Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFOMA NÝCH TECHNOLÓGIÍ

Odbor: Softvérové inžinierstvo

Distribuované riešenie symetrickej hry

Tímový projekt

Tím íslo 13: **Švábi**

lenovia tímu: Bc. Matúš Svr ek,
Bc. Alexander Šimko,
Bc. Michal Šteklá ,
Bc. Miroslav Štolc,
Bc. Jaroslav Tešlár,
Bc. ubomír Varga

Vedúci tímu: Ing. Peter Lacko

Školský rok: 2007/08

Zadanie

Od vzniku po íta ovej vedy sa udia snažili vytvori programy, ktoré by hrali hry lepšie ako udia. V niektorých hrách sa podarilo vytvori programy hrajúce na majstrovskej úrovni, ale niektoré hry zostávajú pre po íta e stále neriešite né.

Cie om projektu bude vytvori s pomocou technológie BOINC (Berkeley Open Infrastructure for Network Computing) systém, ktorý bude schopný zisti aspo ultra slabé (prípadne slabé) riešenie skúmanej symetrickej hry. Ultra slabé riešenie nám dáva odpove na otázku, i prvý hrá vyhrá, prehrá alebo remizuje, ak bude aha najlepšie ako sa dá.

Výsledný produkt bude obsahova dve asti (server a klient). Server projektu, ktorý bude rozde ova úlohy pre klientov a zbiera ich výsledky. Server taktiež poskytuje globálne štatisti ky projektu a individuálne štatistiky pre používate ov, riešiacich daný problém. Klientska aplikácia, ktorá vykonáva požadované výpo ty na po íta i používate a, ktorý sa rozhodol darova svoju výpo tovú silu projektu.

Funk nos systému overte na hre reversi 8x8 aGO 7x7.

I Dokumentácia Projektu

Obsah

1 Úvod....................I-5
2 Špecifikácia zadania....................I-6
  2.1 Server....................I-6
  2.2 Klient....................I-6
3 BOINC....................I-7
  3.1 Popis priebehu komunikácie serveru a klienta....................I-8
  3.2 Démoni bežiaci na serveri....................I-9
  3.3 Možnosti komunikácie klienta a serveru....................I-10
4 Hry....................I-11
  4.1 Symetrické hry....................I-11
  4.2 Reversi....................I-11
  4.3 GO....................I-13
5 Teoretický základ pre riešenie hier....................I-17
  5.1 Aké riešenie h adáme....................I-17
  5.2 MiniMax....................I-17
  5.3 Ve kos stromu h adania....................I-19
  5.4 Alfa beta usekávanie....................I-20
  5.5 NegaMax....................I-22
  5.6 Alfa beta usekávanie s NegaMax rozšírením....................I-23
  5.7 NegaScout....................I-24
  5.8 MTD(f)....................I-26
  5.9 Heuristiky....................I-26
    5.9.1 Heuristika vražedného ahu....................I-26
    5.9.2 Heuristika založená na histórii ahov....................I-27
    5.9.3 Problémovo závislé ohodnotenie vhodnosti pozície....................I-27
  5.10 Transpozi ná tabu ka....................I-30
  5.11 Symetrické pozície....................I-30
  5.12 Serverové a klientske preh adávanie....................I-30
6 Existujúce riešenia....................I-32
  6.1 Reversi....................I-32
  6.2 GO....................I-33
7 Možnosti ukladania stromu na disk....................I-34
  7.1 Reprezentácia stavu hry....................I-34
  7.2 Ve kos uložených dát....................I-34
  7.3 Reprezentácia stromu....................I-34
  7.4 Konkrétne možnosti ukladania stromu na disk....................I-35
    7.4.1 Existujúci systém súborov....................I-35
    7.4.2 Vlastný FS....................I-36
    7.4.3 Databáza....................I-36
  7.5 Záver k ukladaniu dát....................I-37
8 Návrh systému....................I-38
  8.1 Návrh paralelného systému....................I-38
  8.2 Návrh spolo ných astí....................I-39
    8.2.1 Komponent hra....................I-39
    8.2.2 Komponent Uzol....................I-40
  8.3 Návrh klienta....................I-41
  8.4 Návrh server....................I-44
9 Prototyp....................I-47
  9.1 Prototyp hry na jeden PC....................I-47

9.1.1 Celý strom do istej h bky........I-47
9.1.2 Odhad faktoru vetvenia........I-48
9.2 Prototyp aplikácie pre BOINC........I-48
9.3 Prototyp BOINC projektu........I-49
9.3.1 Návrh databázy........I-49
9.3.2 Klient........I-51
9.3.3 Generátor úloh........I-52
9.3.4 Asimilátor........I-53
10 Zmeny oproti návrhu........I-54
10.1 Spolo né asti........I-55
10.1.1 Komponenta search........I-55
10.1.2 Komponenta uzol........I-55
10.1.3 Komponenta hra........I-56
10.2 Klient........I-57
10.2.1 Heuristická komponeta........I-58
10.3 Server........I-59
10.3.1 Vstupno-výstupná komponenta........I-59
10.3.2 Databázová komponenta........I-60
11 Implementácia........I-61
11.1 Ktorý hrá je ktorý........I-61
11.2 Použitá heuristika........I-61
11.3 Implementované algoritmy na klientovi........I-61
11.4 Ukladanie stavu výpo tu na klientovi........I-62
11.5 Databáza........I-63
11.5.1 Databáza pre hru reversi........I-64
11.5.2 Identifikácia uzla........I-64
11.5.3 Prepojenie uzlov........I-65
11.5.4 Ohodnotenie uzlov........I-66
11.5.5 Ohodnocovanie stromu........I-66
11.5.6 Dátová reprezentácia v databáze........I-66
11.5.7 Implementácia databázového modulu........I-67
11.5.8 Kompilácia databázového modulu........I-68
11.6 Implementácia asimilátora........I-69
12 Testovanie........I-70
12.1 Testovanie algoritmov a výkonnosti heuristík........I-70
12.2 Testovanie klienta........I-72
12.3 Testovanie asimilátora........I-73
12.4 Testovania generátora........I-73
13 Výsledky........I-75
14 Zhodnotenie........I-76
15 Použitá literatúra........I-77
Príloha A – Slovník pojmov........A-1
Príloha B – Použitá notácia diagramov........B-1
Príloha C – Používate ská príru ka........C-1
Príloha D – Technická príru ka........D-1

1 Úvod

udia sa venujú hraniu hier oddávna. Ich motívy sú rôzne. i už je to hranie prezábavu, pokorenie kamaráta alebo vidina miliónových ziskov v Monte Carle, hry zaujímajú udskú pozornos stále.

Deterministické stolové hry predstavujú samostatnú kapitolu. lovek sa nemusí namáha , nespotí sa, ani si nezlomí nohu. Môže sedie v teple svojej izby, húta , namáha svoje mozgové závity a ozlomkrky aha figúrkami alebo kame mi pohracej ploche. Sta í mu iba jeho intelekt.

Niektorí sa pri ich hraní uspokoja s tým, že pokoria svojho suseda, vyhrajú miestny, i svetový šampionát. Sú však udia, ktorí pri hraní hier zachádzajú alej a pýtajú sa: “Existuje spôsob ako vyhra stále?“

Koho by táto otázka nelákala? Málo z nás je nato ko nadaných, aby vedeli poskytnú matematický dôkaz, alebo má k dispozícií nieko ko miliónov rokov na preskúmanie všetkých možností hry.

My sme túto výzvu vrámci predmetu Tvorba softvérového systému v tíme prijali a v tomto dokumente opisujeme pokus o h adanie odpovede na stanovenú otázku. Nerobíme to pre jedinú konkrétnu hru, no venujeme sa všeobecnému systému na nájdenie ultra slabého riešenia pomocou distribuovaného systému BOINC.

T tomto, prvom, dieli dokumentu sa venuje riešeniu projektu.

V kapitole dva uvádzame špecifikáciu zadania a požadované vlastnosti riešenia. Kapitola tri pojednáva o distribuovanom systéme BOINC, jeho architektúre, astiach a fungovaní. Vkapitole štyri popisujeme hry reversi a GO, ich pravidlá a základné stratégie hry. Kapitola pä poskytuje nevyhnutné teoretické záležitosti oh adom riešenia hier, rozoberá vybrané algoritmy, heuristiky a vylepšujúce techniky. Existujúce riešenia hier reversi a GO o menších rozmeroch rozoberá kapitola šes . V kapitole sedem analyzujeme rôzne možnosti ukladania stromu preh adávania. Kapitola osem sa venuje hrubému návrhu systému a v kapitole devä popisujeme vytvorené prototypy. Zmenami, ktoré sme museli spravi oproti návrhu zo zimného semestra sa zaoberá kapitola desa . V jedenástej kapitole je popísaná samotná implementácia nášho projektu. Testovaniu rôznych astí produktu sa venuje kapitola dvanás . Stru né zhrnutie nami dosiahnutých výsledkov je v trinástej kapitole. Príloha A obsahuje slovník pojmov a príloha B notáciu použitú v kapitole osem. Príloha C obsahuje používate skú príru ku. Technická príru ka sa nachádza v prílohe D.

2 Špecifikácia zadania

Cie om projektu je za pomoci technológie BOINC vytvori systém, ktorý bude schopný dosta ako výstup ultra slabé riešenie symetrickej hry. My budeme rieši symetrické hry reversi 8x8 a GO 7x7. Snahou je vytvori taký systém, aby bol o najviac nezávislý na hre. To znamená, aby bolo možné v om rieši aj iné symetrické hry, ako je napr. šach a pod.

Systém bude obsahova dve hlavné asti:

1) server
2) klient

Aby bolo možné pracova na tomto projekte aj v nasledujúcich rokoch, je potrebné, aby vytvorené zdrojové kódy boli preh adné, modulárne a pochopite né.

alšou dôležitou astou projektu je dokumentácia. Je ju potrebné vytvára zodpovedne a nemôže by zanedbaná.

2.1 Server

V systéme bude jeden server. Jeho úlohami sú:

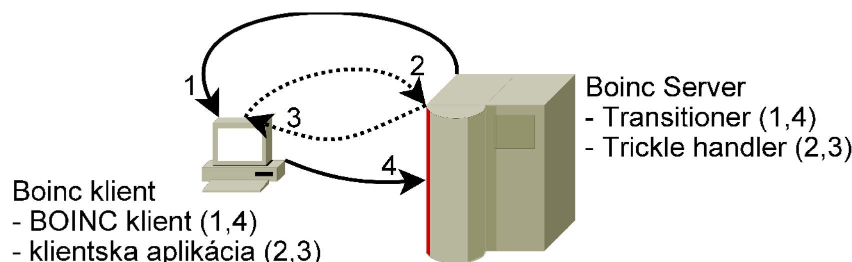
1) rozdelenie hlavnej úlohy na iastkové podúlohy
2) rozdelenie iastkových podúloh medzi klientov
3) zabezpe enie komunikácie medzi serverom a klientmi
4) zber iastkových úloh od klientov
5) sumarizácia iastkových podúloh
6) spo ítanie celkového riešenia
7) vytvorenie štatistík

2.2 Klient

Jeho úlohou bude rieši iastkové podúlohy, ktoré dostal zadané od servera. Klientov bude v našom systéme o najviac – n. ím viac klientov bude pracova na riešení úlohy, tým bude riešenie rýchlejšie dosiahnuté.

3 BOINC

BOINC[1] (Berkeley Open Infrastructure for Network Computing) je open-source-ový softvérový projekt, ktorý umož uje využíva vo ný výkon dobrovo níkov na asovo náro né výpo ty. Jeho architektúra je typu klient – server. Na BOINC softvéri je postavených nieko ko desiatok výpo tovo náro ných, vedeckých i menej vedeckých projektov. Jedným z najznámejších je SETI (Search for Extraterrestrial Intelligence), h adanie mimozemskej inteligencie. Na Zemi je rozmiestnených nieko ko satelitov, jeden z nich však zachytáva signály prichádzajúce z kozmu a snaží sa v nich h ada po in inteligencie. Zachytených signálov je obrovské množstvo a je ich snaha analyzova všetky. Pre analýzu takého množstva dát by však bolo treba superpo íta , o je drahá hra ka, a tak vedci vyvinuli SETI@home. Vznikla možnos pomôc s výpo tami doslova zo svojho domu. Stiahnutím malej aplikácie, ktorá z asu na as stiahla dáta z laboratórií, spracovala ich a výsledok vrátila naspä . Tento spôsob pomoci pri h adaní mimozemskej inteligencie si našiel obrovské množstvo prívržencov a dobrovo níkov, ktorí pomáhali. Tak vznikol základ pre platformu umož ujúcu zapriahnu množstvo domácich po íta ov do ve kých výpo tov. Vznikol BOINC. Koncom roku 2005 bola vypustená posledná úloha pre klasické SETI klientske programy a aj samotný SETI odvtedy beží tiež na BOINC platforme.



Obr. 1: Komunika né možnosti BOINC projektu s BOINC aplikáciou na klientskej strane. 1. poslanie úlohy od serveru klientovi po vyžiadaní práce, 2. asynchrónna komunikácia od klienta k serveru, 3. asynchrónne oznámenie udalosti zo serveru klientovi, 4. poslanie výsledku na server

BOINC server poskytuje klientske aplikácie, ktorých úlohou je uskuto ova náro né výpo ty. Aplikácie však pre svoju innos potrebujú ešte úlohu (angl. workunit). Je to súbor obsahujúci popis úlohy a dáta k nej. Klientska aplikácia berie ako vstup danú úlohu, transformuje ju na výstup, ktorým je výsledkom výpo tu (angl. result) a je uložený ako súbor. Tento výsledok následne pošle BOINC klient na projektový server.

Dobrovo ník pre zapojenie sa do pomoci vo výpo toch nainštaluje BOINC klient aplikáciu a následne si v BOINC klientovi pridá projekty, na ktorých sa chce podie a . Každý projekt má svoju stránku, na ktorej sa každý dobrovo ník môže pozrie na svoje výsledky, ko ko kreditov za ne získal a môže meni svoj podiel v prispievaní danému projektu. Je teda možné podie a sa na h adaní proteínu proti malárii, HIV, ale zárove je možné vužívate om zadefinovanom pomere pomáha v rendrovaní po íta ových animácií, poprípade pomáha v h adaní gravita ných v n.

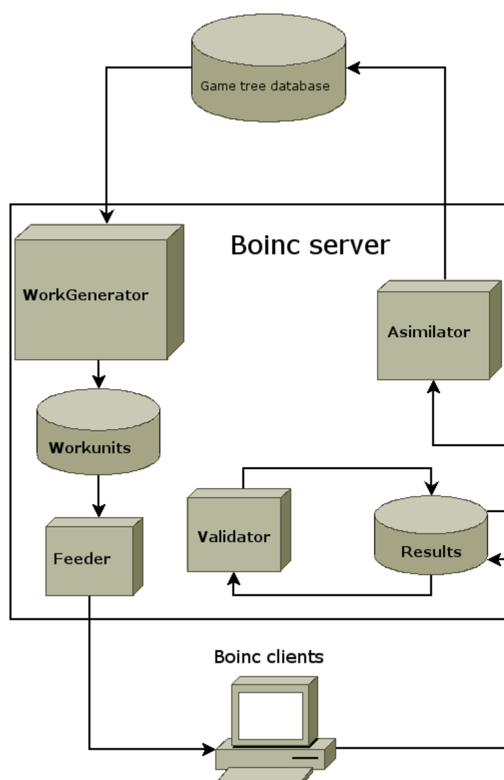
1 http://boinc.berkeley.edu/

Projektov pre BOINC platformu je aktuálne asi 40 a zaoberajú sa rôznymi oblas ami. Nájdu sa napríklad i problémy ako h adanie minimálnej konfigurácie sudoku hry, z ktorej je možné získa iba jedno správne vyplnenie hracej plochy. Okrem herných, fyzikálnych, chemických a lekárskych projektov sa nájdu i po íta ovo orientované.

3.1 Popis priebehu komunikácie serveru a klienta

Štandardný priebeh výpo tu sa za ína vyžiadaním práce klientom od serveru. Server následne pošle klientovi nejakú úlohu, a ak je potrebné, tak i klientsku aplikáciu, ktorá vie danú úlohu spracova . Klient po prijatí všetkých potrebných dát spustí klientsku aplikáciu a odovzdá jej na vstup prijatú úlohu. Aplikácia vykoná potrebné výpo ty, ako výstup pripraví výsledok a ukon í svoj beh. Klientsky softvér následne oznámi projektu, že daná úloha je spracovaná, a pošle výsledný výsledok serveru.

Nasledujúce podkapitoly budú jednotlivo popisova asti systému, ktoré sú na Obr. 2. Pre vytvorenie nového BOINC projektu nie je nutné implementova všetky asti systému. Niektoré asti je možné v niektorých prípadoch použi štandardné. Bližšie tieto možnosti opíšem v príslušných podkapitolách.



Obr. 2: Podrobný popis na architektúru BOINC serveru a jeho asti

3.2 Démoni bežiaci na serveri

Generátor úloh

Úloha generátoru úloh (angl. WorkGenerator) je jednoduchá. Na požiadanie vygenerova daný po et úloh, uloži ich nadisk do adresára pre odoslanie klientom a každú takto vygenerovanú a uloženú úlohu zaregistrova v projektovej databáze. V našom prípade bude generátor úloh používa pre generovanie nových úloh informácie z databázy stromu hry. Databáza stromu hry je vzh adom na BOINC projekt externé úložisko aktuálneho stavu výpo tu. Generátor úloh je závislý na klientskej aplikácii, teda pre každú klientsku aplikáciu je potrebné ma jeden takýto generátor.

Rozdelova úloh

Rozdelova úloh (angl. Feeder) je démon, ktorý je implementovaný a nie je ho potrebné nahradzova , alebo upravova . Stará sa o rozde ovanie úloh klientov, ktorí žiadajú o prácu. V našom projekte sa touto as ou BOINC softvéru nebudeme zaobera . Štandardný rozde ova úloh je možné inštruova k rôznemu poradiu odosielania úloh. Úlohy je možné posiela klientom pod a asu vytvorenia, pod a priority alebo náhodne. Pri viacerých klientskych aplikáciách, teda viacero problémov riešených v rámci jedného projektu, je možné posiela úlohy bu v náhodnom poradí, o do typu aplikácie, alebo v nastavenom pomere.

Validátor

Validátor (angl. Validator) je zodpovedný za validáciu prišlých výsledkov. Pri generovaní úloh generátorom úloh si tento démon ur uje po et klientov, ktorí majú po íta tú istú úlohu. Je to kvôli bezpe nosti. Klienti môžu z rôznych dôvodov falšova vracané výsledky, a tak sa dáva tá istá úloha prepo íta viacerým klientom a výsledky sa porovnávajú. Porovnanie má na starosti práve validát or.

Po prijatí výsledku je úlohou validátoru vytvori alebo ur i takzvaný kanonický výsledok (angl. canonical result), ktorý zastupuje všetky výsledky. Tento následne použije asimilátor. Prijaté výsledky validátor porovná a ur í kredit, ktorý sa pripíše na ú et klientom, ktorí úlohu po ítali. Kredit je možné získa dvoma spôsobmi. Z klientskych výsledkov vy íta po et operácií, ktoré zhruba procesor vykonal pri výpo te (as krát benchmark), alebo, ak generátor úloh takú informáciu k úlohe uložil, vy íta kredit z úlohy prislúchajúcej k výsledku. Ak to výpo et dovo uje, je lepšie ur ova kredit pri tvorbe úlohy a teda pri validácii ho vy íta z úlohy.

Asimilátor

Asimilátor (angl. Assimilator) je zodpovedný za spracovanie kanonického výsledku úlohy. Spracovanie z poh adu BOINC serveru znamená prenesenie informácií z výsledkov do externého úložiska. Po „asimilovaní“ výsledku úlohy je úloha i jej výsledok zmazaný. Asimilátor v našom prípade bude uklada ohodnotenia stromu do externého úložiska stromu hry.

Iné sú asti BOINC serveru

BOINC server obsahuje i tu nemenované démony, ako napríklad file deleter démon. Všetky démony, ktoré neboli menované sa používajú štandardné a v žiadnom z projektov nie sú nahradzované vlastnými verziami. Kompletný popis démonov sa nachádza na dokumenta nej stránke projektu BOINC[2].

3.3 Možnosti komunikácie klienta a serveru

Komunikácia je pri distribuovaných výpo toch ve mi dôležitá as výpo tu. Na superpo íta och sa na komunikácii stratí i nieko ko desiatok percent výkonu. Záleží od paralelizovate nosti problému, ko ko komunikácie a v akom ase je potrebnej. Pre BOINC projekty, kde komunikácia prebieha cez internet, a nemôže sa spolieha na stálu dostupnos klientov, je možné po íta iba ve mi dobre paralelizovate né úlohy. Je snaha spravi BOINC platformu vhodnú i pre „low latency" projekty, kde nie je dlhšiu dobu dostupná žiadna práca, ale ke je dostupná, musí sa spravi rýchlo a teda je delená na menšie asti, aby sa lepšie využili aktuálne pripojené po íta e.

Komunikácia serveru s klientom je možná v zásade dvoma spôsobmi. Prvým spôsobom, ktorý je v BOINC architektúre od jej zrodu, je odovzdanie úlohy klientovi po tom, o si klient vyžiada úlohu. V danej úlohe je všetko potrebné pre aplikáciu v klientovi na to, aby splnila svoju výpo tovo náro nú úlohu. Druhou, novšou, možnos ou komunikácie od serveru ku klientovi je posielanie si správ cez rad správ. Ke sa klient z asu na as kontaktuje so serverom v priebehu výpo tu, server odovzdá klientovi správy, ktoré mu adresoval niektorý z démonov. Klientovi sa dá vnúti pripájanie sa na server v daných asových intervaloch. Klient sa pripája scie om zisti , ko ko kreditov má daný užívate , a tiež je pripravený vyzdvihnú si správu ur enú pre klientsku aplikáciu. Tento spôsob komunikácie bol zavedený pre možnos kontaktovania klientskej aplikácie po as výpo tu a oznámi jej nové skuto nosti. Odzrušenia úlohy, cez odovzdanie nových poznatkov, ktoré urýchlia riešenie problému, až po oznámenie ukon enia projektu z dôvodu úspešného vyriešenia problému.

Komunikácia od klienta k serveru prebieha obdobne. Najstarším spôsobom je odoslanie hotovej úlohy, v ktorej klientska aplikácia navrhne po et kreditov za vypo ítanú as . Do výsledku okrem samotného výsledku uvedie strávený as výpo tom a podobne. Tu je pre klientsku aplikáciu otvorený priestor na informovanie serveru o omko vek. Tento systém sa pri projektoch, ktorých úloha trvala extrémne dlho (mesiace), ukázal ako nedosta ujúci, a tak obdobne, ako i na serverovej strane, sa pridala možnos asynchrónnej komunikácie klienta so serverom po as priebehu výpo tu. Klientska aplikácia môže odovzda správu pre server klientovi a ak je to potrebné, vyžiada si okamžité pripojenie sa na server a odovzdanie správy. Takto napríklad pri dlhých úlohách môže klientska aplikácia oznámi serveru, že úloha sa po íta, a že má zmysel aka na výsledok. Môžu sa i prideli pred asne body za vypo ítanú as úlohy.

---

2 http://boinc.berkeley.edu/trac/wiki/BackendPrograms.

4 Hry

4.1 Symetrické hry

Symetrická hra je hra, v ktorej majú obidvaja hrá i rovnaké – symetrické – postavenie. Obaja hrá i sú rovnocenní, môžu robi rovnaké ahy vyplývajúce z presných pravidiel hry. Odlišujú sa len v tom, že jeden z nich je na ahu ako prvý. Obaja hrá i riešia rovnaký strategický cie – maximalizujú svoj zisk, a sú asne zabra ujú v tejto akcii súperovi, t.j. minimalizujú zisk súpera. Medzi známe a ob úbené symetrické hry patrí aj reversi a go, ktoré sú predmetom riešenia tohto projektu, a preto ich podrobnejšie popíšem v nasledujúcich kapitolách.

4.2 Reversi

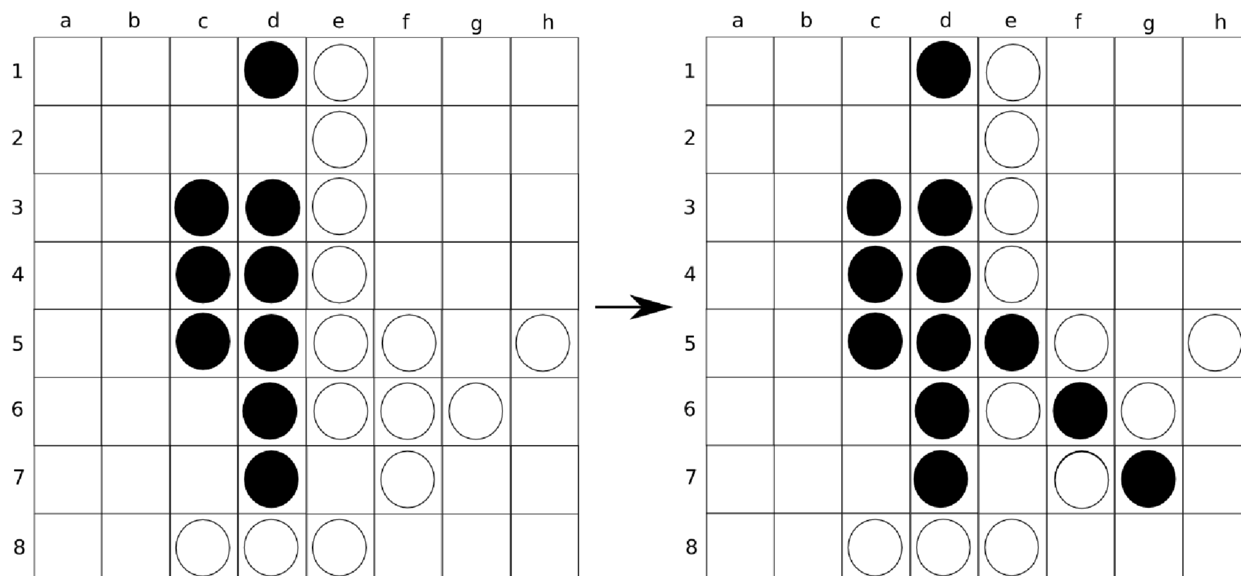
Pravidlá hry a herné stratégie zachytáva dokument Randyho Fanga [4]. Hra reversi bola pôvodne objavená Angli anmi Lewisom Watermanom a Johnom W. Molletom koncom 19. storo ia. oskoro nadobudla ve kú popularitu. Nové pravidlá známe ako othelo sa zaviedli v 70. rokoch 20. storo ia v japonskom meste Miko. Prvé majstrovstvá sveta sa uskuto nili v roku 1977. Najúspešnejšími individuálnymi hrá mi sú doteraz Japonci, z tímov Francúzi.

Reversi hrajú na hracej ploche bežne rozmerov 8x8 dvaja hrá i – tmavý a svetlý. Hracia plocha pozostáva zo st pcov, ktoré definujú z ava doprava písmená abecedy, a z riadkov, ktoré sú o íslované zhora nadol. Pôvodne reversi nemalo ur enú za iato nú pozíciu. Až pravidlá othelo definujú, že hra sa za ína so štyrmi kame mi v strede hracej plochy, dvomi kame mi tmavého hrá a advomi kame mi svetlého hrá a usporiadanými do diagonál. Tmavý hrá má kamene na pozíciách E4 a D5 a svetlý na D4 a E5 ako znázor ujeObr. 3. Za ína ierny hrá .

Obr. 3: Začiatočná pozícia pri reversi

Každý hrá musí umiestni svoj alší kame na vo nú pozíciu hracej plochy tak, aby existovala aspo jedna horizontálna, vertikálna alebo diagonálna úse ka medzi novým kame om a iným

kame om toho istého hrá a, pri om všetky medzi ahlé kamene patria súperovi. Po uskuto není takéhoto ahu sa spomínané súperove kamene stanú kame mi hrá a, ktorý takýto ah uskuto nil, t.j. zmenia farbu. V praxi to znamená, že sa kame prevráti na opa nú stranu. Ak hrá nemá možnos vykona ah pod a tohto pravidla, na rade je opä protihrá . Na Obr. 4 je zobrazený jeden ah v hre reversi, ke tmavý hrá dá kame na pozíciu g7.



Obr. 4: Ukážka ťahu v reversi

Hra sa kon í, ak už ani jeden hrá nemôže hra alej. Táto situácia môže nasta v troch rôznych prípadoch:

1. celá hracia plocha je zaplnená kame mi
2. na hracej ploche sa nachádzajú len kamene jedného hrá a a druhý hrá nemá ani jeden
3. na hracej ploche sú ešte vo né miesta, ale ani jeden z hrá ov nemá platný ah k dispozícii

Hru vyhráva ten hrá , ktorý má na konci hry viac svojich kame ov na hracej ploche.

Pre othelo zatia nebola objavená stratégia, ktorá by znamenala ví azstvo v hre. Výhoda v podobe ve kého po tu kame ov hrá a môže ahko vyústi do prehry, pretože obmedzuje možnos ahov hrá a a dáva ve kú výhodu pre protihrá a získa naraz ve ké množstvo kame ov. Existujú však elementy, ktoré vedú k úspešnej stratégii:

- *Rohy:* Ak si jeden z hrá ov obsadí rohové pozície, ostanú imúnne až do konca hry. Nie je totiž za nimi žiadna pozícia pre alší kame , a tak ani možnos pre súpera získa ich. Je výhodné obsadi ich v priebehu hry, nie na za iatku.
- *Pohyblivos :* Táto metóda sa snaží obmedzi možnos krokov protihrá a. Ak hrá robí postupne také kroky, ktoré obmedzujú legálne pohyby protihrá a, tak protihrá je skôr i neskôr donútený urobi nežiaduci ah pod a vôle toho druhého. Ideálna pozícia je napríklad, ke sú kamene hrá a sústredené v strede a obklopené kame mi protihrá a. V takom prípade hrá s kame mi v strede ur uje, aké ahy bude ma protihrá k dispozícii.
- *Hrany:* Ide o podobný princíp ako u rohov. Hrá sústre uje kamene pri hranách hracej plochy, ktoré je potom ažké získa protihrá om. Odporú a sa hra na hrany v rozbehnutej

hre, ke už hrá má výhodu v pohyblivosti alebo mnoho kame ov, ktoré mu už ur ite ostanú.

- *Parita:* Koncepcia parity patrí medzi najdôležitejšie sú asti stratégie. Hrá sa snaží o to, aby aj jeho ahy pri takmer zaplnenej hracej plochy na konci hry boli uskuto nite né, a aby sa mu tak v tejto finálnej fáze hry podarilo zvä ši po et svojich stabilných kame ov.
- *Dopredné pozeranie:* Ako platí aj v šachu, v reversi sa oplatí premýš a nie len nad aktuálnou situáciou na hracej ploche, ale aj nad možnými ahmi oponenta a vývojom situácie.
- *Koniec hry:* Ke je hra nieko ko ahov pred koncom, hrá sa zameriava na iné stratégie, ktoré mu zabezpe ia o najlepší výsledok hry.

Pod a práce Allisa [1] je stavový priestor hry obrovský, rádovo asi 10^{30}. Ke že hra sa v priemere kon í pri 58. ahu apriemerný po et možných ahov na jeden stav hracej plochy je 10, kompletný strom hry by obsahoval 10^{58} uzlov. Aj to je dôvod, pre o problém ví aznej stratégie pre hru reversi nebol doposia matematicky vyriešený. Experti sa však domnievajú, že pri štandardnej hre na hracej ploche o rozmeroch 8x8 by pri perfektných ahoch oboch hrá ov vyústil výsledok do remízy. Pri rozmeroch 4x4 a 6x6 bolo dokázané, že vyhráva druhý (svetlý) hrá .

Programy na hru reversi hrajú na úrovni svetových šampiónov od roku 1980. V tomto roku program The Moor vyhral hru proti aktuálnemu majstrovi sveta. Odvtedy sa programy neustále zlepšovali. Všetky silné programy sú založené na štandardných technikách, teda hlbokom alfa-beta preh adávaní, ve kej databáze, heuristike a ohodnocovacej funkcii.

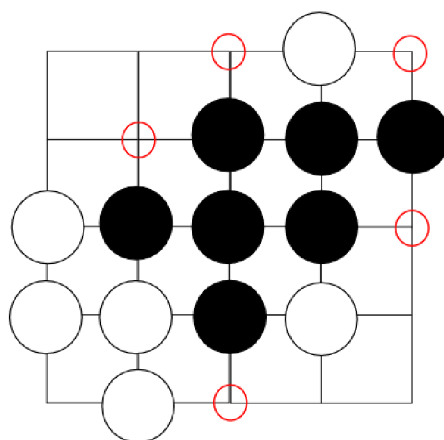
4.3 GO

Go je symetrická dosková hra pre dvoch hrá ov, o ktorej prvá písomná zmienka pochádza už z 5. storo ia pred n.l. z íny. Dnes má popularitu už na celom svete. Hoci pravidlá go sú ve mi jednoduché, stratégia hry je extrémne komplexná. Go je deterministická strategická hra ako šach, dáma alebo reversi, avšak jej zložitos prekonáva všetky tieto hry.

Go sa hrá s iernymi a bielymi kame mi na doske s mriežkou s rozmermi 19x19. Kamene sú ukladané na priese níky iar mriežky, ktorých je v tomto prípade 361. Samozrejme Go môže by prispôsobené aj na rôzne alšie ve kosti, populárne sú ešte rozmery 13x13 a 9x9. Cie om hry je kontrolova vä šiu as hracej plochy ako súper.

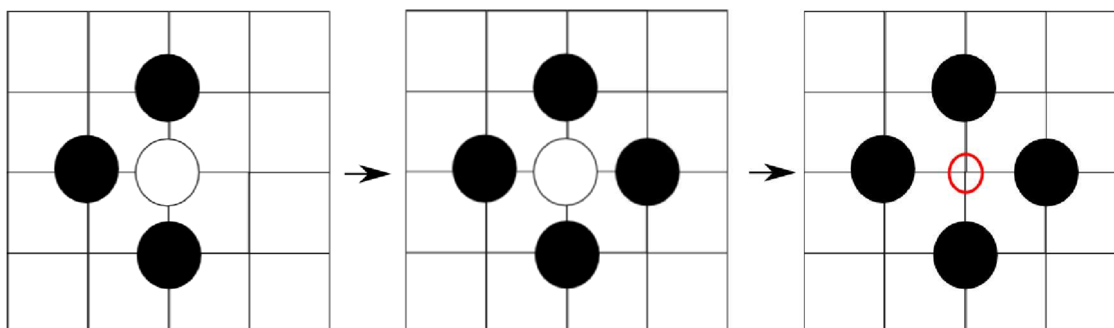
Základné pravidlá hry go uvedené v zdroji [13] zah ajú:

- Biely a ierny hrá sa striedajú v ukladaní kame ov na body mriežky. ierny hrá za ína. Kame om, ktorý už je uložený na mriežke, sa nesmie hýba .
- Každý vo ný priese ník susediaci s kame om sa nazýva *sloboda*. Kame môže ma maximálne štyri slobody.
- Kamene rovnakej farby, ktoré sú navzájom na mriežke spojené, tvoria *skupinu*. Celá skupina kame ov má vlastné slobody, nemôže by rozdelená a stáva sa tak ve kým kame om na mriežke. Skupinu tvoria len kamene, ktoré neodde uje prázdny priese ník. Ukážková skupina kame ov a jej slobody sa nachádza naObr. 5



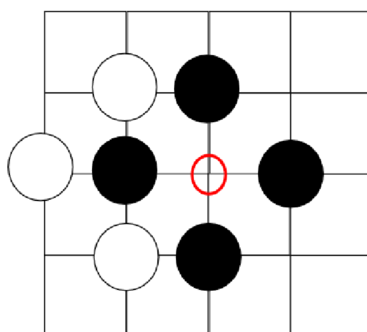
Obr. 5: Skupina čiernych kameňov a jej slobody

- Skupina kame ov môže by   rozšírená prikladaním kame ov rovnakej farby na jej slobody, ideálnym rozšírením je uloženie kame a na spolo nú slobodu dvoch skupín tej istej farby.
- Ak hrá   postupne obsadí svojimi kame mi všetky slobody kame a alebo skupinu kame ov nepriate a, druhý kame   resp. skupina kame ov je *zajatá*. Po strate poslednej slobody musí by   kame   i celá skupina okamžite odstránená z dosky. Za ínajúci hrá   si zhromaž uje súperove kamene pre neskoršie po ítanie skóre. Obr. 6 ukazuje situáciu, kedy je zajatý a z plochy odstránený jeden biely kame .



Obr. 6.:Zajatie bieleho kameňa

- Hrá   nesmie zahra   taký   ah, aby jeho kame   nemal po   ahu žiadnu slobodu. Do tejto tzv. *samovraždy* môže hrá   uloži   kame   len vtedy, ak týmto   ahom zajme súperov kame   i celú skupinu. Jeho kame   tak získa slobody a nestojí po   ahu v samovražde.
- Pod a pravidla *ko* (v preklade nekone ný) hrá   alej nesmie zahra   taký   ah, aby sa po jeho   ahu presne zopakovala pozícia pred posledným   ahom súpera. Toto pravidlo sa týka situácií, kde by inak bolo možné bra   jeden kame   stále dookola, a tak znemožni   koniec hry. Ukážka takejto situácie je na Obr. 7.



Obr. 7: Pravidlo Ko

- Postupom hry sa na doske vytvárajú oblasti, kde sa sústredia kamene jednej farby. Vo né priese níky vo vnútri hranice tvorenej kame mi jednej farby sa nazývajú *územie*. V priebehu hry sa hrá i snažia obhajova svoje územia a sú asne ni i útokmi územia súpera.
- Hrá môže vynecha ah, ak si myslí, že to pomôže zvä šeniu jeho teritória, a zmenšeniu teritória súpera. Ak za sebou vynechajú ah obaja hrá i, hra kon í. Každý hrá si spo íta body vlastných území na doske a body za zajatcov. Ví azí hrá s vä ším bodovým sú tom.

Hranie go je ve mi ažká úloha. Hrá musí ma za sebou tisícky hier, aby získal ur ité zru nosti. Postupne sa zdokona uje v chápaní, ako spája kamene do skupín, aby tak mali vä šiu moc. Dobré je pochopi úvodné sekvencie hry (Fuseki), ktoré sú ve mi dôležité, aby sa hra v strednej fáze rozbehla. Na štandardnej doske je otváranie hry v rohoch viac efektívne ako vytváranie území poblíž stredu. alej je potrebné pochopi základné koncepty a princípy využívané v komplexných stratégiách hry, ako to vysvet uje Baker [2]:

- *Spájanie a rozde ovanie:* Kamene je potrebné drža pohromade. Spájaním kame ov a vytváraním skupín dochádza k rozšíreniu slobôd. Aby potom oponent zajal celú skupinu, musí obsadi všetky jej slobody. Skupiny kame ov sú teda silnejšie, pretože si navzájom zdie ajú slobody. Na druhej strane, ke že vytváranie skupín robí skupinu silnejšiu, dôležitá ofenzívna taktika tiež spo íva v tom, že hrá sa snaží zabráni súperovi vo vytváraní takýchto skupín tým, že ich rozde uje na menšie skupinky. Hovoríme, že ich odstriháva.
- *Život a smr :* K ú ovým konceptom taktiky v go je klasifikácia skupín kame ov naživé, m tve a neobývané. Na konci hry skupiny kame ov, ktoré nemôžu uniknú zajatiu po as hry, sú odstránené z dosky ako zajaté. Tieto kamene sú m tve. Ich pochopenie spo íva v tom, že hrá sa namiesto toho, aby sa silou mocou snažil zaja ich po as hry, snaží venova iným astiam dosky, ke že na konci hry sú tak i tak zajaté. Skupina kame ov, ktorá nemôže by zajatá, sa nazýva živá. Dôvodom toho je napríklad blízkos inej skupiny tej istej farby, ku ktorej by sa v prípade útoku súpera na skupinu, dalo priblíži , spoji a zabráni tak úmyslu súpera. Kamene, ktoré nie sú ani živé ani m tve, sa nazývajú neobývané.
- Bitky o ko: Existencia bitiek ko súvisí s jedným zo spomenutých pravidiel hry go – ko, ktoré zabra uje možnosti okamžitého opakovania rovnakej pozície v hre, pri ktorej jeden kame je zajatý a následne je zajatý kame , ktorý ho zajal. Pravidlo hovorí, že okamžité znovuzajatie je zakázané, ale len na jeden ah. Toto umož uje nasledujúcu procedúru: hrá so zákazom zahrá taký ah, ktorý vyžaduje okamžitú odpove . Medzitým tak pre neho skon i zákaz ko a následne môže znovu zaja súperov kame . Tomuto odpútaniu pozornosti

sa hovorí hrozba ko. Situácie v hre, pri ktorých si hrá i vymie ajú ko hrozby za ú elom získania požadovanej pozície na hracej ploche sa nazývajú bitky o ko. V hre môžu nasta situácie s ko, ktoré rozhodujú partiu, pretože rozhodujú o územiach hrá ov. Preto bitky oko v strategických miestach sú asto k ú ové, a vždy je potrebné zváži , ko ko má ktorý hrá hrozieb v rukáve.

- Yose: Na konci partie sú stále v go možnosti, ako získa dôležité a rozhodujúce body. Yose znamená špeciálny spôsob, ako hra go na konci hry. Podstatné územia sú už síce na konci hry odstavené, ale napriek tomu sú stále menšie územia, o ktoré sa bojuje. Hrá musí dobre zhodnoti a vypo íta , o ko ko bodov sa môže v jednotlivých malých územiach hra , a o sa mu teda viac oplatí.

Expertmi je hra go považovaná za kombina ne najzložitejšiu hru vôbec, o sa prejavuje najmä v zložitosti naprogramovaní hry a obrovskému množstvu variantov ako hra . Zatia o v šachu už po íta dvakrát zdolal aktuálneho majstra sveta, najlepšie programy hrajúce go dokážu zdola aj talentované deti. Typické hry go na hracej ploche s rozmermi 19x19 trvajú pod a Allisa [1] 150 ahov a priemerne 250 možnými pohybmi v jednom ahu. Kompletný strom hry go obsahuje asi 10^{360} vrcholov.

Kvôli zložitosti a ob úbenosti sa hrou go zaoberá celá as umelej inteligencie. Prvý program na hranie go pochádza ešte z roku 1968. Sú asné programy už dosahujú dos výrazné výsledky na malej ploche 9x9, avšak nie je jasné, ako úspešné techniky dosta aj na štandardnú hraciu plochu. Priemerný hrá tak nemá problém zdola po íta . Silní hrá i dokonca dokážu vyhra aj s hendikepom 25, ba až 30 kame ov. Najob úbenejšie programy na go využívajú kombináciu výhod stromového preh adávania priestoru hry, metód Monte-Carlo, bázy znalostí a strojového u enia, najmä rozpoznávania vzorov.

5 Teoretický základ pre riešenie hier

V tejto kapitole sa venujeme teoretickým základom pre riešenie hier, popisujeme základné a vylepšené algoritmy na riešenie hier, heuristiky a alšie vylepšenia.

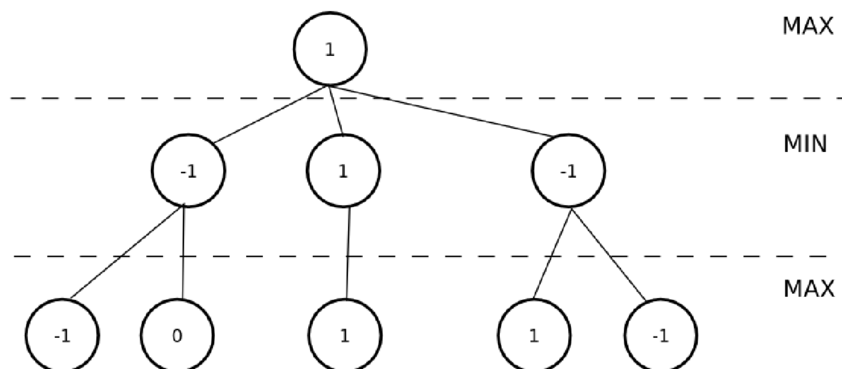
5.1 Aké riešenie h adáme

Pod a Allisa [1] je *ve mi slabým riešením* (angl. ultra-weak solution) ur enie, i hrá , ktorý za ína ako prvý, vyhrá, prehrá alebo sa hra skon í remízou, ak obaja hrá i hrajú perfektne. Za perfektnú hru hrá a sa považuje taká hra, ktorá vedie k o najlepšiemu možnému výsledku bez oh adu na to ako ahá protihrá . Vo ve mi slabom riešení sta í stanovi výsledok, nie je potrebné pozna ako je potrebné hra .

V našom prípade na ur enie výsledku použijeme algoritmu, ktorý takéto riešenie nájde. V riešení uvažujeme iba deterministické hry s nulovou sumou. *Hra s nulovou sumou* je taká, kde zlepšenie šance na výhru jednému hrá ovi zníži šancu druhému. [1]

5.2 MiniMax

Najjednoduchšie riešenie problému riešenia hry predstavuje použitie algoritmu *MiniMax* [9]. Algoritmus je založený na ohodnotení stromu h adania (Obr. 8)



Obr. 8: Strom hľadania v MiniMax algoritme

Strom h adania je kore ový strom, ktorého uzly predstavujú stav hry v danom momente (napr. rozloženie šachovnice). Deti uzla predstavujú stavy, v ktorých sa hra ocitne, ak hrá ktorý je na rade vykoná povolený ah. Ke že sa hrá i pri svojich ahoch striedajú, h bka uzlu v strome ur uje, ktorý hrá je na ahu. Špeciálne uzly sú:

- kore – reprezentuje po iato ný stav hry
- listy – stavy, v ktorých hra kon í

Uzly sú ohodnotené celým íslom nasledovne:

- 0 – ak hra vedená z daného stavu skon í remízou
- 1 – ak hru vyhrá prvý hrá

- -1 – ak hru vyhrá druhý hrá

Ohodnotenie uzlov vlastne predstavuje šancu hrá a vyhra hru, pri om v tomto prípade sa na ohodnocovanie uzlov pozeráme vždy z poh adu prvého hrá a. Ten sa snaží maximalizova svoju šancu na výhru, preto z možných ahov vyberá taký, ktorý má maximálne ohodnotenie. Jeho oponent, druhý hrá , sa snaží rovnako maximalizova svoju šancu vyhra . Ke že sa ale jedná o hru s nulovou sumou, automaticky to znamená minimalizova šancu vyhra prvého hrá a. Druhý hrá preto vyberá z možných ahov taký, ktorý má minimálne ohodnotenie. Z tohto dôvodu je prvý hrá ozna ovaný ako MAX, druhý ako MIN, rovnako sú týmito pojmami ozna ované uzly, z ktorých prislúchajúci hrá i vykonávajú ah. Na základe tohto sú jednotlivé uzly stromu h adania ohodnotené nasledovným algoritmom:

1. listom priradíme hodnotu -1, 0, 1 pod a pravidiel hry
2. hodnota MAX uzla je maximom z ohodnotení jeho detí
3. hodnota MIX uzla je minimom z ohodnotení jeho detí

Algoritmus kon í ohodnotením kore a.

Rozvíjanie uzlov môžeme v algoritme MiniMax výhodne implementova ako h adanie do h bky, ím si zabezpe íme lineárnu pamä ovú náro nos v závislosti od h bky stromu h adania, t.j. O(n). Pri svojej innosti však potrebuje prejs celým stromom h adania, o ho robí asovo neprijate ne náro ným, t.j. asová náro nos je $O(k^{d})$, kde k je faktor vetvenia a d je h bka stromu h adania.

Zápis algoritmu MiniMax v pseudokóde je zobrazený na Obr. 9

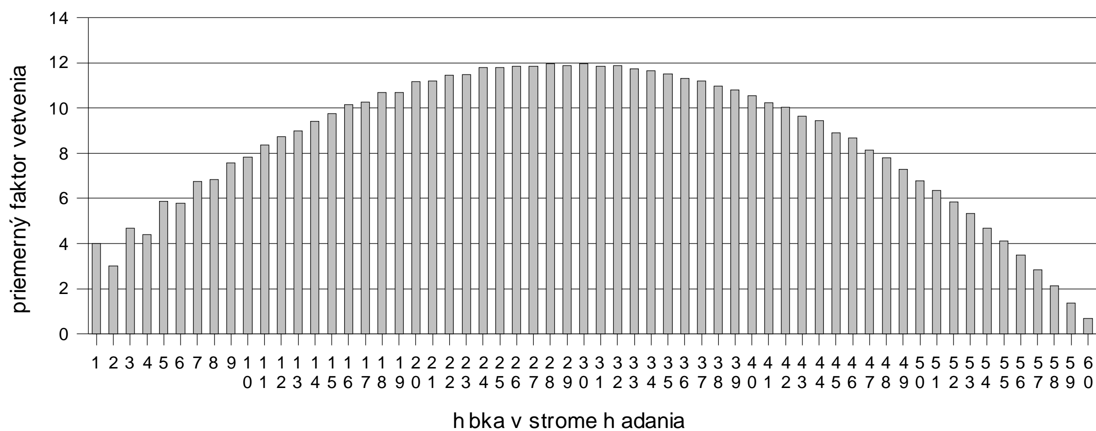
```
//vstupný bod algoritmu
//pos: po iato ná pozícia hry
int MiniMax(pos){
     return MaxValue(pos);
}
//vypo íta ohodnotenie MAX uzla
//pos: pozícia hry, z ktorej vykonáva  ah MAX
int MaxValue(pos){
     if(pos is end position)
          return eval(pos);
     moves = generate(pos);
     score = -INFINITY;
     foreach(move in moves) {
          make(move);
          score = MAX(score, MinValue(move));
          undo(move);
     }
     return score;
}
//vypo íta ohodnotenie MIN uzla
//pos: pozícia hry, z ktorej vykonáva  ah MIN
int MinValue(pos){
     if(pos is end position)
```

```
            return eval(pos);
      moves = generate(pos);
      score = +INFINITY;
      foreach(move in moves) {
            make(move);
            score = MIN(score, MaxValue(move));
            undo(move);
      }
      return score;
}
```

Obr. 9: Algoritmus MiniMax - presudokód

5.3 Ve kos stromu h adania

Pre stanovenie ve kosti stromu h adania potrebujeme pozna faktor vetvenia. Je to priemerný po et detí uzla. Pre hru reversi sme experimentálne prototypom (pozri 9 Odhad faktoru vetvenia) na základe prechodu stromu h adania zistili faktor vetvenia. Vzh adom napovahu nášho riešenia je potrebné vedie závislos faktora vetvenia na h bke stromu, preto ho uvádzame vo forme funkcie (Obr. 10).



Obr. 10: Graf závislosti faktora vetvenia v strome h adania od h bky uzla

Po et uzlov v strome h adania môžeme potom odhadnú pod a vz ahu $C \approx \sum_{k=0}^{n} \prod_{i=0}^{k} k_i$,kde k_i je faktor vetvenia v h bke i, na: $2{,}27 \times 10^{53}$

Napriek tomu, že tento odhad je približný, vidíme, že po et uzlov v strome je príliš ve ký. Preto je potrebné použi vylepšenie algoritmu MiniMax, ktoré redukuje po et uzlov, ktoré je potrebné prezrie .

5.4 Alfa beta usekávanie

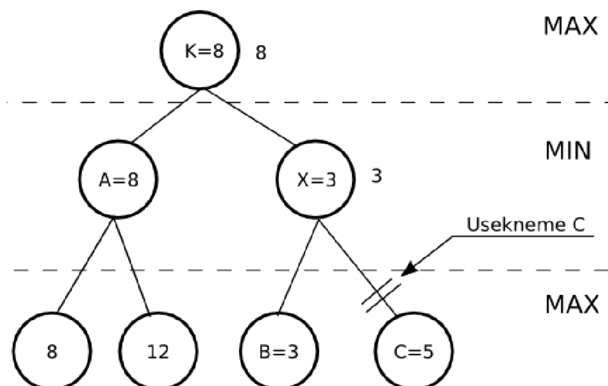
Princíp fungovania

Alfa beta usekávanie je vylepšením MiniMax algoritmu. Spo íva na vlastnostiach[7]:

1. $MAX(A, MIN(B, C)) = A$ bez oh adu na C ak $A \geq B$
2. $MIN(A, MAX(B, C)) = A$ bez oh adu na C ak $A \leq B$

Interpretácia týchto vlastností v strome h adania je nasledovná:

A, B, C predstavujú uzly v strome h adania, pri om rovnakým ozna ením vyjadrujeme aj ich ohodnotenie. Uvažujme prípad 1 zobrazený na Obr. 11.



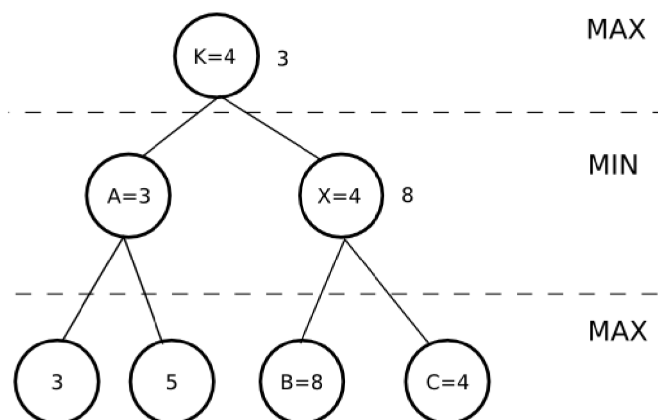
Obr. 11: Ukážka algoritmu alfa beta usekávanie

Predpokladajme, že na ahu je momentálne hrá MAX, uzol *K*. Ke že môže vykona ah, ktorým sa dostane do uzla s ohodnotením *A*, kore ový uzol *K* nemôže by ohodnotený nižšou hodnotou ako je *A*. Ohodnotenie zvyšného ahu *X* ešte nepoznáme. Po vykonaní ahu *X* je však na rade MIN. Ten môže vykona ah s ohodnotením *B*, a teda ohodnotenie uzla *X* nemôže vystúpi nad *B*, t.j. $X = MIN(B, C) \leq B$. Ak je navyše $B \leq A$, platí aj $X \leq A$, a teda $K = MAX(A, X) = A$.

Analogicky by sme mohli na strome h adania ukáza aj vlastnos 2.

V oboch prípadoch (splnenie podmienky 1 alebo 2) je teda ohodnotenie uzla K nezávislé na ohodnotení uzla C. Pri preh adávaní stromu h adania môžeme teda uzol C ignorova , iže useknú celý podstrom, ktorý v tomto uzle za ína.

Implementácia algoritmu generuje deti uzla a preh adáva ich v presne ur enom poradí. V našich príkladoch uvažujeme, že je to v podarí z ava doprava pod a obrázku. ahko si môžeme ukáza , že usekávanie podstromu je závislé na poradí, v ktorom sú uzly vygenerované.



Obr. 12: Ak zmeníme poradie vrcholov, k useknutiu nemusí dôjsť

Na Obr. 12 sme vo iObr. 11 navzájom zamenili avý a pravý podstrom . Táto zmena mala za následok to, že po ohodnotení uzla *A* vieme, že uzol *K* nemôže klesnú pod hodnotu 3. Po ohodnotení uzla *B* vieme, že *X* nemôže stúpnu nad 8. Avšak podmienka $A \geq B$ nie je splnená, takže useknutie nemôžeme vykona . Teda hodnota uzla $C = 4$ sa musí bra do úvahy a v tomto prípade sa ukazuje, že je to aj výsledné ohodnotenie uzla *X* a *K*.

Efektívnos algoritmu alfa beta osekávania je teda závislá na poradí, v ktorom sú deti uzla vygenerované. Na stanovenie tohto poradia sa používajú rôzne heuristiky popísané v kapitole 5 Heuristiky.

Implementácia afla beta usekávania

Alfa beta usekávanie je implementované ako preh adávanie do h bky, pri ktorom sa na základe ohodnotení preskúmaných uzlov aktualizujú premenné α a β , ktoré slúžia pri ur ovaní, i uzol môže by useknutý:

- α predstavuje maximálnu hodnotu, ktorú zatia uzol MAX dosiahol (posledný MAX uzol na ceste k aktuálnemu uzlu)
- β predstavuje minimálnu hodnotu, ktorú zatia uzol MIN dosiahol (posledný MIN uzol na ceste k aktuálnemu uzlu)

Na za iatku sú nastavené nasledovne: $\alpha = -\infty$, $\beta = \infty$. Dvojica (α, β) sa nazýva okno.

Pri stanovovaní ohodnotenia MAX uzla sa aktualizuje α vždy na maximum zo zatia získaných ohodnotení detí (MIN uzly). Ak sa α zmení tak, že bude plati $\alpha \geq \beta$, predstavuje to splnenie podmienky vlastnosti 2 $MIN(A, MAX(B, C)) = A$, a teda alšie deti MAX uzla nie je potrebné prešetri .

Pri stanovení ohodnotenia MIN uzla sa aktualizuje β vždy na minimum zo zatia získaných ohodnotení detí (MAX uzly). Ak sa β zmení tak, že bude plati $\alpha \geq \beta$, predstavuje to splnenie podmienky vlastnosti 1 $MAX(A, MIN(B, C)) = A$, a teda alšie deti MIN uzla nie je potrebné prešetri .

Algoritmus vyjadrený v pseudokóde je na Obr. 13.

```
//vstupný bod algoritmu
//pos: po iato ná pozícia hry
```

```
int AlphaBeta(pos) {
     return MaxValue(pos, -INFINITY, +INFINITY);
}
//vupo íta ohodnotenie MAX uzla
//pos: pozícia, z ktorej vykonáva  ah MAX
int MaxValue(pos, alpha, beta){
     if(pos is end position)
          return eval(pos);
     moves = gemerate(pos);
     foreach(move in moves){
          make(move);
          alpha = MAX(alpha, MinValue(move, alpha, beta));
          undo(move);
          if(alpha >= beta)
               return beta;
     }
     return alpha;
}
//vypo íta ohodnotenie MIN uzla
//pos: pozícia, z ktorej vykonáva  ah MIN
int MinValue(pos, alpha, beta){
     if(pos is end position)
          return eval(pos);
     moves = gemerate(pos);
     foreach(move in moves){
          make(move);
          beta = MIN(beta, MaxValue(move, alpha, beta));
          undo(move);
          if(alpha >= beta)
               return alpha;
     }
     return beta;
}
```

Obr. 13: Algoritmus Alfa beta usekávanie - pseudokód

5.5 NegaMax

NegaMax [7] je implementa né vylepšenie MiniMaxu, ktoré využívajú všetky praktické implementácie ostatných algoritmov po núc od alfa beta usekávania.

Je založené na tom, že ak je ohodnotenie nejakého uzla vzh adom na hrá a MAX A , tak potom ohodnotenie toho istého uzla vzh adom na hrá a MIN je $-A$. Vyplýva to z toho, že výhra hrá a MAX predstavuje prehru hrá a B.

Využitím tejto vlastnosti môžeme MiniMax upravi nasledovne:

1. Ohodnotenie uzla bude vždy vzh adom na toho hrá a, ktorý z daného stavu vykonáva svoj ah.
2. V aka bodu 1 teraz v každom ahu maximalizuje každý hrá priamo svoju šancu vyhra . Vnútorný uzol stromu sa teda ohodnotí nasledovne: $V = \underset{i}{MAX}\{-V_i\}$, kde V_i je ohodnotenie die a a uzla V. $-V_i$ predstavuje transformáciu ohodnotenia tak, aby bolo vyjadrené vzh adom na hrá a, ktorý vykonáva svoj ah z uzla V.

Tento prístup umož uje teda zjednoti poh ad na MAX a MIN uzly.

5.6 Alfa beta usekávanie s NegaMax rozšírením

V tomto vylepšení platí to, o pre NegaMax. Rozdiel vo i klasickému alfa beta usekávaniu je navyše v práci s α a β koeficientami:

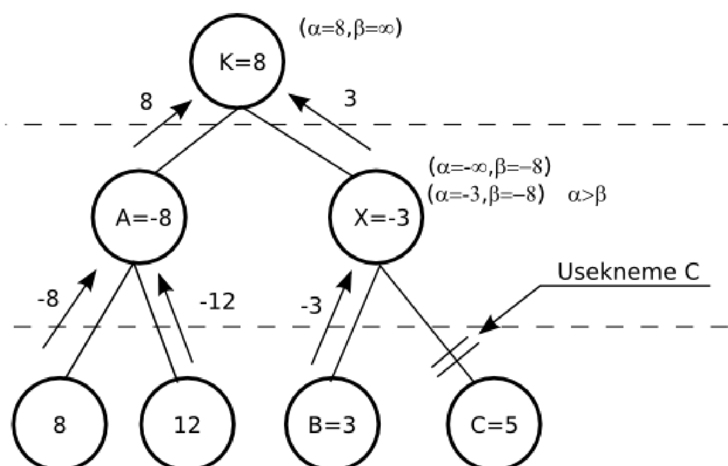
- α je zatia najlepšie ohodnotenie skúmaného uzla.
- β je zatia najlepšie ohodnotenie rodi a skúmaného uzla.

Koeficienty α a β sú rovnako vyjadrené vzh adom na aktuálny uzol, preto pri prechode na potomka je potrebné tieto koeficienty aktualizova :

- $\alpha^{new} = -\beta^{old}$
- $\beta^{new} = -\alpha^{old}$

V procese získavania maxima z ohodnotení detí aktualizujeme rovnako α^{new} na maximálnu zatia zistenú hodnotu, teda hodnota α^{new} nemôže klesnú . Ak sa splní podmienka $\alpha^{new} \geq \beta^{new}$, platí rovnako $-\alpha^{new} \leq -\beta^{new}$, o je $-\alpha^{new} \leq \alpha^{old}$. Z poh adu rodi a vyhodnocovaného uzla to znamená, že ohodnotenie uzla $-\alpha^{(new)}$ už nemôže nijako zlepši jeho ohodnotenie. Preto pri splnení tejto podmienky ukon ujeme prezeranie alších potomkov skúmaného uzla.

Na Obr. 14 je na príklade ukázaná táto situácia.



Obr. 14: Ilustrácia funkcie algoritmu alfa beta usekávania s negamax rozšírením

Ohodnotenie avého podstromu je $A = -8$, ím sa hodnota uzla K aktualizuje na $K = -A = 8$ a okno na $(\alpha, \beta) = (8, \infty)$. Pri ohodnocovaní uzla X sa použije okno

$(\alpha^{(new)}=\beta^{(new)})=(-\beta,-\alpha)=(-\infty,-8)$. Po ohodnotení uzla $B=3$ je hodnota $\alpha^{(new)}$ aktualizovaná na $\alpha^{(new)}=-3$, ím je okno upravené na $(\alpha^{(new)},b^{(new)})=(-3,-8)$. Ke že platí $\alpha^{(new)}>\beta^{(new)}$, uzol C je useknutý.

Algoritmus takto upraveného Alfa beta usekávania vyjadrený v pseudokóde je Obr. 15.

```
//vstupný bod algoritmu
//pos: aktuálna pozícia hracej plochy
//alpha, beta: okno h adania nastavené na -∞,∞
int AlphaBeta(pos, alpha, beta){
     if(pos is end position)
          return eval(pos);
     score = - INFINITY;
     moves = gemerate(pos);
     foreach(move in moves) {
          make(move);
          cur = - AlphaBeta(move, -beta, -alpha); //ohodnotenie
die a a
          if'(cur > score) score = cur;      //aktuálne najlepšie
riešenie
          if(score > alpha) alpha = score;
          undo(move);
          if(alpha >= beta) return alpha;         //odseknutie
     }
     return score;
}
```

Obr. 15: Algoritmus Alfa beta usekávanie s NegaMax rozšírením – pseudokód [7]

5.7 NegaScout

NegaScout je vylepšené riešenie usekávania založené na alfa beta usekávaní. Snaží sa ešte vo výraznejšej miere redukova po et uzlov, ktoré je potrebné preh ada . Je založený na myšlienke, že ke že vä šina uzlov po prvom die ati sa usekne, ich presné ohodnocovanie je nepotrebné. Preto sa NegaScout snaží preh adávaním s minimálnym oknom ukáza , že sú horšie. Preh adávanie s minimálnym oknom znamená, že α a β koeficienty sa nastavia tak, aby platilo: $\alpha=\beta-1$. Prvý uzol je ohodnotený vždy preh adávaním s pôvodným oknom.[7]

V prípade, že h adanie s minimálnym oknom nájde lepšie ohodnotenie ako je aktuálne, preh adávanie musíme zopakova s vä ším oknom, aby sme získali správne ohodnotenie. Iba ak h adanie s minimálnym oknom vráti nižšie ohodnotenie, môžeme tento výsledok akceptova .[7]

Algoritmus NegaScout zapísaný v pseudokóde je na Obr. 16.

```
//vstupný bod algoritmu
//pos: aktuálna pozícia hracej plochy
//d: hlbka hladania
//alpha, beta: okno h adania nastavené na -∞,∞
int NegaScout(pos, d, alpha, beta){
     if(pos in end position)
          return eval(postion);
     score = -INFINITY;
     n = beta;
     moves = generate(pos);
     foreach(move in moves) {
          make(move);
          cur = - NegaScout(pos, d-1, -n, -alpha);
          if(cur > score) {
               if(n == beta || d <=2)
                    score == cur;
               else
                    score = - NegaScout(pos, d-1, -beta, -cur);
          }
          if(score > alpha) alpha = score;
          undo(move);
          if(alpha >= beta) return alpha;
          n = alpha + 1;
     }
     return score;
}
```

Obr. 16: Algoritmus NegaScout - pseudokód[7]

5.8 MTD(f)

MTD(f) je vylepšením algoritmu usekávania, ktoré vždy preh adáva s minimálnym oknom. Tento prístup môže vies vä šiemu usekávaniu uzlov, avšak h adanie s minimálnym oknom nezistí presné ohodnotenie uzla ale iba jeho odhad. Niekedy musí preh adáva ten istý podstrom opä s upraveným oknom, aby sa zistilo správne ohodnotenie. [7]

Algoritmus MTD(f) zapísaný v pseudokóde je na Obr. 17.

```
//vstupný bod algoritmu
//pos: aktuálna pozícia hracej plochy
//f: prvý odhad ohodnotenia hracej plochy
int MTDF(pos,f){
     int score = f;
     upperBound = +INFINITY;
     lowerBound = -INFINITY;
     while(upperBound > lowerBound){
          if(score == lowerBound)
               beta = score + 1;
          else
               beta = score;
          score = AlphaBeta(pos, beta-1, beta);
          if(score < beta)
               upperBound = score;
          else
               lowerBound = score;
     }
     return score;
}
```

Obr. 17: Algoritmus MTD(f) - pseudokód[7]

5.9 Heuristiky

Heuristiky sa využívajú v algoritmoch s usekávaním na zoradenie ahov pod a ich s ubnosti. Ak sa lepšie uzly ohodnocujú skôr, zlepšuje to šancu sa useknutie podstromu a redukovanie po tu uzlov, ktoré je potrebné preh ada .

5.9.1 Heuristika vražedného ahu

Táto heuristika je založená na myšlienke, že rôzne pozície nachádzajúce sa v uzloch rovnakej h bky majú podobnú povahu. Ak je nejaký ah v jednej asti stromu dobrý, potom bude s vysokou pravdepodobnos ou dobrý aj v inej asti stromu na tej istej h bke.

Preto si na každej h bke zapamätávame uzly, ktoré viedli k usekávaniu stromu. Takéto uzly predstavujú vražedné ahy. Deti každého z uzlov potom preh adávame v takom poradí, aby sme vražedné ahy preskúmali ako prvé.[7]

5.9.2 Heuristika založená na histórii ahov

Je rozšírením heuristiky vražedného ahu. V tabu ke sa uchováva skóre priradené uzlom. Zakaždým, ke na základe ohodnotenia uzla usekneme as stromu preh adávania, zvýšime uzlu skóre.

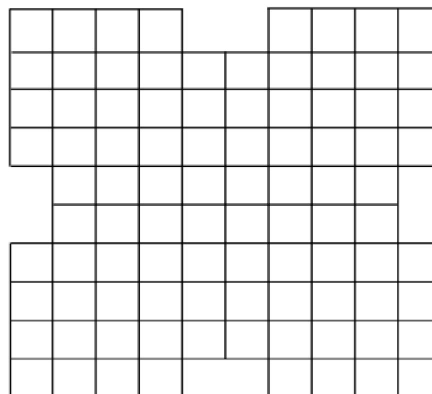
Skóre môžeme navyšova napr. o hodnotu $2^{h_{max}-h}$, kde h je h bka uzla v strome h adania. Takýto vz ah je výhodné použi , aby vyššie skóre získali stavy bližšie pri koreni, kedže usekávanie v nižšej h bke usekne vä ší podstrom.

Pri preh adávaní detí uzla postupujeme v poradí ur enom pomocou získanej tabu ky. Uzly s vyšším skóre sú preh adávané skôr.[7]

5.9.3 Problémovo závislé ohodnotenie vhodnosti pozície

alší spôsob ohodnotenia vhodnosti pozícií môže by problémovo závislý, vychádzajúci z pravidiel hry. Spôsob ohodnotenia, ktorý použili [3] vo svojej práci pre hru reversi, je v zásade založený na po ítaní rozdielu medzi po tom kame ov oboch hrá ov. Kamene na rôznych pozíciách majú rôzne kvalitné postavenie – napr. kamene v rohoch nemôže súper nijako ovládnu . Rôznym polí kam sú preto priradené rôzne váhy, ktoré sa berú do úvahy.

Hracia plocha, pre ktorú zostavovali heuristickú funkciu, je na Obr. 18.



Obr. 18: Hracia plocha, pre ktorú autori zostavovali heuristikú funkciu

Heuristická funkcia berie v úvahu tieto zložky:

Rozdiel po tu kame ov δ_K (piece differential)

Ve mi jednoduchá metrika, ktorá po íta rozdiel po tu polí ok, ktoré má hrá 1 a hrá 2 obsadené. Je jednoduchá, no potrebná, ke že koniec hry sa stanovuje iba na základe δ_K .

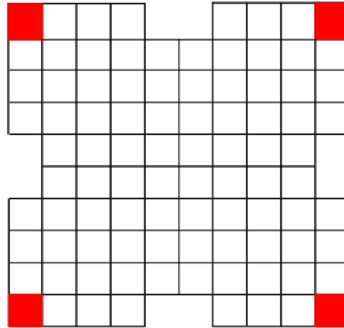
Rozdiel po tu ahov δ_T (mobility differential)

Je rozdiel po tu ahov, ktoré môže z daného uzla vykona hrá 1 a hrá 2. Na výpo et je potrebné vedie , ktorý hrá je na rade, aby sa ur ilo správne poradie v od ítaní.

Rozdiel po tu štvorohov $\delta_{R^{(4)}}$ (4-corners differential)

Je rozdiel po tu štvorrohov obsadených hrá om 1 a hrá om 2. Ako štvorrohy ozna ili autori zvýraznené polí ka na Obr. 19.

Rohy sú považované za dobré ahy. Nielenže ich súper nemôže získa , predstavujú aj strategický

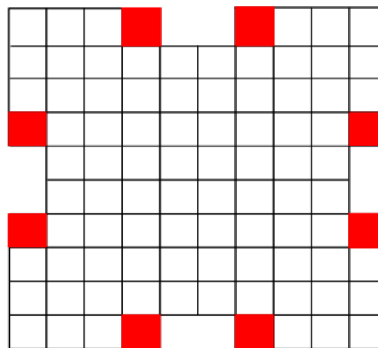


Obr. 19: Štvorrohy na hracej ploche reversi

bod pre získavanie súperových kameňov.

Rozdiel po tu osemrohov $\delta_{R^{(8)}}$ (8-corners differential)

Je rozdiel po tu osemrohov obsadených hrá om 1 a hrá om 2. Ako osemrohy ozna ili autori zvýraznené polí ka na Obr. 20.



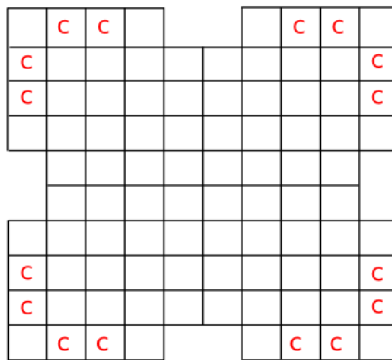
Obr. 20: Osemrohy na hracej ploche reversi

Podobne ako štvorrohy sú považované za dobré ahy. Vzh adom na odlišný vz ah k ostatným polí kam ich autori vy lenili zvláš .

Rozdiel po tu C polí ok δ_C (C-squares differential)

Je rozdiel po tu C polí ok obsadených hrá om 1 a hrá om 2. Ako C-polí ka ozna ili autori zvýraznené polí ka na Obr. 21.

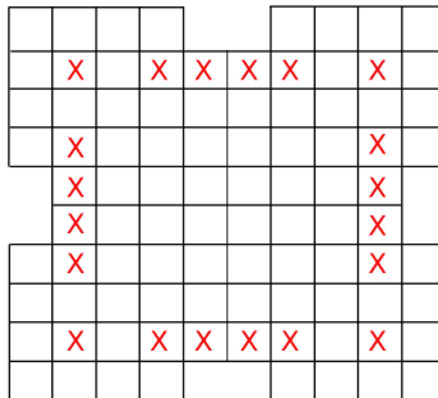
Tieto polí ka sú považované aj za dobré aj za zlé. Zlé, pretože umož ujú súperovi obsadi   rohy. Dobré, lebo sa nachádzajú na okraji.



Obr. 21: C-štvorce na hracej ploche hry reversi

Rozdiel po tu X-polí ok δ_X (X-squares differential)

Je rozdiel po tu X-polí ok obsadených hrá om 1 a hrá om 2. Ako X-polí ka ozna ili autori zvýraznené polí ka na Obr. 22.



Obr. 22: X-štvorce na hracej ploche hry reversi

Tieto polí ka sú považované za zlé, pretože dávajú súperovi šancu obsadi   rohy.

Rozdiel po tu hrani ných polí ok δ_H (frontier differential)

Je rozdiel po tu hrani ných polí ok obsadených hrá om 1 a hrá om 2. Hrani ké polí ko je také, ktoré susedí v ubovo nom smere s aspo   jedným prázdnym polí kom.

Rozdiel po tu stabilných polí ok δ_S (Stable pieces differential)

Je rozdiel po tu stabilných polí ok obsadených hrá om 1 a hrá om 2. Stabilné polí ko je také, ktoré nemôže protihrá   teraz a ani nikdy v budúcnosti nijako prevzia .

Rozdiel po tu sendvi ových polí ok δ_B (Sandwich Squares Differential)

Je rozdiel po tu sendvi ových polí ok obsadených hrá om 1 a hrá om 2. Sendvi ové polí ko je také, ktoré z oboch strán v nejakom smere susedí s protihrá om.

Ošetrenie vyhladenia

Potreba špeciálneho ošetrenia prípadu, ak sú všetky kamene jedného hrá a prevzaté. V takomto prípade má heuristická funkcia vraca $-\infty$. Je to ošetrenie z poh adu hrani ných polí ok, z ktorého poh adu je vyhladenie dobrým stavom.

5.10 Transpozi ná tabu ka

Je to tabu ka, do ktorej sa k pozíciám hracej plochy ukladá jej ohodnotenie. Využíva sa to, že do rovnakej pozície sa vieme dosta rôznymi spôsobmi. Preto je výhodné zapamäta si už vypo ítané ohodnotenie hracej plochy a neskôr, ke sa rovnaká pozícia zopakuje, ho len z transpozi nej tabu ky pre íta . Týmto spôsobom alej redukujeme po et potrebných preh adaných uzlov.

Ke že vzh adom na pamä ové nároky a výpo tovú réžiu nie je možné uchováva si všetky ohodnotenia a kontrolova všetky uzly vo i transpozi nej tabu ky, je potrebné pri návrhu stanovi :

1. ohodnotenia ktorých uzlov sa budu uklada do tabu ky
2. ak tabu ka bude v pamäti zabera maximálnu povolenú ve kos , ako sa ur í, ktoré ohodnotenia sa budú vyhadzova
3. ktoré uzly sa budu vo i transpozi nej tabu ke kontrolova

5.11 Symetrické pozície

Dve hracie plochy sú symetrické, ak oto ením, preklopením pod a osi alebo stredu jednej získame tú druhú. V niektorých hrách, ako aj v reversi (šach nie) sú symetrické pozície vlastne rovnaké. Z h adiska ohodnocovania stromu preh adávania môžeme takéto pozície považova za totožné, ím opä redukujeme po et uzlov, ktoré potrebujeme preskúma . Riešením takéhoto problému môže by tabu ka, do ktorej vkladáme ohodnotenie uzla. K ú , pod ktorým ho vkladáme, však musí sp a vlastnos , aby bol pre symetrické plochy rovnaký. Navyše musí by jednozna ný alebo musí ma o najmenej kolízií, aby sa nemuseli plochy pracne porovnáva vo i všetkým možným transformáciám.

5.12 Serverové a klientske preh adávanie

Strom h adania pre uvažované hry je ve mi ve ký. Jeho preh adávanie najednom po íta i, aj s množstvom vylepšení, zaberie ve mi ve a asu. H adanie riešenia je preto rozdelené pomocou BOINC systému medzi viacero po íta ov – klientov. Tí po ítajú iastkové úlohy, ktoré im server pride uje, a následne aj vyhodnocuje výsledky.

Z povahy BOINC systému a riešenia vyplýva, že server aj klient budú obaja rieši preh adávanie stromu, ohodnocovanie uzlov a usekávanie nepotrebných astí stromu. Každý z nich však k tejto úlohe bude pristupova iným spôsobom daným ich rôznymi špecifikami.

Špecifiká servera a klienta

Server

- vä šia dostupnos pamäte
- potreba vytvori úlohy pre klientov
- nemôže sa zamestna jednou úlohou, ktorej výpo et trvá ve mi dlho

Klient

- obmedzené množstvo pamäte a výkonu, ktorým hos ovský po íta disponuje
- je ur ený na po ítanie jednej iastkovej úlohy. Môžeme ho ou úplne zamestna (vy aži).

Spôsob realizácie preh adávania

Klient

- preh adávaním do h bky prostredníctvom niektorého z vylepšených alfa beta usekávaní so zapracovanými heuristikami, tabu kami a podobne

Server

- pri preh adávaní musí preh adáva do šírky, aby vygeneroval dostatok úloh pre klientov
- pri preh adávaní musí preh adáva zárove do h bky, aby zišiel dostato ne hlboko do stromu, a aby mohol klientom generova reálne vyriešite né úlohy
- používa usekávanie podstromu h adania po asynchrónnom získaní výsledkov od klienta

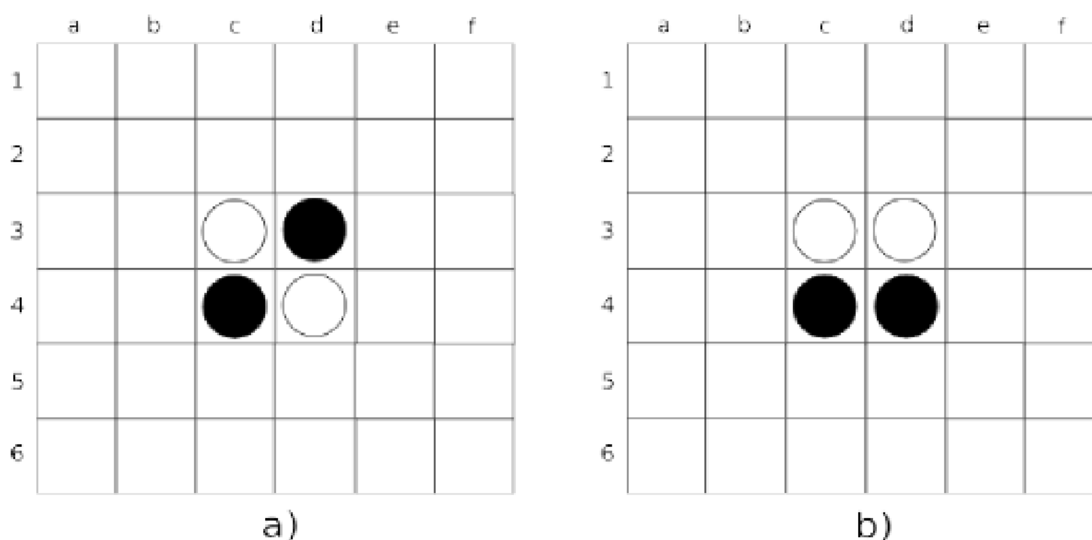
Ako poukazuje Liang [7], použitie konkrétneho algoritmu v klientskej asti je otázkou experimentálneho vyskúšania. Pod a autora sa nedá poveda , ktoré z uvedených vylepšených riešení je lepšie. Ke že charakteristiky stromu preh adávania sú rozdielne pre každú hru, pre každú sa môže ukáza iný algoritmus ako vhodný.

6 Existujúce riešenia

V tejto asti sú popísané existujúce riešenia symetrických hier reversi a GO. Tieto riešenia boli dosiahnuté na jednom po íta i, na rozdiel od nášho projektu, kde budeme rieši problémy distribuované.

6.1 Reversi

Dr. Joel Feinstein v roku 1993 vyriešil problém reversi pre ve kos hracej plochy 6x6. [5] Dva týždne po íta ového asu mu trvalo, aby sa dopracoval k výsledku. Výsledok znel, ak obaja hrá i hrajú perfektne, tak 20:16 vyhrá hrá , ktorý ide ako druhý v poradí (20 krát vyhrá druhý hrá , zatia o prvý hrá vyhrá len 16 krát). Po as behu programu bolo vygenerovaných okolo 40 mld. pozícií. Feinstein tiež spo ítal trvanie výpo tu pre hraciu plochu ve kosti 8x8. Pod a jeho výpo tov by to trvalo 10^{14}, približne 3,8 milióna rokov. Riešenie reversi 6x6 s alternatívnou štartovacou pozíciou (Obr. 23b) trvalo Feistaeinovi spo íta 5 týžd ov a vygenerovalo sa okolo 100 mld. pozícií. Tu bol pomer 19:17 v prospech druhého hrá a. Feinstein robil výpo ty na jednom po íta i.



Obr. 23: Počiatočné pozície: a)normálna b)alternatívna

Pri riešení použil preh adávanie hrubou silou za pomoci algoritmu alfa-beta usekávania s usporiadaním ahov pod a jeho ohodnocovacej funkcie. Program používal 70 KB pamäti.

6.2 GO

D a 19. októbra 2002 sa podariloErik van der Werf vyrieši hru GO pre ve kos hracej plochy 5x5. [17] Program pracoval tak, že prvý ah bol v strede hracej plochy. Ví azom bol ierny hrá (prvý v poradí ahania). Kone né skóre bolo 25 pre ierneho, celá hracia plocha bola obsadené iernymi kame mi.

Pri riešení boli použité algoritmus iteratívneho prehlbovania alfa-beta h adania (PVS - Principal Variation Search). V tomto algoritme boli použité:

1) transpozi ná tabu ka – mala 2 x 2^{24} položiek
2) rozšírené transpozi né osekávanie
3) symetria bola vyh adávaná v transpozi nej tabu ke
4) 2 killer moves
5) historické heuristiky
6) Bensonov algoritmus
7) heuristické ohodnotenie pre pozície, ktoré neboli ur ené Bensonovým algoritmom

Riešenie bolo nájdené v 22. úrovni (pre prázdnu plochu v 23. úrovni). Výpo et prebiehal na po íta i P4 2.0Ghz. Po as výpo tu bolo vygenerovaných približne 4,5 mld. uzlov. as trvania tohto výpo tu bol asi 4 hodiny.

Nadôležitejším pri riešení bol Bensonov algoritmus, ktorý zmenšil h bku preh adávaného stromu hry. V riešení boli použité pravidlá ko, teda riešenie je nezávislé na super-ko pravidlách. Je to z toho dôvodu, že pri super-ko je obtiažne sa vyhnú všetkým dlhým cyklom.

V decembri 2002 Erik van der Werf vylepšil svoj program na riešenie GO 5x5, nazval ho MIGOS (Mini GO Solver). alej rozšíril program o riešenie hier, ktoré nie sú otvorené v strede hracej plochy. Program bol schopný rieši akéko vek otvorenie na prázdnej hracej ploche rozmerov 5x5.

7 Možnosti ukladania stromu na disk

7.1 Reprezentácia stavu hry

Dátová štruktúra, s ktorou pracujeme, je strom, konkrétne n-árny strom, kde uzly sú ohodnotené a každý uzol predstavuje pozíciu na šachovnici. Pre uvažovanú hru reversi má šachovnica rozmery 8×8 , pritom každé polí ko môže ma 3 rôzne stavy (ierny, biely, prázdny). Na reprezentáciu týchto 3 stavov nám treba 2 bity ($2^2 = 4, 4 > 3$). Teda na reprezentáciu stavu šachovnice potrebujeme $2 \times 64 = 128$ bitov. Takáto šachovnica má viacero potomkov - alšie šachovnice, ktoré sú od aktuálnej šachovnice vzdialené 1 ah (teda + 1 ah). Šachovnica má zárove ohodnotenie (nemá ho pri vytvorení, ale získa ho až pri ohodnocovaní stromu). To môže ma 3 hodnoty - vyhral biely, ierny, remíza.

7.2 Ve kos uložených dát

Strom celej hry reversi $8x8$ má maximálnu h bku 64. Faktor vetvenia sa zo za iatku pohybuje okolo 6 (podrobnejšie kapitola 9 Odhad faktoru vetvenia) a celý strom obsahuje približne $2{,}2716\, x10^{53}$ uzlov. Z toho bude as uložená na klientovi a as na serveri.

Predpokladajme, že máme k dispozícii 500 GB diskového priestoru na uloženie stromu. Na jeden stav potrebujeme $128b = 16B$. Teda na disk sa nám zmestí cca $3{,}355 \times 10^{10}$ uzlov. Tento výpo et nie je ale správny, lebo nezah a informácie o hierarchii ani o ohodnotení jednotlivých uzlov. Preto môžeme pouåi takúto dátovú štruktúru (Tab. 1), ktorá implementuje aj hierarchiu a ohodnotenie:

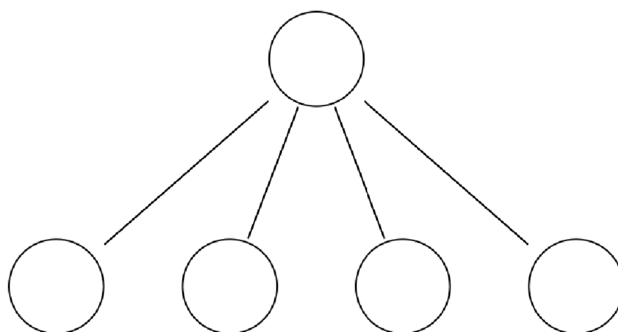
Uzol	
Stav hry	128b
Rodi ovský stav	128b
Ohodnotenie	2b

Tab. 1: Pamä ové nároky uzla

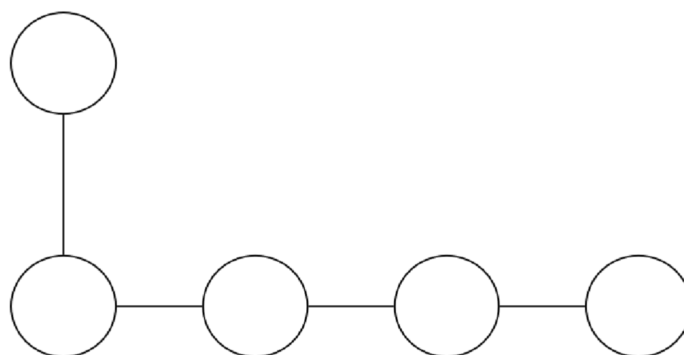
Potom by sme potrebovali na uloženie 1 záznamu 258b, o je 33 B, teda na disk by sa nám zmestilo $1{,}627 \times 10^{10}$ uzlov. To zodpovedá celému stromu do h bky 11 (pod a 5 Ve kos stromu h adania).

7.3 Reprezentácia stromu

Pre reprezentáciu stromu existujú dva základné spôsoby zobrazené na Obr. 24 a Obr. 25.



Obr. 24: Jedna z možností, ako uložiť strom, každý potomok má uloženú referenciu na rodiča



Obr. 25: Iná možnosť uloženia stromu, prvý potomok má referenciu na rodiča a na súrodenca, ten potom odkazuje na ďalšieho súrodenca

V strome budeme potrebova jednoducho zisti nasledovníka a predchodcu daného uzla, preto je pod a m a prvá možnos pre nás výhodnejšia.

7.4 Konkrétne možnosti ukladania stromu na disk

7.4.1 Existujúci systém súborov

V prípade existujúcich systémov súborov (alej len FS) by boli jednotlivé stavy uložené ako súbory. Stav by mohol by uložený bu ako meno súboru alebo ako obsah súboru. Postupnos stavov by bola reprezentovaná štruktúrou adresárov na disku. Ohodnotenie daného stavu by potom mohlo by uložené tiež v súbore alebo v mene súboru.

Z toho vyplýva, že potrebujeme FS, ktorý bude ma o najmenšiu minimálnu ve kos bloku kvôli efektívnemu ukladaniu na disk, bude podporova ve mi dlhé názvy súborov (ak sa ako identifikátor pozície použije meno súboru, cesta môže ma maximálne $64\times 64=2^{12}=4096$ znakov v prípade použitia kódovania 1 b = 1 znak). Tiež je potrebné, aby bol FS schopný pracova s ve kým množstvom malých súborov.

Výhodou tohto prístupu je, že hierarchiu uzlov reprezentuje samotný FS, teda v porovnaní so štruktúrou spomenutou v úvode bude FS schopný uloži zhruba dvojnásobok uzlov. To by stále zodpovedalo celému stromu do h bky 12.

V sú asnosti existujú FS, ktorým nerobí problém tých cca 3×10^{10} súborov. Hlavným problémom je minimálna ve kos bloku. Ve kos sektora disku neumož uje ma menšiu najmenšiu ve kos bloku ako 512 B. alším problémom je, že na vä šine FS sa pri tak malom bloku nedá vytvori dostato ne ve ký oddiel. Z dostupných FS pre bežné opera né systémy by z h adiska ve kosti oddielu pri minimálnej ve kosti bloku vyhovoval napr. ReiserFS4 [18], ReiserFS [19] (*nix) alebo ZFS (Sun Microsystems). Tie sú ale nevhodné kvôli obmedzenému maximálnemu po tu súborov nadisku (2^{32}) . Vä šina používaných FS, napr. FAT, NTFS, nesp a tieto požiadavky [8]. Z vhodných spomeniem napr. XFS [12], ZFS [14], JFS2 [6], UFS2 [15]. Aj tu ale ostáva problémom neefektívne ukladanie dát (na 1 stav potrebujeme 16 B, pritom na disku tento stav zaberie 512 B, o je 32 krát viac).

7.4.2 Vlastný FS

Týmto riešením by sme dokázali obís vyššie spomenuté limity. Realizácia by bola pravdepodobne pomocou priameho zápisu na disk, napr. do jedného ve kého súboru. Problém je však v implementácii riešenia. Tá sa zdá by príliš zložitá a nerealizovate ná v danom ase.

7.4.3 Databáza

Pod a [1] ju použili na ukladanie sekvencií genómu. Preto som testoval, i by bola použite ná aj v našom prípade.

V mojich pokusoch som predpokladal trojice parent - child - ohodnotenie_child. Použil som PostgreSQL 8.1 (alej len PG). Postupne som generoval databázu s cca $2{,}6 \times 10^{6}$ záznamov (na disku zaberá zhruba 500 MB). Po as generovania som meral efektivitu využívania diskového miesta (ko kokrát viac zaberajú dáta na disku).

V tejto reprezentácii potrebujeme 2×128 b na 2 šachovnice (v PostgreSQL reprezentované ako bitové vektory) a jeden boolean na ukladanie ohodnotenia (true, false, null).

Samotná PG databáza nebola nijak optimalizovaná, boli použité indexy na st pce parent a child. Select pod a stavu hry pracoval rýchlo (rádovo ms). Merania som robil vždy v jednej a tej istej tabu ke po spustení VACUUM FULL. VACUUM je pomerne asovo náro ná operácia (cca 2 minúty pre $2{,}7 \times 10^{6}$ záznamov), ale šetrí miesto (rádovo percentá) a optimalizuje uloženie dát v databáze. V Tab. 2 sú uvedené výsledky testu.

Po et záznamov	Efektivita(menšie íslo je lepšie)
8,50E+004	4,5
1,30E+006	6,9
2,70E+006	5,45

Tab. 2: Zavislos efektivity od po tu záznamov v DB

Z meraní vidno, že efektivita ukladania dát na disk je zhruba konštantná, a neklesá so zvä šujúcim sa množstvom dát. Pre úplnos informácie ešte v Tab. 3 uvediem limity pre ve kosti tabuliek v PG [11].

Max. ve kos DB	neobmedzená
Max. ve kos tabu ky	32 TB
Max. ve kos riadku	1,6 TB
Max. ve kos po a	1 GB
Max. po et riadkov v tabu ke	neobmedzený
Max. po et st pcov v tabu ke	250 – 1600, pod a typu
Max. po et indexov v tabu ke	neobmedzený

Tab. 3: Obmedzenia DB

7.5 Záver k ukladaniu dát

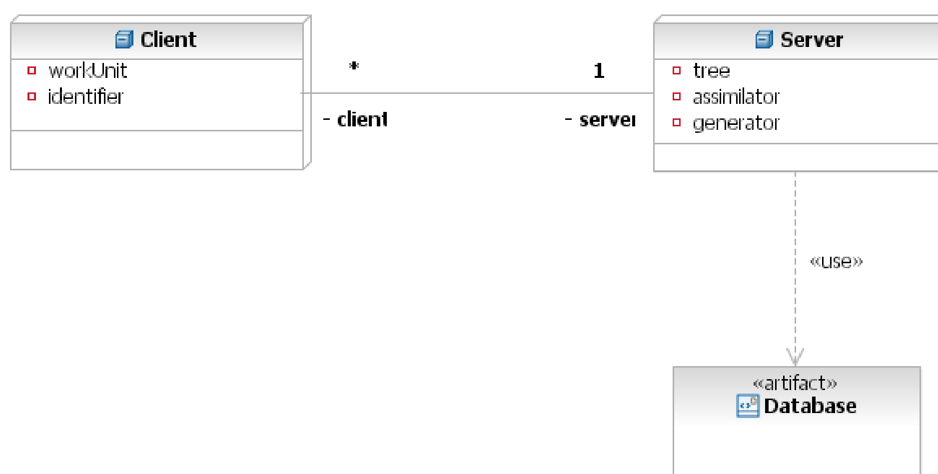
Ako najefektívnejší a najjednoduchší spôsob na ukladanie dát sa javí databáza. Bolo by možno dobré otestova , ako sa budú správa na tomto type dát iné databázy, napr. MySQL, SQlite, Firebird.

8 Návrh systému

Táto kapitola sa zaoberá návrhom systému nášho projektu. Hne na za iatku je tu otázka, i vôbec ide o systém. Nájs ve mi slabé riešenie hry je v podstate len jeden algoritmus, prispôsobený naším konkrétnym podmienkam. Na druhej strane, ke že sa jedná o využitie existujúceho systému na paralelné po ítanie - BOINC (pozri kapitola 3), ktoré obsahuje pojmy ako server a klient, tak je namieste hovori o architektúre - návrhu aj ke zna ne odlišného od ve kých softvérových produktov.

8.1 Návrh paralelného systému

Využitím systému BOINC sa nám do problému dostalo isté ohrani enie, ktoré definuje prvotnú architektúru klient - server. as servera sa ešte delí na dve hlavné komponenty (asti), a to generátor a asimilátor (Obr. 26), ktoré sú bližšie popísané v kapitole 3Démoni bežiaci na serveri. Keby sa to malo zhrnú , tak generátor sa stará o generovanie zadaní pre klientské po íta e, tzv. *workunit,* a asimilátor je zodpovedný za spracovanie výsledku od jednotlivých klientov.



Obr. 26: Celkový poh ad na systém

Systém je navrhnutý o najviac modulárny, tak ako dovo uje jazyk C. Dôvodom je použitie pre rôzne typy distribuovaných hier. Ako bolo práve spomenuté, naším implementa ným jazykom bude oby ajný jazyk C, hlavne a jedine kvôli rýchlosti, pretože asová náro nos riešenia je zna ne ve ká. Problém je riešený preh adaním celého priestoru riešení použitím distribuovaného preh adávanie stavového stromu a nevyhnutne aj osekávaním (pozri kapitolu 5 Teoretický základ pre riešenie hier). Ako už bolo spomenuté, systém bude písaný v jazyku C, preto je aj návrh systému tomu zna ne prispôsobený. Architektúra nebude klasická o sa týka notácie UML, ale prispôsobená danému problému a jazyku C. Na zachytenie modularity budú použité diagramy komponentov, kde komponent predstavuje zdrojový kód v jazyku C a hlavi kové súbory budú modelované ako rozhrania medzi komponentmi.. Asociácia v diagramoch typu používa (*use*), v zásade predstavuje deklaráciu zahrnutých modulov (deklarácia `#include`) v zdrojových súboroch. Pre názorný opis vybratej funk nosti bude použitý klasický sekven ný diagram UML, s

tým že jeho vykonávatelia sú komponenty alebo rozhrania realizované komponentom. Celý návrh vrátane jeho astí komponent arozhraní bude kvôli znovu použite nosti pomenovaný anglickými výrazmi, ktoré sú sú as ou projektového slovníka (pozriPríloha A – Slovník pojmov) a jednotlivé prvky diagramov sú presnejšie opísané v Príloha B – Použitá notácia diagramov.

Pod pojmom zoznam budeme rozumie lineárne jednosmerne zre azený zoznam, klasickú dátovú štruktúru. Prvý hrá je ozna enie pre hrá a, ktorý za ína partiu hry, a existuje práve jeden alší - druhý hrá . Rozhranie nenavrhované nami, inak povedané externé - definované v BOINC, bude na diagramoch farebne oddelené. Tieto pojmy budú alej bez odvolávania sa používané.

Spomínaná modularita má výhodu aj v existencii spolo ných astí pre klient a server, na strane servera spolo ných pre generátor a asimilátor. V nasledujúcich kapitolách budú tieto asti podrobne opísané.

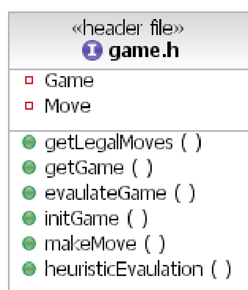
8.2 Návrh spolo ných astí

Prvky v tejto kapitole sú potrebné pre klienta a server, opisujú stavový priestor hry, iže aj musia by zhodné. Preh adávanie s využitím usekávania je naopak jedine né, to je popísané vkapitole 5 Teoretický základ pre riešenie hier. Prvky tejto, ale aj iných astí, budeme nazýva komponentmi. V prípade jazyka C sa vä šinou bude jedna o jedno rozhranie, ktoré predstavuje hlavi kový súbor *.h*, a jednu alebo viac implementácií - súbory *.c*. Globálne táto kapitola je návrhom, preto dôraz kladieme na popis rozhraní, nie konkrétnych algoritmov použitých v implementácii.

8.2.1 Komponent hra

Je zrejmé, že klient aj server budú potrebova štruktúry na uchovanie, a funkcie na manipuláciu so šachovnicou. Táto je pre každú hru iná, najmä o sa týka rozmeru, hracích figúrok (kame ov), ke že samotné pravidlá hry sú tiež špecifické pre tú ktorú hru, tie sme tiež zaradili do tohto komponentu.

Základnou jednotkou je rozhranie, pod ktorým sa budeme pozera na jednotlivú situácie v hre. Toto rozhranie je špecifikované súborom .h, obsahuje potrebné typy a funkcie. Na nasledujúcom obrázku (Obr. 27) je vidie zoznam týchto typov a funkcií, následne budú podrobnejšie popísané.



Obr. 27: Rozhranie hra - game.h

Štruktúry a premenné

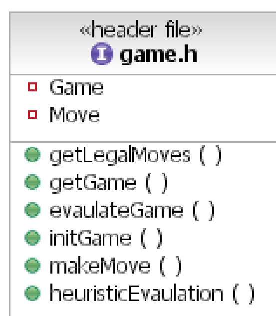
- `Game` - popisuje aktuálnu konfiguráciu hry, napr. pozície kame ov. Má presne definované rozmery a kódovanie pre jednotlivé možné stavy polí ka šachovnice (napr. kame prvého hrá a predstavuje íslo jedna).
- `Move` - popisuje možný ah hrá a ktorý je práve na ahu. V zásade obsahuje jednu alebo viac súradníc šachovnice z ktorými stavmi polí ok je vykonaná zmena v dôsledku ahu hrá a.

Funkcie

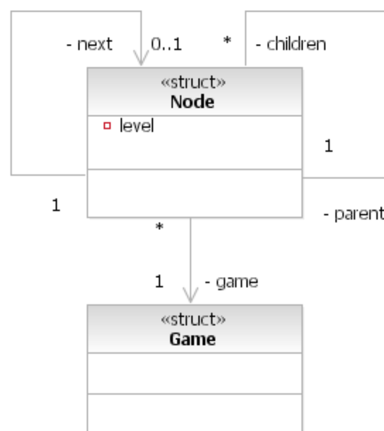
- `getLegalMoves(Game)` - vráti zoznam možných ahov pre konkrétnu hru pod a pravidiel zahrnutých v implementácii tohto rozhrania. Ke že sa nejedná o h adanie stratégie, ale (len) o h adanie ve mi ahkého riešenia, môžu by pravidlá hry takto oddelené od ostatných astí.
- `getGame(Game, Move)` - vráti novú inštanciu štruktúry hra, po spravení zadaného ahu na zadanej hre.
- `evaulateGame(Game)` - ohodnotí zadanú kone nú (neexistuje ah pre hrá a na rade) konfiguráciu hry jedným íslom, ktoré ur uje výsledok tejto hry. V našom prípade hrajú vždy dvaja hrá i, nula znamená remízu, kladné íslo ím vä šie tým vyššia výhra prvého hrá a, analogicky záporná hodnota.
- `initGame()` - vráti po iato ne nastavenú hru s "vynulovanou" šachovnicou.
- `makeMove(Move)` - podobne ako getGame(), len teraz vykoná zadaný ah na danej hre.
- `heuristicEvaulation()` - vráti ohodnotenie hry na základe špecifickej heuristiky definovanej pre konkrétnu hru.

8.2.2 Komponent Uzol

Preh adávaný priestor je graf, presnejšie strom. Možnos opakovania sa rovnakej konfigurácie v grafe bude riešená porovnávaním na strane servera, na strane klienta bude riešená len interne, alebo nebude riešená. Podrobnejšie túto záležitos opisuje kapitola 5 Transpozi ná tabu ka Graf je dvojica uzly a hrany. Budeme reprezentova uzol ako dátovú štruktúru, a hrany ako referencie uzla na iné. Nasledujúce obrázky popisujú rozhranie *node.h* (Obr. 28) a znázor uje prepojenie štruktúr typu Game a typu Node (Obr. 29).



Obr. 28: Rozhranie node.h



Obr. 29: Vz ah node so štruktúrou game

Štruktúry

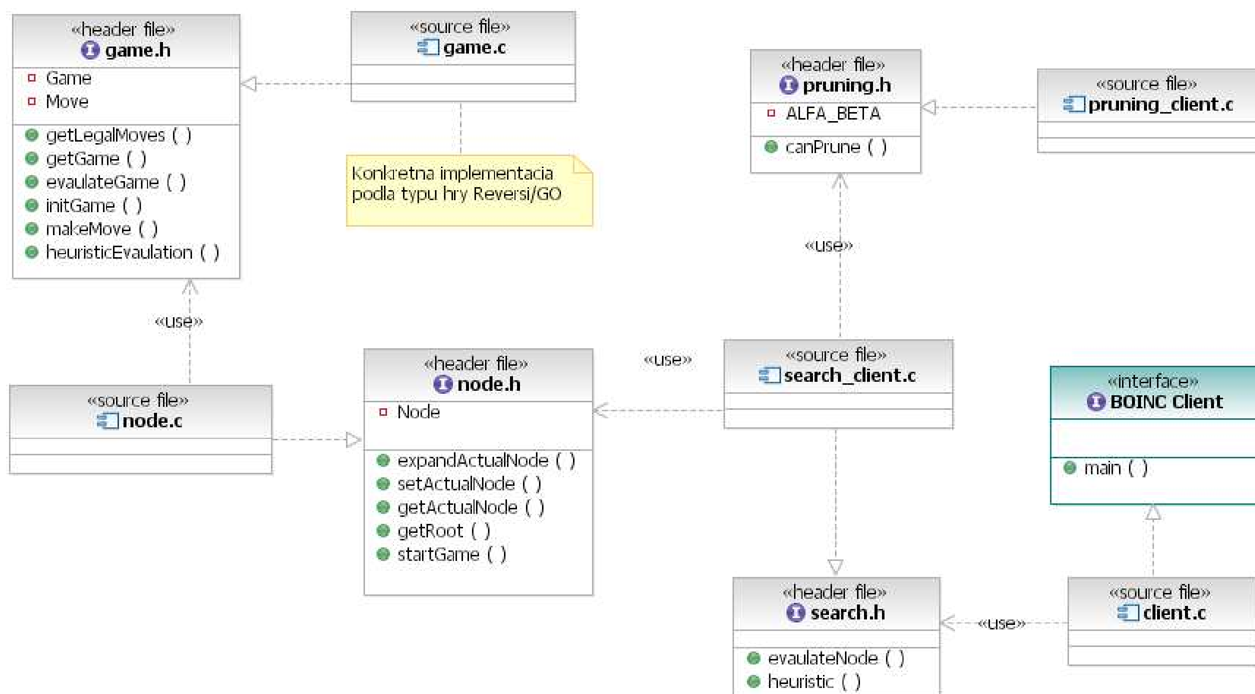
- `Node` - predstavuje uzol grafu, obsahuje referenciu na potomkov a na rodi a. Pre rýchle zistenie h bky v strome obsahuje aj tento atribút - level, z ktorého a vrátane aktuálnej konfigurácie hry, bude možné zisti , kto z hrá ov je na ahu.

Funkcie

- `expandActualNode()` - komponent implementujúci toto rozhranie obsahuje referenciu na aktuálny uzol, t.j. ukazuje na jeden uzol stromu. Ak tento uzol nebol expandovaný, nemá potomkov, po volaní tejto metódy bude. Sekven ný diagram pre túto operáciu sa nachádza na obr. 27.
- `setActualNode(Node)` - nastaví ako aktuálny uzol ten zadaný ako parameter.
- `getActualNode()` - vráti referenciu na aktuálny uzol, potrebné pre preh adávanie stromu.
- `getRoot()` - vráti referenciu na špeciálny uzol - kore stromu.
- `startGame()` - vytvorí kore ový uzol s po iato nou hernou konfiguráciou - inštanciou typu hra.

8.3 Návrh klienta

Úlohou klienta je ohodnoti as stromu použitím algoritmu *minimax* (kapitola 5 MiniMax) spolu s usekávaním. Jednoducho prejde as stromu a vráti jedno íslo, istý opis komplexného výsledku hry v danej asti stromu. Vstupom pre klienta, ako bolo povedané v úvode návrhu, je tzv. *workunit*, ktorý v podstate opíše vstupný uzol, ktorým má klient pri preh adávaní za a , t.j. berie ho ako kore stromu. Chceli sme docieli nezávislos na hre, na výbere algoritmu osekávania a na výbere heuristiky. Heuristika je sú as ou už opísaného rozhrania *game.h*. Architektúra klienta je zhrnutá na nasledujúcom obrázku (Obr. 30), ktorý obsahuje hlavné asti - komponenty a ich závislosti.

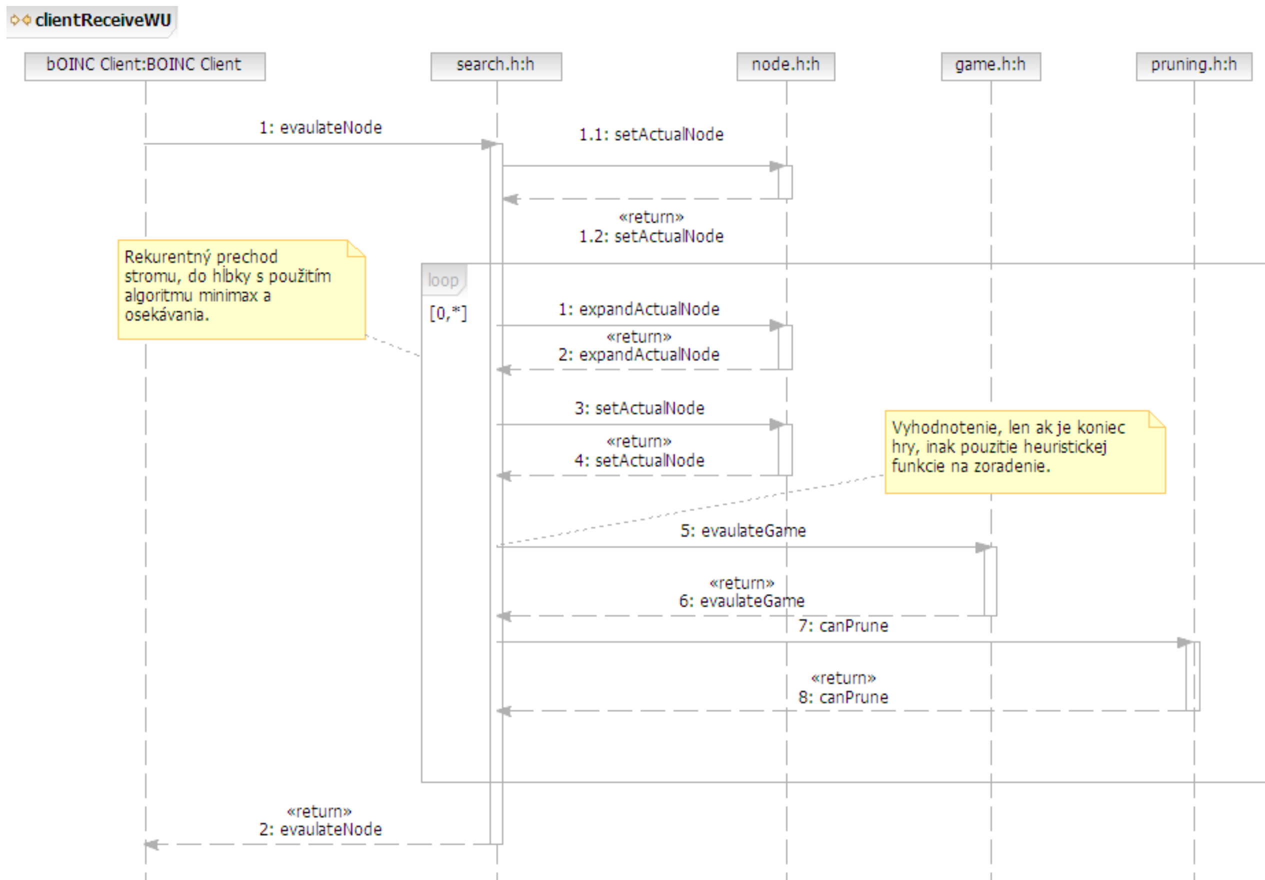


Obr. 30: Architektúra klientskej strany

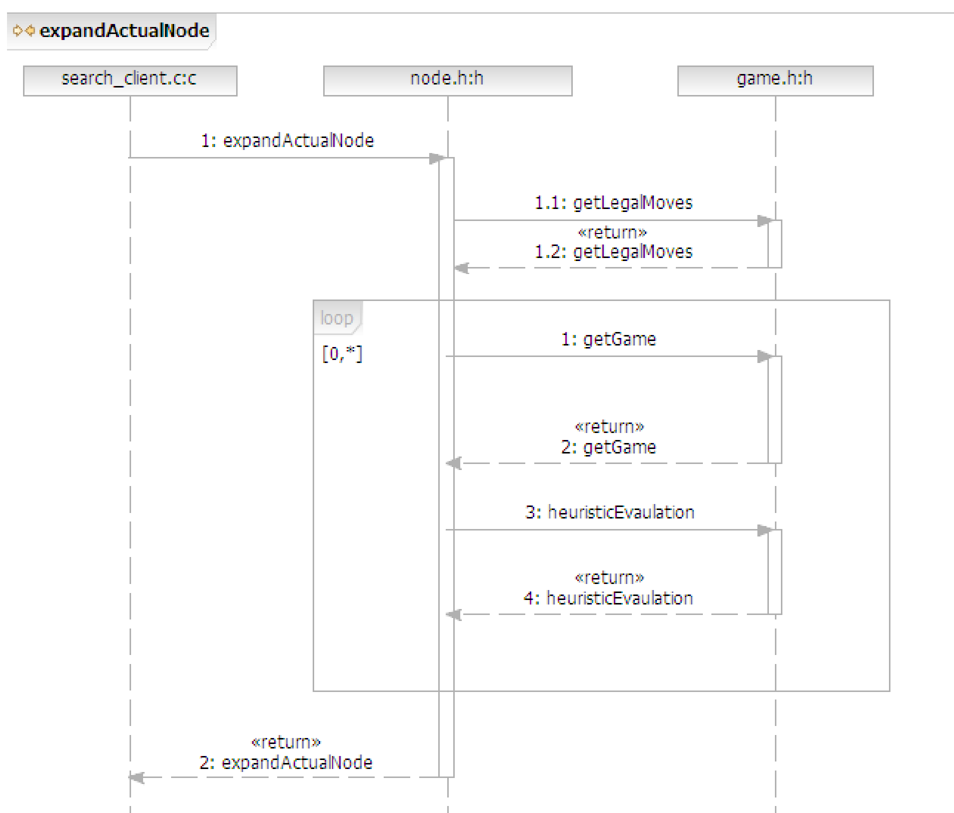
Rozhrania *game.h* a *node.h* boli už podrobnejšie popísané. Komponent **game.c** je konkrétna implementácia rozhrania *game.h* a popisu pravidla hry, v našom prípade Reversi a Go (pozri kapitolu 2.4). Podobne komponent node.c, ktorá implementuje manipuláciu s uzlami stromu a samotným stromom. Je zodpovedný za referencie na preh adávaný strom, najmä kore stromu. Popis alších astí klienta:

- *pruning.h* - rozhranie pre osekávanie prechodu stavovým stromom hry. Ako príklad uvedieme alfa-beta osekávanie, vtedy si implementácia tohto rozhrania uchováva hodnoty alfa a beta pre práve skúmaný uzol - vrchol stromu.
- **pruning_client.c** - konkrétna implementácia predchádzajúceho rozhrania, nemusí by striktne alfa-beta osekávanie.
- *search.h* - v podstate hlavné rozhranie pre klienta, t.j. pre komponentu client.c. Zapúzdruje operácie a atribúty potrebné pre preh adávanie stavového stromu. Je spúštané funkciou `evaulateNode(Node)`, ktorá má vstupný parameter uzol ur ený na vyhodnotenie klientom. Vracia íselnú hodnotu pre výsledok hry za ínajúcej konfiguráciou, ktorú obsahuje tento uzol.
- **search_client.c** - konkrétna implementácia predchádzajúceho rozhrania, vyžaduje popis stromu a jeho inštancii (node.h), akým spôsobom bude osekáva nepotrebné vetvy (pruning.h).
- **client.c** - vykonate ný klientský kód u klienta, stru ne povedané, tu sa všetko pre klienta za ína a kon í, od prijatia *workunit* (po iato ného uzla) po odoslanie výsledku (ohodnotenie uzla). Tento komponent sa podria uje pravidlám BOINC-u, na diagrame je to znázornené implementáciou externého rozhrania. Sekven ný diagram (Obr. 31) opisuje udalosti pri

prijatí *workunit*-u klientom. as ou tohto diagramu je funkcia `expandActualNode()`, ktorá nieje triviálna a jej bližší popis poskytuje sekven ný diagram (Obr. 32).



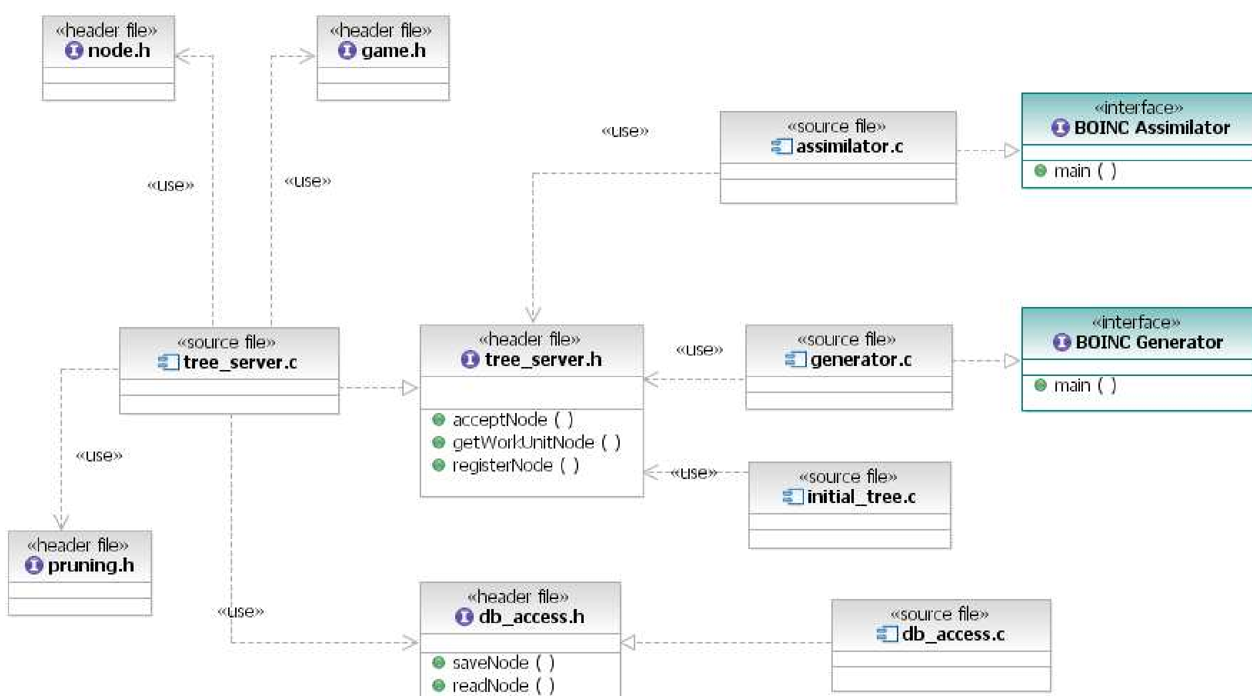
Obr. 31: Sekven ný diagram popisujúci prijatie práce (workunit) klientom



Obr. 32: Sekven ný diagram pre expandovanie jedného uzla

8.4 Návrh server

Ako už bolo spomenuté na za iatku návrhu, server je delený na dve hlavné asti v závislosti od architektúry BOINC-u: generátor a asimilátor. Toto je na Obr. 33 znázornené tým, že tieto dva komponenty implementujú isté rozhrania, ktoré sú definované mimo nášho systému.

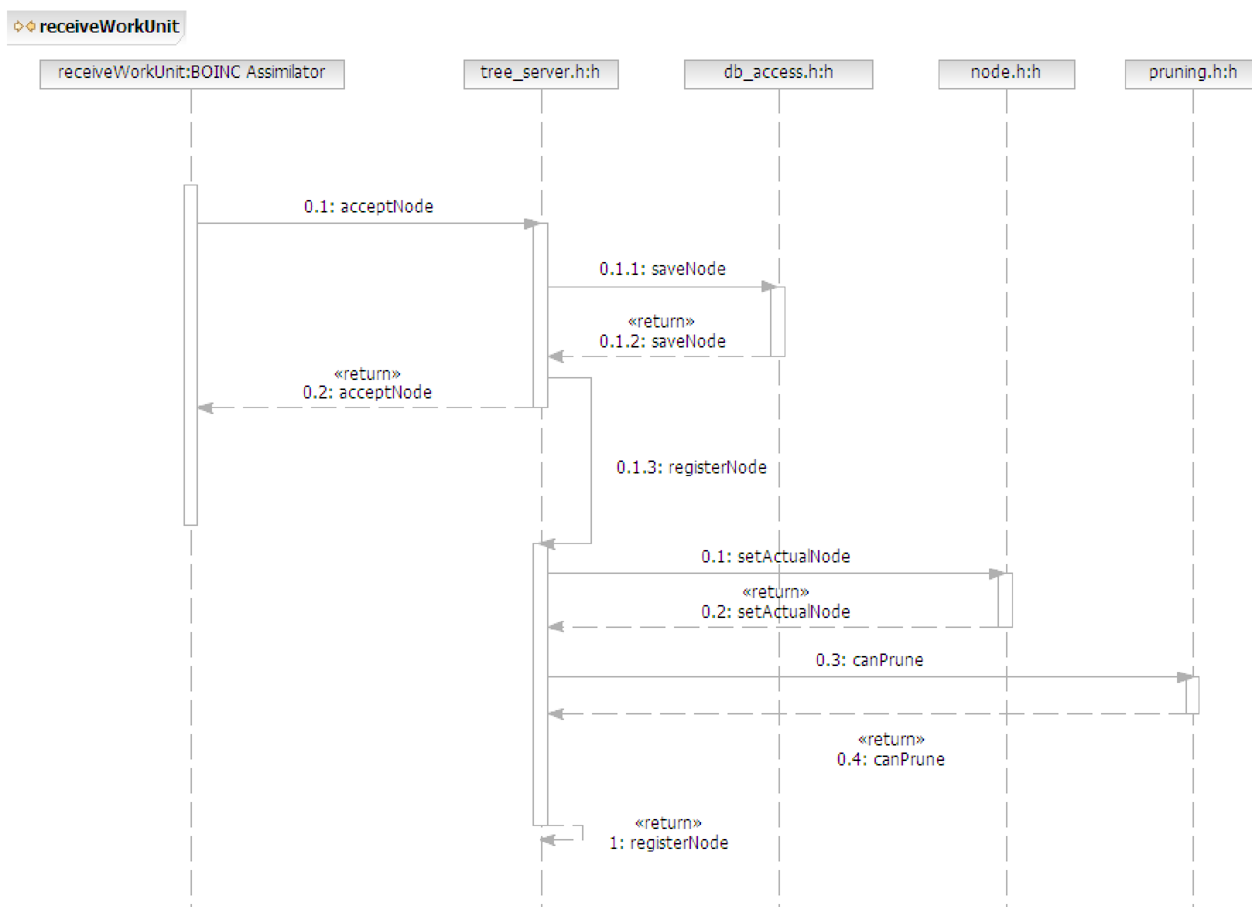


Obr. 33: Diagram komponentov pre server

Nasleduje popis jednotlivých astí servera. Rozhrania na komunikáciu s klientom node.h a game.h boli opísané v predchádzajúcich astiach. alšie asti servera:

- *tree_server.h* - komplexné rozhranie pre celý server, popisujúce preh adávaný strom. K jeho zodpovednosti patrí prijatie ohodnoteného uzla od klienta (`acceptNode()`) a jeho zaregistrovanie (`registerNode()`), o je lokalizácia v strome, uloženie, prípadne spustenie usekávania. Heuristika na vyžiadanie generátora vráti klientovi pod a preferencií uzol na preh adanie, inak povedané, hlavnú as *workunit*-u (`getWorkUnitNode()`).
- **tree_server.c** - implementácia rozhrania *tree_server.h*, dá sa poveda , že táto komponenta je akési srdce celého servera. Uchováva informácie o práve skúmaných uzloch, vie, ktorá as celého stavového priestoru bola preskúmaná a ohodnotená. Registruje ohodnotené uzly od klientov, to môže spôsobi osekávanie istej asti stromu, z toho dôvodu má k dispozícii rozhranie *pruning.h*.
- *pruning.h* - rozhranie osekávania, ktoré môže, ale nemusí, ma takú istú implementáciu ako klient.
- *db_access.h* - rozhranie pre prístup k disku, konkrétne k databáze uložených uzlov. Poskytuje funkcie na ítanie a zápis uzla do databázy (`saveNode()`, `readNode()`). Týmto je náš systém nezávislý od použitej databázy, resp. od spôsobu ukladania uzlov, ktoré sa už nezmestia do opera nej pamäti.
- **db_access.c** - konkrétna implementácia posledne spomenutého rozhrania, volá CRUD (Create, Read, Update, Delete, pozri Príloha A – Slovník pojmov), príkazy nad použitou, nasadenou databázou.
- **initial_tree.c** - jednoduchá a priamo iara as , pred spustením samotnej BOINC komunikácie klient-server je vygenerovaná istá as stromu, ktorej koncové uzly sú posielané klientom.
- **generator.c** - hlavný zdrojový kód pre generátor, ktorý je definovaný systémom BOINC. Generuje *workunit*-y, ktorých uzly si pýta od rozhrania *tree_server.h*.

- **assimilator.c** - analogicky ako generátor, ale prijíma výsledky od jednotlivých klientov a tieto ohodnotené uzly registruje cez rozhranie *tree_server.h*. Posledné oba komponenty sú presnejšie špecifikované preh ade systému BOINC (kapitola 3 Démoni bežiaci na serveri). Názorný popis prijatia výsledku od klienta je na Obr. 34.



Obr. 34: Sekven ný diagram popisujúci prijatie výsledku (result) spracovaného workunit-u od klienta a jeho spracovanie

9 Prototyp

Rozhodli sme sa vytvori prototypy v dvoch zatia oddelených astiach:

- Prototyp preh adávania stavového priestoru na jednom stroji zatia iba pre hru *reversi*, pre získanie informácie o ve kosti stavového priestoru o závislosti faktoru vetvenia od h bky v strome, t.j. od ísla ahu hry.
- Prototyp pre systém BOINC, triviálny program pre otestovanie funk nosti a získanie informácií o vytváraní programu s paralelným po ítaním.

Preto je náš prototyp odlišný od klasického prototypu informa ného systému, sú to v podstate dva oddelené programy. Výsledný systém bude predstavova spojenie týchto astí.

9.1 Prototyp hry na jeden PC

Tento prototyp bol vytvorený ako program v jazyku C v prostredí Eclipse (presnejšie Eclipse C/C++ Development Tools[3]) skompilovaný pre platformu Windows kompilátorom MinGW[4], preto bude odteraz spomínaný ako program. Bol implementovaný na základe už opísaného návrhu, aby to mohol by základ pre alšie programovanie samotnej aplikácie.

Najprv bol implementovaný mechanizmus pre definovanie hry (v našom prípade *reversi*), to spo ívalo v naprogramovaní implementácie rozhrania *game.h*, následne komponentu pre prácu so stavovým stromom (implementácia *node.h*). Bolo spravených viacej pokusov nad týmto stromom, tu budú spomenuté dve najdôležitejšie: celý stromu do istej h bky, odhad faktoru vetvenia.

9.1.1 Celý strom do istej h bky

Pokus o vygenerovanie o najhlbšieho stromu na jednom po íta i v rozumnom ase (tým je myslené nieko ko minút). Toto bolo dosiahnuté do h bky devä s výsledkami, ktoré popisuje Tab. 4. Obsahuje po et uzlov v jednotlivej h bke stromu a k tomu dopo ítaný faktor vetvenia v danej h bke ako pomer po tu uzlov v susedných h bkach.

Ako vidie , v h bke devä je potrebné uchova si takmer tri a pol milióna uzlov, o pre klasický po íta je relatívne dos . Cie om tejto asti bolo najmä odhadnú rozsah serverom uchovávanej asti stromu. Rátame aj s tým, že v nasadenej aplikácii ešte minimalizujeme množstvo pamäti pre uchovanie jedného uzla.

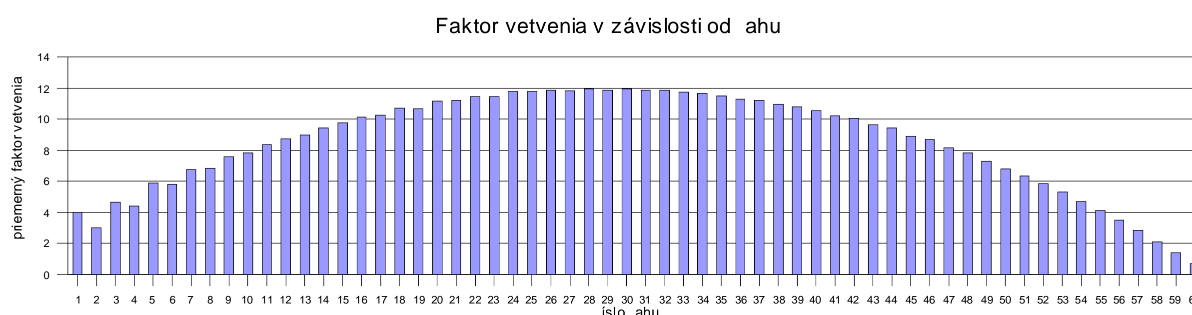
3 http://www.eclipse.org/cdt/
4 http://www.mingw.org/

H bka	Po et uzlov	Faktor vetvenia
1	4	4
2	12	3
3	56	4,67
4	244	4,36
5	1396	5,72
6	8200	5,87
7	55092	6,72
8	390216	7,08
9	3005264	7,7

Tab. 4: Skúmanie faktoru vetvenia, v dosiahnute ných h bkach

9.1.2 Odhad faktoru vetvenia

Druhý experiment a cie prototypu bolo získa faktor vetvenia pre každú h bku, a tým nepriamo odhad ve kosti celého stavového priestoru pre hru Reversi. Toto bolo riešené náhodným prechodom stavového priestoru až do najnižšej h bky - koniec hry. Po nieko konásobnom zopakovaní (minimálne tisíc krát) bol vyrátaný priemerný faktor vetvenia pre každú h bku, t.j. každý ah. Výsledky tohto pokusu sú znázornené na Obr. 35



Obr. 35: Graf závislosti faktoru vetvenia od ísla ahu

Ako vidie z grafu, faktor vetvenia sa najprv zvyšuje, maximum dosahuje asi v strede hry, pri ahu okolo tridsa . Tu ešte chceme poukáza na faktor vetvenia na konci hry, je menší ako jeden, o je logicky podložené tým, že ku koncu môže nasta situácia, že hrá na ahu nemá pod a pravidiel možný ah. alším poznatkom z tohto pokusu je odhad ve kosti stavového priestoru hry Reversi, zrátaním sumy pre všetky h bky použitím zistených faktorov vetvenia sme dospeli k íslu $2,27 x 10^{53}$, o je dos ve ké íslo, ale bolo o akávané.

9.2 Prototyp aplikácie pre BOINC

Platformu BOINC si bolo potrebné pred samotnými úvahami o spôsobe riešenia úlohy „ohmata ". Na našom serveri sme teda okrem podporných nástrojov ako dotProject nainštalovali i funk ný BOINC server a vytvorili si projekt test1. Projekt bežal bez incidentov, bolo sa možné k nemu prihlási , ale neobsahoval žiadnu klientsku aplikáciu a ani žiadne úlohy. Bolo potrebné sa s touto platformou bližšie zoznámi a preto sme si v rámci analýzy vyskúšali skompilova a zverejni klientsku aplikáciu. Pokusnou aplikáciou bola ukážková aplikácia „uppercase“, ktorej úlohou bolo prija ako úlohu akýko vek textový vstup a transformova ho na výstup s pôvodným znením, avšak

so všetkými znakmi „ve kými“. Táto ukážková aplikácia pre svoje spomalenie po ítala po každom transformovanom znaku sumu zo sínus pí, krát íslo iterácie v cykle s miliónom krokov. alej táto aplikácia používala „checkpointing“ (pozri Príloha A – Slovník pojmov) a previazanie na grafickú as . Aplikáciu sa nám podarilo úspešne skompilova i zverejni na BOINC projektovom serveri. Funk nos sme overili pripojením dvoch BOINC klientov a vypo ítaním nieko ko desiatok úloh. Body sa pripo ítavali po úspešnej validácii a výsledky sa vymazávali správne.

9.3 Prototyp BOINC projektu

Pôvodne sme uvažovali, že sa pri vytváraní prototypu zameriame na zistenie:

- ktorý z algoritmov preberaných v asti 5 Teoretický základ pre riešenie hier sa na klientovi použije
- na základe h bky stromu, ktorý klient dokáže preh ada , ktorý mechanizmus bude potrebné použi pri generovaní stromu h adania na serveri a vytváraní úloh pre klientov.

Do úvahy prichádzajú nasledovné prístupy generovania stromu na serveri:

- vygenerova na za iatku strom na serveri do ur itej pevne danej h bky
- dynamicky generova strom , nevygenerova celú šírku stromu na každej h bke, ale zís do stromu hlbšie, aby klientom ostal zvládnute ný menší strom h adania

Nakoniec, po konzultácií s vedúcim projektu, sme sa však rozhodli, že prototyp bude predstavova zjednodušenú verziu celého systému riešenia a nebudeme sa sústredi na jednú špecifickú as . Z toho sme stanovili, že cie om prototypu je vytvori BOINC projekt, ktorý bude schopný vyrieši distribuovaným spôsobom hru reversi. Prototyp má takto overi principiálnu funk nos navrhnutého prístupu. Ke že sa v tejto asti nebudeme sústredi na žiadne optimalizácie ani pokro ilé algoritmy, systém bude pod a odhadov schopný vyrieši hru reversi do rozmerov 4x4. Riešenie na rtnutých otázok bude presunuté do nasledujúceho semestra.

V rámci prototypu vytvárame nasledovné asti tak, ako sú popísané v kapitole 3 BOINC:

- generátor úloh
- asimilátor
- klient

9.3.1 Návrh databázy

Na základe analýzy (7 Možnosti ukladania stromu na disk) sa ur ila na ukladanie dát databáza. Sú v nej uložené iba údaje priamo sa týkajúce riešenej úlohy. Réžijné údaje spojené s mechanizmom rozde ovania úloh klientom si uchováva systém BOINC vo svojej inernej databáze.

Dáta, ktoré v databáze uchovávame sú:

- konfigurácia hracia plocha
- konfigurácie hracích plôch, do ktorých sa dá z danej kofigurácie dosta
- ohodnotenie konfigurácie

- ktorý hrá je v danom ahu na rade

Nami zvolené kódovanie hernej pozície je takéto:

Každé polí ko môže ma 3 rôzne ohodnotenia:

- kame bieleho hrá a (01)
- kame ierneho hrá a (00)
- prázdne polí ko (10)

Teda potrebujeme 3 rôzne hodnoty pre každé polí ko, pre jednoduchos sme zvolili nie najefektívnejšie kódovanie dvoma bitmi (jeden bit je nevyužitý)

Ak je rozmer šachovnice n, polí ok je $n^{2,}$ a potrebujeme n x n x 2 bitov, pre konkrétny rozmer šachovnice 8 potrebujeme 128 bitov na zakódovanie stavu

Pole informujúce o hrá ovi, ktorý je na rade, môže ma 2 rôzne hodnoty, preto na jeho kódovanie používame typ boolean.

Pole s informáciou o ohodnotení aktuálneho stavu môže ma 4 rôzne hodnoty:

- vyhral biely (10)
- vyhral ierny (01)
- remíza (00)
- neohodnotené (11)

Na kódovanie ohodnotenia preto používame 2 bity.

Fyzický model databázy

Obr. 36: Fyzický model databázy

Primárnym k ú om sú polia id a move. Pole id samo o sebe ako primárny k ú nesta í, dve rovnaké pozície na hernej ploche sa totiž od seba môžu líši tým, ktorý hrá je na ahu.

Ke že potrebujeme vyh adáva v DB pod a aktuálnej pozície a pod a rodi ovskej pozície, tieto dve polia sú indexované.

Databázu tvorí jediná tabu ka reversi.

Použité technológie

Používame databázový server PostgreSQL 8.1.

Modul komunikácie s databázou

Na komunikáciu s databázou na serverovej strane systému sme implementovali prístupové rozhranie v jazyku C pomocou knižnice libpq[5]. Ide o aplika né programátorské rozhranie pre aplikácie napísané v jazyku C, ktoré je priamo sú as ou databázy PostreSQL. API poskytuje klientským programom funkcie na poslanie databázových dotazov serveru PostgreSQL a získanie výsledkov z týchto dotazov.

Celý modul komunikácie s databázou je oddelený a nezávislý. Používa funkcie knižnice liqpq na ukladanie uzlov stromu hry do databázy, uloženie ohodnotenia uzla, na ítanie uzla z databázy a na ítanie potomkov uzla z databázy.

9.3.2 Klient

Pod klientom sa v tejto asti myslí klientska aplikácia. Teda aplikácia, ktorá je za pomoci BOINC systému distribuovaná klientom, ktorí ju následne spúš ajú a vracajú jej výsledky na server. Implementácia klienta spo ívala prakticky vupravení prvého prototypu, ktorý po ítal priemerný faktor vetvenia tak, aby po ítal celú hru a nie iba priemerný faktor vetvenia. Po tejto úprave bolo nutné doplni volania BOINC klientskej knižnice, ktoré oznamujú samotnému klientovi stav úlohy a iné potrebné informácie. Posledným krokom bolo skompilovanie samotného klienta a jeho zlinkovanie s BOINC klientskou knižnicou a našimi knižnicami. Popis podstatných astí knižníc je v nasledujúcich podkapitolách.

5 http://www.postgresql.org/docs/8.1/static/libpq.html

Algoritmus riešenia hry

Tak, ako bolo spomenuté, klientska as prototypu sa nesústredí na žiadne optimalizácie ani pokro ilé algoritmy a techniky ohodnocovania stromu h adania symetrickej hry. Za algoritmus na strane klienta sme preto zvolili jednoduchý a ahko implementovate ný MiniMax algoritmus. Použitie tohto algoritmu automaticky obmedzilo triedu riešite ných problémov na problémy s malým stavovým priestorom. Na túto skuto nos je potrebné pri testovaní prototypu myslie , nako ko MiniMax algoritmus musí navštívi každý uzol stromu preh adávania.

Vstupné a výstupné súbory

Komunikácia BOINC klienta so serverom je realizované prostredníctvom výmeny vstupných a výstupných súborov. O samotný prenos tých súborov sa pri implementácii klienta stara nemusíme, zabezpe uje ju BOINC mechanizmus. Štruktúru súborov sme stanovili nasledovne:

- vstupný súbor – obsahuje stav pre ktorý sa bude po íta ohodnotenie a druhá hodnota je ktorý hrá je na ahu. Je generovaný generátorom úloh a je vstupom pre po ítanie výsledku na klientovi.
- výstupný súbor – obsahuje údaj pre ktorú úlohu sa daný výsledok vypo ítal a výsledok. Je to výstup od klienta, ktorý sa následne spracúva na serveri.

Jednotlivé položky sú v oboch súboroch od seba oddelené znakom „ ; “.

9.3.3 Generátor úloh

Generovanie úloh je možné viacerými spôsobmi. Ako prvý spôsob sme využili „ru né“ zverejnenie súboru. To sa prevádza za pomoci skriptov. Jednoduchá ukážka je tu:

```
cp sample_wus/inReversi6x6 `bin/dir_hier_path inReversi6x6`


./bin/create_work -rsc_fpops_bound 7e11 -appname reversi -wu_name
pokusnyWorkunit -wu_template templates/reversi6_wu
-result_template templates/reversi6_result inReversi6x6
```

Prvý príkaz zabezpe í nakopírovanie vstupného súboru na správne miesto do adresárovej štruktúry a druhý príkaz vykoná samotné zverejnenie súboru spolu s vytvorením úlohy. Vytvorenie úlohy znamená, že sa úloha zaregistruje v BOINC databáze a je pripravená na odoslanie klientom. Dôležitým parametrom je rsc_fpopos_bound. Udáva maximálny po et flopov (flops) potrebných pre výpo et daného workunitu. Ak je tento parameter nastavený na malé íslo, tak výpo et skon í hláškou o prekro ení asového limitu. Výpo et vhodného horného ohrani enia si ponechávame na letný semester.

Druhý spôsob generovania úloh je vygenerova ich programovo, skopírova ich na vhodné miesto a zaregistrova . Táto programová metóda bola implementovaná, ale zatia nebola využívaná. Bolo potrebné pouči nieko ko BOINC knižníc a ich volania. Pri generovaní úloh si ukladáme celý stavový priestor hry do databázy, až po úrove , naktorej generujeme z listov stromu úlohy. Teda preh adávame stavový priestor hry do h bky a ukladáme si do našej databázy všetky vygenerované stavy. V prípade, že h bka aktuálneho uzla je rovná íslu 3, tak daný uzol uložíme nadisk ako úlohu a nevykonáme generovanie nasledovníkov.

Prototypová verzia má zatia obmedzenia, ktoré však neskôr odstránime. Zvažuje sa však použitie tretej možnosti generovania úloh.

Tretím spôsobom je generovanie úloh (workunit-ov) na požiadanie. Teda, ke klesne po et neodoslaných úloh pod nejakú hranicu, „dogeneruje“ sa daný po et úloh na požiadanie. Tento spôsob zatia neplánujeme využi .

9.3.4 Asimilátor

Asimilátor je jednou z dvoch hlavných sú astí servera. V prototype sme sa sústredili na vytvorenie jednoduchého asimilátora, ktorý bude uklada prijaté výsledky od klientov do databázy. Implementovanie asimilátora spo íva v napísaní jednej konkrétnej funkcie, ktorá je BOINC-om volaná pri obdržaní výsledku. Hlavi ka funkcie má tvar:

```
int assimilate_handler(WORKUNIT& wu, vector<RESULT>& results, RESULT&
      canonical_result)
```

Kde typy WORKUNIT a RESULT sú sú as ou BOINC knižnice. Pre nás je dôležité, že RESULT obsahuje referenciu (cestu) k súboru, ktorý bol vytvorený klientom a obsahuje výsledok. Po na ítaní hry (game) z tohto súbora je hra nájdená v databáze. Ak je výsledok relevantný, tak v databáze musí by záznam o tejto hre. Následne je vytvorený uzol (node), ktorý je akceptovaný funkciou `acceptNode(Node)` a príslušný záznam v databáze je aktualizovaný o výsledné ohodnotenie tohto uzla.

Cie om prototypu bolo iba otestova funk nos a napojenie na databázu. Výsledný produkt bude samozrejme v spomínanej asimila nej funkcii obsahova viac logiky, napr. vyhodnotenie uzla aj v zmysle možného osekávania susedných uzlov.

10 Zmeny oproti návrhu

Zmeny globálnej architektúry sa nevyskytli, samozrejme aj preto, lebo táto architektúra vychádza z prostredia BOINC a bolo by priam zbyto né ju modifikova . Po as implementácie sa samozrejme vyskytli situácie, kedy bolo treba modifikova funkciu, prípadne prida novú funkciu. Práve tieto situácie a ich dopady na pôvodný návrh budú obsahom nasledujúcich astí. Identifikovali sme tri komplexnejšie zmeny, ktoré sa týkali návrhu ako celku:

- za ali sme používa kompilátor C++, preto sú odteraz názvy komponent s príponou cpp.
- prešli sme od "pseudo objektovej štruktúry" na procedurálnu štruktúru t.j. neuchovávali sme stav v globálnych premenných ale pridali sme do stavových funkcií parameter a sú asne návratovú hodnotu rovnakého typu. iže jednotlivé operácie boli potom vykonávané cez znovu priradenie a nie cez globálnu premennú (napr. v *node.h*: expandActualNode() -> Node *expandNode(Node *)).
- za ali sme používa knižnicu STL na spracovanie kolekcií (zoznam, tabu a, ..) o zmenilo niektoré typy parametrov vo funkciách.
- všetky súbory spomenuté v návrhu alebo tejto implementácii boli rozdelené do dvoch adresárov v adresárovej štruktúre, kvôli použite nosti a jednoduchšej kompilácii. V adresári **cockroach** sa nachádzala vä šina súborov spolo ných pre klient aj server. Druhým adresárom bol adresár **sched**, ktorý je sú as ou boinc-u a do neho boli umiestnené hlavné asti serverovskej strany: generátor a asimilátor.

Kvôli tejto asti sme zaviedli upravenú notáciu diagramov s nasledovnými zmenami:

- pridané komponenty budú podfarbené na oranžovo
- zrušené asti na sfarbené na erveno
- modifikované funkcie (premenované, pridaný parameter) s prefixom <<*mod*>>
- pridané funkcie s prefixom <<*add*>>

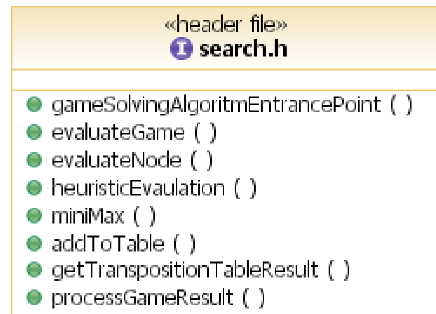
Opis každej komponenty bude obsahova podkapitoly s názvami zmenené funkcie a pridané funkcie, ich význam je zrejmý z ich názvu. o sa týka zrušených funkcií, tie podrobne nerozoberieme, lebo boli bu nahradené novou funkciou, ktorá bude popísaná alebo stratili význam.

Podobne ako v návrhu (8Návrh systému) budú alšie podkapitoly rozdelené do troch astí: Spolo né asti, Klient a Server.

10.1 Spolo né asti

10.1.1 Komponenta search

Medzi spolo né asti pribudlo alšie rozhranie *search.h*, následne aj jeho implementácia **search.cpp**.



Obr. 37: search

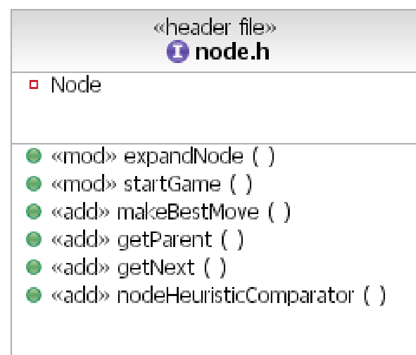
Toto rozhranie vzniklo spojením rovnomenného rozhrania z klientskej strany a rozhrania *tree_server.h* zo serverovskej strany, preto sa uvádza v tejto spolo nej asti.

Pridané funkcie

- gameSolvingAlgoritmEntrancePoint() - predstavuje spolo né rozhranie pre všetky testované algoritmy osekávania.
- evaluateGame() - ohodnotenie listu na konci, iže neexistuje alší ah, výsledkom je i vyhral prvý alebo druhý hrá alebo je remíza.
- miniMax() - vstupný bod pre klasický algoritmus minimax.
- addToTable() - pridá výsledok pre daný uzol do transpozi nej tabu ky.
- getTranspositionTableResult() - získa výsledok uložený v transpozi nej tabu ke pre danú hraciu plochu.
- processGameResult() - serverovská funkcia, je volaná po obdržaní výsledku od klienta, ktorý je následne zaznamenaný do našej databázy a spracovaný vyššie (rekurzívne volaná funkcia).

10.1.2 Komponenta uzol

alšími spolo nými as ami sú rozhranie *node.h* a *game.h*, nasledujúce riadky popisujú zmeny v týchto dvoch komponentoch. Najprv to bude rozhranie *node.h*:



Obr. 38: node

Zmenené funkcie

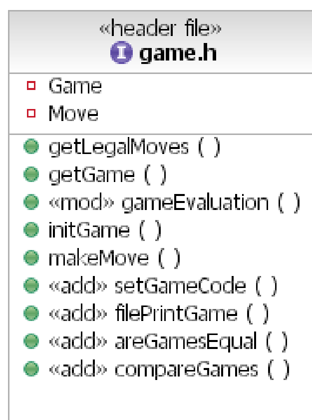
- expandNode(*Node) - zmena na parametrickú funkciu s parametrom *Node.
- startGame(*Game) - premenovaná z pôvodne navrhnutej funkcie initGame().

Pridané funkcie

- makeBestMove(*Node) - vykoná sa najlepší ah v danej situácii a funkcia vráti nový uzol.
- getParent(*Node) - vráti rodi a uzla, ktorý je vstupným parametrom.
- getNext() - vráti bezprostredne najbližšieho suseda uzla, ktorý je vstupným parametrom.
- nodeHeuristicComaprator(*Node, *Node) - porovná dva vstupné uzly na základe implementovanej heuristiky. Vráti jedno z ísel: -1, 0, 1.

10.1.3 Komponenta hra

V tejto komponente boli viacmenej len pridané pomocné funkcie, ktoré vyplynuli z implementa ných požiadaviek.



Obr. 39: game

Zmenené funkcie

- gameEvaluation(*Game) - vráti ohodnotenie šachovnice, v zmysle kto vyhral

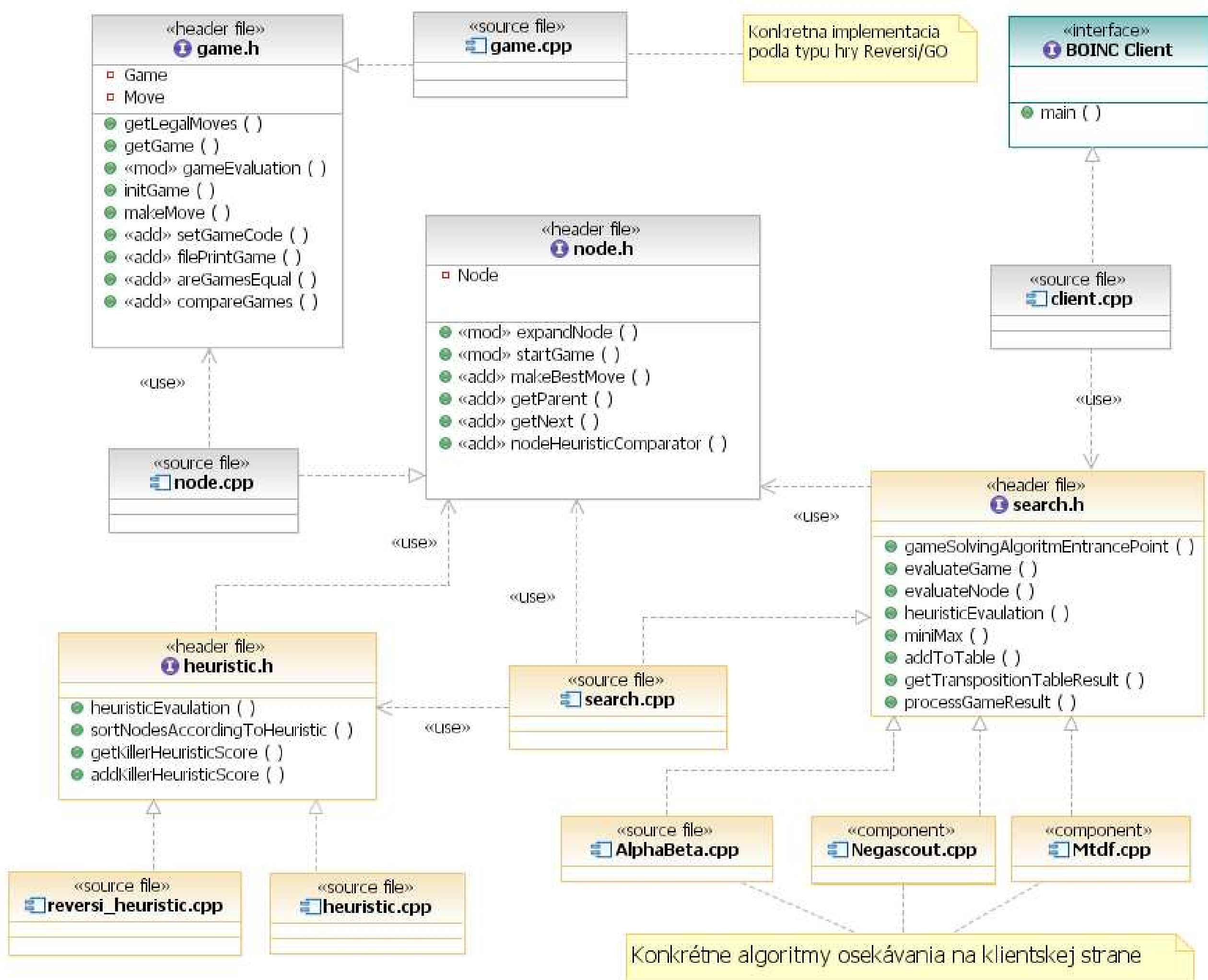
Pridané funkcie

- setGameCode(*Game) - nastaví hracej ploche kód - bitová reprezentácie hracej plochy.

- filePrintGame(*File, *Game) - zapíše zadnú hru do zadaného súboru pod a stanoveného formátu, ktorý je popísaný v technickej dokumentácii.
- areGamesEqual(*Game, *Game) - zistí, i sú zadané hracie plochy rovnaké, neberie sa do úvahy, kto je na ahu.
- compareGames(*Game, *Game) - porovná vstupné hry, vráti jednu hodnotu z -1, 0, 1. Táto funkcia už berie do úvahy aj to, kto je na ahu.

10.2 Klient

Výsledná štruktúra jednotlivých komponent na strane klienta je znázornená na nasledujúcom diagrame. V alšom texte budú zmeny opísané podrobnejšie.



Obr. 40: Klient

Pribudlo nové rozhranie *heuristic.h*, ktoré je používané komponentov **search.cpp**, ktorá pri preh adávaní stromu uzlov používa heuristické funkcie na rozhodovanie. V dôsledku vzniku tohto rozhrania pribudli aj prislúchajúce implementácie.

Vznik komponenty search.cpp je logický a vyplynul zo vzniku rozhrania *search.h*. Komponenty **AlphaBeta.cpp**, **Nedascout.cpp** a **Mtdf.cpp** sú konkrétne implementácie rovnomenných algoritmov osekávania, ktoré sú popísané v asti analýza (5Teoretický základ pre riešenie hier). Všetky tieto usekávania sú prístupné cez jedno rozhranie *search.h*, presnejšie cez jednu jeho funkciu gameSolvingAlgoritmEntrancePoint().

10.2.1 Heuristická komponeta

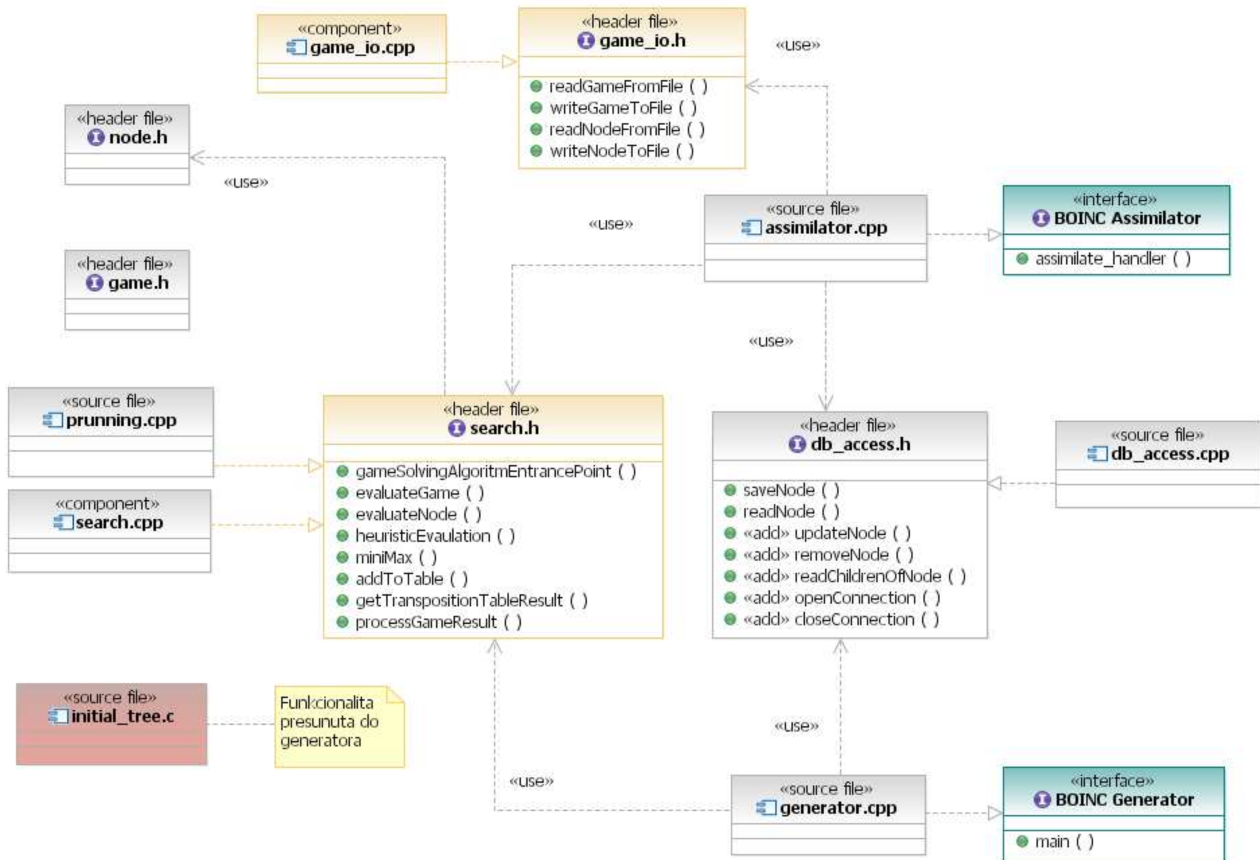
V tejto asti popíšeme pridanú komponentu *heuristic.h*:

Pridané funkcie

- sortNodesAccordingToHeuristic(*Node) - usporiada uzly pod a heuristickej funkcie, využíva sa zre azenie cez atribút Node::next.
- heuristicEvaluation(*Node) - vráti heuristické ohodnotenie vstupného uzla.
- getKillerHeuristicScore(*Game) - získa skóre pre hraciu plochu podla heuristiky vražedného ahu.
- addKillerHeuristicScore(*Game, int) - nastaví skóre uzlu pod a heuristiky vražedného ahu.

10.3 Server

Výsledná štruktúra jednotlivých komponent na strane servera je znázornená na nasledujúcom diagrame. V alšom texte budú zmeny opísané podrobnejšie.



Obr. 42: Server

Na serveri boli zmeny menšie oproti zmenám na klientovi. V podstate sa len premenovalo už spomenuté rozhranie *search.h* (vzniklo z rozhrania *tree_server.h*). Bola odstránená komponenta **initial_tree.c** z dôvodu presunutia jej funkcionality do komponenty **generator.cpp**. Pre potreby vstupno-výstupných operácii na disku bola vytvorená samostatná komponenta **game_io.cpp** a k tomu prislúchajúce rozhranie *game_io.h*. alej na strane servera boli pridané funkcie do rozhrania *db_access.h*. Toto bolo už pri návrhu o akávané, lebo v tom ase sme toto rozhranie definovali ako prístupový bod k databáze a jeho funkcie sme deklarovali len približne a predbežne.

10.3.1 Vstupno-výstupná komponenta

Táto komponenta (*game_io*) predstavuje rozhranie resp. implementáciu zápisu na disk. Je potrebná v troch hlavných krokoch ako:

- funkcia zápisu WorkUnit-ov pripravených pre klientov.
- funkcia ítania výsledkov, vrátených od klienta.

- na ítanie vstupnej konfigurácie pre generátor.

Pridané funkcie

- readGameFromFile(*File) - vráti údajovú štruktúru typu Game, naplnenú vstupnými údajmi zo zadaného súboru.
- writeGameToFile(*File, *Game) - zapíše zadanú štruktúru Game (hraciu plochu) do zadaného súboru v presne definovanej štruktúre zápisu.
- readNodeFromFile(*File) - pre íta stav hracej plochy zo zadaného súboru a vytvorí štruktúru typu Game, ktorá je sú as ou návratovej hodnoty typu Node. Používané asimilátorom pri príchode výsledku od klienta.
- writeNodeToFile(*File, *Node) - zapíše zadanú štruktúru typu Node do zadaného súboru, používané klientom pri zápise výsledku.

10.3.2 Databázová komponenta

Ako už bolo spomenuté táto komponenta (*db_access*) prešla na prvý poh ad vä šou zmenou, ale to sme aj akali, ke že sme ešte nemohli vedie presnú množinu funkcií vo fáze návrhu. Táto komponenta obsahuje aj pár pomocných funkcií, ktoré tu nebudú spomenuté podrobne, lebo predstavujú len isté pretypovania medzi našimi štruktúrami a databázou.

Pridané funkcie

- updateNode(*Node) - vykoná klasický databázový update na uzly, ktorý je vstupným parametrom.
- removeNode(*Node) - odstráni vstupný uzol z našej databázy.
- readChildrenOfNode(*Node) - na íta potomkov vstupného uzla z databázy a vráti tento uzol s naplneným zoznamom potomkov.
- openConnection() - otvorí pripojenie na databázu, toto pripojenie je reprezentované globálnou premmenou, táto aj nasledujúca funkcia vznikla pre potreby vykonávania viac CRUD príkazov za sebou a bolo by neefektívne stále otvára a zatvára pripojenie.
- closeConnection() - zatvorí pripojenie na databázu.

11 Implementácia

11.1 Ktorý hrá je ktorý

Pri implementácii funkcií klienta je potrebné klás pozornos na to, o sa myslí pod pojmom prvý hrá . Prvým hrá om budeme vždy rozumie hrá a, ktorý bude ako prvý aha z hracej plochy, ktorá predstavuje po iatok hry. Vo všeobecnosti, napr. pri testovaní, totiž nemusí by algoritmus spracovania stromu aplikovaný na po iato nú hraciu plochu hry. Môže sa teda sta , že pri testovaní sa algoritmus spustí z uzla, z ktorého bude aha druhý hrá . Takýto poh ad bolo potrebné zavies kvôli testovaniu a možnosti vzájomného porovnávania výsledkov algoritmov, ktoré používajú interne iné poh ady na to, vzh adom na ktorého zhrá ov ohodnotenie uzla po ítajú.

11.2 Použitá heuristika

V asti analýza bola prezentovaná heuristika, ktorú použili [3] vo svojej práci. My sa však venujeme hre reversi s iným tvarom hracej plochy než oni. Preto sme si prispôsobili prezentovaný princíp na naše podmienky. Použili sme sú ty:

- a - rohových pozícií
- b - pozícií po bokoch (okrem rohov)
- c - rohom susediacich pozícií s rohovými pozíciami
- d - obsadených pozícií hrá a
- e - hrani ných pozícií hrá a

Pre každý typ sme vytvorili rozdiel po tu pozícií obsadených prvým a druhým hrá om. Ozna enie δa, δb, δc, δd, δe predstavuje rozdieli počtu obsadených pozícií v prislúchajúcich skupinách Výsledné heuristické ohodnotenie je potom nasledovné: $h = 10*\delta a +5*\delta b -2*\delta c + 1 * \delta d - 1 *\delta e$

11.3 Implementované algoritmy na klientovi

V klientskej asti systému sme implementovali viaceré algoritmy spracovania stromu preh adávania:

- minimax
- alfa-beta usekávanie
- negascout
- mtd(f)

Prvé dva algoritmy boli implementované hlavne z dôvodu kontroly správnosti, zvyšné dva na reálnu prevádzku. Na základe výsledkov testov, ktoré sú popísané v asti (12Testovanie algoritmov a výkonnosti heuristík), sme sa rozhodli alej používa negascout.

11.4 Ukladanie stavu výpo tu na klientovi

Po zvážení asu, ktorý môže klient po íta jednu úlohu sme sa rozhodli do klientského algoritmu zapracova možnos ukladania stavu výpo tu a pokra ovanie z uloženého stavu – tzv. *checkpointing*. Tento prístup je potrebné zahrnú vzh adom na to, že doba po ítania úlohy môže presiahnu 24 hodín. Vä šina používate ov po takomto ase svoje po íta e vypína. iasto né riešenie by tak bolo stratené.

Ukladanie stavu výpo tu bolo potrebné doplni do algoritmu *Negascout* – ten bol zvolený ako najlepší pre klienta. *Negascout* je implementovaný ako rekurzívny algoritmus. Každý bod výpo tu je charakterizovaný históriou rekurzívnych volaní, lokálnych premenných a dynamických štruktúr priradených volaniu. Navyše môže by rekurzívne volanie vyvolané z dvoch rôznych miest na základe splnenia podmienky.

Na ukladanie stavu sme použili štruktúru kontext:

```
struct Context {
        int alpha;
        int beta;
        int n;
        int score;
        int childrenIndex;
        bool curEvaluated;
        int cur;
}
```

Uchovávame si pole štruktúr *Context*. V každom volaní rekurzívnej funkcie *Negascout* si ukladáme výpo tový kontext do po a do i-teho prvku, kde i je h bka rekurzívneho volania. Rekurzívna funkcia pracuje tak, že každé rekurzívne volanie rozvinie deti uzla a na ne aplikuje rekurziu. V premennej *childrenIndex* si uchovávame, ktorý posledný uzol-die a bol úspešne spracovaný a výpo et je uložený v premenných *alpha*, *beta*, *n*, *score*. Problém viacerých vstupných bodov rekurzívneho volania je vyriešený zna kou *curEvaluated*, pomocou ktorej si zapamätáme i prvé volanie bolo vykonané a v premennej *cur* si uložíme jeho výsledok.

Pri spúš aní výpo tu z odloženého kontextu každé rekurzívne volanie pristupuje ku kontextu, ktorý prislúcha jeho h bke. Na základe *childrenIndex* vieme presko i už spracované deti a v aka *curEvaluated* a *cur* sa vieme rozhodnú , ktoré rekurzívne volanie vykona .

Pri ukladaní stavu výpo tu taktiež serializujeme transpozi nú tabu ku.

Podrobnosti, ako integrova ukladanie stavu výpo tu na klientovi s boinc systémom, sú uvedené v technickej príru ke, asti Integrácia ukladania stavu do boinc systému.

11.5 Databáza

Ako sme už spomínali, z dôvodu mohutnosti stromu h adania pre uvažované symetrické hry, je potrebné, aby as stromu bola uložená na serverovej strane systému BOINC. Na serveri sa teda postupne generuje strom h adania a jeho z jeho spodných uzlov sa vytvoria workunity pre BOINC klientov. Priebežne sa po spracovaní ur itých workunitov, t.j. po ohodnotení niektorých listov stromu v databáze, ohodnotí as stromu a na serveri ostane a nepotrebné uzly sa zmažú. Následne po vypo ítaní všetkých workunitov ostane v databáze len jeden kore ový uzol s výsledkom, ku ktorému sa snažíme dopracova . Ohodnotenie kore ového uzla nám totiž dá odpove na otázku, i sa hra skon í ví azstvom jedného z hrá ov alebo remízou, ak obaja hrá i budú aha najlepšie ako je to možné.

Z výsledkov analýzi vyplynulo, že najefektívnejším spôsobom ukladania stromu na pevný disk a operovaním nad jeho uzlami je použite databázy. Dokonca úplne posta í jedna tabu ka. My sme sa rozhodli použi objektovo-rela ný databázový systém PostgreSQL, ktorý je open-source. Jeden záznam v databáze bude reprezentova jeden uzol stromu, pri om konkrétna dátová reprezentácia uzlu sa môže líši v závislosti od typu symetrickej hry, jej rozsahu a pravidiel.

11.5.1 Databáza pre hru reversi

Dátovú reprezentáciu uzla stromu stavového priestoru pre hru reversi zaoba uje v programe štruktúra *Node* spolo ne s vnorenou štruktúrou *Game*. Tie obsahujú dôležité informácie z h adiska identifikácie uzlov, prepojenia uzlov a ohodnotenia uzlov, ktoré dokáže zabezpe i generovanie a ohodnocovanie stromu na strane servera. Súhrn týchto kritických informácii, ktoré je potrebné efektívnym a minimálnym spôsobom uchováva v serverovej databáze, je uvedený v nasledujúcom preh ade:

```
typedef struct Node {
     // referencia na údaje o hracej ploche
     Game *game;
     // referencia na spájaný zoznam detských uzlov
     struct Node *parent;
     // ohodnotenie uzla pre algoritmus preh adávania
     int value;
} Node;

typedef struct Game {
     // hracia plocha
     Field field[N][N];
     // ktorý hrá  je na rade
     int secondMove;
} Game;
```

11.5.2 Identifikácia uzla

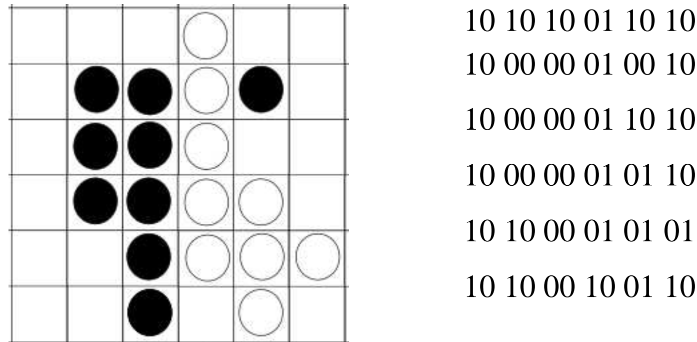
Na to, aby bol uzol uložený do tabu ky databázy je potrebné ho jednozna ne odlíši od iných uzlov, nájs primárny k ú tabu ky. Musíme teda posklada také parametre uzla, ktoré vytvoria jedine ný súbor st pcov tabu ky.

Uzol má reprezentova stav hry reversi, takže ur ite by v databáze nemala chýba **konfigurácia hracej plochy**. Táto konfigurácia ur uje rozloženie kame ov oboch hrá ov na hracej ploche. Je uložená v matici v štruktúre *Game*. Táto matica o rozmeroch N x N imituje hraciu plochu tak, ako ju vidíme kolmo zhora. Jednotlivé prvky matice sú vyplnené hodnotami FIRST pre kame prvého hrá a, SECOND pre kame druhého hrá a a EMPTY pre prázdne polí ko hracej plochy. Ke že do databázy sa snažíme vygenerova strom do najvä šej h bky v závislosti od dostupného pamä ového média na serveri, konfiguráciu hracej plochy sme sa rozhodli reprezentova ako bitové pole. Tri hodnoty pre jedno polí ko hracej plochy vieme reprezentova ako dva bity:

- FIRST = 00
- SECOND = 01

- EMPTY = 10

A teda na uloženie jednej konfigurácie hracej plochy N x N na serveri je potrebné bitové pole ve kosti 2 x N x N. Do bitového po a sa budú uklada hodnoty matice postupne po riadkoch. Pre N=6 uvádzame príklad prepisu stavu hracej plochy na bitový re azec d žky 72:



Obr. 43: Príklad stavu hracej plochhy

Výsledok:101010011010100000010010100000011010100000010110101000010101101000100110

V stavovom priestore hry reversi môže existova rovnaká konfigurácia hracej plochy na viac krát. Jednoduchým príkladom toho je situácia, ke jeden z hrá ov nemá žiaden dostupný ah. Potom hrá , ktorý je alej rade, má k dispozícii tú istú konfiguráciu hracej plochy. Preto zahrnieme do primárneho k ú a tabu ky pre strom okrem konfigurácie hracej plochy aj údaj o tom, kto je na rade. Ako je uvedené vyššie, táto informácia je uložená v premennej *secondMove* štruktúry *Game* a nadobúda hodnoty 0, ak je na rade prvý hrá , a 1 v opa nom prípade. Na dátovu reprezentáciu v databáze posta uje booleovský atribút:

- 0 = false
- 1 = true

11.5.3 Prepojenie uzlov

V predchádzajúcej kapitole sa nám podarilo ur i parametre uzla, ktoré ho jednozna ne ur ujú v stavovom priestore hry a zdá sa, že aj v databáze pre hru reversi: konfigurácia hracej plochy a informácia o tom, ktorý hrá je práve na ahu. Aby sme však strom správne do databázy vygenerovali a mohli ho neskôr aj ohodnoti , musíme vytvori v uzloch prepojenia medzi rodi mi a ich de mi.

Jeden rodi má v strome v zásade jedného a viac potomkov, ale každý potomok má len jedného rodi a. Preto sme sa rozhodli do databázového záznamu o jednom uzle stromu doplni konfiguráciu hracej plochy rodi ovského uzla. Záznam o rodi ovi nejakého uzla je potom vdatabáze jednozna ne ur ený pomocu hracej plochy rodi ovského uzla a ahu opa ného hrá a. Referencia na rodi ovský uzol sa nachádza v programe v štruktúre *Node*, hracia plocha sa opä zakóduje ako bitový re azec o ve kost N x N. Pre kore ový uzol sa zakóduje do postuposti ísla 1.

Po dlhšej úvahe o stavovom priestore hry a jeho rozoberanej databázovej reprezentácii sme však narazili na problém. Zistili sme totiž, že môže nasta situácia, kedy sa do rovnakého uzla môžeme

dosta rôznymi ahmi. V takomto prípade má daný uzol viac ako jedného rodi a, o iasto ne narúša stromový priestor hry a vytvára graf. Hoci takýchto uzlov je ve mi málo, museli sme sa pokúsi rieši aj takúto výnimku. Naše riešenie je nasledovné:

- primárny k ú databázovej tabu ky sme rozšírili na konfiguráciu hracej plochy uzla, ah hrá a a konfiguráciu hracej plochy rodi ovského uzla
- uzly s rovnakými parametrami (hracia plocha a ah), ale rôznymi rodi mi, sú síce všetky uložené do databázy, ale len jeden z nich sa alej rozvíja, „duplicitné“ uzly sa neriešia
- „duplicitné“ uzly predstavujú rovnaký stav hry, takže budú ma rovnaké ohodnotenie ako je ohodnotenie jediného ekvivalentného uzla, ktorý sa alej rozvíja

11.5.4 Ohodnotenie uzlov

Každý uzol v strome hry reversi má prestor pre ohodnotenie preh adávacím algoritmom. Ako sme už spomínali, ohodnocovanie stromu sa bude kona aj na strane servera a teda posledným st pcom, ktorý skompletizuje databázovú tabu ku na serveri, bude práve ohodnotenie uzla.

Ohodnotenie Je uvedené ako premenná *value* v štruktúre *Node*. Nadobúda štyri hodnoty: 1 pre výhru prvého hrá a, -1 pre výhru druhého hrá a, 0 pre remízu a 2 pre ešte neohodnotený uzol. Štyri ísla dokážeme zakódova do dvojbitového po a v databáze takto:

- 0 = 00
- 1 = 01
- -1 = 10
- 2 = 11

11.5.5 Ohodnocovanie stromu

Strom stavového priestoru hry reversi bude na serveri podlieha postupnému ohodnocovaniu preh adávacím algoritmom, tak ako mu to umožnia vypo ítané workunity a teda usporiadanie ohodnotených uzlov v strome. Ke že sa ale strom do databázy postupne generuje (work-generátorom) a sú asne sa postupne ohodnocuje (asimilátorom), bolo potrebné zabezpe i , aby sa vedelo, ktoré uzly stromu ešte v databáze nie sú a preto nie je možné ohodnocovanie ich rodi ov.

Tento problém sme sa rozhodli vyrieši tak, že do databázy sa bude priebežne uklada o každom uzle aj informácia o tom, i už má vygenerovaných všetkých potomkov a teda môže podlieha preh adávaciemu algoritmu.

11.5.6 Dátová reprezentácia v databáze

Na základe predchádzajúcich analýz a diskusií sme vytvorili pä st pcov tabu ky pre databázovú reprezentáciu stromu hry reversi. Tabu ka sa volá *reversi* a jeden záznam z nej pre rozmer hracej plochy N má ve kos $(4 \times N^2 + 4)$ bitov, iže pre reversi 6x6 to iní 148 bitov (19 bajtov) a pre reversi 8x8 zase 260 bitov (33 bajtov).

Názov st pca	Typ st pca	Popis	Viazaný údaj uzla
id	(2xNxN)-bitové pole	konfigurácia hracej plochy uzla	node->game->field
parent	(2xNxN)-bitové pole	konfigurácia hracej plochy rodi a uzla	node->parent->game->field
move	booleovská premenná	ktorý hrá je na ahu	node->game->secondMove
eval	2-bitové pole	ohodnotenie	node->value
allchildrengenerated	booleovská premenná	indikátor, i má uzol všetkých potomkov v databáze	-

Tab. 5: Definícia databázovej tabu ky reversi

11.5.7 Implementácia databázového modulu

Z dôvodu automatizácie práce s databázou stavového priestoru hry reversi sme implementovali osobitný modul zodpovedný za komunikáciu s databázou. Modul bol zakomponovaný do systému ako jedna z astí servera v zdrojových súboroch `db_access.h a db_access.cpp` tak, aby poskytoval rozhranie pre generovanie a ohodnotenie stromu v databáze. Modul využíva na ú el komunikácie s databázou, udržiavanie spojenia, uskuto ovania jednotlivých otázok na databázu knižnicu jazyka C libpq, ktorá je priamo sú as ou PostgreSQL.Databázový modul pozostáva zo siedmych hlavných funkcií, ktoré sú nasledovné:

- `saveNode` – funkcia uloží uzol do databázy t.j. zakóduje pomocou transforma ných funkcií údaje vstupného uzla (štruktúry *Node*) a použije príkaz INSERT na ich uloženie (spolu s údajom indikujúcim, i má uzol vygenerovaných všetkých potomkov v databáze) ako samostatného databázového záznamu
- `updateNode` – funkcia ohodnotí uzol databázy, t.j. použije databázový príkaz UPDATE pre atribút *eval* v uzle jednozna ne definovanom primárnym k ú om (*id*, *parent*, *move*)
- `readNode` – funkcia pre íta ohodnotenie uzla a vráti aj jeho rodi a, t.j. najskôr použije príkaz SELECT atribútu *eval* pre uzol daný údajmi primárneho k ú a, následne vykoná príkaz SELECT pre rodi ovský uzol pod a atribútu *parent* a opa nej hodnoty *move*, a napokon vráti oba uzly naplnené údajmi z databázových st pcov ako štruktúry *Node*
- `readChildrenOfNode` - funkcia získa potomkov uzla, t.j. prostredníctvom zakódovania konfigurácie hracej plochy vstupného uzla použije príkaz SELECT pre uzly, kde atribút *parent* bude zhodný s takouto konfiguráciou pri opa nom *move*, potom spracuje nájdené záznamy a vytvorí potomkov uzla
- `removeNode` – funkcia vymaže uzol z databázy, t.j. pomocou príkazu DELETE vymaže uzol daný údajmi primárneho k ú a
- `setAllChildrenGeneratedBit` – funkcia nastaví uzlu v databáze indikátor, i má uzol všetkých potomkov vygenerovaných do databázy, t.j. použije príkaz UPDATE pre atribút *allChildrenGenerated* pre uzol daný údajmi primárneho k ú a

- `getAllChildrenGeneratedBit` – funkcia zistí, i má uzol všetkých potomkov vygenerovaných do databázy, t.j. použije príkaz SELECT pre atribút *allChildrenGenerated* pre uzol daný údajmi primárneho k ú a

Spojenie s databázou sa vytvorí pred vkladaním kore ového uzla pomocou zavolania funkcie `openConnection` a zatvára sa v prípade, kedy je to potrebné funkciou `closeConnection`. Spomínané pomocné transforma né funkcie na prepis uzlov v podobe dátových štruktúr jazyka C na atribúty databázových záznamov a naopak pod a vyššie uvedených princípov sú:

- `getBitValue` – prepis hracej plochy z premennej do databázy
- `getEval` – prepis ohodnotenia z premennej do databázy
- `getMove` – prepis ahu hrá a z premennej do databázy
- `setSecondMove` – na ítanie ahu hrá a z databázy do premennej
- `setEval` – na ítanie ohodnotenia z databázy do premennej
- `setGameField` – na ítanie hracej plochy z databázy do premennej

11.5.8 Kompilácia databázového modulu

Pre kompiláciu databázového modulu je potrebné kvôli použitiu knižnice *libpq* prida do programu hlavi kový súbor `libpq-fe.h`. Následne sa do *makefilu* pridajú pre kompilátor tri parametre do premennej *CPPFLAGS*:

1. parameter s adresárom, kde boli nainštalované PostgreSQL hlavi kové súbory
2. parameter `-lpq` pre natiahnutie knižnice *libpq*
3. parameter s adresárom, kde sa nachádza samotná knižnica *libpq*

Pre Linuxový server a inštaláciu databázového systému PostgreSQL, s ktorou sme pracovali my, pridáme do premennej *CPPFLAGS* tieto riadky:

```
CPPFLAGS += -I/usr/local/pgsql/include \
            -L/usr/local/pgsql/lib -lpq
```

11.6 Implementácia asimilátora

Asimilátor v skuto nom projekte beží ako démon a je v nekone nej slu ke volaná funkcia:

```
int assimilate_handler(WORKUNIT& wu, vector<RESULT>& , RESULT&
canonical_result)
```

funkcia sa nachádza v hlavi kovom súbore *assimilate_handler.h* :

```
#include <vector>
#include "boinc_db.h"
extern int assimilate_handler(WORKUNIT&, std::vector<RESULT>&,
RESULT&);
```

preto je potrebujeme implementova . My sme ako prvotnú verziu vzali príklad z demo aplikácie v boinc-u **sample_assimilator.c** a ten sme prispôsobili nášmu problému. V tomto zdrojovom súbore je zretelný vnútorný cyklus, ktorý prebehne pre všetky zvalidované výsledky. Preto je naša implementácia umiestnená len v tomto cykle a prebieha v nasledovných krokoch:

1. na ítanie výsledku (konvertovanie do štruktúry Node), zo súboru, ktorý poslal klient
2. uloženie došlého výsledku (volanie saveNode(Node) z databázového modulu)
3. volanie funkcie processGameResult(Node) pre získaný uzol - Node

Pri implementácii je potrebné pozna štruktúru WORKUNIT, ktorá predstavuje opis práce vrátenej od klienta, iže vypo ítaný workunit.

Používali sme aj naše informa né a ladiace výpisy pomocou zabudovaného logovacieho mechanizmu v boinc-u.

12 Testovanie

V tejto kapitole popisujeme testy ktoré sme po as alebo po implementácii systému vykonali.

12.1 Testovanie algoritmov a výkonnosti heuristík

Po implementovaní heuristiky do algoritmu spracovania stromu preh adávania sme vykonali testy. Porovnávali sme as potrebný na vyriešenie hry z rôznych pozícií, v ktorých bolo treba doplni 14 až 19 kame ov. Tabu ka 6 zobrazuje asy potrebné na spracovanie prislúchajúcich stromov preh adávania rôznymi algoritmami bez použitia heuristiky, tabu ka 3 s použitím heuristiky.

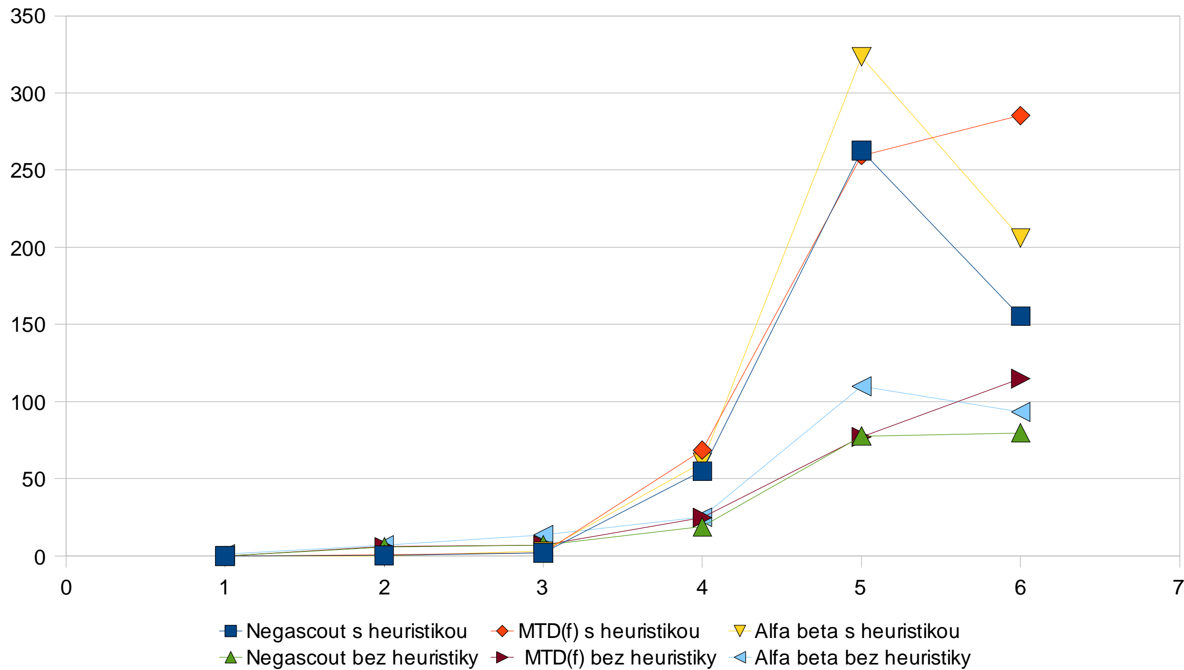
vo ných pozícií	**as (s)**		
	negascout	**mtd(f)**	**alfa beta**
14	0.00	0.00	0.00
15	0.47	0.86	0.00
16	2.08	2.16	3.17
17	54.99	68.53	60.74
18	262.71	259.49	323.37
19	155.353	285.34	206.07

Tab. 6: Porovanie d žky spracovania stromu preh adávania algoritmami Negasout, Mtd(f) a Alfa-beta usekávanie bez použitia heuristiky

vo ných pozícií	**as (s)**		
	negascout	**mtd(f)**	**alfa beta**
14	0.00	0.00	1.19
15	5.85	6.04	7.15
16	7.06	7.16	13.86
17	18.89	24.83	25.14
18	77.72	77.15	110.02
19	79.84	114.84	93.39

Tab. 7: Porovanie d žky spracovania stromu preh adávania algoritmami Negasout, Mtd(f) a Alfa-beta usekávanie s použitím heuristiky

Na Obr. 44 porovnávame časy jednotlivých algoritmov s heuristikou aj bez nej.



Obr. 44: Porovanie d žky spracovania stromu preh adávania algoritmami Negasout, Mtd(f) a Alfa-beta usekávanie s a bez použitia heuristiky

Ako môžeme z grafov vidie , pre uvažovanú hru reversi sa osved il najviac algoritmus negascout. Rozhodli sme sa ho teda použi v kone nej verzii klienta.

Profilérom kódu kompilátora g++ sme zistili, že neustále vyhodnocovanie heuristickej funkcie zabralo až 60% asu vykonávania programu. Ako však zozískaných hodnôt môžeme sledova , aj napriek tomu heuristika výrazne zvýšila výkon algoritmu.

12.2 Testovanie klienta

Pôvodne sme chceli testova klienta tak, že na internete sa nájdu nejaké konfigurácie hracej plochy. Pre tieto konfigurácie by boli známe riešenie. Tento postup testovania sa ukázal ako nefunk ný, lebo pri h adaní na internete sa nám podarilo nájs iba jednu, ktorá sp ala naše požiadavky, vedeli sme kto má z danej konfigurácie hracej plochy vyhra .

Pre to sme sa rozhodli testova správnos práce klienta nasledovne. Vytvorili sme jednoduchý program, ktorý na ítava zo vstupného súboru rôzne konfigurácie hracej plochy. Tieto konfigurácie boli generované dvomi spôsobmi. Prvý bol taký, že sme náhodne vygenerovali nejakú konfiguráciu hracej plochy. Druhou alternatívou bolo vygenerovanie reálnych konfigurácií hracej plochy. V prvom prípade sme mohli vygenerova aj hraciu plochu, ku ktorej sa pri reálnej hre nedá dopracova . Pre tieto vstupy sme nevedeli kto má z danej konfigurácie vyhra . Postupne sme spúš ali každého z testovaných klientov pre jednotlivé konfigurácie hracej plochy zo vstupu. Ke dali všetky testované algoritmy rovnaký výstup pre danú konfiguráciu, tak sme to považovali za úspešné otestovanie (funk nos klientov je správna). Ak nedali všetci klienti rovnaký výsledok, tak sme zistili, že niekde v testovaných klientoch je chyba a snažili sme sa ju odstráni . Týmto

spôsobom testovania boli odhalené drobné nedostatky v klientoch, ktoré prevažne vznikli z nepozornosti. Tieto chyby boli odstránené. Následne sa opakoval proces testovania na upravených klientov.

12.3 Testovanie asimilátora

Asimilátor bol testovaní len na reálnych údajoch získaných od klienta. Lebo vytvára výsledky tzv. result-y od klienta by teoreticky išlo, ale bolo by to obchádzaním klasických postupov. Samotné testovanie prebiehalo v týchto krokoch:

- vygenerovanie *workunit* generátorom zadaním vstupného súboru, ktorý obsahoval po iato ný stav hry.
- po kanie na vyriešenie klientmi, d žka trvania v závislosti od stavu hry, ktorý sme poslali do genrátora v predchádzajúcom bode.
- pred asimiláciou výsledkov musí zbehnú validácia výsledku a až potom asimilácia, validátor je štandardne pustení pri pustení *boinc*-u ako démon. Bližšie popísané v asti o konfigurácii *boinc*-u.
- asimilátor môže samozrejme tiež beža ako démon, ale pri testovaní sme ho spúš ali aj ru ne pre jednoduchšie spracovanie logov - hne sme na konzole videli *debug* výpis.
- po asimilácii alebo aj po as (ak sme ju spúš ali postupne alebo ak asimilátor bežal ako démon) sme pozerali do našej databázy, i sú uzly ohodnotené.
- úspešný test asimilátor skon il tak, že boli spracované všetky *workunit*-y a prijaté všetky *result*-y a v našej databáze zostal jediný (vstupný) uzol a bol ohodnotený.

V podstate jediným premenným parametrom v tomto teste bol po iato ný po et *workunit*-ov. My sme testovali od minimálneho po tu 2 *workunit*-y (4 *result*-y od klientov) po cca 500 *workunit*-ov (potom 1000 *result*-ov).

Pri testovaní sa vyskytli chyby najmä v databázovom modulu a spomínanej funkcie processGameResult(). Týchto chýb nebolo málo a nedali sa nájs naraz, lebo pri výskyte jednej chyby asimilátor skon il a pusti sa dal až po opravení chyby, skompilovaní a znovu spustení, preto testovanie sa dos pre ahovalo.

12.4 Testovania generátora

Testovanie generátoru úloh (workgenerator-u) na hre reversi6x6 nám pomohlo odhali nieko ko chýb. Testovanie bolo prevádzané zásadne osobou odlišnou od programátora, ím sa dosiahlo rozšírenie pochopenia systému a pomohlo to pri h adaní záludných chýb, ktoré autor kódu prehliadal.

Odhalené chyby boli opravené a výsledky testovania ukázali napríklad i to, že sto tisíc uzlov sa do boinc databázy na 3 GHz P4 po íta i so sata diskom dá vygenerova za as okolí troch minút. Uzlov v celom priestore hry je však nieko ko násobne viac a tak ani do h bky 13 od za iatku hry sa nedá dosta v rozumnom ase. Generovanie prvých 13 vrstiev trvá asi de . Trinásta vrstva je zaujímavá tým, že je to limit diskového miesta asi 1 gigabajt v externej databáze. Pri tomto

obmedzení sme však nepo ítali s obmedzením databázy systému BOINC a i s obmedzením diskového miesta pri ukladaní úloh (workunit-ov) na disk. Pre takto ve ké problémy bude treba generovanie úloh rozdeli na menšie asti.

13 Výsledky

Cie om tohto projektu bolo získanie odpovede na otázku, kto má výhernú stratégiu pre reversi 8x8. i hrá idúci ako prvý, alebo jeho oponent. Okrem hry reversi bolo cie om projektu odpoveda na obdobnú otázku i pre hru go. Tieto ambiciózne ciele sme sa snažili naplni v o najvä šej miere, ale dbali sme i na rôzne aspekty softvérového inžinierstva, ktoré neboli v cie och spomenuté.

Dosiahnu modulárnos , pri ktorej by sta ilo doplni malý po et modulov pre zmenu hry bolo naším vnútorným cie om od za iatku. Tento ciel mal u ah i zmenu hry reversi na go. Pri návrhu sa objavilo nieko ko problémov, ktoré sme preklenuli, avšak problémy preh adávania stromu hry ukázali, že sa nejedná tak celkom o strom.

Aplikácia pre generovanie úloh bola pôvodne navrhnutá tak, aby jednorazovo vygenerovala na serveri hru do ur itej h bky, ktorá sa bude alej distribuova . Toto riešenie sa ukázalo ako neposta ujúce pre vä šie problémy a generovanie úloh sa dorobilo i na postupnú verziu, ktorá bola schopná doda požadovaný po et úloh pre klientov a vygenerovala k nim príslušnú as stromu. Nakoniec bola pridaná ešte funkcionalita potrebná pre osekávanie na serveri a nieko ko testovacích pomôcok.

Preh adávací algoritmus klientskej bol pilovaný takmer po as celého trvania projektu a rôzne vylepšenia pôvodných algoritmov priniesli skrátenie výpo tu. Posledná verzia dokáže na klientskej strane uklada „chceckpoint" a pokra ova v prerušenom výpo te i po reštarte po íta u. Algoritmus preh adávania na klientskej strane používa algoritmus negascout, ktorý s pomocou heuristík a transpozi nej tabu ky je schopný vypo íta výsledok posledných 17 ahov hry reversi 6x6 do pol minúty na bežnom po íta i.

Posledný ve ký modul v re azci výpo tu je asimilátor. Stará sa o spracovanie prijatých výsledkov. Okrem uloženia prijatého výsledku do databázy sa však stará o ohodnotenie rodi ov, pokia boli prijaté všetky ohodnotenia potomkov a o osekávanie. Osekávanie bolo pridané ako posledné a prinieslo o akávané urýchlenie výpo tu, nako ko umožnilo použi na serveri jednoduché, no ú inné osekávanie. V spojení s vhodnou heuristikou prinieslo zna né zlepšenie.

14 Zhodnotenie

Projekt je ukon ený a jeho sú asti sú funk né. Implementovaná bola hra reversi, na ktorej sa uskuto nilo nieko ko výpo tov, avšak as nebol dostato ný pre vypo ítanie reversi 8x8. Pripojením vä šieho množstva po íta ov by sa mohol výpo et zna ne urýchli , ale zlepši by bolo možné i heuristické algoritmy na serveri, prida priority pre úlohy na serveri tak, aby sa umožnilo efektívne usekávanie, alebo použi iný algoritmus na klientskej strane.

Pred dosiahnutím cie u brzdilo tím za iato né zosúladenie sa v práci i komunikácii a z asti i pre lenov tímu menej známy programovací jazyk. Samotné prostredie BOINC zohralo nemalú úlohu, kedy jeho detaily neboli zdokumentované a napríklad h adanie významu návratovej hodnoty pre ukladanie úloh na serveri bolo nutné previes preh adávaním zdrojových súborov serveru.

15 Použitá literatúra

1. ALLIS, L.V. *Searching for Solutions in Games and Artificial Intelligence. Ph.D. thesis*. University of Limburg, Maastricht, 1994. ISBN 90-9007488-0.
2. BAKER, K. *The Way To Go* [online]. 2001 [cit. 2007-11-14]. Dostupné na: <http://www.usgo.org/usa/waytogo/W2Go8x11.pdf>.
3. DO, CH., CHONG, S., TONG, M., HUI, A. CS 221 *Othello Report Demostenes*, 2002.
4. FANG, R. *Othello: From Begginer To Master* [online]. 2003 [cit. 2007-11-14]. Dostupné na: <http://www.fnork.ath.cx/ocd/data/books/randy-fang-beginner.pdf>.
5. FEINSTEIN, J. *Perfect play in 6x6 Othello from two alternative starting positions* [online]. 2004 [cit. 2007-11-14]. Dostupné na: <http://www.feinst.demon.co.uk/Othello/6x6sol.html>.
6. International Business Machines Corp. *AIX documentation : JFS size limits* [online]. [cit. 2007-11-14]. Dostupné na: <http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.baseadmn/doc/baseadmndita/jfssizelim.htm>.
7. LIANG., Q. *The evolution of Mulan: Some studies in game-tree pruning and evaluation functions in the game of Amazons. Master's thesis*. University of New Mexico, 2003.
8. Microsoft Corporation. *Local File Systems for Windows* [online]. 2004 [cit. 2007-11-14]. Dostupné na <http://download.microsoft.com/download/5/b/5/5b5bec17-ea71-4653-9539-204a672f11cf/LocFileSys.doc>.
9. NÁVRAT, P., et. al. *Umelá inteligencia*. Bratislava: Slovenská technická univerzita v Bratislave, 2006.
10. PHOOPHAKDEE, B., ZAKI, M. J. Genome-scale disk-based suffix tree indexing. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. New York : ACM, 2007, s. 833-844.
11. PostgreSQL Global Development Group. *PostgreSQL : About*. 2007 [cit. 2007-11-14]. Dostupné na: <http://www.postgresql.org/about/>.
12. Silicon Graphics, Inc. *XFS Overview & Internals : 02 – Overview* [online]. 2006 [cit. 2007-11-14]. Dostupné na: <oss.sgi.com/projects/xfs/training/xfs_slides_02_overview.pdf>.
13. STEEN, J. van der. *Go, an addictive game : the rules of go* [online]. 2007 [cit. 2007-11-14]. Dostupné na: <http://gobase.org/studying/rules/?id=0&ln=uk>
14. Sun Microsystems, Inc. *Solaris ZFS—The Most Advanced File System on the Planet* [online]. 2006 [cit. 2007-11-14]. Dostupné na: <http://www.sun.com/software/solaris/ds/zfs.jsp>.
15. The FreeBSD Project. *Large data storage in FreeBSD* [online]. 2006 [cit. 2007-11-14]. Dostupné na: <http://www.freebsd.org/projects/bigdisk/index.html>

16. The University of California, Berkeley. *Berkeley Open Infrastructure for Network Computing* [online]. 2007 [cit. 2007-11-14]. Dostupné na: <http://boinc.berkeley.edu>.
17. WERF, E. van der. *5x5 GO IS SOLVED* [online]. 2002 [cit. 2007-11-14]. Dostupné na: <http://erikvanderwerf.tengen.nl/5x5/5x5solved.html>.
18. Wikipedia. *Reiser4* [online]. 2007 [cit. 2007-11-14]. Dostupné na: <http://en.wikipedia.org/wiki/Reiser4>.
19. Wikipedia. *ReiserFS* [online]. 2007 [cit. 2007-11-14]. Dostupné na: <http://en.wikipedia.org/wiki/ReiserFS>.

Príloha A – Slovník pojmov

Alfa beta usekávanie – algoritmus na ohodnotenia stromu preh adávania hry. Vylepšenie algoritmu MiniMax založené na useknutí uzlov, ktoré nemôžu ovplyvni výsledok. (angl. *alpha beta pruning*)

Canonical Result – Je to výsledok (result), ktorý vytvorí validátor (alebo za neho ozna í nejaký existujúci) pri validácii prišlých result-ov. Je to akýsi zástupca výsledku. Pod a aplikácie ním môže by napríklad priemer všetkých validných výsledkov, alebo prvý výsledok doru ený naserver.

Dobrovolník – Užívate vlastniaci minimálne jeden po íta , ktorého výkon chce poskytnú pre BOINC projekty.

Faktor vetvenia – priemerný po et detí uzla v strome h adania

Flop (flops) – skratka pre (**fl**oating-point **op**erations per **s**econd), teda po et desatinných operácií za sekundu. Je jednou z jednotiek merania výkonu procesora.

GO – stolová hra pre dvoch hrá ov, v ktorej sa hrá i snažia ovládnu o najvä šie územie

Heuristika – odporú anie, ktoré má vies k dobrému výsledku

Hra s nulovou sumou – hra, v ktorej zlepšenie situácie jedného hrá a znamená zhoršenie situácie druhého hrá a (angl. *zero-sum game*)

Checkpointing - metóda, pri ktorej si klientska aplikácia z asu na as (ur uje BOINC klient) ukladá na disk aktuálny stav výpo tu scie om pokra ova z aktuálneho bodu po vypnutí a následnom zapnutí aplikácie.

Klient – BOINC klient, teda aplikácia, za pomoci ktorej sa prihlasuje na PC dobrovo níka do projektov. Táto aplikácia s ahuje z projektových serverov klientske aplikácie, workunit-y a manažuje celú komunikáciu i zdie anie prostriedkov viacerými projektami.

Klientska aplikácia – je výpo tovo náro ná aplikácia, ktorú distribuuje BOINC server a na po íta dobrovo níka sa dostane za pomoci BOINC klienta

MiniMax – základný algoritmus na ohodnotenie stromu preh adávania hry

NegaScout – vylepšenie algoritmu alfa beta usekávanie na základe preh adávania s minimálnym oknom

MTD(f) – vylepšený algoritmus ohodnotenia stromu preh adávania hry založený na h adaní s minimálnym oknom

NegaMax – vylepšenie MiniMax algoritmu. Zjednocuje hrá ov MIN a MAX do jedného typu.

Okno h adania – dvojica (α,β) ktorá pri algoritmoch zalozených na alfa beta usekávaní ur uje, ktoré uzly má zmysel preh adáva (angl. *search window*)

Othelo – obdoba hry reversi, pri ktorej je však jednozna ne ur ená za iato ná pozícia štyroch kame ov na hracej ploche

Minimálne okno – také okno h adania, pre ktoré platí $\alpha=\beta-1$

Reversi – stolová hra pre dvoch hrá ov, v ktorej si hrá i navzájom preberajú figúrky.

Strom h adania – strom, ktorý sa vytvára pri preh adávaní stavového priestoru.

Symetrická hra – hra, pri ktorej majú obaja hrá i rovnaké (symetrické) postavenie

Transpozi ná tabu ka – tabu ka, v ktorej sa ukladajú ohodnotenia už preh adaných uzlov stromu h adania

Ve mi slabé riešenie – je ur enie, ktorý z hrá ov hru vyhrá (angl. *ultra-weak solution)*

Vyhladenie – stav, ke jeden z hrá ov už nemá žiadne kamene

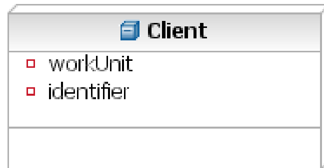
Workunit – súbor, ktorý obsahuje úlohu pre klientsku aplikáciu. Niekedy sa takto nazýva i samotná úloha.

UML – Unified Modeling Language - štandardný špecifika ný jazyk

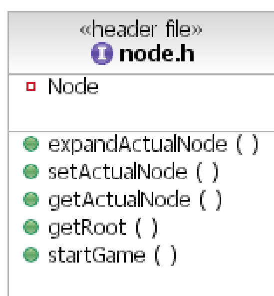
CRUD - štyri základné databázové funkcie (skratka z angl. Create, Read, Update, Delete) popisuje možné operácie s databázou.

Príloha B – Použitá notácia diagramov

Z dôvodu implementácie projektu   isto v jazyku C, bez použitia objektových štruktúr, je použitá notácia istou nadstavbou štandardu UML.
Objekty a asociácie v diagramoch

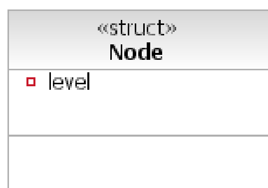


- jednotka systému, opisuje jeden fyzický stroj - po íta
- má svoje vlastnosti (▫), v našom prípade sú asti na najvyššej úrovni abstrakcie



- rozhranie (*interface*), pri C programovaní (náš prípad) predstavuje jeden hlavi kový súbor
- obsahuje mená štruktúr (▫) v jazyku C
- v spodnej   asti prototypy funkcií (●)

- notácia pre zdrojový súbor v jazyku C



- notácia pre klasickú C štruktúru
- štruktúra obsahuje premenné (▫)

- notácia pre vzdialené rozhranie, predstavuje pravidlá pre konkrétnu implementáciu

- realizácia, v našom prípade istý zdrojový súbor implementuje daný hlavi kový súbor

- používa, v našom prípade, istý zdrojový súbor obsahuje deklaráciu *#include* pre daný hlavi kový súbor

Tab. 8:

Príloha C – Používate ská príru ka

1. Inštalácia BOINC servera a jeho nastavenie

Táto kapitola popisuje inštaláciu, základnú konfiguráciu BOINC servera. Nepopisujeme tu kroky potrebné na konfiguráciu závislostí, ako napr. webservera Apache a databázového servera MySQL.

V našom prípade sme ako server použili Linux, konkrétne distribúciu Gentoo Linux. Kroky popísané v tejto príru ke preto bude pravdepodobne potrebné upravi v prípade použitia inej distribúcie alebo inej platformy.

1.1. Postup pri inštalácii

- Stiahneme si poslednú verziu BOINCu.
```
svn co http://boinc.berkeley.edu/svn/trunk/boinc
```
- Zabezpe íme inštaláciu všetkých závislostí pod a zoznamu na
  http://boinc.berkeley.edu/trac/wiki/SoftwarePrereqsUnix
- Skompilujeme BOINC
```
cd boinc
./_autoinstall
./configure -disable-client
./make
```
- Inštalátor by mal automaticky prida do systému užívate a a skupinu boinc. Ak sa tak nestalo, pridáme ju ru ne teraz.
```
useradd boinc
groupadd boinc
```
- Pridáme užívate a boinc do skupiny apache
```
gpasswd -a apache boinc
```
- Pridáme do MySQL databázy užívate a s právami administrátora
```
mysql -u root -p
grant all on *.* to boinc identified by 'heslo'
```
- Nastavíme http server Apache
  Do httpd.conf pridáme
```
DefaultType application/octet-stream
```
  Z httpd.conf odstránime riadok
```
AddHandler imap-file map
```
- Nastavíme PHP. Do php.ini pridáme
```
magic_quotes_gpc = On
```

Pár tipov v prípade problémov:

- skontrolova užívate ské práva
- skontrolova nastavenie webservera

Ako vidno, inštalácia nie je jednoduchá. Správanie inštalátora je v niektorých situáciách nepredvídate né, preto odporú ame vytvori pre BOINC vlastnú MySQL inštanciu. Tiež by bolo dobré kvôli jednoduchejšej správe servera Apache inštalova boinc do nového virtuálneho hostu.

2. Vytvorenie nového projektu

Táto kapitola popisuje vytvorenie nového projektu na strane BOINC servera.

V našom prípade sme ako server použili Linux, konkrétne distribúciu Gentoo Linux. Kroky popísané v tejto príru ke preto bude pravdepodobne potrebné upravi v prípade použitia inej distribúcie alebo inej platformy.

2.1. Postup pri vytvorení nového projektu

- Spustíme skript make_project
  ```
  cd boinc/tools/
  make_project --test_app myproject
  ```
- Skript vytvorí adresárovú štruktúru a súbory potrebné na beh projektu, zárove vytvorí konfigura né súbory, ktoré je potrebné prida do systému
  ```
  echo .../myproject/myproject.httpd.conf
  >> /etc/apache2/httpd.conf
  echo .../myproject/myproject.crontab >> /etc/crontab
  ```
- Prispôsobíme .../html/project/project.inc
  ```
  vim .../html/project/project.inc
  ```
- Zabezpe íme prístup k .../html/ops napr. pomocou .htaccess
  ```
  vim .../html/project/.htaccess
  ```
- Odteraz je stránka projektu prístupná na
  http://hostname/myproject
- Inicializujeme databázu projektu
  ```
  .../bin/xadd
  .../bin/update_versions
  ```
- Naštartujeme démonov projektu
  ```
  .../bin/start
  ```

3. Konfigurácia BOINC servera

Konfigurácia servera z h adiska spustených démonov a príkazov spúš aných pri štarte *boinc* servera je zabezpe ená cez konfigura ný XML súbor **config.xml**, ktorý sa nachádza v kore ovom adresári *boinc* projektu. Základná štruktúra tohto súboru je:

```
<boinc>
  <config>
    [ configuration options ]
  </config>
  <daemons>
    [ list of daemons ]
```

```
  </daemons>
  <tasks>
    [ list of periodic tasks ]
  </tasks>
</boinc>
```

Napríklad zaregistrovanie nami skompilovaného asimilátora s názvom cockroachAssimilator prebehne nasledovne:

```
...
<daemon>
     <cmd>
          cockroachAssimilator -d 3 -app reversi
      </cmd>
</daemon>
...
```

Vo vstupnom súbore sa menia iba tieto jednotlivé príkazy (*cmd*), ostatné sme nechali ako default hodnoty v pôvodnom **config.xml** súbore vytvorenom pri inštalácii servera.

3.1. Klientska aplikácia na serveri

Táto kapitola popisuje kroky potrebné pre pridanie a údržbu klientskych aplikácií na serveri. V boinc serveri sa udržuje sada klientskych aplikácií skompilovaných pre rôzne platformy. Tieto aplikácie sú potom posielané príslušným klientom pod a HW a SW platformy, na ktorej bežia.

Pridanie klientskej aplikácie

Pridanie novej klientskej aplikácie je nezávislé na iných astiach projektu. Po pridaní klientskej aplikácie je možné generova úlohy pre tieto aplikácie. Pridanie aplikácie si vyžaduje nakopírovanie samotného binárneho súboru aplikácie do aplika ného adresáru a spustenie skriptu, ktorý sa postará o zaregistrovanie aplikácie v boinc databáze. Presný popis je uvedený v neskôr uvedenom postupe.

```
http://boinc.berkeley.edu/trac/wiki/UpdateVersions
```

Na uvedenej stránke sa nachádza postup, ktorý bude popisovaný v tejto kapitole. Je to dokumentácia vo wiki štýle (sú as trac-u), ktorá je udržovaná prevažne tvorcami BOINC platformy.

Postup pridávania nového programu:

presun do adresára ***/boinc/menoProjektu/apps***

vytvori adresár s krátkym menom programu, napíklad ***reversi***

nahranie binárky programu do vytvoreného adresára

premenovanie binárky do tvaru: NAME_VERSION_PLATFORM[__PLAN-CLASS][.ext], například ***reversi6x6_1.0_windows_intelx86.exe***, alebo ***reversi6x6_1.2_i686-pc-linux-gnu***

presun do hlavného adresára projektu, teda ***/boinc/menoProjektu***

spustenie ***./bin/update_versions***

Upgrade klientskej aplikácie

Upgrade nejakej klientskej aplikácie na strane serveru sa robí obdobne, ako pridanie novej. Upgradom aplikácie sa táto aplikácia nemusí nutne upgradnú u klientov. Ak chceme, aby sa na alšie workunit-y používala výhradne najnovšia verzia aplikácie, je to potrebné uvies pri registrácii workunit-u.

Postup upgradu klientskej aplikácie:

nahranie novej binárky do adresára /boint/mwnoProjektu/apps/menoProgramu

premenovanie binárky do tvaru: NAME_VERSION_PLATFORM[__PLAN-CLASS][.ext], například ***reversi6x6_1.0_windows_intelx86.exe***, alebo ***reversi6x6_1.2_i686-pc-linux-gnu*** (verzia musí vy vyššia, ako existujúca najvyššia verzia!)

presun do hlavného adresára projektu, teda ***/boinc/menoProjektu***

spustenie ***./bin/update_versions*s**

4. Workgenerátor

Tento dokument popisuje prácu a možnosti generátora úloh (workgenerator), ktorý je výsledkom tímovej práce. Bude popísaný jeho štandardný spôsob použitia, jeho prepína e. Ku každému prepína u bude napísané podrobné vysvetlenie. Prvý odsek bude obsahova popis pre „používate a" a alšie odseky budú obsahova podrobné informácie pre programátora.

4.1. Štandardný spôsob použitia programu

Program cockroachWorkgenerator sa štandardne používa na vygenerovanie workunit-ov pre boinc server. Generujú sa uzly v hre v ur itej h bke. Okrem ukladania workunit-ov doboinc databázy a na disk pre upload, sa generuje i celý strom až do zadanej h by v externej databáze. Podrobnejšie popísané je generovanie neskôr.

```
dawn bin # ./cockroachWorkGenerator -h
Help:

-h      Help (this help)
-d 3    Debug level of logs
-p      DONT save nodes to posgre DB (no saveNode() call)
-b      DONT save anything to boinc. (no file is saved to disc, now
        create_work() call)
-f in   File with initial game configuration.
```

```
-j n   How deep to go in generating nodes. Default is 3.
```

Predchádzajúci výpis ukazuje help programu. Ukazuje možnosti programu. Nasledujúci príkaz demonštruje klasické použitie programu.

```
dawn bin # ./cockroachWorkGenerator -j 3
2008-04-26 23:41:18.2903 [normal  ] Setting new depth.
2008-04-26 23:41:18.2992 [normal  ] Starting
2008-04-26 23:41:18.2992 [normal  ] Príprava štruktúry wu-
 ka2008-04-26 23:41:18.2993 [normal  ] node = stratGame(null)
2008-04-26 23:41:18.2993 [normal  ] Po iatok rekurzívneho volania
generovania stromu.
2008-04-26 23:41:18.2996 [normal  ] Rekurentné generovanie stromu
bolo ukon ené. (recordsInsertedToBoinc=11, recordsInsertedToDB=33,
recordsNotInsertedToDB=0, recordsDoubledDB=0)
```

Výpis programu informuje o úspešnom ukon ení generovania workunit-ov. Zposledného riadku vidno po ty uzlov, ktoré program vygeneroval a nejako spracoval. Konkrétne parameter „-j 3" povedal, že program má prejs do h bky 3 (v strome hry). Program pri ceste do cie ovej h bky ukladá každý jeden vygenerovaný uzol do externej databázy (databáza mimo boinc-u) a každý uzol v cie ovej h bke uloží i do databázy boincu a do upload adresáru boincu zoserializuje daný uzol.

Popis parametru -p

Parameter -p slúži hlavne pri ladení projektu. Pri použití tohto parametru sa zamedzí ukladaniu akýchko vek uzlov do externej databázy.

Popis parametru -b

Parameter -b je obdobný ako parameter -p, ale tento parameter zamedzuje ukladaniu akýchko vek dát do boinc projektu. Neukladanú sa ani informácie do databázy a ani sa neserializujú uzly do upload adresára boinc projektu. Využitie je opä hlavne pri ladení projektu a jeho sú astí.

Popis parametru -f FILE

Tento parameter dovo uje nastavi súbor so za iato nou konfiguráciou. Odovzdaný súbor ako parameter sa použije ako za iato ná konfigurácia, z ktorej sa bude generova herný strom.

Odovzdaný súbor sa otvorí a deserializuje sa z neho herný stav, ktorý sa použije ako východiskový. Tento parameter sa používa hlavne pri ladení projektu. Dovo uje užívate ovi tak vygenerova iba malú as stromu hry a odskúša si tak funk nos samotného projektu.

Popis parametru -d

Parameter d slúži na nastavenie úrovne výpisu hlášok. Použitie je hlavne pri nájdení chyby a potrebe podrobne zisti operáciu, po ktorej aplikácia vypísala chybovú hlášku a skon ila. Ukážkové použitie nastavuje úrove výpisov nadebug, teda na maximálnu.

```
dawn bin # ./cockroachWorkGenerator -j 3 -d 3
```

Popis parametru -t FILE

Parameter **-t** sa používa pri postupnom generovaní. Tento typ generovania uzlov stromu sa používa pri generovaní stromu do takej h bky, v ktorej je po et uzlov presahujúci kapacitu externej databázy, alebo BOINC serveru. Treba pamäta na to, že generovanie vrstvy **j** znamená, že všetky uzly na ceste do tejto vrstvy vrátane vrstvy samotnej majú by uložené v externej databáze a posledná vrstva sa musí uchováva v BOINC serveri ako úlohy (workunit-y), o znamená záznamy v databáze, i súbory na disku.

Parametrom sa nastavuje súbor, do ktorého sa ukladá a následne íta „ChceckpointVector". Tento vektor udáva polohu posledne vygenerovaného uzlu v strome. Formát súboru je jednoduchý. Na prvom riadku je po et zložiek vektoru a nakaždom alšom riadku sú jednotlivé zložky vektoru. Pri spustení generátoru úloh po prvý krát bez existencie súboru FILE sa nastaví vektor tak, aby všetky jeho zložky boli rovné íslu -1. Program zavolá skript s menom *workunit_row_count* a jeho návratovú hodnotu považuje za po et úloh, ktoré je potrebné vygenerova . Vygeneruje daný po et úloh do BOINC serveru a uloží si svoj aktuálny „CheckpointVector". Pri každom alšom spustení daný vektor na íta a pokra uje sa v generovaní tam, kde sa skon ilo. Po poslednom generovaní bude vektor nastavený takto: *0 -> -1 -> -1 -> -1....*

Popis parametru -s FILE

Tento parameter slúži na ladenie generátoru úloh. Spôsobí, že sa budú úlohy uklada do zadaného súboru. Na každý riadok sa uloží úloha tak, ako keby sa serializovala pre BOINC server. Ak by sa úloha uložila iba do externej databázy, je tu iba samotná serializácia uzlu na riadku. Ak by sa úloha vygenerovala i pre BOINC server, bude za serializovaným stavom hry ešte medzera a písmeno B.

Týmto parametrom je teda možné jednoducho odskúša i postupné generovanie a generovanie úplné funguje zhodne. Prevedenie takéhoto testu je uvedené na príslušnom mieste v dokumentácii.

5. Inštalácia a práca s BOINC klientom pre windows

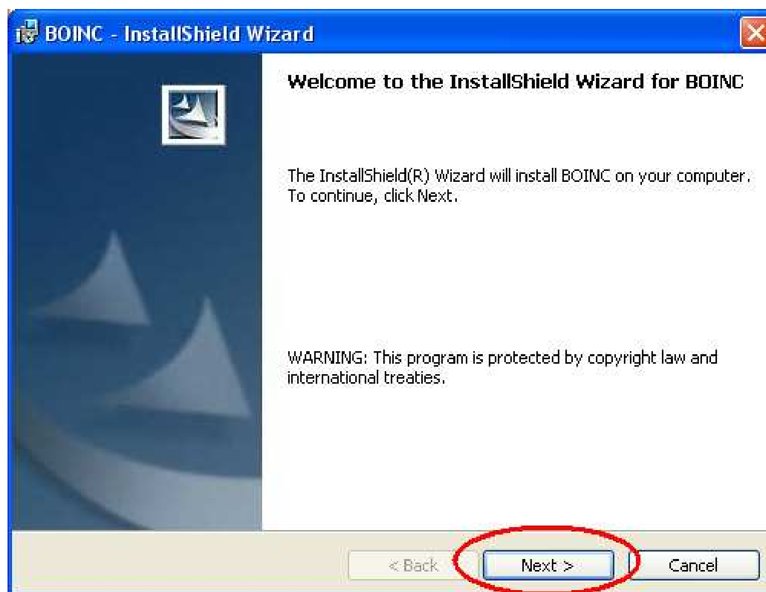
Táto kapitola súži ako návod pre inštaláciu BOINC klienta na po íta .

5.1. Stiahnutie BOINC klienta z internetu

Pred samotnou inštaláciou BOINC klienta na po íta je potrebné si stiahnu BOINC klienta. Tento je vo ne dostupný na internetovej stránke http://boinc.berkeley.edu/download.php.

5.2. Inštalácia BOINC klienta:

1. krok: spustenie stiahnutého exe súboru
2. krok: kliknú na tla idlo „Next >“ pre pokra ovanie inštalácie (Obr. 45)

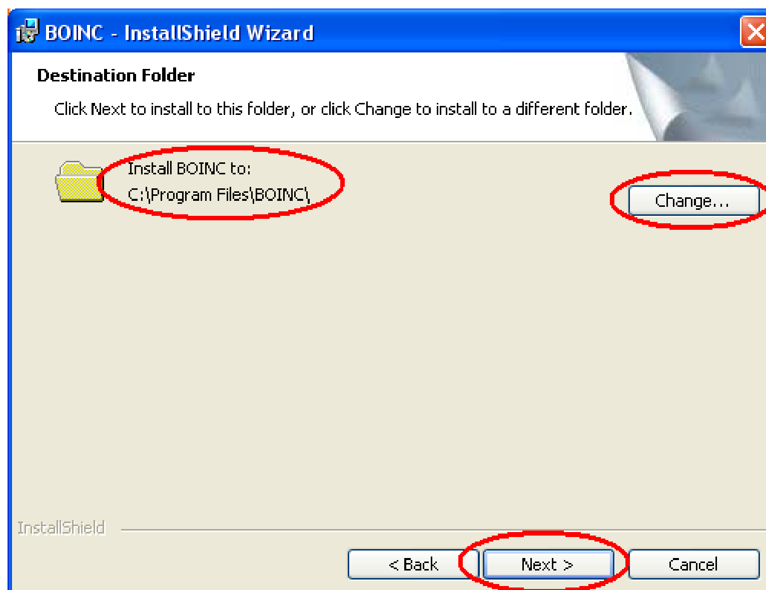


Obr. 45: 2. krok inštalácie BOINC klienta

3. krok: ozna enie 'I accept the terms in the license agreement”. Kliknú na tla idlo „Next >“ pre pokra ovanie inštalácie. (Obr. 46)

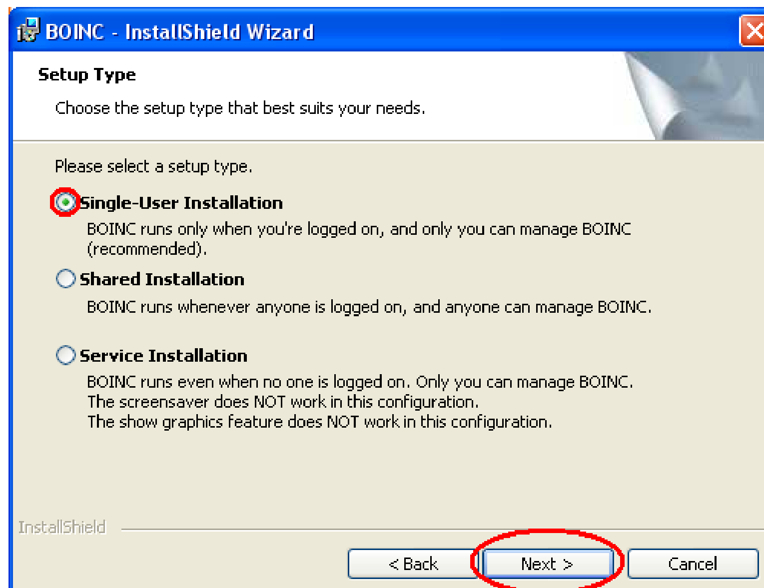
Obr. 46: 3. krok inštalácie BOINC klienta

4. krok: v tomto kroku je možné vybra kam má by klient nainštalovaný. V našom prípade sme ponechali prednastavenú vo bu. Umiestnenie sa dá zmeni po kliknutí na tla idlo “Change”. Kliknú na tla idlo „Next >“ pre pokra ovanie inštalácie. (Obr. 47)



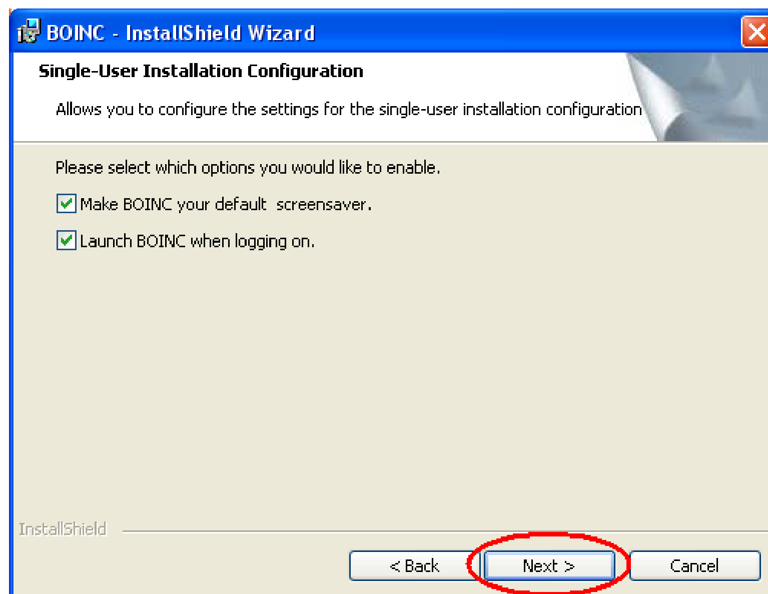
Obr. 47: 4. krok inštalácie BOINC klienta

5. krok: ozna enie „Single-User Installation“. Kliknú na tla idlo „Next >“ pre pokra ovanie inštalácie. (Obr. 48)



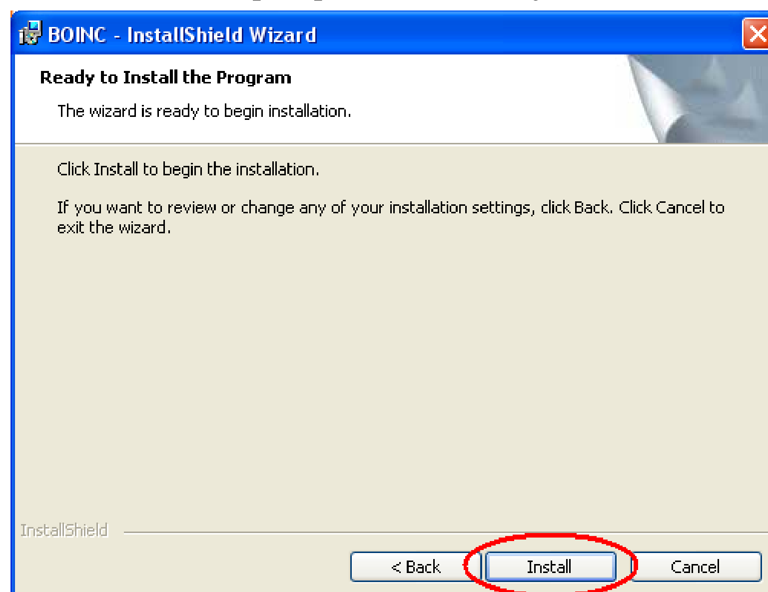
Obr. 48: 5. krok inštalácie BOINC klienta

6. krok: ponechanie prednastaveného nastavenia. Kliknútím na tla idlo „Next >“ pre pokra ovanie v inštalácii. (Obr. 49)



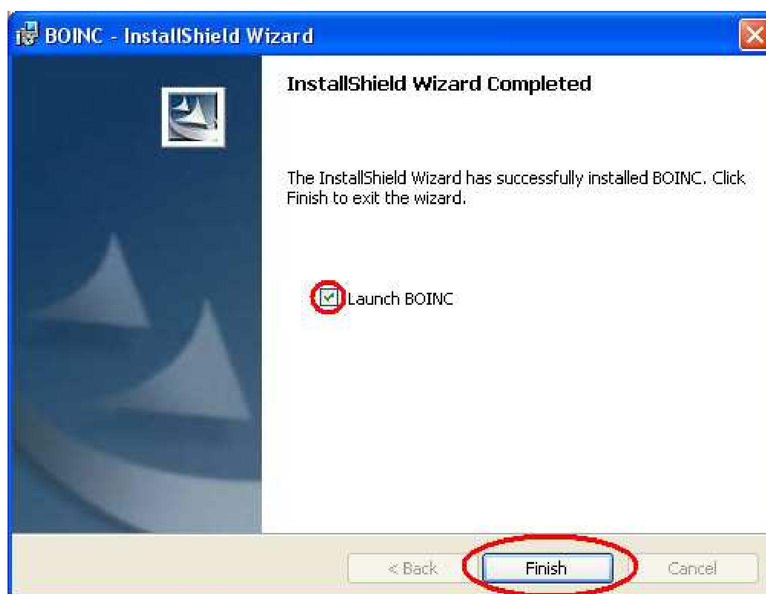
Obr. 49: .krok inštalácie BOINC klienta

7. krok: kliknú na tla idlo “Install” pre spustenie samotnej inštalácie klienta. (Obr. 50)



Obr. 50: 7. krok inštalácie BOINC klienta

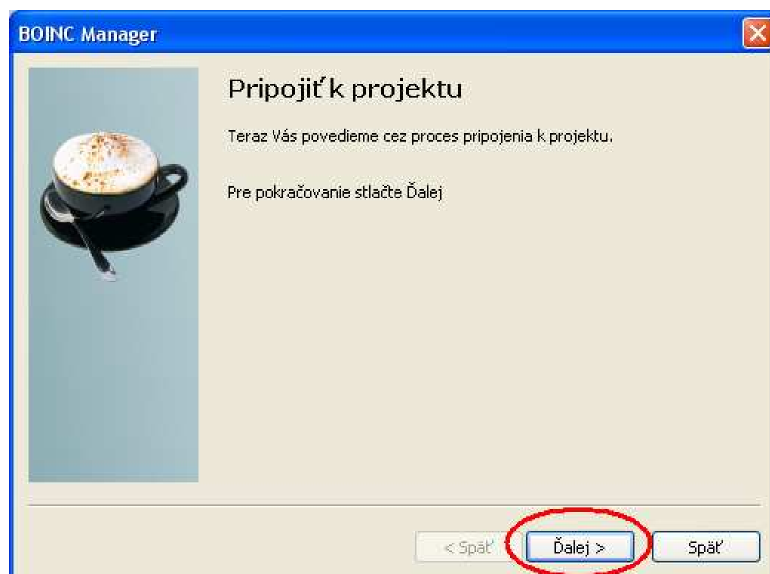
8. krok: ponechanie zaškrtnutia polí ka “Launch BOINC”. Dôvod pre o ho necháme zaškrtnuté je ten, aby sa nám klient hne spustil.Kliknú na tla idlo “Finish” pre ukon enie inštalácie klienta. (Obr. 51)



Obr. 51: 8. krok inštalácie BOINC klienta

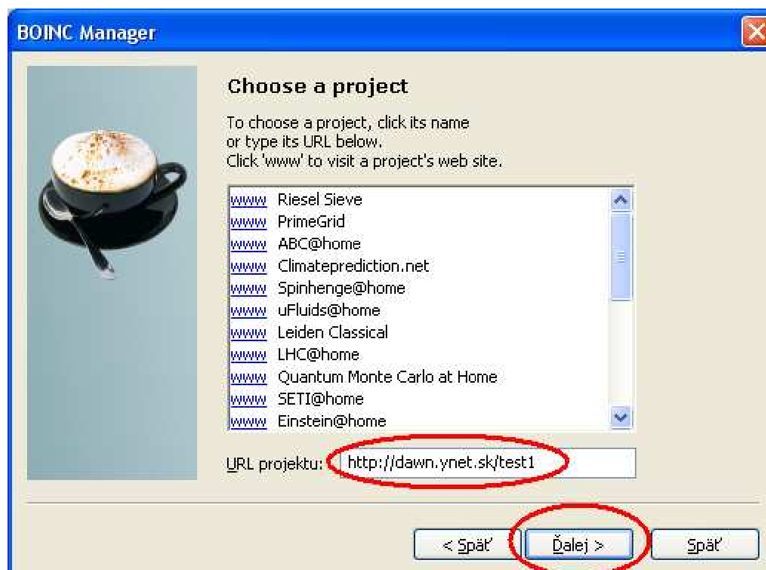
5.3. Pripojenie BOINC klienta k projektu:

1. krok: Kliknú " alej" pre pokra ovanie.(Obr. 52)



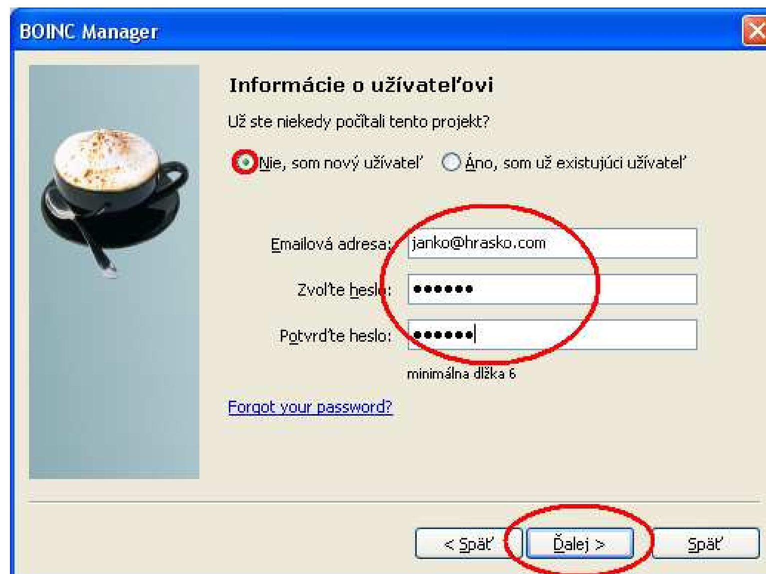
Obr. 52: 1. krok pripojenie BOINC klienta k projektu

2. krok: do polí ka "URL projektu" je potrebné zada http://dawn.ynet.sk/test1. Kliknú " alej" pre pokra ovanie.(Obr. 53)



Obr. 53: 2. krok pripojenie BOINC klienta k projektu

3. krok: v tomto kroku je potrebné vyplni polia "Emailová adresa, Zvo te heslo, Potvr te heslo". Po zadaní potrebných údajov sa pokra uje stla ením tla idla " alej"(Obr. 54)



Obr. 54: 3. krok pripojenie BOINC klienta k projektu

4. krok: stla i tla idlo "Dokon i ". Týmto je proces pridávania projektu ukon ený. (Obr. 55)

Obr. 55: 4. krok pripojenie BOINC klienta k projektu

6 Popis adresárovej štruktúry

V tejto kapitole popíšem hlavné adresáre v BOINC serveri a význam niektorých súborov v týchto adresároch. Popis je dôležitý hlavne pre developera projektu pre BOINC platformu, ale i pre administrátora, ktorý sa o beh BOINC serveru stará.

Adresár / súbor	**Význam**
/boinc/test1	Kore ový adresár pre BOINC projekt. Obsahuje samotný projekt a súbory potrebné pre jeho beh.
/boinc/test1/apps	Adresár s aplikáciami pre klientov. Obsahuje podadresáre so skrátenými menami klientskych aplikácií, v ktorých sa nachádzajú samotné klientske aplikácie a všetky ich verzie.
/boinc/test1/download	Adresár obsahujúci súbory, ktoré je možné stiahnu cez http protokol, ke projekt beží. Nachádza sa tu napríklad ikonka projektu, klientske aplikácie a adresáre, ktorých mená sú heše úloh, ktoré sú v nich obsiahnuté. Teda obsahuje i samotné súbory s úlohami.
/boinc/test1/templates	XML súbory s informáciami o úlohách (wokrunit-och) a výsledkoch (result-och). Teda hlavne zoznamy súborov, ktoré prezentujú daný objekt.
templates/reversi6_result	XML súbor obsahujúci informácie o tom, ktoré súbory sa majú uploadnú po ukon ení práce, definuje napríklad aj limit ve kosti súboru.
templates/reversi6_wu.xml	Obsahuje popis súborov, ktoré tvoria úlohu (workunit) a napríklad i pomocné príkazy k spusteniu klientskej aplikácie.

7. Administrátorská konzola

Administrátorská konzola projektu BOINC sa ovláda cez web rozhranie. Jej stránka je rovnaká ako stránka projektu, avšak na konci je prípona „_ops“. Teda ak sa Váš projekt nachádza na stránke http://dawn.ynet.sk/test1, tak sa administrátorská konzola (project management konzola) nachádza na adrese http://dawn.ynet.sk/test1_ops/. Na nasledujúcom obrázku je ukážka tejto základnej stránky.

Test Project: Project Management

- Currently used SVN revision: 13835 ; Latest SVN revision:
- There are 0 remaining candidates for User of the Day.

Browse database:

- Platforms
- Applications
- Application versions
- Users
- Teams
- Hosts
- Workunits
- Results

Regular Operations:

- Screen user profiles
- Manage special users

Special Operations:

- Manage applications
- Manage application versions
- Send mass email to a selected set of users
- Email user with misconfigured host
- FLOP count statistics
- Cancel workunits
- Manage user ID:
- Clear Host: Clear RPC

Obr. 56: Hlavná stránka administrátorskej konzoly

Z hlavnej stránky sú dôležité hlavne stránky „Workunits“ a „Results“, v ktorých je poskytnutá možnos prezerania si úloh a ich podrobných stavov a výsledkov a ich stavov. Je dôležité si uvedomi , že výsledok pre úlohu vzniká pri vytvorení úlohy.

Stránka s úlohami sa najprv spýta na to, aké úlohy chceme vidie . Ak všetky, sta í kliknú naOK a to nás presunie na nasledujúcu stránku zobrazujúcu úlohy v databáze BOINC serveru.

Test Project: Workunits

Query: **select * from workunit order by id desc limit 20**

406 records match the query. Displaying 1 to 20.

Next 20

More detail | Return to main admin page

WU ID	canonical result	error_mask	file delete	assimilate
16063	33416 all		Deleted	Assimilated
16062	33414 all		Deleted	Assimilated
16061	33412 all		Deleted	Assimilated
16060	33410 all		Deleted	Assimilated

Obr. 57: Administrátorská konzola – podstránka workunits

Zobrazené úlohy je možné prezera podrobnejšie. Po et podrobných informácií zah a napríklad po ty výsledkov a ich stavy, po ty výsledkov, ktoré sa majú generova , aplikácia, ktorou sa majú úlohy spracováva a podobne.

V asti „Results“ je opä na výber filter, ktoré výsledky chceme vidie . Po odkliknutí tla idla OK sa zobrazia všetky výsledky, ktoré sú v databáze BOINC serveru.

Test Project: Results

Query: **select * from result limit 20**

812 records match the query. Displaying 1 to 20.

Next 20

Summary | More detail | Return to main admin page

result ID	WU ID	server state	outcome	validate state	delete state	exit status	host (user)	app ver	received or deadline or created	CPU hours	claime[d] credit
32606	15658	Over	Success	Valid	Deleted	0	23 (Administrator)	105	26 Apr 2008 9:23:39 UTC	0.0	0.015
32607	15658	Over	Success	Valid	Deleted	0	23	105	26 Apr 2008	0.0	0.016

Obr. 58: Administrátorská konzola – podstránka results

Prezretie podrobností o výsledkoch je opä možné kliknutím na príslušné „result ID“.V podrobnostiach je okrem stavu výsledku uvedený i CPU as strávený výpo tom, error výstup klientskej aplikácie, verzia klientskej aplikácie, ktorá výsledok vyprodukovala, po et kreditov pridelených za výpo et a podobne.

Administrátorská konzola je silným nástrojom a hoci je tu popísaný iba jej zlomok, nie je problém sa nau i používa viaceré jej funkcie. Je intuitívna a poskytuje ucelený preh ad o aktuálnom stave projektu. Pri práci na tímovom projekte sme ju vo fáze testovania generátoru úloh (workgenerator-u) a asimilátoru ozna ili ako nepostrádate nú. Cez toto web rozhranie je jednoduché zisti , aké chyby vracajú klientske aplikácie pri spracovávaní úloh, ko ko úloh je vypo ítaných, prípadne iné podrobnosti.

Príloha D – Technická príru ka

1 Implementácia novej hry

Ak chceme vytvorený systém použi na riešenie inej hry, musíme namiesto game.h a game.cpp poskytnú vlastnú implementáciu.

Vyžaduje si to úpravu dvoch základných štruktúr:

- Game – predstavuje stav hracej plochy hry
- Move – predstavuje ah, ktorého aplikovanie mení stav hracej plochy

To, o presne tieto štruktúry obsahujú, záleží na tej ktorej konkrétnej hre. Štruktúra *Game* by ale mala obsahova minimálne:

- rozloženie figúrok na hracej ploche
- informáciu o tom, kto je na ahu

Štruktúra Move musí obsahova :

- Move *next - smerník na alšiu štruktúru Move, kvôli zre azeniu ahov do zoznamu

alej musíme poskytnú implementácie nasledujúcich funkcií, ktoré sa používajú pri generovaní a ohodnocovaní stromu odvodenia. Popíšeme si iba ich význam, ich prototypy môžeme nájs v referen nej príru ke. Teda funkcie tvoriace rozhranie sú:

- initGame – vytvorí po iato ný stav hracej plochy
- getLegalMoves – získa spájaný zoznam ahov, ktoré je možné z poskytnutého stavu vykona
- makeMove – vykoná ah na poskytnutom stave
- getGame – vykoná ah z poskytnutého stavu, no alokuje nový stav hracej plochy, ktorý vráti
- freeMoves – dealokuje zoznam ahov
- areGamesEqual – kontroluje, i sú hracie plochy rovnaké. Hrá , ktorý z nich vykonáva ah sa neberie do úvahy. Používa sa v situácii, ke hrá vynechá svoj ah. Kontroluje sa ou, i aj protihrá v predchádzajúcom ahu nevynechal ah z presne rovnakej hracej plochy. Takáto situácia znamená totižto koniec hry.
- cloneGameWithOtherPlayer – naklonuje hraciu plochu, no nastaví, že na ahu je druhý hrá . Použije sa v prípade, ak hrá nemôže aha . Zrovnakej hracej plochy ahá v alšom kole jeho protihrá .
- pre ažený operator < nad štruktúrou Game – potrebný pre ú ely ukladania štruktúry Game do kolekcií ako napr. do mapy použitej v transpozi nej tabu ke. Pri porovnaní sa musí bra do úvahy rozostavenie figúrok na hracej ploche aj informácia o tom, kto je na ahu. Je potrebné situácie, ktoré môžu po as hry nasta , úplne odlíši .

- gameEvaluation – pre hraciu plochu, ktorá predstavuje koncový stav hry, ur í ktorý z hrá ov vyhral. To, vzh adom na ktorého hrá a, sa ohodnotenie po íta preberá funkcia ako parameter.

V prípade použitia heuristiky je potrebné poskytnú heuristickú funkciu, najlepšie vo zvláštnom súbore, napr. *heuristic_my_alg.cpp*,

```
double heuristicEvaluation(Node *node)
```

ktorá bude ohodnocova hraciu plochu novej hry.

Taktiež je potrebné vytvori prispôsobenú implementáciu funkcií z *game_io.h* tak, aby sa cez vstupno/výstupné súbory prenášalo medzi klientom a serverom všetko, o je potrebné na reprezentáciu hracej plochy a výsledku. Jedná sa o tieto funkcie:

- writeGameToFile – generátor úloh zapíše do súboru hraciu plochu a informáciou o tom, kto je na ahu
- readGameFromFile - klient si pre íta zo súboru hraciu plochu a kto je na ahu
- writeNodeToFile - klient zapíše hraciu plochu, kto je na ahu a kto z danej pozície vyhrá
- readNodeFromFile - asimilátor si zo súboru pre íta hraciu plochu, kto bol na ahu a kto vyhral

Poskytnutím implementácie týchto funkcií sa zabezpe í, aby zvyšok kódu vedel spolupracova s novou hrou.

1.1 Kompilovanie

Výsledné spojenie implementácie novej hry s pôvodný kódom ohodnocovania stromu preh adávania a klientom sa deje v ase linkovania. Teda po kompilácii jednotlivých zdrojových súborov linkujeme nové objektové súbory prislúchajúce k vlastnej implementácii *game.cpp, heuristic_my_alg.cpp* a game_io.cpp so zvyšnými z pôvodnej implementácie.

Pre ilustráciu si ukážeme as *makefile* súboru, ktorý vytvorí klienta pre novú hru.

```
client_my: search_negascout.o heuristic_my_alg.o heuristic.o
game.o node.o search.o game_io.o common.o client.o $
(BOINC_API_DIR)/libboinc_api.a $(BOINC_LIB_DIR)/libboinc.a

     $(CXX) $(CXXFLAGS) -Wl,--export-dynamic -o client_my
search_negascout.o heuristic_my_alg.o heuristic.o node.o game.o
game_io.o search.o common.o client.o libstdc++.a -pthread
-lboinc_api -lboinc -ldl

heuristic_my_alg.o: heuristic_my_alg.cpp

     $(CXX) $(CXXFLAGS) -c -o heuristic_my_alg.o
heuristic_my_alg.cpp

game_io.o: game_io.cpp

     $(CXX) $(CXXFLAGS) -c -o game_io.o game_io.cpp

game.o: game.cpp
```

```
     $(CXX) $(CXXFLAGS) -c -o game.o game.cpp
```

1.2 Jednoduchý príklad

V tejto asti si najednoduchom príklade ukážeme štruktúry *Game* a *Move* spolu s implementáciou nieko ko hlavných funkcií. Uvažujme hru piškvorky na ploche NxN.

Štruktúry *Game* a *Move* by vyzerali nasledovne:

```
struct Game {
     int field[N][N];    // rozloženie kame ov na hracej ploche
     int playerOnMove;   // ur enie, kto je na  ahu
     bool operator< (const Game& rhs) const; // porovnanie plôch
}

struct Move {
     int x;    // x-ová súradnica polí ka, na ktoré položíme kame
     int y;    // y-ová súradnica polí ka, na ktoré položíme kame
     Move *next;    //aby sme zabezpe ili spájaný zoznam
}
```

Funkcia *initGame* by v našom prípade nastavila všetky poli ka na prázdne.

```
void initGame(Game *game) {
     int i, j;
     for (i = 0; i < N; ++i) {
          for (j = 0; j < N; ++j) {
               game->field[i][j] = 0;
          }
     }
}
```

Funkcia *makeMove* by zazna ila, že na polí ku z ahu *move* sa nachádza kame hrá a, ktorý je na ahu. Tiež by nastavila ah na druhého hrá a.

```
void makeMove(Game *game, Move *move) {
     game->field[move->x][move->y] = game->playerOnMove;
     if(game->secondMove == 1) {
          game->secondMove = 2;
     } else {
          game->secondMove = 1;
```

```
    }
}
```

Funkcia *getLegalMoves* by vytvorila zoznam platných ahov. V našom prípade je platný ah taký, ktorý sa snaží položi kame navo né polí ko.

```
Move *getLegalMoves(Game *game) {
    Move *result = NULL;
    int numMoves = 0;
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            if(game->field[i][j] == 0) {
                Move *move = (Move*) malloc(sizeof(struct
Move));
                move->x = i;
                move->y = j;

                move->next = result;
                result = move;
            }
        }
    }
    return result;
}
```

2 Implementácia algoritmu preh adávania

V tejto asti si povieme, o musíme spravi , ak chceme implementova iný algoritmus na riešenie hry, teda na spracovanie stromu preh adávania.

Vytvoríme si nový súbor, povedzme *search_my_alg.cpp*. V tomto súbore musíme implementova metódu

```
int gameSolvingAlgoritmEntrancePoint(Node* node, void
(*callback_fraction_done)(double))
```

deklarovanú v *search.h*, ktorá slúži ako vstupný bod spúštania algoritmu preh adávania. Má dva parametre:

- node predstavuje kore ový uzol stromu preh adávania
- funkcia callback_fraction_done slúži na nastavovanie, na ko ko máme hotovú prácu (spracovaný celý strom)

Návratová hodnota zase udáva, ktorý z hrá ov je ví azom:

- 1 – vyhrá prvý hrá
- 0 – remíza
- -1 – vyhrá druhý hrá

Popísanou funkciou získa náš kód riadenie od klienta, alej je už celá práca generovania a ohodnotenia stromu na nás.

2.1 Generovanie stromu preh adávania

Pri generovaní stromu nám pomáhajú funkcie zo *node.h* a *heuristic.h.* Pre úplný zoznam pozri referen nú príru ku. My si ukážeme použitie na typickom príklade generovania stromu.

```
void depthSearch(Node *node) {
     Node *child = expandNode(node);
     Node *actual;
     Node *last;

     if(child != NULL) {           // ak node nie je list
          for(actual = child; actual != NULL; ) {

               depthSearch(actual);

               last = actual;
               actual = actual->next;

               free(last->game);
               free(last);
          }
          resetNodeInternals(node);
     }
}
```

Za zmienku stoja dve hlavné funkcie:

- expandNode – vracia deti uzla. Pri jej použití si treba ale uvedomi , že funkcia alokuje pamä pre uzly a hracie plochy, ktoré vracia. Preto je po spracovaní uzlov potrebné zavola funkciu *free* na uzly samotné (*free(actual)*), aj na ich hracie plochy (*free(actual->game)*).
- resetNodeInternals - „vynuluje interné premenné uzla“, ku ktorým patrí napríklad smerník na deti uzla. Použitie tejto funkcie po skon ení práce suzlom je potrebné v prípade algoritmov, ktoré môžu viackrát po sebe spusti algoritmus preh adávania nad tou istou štruktúrou *Node*. Príkladom takých algoritmov je *negascout* a *mtd(f)*.

2.2 Použitie heuristík

Výkonnos algoritmov preh adávania založených na usekávaní záleží od zoradenia uzlov pri prechádzaní stromom. Pre ú el zoradenia uzlov na základe heuristiky môžeme použi funkciu z *heuristic.h*

```
Node *sortNodesAccordingToHeuristic(Node *list, int length)
```

kde *list* je pôvodný zoznam uzlov, *length* je po et prvkov v zozname. Funkcia vracia zoznam uzlov zoradený pod a heuristiky. Tento zoznam nie je potrebné uvo ova , pretože ho tvoria pôvodné štruktúry zo zoznamu *list*, len so zmenenými smerníkmi na nasledujúci prvok. Ako heuristická funkcia sa použije funkcia *heuristicEvaluation* deklarovaná v *heuristic.h*. Môžeme implementova heuristickú funkciu pre konkrétnu hru alebo jednoducho vola funkciu heuristiky založenej na vražedných ahoch:

```
int getKillerHeuristicScore(Game *game)
```

Pridávanie skóre pre heuristiku vražedného ahu sa realizuje funkciou

```
void addKillerHeuristicScore(Game *game, int depth)
```

kde *game* je hracia plocha, v aka ktorej k useknutiu došlo a *depth* je h bka, v ktorej sa useklo.

Ak sa rozhodneme použi vlastnú funkciu, je vhodné ju vloži do samostatného súboru, napr. *heuristic_my_alg.cpp*.

2.3 Použitie transpozi nej tabu ky

Pri spracovaní stromu preh adávania môžeme použi techniku transpozi nej tabu ky, ktorá je prístupná funkciami zo *search.h*:

```
void addToTable(Game *game, int result);
int getTranspositionTableResult(Game *game);
```

2.4 Integrácia ukladania stavu do boinc systému

Konkrétny spôsob realizácie ukladania stavu výpo tu je na tvorcovi klienta. Boinc cez svoje API funkcie poskytuje mechanizmus oznamovania, i je už potrebné vykona uloženie. Presný bod, v ktorom sa uloženie vykoná, je ale na vývojárovi. Boinc API taktiež zabezpe uje vzájomné vylú enie, aby nebol klient vypnutý po as ukladania stavu. Využívajú sa nasledovné funkcie:

```
int boinc_time_to_checkpoint();
```

Ak je výsledok tohto volania rôzny od nuly, je potrebné vykona uloženie stavu výpo tu

```
boinc_checkpoint_completed();
```

Túto funkciu je potrebné zavola po úspešnom uložení stavu.

Do algoritmu výpo tu, v mieste v ktorom vieme uloži stav výpo tu, je teda potrebné vloži nasledovnú schému:

```
if(boinc_time_to_checkpoint() != 0) {
```

```
    //konkretny kod na ulozenie stavu vypoctu
    boinc_checkpoint_completed();
}
```

2.5 Kompilovanie

Výsledné spojenie vlastnej implementácie ohodnocovania stromu preh adávania s pôvodným kódom sa deje v ase linkovania. Teda po kompilácii jednotlivých zdrojových súborov linkujeme nové objektové súbory prislúchajúce k vlastnej implementácii *search_my_alg.cpp* a *heuristic_my_alg.cpp* so zvyškom klientského kódu.

Pre ilustráciu si ukážeme as *makefile* súboru, ktorý vytvorí klienta s použitím vlastnej implementácie heuristiky aj algoritmu spracovania stromu.

```
client_my: search_my_alg.o heuristic_my_alg.o heuristic.o game.o
node.o search.o game_io.o common.o client.o $
(BOINC_API_DIR)/libboinc_api.a $(BOINC_LIB_DIR)/libboinc.a

    $(CXX) $(CXXFLAGS) -Wl,--export-dynamic -o client_my
search_my_alg.o heuristic_my_alg.o heuristic.o node.o game.o
game_io.o search.o common.o client.o libstdc++.a -pthread
-lboinc_api -lboinc -ldl

heuristic_my_alg.o: heuristic_my_alg.cpp

    $(CXX) $(CXXFLAGS) -c -o heuristic_my_alg.o
heuristic_my_alg.cpp

search_my_alg.o: search_my_alg.cpp

    $(CXX) $(CXXFLAGS) -c -o search_my_alg.o search_my_alg.cpp
```

3 Testovanie klienta a algoritmov

Na to aby sme mohli otestova správnos práce klientov sme naprogramovali program, ktorý túto úlohu vykonáva. Pre jeho innos sú potrebné dva súbory, z ktorých íta potrebné dáta pre vykonanie samotného testovania. Sú to súbory testy a test.txt.

Súbor testy obsahuje cesty k testovaným klientom. Jeho štruktúra je nasledovná. Najskôr je uvedený po et testovaných klientov vyjadrený íslom. Za ním nasledujú cesty k testovaným klientom.

Súbor test.txt obsahuje na za iatku dve ísla. Prvé udáva po et testovacích konfigurácií hracej plochy, druhé íslo udáva rozmer hracej plochy. Za týmito dvomi íslami nasledujú samotné testovacie konfigurácie hracej plochy. Zadávajú sa v rovnakom formáte ako je súbor in.

Zdrojový kód testovacieho programu je test.c. Pre skompilovanie je potrebné zada príkaz make. Po vykonaní príkazu make máme vytvorený spustite ný súbor. Jeho meno je test. Pred samotným spustením testovania klientov sa musíme uisti , že máme v jednom adresári súbory testy, test.txt a test. Pre spustenie testovania klientov zadáme do konzoly príkaz ./test. Výsledok z testov je v súbore výsledky.

Súbor výsledky obsahuje výsledok každého z klientov na každú testovaciu konfiguráciu spolu aj s asom trvania výpo tu.

4 Testovanie asimilátora

Ke že generátoru sme pridali možnos vygenerova hru s ubovolného vstupného bodu (v priebehu hry), mohli sme jednoducho a hlavne rýchlo otestova asimilátor so skuto nými výsledkami vrátenými od klienta. Ak by sme chceli tieto výsledky od klientov nejako simulova museli by sme vola rôzne databázové skripty pre *Boinc* databázu, o by mohlo predstavova riziko v nekonzistentnosti.

Preto testy prebiehal vygenerovaním nieko ko WorkUnit-ov, následným " akaním" na spracovanie klientmi a potom sme mohli spusti asimilátor. Pre spustenie asimilátora je potrebné ho skompilova volaním príkazu

make cockroachAssimilator

v adresári zdrojových súborov: **boinc/sched**. Výsledný spustite ný súbor prekopírujeme do **bin** adresára projektu *boinc*. Spustenie asimilátora je možné uskuto ni shell príkazom:

./cockroachAssimilator -app reversi

prepína **-app** ur uje meno *boinc* aplikácie, ak chceme spusti aj *debug* výpisy tak pridáme alší prepína : **-d 3** (pre debug level 3, kde sa vypisujú aj informa né výpisy).

V hlavnom adresári *boinc* projektu sa nachádza súbor **config.xml**, ktorý predstavuje hlavnú konfiguráciu *boinc* projektu. Samotný asimilátor je spustený ako démon, takže beží v nekone nej slu ke a aká na volanie funkcie pri príchode výsledku od klienta. Fragment z **config.xml**, ktorým sa nastavia parametre pre asimilátor:

```
...
<daemon>
     <cmd>
          cockroachAssimilator -d 3 -app reversi
      </cmd>
</daemon>
...
```

spustite ný súbor cockroachAssimilator, sa musí nachádza v **bin** adresári *Boinc* projektu.

5 Testovanie databázového modulu

Po implementovaní databázového modulu bolo potrebné otestova , i jednotlivé funkcie, ktoré pristupujú k databáze, vykonávajú presne špecifikované operácie.

Pred testovaním sme si manuálne predpripravili as stromu v podobe databázových záznamov. Následne sme pripravili testovaciu funkciu, ktorá vytvorila zopár poprepájaných uzlov a postupne ich posielala na vstup každej funkcii zvláš . Prostredníctvom výpisu záznamov v databáze a pomocných výpisov funkcie sa nám podarilo postupne vyladi všetky databázové funkcie.

Pri testovaní databázového modulu sme odstránili nasledujúce problémy:

- nesprávna práca s alokovanou pamä ou, použitie premenných bez alokácie, dealokácia potrebných premenných pri chybových hláškach databázy
- opa ne implementovaná transformácia údajov v navzájom „inverzných“ metódach `getMove` a `setSecondMove`
- problémy spojené s prácou s uzlami, ktoré majú viac rodi ov, spomenuté v kapitole *Prepojenie uzlov*, kde je uvedený aj návrh ich riešenia

6 Testovanie generátoru úloh

Generátor úloh (workgenerator) je aplikácia, ktorá sa spúš a jednorazovo. Jej úlohou je vytvori na BOINC serveri úlohy pre klientov a v prípade potreby vytvori i úlohy v externom úložisku projektu. Workgenerator v tomto prípade generuje strom hry do zadanej h bky a listy tohto stromu ukladá do BOINC serveru ako úlohy pripravené na odoslanie a všetky uzly cez ktoré prešiel a aj listy, ukladá do externej databázy. Viac informácií nájdete v príru ke ktejto aplikácii.

Testovanie workgenerator-u vo fáze ladenia bolo nutné a žiadané. Pri zmene hry nebude potrebné túto aplikáciu preprogramova , avšak jej rekompilácia a testovanie bude nutnos ou. Testovanie spo íva vdvoch fázach. Jedna testuje spoluprácu s knižnicou externej databázy a druhá fáza testuje ukladanie úloh (workunit-ov) pre BOINC server.

6.1 Testovanie ukladania úloh do externej databázy

Testovanie ukladania prejdených uzlov do externej databázy preveruje spoluprácu s knižnicou **db_access**. Za iatok predpokladá prázdnu externú databázu. Pre vyskúšanie si ukladania na menšom probléme je potrebné si pripravi zoserializovaný uzol hry niekde asi 10 až 15 ahov pred koncom. Nasledujúce ukážky budem predvádza na hre reversi 6x6.

```
dawn root # cd /boinc/test1/bin/
dawn bin # echo "000010211100122210212110200000000000;1" >
vstupnySubor
```

Vstupný súbor máme pripravený. Teraz spustíme workgenerator s parametrami, ktoré zamedzia ukladaniu sa vygenerovaných úloh (workunit-ov) do BOINC serveru.

```
dawn bin # ./cockroachWorkGenerator -j 2 -b -f vstupnySubor
2008-04-28 14:48:59.1641 [normal  ] Setting new depth.
2008-04-28 14:48:59.1646 [normal  ] Setting saveToBoinc to false.
2008-04-28 14:48:59.1840 [normal  ] Starting
2008-04-28 14:48:59.1841 [normal  ] Príprava štruktúry wu- ka
2008-04-28 14:48:59.1844 [normal  ] node = stratGame(null)
2008-04-28 14:48:59.1844 [normal  ] Po iatok rekurzívneho volania
generovania stromu.
2008-04-28 14:48:59.5309 [normal  ] Rekurenté generovanie stromu
```

```
bolo ukon ené. (recordsInsertedToBoinc=0, recordsInsertedToDB=65,
recordsNotInsertedToDB=0, recordsDoubledDB=0)
```

Beh programu skon il úspešne a na záver sa vypísali po ty uložených uzlov. Teraz je potrebné skontrolova , i sa vložili správne uzly do databázy a i je i ich previazanos v strome správna. Najjednoduchším spôsobom je spusti program psql a zobrazi si v om obsah databázovej tabu ky reversi a tento obsah potom skontrolova za pomoci vizualizácie.

```
dawn bin # psql boinc postgres
Welcome to psql 8.0.12, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

boinc=# select * from reversi;
```

Po zadaní selekt príkazu v programe psql sa zobrazí zoznam záznamov v databáze. Tento prekontrolujeme a získané výsledky sú výsledkami funk nosti astí game, node a db_access v sú innosti s workgenerator-om.

7. Testovanie ukladania úloh pre BOINC server

Tento test má za úlohu vyskúša funk nos serializácie stavov hry nadisk a preveri komunikáciu generátoru úloh (workgenerator-u) s boinc knižnicami. Workgenerator si vygeneruje pre každú úlohu, ktorú má uloži na BOINC server jej meno (íslo udávajúce poradie) a pod týmto menom sa snaží zoserializova prislúchajúci uzol. Tento uzol takisto zaregistruje pre BOINC server do jeho databázy cez prislúchajúce funkcie. Pri registrácii je dôležité napríklad uvies rozumné ohrani enie úlohy na pamä a výpo tový výkon. Pri testovacích úlohách, ako sa generujú tu, posta uje obmedzenie na 1e11 fpops (floating point operations), o sa nabežných strojoch pohybuje v okolí minúty.

```
dawn bin # ./cockroachWorkGenerator -j 2 -p -f vstupnySubor
```

Spustenie je obdobné ako pri predošlom teste a aj priebeh je obdobný. Po ukon ení generovania úloh je na mieste skontrolova i sa serializované súbory s uzlami vytvorili a prípadne, i obsahujú správne dáta. Súbory sa ukladajú do adresáru */boinc/test1/download* a v om doadresáru s hešhom svojho mena. Kontrola takto uložených súborov môže by náro ná a preto je vhodné si súbory prekopírova do jedného spolo ného adresára vhodným príkazom, ako napríklad týmto:

```
dawn root # cd /boinc/bin/download
dawn download # mkdir spolocnyAdresar
dawn download # cp */* spolocnyAdresar
```

Kopírovanie môže trva dlhú dobu v prípade ve kého po tu súborov, ktorý nie je problém dosiahnu .

8. Zmena algoritmu preh adávania na serveri

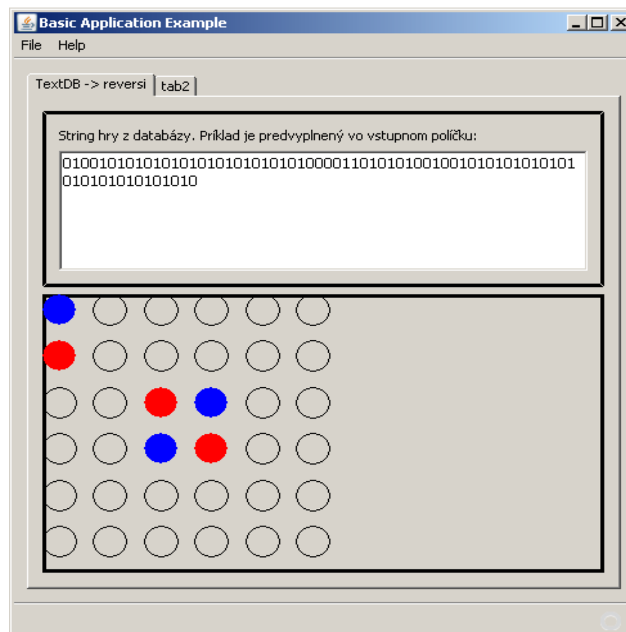
Algoritmus preh adávania na serveri je "skrytý" v jedinej funkcii (processGameResult() - ref). V podstate je to klasický minimax. Ak príde výsledok od klienta, iže ohodnotený uzol. Algoritmus má potom tieto kroky:

1. uloženie výsledku uzla do databázy
2. ak má uzol ohodnotených všetkých susedov, tak pokra ujeme alším krokom, inak koniec
3. rekurzívne vyhodnocujeme rodi a, pod a druhého bodu má ohodnotených všetkých potomkov, takže vieme aplikova klasický minimax algoritmus a pokra ujeme prvým krokom s novým uzlom.

iže zmena algoritmu na klientovi následne predstavuje zmenu spomenutej procedúry na žiadaný algoritmus.

9. Vizualizácia

Vizualizácia bola výbornou pomôckou pri ladení projektu. Využívala sa na vizualizovanie hracích plôch uložených v databáze. Na Obr. 59 vidie rozhranie vizualiza nej aplikácie.



Obr. 59: Ukážka vizualiza nej aplikácie k databáze

Vizualizácia bola nepostrádate ná pri ladení projektu. Slúži na grafické zobrazenie stavu šachovnici z databázy. Vizualizuje hru reversi ubovolných rozmerov. Do vrchnej asti sta í vloži informáciu z databázy o stave hry a stav hry sa zobrazí automaticky. Pri nesprávnom vstupe sa nezobrazí hracia plocha. Vizualizáciu sme používali v kombinácii s nástrojom psql, ktorý umož uje vCLI rozhraní listova tabu ky v databáze. Re azec so stavom hry sme skopírovali a vložili do polí ka vo vizualizácii.

Kódovanie stavov hry v databáze je binárne. Na každé polí ko sa využívajú dva bity. Postupnos dvoch núl znamená ervenú farbu, alias prvého hrá a. Postupnos 01 znamená hrá a modrého a postupnos 10 znamená prázdne polí ko. Postupnos 11 nie je povolená.