Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Bughouse s asistenciou programu pre klasický šach

Bakalárska práca

2021
Patrik Černý

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Bughouse s asistenciou programu pre klasický šach

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	doc. Mgr. Tomáš Plachetka, Dr.

Bratislava, 2021
Patrik Černý

19256178

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Patrik Černý
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Bughouse s asistenciou programu pre klasický šach
Bughouse assisted by a classical chess engine

Anotácia: Navrhnúť herný algoritmus pre variant šachu Bughouse, ktorý využíva asistenciu existujúceho programu pre klasický šach. Implementovať navrhnutý algoritmus a porovnať ho s existujúcimi programami pre Bughouse.

Vedúci: doc. Mgr. Tomáš Plachetka, Dr.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
archív

Dátum zadania: 29.10.2020

Dátum schválenia: 31.10.2020

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Touto cestou by som sa chcel poďakovať hlavne môjmu školiteľovi, doc. Mgr. Tomáš Plachetka, Dr. Úprimné ďakujem mu patrí hlavne za pochopenie, nápady, čas, ktorý venoval mojej práci, a v neposlednom rade za ochotu, ktorá sprevádzala naše konzultácie a tvorbu práce počas celého roku.

Abstrakt

Hlavnou témou práce je vytvorenie architektúry a následne implementácia programu hrajúci šachový mód bughouse. Odôvodnili sme potrebnú zmenu bughouse pravidiel a ukázali problémy, ktoré sprevádzajú implementovanie kvalitných bughouse programov, popísali algoritmy, ktoré sú pri tvorbe používané. Naša architektúra priamo využíva asistenciu programu hrajúci klasický šach pri výbere ťahu, ktorý program vykoná. Taktiež sa riadi pravidlami, ktoré sme presne zadefinovali a bola testovaná proti iným bughouse programom.

Kľúčové slová: bughouse šach, evaluačná funkcia, algoritmus minimax

Abstract

Main goal of this thesis is to create an architecture and implement an engine for bughouse chess variant. We will explain changes to bughouse rules that needed to be done and showcase some of the problems that follow the implementation of a bughouse chess engine and optimization techniques that said engines use. Architecture that we created uses a classical chess engine during move making decisions and follows a set of bughouse rules we established. It was tested against other bughouse engines.

Keywords: bughouse chess, evaluation function, algorithm minimax

Obsah

Úvod **1**

1 Šachové varianty **3**

1.1 Vznik šachu . . . 3

1.1.1 Časová kontrola partie . . . 4

1.2 Vybrané varianty šachu . . . 5

1.2.1 Bughouse šach . . . 5

1.2.2 Chess 960 . . . 5

1.2.3 Horde chess a Endgame chess . . . 6

1.2.4 Varianty netradičných šachovníc . . . 6

1.2.5 Crazyhouse a atomic chess . . . 6

2 Programy hrajúce šach a bughouse **9**

2.1 Základné pojmy v šachovom programovaní . . . 9

2.2 Súčasný stav . . . 10

2.2.1 Existujúce bughouse programy . . . 10

2.2.2 Spoločné črty crazyhouse a bughouse šachu . . . 11

2.3 Vybrané používané algoritmy . . . 11

2.3.1 Reprezentácia šachovnice . . . 12

2.3.2 Evaluačná funkcia . . . 13

2.3.3 Minimax . . . 15

2.3.4 Alfa–beta . . . 15

2.3.5 Databázy otvorení . . . 16

2.3.6 Prehľadávanie do šírky a usporiadanie ťahov . . . 16

2.3.7 Tabuľka transpozícií . . . 17

2.3.8 Tiché vyhľadávanie . . . 18

3 Úprava bughouse pravidiel **19**

3.1 Analýza pravidiel v partii bez časovej kontroly . . . 19

3.1.1 Ťah tímu . . . 20

3.1.2 Definovanie pravidiel nezávislých na čase . . . 20

3.1.3 Generalizácia bughouse hry 20

4 Implementácia **23**

4.1 Klasická bughouse architektúra 23

4.1.1 Rozdiely v architektúre 23

4.1.2 Program Stockfish 24

4.2 Program Bagatur 25

4.2.1 Bagatur rozhranie BoardImpl 25

4.2.2 Generovanie ťahov 26

4.3 UCI 27

4.4 Výber ťahu 29

4.4.1 Statická bughouse evaluácia v programe Sjeng 29

4.4.2 Evaluácia figúr v zásobníku 30

5 Testovanie **31**

5.0.1 Používateľské prostredia DoubleChessBoard 31

5.0.2 Sjeng a CECP 31

5.0.3 Rozbor odohratých partií 32

5.0.4 Vylepšenia programu 33

Záver **35**

Zoznam obrázkov

1.1 Ukážka hry chaturanga 3
1.2 Ukážka variantu chess960 6
1.3 Ukážka variantu Four-player chess 6

2.1 Vysvetlenie FEN zápisu na pozícii 10
2.2 Ukážka pola 10x12 13
2.3 Ukážka možného ohodnotenia figúr 14
2.4 Ukážka minimax algoritmu 15
2.5 Ukážka prehľadávanie do šírky 17

4.1 Obrázok klasickej architektúry bughouse programu 24
4.2 Architektúra navrhnutého bughouse programu 25
4.3 Prekrytie šachu drop ťahom 27

5.1 Ukážka prostredia DoubleChessBoard 31

Zoznam tabuliek

2.1 Ohodnotenie figúr v zásobníku (ktoré sú k dispozícii pre drop ťahy) rôznych bughouse programov 14
2.2 Ohodnotenie pešiaka k dispozícii pre drop ťahy programom Sjeng . . . 15

4.1 V tabuľke uvádzame presné hodnoty tropizmu pre jednotlivé figúry programu Sjeng 30

5.1 Tabuľka ukazuje presné hodnoty evaluácie ukážkovej partie. Je vidieť, že aj keď sa evaluácia postupne zhoršuje (teda pozícia programu sa zhoršuje), evaluácia Stockfish programu a zásobníku figúr sú postačujúce ako evaluačná funkcia pre bughouse programy. 33

Úvod

Cieľom tejto práce je návrh a implementácia programu hrajúceho šachový mód bughouse, ktorý bude čiastočne využívať už existujúci šachový program pre klasický šach. Programy hrajúce klasický šach sa vytvárajú od 90. rokov minulého storočia a za tento čas sa daná oblasť preskúmala do hĺbky. Avšak pre populárny mód šachu bughouse je takýchto programov menej kvalitatívne aj kvantitatívne, preto sa ponúka využiť časti spomenutých klasických programov, tzv. evaluačné funkcie, a s ich pomocou vytvoriť program hrajúci bughouse.

Bughouse je šachový mód, ktorý hrajú štyria hráči rozdelení do dvoch teamov, na dvoch šachovniciach. K pravidlám klasického šachu pribúda nasledovné, ktoré charakterizuje tento variant — po vyhodení figúrky súpera túto figúrku predávame spoluhráčovi, ktorý má možnosť ju položiť na voľné políčko svojej šachovnice namiesto svojho ťahu. Práca nadväzuje na ročníkový projekt, v ktorom som implementoval program hrajúci bughouse s vlastnou evaluačnou funkciou. Cieľom je navrhnutie algoritmu a architektúry programu, jednoznačné definovanie bughouse pravidiel, ktorými sa bude program riadiť a porovnanie takto vytvoreného programu proti existujúcim bughouse programom.

Bughouse odvetvie šachového programovania je stále len veľmi málo preskúmané. Programy hrajúce klasický šach sú už roky silnejšie ako najlepší šachoví veľmajstri, v bughouse to tak nie je. Práve toto nás inšpiruje pozrieť sa na bughouse programy trochu inak a zamerať sa na využitie poznatkov klasických programov na zlepšenie tých bughousových.

V ďalších kapitolách sa postupne pozrieme na vývoj programov hrajúcich šach, na presné pravidlá variantu bughouse a spomenieme aj iné šachové varianty, problémy nastávajúce pri tvorbe bughouse programov, evaluačné funkcie a algoritmy, ktoré sú v týchto funkciách použité. Kapitola Úprava bughouse pravidiel sa bude venovať prevažne odôvodneniu zmien v bughouse pravidlách, ktorými sa budeme riadiť pri tvorbe programu. V kapitole Implementácia sa pozrieme na konkrétnu architektúru nášho programu a objasníme naše rozhodnutia pri jej tvorbe. Prácu uzavrieme testovaním programu proti iným bughouse programom, popísaným v kapitole Testovanie a zhodnotíme silu a prípadné vylepšenia takto vytvoreného programu.

Kapitola 1

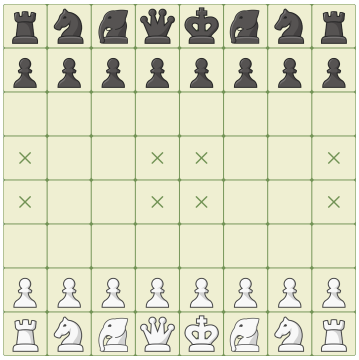
Šachové varianty

V tejto kapitole sa budeme venovať histórii šachu, predstavíme si rôzne varianty, ktoré časom vznikli a sú populárne dodnes. Viac si povieme o pravidlách bughouse, teda o variante, ktorý je hlavnou témou práce.

1.1 Vznik šachu

Šach ako taký sa v Európe začal šíriť okolo 15. storočia. Šachové figúrky metaforicky poukazujú na rozdelenie stredovekej spoločnosti, pravidlá šachu sa dodnes až na drobné úpravy nemenili. Pôvodná hra chaturanga, z ktorej sa postupne vyvinul šach vznikla okolo 8. storočia v Indii, je zobrazená na obrázku 1.1.

Peržania si osvojili chaturangu a pridali pravidlo definujúce dnešný šach. V indickom predchodcovi šachu bolo cieľom hry vyhodiť kráľa súpera. Na druhej strane, v peržskom chatrangu, bol cieľ dať mat — tak, ako v súčasnom šachu. Veľký vplyv na pravidlá mala aj čínska hra Xiangqi, ktorá bola inšpiráciou pre pridanie dynamických, viac útočných pravidiel [8].



Obr. 1.1: Ukážka základnej pozície hry chaturanga, pochádzajúcej z Indie. Považuje sa za predchodcu šachu, je vidieť drobné rozdiely, napr. králi nestoja na rovnakom stĺpci a figúrky majú k dispozícii iné ťahy.

Prvé rozsiahle štúdium dnešného šachu prišlo koncom 15. storočia, kedy mnísi vytvorili prvé primitívne spisy o jednotlivých častiach šachovej partie, tzv. otvorení, strednej hre a koncovke. O najväčší rozvoj šachu sa však zaslúžili počítače a programy hrajúce šach. Veľkým rokom vo vývoji takýchto programov bol 1997, kedy sa odohrala odveta zápasu medzi Garrym Kasparovom a programom *DeepBlue*. Bol to prvý zápas medzi svetovým šampiónom a programom, v ktorom nestačili ľudské sily na výpočtovú silu počítačov. Hoci zápas Kasparov-Deep Blue niektorí spochybňovali, v roku 2000 sa hral v nemeckom meste Lippstadt silne obsadený turnaj (8 veľmajstrov z 12 účastníkov), ktorý prekvapujúco vyhral šachový program PConners (9 bodov z 11). Bolo to po prvýkrát v histórii, čo šachový program zvíťazil v silnom uzavretom turnaji [6]. Viac o algoritmoch a technikách, ktoré *DeepBlue* a iné dnešné programy používajú, si povieme v kapitole 2.3.

1.1.1 Časová kontrola partie

Časová kontrola partie, anglicky *time control*, určuje čas, ktorý majú hráči k dispozícii na ukončenie partie. Ak niektorému z hráčov vyprší čas, partiu automaticky prehráva. Existujú pravidlá, podľa ktorých sa partia končí remízou aj napriek vypršanému času, avšak tieto prípady sú len ojedinelé. Tri základné rozdelenia podľa časovej kontroly sú tzv. bleskovky, rapid a klasická časová kontrola.

Bleskovky, anglicky *blitz*, sú partie, v ktorých majú obaja hráči 5 alebo menej minút na dokončenie partie. Bleskové partie sú populárne hlavne na internetových šachových serveroch a sú divácky atraktívne. Hlavným lákadlom je v tomto prípade rýchly priebeh partie a menšie šance pre skončenie partie remízou. Klasická časová kontrola je používaná vo väčšine turnajov s malými rozdielmi. Vyznačuje sa časom na ukončenie partie dlhším ako 1 hodina. Častokrát je rozšírená o ďalší pridaný čas po odohratí určitého počtu ťahov, napr. po odohratí 40 ťahov sa obom hráčom pridáva hodina k ich aktuálnemu zostávajúcemu času. Tento formát je používaný v takmer všetkých šampionátoch a jedna partia trvá častokrát dlhšie ako 5 hodín. Za rapid partie sa neformálne považujú partie s viac ako 10 minútami a menej ako 60 minútami pre hráčov. Práve rapid časovú kontrolu sa snažia vrcholoví šachoví hráči presadiť do turnajov. Poskytuje ideálnu rovnováhu medzi bleskovkami, ktoré sú až na škodu chaotické a rýchle, a medzi klasickou časovou kontrolou.

Všetky časové kontroly zdieľajú možnosť tzv. inkrementu, teda pridanie určitého času na hodiny v partii za každý odohratý ťah. V bleskových partiách sa inkrementuje najčastejšie 1 sekunda za každý ťah, v rapid partii je to väčšinou 10 sekúnd a v klasickej časovej kontrole 45 sekúnd za odohratý ťah.

1.2 Vybrané varianty šachu

Časom začali vznikať rôzne varianty šachu, populárne dodnes. Dôvody sú rôzne — v niektorých variantoch je hlavným motívom čisto zábava, v iných ako Chess960 je to pokus o odtrhnutie sa od klasického šachu, ktorý je v dnešnej dobe dominovaný programami. Podstatou variantov je napríklad zmena pravidiel, iná štartovacia pozícia, pridanie pravidiel či varianty pre viac hráčov a iné. O niektorých z nich si povieme si o nich viac v nasledujúcich podkapitolách [10].

1.2.1 Bughouse šach

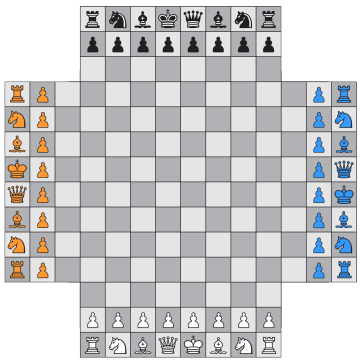
Bughouse, niekedy nazývaný aj *holanďania*, je variant, ktorému sa budeme venovať v tejto práci. Štyria hráči hrajú na dvoch šachovniciach v tíme po dvoch, vždy jeden hráč s bielymi a druhý s čiernymi figúrami. Pravidlá sú obohatené o tzv. *drop* ťah, každá vyhodená figúrka súpera sa posunie spoluhráčovi do tzv. zásobníka a ten ju môže položiť na prázdne políčko na svojej šachovnici na svojom ťahu, prípadne si ju odloží na viditeľné miesto pre neskoršie použitie. *Drop* ťahy majú svoje obmedzenia, pešiakov nemôžeme položiť na ôsmu radu ak hráme bielymi figúrkami resp. prvú radu ak hráme čiernymi. Takto položení pešiaci môžu dosiahnuť promóciu, ale po ich vyhodení sa vracajú spoluhráčovi ako pešiaci. Hra končí matom na jednej z dvoch šachovníc. Veľmi zaujímavý je tu element času — ponúka možnosť čakať na dôležitú figúrku od spoluhráča, ktorá je potrebná k matovému útoku. Spoluhráč má možnosť vyhodiť žiadanú figúrku aj za cenu materiálnej nevýhody na vlastnej šachovnici.

1.2.2 Chess 960

Dnes asi najznámejší variant, ktorý má ako jediný reálny potenciál nahradiť klasické pravidlá šachu. Inak pomenovaný aj Fischerchess, tento variant zdedil meno po svojom tvorcovi Robertovi Fischerovi, ktorý bol jedným z najlepších hráčov šachu v histórii. Fischer ho predstavil v 19. júna, 1996 v Buenos Aires a od svojich počiatkov sa tešil veľkej popularite. Dodnes sa uskutnočňujú turnaje v tomto variante a veľa z najlepších hráčov šachu sa snaží pretlačiť tento variant ako hlavný.

Rozdielom je rozostavenie figúr na šachovnici. Pravidlá dovoľujú upraviť rozmiestnenie figúr na prvom rade do 1 z 960 rôznych povolených počiatočných pozícií podľa náhodného žrebovania. Figúry čierneho hráča potom zrkadlia rozmiestnenie bielych figúr, ako je vidieť na obrázku 1.2.

Obr. 1.2: Ukážka základnej pozície variantu Chess960. Na obrázku je zobrazená jedna z 960 možných pozícií, prvá a posledná rada sú odzrkadlené podľa pravidiel.



Obr. 1.3: Ukážka začiatočnej pozície šachu pre 4 hráčov. Variant má viacero podvariantov, medzi inými aj pre 2 tímy, kde ide o spoluprácu, či podvariant bez tímov.

1.2.3 Horde chess a Endgame chess

Oba varianty menia počet figúr na šachovnici. Horde je variant, v ktorom má jeden hráč 36 pešiakov a druhý 16 figúr, ktoré sú asymetricky postavené. Endgame chess zase redukuje počet figúr každého hráča na kráľa a 8 pešiakov.

1.2.4 Varianty netradičných šachovníc

Hlavným rozdielom je netradičná hracia plocha, sem patrí napr. šach pre 4 hráčov na jednej šachovnici ako na obr. 1.3, v ktorom hráči ťahajú v smere hodinových ručičiek či rôzne varianty hrané na hexagonálnych šachovniciach. Zaujímavý je aj nekonečný šach, variant v ktorom nemá šachovnica ohraničenie.

1.2.5 Crazyhouse a atomic chess

Varianty používajúce klasickú šachovnicu a zmenené pravidlá. Crazyhouse je obdoba bughouse variantu, ale je určený pre dvoch hráčov. Podobne ako pri bughouse, hráči môžu položiť vyhodené figúrky na prázdne políčko ako svoj ťah. Hlavný rozdiel medzi bughouse a crazyhouse je pôvod figúr, ktoré môžu byť *dropped*. V bughouse nám

tieto figúry podáva spoluhráč, ktorý ich získal vyhodením na jeho šachovnici, naopak v crazyhouse variante je iba jedna šachovnica a figúry, ktoré potenciálne budú použité na *drop* ťah získavame my, vyhodením figúr súpera. Preto musí byť k dispozícii náhradný set figúr alebo musia byť tieto figúry označené na reprezentovanie správnej farby (keď vyhodíme figúru opačnej farby, získame ju, ale je nelogické položiť na šachovnicu figúru, ktorou disponuje súper). Výskum a programy crazyhouse variantu sú dôležité pri analýze bughouse programov, sú vďaka pododbným pravidlám úzko späté. Atomic šach pri vyhodení figúry vyhadzuje aj všetky figúry okrem pešiakov v okolí (teda 8 políčok okolo vyhodenej figúry).

Kapitola 2

Programy hrajúce šach a bughouse

Na začiatku si predstavíme dôležité pojmy potrebné na pochopenie problémov, pozrieme sa na aktuálny stav šachových programov, vysvetlíme ako sa meria ich sila. Uvedieme porovnanie najsilnejších šachových veľmajstrov a programov a bližšie sa budeme venovať algoritmom, ktoré tieto programy používajú.

2.1 Základné pojmy v šachovom programovaní

ELO rating definuje relatívnu silu hráča v šachu či iných hrách.

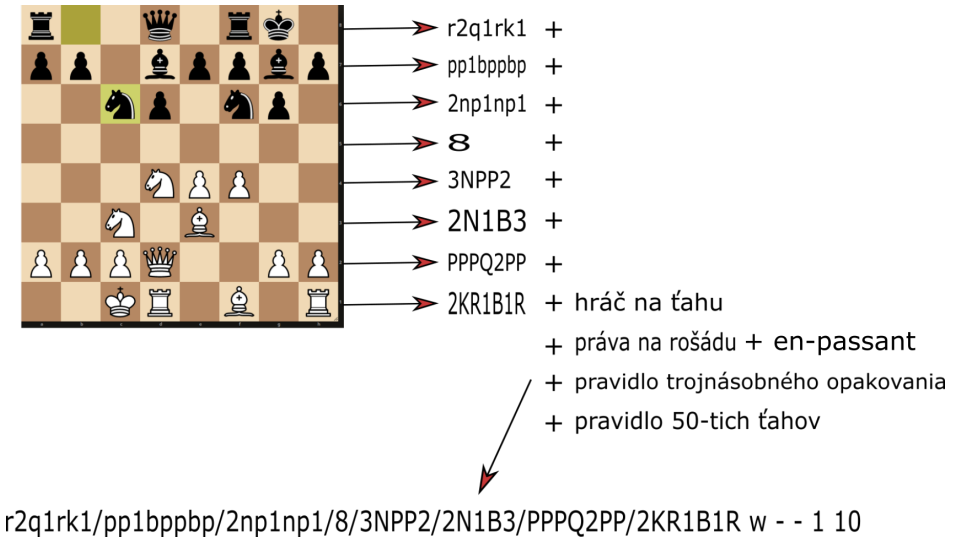
Ten definoval fyzik *Arpad Elo* maďarského pôvodu. Snaží sa o popísanie sily hráčov v hrách s nulovým súčtom, ktorých definícia je uvedená ako doslovný citát [18].

> Nulový súčet v teórii hier a v ekonomickej teórii opisuje situáciu, v ktorej každý zisk alebo strata jedného účastníka je vyvážená ziskom alebo stratou iného hráča, teda, ak sa celkové zisky a straty všetkých účastníkov sčítajú, výsledok bude nula.

Keďže hra šachu môže končiť iba výhrou, prehrou alebo remízou, má nulový súčet. ELO najlepších šachistov v histórii v klasickej časovej kontrole sa pohybuje okolo 2850, ELO priemerného hráča šachu je 1500. Zvyšuje sa výhrou, znižuje prehrou hráča. Rozdiel dvoch ELO ratingov hráčov predpokladá približný výsledok hry medzi dvoma hráčmi.

Polťah definuje jeden ťah jedného hráča v partii šachu. Pretože v šachu znamená ťah potiahnutie oboch hráčov, je potrebné zadefinovať potiahnutie jedného hráča, teda spomenutý *polťah*.

Faktor vetvenia, anglicky *branching factor*, označuje počet možných polťahov, ktoré sú k dispozícii hráčovi na ťahu. Tento počet sa počas rôznych častí šachovej partie mení a práve preto sa väčšinou pre šach ale aj iné hry vyjadruje priemerom. Pre šach je priemerný faktor vetvenia 35, teda v priemernej partii má hráč, ktorý je na ťahu možnosť výberu 1 z 35 ťahov [3].



Obr. 2.1: Vysvetlenie jednotlivých častí FEN zápisu, ich vzniku a významu na príklade.

Strom hry, anglicky *game tree*, je orientovaný graf, ktorý reprezentuje vývoj šachovej partie. Jednotlivé uzly popisujú pozície a hrany predstavujú všetky dostupné polťahy. Počet jeho listov je počtom dosiahnuteľných, rôznych priebehov hry.

Forsyth-Edwards notácia, skrátene FEN, je štandard, pomocou ktorého sa zapisuje šachová pozícia. Tento zápis poskytuje všetky potrebné informácie o stave šachovnice a taktiež poskytuje doplnkové informácie kvôli pravidlu trojnásobného opakovania a pravidlu 50-tich ťahov. Bližšie ho popisujeme na obrázku 2.1. Nadstavbou FEN štandardu pre bughouse šach je BFEN notácia, ktorú pre zápis bughouse pozícií používajú viaceré programy.

2.2 Súčasný stav

Už v úvode sme spomínali ideu tvorby našeho bughouse programu. Aby sme bližšie objasnili naše rozhodnutie pre vybranú architektúru, pozrime sa na všeobecnú architektúru šachových programov.

2.2.1 Existujúce bughouse programy

Práve *faktor vetvenia* a komplexnosť bughouse variantu zabraňuje tvorbe kvalitnejších programov. *Sjeng*, *Sunsetter* a *TJChess* sú najznámejšie programy variantu bughouse, avšak všetky spomenuté sú určené pre viaceré hranie viacerých variantov, medzi ktorými prevažuje okrem bughouse aj crazyhouse [16, 2, 11].

Prvým pokusom o vytvorenie programu hrajúceho bughouse bol *Sjeng* v roku 1999. Nasledoval *Sunsetter* v roku 2000 a *TJChess*, 2006. Vývoj všetkých troch programov bol dávnejšie pozastavený. Avšak už vtedy si tvorcovia uvedomovali zložitosť prog-

ramovania bughouse a crazyhouse variantu. Všetky tri programy pri hre komunikujú vlastným protokolom a snažia sa využiť čas na získanie výhody v šachovej partii. Luke Nikolas a kol. sa v roku 2011 sa úspešne pokúsili o implementáciu bughouse variantu do programu pre klasický šach Mediocre [15]. Narazili na podobné problémy ako predošlé programy, avšak podarilo sa mu implementovať program, ktorý porazí priemerných šachistov a ukázal, že algoritmy použité pri tvorbe klasických programov sú aplikovateľné aj pri tvorbe bughouse programu.

2.2.2 Spoločné črty crazyhouse a bughouse šachu

Významným prínosom bola práca autorov *Sacha Droste a Johannes Furnkranz*, ktorí v roku 2008 riešili evaluáciu figúr pre tri varianty šachu, crazyhouse, atomic a suicide [9].

V roku 2019 sa publikovali práce autorov *Czech a kol., Chi*, ktorí s využitím neurónových sietí vytvorili crazyhouse programy [5, 7]. V prípade prvého menovaného sa podarilo zorganizovať zápas proti neoficiálne najlepšiemu crazyhouse hráčovi za rok 2017, ktorý vytvorený program vyhral 4-1. Tieto programy využívali podobné optimalizačné techniky a algoritmy ako programy hrajúce klasický šach.

2.3 Vybrané používané algoritmy

Základným stavebným kameňom algoritmu používaného na generovanie ťahov takmer v každom šachovom programe tvorí algoritmus minimax (prípadne nejaká upravená verzia minimax algoritmu). Ten je založený na prehľadávaní šachovej pozície do hĺbky a keďže šach je hra s nulovým súčtom, ako sme definovali v kapitole 2.1, maximalizujeme naše šance a minimalizujeme šance súpera na výhru. Následne listy takto vygenerovaného stromu evaluuje funkcia, ktorá každý list resp. pozíciu ohodnotí. V ďalších kapitolách si viac povieme o nadstavbách minimax algoritmu, evaluačnej funkcii a optimalizáciách, ktoré sa používajú.

Tieto nadstavby ako *alpha-beta*, *tiché prehľadávanie* či *tabuľka transpozícií* vynechávajú listy alebo niektoré vetvy, čím sa zmenšuje počet evaluovaných pozícií. Samotný minimax bez optimalizácií by v hĺbke n polťahov v klasickom šachu vytvoril približne 30^n listov. Pri bughouse je faktor vetvenia priemerne 200, práve kvôli možnosti *drop* ťahov, ktoré ho výrazne zvyšujú. Optimalizovanie je teda vhodné pri oboch variantoch, úspešnejšie je v prípade klasického šachu. Stockfish, najlepší program pre klasický šach, má tento faktor priemerne 2, teda pre hĺbku n bude evaluovať približne 2^n pozícií.

2.3.1 Reprezentácia šachovnice

Definovanie pravidiel a reprezentovania šachovnice je prvým krokom pri tvorbe šachového programu. V súčasnosti sa používajú dve rozdielne reprezentácie, matica 10×12 a bitboards, každá má svoje výhody a nevýhody. Pri reprezentácii šachovnice je dôležité uchovávať nielen pozície figúr, ale aj iné informácie o hre. Medzi tie patrí napr. počet odohratých ťahov, možnosť rošády pre oboch hráčov a históriu ťahov. História je dôležitá najmä kvôli pravidlu o trojnásobnom opakovaní pozície a pravidlu 50-tich ťahov.

V nasledujúcich podkapitolách sa budeme venovať rozdielom a výhodám týchto reprezentácií.

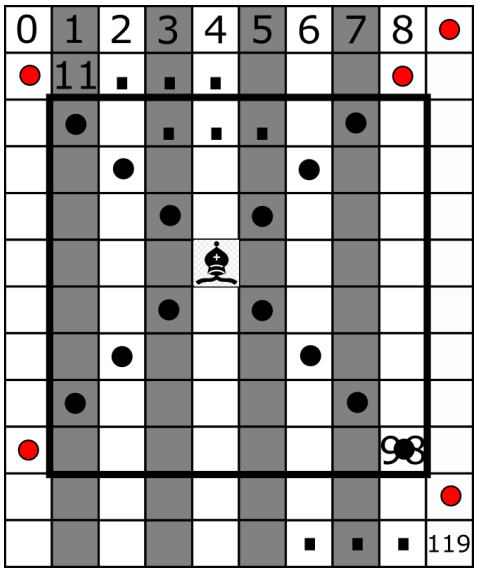
Bitboards

Šachovnicu reprezentuje pomocou polí bitov dĺžky 64. Pre každý typ figúry je vytvorené pole, v ktorom každý bit označuje jedno pole na šachovnici. Ak je niektorý bit nastavený na 1, nachádza sa na ňom figúrka daného typu, ak je nastavený na 0, políčko je prázdne (prípadne obsadené iným typom figúry). Hlavnou výhodou bitboards je možnosť využitia rýchlych bitových operácií OR, NOT, XOR, AND a posúvania bitov. Pomocou posúvania bitov sa počítajú možné ťahy figúr, útoky na kráľa a iné informácie o partii. Pri takto vytvorených poliach nevieme explicitne povedať ktorá figúrka obsadzuje dané pole, ale vieme to ľahko zistiť kombináciou boolovských binárnych operácií na správnych poliach.

Pole 10x12

V tomto prípade ide o reprezentáciu šachovnice maticou 8x8, ktorá je zhora a zdola rozšírená o dva riadky a z oboch strán o jeden stĺpec. Rozšírenie o stĺpce a dva riadky je dôležité kvôli pohybu jazdca po šachovnici. Pri intuitívnej reprezentácii 8x8 nastáva problém generovania ťahov, je potrebné kontrolovať pozíciu figúr po každom ťahu. Pri upravenej verzii, 10x12 stačí kontrolovať indexy po generácii ťahov a následne odstrániť nemožné ťahy, ktoré by skončili mimo šachovnice, ako bližšie popisujeme na obrázku 2.2. V ročníkovom projekte venujúcemu sa totožnej téme sme využili objektovo-orientovanú funkcionalitu ponúkanú prog. jazykom Java a maticovú reprezentáciu. To sa však javí ako neoptimálne a prehľadávanie listov v takto generovanom strome je už pri hĺbke 8 polťahov nepraktické.

Druhou možnosťou je využiť nejaký existujúci bughouse program pre klasický šach a nadstavbou ho upraviť tak, aby hral bughouse variant, ako popisujeme v kapitole 4.1. Ideálne v tomto prípade je využiť najsilnejší program v jazyku Java, teda Bagatur [14]. Ten šachovnicu reprezentuje pomocou bitboards.



Obr. 2.2: Číslovanie pri reprezentácii poľom a potenciálne ťahy strelcom. Nelegálne ťahy zvýraznené červenou farbou.

2.3.2 Evaluačná funkcia

Úloha evaluačnej funkcie je akési číselné aproximovanie šachovej pozície, ktorú jej odovzdáme. Toto reálne číslo je kladné, ak je pozícia lepšia pre bieleho hráča, záporné ak pre čierneho, 0 ak je vyrovnaná a väčšinou obmedzené na niekoľko desatinných miest.

Zo šachového hľadiska má každá pozícia vlastnosti, podľa ktorých ju vieme objektívne ohodnotiť. Ak má biely hráč o dámu viac ako čierny, vie obvykle doviesť hru do víťazného konca. Najjednoduchším indikátorom ohodnotenia pozície je teda počet figúr na oboch stranách, medzi zložitejšie patria bezpečnosť kráľa, pešiaková štruktúra, sila možného útoku a iné, základné ohodnotenie figúr je na obrázku 2.3. Je dôležité spomenúť, že evaluačná funkcia staticky hodnotí iba danú pozíciu a nie možné ťahy. Nebuduje si teda vlastný strom, len zo šachového hľadiska (bez počítania ťahov) ohodnotí pozíciu. Evaluácia aspektov šachovej pozície je často udávaná v tzv. *centipawn* jednotkách (stotina hodnoty pešiaka), 100 *centipawn* jednotiek je teda rovných výhode o jedného pošiaka. Ak je teda napr. ohodnotenie pozície +2, biely hráč má poziční výhodu dvoch pešiakov resp. 200 *centipawns*.

Evaluačná funkcia je jednou z hlavných častí každého šachového programu, jej kvalita nepriamo indikuje aj kvalitu programu. Kým minimax a jeho optimalizácie zabezpečujú tvorbu a odstránenie niektorých vetiev a listov stromu a napomáhajú menšiemu počtu evaluácií vo výpočte, bez kvalitnej evaluačnej funkcie nemajú zmysel, pretože nesprávne ohodnotenie pozície znehodnocuje predošlé výpočty.

Obr. 2.3: Ukážka jednoduchého ohodnotenia figúr v evaluačnej funkcii. Ak takéto ohodnotenie priradíme rôznym aspektom pozície, dostaneme jej kvantitatívne ohodnotenie.

Evaluačná funkcia drop ťahov v bughouse programoch

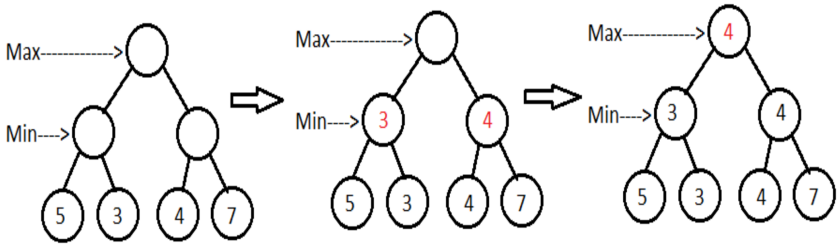
Evaluačná funkcia v bughouse šachu musí byť rozšírená aj o ohodnotenie figúr, ktoré sú k dispozícii hráčom pre *drop* ťahy. Výsledky práce autorov Sacha Droste a kol. o evaluácii takýchto figúr naznačujú, že hodnota pešiakov, jazdcov a veže sa zvyšuje, kým hodnota strelca a dámy klesá [9]. V tabuľke 2.1 je ukážka základnej evaluácie figúr v najznámejších bughouse programoch, hodnoty sú udávané v jednotkách *centipawn*.

Figúra	Sjeng	Sunsetter
Pešiak	100	100
Veža	250	200
Strelec	230	195
Jazdec	210	192
Dáma	450	390

Tabuľka 2.1: Ohodnotenie figúr v zásobníku (ktoré sú k dispozícii pre drop ťahy) rôznych bughouse programov

Evaluačná funkcia postavenia figúr v bughouse programoch

Evaluácia bughouse postavenia figúr na šachovnici je rozdielna od evaluácie klasickej pozície. Veľký vplyv na evaluáciu má zvýšená dynamika a spád hry, dôraz v statickej evaluácii je na okamžitý útok a zvýšenú bezpečnosť kráľa na oboch stranách. Ako príklad poskytujeme základnú evaluáciu pozície *drop* ťahu bielym pešiakom programu Sjeng v tabuľke 2.2. Evaluácia programom Sjeng priamo podporuje útočné *drop* ťahy, ktoré smerujú na čierneho kráľa, ako je vidieť na zvyšujúcich sa *centipawn* hodnotách priradeným okoliu čierneho kráľa a zanedbáva stred šachovnice.



Obr. 2.4: Ukážka priebehu minimax algoritmu. Pri výbere sa začína v listoch, úloha minimalizéra je vybrať pozíciu s najmenším ohodnotením, maximalizéra s najväčším ohodnotením. Ak sú všetky rozhodnutia v danej hĺbke ukončené, úlohy sa vymenia a hĺbka zníži. Keď sa takto vyplní celý graf od listov ku koreňu, máme „cestu“, ktorá vedie k optimálnemu ťahu.

0	0	0	0	0	0	0	0
25	25	35	5	5	50	45	30
0	0	0	7	7	5	5	0
0	0	0	14	14	0	0	0
0	0	10	20	20	10	5	5
12	18	18	27	27	18	18	18
25	30	30	35	35	35	30	25
0	0	0	0	0	0	0	0

Tabuľka 2.2: Ohodnotenie pešiaka k dispozícii pre drop ťahy programom Sjeng

2.3.3 Minimax

V predchádzajúcich podkapitolách sme si povedali o tvorbe stromu, evaluačnej funkcii a reprezentácii šachovnice, ale nespomínali sme algoritmus, ktorého úlohou je vybrať finálny ťah zo stromu hry. Každá pozície v strome sa ohodnotí evaluačnou funkciou. Hlavnú myšlienku algoritmu minimax vieme zhrnúť nasledovne — my sa snažíme maximalizovať naše šance na výhru a minimalizovať šance súpera, pričom predpokladáme, že v každej pozícii bude zahraný najsilnejší ťah. Ako vysvetlenie si môžeme si predstaviť situáciu ako na obrázku 2.4. Cestou od listov má náš súper možnosť vybrať ťah, prirodzene si nevyberie ťah, ktorý nám dáva výhodu, ale vyberie si ťah, ktorý dáva výhodu jemu. Celé rozhodovanie je krok po kroku znázornené.

2.3.4 Alfa–beta

Alfa-beta je prvá optimalizácia minimaxu. Jej úlohou je odstrániť niektoré vetvy, do ktorých sa určite nedostaneme, pretože súper si ich nevyberie. Navyše, nestratíme pri

nej žiadne ťahy, ktoré by boli lepšie ako najlepší ťah nájdený použitím minimaxu s touto optimalizáciou. Alfa-beta si udržiava dve hodnoty, α a β, označujúce evaluáciu doteraz najhoršieho resp. najlepšieho nájdeného ťahu. Tieto hodnoty sa prehľadávaním stromu menia a s ich pomocou sa zo stromu odstraňujú niektoré jeho vetvy.

Ako presne teda táto optimalizácia odstraňuje vetvy? Pre objasnenie si strom hry si zadefinujme na hĺbke 2. Teda pozostáva zo všetkých ťahov, ktoré môžeme vykonať a všetkých odpovedí súpera na tieto ťahy. V doteraz evaluovaných pozíciách sme našli najlepší ťah, ktorý zaručuje pozíciu, kde máme o jednu vežu naviac. Evaluujeme teda náš ďalší ťah a odpovede súpera na tento ťah. Nech v tejto vetve existuje odpoveď protihráča, ktorá mu zaručuje vyrovnanú pozíciu. Môžeme teda ignorovať evaluovanie ostatných odpovedí súpera na náš ťah v tejto vetve, pretože táto pozícia je horšia, ako doteraz najlepší nájdený ťah. Ak by aj existovala odpoveď súpera v tejto vetve, ktorá vyústi do pozície, kde máme o dámu viac, je zbytočná, pretože súper zahrá vždy najlepší ťah. V našom prípade si protihráč vyberie vyrovnanú pozíciu než by si vybral tú, kde má o dámu menej (podotýkame, súper minimalizuje).

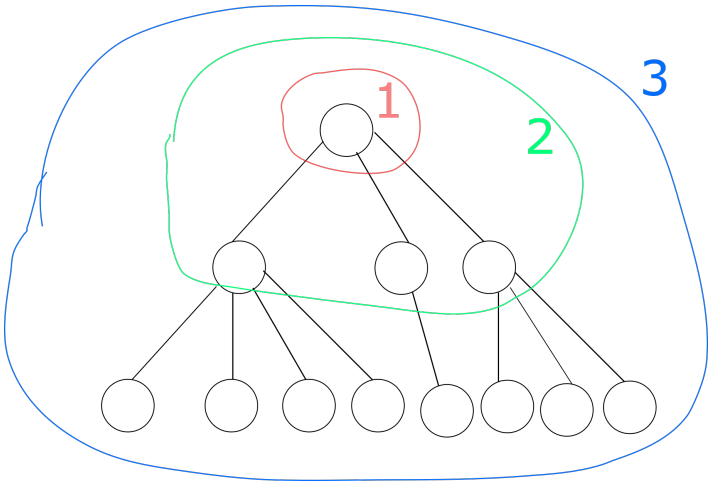
Celá optimalizácia je teda založená na pozorovaní, že ak je nájdený ťah horší, ako doteraz najlepší ťah, nie je potrebné evaluovať zbytok vetvy, pretože bol nájdený ťah, ktorý je pre protihráča výhodnejší ako doteraz najlepší nájdený ťah. Potenciálny problém nastáva, ak je najlepší ťah nájdený na konci stromu. V tom prípade by bol evaluovaný celý strom a nenastali by tzv. *alpha* alebo *beta zrezania*, teda neprehľadávania určitých vetiev, čím by bola optimalizácia zbytočná a neodlíšiteľná od minimaxu. Je preto vhodné usporiadať ťahy podľa statickej evaluácie pred vstupom do alfa-bety.

2.3.5 Databázy otvorení

Ďalšou vhodnou optimalizáciou sa javí nerobiť evaluáciu prvých ťahov na začiatku partie, ale použiť databázu. Keďže prvé ťahy partie sa opakujú, existuje veľa databáz, ktoré si uchovávajú možné ťahy vo vetvách a percentuálnu úspešnosť oboch strán na výhru a remízu podľa odohratých partií v danej vetve. To uľahčuje výber ťahu a maximalizuje šancu na výhru. Do databáz sa kontinuálne pridávajú odohraté hry. Väčšina vetiev v takejto databáze si udržuje štatistiku do 25. polťahu [1].

2.3.6 Prehľadávanie do šírky a usporiadanie ťahov

Prehľadávanie do šírky a *usporiadanie ťahov* sú úzko prepojené a ich výkon sa zvyšuje, ak sú používané spolu. Prehľadávanie do šírky sa týka postupného generovania a prehľadávania stromov od hĺbky 1 až po hľadanú hĺbku. Teda v prvom strome budú analyzované iba všetky možné ťahy v aktuálnej pozícii, v druhom už pribudnú aj odpovede súpera, atď. až kým nenarazíme na požadovanú hĺbku. Takéto postupné zvyšovanie hĺbky stromu poskytuje aj istý typ garancie. Pri prehľadávaní stromu do hĺbky



Obr. 2.5: Na obrázku je strom, ktorý vznikol prehľadávaním do šírky.

sa častokrát stáva, že strávime podstatne viac času na ťahu, ako je potrebné v danom úseku partie. Keďže šach je hra, v ktorej majú hráči časový limit, ich úlohou je aj strážiť si čas strávený premýšľaním na každom ťahu.

Postupné prehlbovanie zaručuje, že ak aj prekročíme časový limit pre daný ťah a neevaluovali sme celý strom, máme k dispozícii najlepší ťah v strome hĺbky o 1 menej. Ak by prehľadávanie bolo ukončené a celý strom nebol evaluovaný bez použitia prehľadávania do šírky, nemáme garanciu že doteraz najlepší nájdený ťah je dostatočne dobrý, pretože sme neevaluovali niektoré vetvy.

Usporiadanie ťahov je zložitá optimalizácia, preto sa jej v tejto práci budeme venovať len krátko. Ide o zadefinovanie poradia ťahov pri evaluácii. Ideálne je analyzovať vetvy, ktoré majú najväčší prospekt na výhru. Kombinácia s prehlbovaním je úspešná, pretože môžme každou ďalšou iteráciou analyzovať najskôr najdôležitejšie vetvy stromu. Tým sa hovorí aj principiálne vetvy, ang. *principal variation* alebo *PV*.

2.3.7 Tabuľka transpozícií

Tabuľka transpozícií je ďalšou vhodnou optimalizáciou. Pri prehľadávaní stromu hry sa častokrát stáva, že narazíme na rovnakú pozíciu. To sa môže stať napríklad prehodením dvoch alebo viacerých ťahov, ktoré vedú k danej pozícii. Tabuľka transpozícií slúži na uchovávanie informácií o doteraz evaluovaných pozíciach pomocou Zobristovho hašovania. Kolízie pri 64-bitovom haši je vhodné očakávať až pri 4 biliónoch evaluovaných pozíciach, čo je rádovo viac ako počet evaluovaných pozícií v priemernej partii najsilnejších programov.

2.3.8 Tiché vyhľadávanie

Posledná optimalizácia, ktorú si spomenieme, je tiché vyhľadávanie. Princíp si ukážeme pri evaluácii stromu hĺbky 1, teda sa pozrieme iba na naše ťahy v aktuálnej pozícii. Ak by sme napr. zobrali dámou súperovho strelca, analýza tejto pozície nám povie, že vyhrávame o strelca. To však môže byť problém, ak má súper ako odpoveď možnosť zobrať našu dámu pešiakom. Teda sme vymenili dámu za strelca, čo je veľmi nevýhodné, ale evaluácia to neodhalila. Preto sa v mnohých šachových programoch po ukončení prehľadávania hlavného stromu a výbere ťahu ešte spustí dodatočné, krátke prehľadávanie začínajúce v pozícii vybraného ťahu. Poskytuje garanciu toho, že súper nemá okamžitú odpoveď na náš ťah, ktorá mu poskytne výhodu, ktorú pôvodné vyhľadávanie neodhalilo.

Kapitola 3

Úprava bughouse pravidiel

Na začiatku kapitoly si rozoberieme problémovú definíciu bughouse pravidiel z pohľadu tvorby programov ale aj odohrania partie a následne navrhneme konzistentné, upravené pravidlá, ktorými sa budeme riadiť pri tvorení programu v kapitole Implementácia. Poslednou témou v tejto kapitole bude generalizácia bughouse hry.

3.1 Analýza pravidiel v partii bez časovej kontroly

Klasický šach, podobne ako väčšina stolných hier, nezávisí na reálnom čase, t.j. je možné hrať partie aj bez časovej kontroly. Dôležité je však pravidlo, ktoré hovorí, že hráč na ťahu má povinnosť eventuálne urobiť nejaký ťah. Nie je tomu inak ani v bughouse šachu, vyššie uvedené pravidlo platí pre každú šachovnicu: hráč, ktorý je na ťahu, eventuálne musí ťah vykonať, aj v momente, kedy nechce (napr. ak po vykonaní ťahu určite dostane mat).

Pre potreby vysvetlenia problému, ktorý takto definované pravidlá tvoria (podotýkame, vzniká pri partii bez časovej kontroly) v bughouse šachu, si označme šachovnice číselne, 1 a 2, a hráčov W a B, teda napr. hráč bielymi figúrkami na prvej šachovnici by mal označenie $1W$. Teda prvý tím tvoria hráči $1W$, $2B$ a ich súperom je tím zložený z hráčov $1B$, $2W$. Na šachovniciach nastane nasledovná situácia: na ťahu je $1B$ a $2B$, ale po vykonaní ťahu dostane hráč istý mat od hráča $1W$, resp. $2W$.

Vznikla symetrická situácia, v ktorej je nevýhodné potiahnuť pre oboch hráčov. Ak potiahne $1B$, jeho tím prehrá. V opačnom prípade, ak ťah vykoná prvý hráč tímu súpera, $2B$, dostane mat a partia sa skončí výhrou pre tím hráčov $1B$ a $2W$.

V klasickom šachu táto situácia nenastáva a je potrebné zdôrazniť, že v bughouse šachu prichádza k symetrii hneď v prvom ťahu. Žiadnemu z hráčov na ťahu sa v princípe neoplatí svoj ťah uskutočniť, lebo poškodzuje svoj tím (druhý tím má viac informácií a má možnosť výberu, na ktorej šachovnici urobí ťah).

Ľudia hrajú bughouse častokrát bez znalosti tohto dôležitého pravidla. Hráč, ktorý

urobí ťah v momente, keď nemusí, nevedome poškodzuje vlastný tím. Pokiaľ by bughouse hrali 4 optimálni hráči, žiaden z nich neťahá skôr, ako musí. Preto je vhodné upraviť pravidlá tak, aby boli definované nezávisle na čase a jasne zadefinovať pojem *tímový ťah*.

3.1.1 Ťah tímu

Ako sme spomínali, bughouse pravidlá neurčujú ťah tímu, ale sú definované ťahmi hráčov na jednotlivých šachovniciach. Každá šachovnica je definovaná pozíciou, zásobníkom figúr, a aktuálnym hráčom na ťahu. Práve kvôli problému vznikajúcej symetrie v partii bez časovej kontroly a problému predčasného potiahnutia je potrebné povinnosť ťahu zadefinovať inak. Uvažujme povinnosť uskutočnenia ťahu nasledovne:

povinnosť uskutočniť ťah sa vzťahuje na tímy a nie na jednotlivých hráčov na šachovniciach. Po odohraní ťahu sa táto povinnosť prenáša na súpera a platí, že tím si vyberie, na ktorej šachovnici bude konkrétny ťah odohraný.

3.1.2 Definovanie pravidiel nezávislých na čase

Vytvorené bughouse programy majú okrem spomenutých aj iný problém — ich architektúra je tvorené dvoma, prípadne n šachovnicami, ktoré sú nezávisle od seba ovládané programom. Šachovnice si posielajú informácie o aktuálnej pozícii počas prehľadávania alebo čakania na ťah súpera, ale táto komunikácia je náročná a znižuje kvalitu programov. Dôsledkom komplikovanej komunikácie je zbytočné zameranie sa programov na správnu úsporu času. Podobne ako v prípade ľudí, aj programy uprednostňujú odohratie rýchlych ťahov a neberú do úvahy, či je vôbec vhodné ťah odohrať. Je pravdepodobné, že tvorcovia programov nepoznajú pravidlá hry, ktorú programujú.

3.1.3 Generalizácia bughouse hry

Konfigurácia stavu hry je teda presne definovaná oboma (prípadne viacerými) šachovnicami a tímom, ktorý si vyberie, na ktorej šachovnici bude odohraný konkrétny ťah. Odstránenie elementu času je vhodným pokračovaním definovania pravidiel, pri upravenej definícii čas nehrá podstatnú úlohu.

Takto definované bughouse pravidlá umožňujú generalizáciu bughouse hry pre $2n$ hráčov, hrajúcich na n šachovniciach. V takom prípade je prirodzené, aby mal jeden tím všetky figúrky biele a druhý všetky figúrky čierne. Keďže ťah na šachovnici môže tím vykonať bez ohľadu na to, ktorý hráč by bol podľa povôdných pravidiel na ťahu na danej šachovnici (teda prvý ťah na šachovnici môže byť aj čiernymi figúrkami), farba figúriek stráca zmysel. Posledná úprava pravidiel je zdieľanie figúr medzi šachovnicami. V bughouse šachu pre 4 hráčov je logické, aby sme posúvali vyhodené figúry

spoluhráčovi. Avšak pri generalizácii bughouse pre $2n$ hráčov je prirodzené, aby boli figúry v zásobníku k dispozícii pre celý tím (samozrejme, okrem šachovnice, z ktorej figúra pochádza).

Kapitola 4

Implementácia

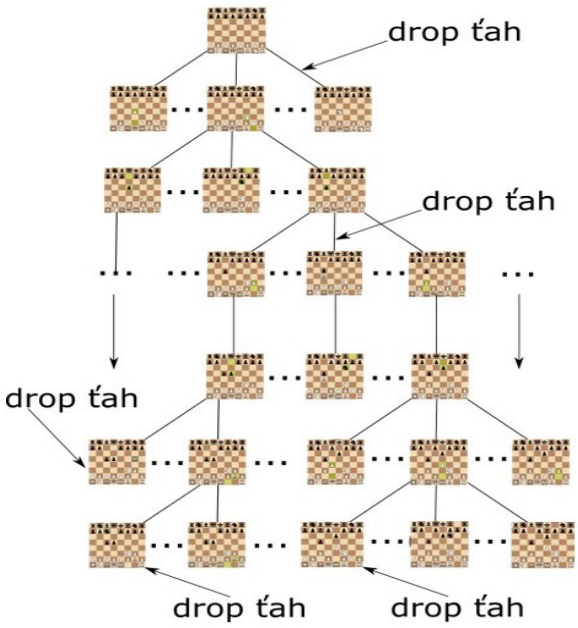
V tejto kapitole si postupne odôvodnime rozhodnutia pri výbere architektúry, objasníme využitie evaluácie klasického programu, komunikáciu prostredníctvom protokolu *UCI* a samotnú implementáciu.

4.1 Klasická bughouse architektúra

Klasická architektúra bughouse programov má svoje limity. Rádovo vyšší *faktor vetvenia* pri bughouse hre naproti klasickému šachu neumožňuje hĺbkové prehľadávanie a nepriamo sa podieľa na nedokonalosti bughouse programov. Časté chyby sú napr. nesprávna úspora času, komunikácia medzi šachovnicami v programe či zlý odhad pozície. Problém zlého odhadu pozície spočíva hlavne vo väčšej dynamickosti vyplývajúcej z pravidiel bughouse šachov a definovaní hodnôt figúr, ktoré sú k dispozícii na drop ťah. Tieto hodnoty sa líšia v závislosti na šachovej pozícii a možnostiach, ktoré ponúka. Medzi problémy taktiež patrí aj robenie predčasných ťahov a veľký dôraz na úsporu času. Nami navrhnuté pravidlá v 4.2.2 majú za úlohu zamerať sa na samotnú bughouse hru a upustiť od časom orientovanej architektúry. Strom bughouse hry pri priemernom faktore vetvenia a hĺbke 5 pozostáva z približne 200^5 pozícií, čo je stále rádovo viac pozícií ako evaluuje Stockfish program pre klasický šach v hĺbke 30. Príklad klasickej architektúry bughouse programov sa nachádza na obr. 4.1.

4.1.1 Rozdiely v architektúre

Nevýhoda architektúry bughouse programov spočíva aj v nepredvídateľnosti bughouse hier pri tvorbe stromu hry. Z doterajších pozorovaní nie je jasné, v akej hĺbke je vhodné skončiť s generáciou bughouse pozícií. Prístup bughouse programov Sjeng, TJChess a Sunsetter je podobný — prehľadávajú do šírky a generujú pozície, pokiaľ im zostáva čas určený pre aktuálny ťah. Hlbšie prehľadávanie však v listoch tvorí nepomer v pozíciach, ktoré vznikli použitím *drop* ťahmi a tými, ktoré vznikli odohraním klasických ťahov. Vo



Obr. 4.1: Na obrázku je ukážka stromu klasickej architektúry pre bughouse šach a príklady možných výskytov drop ťahov.

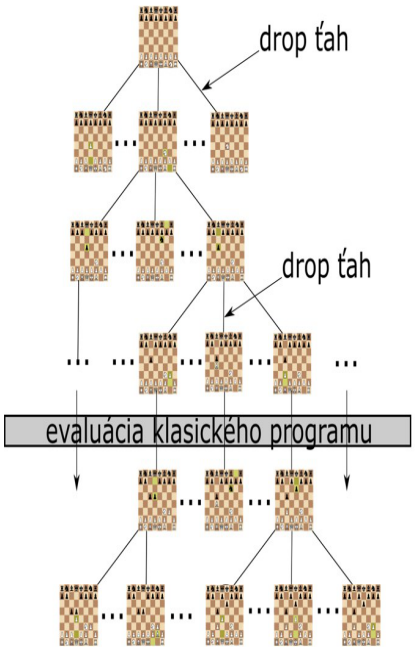
všeobecnosti je ľahšie odstrániť vetvy stromu, ktoré vznikli klasickými ťahmi, pretože touto problematikou sa úspešne zaoberajú aj klasické programy hrajúce šach.

Práve spomenuté dôvody sú hlavnou motiváciou vytvorenia odlišnej architektúry bughouse programu v našej práci a využitie evaluácie klasického programu, ako je vidieť na obr. 4.2. Použitie *drop* ťahov povolíme iba do určitej hĺbky (napr. hĺbka 2 polťahy) a následne listy takto vytvoreného stromu evaluujeme kombináciou kvalitnej evaluácie a vlastnou evaluačnou funkciou, ktorá sa zameriava na bughouse aspekty pozície. Použitím algoritmu minimax následne z vytvoreného vrátime najlepší ťah.

4.1.2 Program Stockfish

Stockfish je open-source šachový program, ktorý je dlhé roky najsilnejším šachovým programom pre klasický šach. Je programovaný v jazyku C++, každý rok vyhráva oficiálne majstrovstvá šachových programov. Jeho úlohou v našej architektúre je poskytovanie kvalitného statického ohodnotenia pozícií, ktoré sú generované v rozhraní Bagatur. Komunikáciu s programom zabezpečuje *UCI*, viac o tomto rozhraní si povieme v časti 4.3.

Na základe statického ohodnotenia ohodnotenia sa potom rozhodneme, ktorý ťah a na ktorej šachovnici bude odohraný. Keďže v generovaných pozíciach sa budú okrem samotnej pozície nachádzať aj figúrky, ktoré môžu byť položené, ohodnotenie musí byť rozšírené aj o hodnoty týchto figúr. Tento problém sa podrobnejšie rieši v podkapitole



Obr. 4.2: Pre porovnanie poskytujeme princíp architektúry nášho bughouse programu.

4.4.2, kde bližšie analyzujeme možnosti ohodnotenia figúr v zásobníku.

4.2 Program Bagatur

Bagatur je pravdepodobne najsilnejší program hrajúci klasický šach programovaný v jazyku Java. Väčšina šachových programov je písaná v jazyku C++ kvôli zvýšenému výkonu. Bagatur je implementovaný v Jave ale ponúka jednoduchšie, programátorsky objektovo-orientované rozhranie a ľahšiu možnosť nadstavby pre účely práce. ELO rating programu Bagatur je 2900, čo je stále viac ako rating najlepších hráčov v šachovej histórii. Bagatur reprezentuje šachovnicu pomocou bitboards a využíva všetky spomenuté optimalizačné algoritmy z kapitoly 2.3. Pre potreby práce budeme používať najmä evaluačnú funkciu a rozhranie reprezentácie šachovnice, najmä funkcie pre generovanie možných klasických ťahov z danej pozície a uskutočnenie ťahov na šachovnici, prípadne kontrolu legálnosti vygenerovaných pozícií.

4.2.1 Bagatur rozhranie BoardImpl

Bagatur poskytuje dve dôležité rozhrania, *BaseMoveList* a *BoardImpl*, týkajúce sa reprezentácie šachovnice a generovania ťahov. *BaseMoveList* si pomocou zásobníku udržiava maximálny počet ťahov pre jednotlivé úrovne vyhľadávania a všetky generované ťahy v danej hĺbke. Okrem spomenutých poskytuje aj statickú evaluáciu v rozhraní *BaseEvalImpl*.

Rozhranie šachovnice poskytuje okrem iných nasledovné metódy, ktoré sú ďalej v práci využívané:

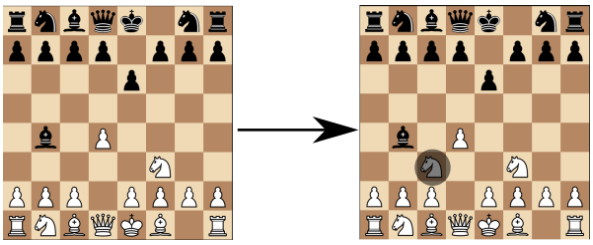
- **BoardImpl(FENposition)** Inicializuje inštanciu šachovnice z pozície, ktorá je argumentom. Táto pozícia je napísana vo FEN štandarde, ktorý je najpoužívanejším na všeobecný prepis šachovej pozície do reťazca (obr. 2.1).
- **genAllMoves(BaseMoveList)** Funkcia vygeneruje všetky ťahy hráča na ťahu a uloží ich do BaseMoveList.
- **makeMoveForward(move)** Šachovnica vykoná ťah *move* a predá možnosť ťahu súperovi. Existuje aj opak tejto funkcie, ktorá vráti šachovnicu o ťah dozadu.
- **toString** Je funkcia vracajúca FEN reťazec reprezentujúci šachovnicu.
- **isInCheck** Funkcia, ktorá je používaná pri tvorbe bughouse pozícií. Kontrola legálnosti generovaných klasických pozícií pozícií nie je potrebná, keďže ju vykonáva samotná funkcia pre generovanie ťahov z rozhrania *BoardImpl*. Avšak pri tvorbe bughouse pozícií je potrebná manuálna kontrola, aby sa zamedzilo generovaniu pozícií, v ktorých kráľ ostáva v šachu z pozícií, v ktorých bol taktiež v šachu, čo je nelegálny ťah.
- **getPieceAttacks()** Každému ťahu, ktorý vyhadzuje figúrku súpera, priradí *centipawn* hodnotu a vráti súčet týchto hodnôt. Ide teda o akési ohodnotenie všetkých potencionálnych ťahov, ktoré vyhazdujú figúrky.
- **hasMovesLeft** Je využívaná na zistenie, či sa hra skončila. Ak je kráľ hráča na ťahu v šachu a nemá žiadne ťahy, hra kočí matom. Ak naopak v šachu nie je, ale nemá žiadne legálne ťahy, hra končí remízou.

Vytvorená funkcia **getPieceIfTakeMove()** vráti znakovú reprezentáciu figúrky, ktorá bola vyhodená, ak bolo na šachovnici vykonané branie figúrky. Porovnáva pozície pred vykonaním ťahu a po vykonaní a v prípade vykonania ťahu, v ktorom sa vyhodila figúra, vráti jej znakovú reprezentáciu.

4.2.2 Generovanie ťahov

Generácia ťahov spočíva v generovaní klasických ťahov z aktuálnej pozície a generovaní ťahov pre jednotlivé figúrky v zásobníku. Pravidlá bughouse šachu hovoria, že figúrky môžu byť až na nasledujúce odmedzenia položené ako ťah na hraciu plochu:

- Pešiaci nemôžu byť položení na poslednú resp. ôsmu radu, ak hráme bielymi (prvú radu ak čiernymi) figúrami.



Obr. 4.3: Názorná ukážka správneho generovania bughouse drop ťahu jazdcom, zvýrazneného čiernou farbou. Hráč bielymi figúrami jazdcom prekryl šach čierneho strelca.

- Ak je pešiak, ktorý bol položený *drop* ťahom na šachovnicu, premenený na dámu a následne vyhodený, vracia sa naspäť spoluhráčovi k dispozícii pre *drop* ťah ako pešiak, nie ako dáma (platí pre všetky premeny, dáma je poskytnutá ako príklad).
- Dve pozície, ktoré vznikli použitím *drop* ťahu musia dodržiavať pravidlá klasického šachu, teda ak v prvej pozícii bol kráľ v šachu, *drop* ťah musel zaistiť prekrytie šachu v pozícii, napr. prekrytím *drop* figúrou. Príklad takéhoto prekrytia je na obrázku 4.3.

Vytvorené rozhranie pre generovanie možných bughouse ťahov v strome hry, *BughousePositionsBuilder*, slúži pre manipuláciu s pozíciami, ktoré sú mu poskytované vo FEN formáte. Keďže číslice vo FEN zápise označujú počet po sebe idúcich prázdnych, neobsadených polí v rade, generácia bughouse ťahov spočíva v nasledovných krokoch:

- uskutočnenie *drop* ťahu na šachovnici
- kontrola legálnosti pozície podľa pravidiel v časti 4.2.2
- aktualizácii FEN zápisu pozície (napr. inkrementovanie počítadla odohratých ťahov a vynulovanie počítadla pravidla 3-tého opakovania)
- odstránení použitej figúry zo zásobníka pre ďalšie volania v rekurzii

Klasické pozície sú tvorené v rozhraní *BoardImpl*, ktoré je súčasťou programu Bagatur. Bughouse pozície sú spolu s klasickými rekurzívne používané pri generovaní pozícií stromu hry v nasledujúcej hĺbke, túto rekurziu bližšie popisujeme v podkapitole 4.3.

4.3 UCI

Po vygenerovaní stromu hry, ktorého listami sú pre $2n$ hráčov pozície na n šachovniciach (v našom prípade 2 šachovnice, 4 hráči), tieto pozície posielame na evaluáciu programu Stockfish. Komunikáciu zabezpečuje protokol $Universal\ Chess\ Interface$, skrátene UCI. Poskytuje jednotnú komunikáciu medzi šachovými programami a používateľským prostredím. Bol vytvorený v roku 2000 autormi Rudolf Huber a Stefan

Meyer-Kahlen [17]. Využíva algebraickú šachovú notáciu pri komunikácii a poskytuje absolútnu kontrolu nad príkazmi, ktoré má program Stockfish vykonať.

Príklad spustenia evaluačnej funkcie programu Stockfish bez časového obmedzenia na základnej šachovej pozícii pomocou *UCI* protokolu by bol nasledovný:

```
ucinewgame
position fen rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq
- 0 1
go infinite
```

Prehľadávanie zastavujeme príkazom stop, po ktorom *UCI* v našom prípade vypíše výsledok nasledovného formátu:

```
info depth 25 seldepth 37 multipv 1 score cp 24 nodes 8090276
nps 1311440 hashfull 1000 tbhits 0 time 6169 pv e2e4
bestmove e2e4 ponder e7e5
```

Po ukončení výpisu sú k dispozícii informácie o prehľadávaní, medzi ktoré patrí počet prehľadaných pozícií, hĺbka prehľadávania, najlepší doteraz nájdený ťah či ohodnotenie pozície a iné [13]. Z informácií je používaná iba evaluácia, teda v príklade hodnota *cp* 24, ktorá značí ohodnotenie pozície v *centipawn* jednotkách.

Rekurzívne generovanie všetkých pozícií

V predchádzajúcich podkapitolách sme objasnili generovanie klasických ako aj bughouse pozícií v hĺbke 1 tímového ťahu, resp. polťahu a komunikáciu potrebnú pre získanie Stockfish evaluácie. Teraz popíšeme rekurzívnu funkciu, ktorá pozície generuje do vopred určenej hĺbky, použitím upraveného minimax algoritmu pre bughouse. Ako sme definovali v časti 3.1.3, pozíciu popisuje presne tím na ťahu, jednotlivé šachovnice a ich zásobníky figúr pre *drop* ťahy. Preto si všetky generované pozície musia pamätať práve tieto informácie. Hlavná rekurzia je rozdelená do dvoch častí, v prvej časti sa tvoria klasické ťahy z danej pozície a v druhej časti sa vytvárajú bughouse ťahy. Na tvorbu bughouse ťahov používame zásobník figúr, ktorý sa v rekurzii posúva ako argument, nasledujúcimi argumentami rekurzie sú aktuálna hĺbka, pozície jednotlivých šachovníc a posledným je tím na ťahu.

Ukážka pseudo-kódu upraveného minimax algoritmu:

```
minimax(positions, pieces, depth, team):
    if depth is equal to 0 or is endgame
        return evaluation + pieces.evaluation
    if firstTeam then
```

```
        value = -999
        for each move in position do
            value := max(value, minimax(makeMove,
            pieces, depth - 1, secondTeam))
        return value
    else (*second Team*)
        value = 999
        for each move in position do
            value := min(value, minimax(makeMove,
            pieces, depth - 1, firstTeam))
        return value
```

4.4 Výber ťahu

Pri výbere finálneho ťahu, ktorý bude vykonaný programom, je dôležitá evaluácia figúr v zásobníku. V nasledujúcich sekciách si viac povieme o dojterajšom prístupe bughouse programov k tejto evaluácii a objasníme potrebu tvorby rôznych evaluačných funkcií pre rôzne úseky partie, ktoré budú dopĺnať klasickú evaluáciu programu Stockfish.

4.4.1 Statická bughouse evaluácia v programe Sjeng

Program Sjeng využíva vo svojej bughouse evaluácii pozičnú evaluáciu pre jednotlivé typy figúriek, tzv. tropizmus a bezpečnosť kráľa.

Pozičná evaluácia pre figúrky bola spomenutá prvykrát v časti 2.3.2, pre všetky typy figúr má program Sjeng definované pole *centipawn* hodnôt. Tieto sú pre všetky figúrky, okrem pešiakov a strelcov, veľmi podobné jednoduchej pozičnej evaluácii programu hrajúceho klasický šach. Keďže pešiakov je počtom najviac a ich hodnota je zo všetkých figúr v šachu najnižšia, ich *centipawn* hodnoty v poli sú vyššie v blízkosti očakávanej pozície súperovho kráľa (napr. najvyššie hodnoty sú pri políčkach po rošáde kráľa). Strelci majú najvyššie hodnoty na diagonálach, ktoré smerujú ku súperovmu kráľovi.

Tropizmus

Angl. *tropism*, označuje ohodnotenie vzdialenosti medzi kráľom a figúrami, ktoré ho môžu potenciálne napadnúť. Do úvahy je okrem vzdialenosti braný aj druh figúriek, napr. pre veže je táto hodnota zmenšená na polovicu a pre strelcov na tretinu. Tropizmus je tzv. pesimistické ohodnotenie, teda nepočíta s figúrami, ktoré môžu brániť v útoku. Ak by bol kráľ a strelec na rovnakej diagonále, ale strelca by blokovala iná figúrka (teda by napádal kráľa nepriamo), hodnota tropizmu je identická so situáciou,

v ktorej by strelec priamo útočil na kráľa bez prekážky. Podobne ako pozičná evaluácia figúr, aj tropizmus je definovaný pre každý typ figúriek 4.1.

Vzdialenosť od kráľa	Dáma	Veža	Pešiak	Jazdec	Strelec
1	100	50	40	40	50
2	60	40	20	70	25
3	20	15	10	35	15
4	5	5	3	10	5
5	2	1	1	2	2
6	0	1	1	1	2
7	0	0	0	0	2

Tabuľka 4.1: V tabuľke uvádzame presné hodnoty tropizmu pre jednotlivé figúry programu Sjeng

Poslednou časťou bughouse evaluácie je bezpečnosť kráľa. Priraďuje *centipawn* ohodnotenie podľa druhu a počtu figúr v súperovom zásobníku.

4.4.2 Evaluácia figúr v zásobníku

Pri tvorbe evaluačnej funkcie, ktorá by dopĺňala evaluáciu programu Stockfish je dôležité, aby reagovala na bughouse aspekty pozície.

Implementovali sme preto tri rozdielne funkcie:

- **KingAttackEval** je evaluácia využívaná v pozíciach, ktoré majú prospekty pre mat súperovho kráľa. Preferuje obetovanie figúr za cenu potenciálneho matového útoku, *centipawn* hodnoty pre držanie pešiakov (ale aj iných figúr, pri pešiakoch je ale zmena badateľná najviac) v zásobníku sú nížšie. Je to základná evaluačná funkcia, ktorou sa začína hra.

- **KingDefenseEval** je protipríkladom útočnej evaluácie, v zásobníku preferuje jazdcov a strelcov a využíva ich na obranu kráľa, indikátorom prechodu na obrannú evaluáciu je vysoký *tropizmus*.

- **PawnPromotionEval** je zriedkavo používaná evaluácia, ktorá preferuje *drop* ťahy pešiakmi na polia pred premenou. Keďže bughouse hry sa väčšinou do koncovej hry nedostávajú (neustály kolobeh figúr medzi šachovnicami), je ojedinelo používaná. Využitie má však aj v prípadoch, kedy sú políčka pred poľom premeny pešiaka neobsadené a existuje možnosť premeny pešiaka na dámu v nasledujúcich ťahoch.

Taktiež je k dispozícii rôzne základné ohodnotenie figúr v zásobníku, okrem nami implementovaného aj ohodnotenie identické programu Sjeng a Sunsetter.

Kapitola 5

Testovanie

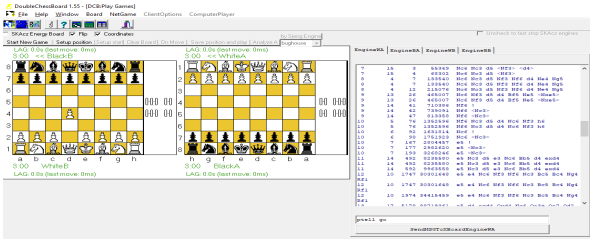
Kapitola obsahuje rozbor výsledkov medzi Sjeng bughouse programamom a našim programom. Viac si taktiež povieme o prípadnom vylepšení programu.

5.0.1 Používateľské prostredia DoubleChessBoard

Testovanie vytvoreného programu prebieha v prostredí DoubleChessBoard [19]. Do prostredia je možné importovať pomocou notácie typu *BPGN* aktuálnu pozíciu aj s históriou a poradím ťahov na jednotlivých šachovniciach. Taktiež ponúka možnosť testovať program proti iným bughouse programom podľa voľby používateľa a komplexné nastavenia pravidiel partie, medzi ktoré patrí výber časovej kontroly, rozdelenie programov do tímov a iné. Dovoľuje priamu komunikáciu s programami pomocou *UCI* protokolu cez príkazový riadok. Vývoj prostredia bol dávnejšie pozastavený, takže medzi podporované programy patrí len Sjeng.

5.0.2 Sjeng a CECP

Pri testovaní programu, ktorý sme vytvorili, bolo potrebné zabezpečiť komunikáciu s programom Sjeng. Sjeng podporuje rozhranie *CECP*, (podobná komunikácia ako pri rozhraní *UCI*) [12]. Sjeng ani iné bughouse programy neposkytujú možnosť výberu



Obr. 5.1: Na obrázku je k vidieť funkcionalita prostredia. Na ľavej strane sú obe šachovnice, na pravej je výstup programu Sjeng.

tímového ťahu ani hru bez časovej kontroly, navrhli sme preto alternatívne riešenie uskutočnenia partie.

Obe šachovnice a ich aktuálna konfigurácia v partii boli samostatne analyzované programom Sjeng a následne bol v partii zahraný ťah, ktorý získal lepšie ohodnotenie z daných šachovníc (teda ťah, ktorý mal najlepšiu evaluáciu). Program mal vždy možnosť vykonania ťahu na oboch šachovniciach, podľa definície definície upravených pravidiel. Hlbšie prehľadávanie minimax algoritmom nebolo možné, kvôli komunikácii s programom Stockfish, ako bližšie popisujeme v podkapitole 5.0.4.

5.0.3 Rozbor odohratých partií

V nasledujúcej časti si ukážeme rozbor ukážkovej partie proti programu Sjeng. Ťahy v partii boli vybrané minimax algoritmom s hĺbkou 2 a s časom 10 milisekúnd na evaluáciu listov stromu. Okrem evaluácie zásobníka figúr nebola použitá žiadna z implementovaných evaluačných funkcií. Sjeng sa riadil upravenými pravidlami a finálny odohraný ťah Sjengu bol ten, ktorý mal v hĺbke 5 najlepšiu evaluáciu. Priebeh partie, implementovaný program hral bielymi figúrami proti Sjeng, ktorý hral čiernymi, bol nasledovný (zápis je v algebraickej notácii [4]):

```
1A.d4 1a.Ng8f6
2B.e4 2a.d5
3A.Nc3 3a.Nc6
4A.e4 4b.Nc6 (!)
5B.Nf3 5b.Ne4
6A.Nxe4 6a.dxe4
7B.N@e3 7a.Qxd4
8A.f3 (?) 8b.Nf6
9B.h4 9b.Nxe4
10A.p@d5 10b.p@g3 (!)
11A.dxc6 (??) 11b.gxf2
12B.Ke2 12b.N@f4# mat
```

Prikladáme aj ohodnotenie evaluačnou funkciou počas partie, v tabuľke 5.1.

Polťah číslo	Evaluácia Sjeng	Evaluácia Stockfish + zásobník
1	0.16	0.1
3	0.2	0.3
5	0.2	-0.7
7	-2	-1.2
9	-2.2	-0.9
11	-2	-2.2
13	-1.2	-1.9
15	-3.5	-2.2
17	-3.1	-2.9
19	-3.2	1
21	MAT v 3	4.3

Tabuľka 5.1: Tabuľka ukazuje presné hodnoty evaluácie ukážkovej partie. Je vidieť, že aj keď sa evaluácia postupne zhoršuje (teda pozícia programu sa zhoršuje), evaluácia Stockfish programu a zásobníku figúr sú postačujúce ako evaluačná funkcia pre bughouse programy.

Závery testovania sú zaujímavé — implementovaný program síce nemal pozitívne výsledky v ani jednej z odohratých partií, ale nami implementovaná evaluačná funkcia mala podobné hodnoty ako evaluácia programom Sjeng. Čas strávený evaluovaním pozície a ani dopĺňajúce evaluačné funkcie neboli tak dôležitým faktorom, ako sa očakávalo, pretože k útoku sa náš program dostal len sporadicky. Prehľadávanie bughouse pozícií spôsobilo hlavný dôvod neúspechu — implementovaný program prehrával partie hlavne z dôvodu prehodenia ťahov v evaluácii.

Čo to presne znamená ? Keďže pozície, v ktorých sa nachádzajú bughouse ťahy sú generované len do veľmi plytkej hĺbky, program nedokázal odhadnúť hrozbu, ktorú predstavujú figúrky v zásobníku súpera. Aj napriek tomu, že evaluácia sa približuje Sjeng evaluácii, nebezpečné *drop* ťahy v hĺbke väčšej ako prehľadávaná sú slabinou programu a partie prehrával hlavne kvôli týmto ťahom.

Partia, ktorú sme rozoberali, sa skončila matom *drop* jazdcom. Program nedokázal odhaliť tento ťah (odhalil ho až v polťahu číslo 22), práve kvôli plytkej hĺbke generovania bughouse ťahov.

5.0.4 Vylepšenia programu

Testovaním sme zistili, že komunikácia s programom Stockfish cez štandardný I/O pre každú vytvorenú pozíciu trvá dlhšie, ako sme očakávali. Spracovanie výstupu a to, že každá pozícia má vopred určený čas na evaluáciu, znemožňuje hĺbkové prehľadávanie, čo má zase za následok prehrávanie partií kvôli plytkému prehľadávaniu. Keďže nami

upravené pravidlá, ktorými sa program riadi, nedefinujú maximálny čas, ktorý môže program využiť na tvorenie a evaluáciu stromu, z pohľadu definície hry to nie je problém generácia bughouse ťahov do väčšej hĺbky. Prakticky je však už prehľadávanie v hĺbke 3 časovo náročné. Ani za testovaných 5 hodín nenašiel algoritmus zo základnej bughouse pozície ideálny ťah. Avšak v prípade potenciálneho zefektívnenia spracovania výstupu sa dá očakávať, že aj kvalita vytvoreného programu bude vyššia.

Ďalším vylepšením je implementovanie optimalizačných algoritmov do programu. Tie by podstatne znížili počet evaluovaných pozícií posielaných Stockfish programu a priamo prispeli k prehĺbeniu generovaného stromu hry.

Záver

Bughouse program, ktorý sme v práci vytvorili, spĺňa všetky požiadavky, ktoré sme stanovili v úvode práce. Architektúra, ktorú sme vytvorili, priamo využíva evaluáciu najkvalitnejšieho klasického programu Stockfish v kombinácii s nami implementovanou bughouse evaluáciou. Komunikácia Stockfish a programu je zabezpečená UCI protokolom. Program pomocou upraveného minimax algoritmu v používateľom zadanej hĺbke hľadá najlepší ťah. Reprezentáciu šachovnice poskytuje klasický program Bagatur, v ktorého rozhraniach sú implementované všetky potrebné funkcie pre generáciu ťahov a kontrolu legálnosti pozícií.

V rámci práce sme taktiež identifikovali problémy, ktoré existujúce bughouse programy majú. Medzi tie patrí komplikovaná komunikácia a preferovanie odohrania rýchlych ťahov kvôli úspore času. Hlavným problémom je však fakt, že programy vykonajú ťah aj vtedy, ak je to objektívne nevýhodné.

Definovali sme preto upravené pravidlá, ktorými sa riadi aj náš program. Tie poskytujú úplné zdieľanie informácie, definujú povinnosť ťahu tímu, nie hráčov, a nie sú závislé na čase. Tieto pravidlá umožňujú následnú generalizáciu bughouse hry pre $2n$ hráčov hrajúcich na n šachovniciach a zdieľanie figúr na všetkých šachovniciach.

Testovanie proti bughouse programom sme prispôsobili upraveným pravidlám, avšak nami implementovanému programu sa nepodarilo doviezť partie do víťazného konca. Navrhnutá architektúra však splnila cieľ, ktorý sme stanovili na začiatku práce. Evaluačná funkcia, ktorú sme implementovali, dosahovala v mnohých prípadoch identické hodnoty ako funkcia bughouse programu Sjeng, a teda sme úspešne použili klasickú evaluáciu pri tvorbe evaluačnej funkcie bughouse programu. Zlepšenie do budúcnosti by sa týkalo najmä zefektívnenia komunikácie, ktorá prebieha pri získavaní evaluácie od programu Stockfish a prehĺbení stromu, ktorý vzniká minimax algoritmom. To by sa dalo dosiahnuť napríklad implementovaním optimalizačných algoritmov, ktoré znížia počet evaluovaných pozícií.

Literatúra

[1] Matthias Wüllenweber a kol. Chessbase databáza. https://en.chessbase.com/, 2021. [Online; prístup 07-marec-2021].

[2] Georg v. Zimmermann Ben Dean-Kawamura. Sunsetter. http://sunsetter.sourceforge.net/index.html, 2016. [Online; prístup 17-november-2020].

[3] Branching factor. Branching factor — wikipédia, slobodná encyklopédia. https://en.wikipedia.org/w/index.php?title=Branching_factor&oldid=981378026, 2020. [Online; prístup 12-november-2020].

[4] chess.com. Chess notation. https://www.chess.com/terms/chess-notation, 2021. [Online; prístup 14-máj-2021].

[5] Sun-Yu Gordon Chi. Exploring the performance of deep residual networks in crazyhouse chess. *arXiv preprint arXiv:1908.09296*, 2019.

[6] Mark Crowther. The week in chess 296 10th july 2000. https://theweekinchess.com/html/twic296.html#6, 2000. [Online; prístup 26-apríl-2020].

[7] Johannes Czech, Moritz Willig, Alena Beyer, Kristian Kersting, and Johannes Fürnkranz. Learning to play the chess variant crazyhouse above world champion level with deep neural networks and human data. *arXiv preprint arXiv:1908.06660*, 2019.

[8] Henry A Davidson. *A short history of chess*. Three Rivers Press, 3 edition, 1949.

[9] Sacha Droste and Johannes Fürnkranz. Learning the piece values for three chess variants. *ICGA Journal*, 31(4):209–233, 2008.

[10] Fergus Duniho. Chess variants. https://www.chessvariants.com/index.html, 2015. [Online; prístup 14-apríl-2020].

[11] Tonny Hecker. Tjchess. http://tonyjh.com/chess/, 2016. [Online; prístup 17-november-2020].

[12] H.G.Muller. Chess-engine communication protocol v2. http://hgm.nubati.net/CECP.html, 2021. [Online; prístup 13-máj-2021].

[13] Stefan-Meyer Kahlen. Uci protocol. http://wbec-ridderkerk.nl/html/UCIProtocol.html, 2004. [Online; prístup 28-apríl-2021].

[14] Topchiyski Krasimir. Bagatur. https://github.com/bagaturchess/Bagatur, 2021.

[15] Nikolas Luke, Claudia Fohry, Gerd Stumme, and Wilhelmshöher Allee. Erweiterung einer schach-engine zu einer team-schachvariante.

[16] Gian-Carlo Pascutto. Sjeng : a chess-and-variants playing program. https://sjeng.org/indexold.html, 2020. [Online; prístup 19-november-2020].

[17] Chess programming Wiki. Uci. https://www.chessprogramming.org/UCI, 2021. [Online; prístup 28-apríl-2021].

[18] Hra s nulovým súčtom. Hra s nulovým súčtom — wikipédia, slobodná encyklopédia. //sk.wikipedia.org/w/index.php?title=Hra_s_nulov%C3%BDm_s%C3%BA%C4%8Dtom&oldid=5351321, 2013. [Online; prístup 12-november-2020].

[19] SKAcz. Double chess board. http://bughousechess.wz.cz/Software/index.html, 2009. [Online; prístup 13-máj-2021].