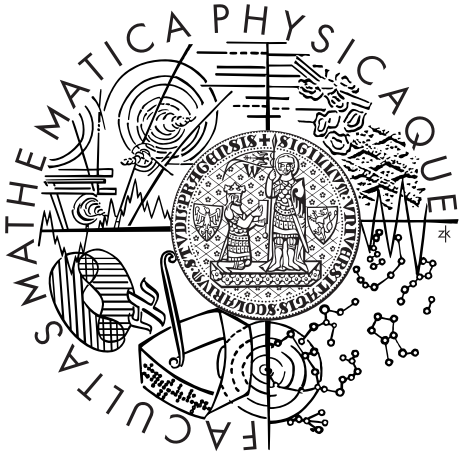
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Marika Ivanová

Algoritmy umělé inteligence pro hru šachy

Kabinet software a výuky informatiky (32-KSVI)

Vedoucí bakalářské práce: Mgr. Martin Petříček
Studijní program: Informatika
Studijní obor: Obecná informatika (IOI)

Praha 2011

Poděkování patří Mgr. Martinu Petříčkovi, vedoucímu mé bakalářské práce. Vážím si času a trpělivosti, které mi věnoval.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

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 v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Algoritmy umělé inteligence pro hru šachy

Autor: Marika Ivanová

Katedra: Kabinet software a výuky informatiky (32-KSVI)

Vedoucí bakalářské práce: Mgr. Martin Petříček, Kabinet software a výuky informatiky (32-KSVI)

Abstrakt: Primárním cílem této práce je prostudovat a shrnout současné algoritmy používané v počítačovém šachu s jejich následnou implementací. Součástí práce je aplikace, která, kromě běžné šachové hry jednoho hráče proti počítači, ponechává uživateli možnost volby používaného algoritmu a nastavení jejich parametrů. Teoretická část se zaměřuje zejména na postupy šachového programování a jejich názorné vysvětlení. Na algoritmech umělé inteligence bylo provedeno několik měření, jejichž výsledky jsou zde prezentovány.

Klíčová slova: šachy, šachové programování, umělá inteligence, hry dvou hráčů

Title: Artificial intelligence algorithms for chess

Author: Marika Ivanová

Department: Department of software and computer science education

Supervisor: Mgr. Martin Petříček, Department of software and computer science education(32-KSVI)

Abstract: The primary objective of this thesis is to study and summarize the current algorithms used in computer chess and their subsequent implementation. The thesis includes a program that in addition to regular single-player game of chess against the computer leaves the user the option of used algorithms and setting their parameters. The theoretical part focuses on chess programming procedures and their illustrative explanation. Several experiments were performed with artificial intelligence algorithms and their results are presented here.

Keywords: chess, chess programming, artificial intelligence, two player games

Obsah

Úvod **3**

1 Obecné aspekty programování šachů **4**
1.1 Stručná historie . . . 4
1.2 Charakteristika problému . . . 5
1.3 Hledání optimální strategie . . . 5

2 Implementace pravidel **6**
2.1 Generátor tahů . . . 6
2.2 Datová reprezentace tahu . . . 7
2.3 Detekce konce partie . . . 7
2.3.1 Výhra jedné strany . . . 7
2.3.2 Remíza . . . 8

3 Metody umělé inteligence **10**
3.1 Herní strom . . . 10
3.2 Prohledávací algoritmy . . . 10
3.2.1 Algoritmus minimax . . . 11
3.2.2 Alfa-beta prořezávání . . . 14
3.2.3 Negascout . . . 15
3.2.4 Iterativní prohlubování . . . 16
3.3 Heuristické metody . . . 17
3.3.1 Třídění tahů . . . 17
3.3.2 Použití dříve vypočtených informací . . . 18
3.3.3 Redukce intervalu alfa-beta . . . 19
3.3.4 Hlubší propočet speciálních tahů . . . 20
3.4 Ohodnocování pozice . . . 23
3.4.1 Cena materiálu . . . 23
3.4.2 Cena pozičních aspektů . . . 23
3.5 Řešení speciálních situací . . . 24
3.5.1 Zahájení . . . 24
3.5.2 Koncovka . . . 25

4 Měření a statistiky **26**
4.1 Základní algoritmy . . . 26
4.2 Třídění tahů v iterativním prohlubování . . . 28
4.3 Efektivita transpozičních tabulek . . . 30

5 Uživatelská část **32**
5.1 Popis grafického rozhraní . . . 32
5.2 Hraní šachové partie . . . 34
5.2.1 Zahájení hry . . . 34
5.2.2 Průběh partie . . . 36
5.2.3 Ukončení partie . . . 36
5.2.4 Ukládání partie . . . 36
5.2.5 Načítání ze souboru . . . 38

5.2.6 Použití šachových hodin 38
5.3 Další funkce programu 38

6 Programátorská dokumentace **39**
6.1 Struktura programu 39
6.2 Reprezentace stavu hry 39
6.2.1 Datová struktura šachovnice 39
6.2.2 Polohy figur na šachovnici 41
6.2.3 Vlastnosti a nastavení 41
6.3 Tahy 41
6.4 Prohledávání stavového prostoru 42
6.4.1 Třída Search 42
6.4.2 Ohodnocování pozice 43
6.5 Interakce s uživatelem 43

Závěr **45**

Obsah přiloženého média **48**

Úvod

Šachy jsou klasická desková hra dvou hráčů. Vznikla v 15. století v jižní Evropě, ale původní kořeny sahají do Indie, 6. století n. l. . Šachy se hrají na čtvercové desce rozdělené na 8×8 střídavě černých a bílých polích. Oba hráči mají na začátku hry šestnáct kamenů šesti druhů: krále, dámu, dvě věže, dva střelce, dva jezdce a osm pěšců. Hráči, označovaní jako bílý a černý podle barvy kamenů, kterými hrají, střídavě provádějí tahy, tedy přesuny kamenů po šachovnici. Cílem hry je mat, takové napadení soupeřova krále, které nelze odvrátit.

V současnosti existují po celém světě šachové kluby a konají se pravidelné šachové turnaje. Ve 2. polovině 20. stol. se vedle soutěží lidských hráčů objevily turnaje šachových strojů a také šachistů proti strojům.

Vývoj počítačového šachu jde ruku v ruce s vývojem samotných počítačů. V průběhu posledních několika desetiletí byly na toto téma publikovány stovky odborných článků. Teorie dosáhla již mimořádných výsledků a současně jsou známy sofistikované algoritmy, jež vítězí nad světovou špičkou mezi šachisty. Kromě počítačových programů, ať už komerčních či open source, existují rovněž šachové stroje určené výhradně pro hraní šachů.

Cílem této práce je vytvořit šachy hrající program, nicméně hlavní oblastí, na kterou se zaměřuje, je podrobnější prostudování existujících algoritmů, jež se v šachu a podobných hrách využívají. Výsledná aplikace by tedy kromě typické hry měla sloužit k demonstraci využitých algoritmů a umožňovat manipulaci s jejich parametry.

První kapitola se zabývá obecně teorií programování šachů. Následující kapitola obsahuje již konkrétnější postupy týkající se implementace pravidel šachové hry. Třetí a asi nejrozsáhlejší kapitola pojednává o konkrétních algoritmech zajišťující umělou inteligenci programu, které byly využity ve vytvořené aplikaci. Ve čtvrté kapitole jsou prezentovány výsledky a měření provedená za pomocí programu Chessplayer a závěrečné dvě kapitoly obsahují uživatelskou a programátorskou dokumentaci.

1. Obecné aspekty programování šachů

Šachy jsou konečná hra dvou hráčů s nulovým součtem a úplnou informací.

Hry s nulovým součtem mají tu vlastnost, že cíle protihráčů jsou právě opačné - vítězství jednoho znamená prohru druhého a naopak. Jestliže p je daná pozice existuje funkce $f(p)$ značící hodnotu této pozice z pohledu hráče na tahu. Hodnota pro soupeře je potom $-f(p)$.

Každý hráč má, na rozdíl od různých např. karetních her, po celou dobu konání partie přehled o dění na hrací desce. Náhoda nemá žádný vliv na průběh šachové partie. Podobné vlastnosti mají hry jako dáma či reversi. Proto se šachy řadí mezi hry s úplnou informací.

Hry dvou hráčů je možno charakterizovat jako množinu pozicí a množinu pravidel pro tahy z jedné pozice do druhé, přičemž hráči na tahu se střídají [9]. Předpokládáme, že pravidla zakazují veškeré nekonečné sekvence pozic[1]a že každá pozice má konečně mnoho následníků, neboli vždy existuje pouze konečné množství přípustných tahů.

1.1 Stručná historie

První šachový stroj zkonstruoval roku 1771 rakouský vynálezce Wolfgang von Kempelen. Dnes je toto historické zařízení známo pod označením Turek a měl sloužit pro pobavení císařovny Marie Terezie. Ve skutečnosti se však jednalo o podvod, neboť v útrobách stroje se ukrýval šachista, jenž vždycky zahrál vhodný tah. Ačkoli se tedy nejednalo o skutečný šachy hrající vynález, lze jej považovat za významný milník v historii počítačového šachu, neboť jde o první myšlenku šachového stroje. V polovině 19. století Turek shořel a ne dlouho poté vyšla série článků v americkém The Chess Monthly, která osvětlila jeho tajemství.

Po 2. světové válce to byl Alan Turing, který teoreticky popsal šachový program. Podobně se šachovými algoritmy zabýval i americký matematik Claude Shannon. Popsal, jakým způsobem probíhá výpočet optimálního tahu počítače, což je minimaxový algoritmus založený na ohodnocovací funkci, ve které byly zahrnuty hodnoty figur a několika pozičních faktorů.

První skutečný program, hrající variantu šachu na hrací desce velikosti 6x6 polí bez střelců, navrhli roku 1956 Paul Stein a Mark Wells pro počítač MANIAC I. V témže roce John McCarthy vynalezl algoritmus alfa-beta. Programy hrající standardní šachy byly vyvinuty o rok později. V roce 1967 byla představena metoda transpozičních tabulek. Od 70. let 20. století probíhalo mnoho turnajů šachových programů, či turnajů lidských hráčů a strojů. Výkon programů stále rostl, nikoli však díky stále sofistikovanějším algoritmům, ale spíše vzhledem k neustále dokonalejšímu hardware. Významný je rok 1997, kdy byl ruský šachový velmistr Garry Kasparov poražen šachovým počítačem Deep Blue, vyvinutým firmou IBM.

[1] Ve skutečnosti tato podmínka není v šachu splněna [8]. Jestliže se vyskytne potřetí stejné postavení, partie končí remízou na základě oprávněné reklamace hráče. Jestliže hráč ale této možnosti nevyužije, hra nadále pokračuje.

1.2 Charakteristika problému

Vývoj šachového programu vyžaduje návrh datových struktur uchovávajících stav hry a funkcí, které s těmito strukturami provádějí příslušné operace, implementaci pravidel a s tím související generátor přípustných tahů. Všechny tyto nezbytné úkoly jsou čistě technického rázu a ačkoli existuje více různých přístupů, nepředstavují výrazný problém Dále je však třeba navrhnout mechanismus pro výběr tahu z dané pozice. Zde se již situace poměrně komplikuje, neboť výkonnost umělé inteligence velmi závisí na hloubce prohledávaného stavového prostoru.

1.3 Hledání optimální strategie

Přestože počítačové programy hrající šachy i hardware, na kterém běží se neustále zdokonalují, stále nebyla objevena optimální strategie pro bílého ani černého hráče. U tzv. vyřešených her jsme schopni předpovědět jejich výsledek z libovolné pozice za předpokladu, že obě strany volí optimální tahy. Vyřešení her dvou hráčů posuzujeme na 3 úrovních:

- Ultra-slabé: jsme schopni dokázat, že první hráč vyhraje, prohraje či remízuje, jestliže obě strany hrají optimálně. Nicméně správné tahy nemusíme umět určit.
- Slabé: Vyřešení hry na této úrovni zaručuje znalost postupu, který zajistí výhru či remízu hráče, ať hraje soupeř jakkoli.
- Silné: zde je znám algoritmus určující nejlepší tah z libovolného stavu hry bez ohledu na předchozí chybné tahy.

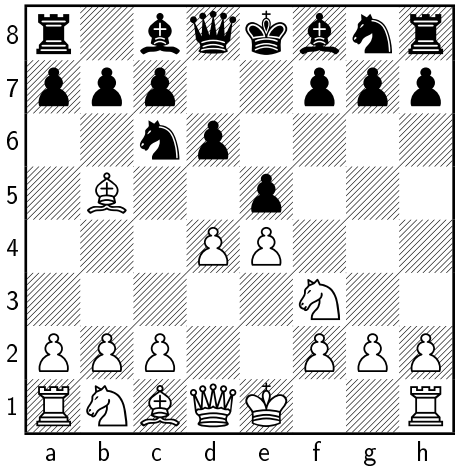
Významného výsledku bylo dosaženo vyřešením hry dáma v roce 2007. Oba hráči mohou z výchozí pozice vynutit remízu, třebaže hraje soupeř bezchybně. Dáma je tedy vyřešena slabě [13]. Šachové koncovky obashující 3 až 6 figur jsou vyřešeny, taktéž některé koncovky se 7 figurami (oba krále započítáváme). Názor, že standardní šachová hra bude v budoucnu vyřešena, má své zastánce i odpůrce.

2. Implementace pravidel

Šachy mají poměrně rozsáhlá pravidla, některá však nejsou nezbytná pro běžnou hru. Spousta z nich má význam pouze na turnajích.

2.1 Generátor tahů

Procedura generování tahů se za běhu programu volá velmi často. V případě, že je na tahu počítač, potřebujeme v paměti přípustné tahy, aby měl prohledávací algoritmus z čeho vybírat. Pokud naopak je na řadě hráč, musíme ověřit, zda jím vybraný tah neporušuje pravidla. K tomu opět využijeme aktuálně vygenerované tahy. Jednoduše projdeme strukturu, která je uchovává a v případě, že se daný tah nachází ve struktuře, vybraný tah se řídí pravidly příslušné figury. To však samo o sobě neznamená, že takový tah můžeme skutečně dle pravidel zahrát. Uvažme např. situaci na diagramu 2.1. Kdybychom chtěli zahrát jezdcem na poli c6, král zůstane odkrytý a napadený, což pravidla zakazují:



Aby se generování takových nepřípustných tahů zabránilo, muselo by se pokaždé kontrolovat napadení krále, což by ale samotný proces značně komplikovalo. Během vývoje a ladění se osvědčil postup, kdy se vygenerují všechny pseudolegální tahy, tzn. nezabýváme se případy, kdy po zahrání tahu král skončí v šachu. Skutečnou přípustnost tahu kontroujeme až ve chvíli, kdy se hráč pokusí zahrát a případný problémový tah mu jednoduše nepovolíme zahrát. Pokud se jedná o výběr tahu počítače, problémový tah je odfiltrován prohledávacím algoritmem.

V šachách rozlišujeme na obou stranách 6 druhů figur, z nichž každá má specifická pravidla pro svůj tah. Všechny druhy figur můžeme rozdělit podle vzdálenosti, kterou mohou na jeden tah překonat, na krátkodosahové a dalekodosahové. Pěšec, král a jezdec jsou figury krátkého dosahu. Výběr jejich tahů spočívá v prozkoumání několika sousedních políček, na která by se případně mohly posunout. U figur dalekého dosahu, mezi které řadíme střelce, věž a dámu, spočívá ve zkoumání směrů. Začínáme vždy testovat sousední políčko v daném směru a iterujeme tak dlouho, dokud nenarazíme na překážku v podobě figury nebo okrajové části šachovnice. Z pohledu implementace jsou tahy věží, jezdců, střelců a dámy poměrně nezajímavé, jelikož tyto figury neprochází během partie žádnými změnami. Zbývající 2 figury již mají své tahy komplikované jistými speciálními případy.

Král: Speciálním tahem krále je tzv. rošáda, která se typicky provádí ze začátku partie a slouží k obraně krále. Tohoto speciálního tahu se kromě krále účastní ještě jedna z věží. Rošádu lze zahrát pouze v případě, že s králem ani příslušnou věží nebylo dosud taženo, král není v šachu, ani žádné políčko, přes které král při rošádě přejde, není napadeno soupeřovou figurou. Proto je třeba zavést speciální proměnné uchovávající tyto informace.

Pěšec: jako jediná figura nemůže couvat. Při běžném tahu se posouvá o jedno políčko dopředu, pokud s ním ještě nebylo zahráno, může se posunout i o 2 políčka, je li tam volno. Soupeřovu figuru může sebrat pouze šikmo dopředu, nikoliv jako běžný tah. Výjimkou je braní mimochodem, neboli en passant, kdy můžeme sebrat soupeřova pěšce, který zahrál dlouhý první tah pěšcem a skončil těsně vedle (vpravo či vlevo) našeho pěšce. Poté můžeme sebrat tohoto soupeřova pěšce, přestože naším tahem neskončíme na políčku se soupeřovou figurou. Dosáhne-li pěšec konce šachovnice (řady 8 resp. 1 jestliže je bílý, resp. černý), dojde k proměně pěšce na libovolnou figuru s výjimkou krále a pěšce.

Výstupem generátoru tahů je spojový seznam obsahující pseudopřípustné tahy.

2.2 Datová reprezentace tahu

Pro jednoznačné určení tahu bychom mohli vystačit pouze se znalostí indexu výchozího a cílového pole. Jak ale později ukážeme, proces hledání optimálního tahu vyžaduje vracení tahů, což vede na nutnost uchovávat ještě informaci o figuře, kterou jsme případně naším tahem sebrali. Kromě toho je třeba ještě zaznamenat proměnu pěšce, přestože k tomu v praxi dochází poměrně zřídka. Při programování se ukázalo jako vhodné uchovávat booleovskou proměnnou, značící, zda došlo k tahu mimochodem.

2.3 Detekce konce partie

Schopnost rozpoznat konečný stav partie je nezbytnou součástí šachového programu. Kromě odpovídajícího zakončení hry, jež zahrnuje informování uživatele o výsledku, by se velmi hodilo, kdyby se konečné stavy zohledňovaly také v procesu výběru vhodného tahu. Šachová partie skončí výhrou jednoho z hráčů, nebo remízou.

2.3.1 Výhra jedné strany

Hráč vyhraje v případě, že

- dá soupeři mat
- soupeři vyprší čas
- soupeř se vzdá

Z pohledu programování je zajímavý zejména první případ, kdy nastává mat. Procedura, která mat rozpoznává, sestává ze dvou částí. Nejdříve určíme, jestli je král v šachu (cílové pole některého soupeřova tahu je stejné, jako pole na němž stojí

král), poté zkoušíme zahrát postupně všechny vygenerované tahy šachovaného hráče a zkoumáme, zda po jejich provedení král zůstává nadále v šachu. Jestliže ano, nastal šach mat, hráč na tahu prohrál a partie tím končí. Kdybychom se místo pseudonáhodného generátoru tahů rozhodli pro generování pouze určitě přípustných tahů, bylo by určení matu jednodušší, neboť bychom měli krále v šachu a žádný přípustný tah.

Ve zvoleném způsobu navíc nestačí kontrolovat pouze nemožnost táhnout králem. Často se totiž objevují situace, kdy samotný král, který je v šachu, nemá žádný tah, kterým by se svého napadení zbavil, avšak ostatní figury se mohou postavit mezi krále a soupeřovu útočící figuru, což zruší šach. Další možností obrany proti šachu je jednoduché sebrání útočící figury, pokud to lze. Proto je nutné zkoušet zahrát postupně všechny tahy ze seznamu, abychom neopomenuli některou možnoou obranu proti šachu.

2.3.2 Remíza

Remízou může partie skončit rovněž z rozličných důvodů:

- Pat. Král není napaden, avšak hráč na tahu nemá žádný přípustný tah.
- Mrtvá pozice. Ani jednomu králi nelze dát mat žádnou posloupností tahů.
- Opakování pozice. Jestliže nastane alespoň potřetí stejná pozice.
- Za posledních 50 tahů nedošlo k sebrání figury a nebylo taženo pěšcem
- Hráči se dohodli na remíze.

Rozpoznání patové situace probíhá podobně jako v případě matu. Jestliže král není napaden, ale nemáme přípustný tah, hra skončila remízou. Jediný rozdíl mezi detekcí matu a patu spočívá v tom, že v prvním případě král není napaden, zatímco ve druhém případě ano.

Mrtvé pozice se objevují především na konci partie, kdy většina figur už není na šachovnici. Zbývající figury ve hře však již nemohou dát soupeři mat. Jsou to postavení se samotným králem, králem a střelcem nebo jezdcem, čí králem a dvěma jezdci.

Trojnásobné zopakování téže pozice je další příčinou remízy. Může být vynucené i nevynucené. Navíc platí, že mezi jednotlivými výskyty sledované pozice se může objevit libovolné množství jiných postavení, neboli pozice se nemusí nutně opakovat těsně za sebou. K vynucenému opakování postavení nejčastěji dochází při tzv. věčném šachu, kdy šachující hráč má zájem o remízu a šachovaný hráč nemá jinou možnost, než ustupovat opakovaně na stejná políčka. Toto pravidlo bylo implementováno pomocí hashování. Hashovací tabulku využijeme také jako transpoziční tabulku a budeme pomocí ní redukovat prohledávací strom (viz. kap. 3). Klíčem v hashovací tabulce je hash pozice, v našem případě máme velké číslo typu Long. Hodnotou je datová struktura nesoucí mimo jiné informaci o počtu výskytů dané pozice. Z každého postavení, které na šachovnici nastane, vypočteme hash a zvýšíme počet výskytů. Jestliže se jedná o první výskyt, znamená to uložení nového záznamu do hashovací tabulky. Jakmile se potřetí narazí na stejný hash, partie končí remízou. Přestože zde mohou nastat kolize, je tento postup

pravděpodobně jediný rozumný. Jakékoli jiné ukládání a zjišťování počtu výskytů daných pozic by neúnosně zatěžovalo již tak časově náročný výběr nejlepšího tahu. Navíc se tato metoda, známá pod označením transpozice, úspěšně využívá v prohledávacím algoritmu (viz. další kapitola).

Pravidlo padesáti tahů se v praxi využívá jen zřídka a spolu s nabízením remízy není v programu implementováno.

3. Metody umělé inteligence

V této kapitole je nejdříve obecně popsána povaha úlohy výběru tahu v nějaké hře dvou hráčů. Poté následuje popis fungování konkrétních algoritmů zajišťující výběr optimálního tahu, které byly použity v programu Chessplayer a několik dalších, které program nezahrnuje.

3.1 Herní strom

Představme si nějakou výchozí situaci na šachovnici jako kořen stromu Γ. Všechna možná postavení dosažitelná na 1 tah budou potomky tohoto kořene a stejným způsobem vytvoříme potomky potomků kořene atd. Pozice, ze které nelze zahrát žádný tah (mat, pat, nebo jiná remízová postavení, viz. pravidla šachu [8]), představují listy p_i a mají přiřazeno číslo $v(p_i)$, přičemž

$$v(p_i) = \begin{cases} 1 & \textit{pokud je pozice vtzn pro blho} \\ 0 & \textit{pokud je pozice remzov} \\ -1 & \textit{pokud je pozice vtzn pro ernho} \end{cases} \tag{3.1}$$

Všechny vrcholy stromu Γ rozdělíme do dvou disjunktních množin B_0 a B_1 tak, že v B_0 jsou právě všechny vrcholy představující pozice, kdy je na tahu bílý a v B_1 pozice, kdy je na tahu černý hráč. Orientovanému stromu Γ se říká strom hry dvou hráčů s úplnou informací [1].[16]. Vhodné je též označení *stavový prostor*.

3.2 Prohledávací algoritmy

Proces vybírání vhodného tahu vyžaduje zkoušet různé sekvence tahů a určovat cenu takto vzniklé pozice. Již v počátcích budování teorie programování podobných typů her se tento postup rozdělil na dva typy[14]: Prohledávání hrubou silou (typ A) a selektivní prohledávání (typ B). Teorie vznikala od 50. let 20. stol. a v té době počítače nedosahovaly takových výpočetních výkonů, aby se docílilo efektivního využití hrubé síly. To je jeden z důvodů, proč se zpočátku teoretici přikláněli spíše k selektivní strategii. S postupným zdokonalováním počítačů se zvyšoval význam brute-force přístupu a v dnešní době se často využívá kombinací obou směrů.

Prohledávání hrubou silou: Prohledávací techniky této skupiny lze považovat za bezpečné ve smyslu, že dokáží najít skutečně nejlepší tah (dle ohodnocovací funkce). Takový způsob prohledávání je možno vylepšovat především co nejlepším využitím dříve získaných informací.

Selektivní prohledávání: V praxi se ukazuje, že i když se bezpečné techniky značně optimalizují, herní stromy stále zůstávají obrovské a nemůžeme si dovolit příliš velké hloubky jejich prohledávání. Na rozdíl od předešlého přístupu, selektivní prohledávání předpokládá znalost povahy problému, na jejímž základě se vybírají vhodné tahy. Bohužel, implementovat obecně známé zásady šachové strategie je velmi obtížné až nemožné. Nicméně alespoň částečné znalosti můžeme využít a vhodně kombinovat s brute-force metodami, což vede ke znatelnému vylepšení. Existuje řada heuristik fungujících na tomto principu.

Cílem prohledávacích algoritmů je dosažení co možná největší prozkoumávané hloubky a aby výpočet netrval příliš dlouho.

Naivní algoritmus pro výběr tahu zkouší z dané pozice zahrát všechny možné tahy a podle různých kritérií vyhodnocuje vzniklé stavy. Ten tah, který vedl do nejlepšího stavu, nakonec skutečně zahraje. Takový algoritmus však nevidí, co by mohl zahrát soupeř, a tedy hraje jako začátečník, přehlíží napadení figur a hrozící maty.

3.2.1 Algoritmus minimax

V obecé rovině je minimax rozhodovací providlo minimalizující možnou ztrátu a maximalizující potenciální zisk. V teorii her je minimaxová strategie smíšená strategie[2], což je součást řešení her s nulovým součtem. Řešením problému naivního algoritmu je užití algoritmu minimax. Metoda minimaxu je ve svém principu metodou prohledávání do hloubky s omezením hloubky prohledávání. [16] Ten dostává na vstupu hloubku prohledávání a v každém rekurzivním volání je hloubka snížena o 1, až dosáhne hodnoty 0. V tu chvíli se volá nějaká ohodnocovací funkce, která vrátí cenu dosažené pozice. Průběh algoritmu lze přehledně prezentovat na příkladu dvou hráčů pojmenovaných *Min* a *Max*[9]: Označme p danou pozici na šachovnici, kterou chceme ohodnotit a hráč *Max* je na tahu, hodnotu pozice p označíme *f(p)*, jestliže je naopak na tahu *Min*, hodnota pozice bude

$$g(p) = -f(p) \tag{3.2}$$

Max se snaží maximalizovat výslednou hodnotu, zatímco *Min* naopak minimalizovat. Dále předpokládejme, že d označuje hloubku prohledávání a $p_1, .., p_n$ jsou všechny možné pozice dosažitelné z p na jeden tah. Pak můžeme popsat nejlepší možnou hodnotu dosažitelnou z p proti optimálně hrajícímu protihráči:

$$F(p) = \begin{cases} f(p) & pro\ d = 0 \\ max(G(p_1), ..., G(p_n)) & pro\ d > 0 \end{cases} \tag{3.3}$$

což je nejlepší zaručená hodnota pro hráče *Max* začínajícího v pozici p a

$$G(p) = \begin{cases} g(p) & pro\ d = 0 \\ min(F(p_1), ..., F(p_n)) & pro\ d > 0 \end{cases} \tag{3.4}$$

je nejlepší hodnota pro hráče *Min*, které určitě může dosáhnout. Pseudokód nastíněného postupu vypadá následovně:

[2]Řešení hry v oboru čistých strategií znamená, že hráč dosáhne svého cíle pouze pomocí jediné své strategie. Naproti tomu řešení v oboru smíšených strategií znamená, že se hráč nemůže řídit pouze jedinou ze svých strategií, ale musí najít způsob používání strategií v jednotlivých partiích, tj. rozdělení četností, podle kterého má svoje strategie střídat [6].

Algoritmus 1 Minimax

```
function max(position, depth)
  if depth = 0 then
    return evaluate(position)
  end if
  maxval ← −∞
  for all possible moves m do
    position ← play_move(position, m)
    score ← min(position, depth − 1)
    position ← play_move_back(position, m)
    if score > maxval then
      maxval ← score
    end if
  end for
  return maxval
end function

function min(position, depth)
  if depth = 0 then
    return -evaluate(position)
  end if
  minval ← ∞
  for all possible moves m do
    position ← play_move(position, m)
    score ← max(position, depth − 1)
    position ← play_move_back(position, m)
    if score < minval then
      minval ← score
    end if
  end for
  return minval
end function
```

Pro implementaci je často vhodnější využít minimaxového algoritmu ve formě zvané *negamax*[3]. Program Chessplayer obsahuje základní prohledávací metodu tímto způsobem implementovanou. Díky rovnosti

$$max(a, b) = -min(-a, -b) \tag{3.5}$$

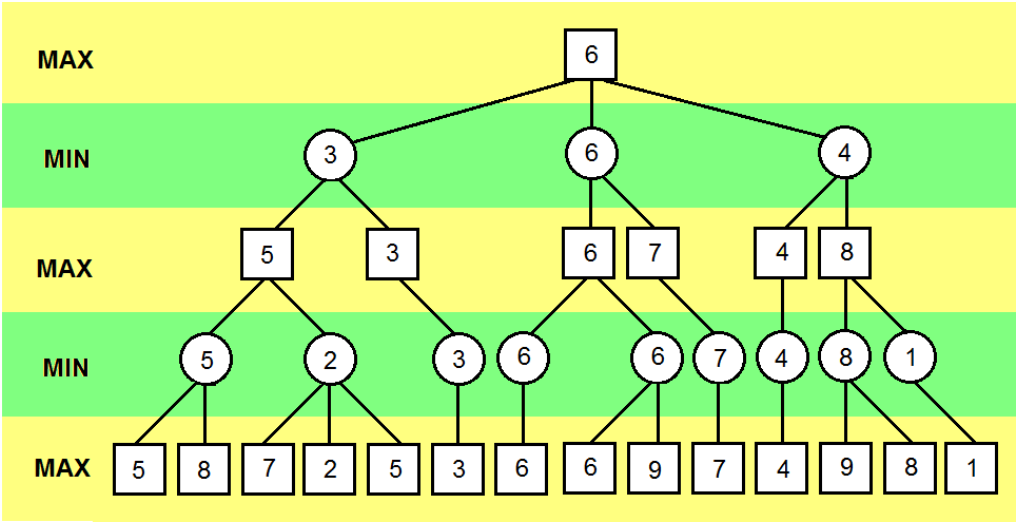
můžeme místo dvou separovaných funkcí použít jedinou ve tvaru 2.

[3] V literatuře se často vysvětluje princip minimaxu na pseudokódu negamaxu

Algoritmus 2 Negamax

```
function negamax(position, depth)
  if depth = 0 then
    return evaluate(position)
  end if
  max ← −∞
  for all possible move m do
    position ← play_move(position, m)
    score ← -negamax(position, depth − 1)
    position ← play_move_back(position, m)
    if score > max then
      max ← score
    end if
  end for
  return max
end function
```

Algoritmus zkoumá vždy všechny přípustné tahy. Na obrázku 3.1 je příklad stromu hry, kde v každém uzlu je vyznačena hodnota vypočtená popisovaným algoritmem. Ze zřejmých důvodů má zobrazený strom vrcholy s pouze 1 až 3 potomky. Průměrný větvící faktor v případě šachů se pohybuje okolo 30. [14][16]



Obrázek 3.1: Kompletní ohodnocení herního stromu do hloubky 4 půltahy pomocí algoritmu minimax. V jednotlivých hladinách se střídavě hodnota minimalizuje a maximalizuje v závislosti na hráči na tahu.

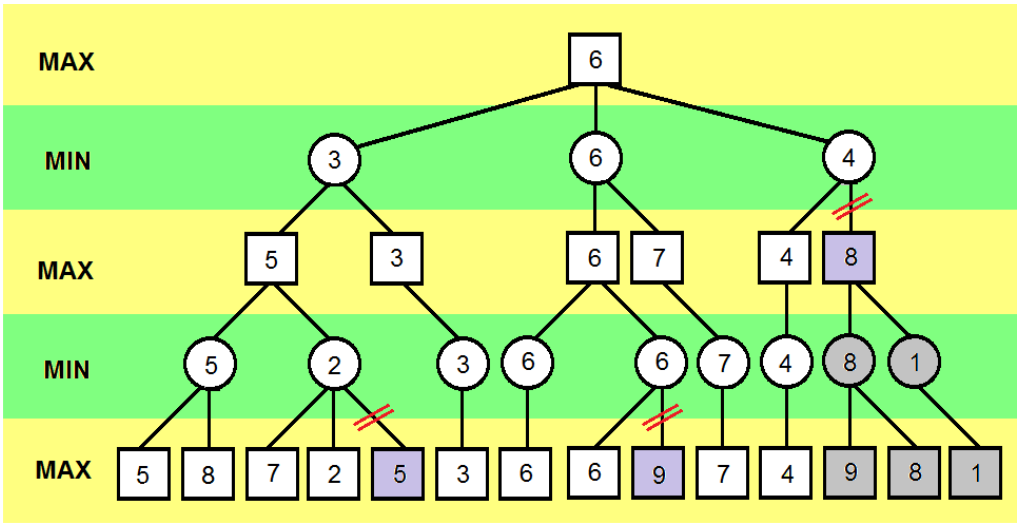
Během činnosti algoritmu je v paměti uložen pouze 1 uzel herního stromu (situace na šachovnici). Zásadní slabinou minimaxu je jeho exponenciální časová složitost. V základní variantě se zkoušejí všechny možné tahy bez jakéhokoli omezení. V programu využívajícím algoritmu minimax si můžeme dovolit prohledávat na 3 až 4 půltahy dopředu, při větších hloubkách se doba odezvy počítačového soupeře stává neúnosnou. Počet uzlů prohledávacího stromu určuje, kolikrát je třeba zahrát tah a vrátit jej. Množství listů představuje počet volání ohodnocovací funkce.

Označme průměrný větvící faktor jako b a hloubku prohledávání d. Pak dostáváme

$$b + b^2 + ... + b^d = b^{d+1} - 1 \tag{3.6}$$

provedených tahů a b^d volání ohodnocovacích funkcí.

3.2.2 Alfa-beta prořezávání



Obrázek 3.2: Kompletní ohodnocení herního stromu do hloubky 4 půltahy pomocí algoritmu alfa-beta. Bílé uzly představují pozice, jejichž hodnota byla propočítána, do šedivých uzlů se propočet dodostal. Jejich ohodnocení se nevešlo do požadovaného intervalu a není třeba se jimi dále zabývat

Efektivita klasického prohledávání do hloubky se může zvýšit použitím techniky *branch and bound*, kdy se zavrhují evidentně horší řešení, než dosud nalezená. Pracujeme zde s určitou mezní hodnotou a všechna horší řešení se neakceptují. V případě her dvou hráčů bude pro každého hráče jedna mezní hodnota. Strategie větví a mezí modifikovaná pro hry dvou hráčů se nazývá *alfa beta prořezávání (alpha-beta pruning)*[16].

Hráč, který maximalizuje ohodnocení, využívá dolní mez ohodnocení uzlu *alfa*, zatímco minimalizujícímu hráči přísluší horní mez ohodnocení *beta*. Obrázek 6.1 znázorňuje stejný strom hry jako obrázek 3.1, ale demonstruje průchod algoritmu alfa-beta. Kořen stromu má 3 potomky. Během vykonávání výpočtu se levý podstrom ohodnotil na 3 a prostřední podstrom na 6. Na hladině pod kořenem se výsledná hodnota minimalizuje, v kořeni naopak maximalizuje. Již po průchodu bílými uzly pravého podstromu je zřejmé, že hodnota nebude vyšší než 5. To znamená, že další propočet pravého podstromu můžeme přerušit, neboť hráč v kořeni si určitě vybere prostřední podstrom s hodnotou 6.

Algoritmus 3 Alpha-beta pruning

```
function alpha-beta(depth, alpha, beta)
  if depth = 0 then
    return evaluate position
  end if
  max ← ∞
  for all possible moves do
    play move
    score ← −Alphabeta(depth − 1, −beta, −alfa)
    play move back
    if score ≥ beta then
      return beta // beta cutoff
    end if
    if score > alpha then
      alpha ← score
    end if
  end for
  return alpha
end function
```

Na začátku běhu algoritmu máme maximální možný rozdíl hodnot alfa a beta, postupně se tento interval zmenšuje. Naší snahou je docílit, aby zužování proběhlo co nejrychleji, což má za následek rychlejší výběr tahu. Snažíme se tedy nejlepší tahy propočítávat brzy, abychom také co nejrychleji ořezávali. Zřejmě v nejhorším případě, kdy tahy zkoušíme postupně od nejhoršího, nedochází k žádnému ořezávání a algoritmus má stejnou časovou složitost jako minimax.

3.2.3 Negascout

Algoritmus negascout byl představen v článku Alexandera Reinefelda [11] z roku 1983. Základní idea vychází z algoritmu Scout, který byl uveden o 3 roky dříve [10].

Idea spočívá v předpokladu, že první tah je zároveň nejlepší. Při jeho prohledávání se nastaví minimální interval alfa-beta, tzv. *null window*. Velmi důležitou roli hraje pořadí tahů, které postupně hrajeme. Algoritmus negascout dosahuje radikálnějšího prořezávání než alfa-beta, ovšem je třeba mít tahy co nejlépe setříděné. V nepříznivém případě se může stát, že algoritmus poběží déle. Samozřejmě nelze mít tahy utříděny zcela přesně od nejlepšího, neboť kdyby byl znám algoritmus, který by toho byl schopen, jakékoli prohledávání by postrádalo smysl a jednoduše by se vždy vrátil nejlepší tah.

Graf na obrázku 3.1 zobrazuje vývoj počtu listů (volání ohodnocovací funkce) v závislosti na pořadí tahu.

Algoritmus 4 Negascout

```
function Negascout(depth, alpha, beta)
  if depth = 0 then
    return evaluate position
  end if
  m ← −∞
  n ← beta
  for all possible moves do
    play move
    score ← −NegaScout(depth − 1, −n, −Max(alpha, m))
    play move back
    if score ≥ m then
      if n = beta||(depth ≤ 2) then
        m ← t
      else
        m ← −NegaScout(depth − 1, −beta, −t) f
      end if
    end if
    n ← Max(alfa, m) + 1
  end for
  return m
end function
```

3.2.4 Iterativní prohlubování

Tato technika je významná zejména v obecném prohledávání. Kombinuje totiž přístupy DFS a BFS. Algoritmus pro DFS může v nepříznivých případech hledat blízký cíl dlouho, dokonce nemusí skončit. Tento nedostatek právě řeší iterativní prohlubování. Opět se jedná o prohledávání do hloubky, ovšem hloubka je omezená a v každém dalším kroku se zvyšuje o 1.

Algoritmus 5 Iterative deepening

```
function IterativeDeepening(maxDepth)
  for depth = 0 to maxDepth do
    bestMove ← findBestMove(depth) //  using DFS
  end for
  return bestMove
end function
```

Časová složitost výsledného prohledávání je samozřejmě horší, nicméně asymptoticky se nemění.

$$db^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d = O(b^d) \tag{3.7}$$

V šachu máme však hloubku prohledávání vždycky vhodně omezenou a tak by se mohlo zdát, že iterativní prohlubování není opodstatněné. Někdy chceme prohledávání neomezovat konkrétní hloubkou, ale časem. To je praktické zejména proto, že větvící faktor se v průběhu partie mění a s ním i doba propočtu. Pokud

bychom omezili čas algoritmu bez iterativního prohlubování, mohlo by dojít k situaci, že algoritmus propočítal od kořene jen několik prvních tahů do hloubky a vypršel stanovený čas. V tom případě vrátí onen prohledávaný tah, který rozhodně nemusí být nejlepší. Použijeme-li iterativní prohledávání a vyprší časový limit, máme uložen nejlepší tah vzešlý z DFS předchozí iterace, tedy s hloubkou o 1 menší. Dokonce zde máme možnost řídit dobu propočtu uživatelsky. Některé heuristiky ukládají dříve vypočtené informace a odříznutí větve prohledávaného stromu může díky tomu nastávat častěji. Tyto informace mohou být mimo jiné vypočteny v předchozí iteraci prohlubování. V případě použití základních variant minimaxu či alfa-beta prořezávání je nicméně zbytečné používat iterativní prohlubování.

3.3 Heuristické metody

V současné době známe mnoho různých vylepšení alfa-beta algoritmu. Obvykle jsou založeny na některém z následujících principů, či jejich kombinaci [12]:

- Třídění tahů
- Opakované použití dříve vypočtených informací
- Redukce intervalu alfa-beta
- Hlubší propočet speciálních tahů

První dvě položky mají význam pouze za předpokladu, že byl použit algoritmus alfa-beta. Algoritmus minimax totiž neořezává neperspektivní větve, a proto u něj lze využít pouze heuristiky založené na znovupoužití dříve vypočtených informací. Nyní podrobněji rozebereme jednotlivé skupiny heuristik a některé jejich zástupce. Ty heuristiky, které se využívají v programu Chessplayer budou rozvedeny hlouběji.

3.3.1 Třídění tahů

Pořadí tahů výrazně ovlivňuje rychlost činnosti alfa-beta algoritmu. Jestliže vyhodnocujeme dobré tahy dříve než špatné, dojde k oříznutí (tzv. beta-cutoff) dříve a výpočet proběhne rychleji. Kdybychom byli schopni seřadit tahy od nejlepšího po nejhorší, nebo alespoň určit nejlepší, nepotřebovali bychom žádné prohledávání. Vhodnost tahu však můžeme odhadovat z jeho vlastností a ty slibné řadit ze začátku seznamu tahů, ty ostatní na konec.

K nalezení co nejlepšího setřídění můžeme přistupovat dvěma směry: dynamicky, kdy se tahy třídí na základě jejich vypočtených hodnot z předchozí iterace, ovšem zde je nutné využití iterativního prohlubování, nebo se můžeme pokusit o třídění dle parametrů tahů. Nabízí se např. třídit tahy podle hodnoty figury, která je zahráním takového tahu sebrána. Dále máme možnost upřednostňovat tahy, kdy co nejlevnější figury seberou ty nejcennější.

Za perspektivní můžeme považovat braní soupeřových figur, tahy na krytá pole, tahy blízko středu šachovnice, napadení soupeřových figur atd., případně jejich kombinace.

Killer heuristic

[2]. Hlavní myšlenkou této heuristiky je skutečnost, že jestliže nějaký tah byl dobrý v jedné větvi stavového prostoru, je velmi pravděpodobné, že bude podobně výhodný i v jiné větvi. Za dobré se považují ty tahy, které způsobí beta-cutoff, tzv. *killer moves*. Pro jednotlivé hloubky prohledávání si ukládáme několik killer moves a v dalších větvích je vždy prozkoumáme přednostně. Jestliže nějaký tah, který dosud nebyl označen jako killer, způsobí beta-cutooff, přepíšeme jím stávající killer move. Pokud si pro každou hloubku ukládáme více než jeden killer move, nově objevený přidáme na začátek pole a stávající posuneme na index o 1 větší, přičemž poslední ztratíme. V praxi se nejčastěji pamatují 2 tahy pro každou hloubku.

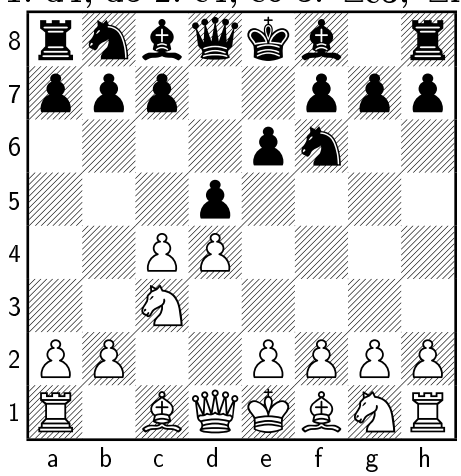
3.3.2 Použití dříve vypočtených informací

V popisovaných typech her je problematická časová složitost, ale prostorová obvykle nečiní potíže. Ukládání vypočtených informací je proto poměrně přímočará myšlenka jak zrychlit běh propočtu.

Transpoziční tabulky

Během hraní šachů si lze všimnout faktu, že dvou totožných pozic je možno docílit různými kombinacemi tahů. Tento jev označujeme jako transpozice. Objevuje se však i v dalších hrách jako dáma, reversi, piškvorky či go. Uvažme například pozici získanou po dámském gambitu:

1. d4, d5 2. c4, e6 3. ♘c3, ♘f6



stejného postavení však dosáhneme po anglickém zahájení **1. c4, ♘f6 2. ♘c3, e6 3. d4, d5** Běžné prohledávací algoritmy ohodnocují pozice bez ohledu na to, že je třeba již dříve propočítávaly. Zde se objevuje možnost ukládat si výsledky propočtů a před každým dalším prohledáváním nejprve zkontrolovat, jestli již příslušné ohodnocení nemáme uloženo. Pro takovou proceduru je vhodná hashovací tabulka, kde klíčem je daná situace na šachovnici a hodnotou je struktura obsahující její cenu. Kromě ceny pozice však potřebujeme uchovávat ještě hloubku, pro kterou byla daná cena určena. Jestliže v hloubce d narazíme na uloženou pozici, která byla ohodnocena pro hloubku $d\text{-}2$, tedy blíže ke kořeni,

můžeme tuto hodnotu klidně použít, neboť kdybychom se propočítali až do maximální hloubky, nedostali bychom přesnější ohodnocení. V opačném případě, kdy nalezená uložená pozice byla propočtena na méně půltahů dopředu, než zbývá aktuálnímu prohledávání, bychom tuto uloženou hodnotu neměli použít, protože jsme schopni ohodnotit přesněji. Poslední informace uložená v tabulce je příznak, který určuje, zda daná hodnota je přesná, nebo zda jde o dolní (alfa) či horní (beta hranici).

Důležitým prvkem této metody je volba vhodné hashovací funkce. U hashovacích funkcí obecně se snažíme o co nejmenší počet kolizí, neboli v našem případě aby se stávalo co nejméně, že dvěma různým pozicím odpovídá stejný hash. Jelikož počet přípustných pozic v šachu je přibližně [14]

$$\frac{64!}{32!(8!)^2(2!)^6} \approx 10^{43} \tag{3.8}$$

občasným kolizím se nevyhneme. Vhodné řešení poskytuje metoda známá jako Zobrist hashing [17]. Její princip využívá program Chessplayer.

Na základě aktuálního obsahu datové struktury šachovnice vytvoříme vícerozměrné pole, ve kterém zachytíme rozestavění figur a hráče na tahu. Vytvoříme si trojdimenzionální pole o rozměrech 13 (6 typů figur, 2 barvy a jedno prázdné pole), 120 (velikost datového pole ve struktuře Board, přestože využijeme pouze 64 jeho hodnot) a 2 (na tahu je vždy jeden ze dvou hráčů). Každý prvek takto vytvořeného pole naplníme 64 bitovým číslem, hodí se např. datový typ long. Tato čísla negenerujeme náhodně, ale jsou uložena v souboru. Tento způsob je zvolen díky možnosti jeho využití v databázi zahájení.

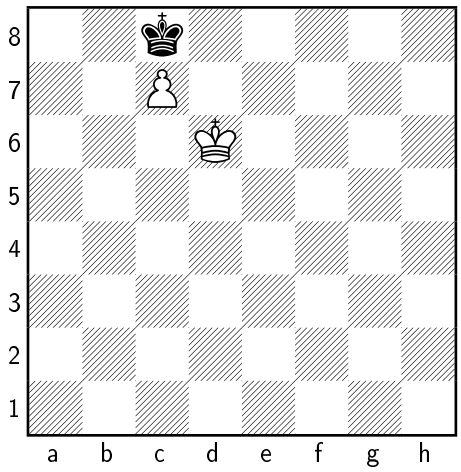
Hash aktuální pozice vytvoříme opakovaným užitím bitové funkce XOR a to následovně: procházíme všechny figury na šachovnici a navzájem provádíme xor s náhodnými čísly na příslušných souřadnicích uvnitř pole. Výsledné číslo představuje klíč v transpoziční tabulce. Pozice je určena rozestavěním figur.

3.3.3 Redukce intervalu alfa-beta

Čím užší je interval alfa-beta, tím častěji dochází k ořezávání a algoritmus pracuje rychleji. Algoritmus alfa-beta interval postupně zužuje, ale jen v případě, že je to skutečně možné. Heuristiky redukující interval jsou méně opatrné a může se stát, že nenaleznou žádný tah, jehož ohodnocení náleží do intervalu. V takovém případě musí znovu spustit prohledávání s širším intervalem, nicméně když ohodnocení padne mezi hraniční meze, znamená to úsporu času.

Null-move heuristic

Jedná se o ořezávací metodu. Využívá se zde předpokladu, že nejlepší tah z dané pozice je lepší než žádný tah (null move), i kdyby tuto možnost pravidla připouštěla [15]. Pomocí této heuristiky jsme schopni zužovat interval alfa - beta a dosáhnout tak častějšího ořezávání. Není však vyloučena situace, kdy skutečně nejlepší nehrát nic - nevýhoda tahu. V literatuře se můžeme setkat s označením *zugzwang positions*. Nevýhoda tahu se objevuje spíše v koncovkách, výjimečně ve střední hře.



Obrázek 3.3.3 zachycuje velmi jednoduchý příklad, kdy černý má nevýhodu tahu. Jediná možnost je zahrát ...♔b7, načež bílý odpoví ♔d7, nyní černému nezbývá než poodstoupit králem, například ...♔b6 a následuje nasazení dámy c8♕. Pro černého by tedy bylo nejvýhodnější nedělat vůbec nic, ale to pravidla nepovolují. V podobných situacích se Null-move heuristika chová nesprávně.

Prohledávající algoritmus musí zajistit, aby se neopakovalo volání s null-move, protože ve výsledku by nedošlo k vůbec žádnému tahu. To zajistíme tak, že prohledávacímu algoritmu posíláme navíc jako parametr booleovskou proměnnou, která má v případě volání null-move hodnotu false a v další iteraci se již použije standardní prohledávání.

Aspiration Search

Tato heuristika ve skutečnosti využívá dříve vypočtené hodnoty pozice prohledané do menší hloubky k odhadování hodnoty propočítané na více tahů dopředu. Vyžaduje tedy iterativní prohlubování. Předpokládá, že hodnota pozice určená na n tahů dopředu se nebude příliš lišit od hodnoty téže pozice prohledávané na $n+1$ tahů. V každém kroku se podívá na očekávanou hodnotu pozice a zredukuje interval alfa-beta okolo této hodnoty na $\alpha = \max(\alpha, value - window_size)$ a $\beta = \min(beta, value + window_size)$ pro pevně zvolenou hodnotu $window\_size$. Obvykle se velikost okna volí o něco menší, než hodnota pěšce.

Jestliže hodnota zkoumané pozice padne do redukovaného intervalu, dosáhlo se razantnějšího prořezávání, a tedy heuristika uspěla. V opačném případě je třeba znovu spustit prohledávání, tentokrát s úplným intervalem, který by za běžných okolností použil obyčejný alfa-beta algoritmus.

K uchovávání předešlých hodnot pozic se opět hodí transpoziční tabulky a v nich uložené hodnoty.

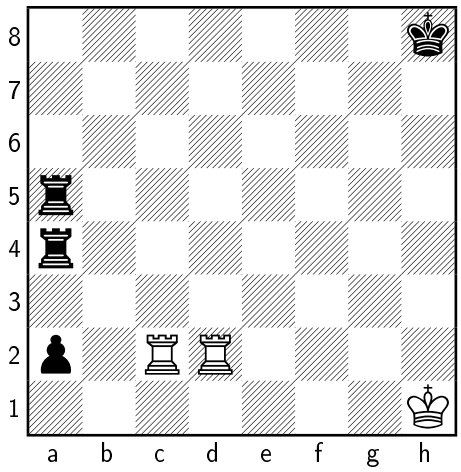
3.3.4 Hlubší propočet speciálních tahů

Na rozdíl od předešlých tříd heuristik, tato metoda nezpůsobuje zrychlení výpočtu, ale spíše jeho mírné zpomalení. Získáme však přesnější ohodnocení a můžeme tak vybrat vhodnější tah.

U různých her se setkáváme s jevem známým pod označením **horizon effect**. Při konstantní hloubce prohledávání se dopočítáme do listu a i kdybychom

následujícím tahem dostali mat, nezjistíme to, neboť takový uzel je již za tzv. horizontem.

Následující diagram zobrazuje příklad efektu horizontu. Bílý na tahu potřebuje sebrat černého pěšce, ten je však krytý dvěma věžemi.



Pro hloubku 1 najde bílý tah ♖×a2 a ten vybere jako nejlepší, neboť odpověď černého . . . ♖×a2 je již za horizontem. Pracuje - li algoritmus s počáteční hloubkou 2, ukáže se, že sebrat pěšce není dobrý nápad, neboť v zápětí příjde bílý o věž. Ovšem 3 půltahy dopředu opět budou preferovat tah ♖×a2, protože bílý má pole pokryto druhou věží. Poslední tah černou věží je za horizontem. Až prohledání na 4 půltahy určí, že tuto výměnu vyhraje černý. Metody založené na použití různé hloubky pro některé tahy snižují efekt horizontu.

Quiescence search

Člověk má obvykle dostatek intuice, aby slibné tahy promýšlel důkladněji, zatímco ty na první pohled nevhodné záhy zavrhne. Toto chování se pokouší simulovat heuristika zvaná *quiescence search*, nebo lze volně přeložit prohledávání do klidové pozice. Běžný prohledávací algoritmus má určenou hloubku prohledávání. Při použití této heuristiky se zaměříme na vhodně se jevící tahy a ty propočítáme ještě o několik dalších půltahů dopředu. Otázkou zůstává, jakým způsobem budeme posuzovat vhodnost tahu. Nejužitečnější se jeví vybírání tahů, kdy sebereme soupeři figuru.

Algoritmus 6 Quiescence search

```
function quiescence_search(position, depth)
  if position appears quiet || position is terminal || depth == 0 then
    return evaluate position
  else
    search children of actual position using recursive call of quiescenceSearch
    return best valued move
  end if
end function


function normal_search(position, depth)
  if position is terminal then
    return estimated value of position
  else
    if depth == 0 then
      if position appears quiet then
        return estimated value of node
      else
        return estimated value from quiescence_search(node, reasonable_depth_value)
      end if
    else
      search children of node using recursive applications of normal_search
      return estimated value of children
    end if
  end if
end function
```

V programu chessplayer se po dosažení předepsané hloubky prohledávaly ještě pozice, které vznikly sebráním nějaké figury. V některých případech však algoritmus pracoval neúměrně dlouho. Jakmile se však strategie nepatrně pozměnila, doba výpočtu byla již přijatelná. Změna spočívala v hlubším prohledávání pouze takových tahů, kdy došlo k sebrání figury cennější než pěšec.

Redukce v iterativním prohlubování

Tato metoda nepatří mezi známé heuristiky, ani o ní nevyšel žádný článek. Její zavedení sloužilo spíše jako menší experiment a pokus o dosažení vyšší hloubky prohledávání.

Funguje na velmi jednoduchém principu: Máme zvolenu pevnou hloubku prohledávání d. Spustí se iterativní prohlubování do hloubky $d-1$ v plném rozsahu a v posledním cyklu již neprocházíme všechny tahy, ale jen prvních n vzešlých z dynamického třídění. Samozřejmě zde existuje riziko zavrhnutí velmi dobrého tahu, jehož síla by se projevila až v poslední iteraci.

Čím nižší n zvolíme, tím více času ušetříme, ovšem samozřejmě riziko volby špatného tahu roste s klesajícím n. Dále se nabízí postupné zmenšování n s rostoucí hloubkou, nebo místo konstanty odvozovat jeho hodnotu od počtu následníků

z dané pozice a dále propočítávat jen nějakou poměrnou část.

3.4 Ohodnocování pozice

Pomocí algoritmů prohledávajících stavový prostor vybíráme nejvhodnější tah. Po dosažení listu je třeba vzniklou pozici ohodnotit vhodně navrženou funkcí. Nejjednodušší postup předpokládá prostý výpočet cen figur na šachovnici. Chceme-li posuzovat vhodnost tahu z různých hledisek, potřebujeme zachytit i další aspekty postavení figur. Během vývoje šachové partie se důležitost jednotlivých parametrů ohodnocovací funkce může měnit. Například v zahájení a střední hře se snažíme schovat krále do bezpečí, v koncovkách je mnohdy nezbytné jej poslat do středu šachovnice a využívat jeho možností.

Konkrétní hodnoty bonusů či penalizací za různé poziční jevy vycházejí zejména z prací šachových teoretiků, jejichž historie sahá daleko před první počítačové šachové programy. V některých odborných článcích autoři představují hledání optimálních parametrů nejen šachových ohodnocovacích funkcí pomocí genetických algoritmů [3, 4]. Parametry ohodnocovací funkce použité v programu Chessplayer byly převzaty z výsledků práce [5], kde se parametry funkce zjišťovaly genetickým algoritmem tak, aby se výsledná ohodnocení co nejvíce přibližovala mentorovi - silnému šachovému programu, jež uspěl na světových soutěžích počítačového šachu. Jako mentor zde byly použity programy Maestro a Falcon.

3.4.1 Cena materiálu

Program s takovou ohodnocovací funkcí hraje poměrně obstojně, dává si pozor na figury a chyby soupeře spolehlivě trestá. Nicméně jeho slabina se objeví v případě, kdy žádným tahem zásadně nemění počty figur. V takové situaci se stává, že program hraje neustále sem a tam např. věží a to samozřejmě není žádoucí stav.

3.4.2 Cena pozičních aspektů

Mezi šachisty se běžně klade důraz nejen na materiál, ale na vzájemné postavení figur. Různé poziční aspekty se dají algoritmicky poměrně snadno detekovat, ale ve srovnání s prostým výpočtem ceny materiálu jde o komplikovanější proces. Výběrem vhodných jevů a jejich ocenění je úloha spíše pro zkušeného šachistu. V šachové teorii je známo několik obecných pravidel, která mohou být zahrnuta v ohodnocovací funkci[14].

1. Pěšcové formace
    - Izolovaní a zdvojení pěšci
    - Ovládnutí středu
    - Pěšci na opačných barvách než střelci
2. Pozice figur
    - Věž na otevřeném sloupci

- Zdvojené věže
- Věž na předposlední řadě
- Vývin jezdců, zejména jsou-li kryti pěšci

3. Vazby a napadení
   - Figury potřebné pro obranu mají omezenou pohyblivost
   - Napadení polí sousedících s králem
   - Zachycení. Figura musí být zachycena méně hodnotnými figurami, než je sama.
4. Pohyblivost

3.5 Řešení speciálních situací

Šachovou partii lze typicky rozdělit do tří fází, které mezi sebou plynule přecházejí: zahájení, střední hra a koncovka. Dosud popisované algoritmy a heuristiky se dobře hodí ve střední hře, kdy je na šachovnici stále dost figur.

3.5.1 Zahájení

Během dlouhé historie vývoje šachových strategií se vyvinulo několik desítek [7] zahájení. Běžné prohledávací techniky většinou nedokáží konkurovat naučeným zahajením, a proto se nabízí změnit strategii na začátku hry. Program má uloženo několik zahájení a v případě, že některé nastane, nevolá se prohledávání, ale příslušný tah se vybírá z databáze. To samozřejmě vyžaduje přesné dodržení tahů daného zahájení. Z tohoto důvodu se šachoví velmistři hrající partie proti nejlepším šachovým strojům často nepatrně odchýlí od standardního zahájení, neboť je pravděpodobnější, že šachový program udělá chybu v prohledávacím algoritmu (viz. problémy některých heuristik), než v naučeném zahájení.

V programu Chessplayer je možnost použití popsané databáze zahájení. Kromě transpoziční tabulky, díky níž jsme schopni detekovat opakování pozic a znovu používat již vypočtené hodnoty pozic, si držíme ještě další hashovací tabulku. I zde je klíčem hash hodnota dané pozice, ovšem hodnotu tvoří příslušný tah, který by se měl v danou chvíli zahrát. Ještě před tím, než započneme samotný prohledávací algoritmus, zkontrolujeme, zda v databázi nemáme uložen nejlepší tah. Jestliže ano, prohledávání vůbec nezahajujeme a vrátíme tah, který jsme našli v databázi.

Popsaný postup však předpokládá, že při každém běhu programu patří určitý hash kód vždy k téže pozici. Z toho důvodu nelze hashovací tabulku na začátku vyplňovat náhodně, ale musíme mít náhodná velká čísla někde uložena a pokaždé použít ty stejné hodnoty. Čísla jsou uložena v externím souboru a po každém spuštění programu se hashovací tabulka vyplní čísly, které se z něj přečtou.

Kromě počátečních hodnot, jimiž se vyplní tabulka při zahájení partie, máme navíc soubor se samotnou databází zahájení. Podle obsahu tohoto souboru se vyplní datová struktura hashovací tabulky. V průběhu partie se obsah této struktury nemění.

Dalším aspektem zahájení je variabilita různých kombinací počátečních tahů. Vezměme si například Francouzskou obranu: **1. e4, e6 2. d4, d5**. Jak bílý, tak černý má ze vzniklé pozice mnoho možností, jak pokračovat. Za bílého můžeme zahrát například 3.e5, nebo 3.♘d2, případně 3.♘c3. Nabízí se samozřejmě mnohem více kombinací. Aby se zamezilo vybírání neustále stejného postupu v zahájení, má každý tah, který najdeme v hashovací tabulce, určitou pravděpodobnost, se kterou jej program zahraje. Snažíme se o to, aby pravděpodobnost byla přibližně úměrná obvyklosti tahu v dané pozici. Jestliže databáze nabízí pouze jediný tah, zahraje se určitě. Pokud jsou dvě možná pokračování, vybere se některé v závislosti na jejich pravděpodobnostech. Tím bylo dosaženo různorodých zahájení.

3.5.2 Koncovka

V koncovce zbývá již jen málo figur na šachovnici, tak by se mohlo zdát, že možných tahů k prohledávání bude méně a algoritmus by tak mohl dopočítat hru až do konce. Bohužel tomu tak většinou nebývá, neboť figury dalekého dosahu (dámy, střelci, věže) mají mnoho možností, kam se posunout, tzn. průměrný větvící faktor se zásadně nezmění. Řešení koncovek s 5 figurami a některé se 6 figurami jsou známa, takže přímočarý postup spočívá v zavedení databáze koncovek. Když se šachový program obsahující databázi ocitne v některé z uložených situací, nepoužívá žádný prohledávací algoritmus, ale jednoduše vybírá tah uložený v databázi. Pokud ale z nějakého důvodu nechceme využívat databáze, můžeme se v některých případech pokusit algoritmizovat postup vedoucí k výhře. V programu Chessplayer jsou tímto způsobem implementovány koncovky, kde na jedné straně zůstal samotný král a silnější soupeř má krále a spolu s ním ještě věž, nebo dámu.

4. Měření a statistiky

Existující šachový software dosahuje vynikajících výsledků, dokonce v dnešní době šachové programy vítězí nad nejlepšími šachovými velmistry. Hlavní motivací programu Chessplayer tedy není implementovat další šachovou aplikaci, ale spíše demonstrovat chování šachového programu pro zvolené parametry prohledávacích algoritmů a různé kombinace použitých heuristik.

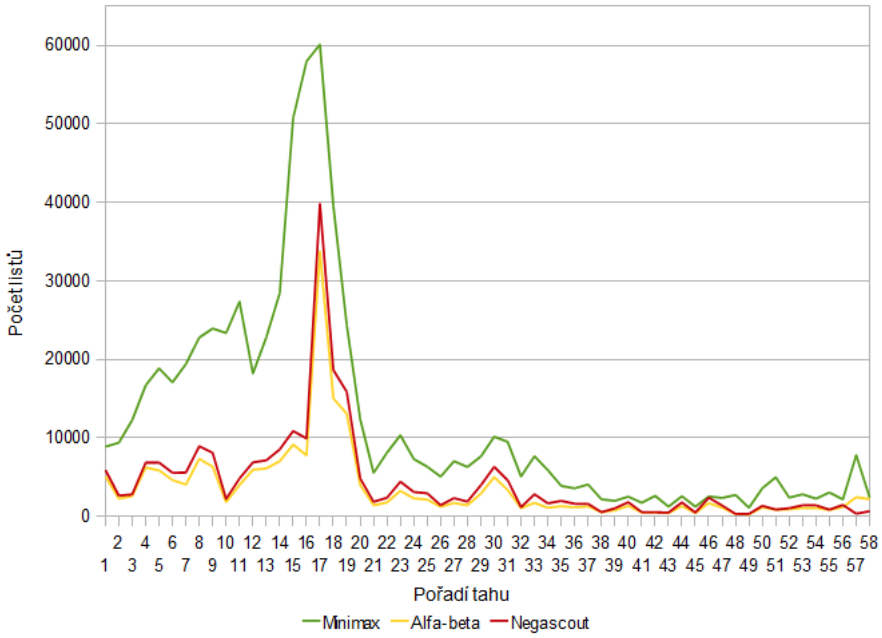
Chessplayer nabízí dva režimy běhu: běžnou hru jednoho hráče proti počítači a tzv. autoplayer, kdy dvě strategie hrají proti sobě. Po spuštění nové hry se předpokládá, že uživatel bude hrát svými figurami a aplikace funguje jako běžný šachový program. Kdykoli během partie je možno spustit autoplayer a sledovat, která strategie zvítězí, případně zda partie skončí remízou.

Užitečným ukazatelem efektivity zvoleného algoritmu je počet volání ohodnocovací funkce, neboli počet listů herního stromu v maximální hloubce. Jestliže používáme algoritmus omezený hloubkou, ale neomezeným časovým limitem, menší počet listů znamená častější prořezávání a tedy vyšší efektivitu.

Pro provedená měření je nezbytný požadavek zcela totožné partie pro všechny srovnávané algoritmy. I malá odchylka od nějaké pozice může způsobit významně odlišná data. Algoritmy s použitím různých třídících postupů mohou v jedné situaci preferovat různé tahy, pokud mají stejnou hodnotu. Partie byly proto voleny tak, aby k těmto jevům nedocházelo.

4.1 Základní algoritmy

První měření se týká výkonnosti základních algoritmů bez použití přídavných heuristik. V kapitole 3 bylo uvedeno, že výhoda algoritmu Negascout se projeví až v případě vhodného uspořádání tahů. Zde se však tahy nijak netřídily a potvrdila se skutečnost, že v takovém případě je obyčejná Alfa-beta nepatrně úspěšnější.



Obrázek 4.1: Graf vývoje počtu listů za použití jednotlivých algoritmů v průběhu šachové partie. Data byla pořízena bez použití třídění tahů a při zvolené hloubce 3 půltahy.

V průměrném případě ohodnocoval Negascout o 61% méně pozic než obyčejný Minimax a Alfa-beta dokonce o 67% méně. V případě tří půltahů jsme si tedy složitějším Negascoutem oproti obyčejnému Alfa-beta algoritmu nepolepšili.

Jak ale ukázalo další měření, efektivita Negascoutu je úměrná použité hloubce prohledávání. Experiment se prováděl následujícím způsobem: Rozehrála se běžná partie tak, že z každé vzniklé pozice, kdy byl na tahu uživatel, se vybral jeden tah a ten se zahrál, načež se čekalo na odpověď soupeře, přičemž hloubka prohledávacího algoritmu byla nastavena na 3 půltahy dopředu. Jakmile soupeř zareagoval, vrátily se 2 půltahy zpátky, tedy hra je opět v původní situaci. Prohledávací hloubka soupeře se zvýší na 4 a uživatel zahraje opět tentýž tah. Odpověď soupeře může být stejná, jako v případě tří půltahů, ovšem může se i lišit. Po jeho odpovědi se znovu vrátily 2 půltahy a celý proces se zopakoval znovu, pouze soupeři byla nastavena hodnota 5 půltahů. Při každém tahu se sledoval počet volání ohodnocovací funkce (počet listů).

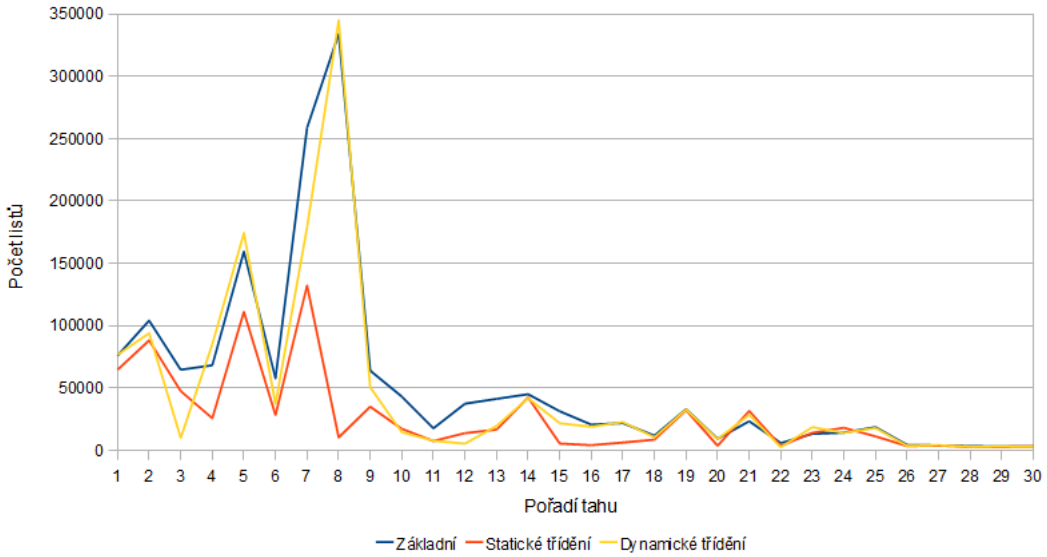
Tímto způsobem se zajistilo, že byla použita porovnatelná data. Kdyby byl experiment proveden tak, že by se spustil režim Autoplayer od výchozí pozice postupně pro 3, 4 a 5 půltahů bez ohledu na to, jaké tahy jsou vybírány, partie by nebyly totožné, a proto by porovnávání algoritmů nebylo možné.

Pro 4 půltahy byly oba algoritmy srovnatelné: Negascout ohodnocoval průměrně 98% listů ohodnocovaných Alfa-betou. Jakmile byla hloubka nastavena na 5 půltahů, Negascout byl o 10% lepší.

4.2 Třídění tahů v iterativním prohlubování

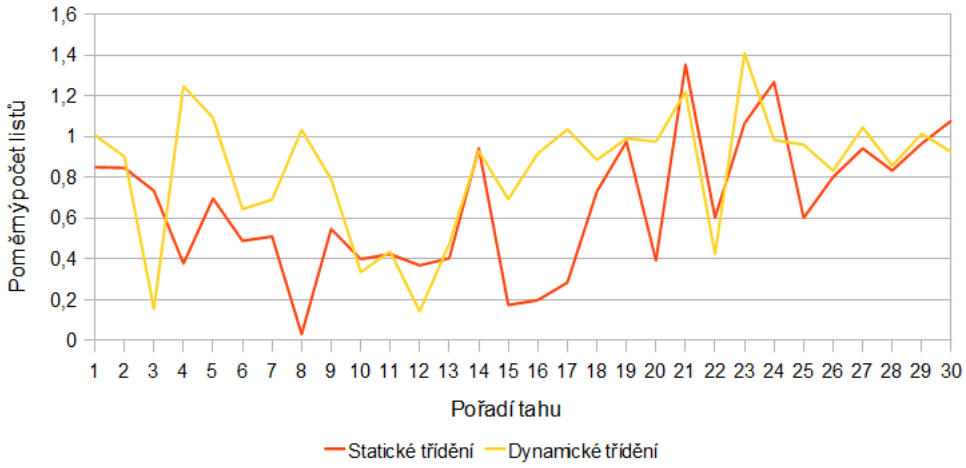
V předešlé kapitole byl vysvětlen princip iterativního prohlubování. Samo o sobě nepřináší žádné zrychlení. Přesto je tato technika nezbytná, pokud má být umožněno omezit čas prohledávání, nebo okamžité vynucení odpovědi uživatelem. Rovněž byl popsán význam třídění tahů a možné přístupy.

V tomto experimentu se měřil počet volání ohodnocovací funkce pro různé varianty iterativního prohledávání. Měření sestává ze dvou různých partií, každá byla hrána nejprve s obyčejným iterativním prohlubováním, poté se přidalo statické třídění a nakonec dynamické třídění. Pokaždé byla hloubka omezena na 4 půltahy a ohodnocovací funkce zahrnovala několik pozičních jevů.



Obrázek 4.2: Vývoj počtu listů během první partie

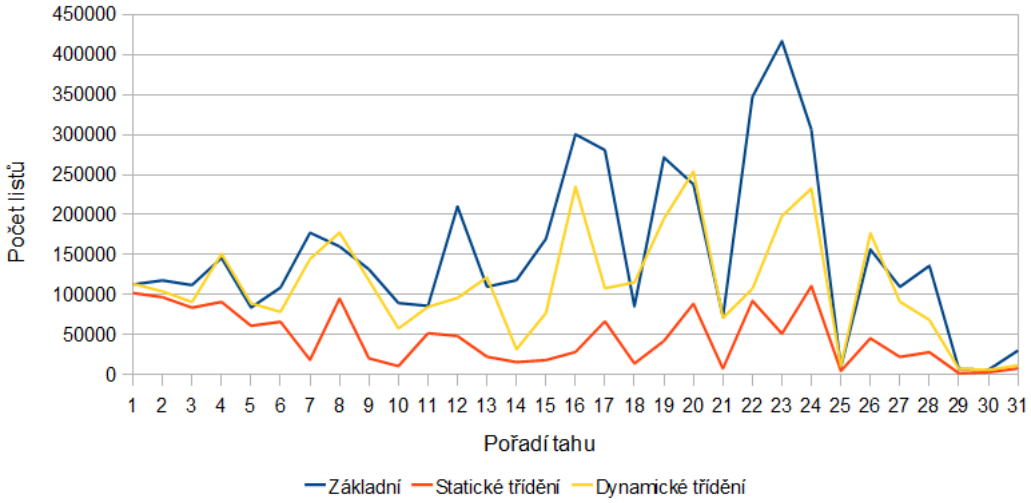
Vývoj počtu volání ohodnocovací funkce během první partie zobrazuje obrázek 4.2. Statické třídění vykazuje zřetelně úspěšnější výsledky, zatímco dynamické přináší jen ne příliš výrazné vylepšení. Někdy je dokonce horší než čistý algoritmus.



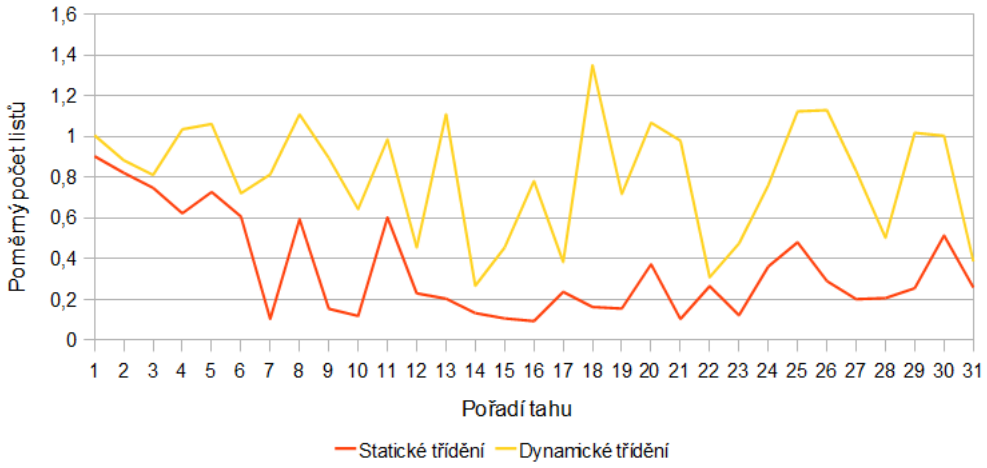
Obrázek 4.3: Průběh poměru počtu listů při iterativním prohlubování s tříděním tahů vůči netříděnému během první partie.

Ze stejné partie vychází data na obrázku 4.3. Zde graf ukazuje vývoj poměru počtu listů při použití třídících technik v iterativním prohlubování vůči netříděnému iterativnímu prohlubování. Statické třídění provádělo v průměru o 34% a dynamické o 16% méně ohodnocovacích funkcí než netříděný algoritmus.

Ve druhé zkoumané partii byly rozdíly ještě výraznější. Oproti první partii byla otevřenější a větvící faktor výrazně vyšší.



Obrázek 4.4: Vývoj počtu listů během druhé partie



Obrázek 4.5: Průběh poměru počtu listů při iterativním prohlubování s tříděním tahů vůči netříděnému během první partie.

Za pozornost stojí skutečnost, že během celé druhé partie dosahovalo statické třídění lepších výsledků než základní algoritmus, dokonce nikdy nebylo horší než dynamické třídění. Statické třídění zaznamenalo oproti základnímu algoritmu vylepšení o 65%, dynamické třídění o 19%.

4.3 Efektivita transpozičních tabulek

Poslední měření se týká heuristiké metody transpozičních tabulek. Cílem experimentu je zjištění, jakou míru zrychlení tato technika přinese v závislosti na použité hloubce prohledávání. Zřejmě se zvyšující se hloubkou prohledávání bude růst i procento transpozic, protože existuje více způsobů, jak dosáhnout určitého postavení.

Sledoval se celkový počet uzlů a počet listů v jednotlivých variantách algoritmu. Výsledek testování shrnuje následující tabulka. Pro obě varianty algoritmu se zaznamenával průměrný počet uzlů a listů, přičemž listy jsou zahrnuty též mezi uzly. U transpozičních tabulek je navíc uveden průměrný počet uzlů, ve kterých byla hodnota pozice nalezena v transpoziční tabulce.

	Základní alfa-beta		alfa-beta s transpoziční tabulkou		
Hloubka	**Uzly**	**Listy**	**Uzly**	**Listy**	**Transpozice**
2	589,2	556,3	589,2	555,7	0,6
3	8972	8575	8972	6092	2484
4	71424	66278	51952	33213	14924
5	1437291	1346378	863765	517679	291708

V případě hloubky dvou půltahů je tato hodnota zanedbatelná. Za povšimnutí stojí hloubka 3 půltahy, kde oba algoritmy mají stejný počet uzlů, ale transpoziční tabulky mají méně listů. To je způsobeno tím, že úspěšné nalezení hodnoty v tabulce se stávalo pouze po dosažení maximální hloubky. Tedy herní strom se nezměnil, pouze již nebylo třeba volat ohodnocovací funkci, ale jednoduše vrátit

hodnotu z tabulky. Za použití větších hloubek již byla úspora zajímavější: Transpoziční heuristika za použití 4 půltahů ušetřila 23% uzlů, u 5 půltahů úspora činila 37 %.

Testování se provádělo na dvou algoritmech. V prvním případě šlo o obyčejnou alfa-betu bez iterativního prohlubování a bez dalších heuristik. Ve druhém případě byla přidána transpoziční tabulka. Nejdříve se hloubka nastavila na 2 půltahy dopředu a algoritmus vybíral vhodný tah postupně za 40 různých postavení. Stejná série pozic se předkládala Postupně se hloubka zvyšovala až na 5 půltahů a pro každou hloubku se testovala stejná série pozic.

Na závěr ještě krátká poznámka: předpokládejme dva algoritmy, z nichž jeden využívá transpozičních tabulek a druhý nikoli. Oba hledají nejlepší tah ze stejné pozice a jejich odpovědi se mohou lišit. K takovému jevu dochází v případě, že v jedné pozici využijeme hodnotu propočítanou na více tahů dopředu, než kolik by počítal samostatný algoritmus. Tudíž tato pozice je ohodnocena přesněji a hodnota se může značně lišit, což způsobí rozdílné odpovědi sledovaných algoritmů. Kdyby se používaly jen ty hodnoty nalezené v tabulce, které jsou propočítány na právě propočítávaný počet půltahů, odpovídaly by oba algoritmy stejnými tahy.

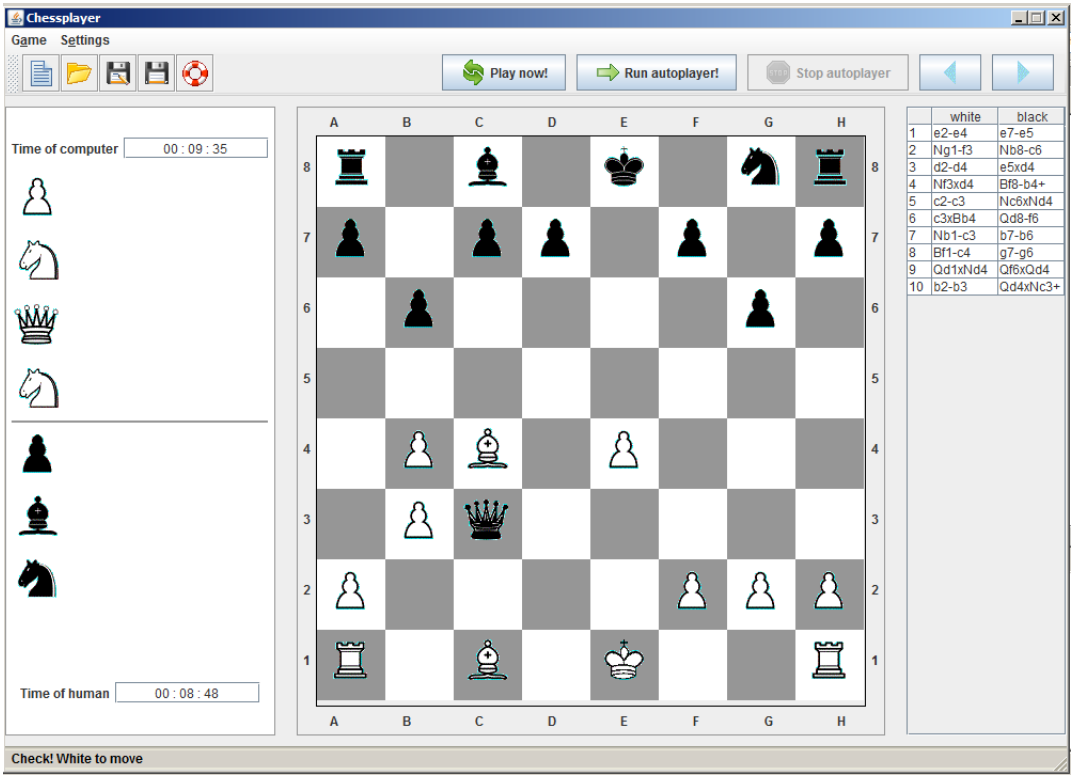
5. Uživatelská část

Koncepce programu chessplayer předpokládá dvě cílové skupiny potenciálních uživatelů. Do první skupiny spadají asi dle očekávání ti, kteří si chtějí zahrát šachy proti počítači a nepotřebují znát princip fungování programu. Zástupci druhé skupiny se zajímají spíše o chování šachových algoritmů a chtějí si vyzkoušet různé jejich kombinace.

Pro spuštění programu z příkazové rádky je třeba se přesunout do adresáře `../Chessplayer/dist` a spustit příkaz `java -jar Chessplayer.jar"`.

5.1 Popis grafického rozhraní

Aplikační okno sestává z hrací desky, levého postranního panelu pro vyřazené figury, pravého postranního panelu pro zobrazení šachové notace odehraných tahů, stavového řádku, nástrojové lišty a aplikačního menu.



Obrázek 5.1: Kompletní ohodnocení herního stromu do hloubky 4 půltahy pomocí algoritmu minimax. V jednotlivých hladinách se střídavě hodnota minimalizuje a maximalizuje v závislosti na hráči na tahu

Aplikační menu

Okno programu obsahuje běžné uživatelské menu se dvěma položkami: **Game** a **Settings**. V první uvedené položce jsou volby umožňující spustit novou, načíst

uloženou, nebo uložit stávající hru. Dále jsou zde volby pro zobrazení nápovědy, informací o programu a ukončení aplikace. Ve druhé položce máme pouze 2 volby. Umožňují nastavovat parametry umělé inteligence počítačového soupeře **AI parameters (opponent)** a rovněž figur hráče **AI parameters (only with autoplayer)**. Tato volba má však smysl pouze při spuštění režimu *autoplayer*.

Panel nástrojů

Panel nástrojů se nachází pod aplikačním menu a obsahuje dvě skupiny tlačítek. Čtvercová tlačítka s ikonami bez popisu provádějí některé akce rovněž spustitelné z položky **Game** aplikačního menu. Po najetí myši na příslušné tlačítko se zobrazí popisek akce, která se provede po jeho stisknutí. Zbývající tlačítka umožňují spouštět a zastavovat režim autoplayer. tlačítko **Play now!** způsobí okamžitou odpověď počítačového soupeře. Poslední dvě tlačítka dovolují vrátit zahraný tah (šipka doleva), nebo přinutí hráče na tahu zahrát (šipka doprava).

Hrací deska

Napodobuje standardní šachovnici opatřenou souřadnicemi A až H pro označení sloupců a 1 až 8 pro označení řádků. Šachovnice respektuje konvence pro svou orientaci: Jestliže hráč vybere bílé figury, v pravém dolním rohu leží pole h1, naopak když hrajeme za černé, v pravém dolním rohu je pole a8. Před postavením figur je šachovnice orientovaná jakoby očekávala bílé figury, pokud ale nakonec zvolíme černé, orientace se opraví.

Panel sebraných figur

Vlevo od šachovnice je umístěn panel rozdělený na dvě části. V průběhu partie se na něj pokládají sebrané figury. Figury, které počítačovému soupeři sebral uživatel, se zobrazují blíže k jeho straně, tzn. ve spodní části pravého panelu, zatímco figury, jež byly sebrány počítačovým soupeřem, mají své místo v části horní. V případě, že jsme na začátku zaškrtli možnost používat šachové hodiny, zobrazí se rovněž v pravém panelu spolu s popisem, kterému z hráčů daný čas náleží.

Panel notace

V pravé části aplikačního okna je umístěna tabulka se třemi sloupci, do které se zaznamenává průběh partie. První sloupec obsahuje číslo tahu, druhý sloupec plnou algebraickou notaci půltahu bílého hráče a třetí totéž pro černého hráče. Nad tabulkou notace jsou zobrazena dvě tlačítka se šipkami vlevo a vpravo, pomocí kterých můžeme vrátit 1 tah (šipka vlevo), nebo přinutit hráče na tahu, aby zahrál (šipka vpravo).

Stavový řádek

Ve spodní části si lze všimnout tenkého pruhu, který informuje uživatele, která barva figur je na tahu nebo co se zrovna děje na hrací desce. Zprávy informativního charakteru (barva figur na tahu, šach) se zobrazují černým písmem, chybové

hlášky (nemožný tah, nesmyslné nastavení výchozích parametrů hry) mají červenou barvu.

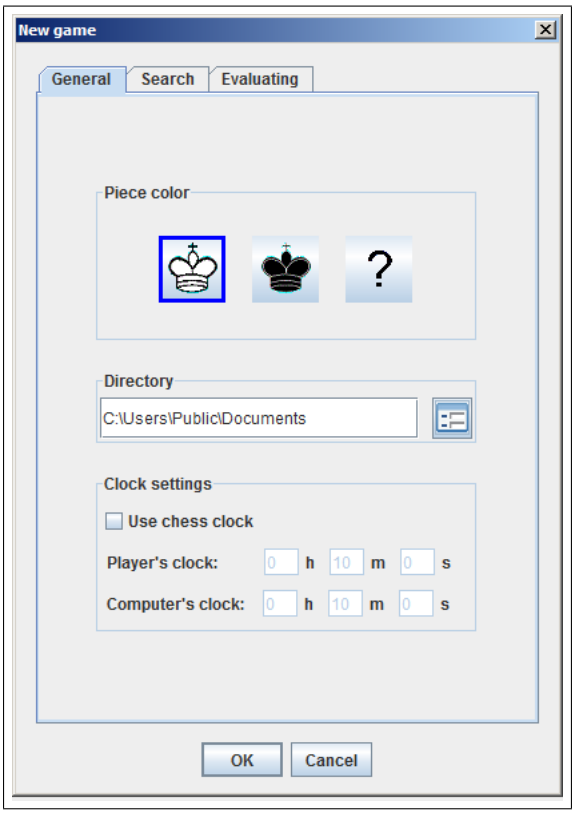
5.2 Hraní šachové partie

5.2.1 Zahájení hry

Po spuštění aplikace se zobrazí pouze prázdná šachovnice. Pro zahájení partie vybereme položku **New** menu **Game**, nebo klikneme na příslušnou ikonu v panelu nástrojů. Zobrazí se dialogové okno vyzývající ke specifikaci vlastností nové hry. Dialogové okno obsahuje tři záložky označené **General** pro všeobecné nastavení, **Search** pro nastavení umělé inteligence počítačového soupeře a **Evaluating** pro volby parametrů ohodnocovací funkce.

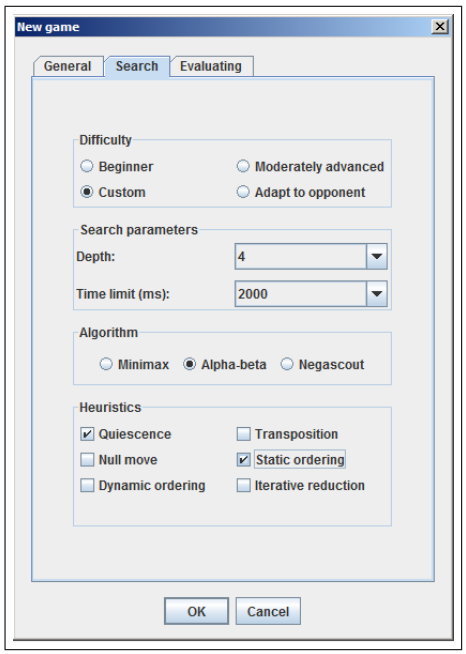
Všeobecné nastavení

V části **Piece color** určíme barvu našich figur. Implicitně je nastavena na bílé, pokud si přejeme černé, označíme tlačítko s černým králem. Jestliže nám na barvě nezáleží, můžeme použít tlačítko s otazníkem a barva figur bude vybrána náhodně. Sekce **Directory** vyzývá ke stanovení adresáře pro uložené hry. Implicitně je přednastavena domovská složka uživatele. Poslední část **Clock settings** zajišťuje nastavení šachových hodin. Abychom je při hře mohli využít, je třeba zaškrtnout políčko **Use chess clock**. Jestliže tak učiníme, bude nám umožněno oběma hráčům nastavit počet hodin, minut a sekund. **Player's clock** označuje parametry pro uživatele, zatímco **Computer's clock** pro počítačového soupeře.



Nastavení umělé inteligence

Pro nastavení míry obtížnosti šachové hry vybereme buď jednu z přednastavených možností **Beginner** (začátečník) a **Moderately advanced** (mírně pokročilý), nebo nastavíme parametry prohledávacího algoritmu ručně. Zvolení možnosti **Custom** zpřístupní pokročilejší možnosti nastavení umělé inteligence. Poslední položka **Adapt to opponent** způsobí, že se soupeř bude snažit, aby partie byla co nejvyrovnanější, tedy přizpůsobuje se schopnostem uživatele.

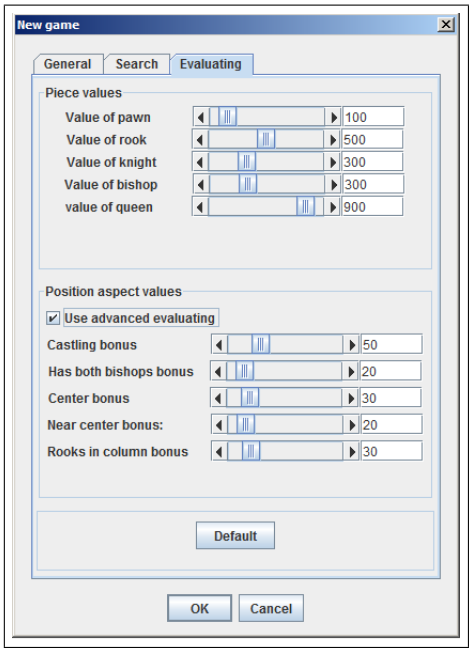


V části **Search parameters** nastavujeme hloubku prohledávacího algoritmu, jinými slovy, na kolik půltahů dopředu propočítává algoritmus svůj tah. Dále lze určit maximální počet milisekund na přemýšlení. Nastavením tohoto parametru zajistíme odpověď soupeře nejpozději po uplynutí stanovené doby, bez ohledu na nastavené hloubce prohledávání. Jestliže algoritmus najde svůj tah před vypršením limitu, je jeho odpověď rychlejší.

Výběrem konkrétního prohledávacího algoritmu se povolí heuristiky pro nastavený algoritmus. Zaškrtnutím možnosti **Use move refreshing** se zpřístupní tlačítko Play Now! na panelu nástrojů a uživatel pak může přinutit soupeře k okamžité odpovědi. Nastavením kombinace použitých heuristik se přesněji specifikuje prohledávací algoritmus.

Nastavení ohodnocovací funce

Chování počítačového soupeře lze rovněž ovlivňovat nastavením parametrů ohodnocovací funkce na záložce **Evaluating** startovacího dialogu. Jejich konkrétní hodnoty se navenek projevují tak, že vysoce ohodnocené aspekty se považují za důležité a program se na ně zaměřuje. Asi nejjednodušším příkladem mohou posloužit hodnoty jednotlivých typů figur. Jejich přednastavené hodnoty jsou voleny dle doporučení šachových teoretiků, nicméně program umožňuje jejich variabilitu. Každému ohodnocovanému prvku přísluší posuvník, pomocí něhož se nastaví požadovaná hodnota. Každému posuvníku přísluší popisek jevu, který nastavuje. Aktuálně nastavená hodnota se zobrazuje vždy vpravo od daného posuvníku. Nastavení lze kdykoli resetovat stisknutím tlačítka default.



Kromě materiálu na šachovnici se též může přihlížet k různým pozičním aspektům. Těch lze vymyslet celá řada, pro demonstraci v programu Chessplayer slouží bonus za provedenou rošádu, oba střelce na šachovnici, 2 věže v jednom sloupci a zvýhodnění figur obsazujících střed šachovnice. Pro jejich zahrnutí do ohodnocovací funkce se zaškrtne volba **Use advanced evaluating**. Pokud tato volba zůstane nezaškrtnutá, program nebere ohled na uvedené aspekty a zaměřuje se pouze na hodnoty figur, které se zohledňují v každém případě.

5.2.2 Průběh partie

V závislosti na vybrané barvě figur se od uživatele očekává provedení tahu, pokud si vybral bílé, nebo zahraje počítačový protihráč a hráč táhne až jako druhý, když má černé. Jakmile jsme na řadě a máme rozmyšlený tah, který zahrajeme, klikneme levým tlačítkem myši na příslušnou figuru. Jestliže jsme vybrali naši figuru, označí se její políčko oranžovou barvou. Následuje kliknutí na políčko, kam chceme figuru přesunout. Oranžové políčko nabude svou původní barvu a pokud je tah přípustný, figura se přesune na zvolené cílové políčko. V opačném případě nás text ve stavovém řádku informuje o snaze o nepovolený tah a pokusíme se zahrát jinak. Tímto způsobem se hráči střídají, dokud nenastane některý z cílových stavů partie.

Zvláštní pozornost si zaslouží rošáda. Při ní se totiž přemisťují dvě figury (král a odpovídající věž), nicméně uživatel zadá pouze tah krále způsobem, který jsme popsali v předešlém odstavci. Program rozpozná požadavek na rošádu a automaticky přemístí věž na správné políčko.

Oproti běžnému hraní partie máme k dispozici funkci vrácení a napovězení tahu. Odehrané tahy můžeme vracet až do základní pozice, ovšem pouze tehdy, pokud jsme nezahájili hru načtením pozice ze souboru. V opačném případě lze tahy vracet pouze do výchozí pozice.

Funkci nápovědy lze využít kdykoliv je uživatel na tahu. Stisknutí šipky doprava se provede tah pomocí prohledávacího algoritmu s parametry nastavenými pro režim autoplayer. Pokud uživatel provede svůj tah tímto způsobem, počítačový soupeř neodpoví a je třeba jej přimět k zahrání tahu dalším stisknutím šipky vpravo. Jakmile uživatel zahraje tah běžným způsobem, tedy posunutím figury na šachovnici, soupeř již odpovídá automaticky.

Poslední zvláštností je možnost vynucení okamžité odpovědi počítačového soupeře. V závislosti na nastavených parametrech prohledávacího algoritmu oponent potřebuje určitý čas pro výběr svého tahu. Pomocí tlačítka **Play now!** na panelu nástrojů máme možnost přimět soupeře k okamžité odpovědi. Tato funkce je však dostupná pouze, pokud jsme v úvodním dialogu před zahájením partie zaškrtli volbu **Use move refreshing**.

5.2.3 Ukončení partie

Jakmile dosáhneme některého z cílových stavů, objeví se dialogové okno s informací o výsledku partie a dotazem, zda si přejeme partii uložit. Na základě uživatelské volby se provede požadovaná akce a aplikace se dostane do výchozího stavu, stejně jako po jejím spuštění.

5.2.4 Ukládání partie

Rozehranou hru lze kdykoli uložit do souboru a později znovu načíst. Nabízí se uložení buď aktuálního rozestavění figur, nebo posloupnosti tahů vedoucí do požadované pozice. V případě, že bychom se omezili pouze na ukládání rozehraných partií, možnost uložení pouze rozložení figur by byla zbytečná. Vystačili bychom si pouze s posloupností tahů. Někdy je však vhodné mít možnost načíst pozici, do které jsme se třeba nikdy předtím nedopracovali v žádné odehrané partii, ale je to

například šachová úloha z knihy, nebo jednoduše pozice, kterou chceme vyzkoušet. Oba režimy ukládání využívají textový soubor s daným formátem, uživateli je tak umožněno vytvářet uložené hry, aniž by kdy nastaly.

Uložení posloupnosti tahů

Soubor s uloženou posloupností tahů má příponu *.cpm a první řádek obsahuje text humansColor 1 (pokud má uživatel bílé figury), nebo humansColor -1 (pokud má černé). Následující sekvence řádků je vlastně uložení obsahu panelu notace, přičemž úplná algebraická notace každého tahu leží na samostatném řádku. Soubor s uloženou posloupností tahů může vypadat takto:

```
humansColor -1
e2-e3
d7-d5
Qd1-g4
Bc8xQg4
Bf1-b5+
c7-c6
Bb5xc6+
b7xBc6
Ng1-f3
```

Informace obsažené v souboru určují, který hráč je na tahu po načtení partie. V případě uživatele jednoduše zahrajeme tah a následuje odpověď soupeře, pokud by měl táhnout soupeř, automaticky svůj tah nezahraje. Je třeba ho zavolat stisknutím tlačítka "next move". Tento způsob byl zvolen proto, aby před odpovědí soupeře bylo hráči umožněno nastavit parametry umělé inteligence. Během vykonávání sekvence tahů se předpokládá, že jsou tahy přípustné a notace je zaznamenána korektně.

Uložení samotné pozice

Příslušný textový soubor má příponu *.cps a stejně jako v předešlém případě, i zde je na prvním řádku uložena informace o barvě figur uživatele. Další řádky obsahují údaj o možnosti rošády ve tvaru canHumansCastle true/false v případě hráče a canCompCastle true/false v případě počítačového soupeře. Následující řádky popisují polohy figur na šachovnici. Jsou ve tvaru pos index_na_šachovnici označení_figury. Po načtení je na tahu vždycky bílý hráč, ať už je to počítač, nebo uživatel. Jestliže má bílé figury počítač, musíme ho opět přimět k tahu pomocí tlačítka "next move". Následující příklad ukazuje soubor s uloženou pozicí koncovky dvou černých věží:

texttt humansColor 1
canHumanCastle false
canCompCastle false
pos 27 -6
pos 28 -6

pos 55 -3
pos 35 3

5.2.5 Načítání ze souboru

Pro načtení hry ze souboru je třeba zvolit položku **Load game** v menu **Game**. Soubory, které nás v souvislosti s načítáním partie zajímají, mají příponu *.cps nebo *.cpm. Soubory s první příponou obashují popis rozmístění figur. Druhou příponu mají soubory s uloženou posloupností tahů zapsaných algebraickou notací.

Jestliže začínáme hrát partii z načtené posloupnosti tahů, můžeme ji kdykoli znovu uložit oběma způsoby. V případě partie načtené jako pozici ji už nemůžeme uložit jako posloupnost tahů, protože není znám průběh partie od základního postavení. Máme pouze povoleno ji uložit opět jako pozici.

Po načtení partie není možné používat šachové hodiny, ani v případě, že v době uložení byly použity. Poznamenejme ještě, že jak v případě otevírání uložené situace, tak v případě posloupnosti tahů se hra otevře s těmi parametry umělé inteligence a ohodnocovací funkce, které byly nastaveny před načtením. Nyní následuje popis formátů obou typů souborů.

5.2.6 Použití šachových hodin

Uživateli je k dispozici časomíra zastávající funkci šachových hodin. Jestliže novou hru nastavíme tak, že chceme používat hodiny, zobrazí se uplynulý čas obou hráčů na panelu sebraných figur. Časomíra v horní části tohoto panelu označuje zbývající čas soupeře, v dolní části náleží uživateli. Stejně jako v běžných šachových partiích, jestliže některému z hráčů vyprší čas, partie končí a dotyčný hráč prohrál i v případě, že má na šachovnici výraznou převahu. Pokud nastane jasně remízová pozice (tzv. mrtvá pozice), tak hra skončí ještě dříve, než někomu uplyne čas. Na počátku partie mohou mít oba hráči rozdílný čas, přestože to v šachu není obvyklé. Využijeme této možnosti v případě, že chceme zvýhodnit jednoho z hráčů.

5.3 Další funkce programu

Kromě hraní šachů lze program Chessplayer využít ještě pro demonstraci chování umělé inteligence. K tomu slouží tzv. režim **autoplayer**. Jakmile je tento režim spuštěn, program hraje za obě strany. Kdykoliv během hraní je možno tento režim zastavit a pokračovat v partii s původními figurami. Program se zastaví vždycky tak, aby byl uživatel na tahu. Z toho důvodu po zastavení se ještě může zahrát jeden tah automaticky.

Každá strana může používat jiný prohledávací algoritmus. Pro modifikaci jeho parametrů vybereme příslušnou položku v menu **Settings**. Pro nastavení parametrů strany, která byla původně soupeřem uživatele zvolíme položku **AI parameters (opponent)**, parametry původně uživatelských figur se mění skrze položku **AI parameters(Only with autoplayer)**

6. Programátorská dokumentace

Program Chessplayer je napsán v jazyku Java verze 1.6.0_18. Grafické rozhraní bylo zajišťěno technologií Java Swing. Tato sekce obsahuje vysvětlení principů fungování programu, popis jednotlivých tříd a jejich důležitých metod.

6.1 Struktura programu

Zdrojový kód je členěn do několika logicky souvisejících balíků:

- chessplayer - horní úroveň. Obsahuje třídy zajišťující reprezentaci hry v paměti, generování tahů a jejich reprezentaci a uchovávání
- chessplayer.actions - třídy s akcemi, které se provádí jako reakce na různé uživatelské události
- chessplayer.clock - šachové hodiny
- chessplayer.exceptions - hlavně I/O výjimky při načítání a ukládání partií do souborů
- chessplayer.gui - prvky grafického rozhraní programu
- chessplayer.search - prohledávacími algoritmy a s nimi související třídy

6.2 Reprezentace stavu hry

6.2.1 Datová struktura šachovnice

Hrací deska je v paměti reprezentována jako jednorozměrné pole o velikosti 120 s celočíselnými prvky. Přestože skutečná šachovnice má rozměry 8 x 8, tedy 64 polí, je zvolená struktura opodstatněná. Kdyby se používalo pouze 64 prvkové pole, muselo by se při každém generování přípustných tahů testovat, zda se nepokoušíme přistoupit mimo rozsah pole. Tento způsob uložení zajišťuje, že okrajová pole se navenek chovají jako obsazená, tedy takový tah se zamítne.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109
110	111	112	113	114	115	116	117	118	119

-10	-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-6	-2	-5	-7	-3	-5	-2	-6	-10
-10	-1	-1	-1	-1	-1	-1	-1	-1	-10
-10	0	0	0	0	0	0	0	0	-10
-10	0	0	0	0	0	0	0	0	-10
-10	0	0	0	0	0	0	0	0	-10
-10	0	0	0	0	0	0	0	0	-10
-10	1	1	1	1	1	1	1	1	-10
-10	6	2	5	7	3	5	2	6	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10	-10

Obrázek 6.1: Rozložení indexů v jednorozměrném poli reprezentujícím šachovnici (vlevo) a obsah pole na začátku partie (vpravo).

Čísla uložená v poli určují typ odpovídajícího políčka na šachovnici, nebo jaká figura na něm stojí. Prázdné pole je označeno hodnotou 0, okrajové nepřípustné pole -100, Bílé figury budou zaznamenány pomocí kladných čísel, černé záporných. Konkrétní hodnoty volíme následovně:

1 - pěšec
2 - jezdec
3 - král
5 - střelec
6 - věž
7 - dáma

Mohlo by se zdát, že nesmyslně vynecháváme číslo 4, ale i toto má své opodstatnění. Provedeme-li bitový AND hodnoty figury s číslem 4, který vydá nenulový výsledek, získáme tak figuru, která je schopna se v jednom tahu přesunout přes celou šachovnici, pokud má volnou cestu. Tyto vlastnosti má střelec, věž a dáma. Když dále tyto figury bitově násobíme s číslem 1, dostaneme při nenulovém výsledku dámu a střelce, což jsou figury s možností pohybu po diagonále a konečně při bitovém násobení dvojkou obdržíme věž a dámu, které se mohou posouvat vertikálně a horizontálně.

Metoda `playMove(OneMove move, boolean solveCastling)` zajišťuje aktualizaci datových struktur po odehrání jednoho tahu `move`. Kromě přepsání příslušných indexů třídy Board se navíc ještě zavolá metoda move třídy PiecesOnBoard.

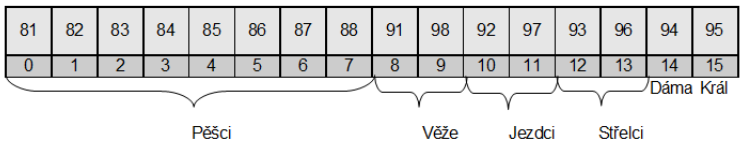
Velmi často se využívá metoda `public void revertMove(OneMove move)`, která naopak vrátí tah zpátky. Toho se využívá při výpočtu nejlepšího tahu, nebo když se uživatel rozhodne vrátit tah.

6.2.2 Polohy figur na šachovnici

K jednotlivým figurám na šachovnici je nutné přistupovat pokaždé při generování přípustných tahů, což se děje velmi často. Procházení celé struktury šachovnice a hledání figur trvá zbytečně dlouho. Proto byla vytvořena dvě jednorozměrná pole s celočíselnými položkami, které slouží jako ukazatelé na obsazená políčka šachovnice. Každému hráči přísluší jedno pole. Velikost těchto polí je 16 a nikdy se nemění. Nabízí se užití dynamické datové struktury (např. v Javě ArrayList nebo LinkedList) a velikost vždy přizpůsobovat aktuálnímu počtu figur. Nakonec se však použila reprezentace pomocí obyčejného pole, jelikož v nejhorším případě je zbytečný nevyužitý prostor zanedbatelný, navíc režie na obsluhu složitějších datových struktur by byly mnohem vyšší než u běžného pole.

Na začátku partie jsou tedy obě pole zaplněna indexy počátečních pozic figur. Provádí-li se tah nějakou figurou, musí se rovněž aktualizovat obsah pole umístění figur. Sekvenčně se vyhledá příslušný prvek a na jeho místo se vloží nová pozice. Pokud je figura sebrána, uloží se na její místo hodnota představující prázdné pole.

Drobné optimalizace bylo dosaženo zavedením pravidla, že každému typu figury přísluší určitý rozsah indexů v poli. Problém ovšem nastává ve chvíli, kdy se pěšec promění v jinou figuru a tedy může vzniknout např. situace, kdy jsou ve hře tři jezdci jedné barvy. V takovém případě by se nový jezdec nenašel na příslušném rozpětí indexů a zřejmě tedy šlo o proměnu pěšce. Hledání se tedy zavolá ještě jednou, tentokrát však v rozmezí indexů jemuž přísluší pěšci. Ačkoliv se asymptotická složitost algoritmu nezmění, dalším přínosem tohoto postupu je jeho využití při rozpoznávání různých pozičních jevů v ohodnocovací funkci.



Obrázek 6.2: Struktura obsahující polohy figur na šachovnici. V tomto případě se jedná o černé figury uživatele v základním postavení

6.2.3 Vlastnosti a nastavení

V závislosti na uživatelském nastavení se ve třídě **GameProperties** uchovávají hodnoty parametrů šachové partie. Jedná se zejména o barvu figur uživatele, jestli se používají šachové hodiny, případně nastavení času na přemýšlení jednotlivých hráčů a především instanci třídy Search každého hráče, která se používá pro výběr tahu.

6.3 Tahy

Třída OneMove

Představuje samostatný tah s parametry, jež ho jednoznačně určují. Implementuje rozhraní **Comparable**, což se využívá při statickém třídění tahů. Poskytuje

metody pro zjišťění zda se jedná o některý ze speciálních tahů. Generátor tahů vytváří spojový seznam s prvky této třídy. **OneMove** obsahuje celkem 5 atributů:

`int from` - index výchozího pole
`int dest` - index cílového pole
`int destPiece` - číselné označení figury, která původně stála na cílovém poli
`int changePawn` - pokud došlo k proměně pěšce, jaká bude nová figura
`boolean enPassant` - zda jde o braní mimochodem

Poslední atribut není nezbytně nutný. Zda došlo k proměně pěšce by se dalo zjistit i z předchozích parametrů, nicméně implementace tímto způsobem byla výrazně snazší.

Tato třída implementuje rozhraní Comparable, což lze využít jako heuristiku v prohledávacím algoritmu. Tah A je větší než tah B, pokud rozdíl hodnoty táhnoucí a sebrané figury tahu A je menší než tentýž rozdíl tahu B.

Třída MoveList

Potomek třídy ArrayList s prvky typu OneMove. Obsahující seznam všech pseudopřípustných tahů hráče na tahu. Jedná se jednak o tahy, které jsou skutečně dle pravidel, ale i takové, které vedou např. do šachu a nejsou tedy povoleny. Ty jsou však odfiltrovány prohledávacím algoritmem.

Součástí třídy MoveList je generátor tahů, který prochází prvky pole PiecesOnBoard a podle druhu figury vytváří příslušné tahy. Generátor tahů se volá v každém uzlu prohledávacího stromu, který není listem.

Jednotlivé typy figur mohou být rozděleny do dvou skupin dle dosahu jejich tahu. Do skupiny s krátkým dosahem patří pěšec, jezdec a král, zbývající figury patří mezi dalekodosahové. Generátor tahů využívá metod `shortDistanceMove` a `longDistanceMove`, které kromě aktuální situace na šachovnici dostávají na vstupu ještě výchozí pole a směr, ve kterém by se měly hledat přípustné tahy.

Třída NotationList

Potomek třídy ArrayList s prvky typu String. Každý prvek obsahuje úplnou algebraickou notaci a celá struktura NotationList uchovává posloupnost dosud odehraných tahů obou hráčů. Pro prosté hraní šachů není existence této třídy nezbytná, nicméně v programu Chessplayer se využívá ze dvou důvodů. Pro uložení posloupnosti tahů do souboru se jednoduše obsah struktury vypíše do souboru. Jestliže chceme tahy vracet, používáme opět data této třídy. Podle řetězce s notací vytvoříme instanci třídy OneMove a zahrajeme tento tah pozpátku.

6.4 Prohledávání stavového prostoru

6.4.1 Třída Search

Metody třídy **Search** zajišťují výběr tahu soupeře. Přesný algoritmus a jeho parametry se volí na základě nastavení uživatele a využívá metod popsaných v kapitole 3.

V případě uživatelem vybrané obtížnosti **Adapt to opponent** se postupuje odlišným způsobem, než u běžného výběru nejlépe se jevícího tahu. Partie má vyrovnaný průběh, pokud se hodnoty vznikajících pozic co nejvíce přibližují 0. Usilujeme tedy o to, aby byl vybrán tah vedoucí do pozice s malou absolutní hodnotou ohodnocení. Prohledávací algoritmus dostává mj. jako vstup seznam přípustných tahů, které jsou obaleny instancemi třídy **MoveInfo**, díky níž můžeme ke každému tahu přiřadit jeho ohodnocení. V každém cyklu iterativního prohlubování se aktualizují hodnoty tahů a po skončení prohledávání máme seznam tahů včetně jejich ohodnocení. Tento seznam se projde a vybere tah s nejmenší absolutní hodnotou ohodnocení. Takový postup se však chová nepřirozeně v koncovkách, proto se na konci hry neuplatňuje.

Obsahuje instanci třídy Evaluation, která zodpovídá za ohodnocení dané pozice. Třída Search počítačového soupeře a uživatele mohou mít rozdílné hodnoty atributů, avšak v případě uživatele se tyto využijí pouze při režimu Autoplayer, nebo při vyvolání nápovědy tahu.

6.4.2 Ohodnocování pozice

Třída **Evaluation** ohodnocuje konkrétní pozici. Atributy nesou informace o používaných hodnotách ohodnocovaných jevů a na základě těchto hodnot se určuje výsledné ohodnocení. Její hlavní metoda **value()** dostává na vstupu aktuální instanci třídy **Board**, seznam přípustných tahů a číslo určující hráče na tahu. Všechny tyto parametry jsou nezbytné k ohodnocení příslušného postavení.

6.5 Interakce s uživatelem

Třída MouseEventHandler

Využíváme principů událostmi řízeného programování. Třída MouseEventHandler definuje rutiny, které se provádějí jako reakce na různé uživatelské akce. Jestliže se hráč pokusí zahrát tah, zkontroluje se jeho přípustnost a zavolají se metody ostatních datových struktur, které jsou tímto tahem ovlivněny. Poté se ještě provede tah graficky, kdy se překreslí hrací deska, přidá se další buňka v panelu notace, aktualizuje se obsah panelu sebraných figur a stavového řádku. Nakonec se předá řízení prohledávacímu algoritmu, který zajistí odpověď počítačového soupeře.

Po odehraní každého tahu je potřeba zkontrolovat, jestli nenastal konec partie. Jestliže se tak stalo, formou dialogového okna oznámíme uživateli, jakým způsobem partie skončila a umožníme mu zvolit, zda chce uložit posloupnost tahů. Partie může skončit remízou (pat, opakování tahu, mrtvá pozice), nebo vítězstvím jednoho z hráčů (mat, vyprší čas soupeři). Z pohledu implementace se mat od patu liší pouze tím, že v prvním případě je král napaden. Abychom se vyhnuli generování všech tahů a testování, jestli některý nekončí na políčku se soupeřovým králem, zavedli jsme funkci isUnderAttack, která prozkoumá všechny směry, ze kterých by mohl být král napaden a kontroluje, jestli se v některém nenachází napadající figura. Pro zjištění, zda existuje nějaký přípustný tah, stačí najít jen jeden.

Šachové hodiny

Pokud se využívají šachové hodiny, ubíhání času se zobrazuje během přemýšlení hráče, ale i v průběhu práce prohledávacího algoritmu. Aby bylo dosaženo požadovaného efektu, využilo se více vláken. Třída **Second** je odvozena od třídy **Thread** a každou sekundu způsobí snížení časového limitu a aktualizaci zobrazeného času.

Závěr

Vytváření her dvou hráčů jako jsou šachy, tak, aby obstály proti zkušeným hráčům, je poměrně rozsáhlý úkol a existuje celá řada možností a směrů, kterými se lze ubírat.

Tato práce popisuje obecné principy tvorby šachových programů, rozebírá tradiční postupy v algoritmech umělé inteligence uváděné zejména v minulých desetiletích.

Vedle teoretické části vznikla aplikace Chessplayer umožňující hrát šachy proti počítači na různých operačních systémech. Kromě běžné partie obsahuje méně typické funkce jako záznam algebraické notace průběhu partie, možnost ukládání a načítání rozehrané hry, vynucení okamžitého tahu, vracení tahů a řadu volitelných nastavení, která ponechávají uživateli větší kontrolu nad činností algoritmu soupeřových figur.

Program Chessplayer se ve smyslu dosažené umělé inteligence nemůže srovnávat se známými šachovými programy, to však nebylo prvotním záměrem. Přínos spočívá zejména v přiblížení principů fungování algoritmů prohledávání herních stromů, možnosti modifikovat jejich parametry v rámci aplikace a navzájem je porovnávat.

Výsledek práce lze shrnout do tří hlavních aspektů, kterých bylo dosaženo:

- Teoretický souhrn hlavních oblastí počítačového šachu
- Vytvoření uživatelsky přívětivé aplikace
- Srovnání a demonstrace chování známých algoritmů používaných v šachových programech

Stojí za zvážení volba datových struktur. Jisté optimalizace by zřejmě bylo možno dosáhnout, pokud by se namísto použitého objektového programování zvolil spíše přístup na nižší úrovni.

Potenciální rozšíření práce by mohlo spočívat v zavedení dalších méně známých heuristik včetně jejich parametrizování a srovnávání. Široké možnosti nabízí různé přístupy k ohodnocování pozice a jejich kombinace.

Literatura

[1] G. M. Adelson-Velskii a kol. Programming a computer to play chess. *Russian Mathematical Surveys*, 1970.

[2] S.G. Akl and M.M. Newborn. The principle continuation and the killer heuristic. In *ACM Annual Conference*, pages 466–473, 1977.

[3] Kenneth J. chisholm and Peter V. G. Bradbeer. Machine learning using a genetic algorithm to optimise a draughts program board evaluation funtion. In *Proceedings of IEEE International Conference on Evolutionary Computation, ICEC 97*, 1997.

[4] Omid David-tabibi and Nathan S. Netanyahu. Extended null-move reductions. In *In Proceedings of the 2008 International Conference on Computers and*, pages 205–216. Springer, 2008.

[5] Omid David-tabibi and Nathan S. Netanyahu. Genetic algorithms for mentor-assisted evaluation function optimization, 2008.

[6] CSc. doc. RNDr. Helena Brožová. Rozhodovací modely a znalostní management, 2007.

[7] James Eade. *Chess Openings For Dummies*. Wiley Publishing, Inc., August 2010.

[8] FIDE. *Laws of Chess*, November 2008.

[9] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artif. Intell.*, 6(4):293–326, 1975.

[10] Judea Pearl. SCOUT: A simple search game-searching algorithm with proven optimal properties. In Robert Balzer, editor, *1st Annual National Conference on Artificial Intelligence*, pages 143–145. American Association for Artificial Intelligence, AAAI Press/MIT Press, August 1980.

[11] Alexander Reinefeld. An Improvement to the Scout Tree-Search Algorithm. *International Computer Chess Association Journal*, 6(4):4–14, December 1983.

[12] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989.

[13] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved.

[14] Claude Shannon. Programming a computer for playing chess. Technical report, Bell Telephone Laboratories, Inc., Murray Hill, N.J., March 1950.

[15] Omid David Tabibi and Nathan S. Netanyahu. Verified null-move pruning, 2002.

[16] Jiří Lažanský a kol. Vladimír Mařík, Olga Štěpánková. *Umělá inteligence I*, volume 1. Academia, 1993.

[17] Albert L. Zobrist. A new hashing method with application for game playing. Technical report, The University of Wisconsin, April 1970.

Obsah přiloženého média

Na přiloženém CD najdete:

- **src:** Zdrojové kódy programu Chessplayer
- **dist:** Spustitelný soubor Chessplayer.jar
- **savegames:** Několik ukázkových uložených her
- **text:** Tento text ve formátu PDF
- **readme:** Poznámky a kontakt