, prvý znak je číslica a nasledujúci znak je „W“. Ak na políčku leží nejaký kameň, znakový reťazec končí dvojbodkou a písmenom na kameni. Reťazec reprezentujúci políčko môže byť teda napríklad „:Č“, „2W:I“, „3L“ alebo „*:V“.

Funkcia `str get_square_str_repr()` vracia `str`, ktorý reprezentuje v csv súbore dané políčko.

Trieda Board

Trieda `Board` reprezentuje stav hry na hracej doske, t. j. hraciu dosku, slová na hracej doske a kotvy na hracej doske. Atribúty triedy:

- `Square[][] board` – 2D pole objektov `Square`, ktoré reprezentuje stav hracej dosky počas hry.
- `Square[] letters_on_board` – zoznam políčok, na ktorých sú vyložené kamene.
- `vertical_words_on_board` – zoznam informácií o vertikálne orientovaných slovách na hracej doske vo formáte `tuple`, kde prvou položkou sú súradnice prvého písmena slova vo formáte `[int row, int column]`, druhou položkou je slovo `str word` a treťou položkou smer položeného slova – pre vertikálne orientované slová je to `bool False`.
- `horizontal_words_on_board` – zoznam informácií o horizontálne orientovaných slovách v rovnakom formáte ako vo `vertical_words_on_board`, smer položeného slova je nastavený na `True`.

- `vertical_adjacent_anchors` – slovník s kľúčmi typu `Square` a hodnotami typu `str` pre prázdne políčka, z ktorých je možné začať vytvárať vertikálne slovo. Ide o prázdne políčka susediace s políčkami, na ktorých sú vyložené kamene. Pridaním kameňa na toto políčko vzniknú v spojení so susednými obsadenými políčkami nové podreťazce. Hodnota v slovníku pre daný kľúč je `str` obsahujúci všetky písmená, ktoré je možné na políčko položiť tak, že v spojení so susednými kameňmi vznikne podreťazec nejakého platného slova v slovníku.

- `horizontal_adjacent_anchors` – slovník podobný slovníku pre vertikálne vedľajšie kotvy určený pre tie horizontálne orientované.

Ak je na začiatku hry prázdna hracia doska, do `vertical_adjacent_anchors` a `horizontal_adjacent_anchors` sa pridá stredové políčko hracej dosky a ostatné atribúty ostávajú prázdne.

Pri inicializácii hracej dosky sa volá funkcia `find_all_words()`, ktorá vyhľadá všetky slová na hracej doske, pridá ich do atribútov `*_words_on_board` a `letters_on_board` a všetky prázdne políčka susediace so slovami pridá ako kotvy do `*_adjacent_anchors`. Po pridaní nového slova v priebehu hry sa aktualizuje obsah atribútov `Square.letter` v poli `board` pre zmenené políčka sa podľa orientácie nového slova zavolajú funkcie `add_word_to_horizontals(param)` s `add_adj_vertical_anchors(param)` alebo `add_word_to_verticals(param)` s `add_adj_horizontal_anchors(param)` s rovnakým parametrom `param = ([int row,int column],str word,bool direction)`.

Funkcia `open_file(str filename,bool abs_path)` načíta obsah hracej dosky z csv súboru zadaného s absolútnou alebo relatívnou cestou. Funkcia `save_board_to_a_file(filename)` obsah hracej dosky uloží do csv súboru. Každé políčko je v csv súbore reprezentované vo formáte špecifikovanom v dokumentácii pre triedu `Square`.

Funkcia `Board copy_board()` vráti inštanciu kópie celého objektu triedy `Board`.

Funkcia `str print_board(str layer)` vypíše na výstup stav hracej dosky v tvare 2D tabuľky políčok. Keď je argument `layer="bonus"`, vypíšu sa informácie o prémiových políčkach hracej dosky, pre `layer="letters"` sa vypíšu iba písmená kameňov vyložených na hracej doske a pre `layer="both"` sa vypíšu údaje ako o prémiových políčkach, tak aj o vyložených kameňoch. Funkcia taktiež vracia stav hracej dosky vo formáte `str`.

Rozloženie prémiových polí a počiatočných písmen na hracej doske je možné manuálne modifikovať v súbore *gameboard.csv*. Všetky bunky tabuľky musia byť vo formáte špecifikovanom v sekcii Trieda Square.

5.2.3 Zásobníky

Trieda Rack

Trieda `Rack` reprezentuje zásobník hráča. Atribút `Letter[] rack` obsahuje zoznam kameňov na zásobníku a atribút `draw_history` obsahuje zoznam kameňov, ktoré si hráč ťahal po jednotlivých kolách.

Funkcia `undo_drawing_tiles` využíva `draw_history` a vráti do vrecúška s kameňmi naspäť naposledy vytiahnuté kamene.

Funkcia `fill_rack_randomly()` podľa kapacity zásobníka náhodne vytiahne z vrecúška zostávajúce kamene. Ak je vrecúško dostatočne plné, vytiahne sa toľko kameňov, aby ich na zásobníku bolo 7, inak sa vytiahne toľko, koľko sa dá.

Funkcia `bool is_rack_full()` vráti `True`, ak je možné dotiahnuť z vrecúška do zásobníka nejaké kamene, inak vráti `False`.

Zásobníky hráčov inicializuje funkcia
`Rack[] initialize_racks(str filename,abs_path=False)` definovaná v skripte *letters.py*. Zásobníky načítava z csv súboru (implicitne z *racks.csv*), ktorý je zadaný buď s relatívnou alebo absolútnou cestou. Obsah tohto súboru je možné upravovať a nastaviť tak počiatočný obsah zásobníkov. V tomto programe budeme počítať výhradne s dvomi hráčmi, a teda aj s dvomi zásobníkmi.
Pre `filename=""` sa inicializujú prázdne zásobníky.

5.3 Umelá inteligencia hráča

5.3.1 Reprezentácia stavu hry

Triedy použité na reprezentáciu stavu hry sa nachádzajú v skripte *game.py*.

Trieda Player

V prvom rade je nutné vhodne reprezentovať hráčov – ich skóre, úroveň a zásobníky. Hráč je reprezentovaný inštanciou triedy `Player`. Atribúty tejto triedy sú:

- `Rack rack` – zásobník hráča.
- `int score` – skóre hráča, implicitne nastavené na 0.
- `IQ iq` – úroveň hráča. `IQ(Enum)` je trieda typu enumerate, ktorá môže mať tri hodnoty: *DUMB*, *NORMAL* a *SMART*. *SMART* hráč zahrá najlepší ťah podľa heuristiky, *NORMAL* zahrá ťah za najvyšší počet bodov a *DUMB* zahrá náhodný platný ťah. Implicitne je hráčova úroveň nastavená na *NORMAL*.

Trieda Game

Okrem hráčov a ich zásobníkov stav hry reprezentuje hracia doska. Všetky tieto informácie zoskupuje trieda `Game` do jedného objektu. Reprezentácia celého stavu hry v jednom objekte je tiež výhodná na zálohovanie stavu hry, čo umožňuje vrátiť späť ťah. Atribúty triedy `Game` sú:

- `Player player1` – hráč, ktorý je momentálne na ťahu.
- `Player player2` – hráč, ktorý momentálne nie je na ťahu.
- `Board board` – hracia doska.
- `Game[] history` – história predošlých stavov hry, umožňuje vrátiť späť zahraný ťah. Implicitne je to prázdny zoznam.

- `GameState state` – stav hry určujúci, či ide o strednú hru alebo koncovku. `GameState(Enum)` je trieda typu enumerate, ktorá môže nadobúdať hodnotu *MIDDLEGAME* alebo *ENDGAME*.

Funkcia `backup_to_history()` vytvorí kópiu objektu a pridá ju do zoznamu `history`. Metóda `swap_players()` vymení hráčov `player1` a `player2`, takže po výmene bude na ťahu hráč uložený pôvodne v `player2`.

Metóda `save_racks(str filename)` uloží obsahy zásobníkov hráčov do csv súboru. Funkcia `undo_last_change()` vráti hru do stavu, v ktorom bola pred potiahnutím posledného ťahu. Inštancii sa nastavia hodnoty posledne pridaného objektu triedy `Game` do `history`.

5.3.2 Umelá inteligencia

Trieda PlayerAI

`PlayerAI` je hlavná trieda zabezpečujúca umelú inteligenciu hráča. Nachádza sa v skripte *playerAI.py*. Základnými atribútmi sú:

- `Game game` – objekt, z ktorého trieda čerpá informácie o hracej doske, hráčoch a zásobníkoch.
- `bool game_ended` – `True`, ak nastane koniec hry, t. j. ak nejaký z hráčov v koncovke vyprázdnil svoj zásobník alebo nie je možné zahrať žiadny ťah dve kolá po sebe. Inak má hodnotu `False`.
- `Move[] newly_sorted_moves` – zoradený zoznam nájdených ťahov. Viac o triede v sekcii Trieda Move.
- `Validator validator` – inštancia pomocnej triedy `Validator` (viac v sekcii Trieda Validator), zabezpečujúca kontrolu platnosti navrhnutých ťahov.

Funkcia `Move[] generate_moves()` vráti zoznam všetkých platných ťahov. Prejde všetky kotvy (t. j. políčka s písmenami a vedľajšie kotvy), ktoré sú uložené v `game.board`. Pre každý druh kotvy (prázdne/obsadené políčko alebo horizontálna/vertikálna kotva) vytvorí novú inštanciu triedy `Validator`, ktorá umožní generovanie slov na 2D hracej doske zúžiť na 1D problém (viac v 5.3.2). Funkcia volá funkciu `find_valid_words`, ktorá s počiatočnými parametrami začne rekurzívne volanie funkcií `generate_opts(int pos,Letter[] rack,str word)` a `process_opts(int pos, Letter[] rack,str L, str word)`.

Funkcia `find_best_move()` podľa úrovne hráča, ktorý je na ťahu (`game.player1.iq`), vráti zo zoznamu možných ťahov ten, ktorý hráč zahrá.

Ťah je zahraný pomocou funkcie `play_move(Move move)`, ktorá položí slovo na hraciu dosku, aktualizuje zoznamy kotiev a skóre hráčov a nakoniec vymení hráčov zavolaním funkcie `game.swap_players()`.

Funkcia `undo_move()` vráti späť naposledy zahraný ťah. Funkcia `add_final_points()` na konci hry pripočíta/odpočíta k skóre hráčov súčet kameňov, ktoré im ostali v zásobníku.

Heuristika pre strednú hru sa vypočítava zavolaním funkcie `find_best_middlegame_move()`, ktorá postupne volá funkcie počítajúce jednotlivé parametre heuristiky.

Funkcia `compute_next_bonus_probability(Letter[] remaining_rack)` počíta pravdepodobnosť binga (e). Pre každý zo septetov uložených v `gdg.septets` vypočíta pravdepodobnosť, že ho po ťahaní kameňov z vrecúška budeme mať na zásobníku a tieto pravdepodobnosti spolu s ďalšími parametrami použije na výpočet hodnoty e.

Defenzívnosť d je počítaná v triede `Validator`.

Na výpočet heuristiky pre koncovku je potrebné prehľadávať stavový strom. Uzol stavového stromu je reprezentovaný triedou `Node`. Atribúty tejto triedy sú:

- `Node parent` – rodič uzla;
- `Node[] successors` – deti uzla;
- `int optim, pessim` – najoptimistickejší a nejpesimistickejší odhad uzla;
- `int score` – skóre v uzle (koreň má skóre 0 a skóre potomkov sa spočíta podľa počtu bodov zahraného ťahu);
- `Move move` – ťah, ktorého zahraním sme sa do uzla dostali.

Heuristika pre koncovku je počítaná zavolaním `find_best_endgame_move()`, ktorá nastaví počiatočné parametre pre funkciu `b_star_top_level(Node startingNode)`. Pokiaľ nie je určený jednoznačne najlepší ťah (alebo pokiaľ neprebehne dostatočný počet iterácií), funkcia rekurzívne volá `b_star_lower_level(Node curNode, bool provebest_strategy)`. Táto funkcia rekurzívne prepočítava intervaly uzlov a keď narazí na list, vyhodnotí ho zavolaním funkcie `evaluate_successors(Node parent)`.

Trieda WordData

Trieda `WordData` reprezentuje jedno slovo, ktoré je možné na hraciu dosku vyložiť. Nachádza sa v skripte *validator.py*. Jej atribúty sú:

- `str word` – slovo, ktoré je možné vyložiť.
- `int[2] coords` – súradnice políčka, na ktoré je položené prvé písmeno slova.
- `score` - počet bodov, ktorý za dané slovo hráč získa.
- `bool direction` – orientácia slova; `True` pre horizontálne položené slovo a `False` pre vertikálne položené slovo.

Všetky inštancie triedy `WordData`, ktoré majú všetky atribúty rovnaké, sa považujú za tú istú inštanciu (to je zabezpečené vlastnou definíciou funkcie `__eq__(WordData other)`).

Trieda Move

V jednom ťahu je možné vytvoriť niekoľko slov naraz. Trieda `Move` obsahuje všetky informácie potrebné na vypočítanie heuristiky pre strednú hru a vyhodnotenie skóre celého ťahu. Nachádza sa v skripte *validator.py*. Atribúty triedy `Move` sú:

- `WordData[] words` – zoznam slov (vrátane informácií o ich položení, skóre a orientácii), ktoré sú zahraním ťahu vytvorené.
- `score` – súčet skóre všetkých vytvorených slov.
- `Letter[] remaining_rack` – zoznam písmen, ktoré po zahraní ťahu ostanú na zásobníku. Používa sa pre vypočítanie heuristiky pre strednú hru.
- `e` – pravdepodobnosť binga – jeden z parametrov heuristiky pre strednú hru.
- `d` – defenzívnosť ťahu – ďalší z parametrov heuristiky pre strednú hru. Jeho výsledná hodnota sa počíta v triede `Validator`.
- `v` – výsledok evaluácie celej heuristiky.

`Dictionary {int,int} defensiveness_dict` je statický atribút triedy, podľa ktorého sa určuje d – defenzívnosť ťahu. Pri preskúmaní okolia novovyložených kameňov sa počíta maximálny multiplikátor, ktorý vie súper potenciálne zahrať použitím jedného z novovyložených kameňov. Prémiové políčko „3-krát hodnota písmena“ má v tomto prípade hodnotu 2,5, prémiové políčka na násobenie hodnoty slova majú hodnotu svojho multiplikátora, t. j. hodnoty 3 a 2.

Skóre ťahu sa nepočíta priamo v konštruktore, pretože, zoznam slov `words` obsahuje aj slová, ktoré už sú na hracej doske vyložené. Zoznam slov sa najprv musí prefiltrovať funkciou `filter_words_in_move(Board board)`, aby v ňom ostali len nové slová, a potom sa skóre ťahu aktualizuje pomocou funkcie `update_score()`.

Keď sú zistené všetky parametre heuristiky, `calculate_final_points()` vypočíta výslednú hodnotu v.

Statická metóda `Move[] get_best_move(Move[] moves)` zoradí zoznam možných ťahov podľa hodnoty v vypočítanej heuristikou.

Funkcia `str generate_move_listview_string()` vráti textovú reprezentáciu ťahu, ktorá sa zobrazí v GUI.

Trieda Validator

Trieda `Validator` má na starosti zjednodušenie generovania ťahov, kontrolu nájdených slov a čiastočnú evaluáciu defenzívnosti platného ťahu. Trieda je uložená v skripte *validator.py*.

Táto trieda v prvom rade umožňuje generovanie platného slova v 1D. V konštruktore sa špecifikujú súradnice štartovacieho políčka `int[2] starting_pos`, smer generovania slova `bool is_horiz_dir`, hracia doska `Board board` a zásobník hráča `Letter[] rack` a inicializuje sa objekt, ktorý umožní funkciám `PlayerAI.generate_opts(...)` a `PlayerAI.process_opts(...)` pohybovať sa

po políčkach hracej dosky len v smere zhora-nadol alebo zľava-doprava v okolí štartovacieho políčka – kotvy.

Po inicializácii objektu a zafixovaní smeru je možné dávať prechádzaným políčkam intuitívne relatívne súradnice: súradnicu 0 má štartovacie políčko – kotva; záporné súradnice majú políčka naľavo, resp. nad kotvou a kladné súradnice majú políčka napravo, resp. pod kotvou. Funkcia `get_relat_square(Square pos)` vráti relatívnu súradnicu pre skutočné súradnice zadaného políčka a funkcia `get_abs_coords(int relat_pos)` vráti podľa relatívnej súradnice dané políčko.

`record_word(Letter[] word,int pos,Letter[] rack,int state)`, hlavná funkcia triedy, dostane ako argument potenciálne platné slovo. Najprv skontroluje, či položením kameňov slova na dané políčka vo všetkých smeroch vzniknú len platné slová; t. j. ak sme položili slovo vertikálne, musíme skontrolovať, či sú všetky horizontálne slová, ktoré položením hlavného slova vznikli, tiež platné a pod.

Funkcia `bool along_check(int[2] coords,int length)` kontroluje, že počiatočný a konečný kameň slova nesusedia v smere slova so žiadnym ďalším kameňom.

Funkcia `(bool,WordData) inspect_cross_check(int pos,str L)` skontroluje, či v smere kolmom na nájdené slovo na riadku, resp. stĺpci s relatívnou súradnicou `pos` nevzniklo nejaké ďalšie slovo. Táto funkcia sa volá pre každý kameň slova zadaného ako parameter funkcii `record_word(...)`. Ak na danom riadku/stĺpci funkcia našla nejaké ďalšie nové slovo, vráti `True, WordData word`, kde `word` je nájdené slovo. Ak sa na riadku slovo nenašlo, vráti sa `True, None` a ak sa zistilo, že v kolmom smere vzniklo neplatné slovo, funkcia vráti `False, None`.

Keď kontrola navrhnutého slova prebehla úspešne, slovo spolu s ďalšími novovzniknutými slovami v kolmom smere sa pridajú ako inštancia triedy `Move` do zoznamu `Move[] found_moves`. Ak sa ťah našiel počas strednej hry, t. j. `state = 1`, funkcia mu vypočíta aj jeho defenzívnosť d, ktorá je potrebná na výpočet heuristiky.

Keď sa vyčerpajú všetky možné ťahy pre danú inštanciu triedy `Validator`, funkcia `PlayerAI.generate_moves()` si uloží zoznam nájdených ťahov `found_moves` a začne prechádzať ďalší typ kotiev vytvorením novej inštancie triedy `Validator`. Táto funkcia po zozbieraní všetkých platných ťahov zavolaním aktualizačných metód triedy `Move` ťahom vyhodnotí skóre ťahu a výsledok heuristiky v a vráti zoradený zoznam všetkých platných ťahov.

5.4 Grafické používateľské rozhranie

Grafické používateľské rozhranie (ďalej už len GUI) sa nachádza v skriptoch *gui.py*, *app_main_window.py*, *helpAboutWidget.py* a *helpSquareColorsWidget.py*, využíva sa knižnica PyQt5. Časť GUI bola vytvorená v programe PyQt5 designer.

Objekty v hlavnom okne sú definované v skripte *app_main_window.py*, dve vedľajšie okná – nápoveda pre farby políčok v hracej doske a okno s informáciami o autorovi programu sú v skriptoch *helpAboutWidget.py* a *helpSquareColorsWidget.py*. Prepojenie jednotlivých prvkov GUI s funkčnými časťami programu a správa udalostí je v skripte *gui.py*.

V *gui.py* sa spúšťa GUI, ktoré je inštanciou triedy `MainWindow`, definovanej v rovnakom skripte. Pri inicializácii sa načíta hracia doska a zásobníky a vytvoria

sa inštancie reprezentujúce hráčov, stav hry a umelú inteligenciu. Sú vytvorené dodatočné objekty typu `QLabel` a prepoja sa signály (spúšťače udalostí) grafických objektov s metódami, ktoré ich spravujú.

Metódy triedy `MainWindow` sa dajú rozdeliť do 4 kategórií: inicializačné metódy, aktualizačné metódy, metódy spravujúce signály objektov a metódy zabezpečujúce komunikáciu medzi back-endom a front-endom.

5.4.1 Inicializačné metódy

- `fix_helpsquare_pixmaps()` – aktualizujú cesty k obrázkom používaným v nápovede *Help»Square color meaning*.
- `initialize_board()` – na začiatku hry inicializuje hraciu dosku. Každé políčko je inštanciou pomocnej triedy `GUISquare`, ktoré obsahuje nemenný `QLabel` zobrazujúci políčko hracej dosky a premenný `QLabel` reprezentujúci kameň položený na danom políčku. Ak je políčko prázdne, kameň sa nezobrazí a v opačnom prípade sa zobrazí `QLabel` s písmenom a jeho bodovou hodnotou, ktoré je na políčku položené. Pri zmene jazyka sa výzor hracej dosky nijako nemení.
- `initialize_racks()` – inicializuje zásobníky, ktoré sú zobrazené hnedými obdĺžnikmi. Zásobník hráča na ťahu je buď prázdny alebo naplnený kameňmi, podobne ako na hracej doske reprezentovanými objektmi `QLabel`.
- `initialize_letter_menu()` – metóda inicializuje tzv. písmenové menu, ktoré hráčovi umožňuje vybrať si z „vrecúška“ také kamene, aké sám chce. Písmenové menu je flexibilné na zmenu jazyka, t. j. veľkosť kameňov, ich usporiadanie a poloha tlačidla *Draw random* sa prispôsobuje veľkosti aktuálne používanej abecedy tak, aby sa tam všetky písmená zmestili.
- `create_letter_label(Letter letter, int size=50)` – funkcia vráti inštanciu `QLabel` s obsahom nastaveným tak, aby zodpovedal zadanému písmenu.
- `create_lettercount_label(int text_size,Letter letter)` – funkcia vráti `QLabel`, ktorý v písmenovom menu zobrazuje početnosť príslušného písmena vo vrecúšku.

5.4.2 Aktualizačné metódy

- `update_board()` – aktualizuje zobrazovaný stav hracej dosky podľa stavu `PlayerAI.game.board`.
- `update_score_labels` – aktualizuje zobrazované skóre hráčov.
- `update_racks()` – aktualizuje obsah viditeľného zásobníka.
- `rewrite_letter(QLabel label, Letter letter)` – prepíše obsah `label` tak, aby zodpovedal písmenu `letter`.
- `update_letter_menu()` – aktualizuje početnosti jednotlivých kameňov vo vrecúšku.

- `update_letter_menu_tile(Letter letter)` – aktualizuje početnosť kameňa `letter`.
- `show_moves()` – ťahy uložené v `PlayerAI.newly_found_moves` zobrazí v ponuke ťahov `QListWidget moveListView`.

5.4.3 Metódy spravujúce signály objektov

- `openZipDialog()`, `openFileNameDialog()`, `saveFileDialog()` – spravujú možnosti *Load»Language*, *Load»Position* a *Game»Save game*.
- `menu_tile_clicked(Letter letter)`, `return_tile_from_rack(Letter letter)` – prvá metóda pridá kameň z vrecúška do zasobníka a druhá vráti kameň zo zásobníka do vrecúška.
- `move_in_list_clicked()`, `move_in_list_doubleclicked()` – po jednom kliknutí zobrazí navrhovaný ťah na hracej doske, po dvojitom kliknutí daný ťah zahrá.
- `difficulty_dumb_checked()`, `difficulty_normal_checked()`, `difficulty_smart_checked()` – nastaví IQ súpera (ak je zapnuté automatické ťahanie za súpera), metódy spravujú možnosti v *Options»Opponent»Difficulty*.
- `opp_autoplay_on_clicked()`, `opp_autoplay_off_clicked()` – zapne, resp. vypne automatické ťahanie súpera v *Options»Opponent»Automatic play for the opponent*.
- `show_all_moves_checked`, `show_top_15_moves_checked` – zobrazí všetky ťahy alebo 15 najlepších ťahov, spravujú možnosti v *Options»Analyser*.
- `play_best_button_clicked()`, `play_random_button_clicked()`, `draw_random_button_clicked` – spravujú signály tlačidiel *Play best*, *Play random* a *Draw random*.
- `change_move_list_visibility` – spravuje zaškrtávacie políčko pod ponukou ťahov, ktorá ponuku skryje alebo zobrazí.
- `new_game_clicked(...)` – spravuje možnosť *Game»New game* a nastaví hru na začiatok stále, keď je to potrebné, t. j. po načítaní nového jazyka alebo novej pozície zo súboru.

5.4.4 Metódy na prepojenie front-endu a back-endu

- `play_move(Move move)` – zahrá zvolený ťah v back-endovej časti a zavolá aktualizačné funkcie na front-end.
- `undo_move()` – vráti späť ťah v back-endovej časti a zavolá aktualizačné metódy na front-end.

- `find_moves_after_drawn_tiles()` – keď je zásobník plný, funkcia zavolá `PlayerAI.find_best_move()`, ktorý nájde všetky platné ťahy a zoradené ich uloží do `PlayerAI.newly_found_moves`, z ktorého čerpá aktualizačná metóda `show_moves()`.

6. Používateľská dokumentácia

6.1 Minimálne požiadavky

Program je momentálne podporovaný na operačných systémoch Windows a Linux. Je potrebné mať nainštalovaný Python 3.8 (alebo vyššie verzie) a správcu balíčkov Pip. Používané dynamické knižnice sú kompilované pre x86_64 architektúru.

6.2 Inštalácia

Pre OS Linux je potrebné najprv spustiť skript *setup_linux.py*. Pre OS Windows sa najprv spúšťa skript *setup_windows.py*. Následne je už možné spustiť program skriptom *gui.py*.

6.3 Ovládanie

Program sa zapína spustením skriptu *gui.py*. Všetok text v programe je písaný v anglickom jazyku, ponuka kameňov a navrhované ťahy sú však stále v takom jazyku, aký sa načítal na hranie Scrabble.

Obr. 6.1: Grafické používateľské rozhranie programu.

6.3.1 Panel s menu

Panel s menu, ktorý sa nachádza v hornej časti GUI na obrázku 6.1, je označený číslom 1.

Game

- *New Game* – reštartuje hru. Obsah hracej dosky a zásobníkov je nanovo načítaný z príslušných súborov, skóre oboch hráčov je vynulované;
- *Undo move* – vráti späť posledný ťah. Ak je súperom počítač, vráti späť aj posledný ťah počítača. Na využitie tejto možnosti je nutné najprv si doplniť zásobník;
- *Save game* – uloží obsah hracej dosky a zásobníkov do dvoch separátnych súborov. Po kliknutí sa spustí dialóg výberu súboru najprv pre hraciu dosku a potom pre zásobníky;
- *Exit* – ukončí program.

Load

- *Language* – otvorí dialóg výberu súboru umožňujúci vybrať ZIP súbor, z ktorého sa načíta slovník, špeciálna abeceda jazyka a obsah vrecúška pre požadovaný jazyk hry Scrabble. Ak bola predtým hra rozohraná, zahodí sa a obsah zásobníkov a hracej dosky sa reštartuje. Vysoko sa odporúča nemodifikovať súbory *racks.csv* a *gameboard.csv*, ak používateľ zamýšľa meniť počas behu programu jazyk hry;
- *Position* – otvorí sa dialóg výberu súboru a umožní vybrať csv súbory na načítanie pozície hry. Najprv sa vyberá súbor definujúci obsah hracej dosky a následne sa vyberá súbor definujúci obsah zásobníkov.

Options

- *Opponent* – ponúka možnosti nastavenia oponenta;
  - *Automatic play for the opponent* – umožňuje zapnúť/vypnúť automatické potiahnutie súpera;
    - *On* – po ťahu používateľa za súpera automaticky potiahne počítač s nastavenou úrovňou;
    - *Off* – po ťahu používateľa sa ukáže zásobník súpera a používateľ potiahne aj za druhého hráča;
  - *Difficulty* – ak je súperom počítač, je možné nastaviť jeho úroveň;
    - *Dumb* – počítač zahrá náhodný platný ťah;
    - *Normal* – počítač zahrá ťah za najvyšší počet bodov;
    - *Smart* – počítač zahrá najlepší ťah podľa implementovanej heuristiky;
- *Analyser* – nastavuje analyzér (č. 3 na obrázku 6.1);
  - *Show all possible moves* – analyzér ukáže všetky platné ťahy, ktoré je možné v tejto pozícii zahrať;
  - *Show 15 best moves* – analyzér ukáže len maximálne 15 najlepších ťahov.

Help

- *Square color meaning* – zobrazí okno s nápovedou pre farby políčok hracej dosky;
- *About* – zobrazí okno, ktoré obsahuje stručné informácie o autorovi programu.

6.3.2 Zoznam kameňov vo vrecúšku

Zoznam písmen vo vrecúšku sa nachádza v ľavej časti okna (č. 2 na obr. 6.1). Ukazuje všetky druhy kameňov vo vrecúšku a ich počty. Ak zásobník hráča nie je plný, kliknutím na ľubovoľný kameň v tomto zozname sa kameň pridá do zásobníka.

Stlačením tlačidla *Draw random* sa do zásobníka dotiahnu náhodné písmená z vrecúška tak, aby bol zásobník plný.

6.3.3 Zásobník

Obsah zásobníka hráča na ťahu je zobrazený v dolnej časti okna. Ak je zásobník plný, analyzér začne vyhľadávať a zoradzovať všetky platné ťahy, ktoré je možné zahrať. Ak zásobník nie je plný, je možné ho doplniť buď stlačením tlačidla *Draw random*, alebo ručným pridaním kameňov zo zoznamu kameňov. Kliknutím na kameň v zásobníku sa tento kameň vráti naspäť do vrecúška.

6.3.4 Analyzér

Analyzér (na obrázku 6.1 označený číslom 3) obsahuje zoznam možných ťahov, zoradených podľa hodnoty v vypočítanej heuristikou. S analyzérom je možné interagovať vtedy, keď je plný zásobník.

Každý ťah v analyzéri je v nasledujúcom formáte:

<poradie> <v> [<prvé políčko>:<smer> <zoznam slov>, <počet bodov>]

V tomto formáte je v hodnota ťahu podľa heuristiky a *smer* je orientácia prvého slova, pre horiontálny smer obsahuje „h“ a pre vertikálny smer „v“.

Kliknutím na ťah v analyzéri sa zvolený ťah zobrazí na hracej doske. Dvojitým kiknutím sa tento ťah zahrá.

Stlačením tlačidla *Play best* sa automatický zahrá ťah na prvom mieste. Stlačením tlačidla *Play random* sa zahrá náhodný ťah.

6.3.5 Ostatné

Informačný text (číslo 4 na obr. 6.1) obsahuje informácie k naposledy vykonaným akciám: vypisuje naposledy potiahnuté kamene, ťah zahraný súperom, upozornenie, že zásobník ešte nie je naplnený atď.

Skóre hráčov napravo od ich zásobníkov sa aktualizuje po každom kole.

Záver

Program na analýzu ťahov v Scrabble ponúka hráčom možnosť trénovať, skúšať, analyzovať odohraté partie a zisťovať, kde hráč urobil v odohranom zápase potenciálnu chybu. Užitočný analyzér by mal vedieť vyhodnocovať ťahy nielen podľa počtu získaných bodov za ťah, mal by brať ohľad aj na strategické aspekty pozície.

V našej práci sme na uloženie slovníka použili dátovú štruktúru GADDAG, ktorá umožňuje rýchle vyhľadávanie slov a generovanie ťahov, čo je pre rýchlosť celého programu kľúčové. Upravili sme pythonovskú knižnicu pre túto štruktúru tak, aby dokázala pracovať aj so znakmi s diakritikou a podporovala abecedy maximálnej dĺžky 63 písmen. To nám umožňuje jednoduché prispôsobovanie programu pre rôzne jazykové verzie Scrabble.

Aby bol analyzér užitočný aj pre skúsenejších hráčov, použili sme heuristiky pre strednú hru a koncovku, ktoré zohľadňujú aj strategickú stránku pozície a možných ťahov. Simulovali sme zápasy umelej inteligencie používajúcej naše heuristiky proti umelým inteligenciám nepoužívajúcim žiadnu heuristiku. Či už bola umelá inteligencia s heuristikou na začiatku hry na ťahu prvá či druhá, predviedla výborné výsledky a potvrdila tým jej funkčnosť a užitočnosť pri analýze pozícií.

Vytvorili sme prívetivé grafické používateľské rozhranie, ktoré umožňuje hrať proti počítaču na 3 rôznych úrovniach inteligencie. Používateľ môže taktiež hrať za obe strany a jednoducho si naplniť zásobník kameňmi, ktoré v ňom chce mať. Analyzér vyhodnocuje všetky možné ťahy a zoradí ich podľa toho, aké dobré sú podľa použitých heuristík. V grafickom používateľskom rozhraní je možné pred zahraním zvoleného ťahu sledovať, kde bude na hraciu dosku slovo položené a koľko bodov tým hráč získa. Program sa dokáže prispôsobiť rôznym abecedám jazykov a umožňuje ukladanie či načítanie rozohratej pozície.

Cieľom našej práce bolo vytvoriť analyzér, ktorý dokáže hodnotiť ťahy aj zo strategického hľadiska, dokáže sa flexibilne prispôsobiť rôznym jazykovým verziám hry Scrabble a bude spustiteľný na rôznych platformách. Náš program je možné spustiť na operačných systémoch Windows a Linux a umožňuje meniť jazykové varianty hry počas behu programu.

Cieľ práce považujeme za splnený. Priestor na zlepšenie vidíme v zrýchlení generovania ťahov, pohodlnejšom načítavaní nového jazyka do programu a v pridaní dodatočnej heuristiky zabezpečujúcej lepší prechod medzi strednou hrou a koncovkou. Veríme však, že program dobre poslúži aj skúseným hráčom Scrabble.

Zoznam použitej literatúry

Appel, A. W. a Jacobson, G. J. (1988). The world's fastest scrabble program. *Communications of the ACM*, **31**(5), 572–578.

Bass, J. (2020). GADDAG 1.0.1. `https://pypi.org/project/GADDAG/`. Stiahnuté: 03-02-2021.

Berliner, H. (1981). The B* tree search algorithm: A best-first proof procedure. In *Readings in Artificial Intelligence*, pages 79–87. Elsevier.

Berliner, H. J. a McConnell, C. (1996). B* probability based search. *Artificial Intelligence*, **86**(1), 97–156.

Česká asociace Scrabble. Pravidla hry scrabble. `http://scrabble.hrejsi.cz/pravidla`. Článok ku dňu: 25-05-2021.

Gordon, S. A. (1994). A faster scrabble move generation algorithm. *Software: Practice and Experience*, **24**(2), 219–232.

Mareš, M. (2021). *Průvodce labyrintem algoritmů*. CZ. NIC, zspo.

Palay, A. J. (1982). B* tree search algorithm-new results. *Artif. Intell.;(Netherlands)*, **2**.

Ramírez, A., Acuña, F. G., Romero, A. G., Alquézar, R., Hernández, E., Aguilar, A. R. a Olmedo, I. G. (2009). A scrabble heuristic based on probability that performs at championship level. In *Mexican International Conference on Artificial Intelligence*, pages 112–123. Springer.

Sheppard, B. (2002). Towards perfect play of scrabble.

Slovenský spolok SCRABBLE. Čo je scrabble? `https://www.hramescrabble.sk/`. Článok ku dňu: 25-05-2021.

Toal, G. a kol. Foreign language scrabble – Czech scrabble. `http://www.gtoal.com/scrabble/details/czech/czech.txt`. Obsah ku dňu: 11-03-2021.

A. Prílohy

A.1 Zdrojový kód

Elektronická príloha *scrabble_analyser.zip* obsahuje zdrojový kód potrebný na spustenie programu.

A.2 Obsah vrecúška v českej verzii Scrabble

Kameň	**Ks**	**Kameň**	**Ks**	**Kameň**	**Ks**	**Kameň**	**Ks**	**Kameň**	**Ks**
A_1	5	$É_3$	2	K_1	3	R_1	3	$Ů_4$	1
$Á_2$	2	$Ě_3$	2	L_1	3	$Ř_4$	2	V_1	4
B_3	2	F_5	1	M_2	3	S_1	4	X_{10}	1
C_2	3	G_5	1	N_1	5	$Š_4$	2	Y_2	2
$Č_4$	1	H_2	3	$Ň_6$	1	T_1	4	$Ý_4$	2
D_1	3	I_1	4	O_1	6	$Ť_7$	1	Z_2	2
$Ď_8$	1	$Í_2$	3	$Ó_7$	1	U_2	3	$Ž_4$	1
E_1	5	J_2	3	P_1	3	$Ú_5$	1	$[\ ]_0$	2

Tabuľka A.1: Kamene pre českú verziu Scrabble a ich početnosti. Číslo v dolnom indexe značí jeho hodnotu, prázdny kameň je žolík.

A.3 Záznam zápasu SMART vs. NORMAL

Elektronická príloha *simulacia_smart_vs_normal.txt* obsahuje výpis jednej zo simulácií zápasov. Záznam obsahuje tieto informácie:

- stav hry (stredná hra alebo koncovka);
- zvolený ťah, jeho skóre a jeho hodnota daná heuristikou;
- obsah zásobníka po dotiahnutí kameňov;
- obsah hracej dosky po zahranom ťahu;
- skóre hráčov po skončení kola.