Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFOMAČNÝCH TECHNOLÓGIÍ

Odbor: Softvérové inžinierstvo

Distribuované riešenie symetrickej hry

Tímový projekt

Tím číslo 13: **Švábi**

Členovia tímu: Bc. Matúš Svrček,
Bc. Alexander Šimko,
Bc. Michal Štekláč,
Bc. Miroslav Štolc,
Bc. Jaroslav Tešlár,
Bc. Ľubomír Varga

Vedúci tímu: Ing. Peter Lacko

Školský rok: 2007/08

Zadanie

Od vzniku počítačovej vedy sa ľudia snažili vytvoriť programy, ktoré by hrali hry lepšie ako ľudia. V niektorých hrách sa podarilo vytvoriť programy hrajúce na majstrovskej úrovni, ale niektoré hry zostávajú pre počítače stále neriešiteľné.

Cieľom projektu bude vytvoriť s pomocou technológie BOINC (Berkeley Open Infrastructure for Network Computing) systém, ktorý bude schopný zistiť aspoň ultra slabé (prípadne slabé) riešenie skúmanej symetrickej hry. Ultra slabé riešenie nám dáva odpoveď na otázku, či prvý hráč vyhrá, prehrá alebo remizuje, ak bude ťahať najlepšie ako sa dá.

Výsledný produkt bude obsahovať dve časti (server a klient). Server projektu, ktorý bude rozdeľovať úlohy pre klientov a zbierať ich výsledky. Server taktiež poskytuje globálne štatistiky projektu a individuálne štatistiky pre používateľov, riešiacich daný problém. Klientska aplikácia, ktorá vykonáva požadované výpočty na počítači používateľa, ktorý sa rozhodol darovať svoju výpočtovú silu projektu.

Funkčnosť systému overte na hre reversi 8x8 a GO 7x7.

I Dokumentácia Projektu

Obsah

1 Úvod

Ľudia sa venujú hraniu hier oddávna. Ich motívy sú rôzne. Či už je to hranie pre zábavu, pokorenie kamaráta alebo vidina miliónových ziskov v Monte Carle, hry zaujímajú ľudskú pozornosť stále.

Deterministické stolové hry predstavujú samostatnú kapitolu. Človek sa nemusí namáhať, nespotí sa, ani si nezlomí nohu. Môže sedieť v teple svojej izby, hútať, namáhať svoje mozgové závity a ozlomkrky ťahať figúrkami alebo kameňmi po hracej ploche. Stačí mu iba jeho intelekt.

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

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

My sme túto výzvu vrámci predmetu Tvorba softvérového systému v tíme prijami a v tomto dokumente opisujeme pokus o hľadanie 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, častiach a fungovaní. V kapitole štyri popisujeme hry reversi a GO, ich pravidlá a základné stratégie hry. Kapitola päť poskytuje nevyhnutné teoretické záležitosti ohľadom riešenia hier, rozoberá vybrané algoritmy, heuristiky a vylepšujúce techniky. Existujúce riešenia hier reversi a GO o menších rozmeroch rozoberá kapitola šesť. V kapitole sedem analyzujeme rôzne možnosti ukladania stromu prehľadávania. Kapitola osem sa venuje hrubému návrhu systému a v kapitole deväť popisujeme vytvorené prototypy. Príloha A obsahuje slovník pojmov a príloha B notáciu použitú v kapitole osem.

2 Špecifikácia zadania

Cieľom projektu je za pomoci technológie BOINC vytvoriť systém, ktorý bude schopný dostať ako výstup ultra slabé riešenie symetrickej hry. My budeme riešiť symetrické hry reversi 8x8 a GO 7x7. Snahou je vytvoriť taký systém, aby bol čo najviac nezávislý na hre. To znamená, aby bolo možné v ňom riešiť aj iné symetrické hry, ako je napr. šach a pod.

Systém bude obsahovať dve hlavné časti:

1) server
2) klient

Aby bolo možné pracovať na tomto projekte aj v nasledujúcich rokoch, je potrebné, aby vytvorené zdrojové kódy boli prehľadné, modulárne a pochopiteľné.

Ďalšou dôležitou častou projektu je dokumentácia. Je ju potrebné vytvárať zodpovedne a nemôže byť zanedbaná.

2.1 Server

V systéme bude jeden server. Jeho úlohami sú:

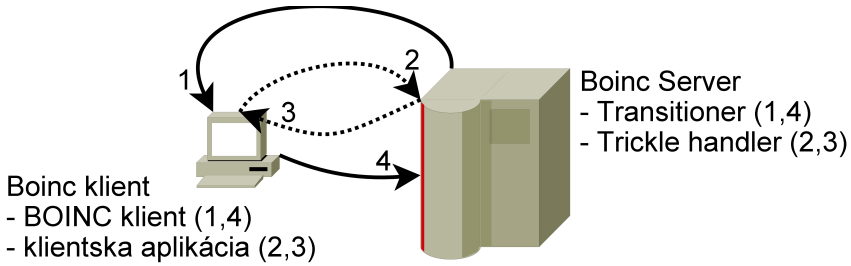
1) rozdelenie hlavnej úlohy na čiastkové podúlohy
2) rozdelenie čiastkových podúloh medzi klientov
3) zabezpečenie komunikácie medzi serverom a klientmi
4) zber čiastkových úloh od klientov
5) sumarizácia čiastkových podúloh
6) spočítanie celkového riešenia
7) vytvorenie štatistík

2.2 Klient

Jeho úlohou bude riešiť čiastkové podúlohy, ktoré dostal zadané od servera. Klientov bude v našom systéme čo najviac – n. Čím viac klientov bude pracovať 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 umožňuje využívať voľný výkon dobrovoľníkov na časovo náročné výpočty. Jeho architektúra je typu klient – server. Na BOINC softvéri je postavených niekoľko desiatok výpočtovo náročných, vedeckých i menej vedeckých projektov. Jedným z najznámejších je SETI (Search for Extraterrestrial Intelligence), hľadanie mimozemskej inteligencie. Na Zemi je rozmiestnených niekoľko satelitov, jeden z nich však zachytáva signály prichádzajúce z kozmu a snaží sa v nich hľadať počin inteligencie. Zachytených signálov je obrovské množstvo a je ich snaha analyzovať všetky. Pre analýzu takého množstva dát by však bolo treba superpočítač, čo je drahá hračka, a tak vedci vyvinuli SETI@home. Vznikla možnosť pomôcť s výpočtami doslova zo svojho domu. Stiahnutím malej aplikácie, ktorá z času na čas stiahla dáta z laboratórií, spracovala ich a výsledok vrátila naspäť. Tento spôsob pomoci pri hľadaní mimozemskej inteligencie si našiel obrovské množstvo prívržencov a dobrovoľníkov, ktorí pomáhali. Tak vznikol základ pre platformu umožňujúcu zapriahnuť množstvo domácich počítačov do veľkých výpočtov. 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: Komunikačné 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 uskutočňovať náročné výpočty. Aplikácie však pre svoju činnosť 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 výpočtu (angl. result) a je uložený ako súbor. Tento výsledok následne pošle BOINC klient na projektový server.

Dobrovoľník pre zapojenie sa do pomoci vo výpočtoch nainštaluje BOINC klient aplikáciu a následne si v BOINC klientovi pridá projekty, na ktorých sa chce podieľať. Každý projekt má svoju stránku, na ktorej sa každý dobrovoľník môže pozrieť na svoje výsledky, koľko kreditov za ne získal a môže meniť svoj podiel v prispievaní danému projektu. Je teda možné podieľať sa

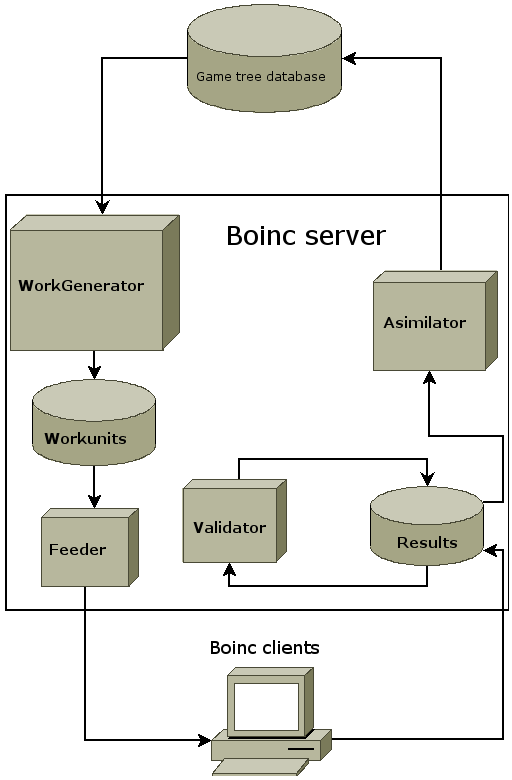
1 http://boinc.berkeley.edu/

na hľadaní proteínu proti malárii, HIV, ale zároveň je možné v užívateľom zadefinovanom pomere pomáhať v rendrovaní počítačových animácií, poprípade pomáhať v hľadaní gravitačných vĺn. Projektov pre BOINC platformu je aktuálne asi 40 a zaoberajú sa rôznymi oblasťami. Nájdu sa napríklad i problémy ako hľadanie minimálnej konfigurácie sudoku hry, z ktorej je možné získať iba jedno správne vyplnenie hracej plochy. Okrem herných, fyzikálnych, chemických a lekárskych projektov sa nájdu i počítačovo orientované.

3.1 Popis priebehu komunikácie serveru a klienta

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

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



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

3.2 Démoni bežiaci na serveri

Generátor úloh

Úloha generátoru úloh (angl. WorkGenerator) je jednoduchá. Na požiadanie vygenerovať daný počet úloh, uložiť ich na disk do adresára pre odoslanie klientom a každú takto vygenerovanú a uloženú úlohu zaregistrovať v projektovej databáze. V našom prípade bude generátor úloh používať pre generovanie nových úloh informácie z databázy stromu hry. Databáza stromu hry je vzhľadom na BOINC projekt externé úložisko aktuálneho stavu výpočtu. Generátor úloh je závislý na klientskej aplikácii, teda pre každú klientsku aplikáciu je potrebné mať jeden takýto generátor.

Rozdelovač úloh

Rozdelovač úloh (angl. Feeder) je démon, ktorý je implementovaný a nie je ho potrebné nahradzovať, alebo upravovať. Stará sa o rozdeľovanie úloh klientov, ktorí žiadajú o prácu. V našom projekte sa touto časťou BOINC softvéru nebudeme zaoberať. Štandardný rozdeľovač úloh je možné inštruovať k rôznemu poradiu odosielania úloh. Úlohy je možné posielať klientom podľa času vytvorenia, podľa priority alebo náhodne. Pri viacerých klientskych aplikáciách, teda viacero problémov riešených v rámci jedného projektu, je možné posielať úlohy buď v náhodnom poradí, čo 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 určuje počet klientov, ktorí majú počítať tú istú úlohu. Je to kvôli bezpečnosti. Klienti môžu z rôznych dôvodov falšovať vracané výsledky, a tak sa dáva tá istá úloha prepočítať viacerým klientom a výsledky sa porovnávajú. Porovnanie má na starosti práve validátor.

Po prijatí výsledku je úlohou validátoru vytvoriť alebo určiť takzvaný kanonický výsledok (angl. canonical result), ktorý zastupuje všetky výsledky. Tento následne použije asimilátor. Prijaté výsledky validátor porovná a určí kredit, ktorý sa pripíše na účet klientom, ktorí úlohu počítali. Kredit je možné získať dvoma spôsobmi. Z klientskych výsledkov vyčítať počet operácií, ktoré zhruba procesor vykonal pri výpočte (čas krát benchmark), alebo, ak generátor úloh takú informáciu k úlohe uložil, vyčítať kredit z úlohy prislúchajúcej k výsledku. Ak to výpočet dovoľuje, je lepšie určovať kredit pri tvorbe úlohy a teda pri validácii ho vyčítať z úlohy.

Asimilátor

Asimilátor (angl. Assimilator) je zodpovedný za spracovanie kanonického výsledku úlohy. Spracovanie z pohľadu 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 ukladať ohodnotenia stromu do externého úložiska stromu hry.

Iné súčasti 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 dokumentačnej stránke projektu BOINC[2].

3.3 Možnosti komunikácie klienta a serveru

Komunikácia je pri distribuovaných výpočtoch veľmi dôležitá časť výpočtu. Na superpočítačoch sa na komunikácii stratí i niekoľko desiatok percent výkonu. Záleží od paralelizovateľnosti problému, koľko komunikácie a v akom čase je potrebnej. Pre BOINC projekty, kde komunikácia prebieha cez internet, a nemôže sa spoliehať na stálu dostupnosť klientov, je možné počítať iba veľmi dobre paralelizovateľné úlohy. Je snaha spraviť BOINC platformu vhodnú i pre „low latency“ projekty, kde nie je dlhšiu dobu dostupná žiadna práca, ale keď je dostupná, musí sa spraviť rýchlo a teda je delená na menšie časti, aby sa lepšie využili aktuálne pripojené počítače.

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, čo si klient vyžiada úlohu. V danej úlohe je všetko potrebné pre aplikáciu v klientovi na to, aby splnila svoju výpočtovo náročnú úlohu. Druhou, novšou, možnosťou komunikácie od serveru ku klientovi je posielanie si správ cez rad správ. Keď sa klient z času na čas kontaktuje so serverom v priebehu výpočtu, server odovzdá klientovi správy, ktoré mu adresoval niektorý z démonov. Klientovi sa dá vnútiť pripájanie sa na server v daných časových intervaloch. Klient sa pripája s cieľom zistiť, koľko kreditov má daný užívateľ, a tiež je pripravený vyzdvihnúť si správu určenú pre klientsku aplikáciu. Tento spôsob komunikácie bol zavedený pre možnosť kontaktovania klientskej aplikácie počas výpočtu a oznámiť jej nové skutočnosti. Od zrušenia úlohy, cez odovzdanie nových poznatkov, ktoré urýchlia riešenie problému, až po oznámenie ukončenia projektu z dôvodu úspešného vyriešenia problému.

Komunikácia od klienta k serveru prebieha obdobne. Najstarším spôsobom je odoslanie hotovej úlohy, v ktorej klientska aplikácia navrhne počet kreditov za vypočítanú časť. Do výsledku okrem samotného výsledku uvedie strávený čas výpočtom a podobne. Tu je pre klientsku aplikáciu otvorený priestor na informovanie serveru o čomkoľvek. Tento systém sa pri projektoch, ktorých úloha trvala extrémne dlho (mesiace), ukázal ako nedostačujúci, a tak obdobne, ako i na serverovej strane, sa pridala možnosť asynchrónnej komunikácie klienta so serverom počas priebehu výpočtu. Klientska aplikácia môže odovzdať správu pre server klientovi a ak je to potrebné, vyžiadať si okamžité pripojenie sa na server a odovzdanie správy. Takto napríklad pri dlhých úlohách môže klientska aplikácia oznámiť serveru, že úloha sa počíta, a že má zmysel čakať na výsledok. Môžu sa i prideliť predčasne body za vypočítanú časť úlohy.

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

4 Hry

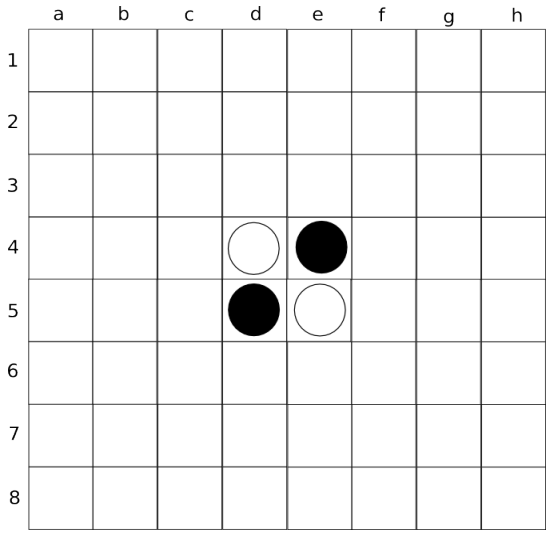
4.1 Symetrické hry

Symetrická hra je hra, v ktorej majú obidvaja hráči rovnaké – symetrické – postavenie. Obaja hráči sú rovnocenní, môžu robiť rovnaké ťahy vyplývajúce z presných pravidiel hry. Odlišujú sa len v tom, že jeden z nich je na ťahu ako prvý. Obaja hráči riešia rovnaký strategický cieľ – maximalizujú svoj zisk, a súčasne zabraňujú v tejto akcii súperovi, t.j. minimalizujú zisk súpera. Medzi známe a obľúbené symetrické hry patrí aj reversi a go, ktoré sú predmetom riešenia tohto projektu, a preto ich podrobnejšie popíšem v nasledujúcich kapitolách.

4.2 Reversi

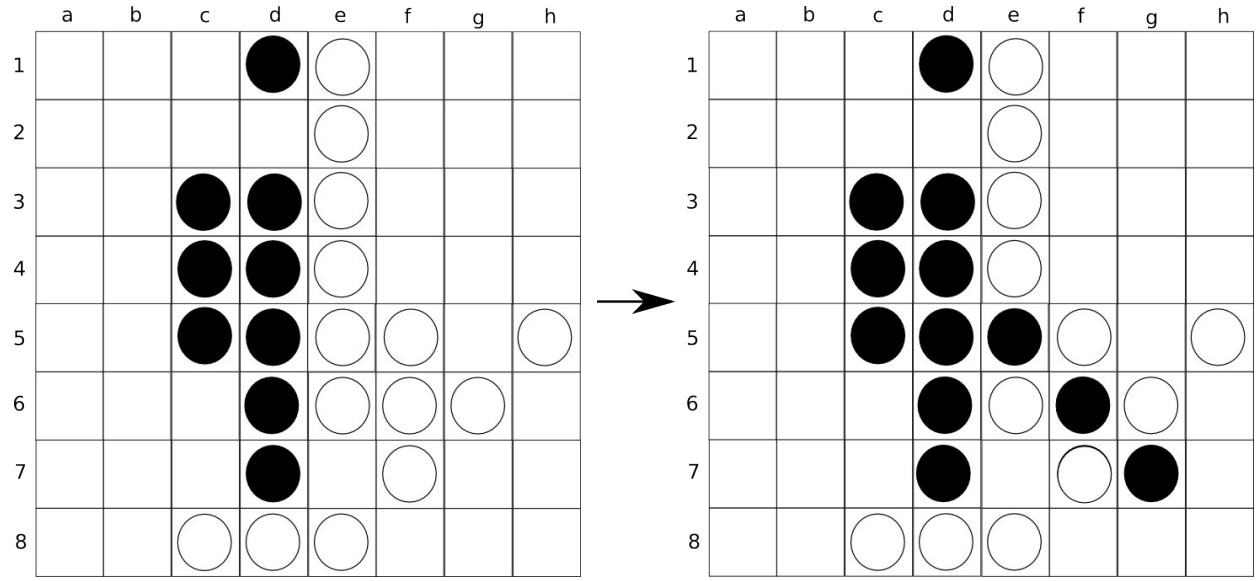
Pravidlá hry a herné stratégie zachytáva dokument Randyho Fanga [4]. Hra reversi bola pôvodne objavená Angličanmi Lewisom Watermanom a Johnom W. Molletom koncom 19. storočia. Čoskoro nadobudla veľkú popularitu. Nové pravidlá známe ako othelo sa zaviedli v 70. rokoch 20. storočia v japonskom meste Miko. Prvé majstrovstvá sveta sa uskutočnili v roku 1977. Najúspešnejšími individuálnymi hráčmi sú doteraz Japonci, z tímov Francúzi.

Reversi hrajú na hracej ploche bežne rozmerov 8x8 dvaja hráči – tmavý a svetlý. Hracia plocha pozostáva zo stĺpcov, ktoré definujú zľava doprava písmená abecedy, a z riadkov, ktoré sú očíslované zhora nadol. Pôvodne reversi nemalo určenú začiatočnú pozíciu. Až pravidlá othelo definujú, že hra sa začína so štyrmi kameňmi v strede hracej plochy, dvomi kameňmi tmavého hráča a dvomi kameňmi svetlého hráča usporiadanými do diagonál. Tmavý hráč má kamene na pozíciách E4 a D5 a svetlý na D4 a E5 ako znázorňuje Obr. 3. Začína čierny hráč.



Obr. 3: Začiatočná pozícia pri reversi

Každý hráč musí umiestniť svoj ďalší kameň na voľnú pozíciu hracej plochy tak, aby existovala aspoň jedna horizontálna, vertikálna alebo diagonálna úsečka medzi novým kameňom a iným kameňom toho istého hráča, pričom všetky medziľahlé kamene patria súperovi. Po uskutočnení takéhoto ťahu sa spomínané súperove kamene stanú kameňmi hráča, ktorý takýto ťah uskutočnil, t.j. zmenia farbu. V praxi to znamená, že sa kameň prevráti na opačnú stranu. Ak hráč nemá možnosť vykonať ťah podľa tohto pravidla, na rade je opäť protihráč. Na Obr. 4 je zobrazený jeden ťah v hre reversi, keď tmavý hráč dá kameň na pozíciu g7.



Obr. 4: Ukážka ťahu v reversi

Hra sa končí, ak už ani jeden hráč nemôže hrať ďalej. Táto situácia môže nastať v troch rôznych prípadoch:

1. celá hracia plocha je zaplnená kameňmi
2. na hracej ploche sa nachádzajú len kamene jedného hráča a druhý hráč nemá ani jeden
3. na hracej ploche sú ešte voľné miesta, ale ani jeden z hráčov nemá platný ťah k dispozícii

Hru vyhráva ten hráč, ktorý má na konci hry viac svojich kameňov na hracej ploche.

Pre othelo zatiaľ nebola objavená stratégia, ktorá by znamenala víťazstvo v hre. Výhoda v podobe veľkého počtu kameňov hráča môže ľahko vyústiť do prehry, pretože obmedzuje možnosť ťahov hráča a dáva veľkú výhodu pre protihráča získať naraz veľké množstvo kameňov. Existujú však elementy, ktoré vedú k úspešnej stratégii:

- *Rohy:* Ak si jeden z hráčov obsadí rohové pozície, ostanú imúnne až do konca hry. Nie je totiž za nimi žiadna pozícia pre ďalší kameň, a tak ani možnosť pre súpera získať ich. Je výhodné obsadiť ich v priebehu hry, nie na začiatku.
- *Pohyblivosť:* Táto metóda sa snaží obmedziť možnosť krokov protihráča. Ak hráč robí postupne také kroky, ktoré obmedzujú legálne pohyby protihráča, tak protihráč je skôr či neskôr donútený urobiť nežiaduci ťah podľa vôle toho druhého. Ideálna pozícia je napríklad,

keď sú kamene hráča sústredené v strede a obklopené kameňmi protihráča. V takom prípade hráč s kameňmi v strede určuje, aké ťahy bude mať protihráč k dispozícii.

- *Hrany:* Ide o podobný princíp ako u rohov. Hráč sústreďuje kamene pri hranách hracej plochy, ktoré je potom ťažké získať protihráčom. Odporúča sa hrať na hrany v rozbehnutej hre, keď už hráč má výhodu v pohyblivosti alebo mnoho kameňov, ktoré mu už určite ostanú.
- *Parita:* Koncepcia parity patrí medzi najdôležitejšie súčasti stratégie. Hráč sa snaží o to, aby aj jeho ťahy pri takmer zaplnenej hracej plochy na konci hry boli uskutočniteľné, a aby sa mu tak v tejto finálnej fáze hry podarilo zväčšiť počet svojich stabilných kameňov.
- *Dopredné pozeranie:* Ako platí aj v šachu, v reversi sa oplatí premýšľať nie len nad aktuálnou situáciou na hracej ploche, ale aj nad možnými ťahmi oponenta a vývojom situácie.
- *Koniec hry:* Keď je hra niekoľko ťahov pred koncom, hráč sa zameriava na iné stratégie, ktoré mu zabezpečia čo najlepší výsledok hry.

Podľa práce Allisa [1] je stavový priestor hry obrovský, rádovo asi 10^{30}. Keďže hra sa v priemere končí pri 58. ťahu a priemerný počet možných ťahov na jeden stav hracej plochy je 10, kompletný strom hry by obsahoval 10^{58} uzlov. Aj to je dôvod, prečo problém víťaznej stratégie pre hru reversi nebol doposiaľ matematicky vyriešený. Experti sa však domnievajú, že pri štandardnej hre na hracej ploche o rozmeroch 8x8 by pri perfektných ťahoch oboch hráčov vyústil výsledok do remízy. Pri rozmeroch 4x4 a 6x6 bolo dokázané, že vyhráva druhý (svetlý) hráč.

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 prehľadávaní, veľkej databáze, heuristike a ohodnocovacej funkcii.

4.3 GO

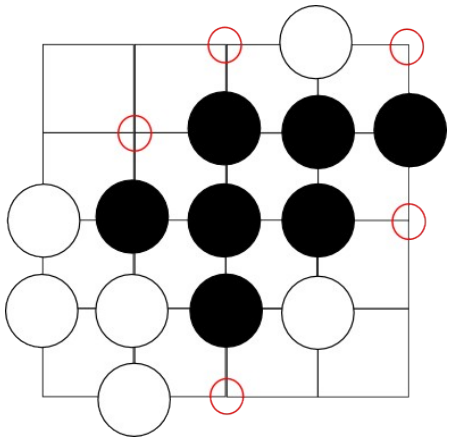
Go je symetrická dosková hra pre dvoch hráčov, o ktorej prvá písomná zmienka pochádza už z 5. storočia pred n.l. z Číny. Dnes má popularitu už na celom svete. Hoci pravidlá go sú veľmi jednoduché, stratégia hry je extrémne komplexná. Go je deterministická strategická hra ako šach, dáma alebo reversi, avšak jej zložitosť prekonáva všetky tieto hry.

Go sa hrá s čiernymi a bielymi kameňmi na doske s mriežkou s rozmermi 19x19. Kamene sú ukladané na priesečníky čiar mriežky, ktorých je v tomto prípade 361. Samozrejme Go môže byť prispôsobené aj na rôzne ďalšie veľkosti, populárne sú ešte rozmery 13x13 a 9x9. Cieľom hry je kontrolovať väčšiu časť hracej plochy ako súper.

Základné pravidlá hry go uvedené v zdroji [13] zahŕňajú:

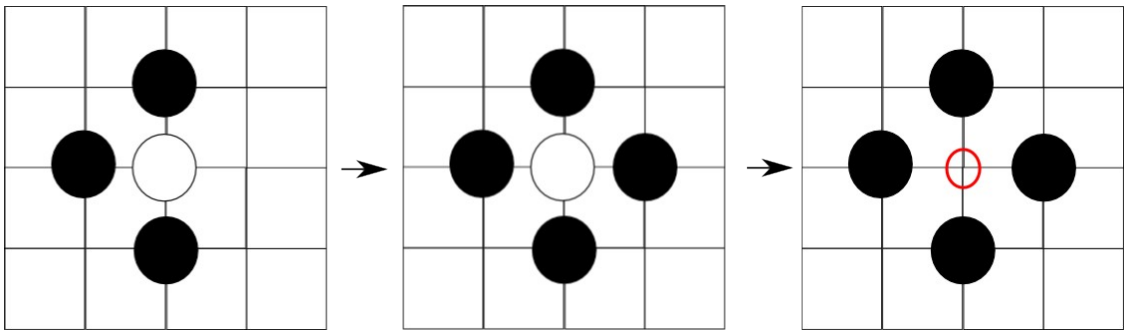
- Biely a čierny hráč sa striedajú v ukladaní kameňov na body mriežky. Čierny hráč začína. Kameňom, ktorý už je uložený na mriežke, sa nesmie hýbať.

- Každý voľný priesečník susediaci s kameňom sa nazýva *sloboda*. Kameň môže mať maximálne štyri slobody.
- Kamene rovnakej farby, ktoré sú navzájom na mriežke spojené, tvoria *skupinu*. Celá skupina kameňov má vlastné slobody, nemôže byť rozdelená a stáva sa tak veľkým kameňom na mriežke. Skupinu tvoria len kamene, ktoré neoddeľuje prázdny priesečník. Ukážková skupina kameňov a jej slobody sa nachádza na Obr. 5



Obr. 5: Skupina čiernych kameňov a jej slobody

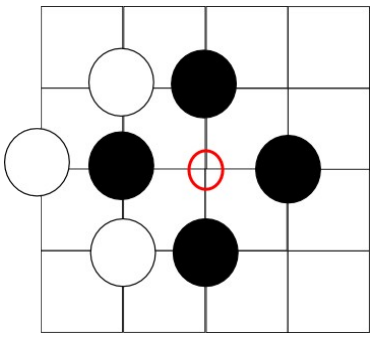
- Skupina kameňov môže byť rozšírená prikladaním kameňov rovnakej farby na jej slobody, ideálnym rozšírením je uloženie kameňa na spoločnú slobodu dvoch skupín tej istej farby.
- Ak hráč postupne obsadí svojimi kameňmi všetky slobody kameňa alebo skupinu kameňov nepriateľa, druhý kameň resp. skupina kameňov je *zajatá*. Po strate poslednej slobody musí byť kameň či celá skupina okamžite odstránená z dosky. Začínajúci hráč si zhromažďuje súperove kamene pre neskoršie počítanie skóre. Obr. 6 ukazuje situáciu, kedy je zajatý a z plochy odstránený jeden biely kameň.



Obr. 6.:Zajatie bieleho kameňa

- Hráč nesmie zahrať taký ťah, aby jeho kameň nemal po ťahu žiadnu slobodu. Do tejto tzv. *samovraždy* môže hráč uložiť kameň len vtedy, ak týmto ťahom zajme súperov kameň či celú skupinu. Jeho kameň tak získa slobody a nestojí po ťahu v samovražde.
- Podľa pravidla *ko* (v preklade nekonečný) hráč ďalej nesmie zahrať taký ťah, aby sa po jeho ťahu presne zopakovala pozícia pred posledným ťahom súpera. Toto pravidlo sa týka situácií,

kde by inak bolo možné brať jeden kameň stále dookola, a tak znemožniť 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. Voľné priesečníky vo vnútri hranice tvorenej kameňmi jednej farby sa nazývajú *územie*. V priebehu hry sa hráči snažia obhajovať svoje územia a súčasne ničiť útokmi územia súpera.
- Hráč môže vynechať ťah, ak si myslí, že to pomôže zväčšeniu jeho teritória, a zmenšeniu teritória súpera. Ak za sebou vynechajú ťah obaja hráči, hra končí. Každý hráč si spočíta body vlastných území na doske a body za zajatcov. Víťazí hráč s väčším bodovým súčtom.

Hranie go je veľmi ťažká úloha. Hráč musí mať za sebou tisícky hier, aby získal určité zručnosti. Postupne sa zdokonaľuje v chápaní, ako spájať kamene do skupín, aby tak mali väčšiu moc. Dobré je pochopiť úvodné sekvencie hry (Fuseki), ktoré sú veľmi dôležité, aby sa hra v strednej fáze rozbehla. Na štandardnej doske je otváranie hry v rohoch viac efektívne ako vytváranie území poblíž stredu. Ďalej je potrebné pochopiť základné koncepty a princípy využívané v komplexných stratégiách hry, ako to vysvetľuje Baker [2]:

- *Spájanie a rozdeľovanie:* Kamene je potrebné držať pohromade. Spájaním kameňov a vytváraním skupín dochádza k rozšíreniu slobôd. Aby potom oponent zajal celú skupinu, musí obsadiť všetky jej slobody. Skupiny kameňov sú teda silnejšie, pretože si navzájom zdieľajú slobody. Na druhej strane, keďže vytváranie skupín robí skupinu silnejšiu, dôležitá ofenzívna taktika tiež spočíva v tom, že hráč sa snaží zabrániť súperovi vo vytváraní takýchto skupín tým, že ich rozdeľuje na menšie skupinky. Hovoríme, že ich odstriháva.
- *Život a smrť:* Kľúčovým konceptom taktiky v go je klasifikácia skupín kameňov na živé, mŕtve a neobývané. Na konci hry skupiny kameňov, ktoré nemôžu uniknúť zajatiu počas hry, sú odstránené z dosky ako zajaté. Tieto kamene sú mŕtve. Ich pochopenie spočíva v tom, že hráč sa namiesto toho, aby sa silou mocou snažil zajať ich počas hry, snaží venovať iným častiam dosky, keďže na konci hry sú tak či tak zajaté. Skupina kameňov, ktorá nemôže byť zajatá, sa nazýva živá. Dôvodom toho je napríklad blízkosť inej skupiny tej istej farby, ku ktorej by sa v prípade útoku súpera na skupinu, dalo priblížiť, spojiť a zabrániť tak úmyslu súpera. Kamene, ktoré nie sú ani živé ani mŕtve, sa nazývajú neobývané.
- Bitky o ko: Existencia bitiek ko súvisí s jedným zo spomenutých pravidiel hry go – ko, ktoré zabraňuje možnosti okamžitého opakovania rovnakej pozície v hre, pri ktorej jeden kameň je

zajatý a následne je zajatý kameň, ktorý ho zajal. Pravidlo hovorí, že okamžité znovuzajatie je zakázané, ale len na jeden ťah. Toto umožňuje nasledujúcu procedúru: hráč so zákazom zahrá taký ťah, ktorý vyžaduje okamžitú odpoveď. Medzitým tak pre neho skonči zákaz ko a následne môže znovu zajať súperov kameň. Tomuto odpútaniu pozornosti sa hovorí hrozba ko. Situácie v hre, pri ktorých si hráči vymieňajú ko hrozby za účelom získania požadovanej pozície na hracej ploche sa nazývajú bitky o ko. V hre môžu nastať situácie s ko, ktoré rozhodujú partiu, pretože rozhodujú o územiach hráčov. Preto bitky o ko v strategických miestach sú často kľúčové, a vždy je potrebné zvážiť, koľko má ktorý hráč hrozieb v rukáve.

- Yose: Na konci partie sú stále v go možnosti, ako získať dôležité a rozhodujúce body. Yose znamená špeciálny spôsob, ako hrať go na konci hry. Podstatné územia sú už síce na konci hry odstavené, ale napriek tomu sú stále menšie územia, o ktoré sa bojuje. Hráč musí dobre zhodnotiť a vypočítať, o koľko bodov sa môže v jednotlivých malých územiach hrať, a čo sa mu teda viac oplatí.

Expertmi je hra go považovaná za kombinačne najzložitejšiu hru vôbec, čo sa prejavuje najmä v zložitosti naprogramovaní hry a obrovskému množstvu variantov ako hrať. Zatiaľ čo v šachu už počítač dvakrát zdolal aktuálneho majstra sveta, najlepšie programy hrajúce go dokážu zdolať aj talentované deti. Typické hry go na hracej ploche s rozmermi 19x19 trvajú podľa Allisa [1] 150 ťahov a priemerne 250 možnými pohybmi v jednom ťahu. Kompletný strom hry go obsahuje asi 10^{360} vrcholov.

Kvôli zložitosti a obľúbenosti sa hrou go zaoberá celá časť umelej inteligencie. Prvý program na hranie go pochádza ešte z roku 1968. Súčasné programy už dosahujú dosť výrazné výsledky na malej ploche 9x9, avšak nie je jasné, ako úspešné techniky dostať aj na štandardnú hraciu plochu. Priemerný hráč tak nemá problém zdolať počítač. Silní hráči dokonca dokážu vyhrať aj s hendikepom 25, ba až 30 kameňov. Najobľúbenejšie programy na go využívajú kombináciu výhod stromového prehľadávania priestoru hry, metód Monte-Carlo, bázy znalostí a strojového učenia, 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 ďalšie vylepšenia.

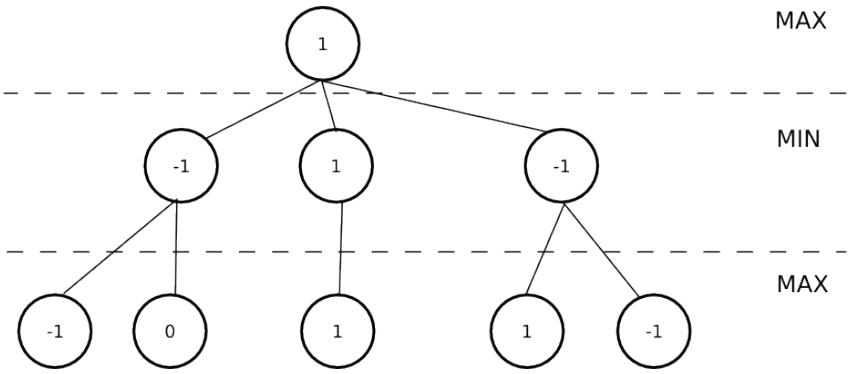
5.1 Aké riešenie hľadáme

Podľa Allisa [1] je *veľmi slabým riešením* (angl. ultra-weak solution) určenie, či hráč, ktorý začína ako prvý, vyhrá, prehrá alebo sa hra skončí remízou, ak obaja hráči hrajú perfektne. Za perfektnú hru hráča sa považuje taká hra, ktorá vedie k čo najlepšiemu možnému výsledku bez ohľadu na to ako ťahá protihráč. Vo veľmi slabom riešení stačí stanoviť výsledok, nie je potrebné poznať ako je potrebné hrať.

V našom prípade na určenie 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 hráčovi zníži šancu druhému. [1]

5.2 MiniMax

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



Obr. 8: Strom hľadania v MiniMax algoritme

Strom hľadania je koreňový strom, ktorého uzly predstavujú stav hry v danom momente (napr. rozloženie šachovnice). Deti uzla predstavujú stavy, v ktorých sa hra ocitne, ak hráč ktorý je na rade vykoná povolený ťah. Keďže sa hráči pri svojich ťahoch striedajú, hĺbka uzlu v strome určuje, ktorý hráč je na ťahu. Špeciálne uzly sú:

- koreň – reprezentuje počiatočný stav hry
- listy – stavy, v ktorých hra končí

Uzly sú ohodnotené celým číslom nasledovne:

- 0 – ak hra vedená z daného stavu skončí remízou
- 1 – ak hru vyhrá prvý hráč
- -1 – ak hru vyhrá druhý hráč

Ohodnotenie uzlov vlastne predstavuje šancu hráča vyhrať hru, pričom v tomto prípade sa na ohodnocovanie uzlov pozeráme vždy z pohľadu prvého hráča. Ten sa snaží maximalizovať svoju šancu na výhru, preto z možných ťahov vyberá taký, ktorý má maximálne ohodnotenie. Jeho oponent, druhý hráč, sa snaží rovnako maximalizovať svoju šancu vyhrať. Keďže sa ale jedná o hru s nulovou sumou, automaticky to znamená minimalizovať šancu vyhrať prvého hráča. Druhý hráč preto vyberá z možných ťahov taký, ktorý má minimálne ohodnotenie. Z tohto dôvodu je prvý hráč označovaný ako MAX, druhý ako MIN, rovnako sú týmito pojmami označované uzly, z ktorých prislúchajúci hráči vykonávajú ťah. Na základe tohto sú jednotlivé uzly stromu hľadania ohodnotené nasledovným algoritmom:

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

Algoritmus končí ohodnotením koreňa.

Rozvíjanie uzlov môžeme v algoritme MiniMax výhodne implementovať ako hľadanie do hĺbky, čím si zabezpečíme lineárnu pamäťovú náročnosť v závislosti od hĺbky stromu hľadania, t.j. O(n). Pri svojej činnosti však potrebuje prejsť celým stromom hľadania, čo ho robí časovo neprijateľne náročným, t.j. časová náročnosť je $O(k^d)$, kde k je faktor vetvenia a d je hĺbka stromu hľadania.

Zápis algoritmu MiniMax v pseudokóde je zobrazený na Obr. 9

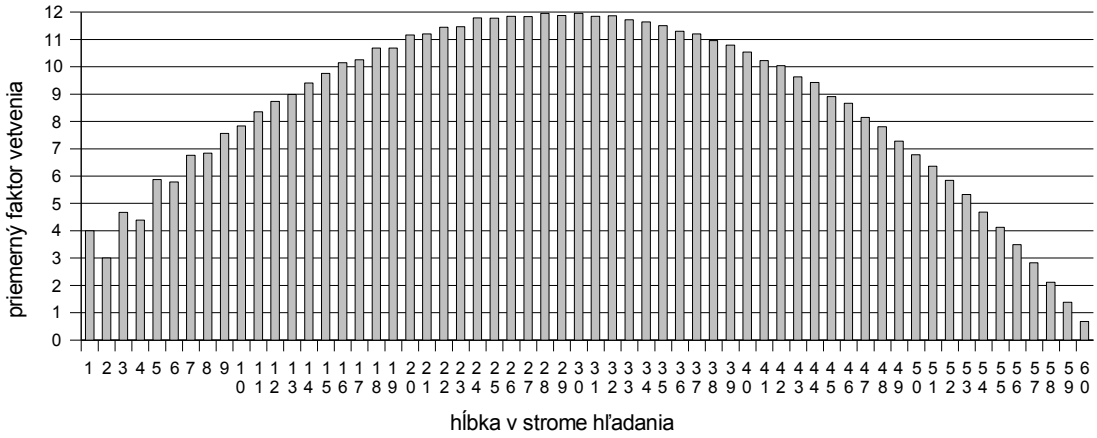
```
//vstupný bod algoritmu
//pos: počiatočná pozícia hry
int MiniMax(pos){
     return MaxValue(pos);
}
//vypočíta ohodnotenie MAX uzla
//pos: pozícia hry, z ktorej vykonáva ťah MAX
int MaxValue(pos){
     if(pos is end position)
          return eval(pos);
     moves = generate(pos);
     score = -INFINITY;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

```
}
//vypočíta ohodnotenie MIN uzla
//pos: pozícia hry, z ktorej vykonáva ťah 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 Veľkosť stromu hľadania

Pre stanovenie veľkosti stromu hľadania potrebujeme poznať faktor vetvenia. Je to priemerný počet detí uzla. Pre hru reversi sme experimentálne prototypom (pozri 9.1.2 Odhad faktoru vetvenia) na základe prechodu stromu hľadania zistili faktor vetvenia. Vzhľadom na povahu nášho riešenia je potrebné vedieť závislosť faktora vetvenia na hĺbke stromu, preto ho uvádzame vo forme funkcie (Obr. 10).



Obr. 10: Graf závislosti faktora vetvenia v strome hľadania od hĺbky uzla

Počet uzlov v strome hľadania môžeme potom odhadnúť podľa vzťahu $C \approx \sum_{k=0}^{n} \prod_{i=0}^{k} k_i$,kde k_i je faktor vetvenia v hĺbke i, na: $2{,}27 \times 10^{53}$

Napriek tomu, že tento odhad je približný, vidíme, že počet uzlov v strome je príliš veľký. Preto je potrebné použiť vylepšenie algoritmu MiniMax, ktoré redukuje počet uzlov, ktoré je potrebné prezrieť.

5.4 Alfa beta usekávanie

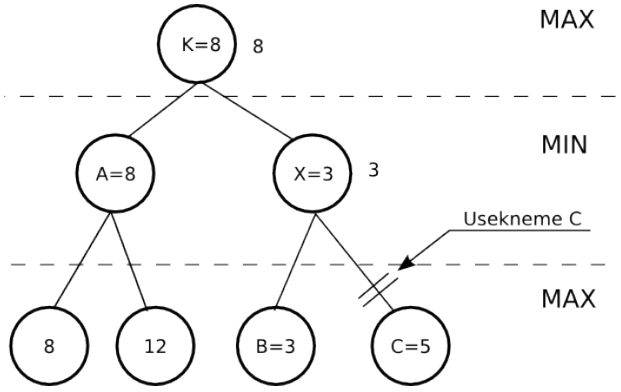
Princíp fungovania

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

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

Interpretácia týchto vlastností v strome hľadania je nasledovná:

A, B, C predstavujú uzly v strome hľadania, pričom rovnakým označením vyjadrujeme aj ich ohodnotenie. Uvažujme prípad 1 zobrazený na Obr. 11.



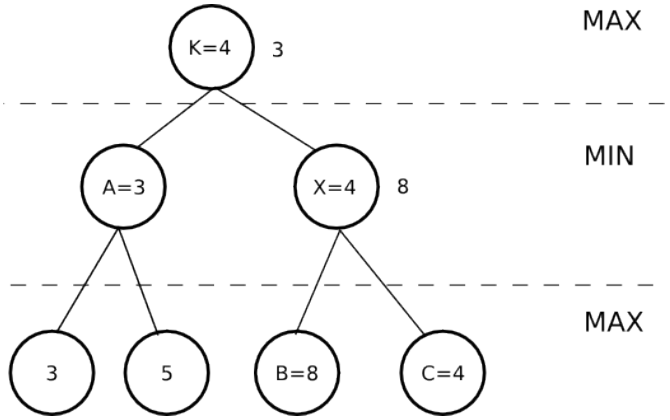
Obr. 11: Ukážka algoritmu alfa beta usekávanie

Predpokladajme, že na ťahu je momentálne hráč MAX, uzol *K*. Keďže môže vykonať ťah, ktorým sa dostane do uzla s ohodnotením *A*, koreňový uzol *K* nemôže byť ohodnotený nižšou hodnotou ako je *A*. Ohodnotenie zvyšného ťahu *X* ešte nepoznáme. Po vykonaní ťahu *X* je však na rade MIN. Ten môže vykonať ťah s ohodnotením *B*, a teda ohodnotenie uzla *X* nemôže vystúpiť 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 hľadania ukázať aj vlastnosť 2.

V oboch prípadoch (splnenie podmienky 1 alebo 2) je teda ohodnotenie uzla K nezávislé na ohodnotení uzla C. Pri prehľadávaní stromu hľadania môžeme teda uzol C ignorovať, čiže useknúť celý podstrom, ktorý v tomto uzle začína.

Implementácia algoritmu generuje deti uzla a prehľadáva ich v presne určenom poradí. V našich príkladoch uvažujeme, že je to v podarí zľava doprava podľa obrázku. Ľahko si môžeme ukázať, 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 voči Obr. 11 navzájom zamenili ľavý a pravý podstrom. Táto zmena mala za následok to, že po ohodnotení uzla *A* vieme, že uzol *K* nemôže klesnúť pod hodnotu 3. Po ohodnotení uzla *B* vieme, že *X* nemôže stúpnuť nad 8. Avšak podmienka $A \geq B$ nie je splnená, takže useknutie nemôžeme vykonať. Teda hodnota uzla $C=4$ sa musí brať do úvahy a v tomto prípade sa ukazuje, že je to aj výsledné ohodnotenie uzla *X* a *K*.

Efektívnosť 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 5.9 Heuristiky.

Implementácia afla beta usekávania

Alfa beta usekávanie je implementované ako prehľadávanie do hĺbky, pri ktorom sa na základe ohodnotení preskúmaných uzlov aktualizujú premenné α a β, ktoré slúžia pri určovaní, či uzol môže byť useknutý:

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

Na začiatku 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 zatiaľ získaných ohodnotení detí (MIN uzly). Ak sa α zmení tak, že bude platiť $\alpha \geq \beta$, predstavuje to splnenie podmienky vlastnosti 2 $MIN(A,MAX(B,C))=A$, a teda ďalšie deti MAX uzla nie je potrebné prešetriť.

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

Algoritmus vyjadrený v pseudokóde je na Obr. 13.

```
//vstupný bod algoritmu
//pos: počiatočná pozícia hry
int AlphaBeta(pos) {
     return MaxValue(pos, -INFINITY, +INFINITY);
}
//vupočíta ohodnotenie MAX uzla
//pos: pozícia, z ktorej vykonáva ťah 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;
}
//vypočíta ohodnotenie MIN uzla
//pos: pozícia, z ktorej vykonáva ťah 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 implementačné vylepšenie MiniMaxu, ktoré využívajú všetky praktické implementácie ostatných algoritmov počnúc od alfa beta usekávania.

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

Využitím tejto vlastnosti môžeme MiniMax upraviť nasledovne:

1. Ohodnotenie uzla bude vždy vzhľadom na toho hráča, ktorý z daného stavu vykonáva svoj ťah.
2. Vďaka bodu 1 teraz v každom ťahu maximalizuje každý hráč priamo svoju šancu vyhrať. Vnútorný uzol stromu sa teda ohodnotí nasledovne: $V = MAX_i \{-V_i\}$, kde V_i je ohodnotenie dieťaťa uzla V. $-V_i$ predstavuje transformáciu ohodnotenia tak, aby bolo vyjadrené vzhľadom na hráča, ktorý vykonáva svoj ťah z uzla V.

Tento prístup umožňuje teda zjednotiť pohľad na MAX a MIN uzly.

5.6 Alfa beta usekávanie s NegaMax rozšírením

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

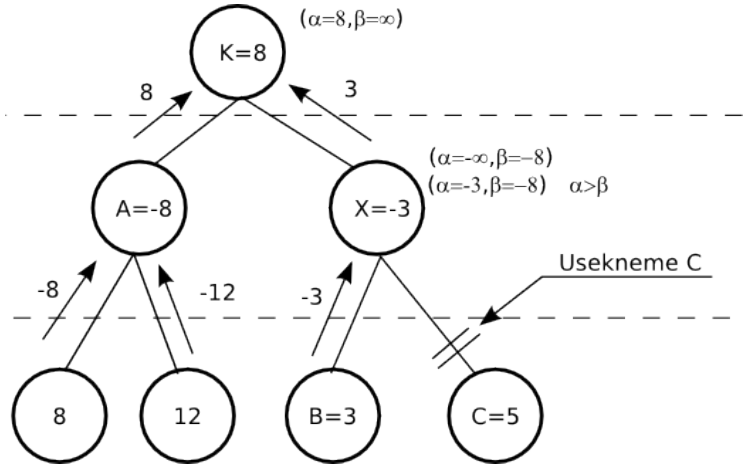
- α je zatiaľ najlepšie ohodnotenie skúmaného uzla.
- β je zatiaľ najlepšie ohodnotenie rodiča skúmaného uzla.

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

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

V procese získavania maxima z ohodnotení detí aktualizujeme rovnako α^{new} na maximálnu zatiaľ zistenú hodnotu, teda hodnota α^{new} nemôže klesnúť. Ak sa splní podmienka $\alpha^{new} \geq \beta^{new}$, platí rovnako $-\alpha^{new} \leq -\beta^{new}$, čo je $-\alpha^{new} \leq \alpha^{old}$. Z pohľadu rodiča vyhodnocovaného uzla to znamená, že ohodnotenie uzla $-\alpha^{(new)}$ už nemôže nijako zlepšiť jeho ohodnotenie. Preto pri splnení tejto podmienky ukončujeme prezeranie ďalších potomkov skúmaného uzla.

Na Obr. 14 je na príklade ukázaná táto situácia.



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

Ohodnotenie ľavého podstromu je $A=-8$, čím 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 $\alpha^{(new)}=-3$, čím je okno upravené na $(\alpha^{(new)},b^{(new)})=(-3,-8)$. Keďže 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 hľadania 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
dieťaťa
          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 redukovať počet uzlov, ktoré je potrebné prehľadať. Je založený na myšlienke, že keďže väčšina uzlov po prvom dieťati sa usekne, ich presné ohodnocovanie je nepotrebné. Preto sa NegaScout snaží prehľadávaním s minimálnym oknom ukázať, že sú horšie. Prehľadávanie s minimálnym oknom znamená, že α a β koeficienty sa nastavia tak, aby platilo: $\alpha=\beta-1$. Prvý uzol je ohodnotený vždy prehľadávaním s pôvodným oknom.[7]

V prípade, že hľadanie s minimálnym oknom nájde lepšie ohodnotenie ako je aktuálne, prehľadávanie musíme zopakovať s väčším oknom, aby sme získali správne ohodnotenie. Iba ak hľadanie s minimálnym oknom vráti nižšie ohodnotenie, môžeme tento výsledok akceptovať.[7]

Algoritmus NegaScout zapísaný v pseudokóde je na Obr. 16.

```
//vstupný bod algoritmu
//pos: aktuálna pozícia hracej plochy
//d: hlbka hladania
//alpha, beta: okno hľadania 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 prehľadáva s minimálnym oknom. Tento prístup môže viesť väčšiemu usekávaniu uzlov, avšak hľadanie s minimálnym oknom nezistí presné ohodnotenie uzla ale iba jeho odhad. Niekedy musí prehľadávať ten istý podstrom opäť 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 ťahov podľa ich sľubnosti. Ak sa lepšie uzly ohodnocujú skôr, zlepšuje to šancu sa useknutie podstromu a redukovanie počtu uzlov, ktoré je potrebné prehľadať.

5.9.1 Heuristika vražedného ťahu

Táto heuristika je založená na myšlienke, že rôzne pozície nachádzajúce sa v uzloch rovnakej hĺbky majú podobnú povahu. Ak je nejaký ťah v jednej časti stromu dobrý, potom bude s vysokou pravdepodobnosťou dobrý aj v inej časti stromu na tej istej hĺbke.

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

5.9.2 Heuristika založená na histórii ťahov

Je rozšírením heuristiky vražedného ťahu. V tabuľke sa uchováva skóre priradené uzlom. Zakaždým, keď na základe ohodnotenia uzla usekneme časť stromu prehľadávania, zvýšime uzlu skóre.

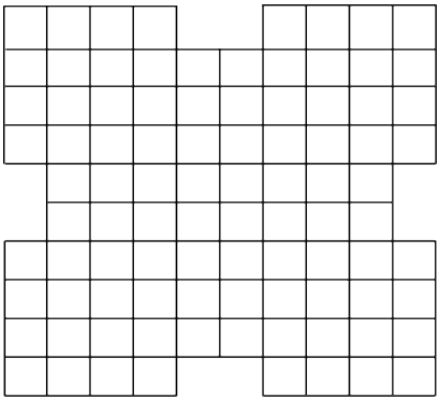
Skóre môžeme navyšovať napr. o hodnotu $2^{h_{max}-h}$, kde h je hĺbka uzla v strome hľadania. Takýto vzťah je výhodné použiť, aby vyššie skóre získali stavy bližšie pri koreni, kedže usekávanie v nižšej hĺbke usekne väčší podstrom.

Pri prehľadávaní detí uzla postupujeme v poradí určenom pomocou získanej tabuľky. Uzly s vyšším skóre sú prehľadávané skôr.[7]

5.9.3 Problémovo závislé ohodnotenie vhodnosti pozície

Ďalší spôsob ohodnotenia vhodnosti pozícií môže byť problémovo závislý, vychádzajúci z pravidiel hry. Spôsob ohodnotenia, ktorý použili [3] vo svojej práci pre hru reversi, je v zásade založený na počítaní rozdielu medzi počtom kameňov oboch hráčov. Kamene na rôznych pozíciách majú rôzne kvalitné postavenie – napr. kamene v rohoch nemôže súper nijako ovládnuť. Rôznym políčkam 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 počtu kameňov δ_K (piece differential)

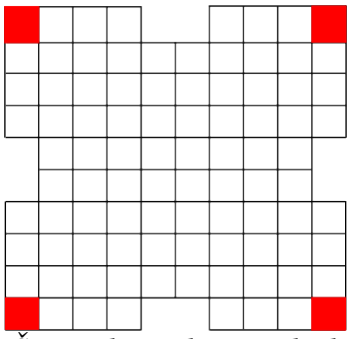
Veľmi jednoduchá metrika, ktorá počíta rozdiel počtu políčok, ktoré má hráč 1 a hráč 2 obsadené. Je jednoduchá, no potrebná, keďže koniec hry sa stanovuje iba na základe δ_K .

Rozdiel počtu ťahov δ_T (mobility differential)

Je rozdiel počtu ťahov, ktoré môže z daného uzla vykonať hráč 1 a hráč 2. Na výpočet je potrebné vedieť, ktorý hráč je na rade, aby sa určilo správne poradie v odčítaní.

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

Je rozdiel počtu štvorrohov obsadených hráčom 1 a hráčom 2. Ako štvorrohy označili autori zvýraznené políčka na Obr. 19.

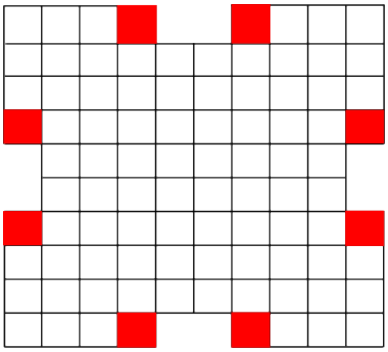


Obr. 19: Štvorrohy na hracej ploche reversi

Rohy sú považované za dobré ťahy. Nielenže ich súper nemôže získať, predstavujú aj strategický bod pre získavanie súperových kameňov.

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

Je rozdiel počtu osemrohov obsadených hráčom 1 a hráčom 2. Ako osemrohy označili autori zvýraznené políčka na Obr. 20.

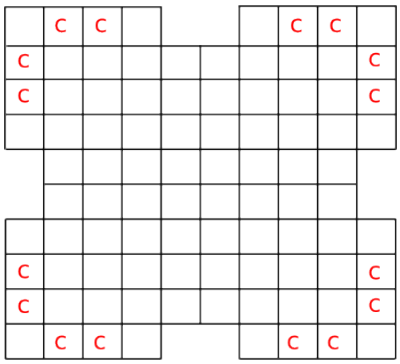


Obr. 20: Osemrohy na hracej ploche reversi

Podobne ako štvorrohy sú považované za dobré ťahy. Vzhľadom na odlišný vzťah k ostatným políčkam ich autori vyčlenili zvlášť.

Rozdiel počtu C políčok δ_C (C-squares differential)

Je rozdiel počtu C políčok obsadených hráčom 1 a hráčom 2. Ako C-políčka označili autori zvýraznené políčka na Obr. 21.

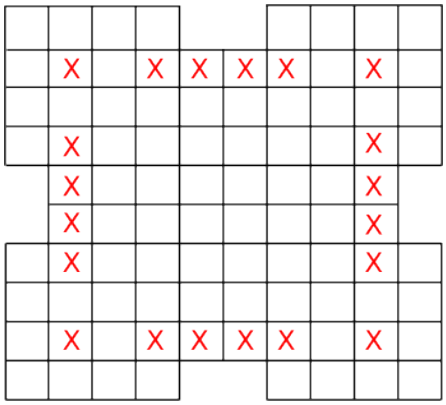


Obr. 21: C-štvorce na hracej ploche hry reversi

Tieto políčka sú považované aj za dobré aj za zlé. Zlé, pretože umožňujú súperovi obsadiť rohy. Dobré, lebo sa nachádzajú na okraji.

Rozdiel počtu X-políčok δ_X (X-squares differential)

Je rozdiel počtu X-políčok obsadených hráčom 1 a hráčom 2. Ako X-políčka označili autori zvýraznené políčka na Obr. 22.



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

Tieto políčka sú považované za zlé, pretože dávajú súperovi šancu obsadiť rohy.

Rozdiel počtu hraničných políčok δ_H (frontier differential)

Je rozdiel počtu hraničných políčok obsadených hráčom 1 a hráčom 2. Hraničké políčko je také, ktoré susedí v ľubovoľnom smere s aspoň jedným prázdnym políčkom.

Rozdiel počtu stabilných políčok δ_S (Stable pieces differential)

Je rozdiel počtu stabilných políčok obsadených hráčom 1 a hráčom 2. Stabilné políčko je také, ktoré nemôže protihráč teraz a ani nikdy v budúcnosti nijako prevziať.

Rozdiel počtu sendvičových políčok δ_B (Sandwich Squares Differential)

Je rozdiel počtu sendvičových políčok obsadených hráčom 1 a hráčom 2. Sendvičové políčko je také, ktoré z oboch strán v nejakom smere susedí s protihráčom.

Ošetrenie vyhladenia

Potreba špeciálneho ošetrenia prípadu, ak sú všetky kamene jedného hráča prevzaté. V takomto prípade má heuristická funkcia vracať $-\infty$. Je to ošetrenie z pohľadu hraničných políčok, z ktorého pohľadu je vyhladenie dobrým stavom.

5.10 Transpozičná tabuľka

Je to tabuľka, do ktorej sa k pozíciám hracej plochy ukladá jej ohodnotenie. Využíva sa to, že do rovnakej pozície sa vieme dostať rôznymi spôsobmi. Preto je výhodné zapamätať si už vypočítané ohodnotenie hracej plochy a neskôr, keď sa rovnaká pozícia zopakuje, ho len z transpozičnej tabuľky prečítať. Týmto spôsobom ďalej redukujeme počet potrebných prehľadaných uzlov.

Keďže vzhľadom na pamäťové nároky a výpočtovú réžiu nie je možné uchovávať si všetky ohodnotenia a kontrolovať všetky uzly voči transpozičnej tabuľky, je potrebné pri návrhu stanoviť:

1. ohodnotenia ktorých uzlov sa budu ukladať do tabuľky
2. ak tabuľka bude v pamäti zaberať maximálnu povolenú veľkosť, ako sa určí, ktoré ohodnotenia sa budú vyhadzovať
3. ktoré uzly sa budu voči transpozičnej tabuľke kontrolovať

5.11 Symetrické pozície

Dve hracie plochy sú symetrické, ak otočením, preklopením podľa osi alebo stredu jednej získame tú druhú. V niektorých hrách, ako aj v reversi (šach nie) sú symetrické pozície vlastne rovnaké. Z hľadiska ohodnocovania stromu prehľadávania môžeme takéto pozície považovať za totožné, čím opäť redukujeme počet uzlov, ktoré potrebujeme preskúmať. Riešením takéhoto problému môže byť tabuľka, do ktorej vkladáme ohodnotenie uzla. Kľúč, pod ktorým ho vkladáme, však musí spĺňať vlastnosť, aby bol pre symetrické plochy rovnaký. Navyše musí byť jednoznačný alebo musí mať čo najmenej kolízií, aby sa nemuseli plochy pracne porovnávať voči všetkým možným transformáciám.

5.12 Serverové a klientske prehľadávanie

Strom hľadania pre uvažované hry je veľmi veľký. Jeho prehľadávanie na jednom počítači, aj s množstvom vylepšení, zaberie veľmi veľa času. Hľadanie riešenia je preto rozdelené pomocou BOINC systému medzi viacero počítačov – klientov. Tí počítajú čiastkové úlohy, ktoré im server prideľuje, a následne aj vyhodnocuje výsledky.

Z povahy BOINC systému a riešenia vyplýva, že server aj klient budú obaja riešiť prehľadávanie stromu, ohodnocovanie uzlov a usekávanie nepotrebných častí stromu. Každý z nich však k tejto úlohe bude pristupovať iným spôsobom daným ich rôznymi špecifikami.

Špecifiká servera a klienta

Server

- väčšia dostupnosť pamäte
- potreba vytvoriť úlohy pre klientov
- nemôže sa zamestnať jednou úlohou, ktorej výpočet trvá veľmi dlho

Klient

- obmedzené množstvo pamäte a výkonu, ktorým hosťovský počítač disponuje
- je určený na počítanie jednej čiastkovej úlohy. Môžeme ho ňou úplne zamestnať (vyťažiť).

Spôsob realizácie prehľadávania

Klient

- prehľadávaním do hĺbky prostredníctvom niektorého z vylepšených alfa beta usekávaní so zapracovanými heuristikami, tabuľkami a podobne

Server

- pri prehľadávaní musí prehľadávať do šírky, aby vygeneroval dostatok úloh pre klientov
- pri prehľadávaní musí prehľadávať zároveň do hĺbky, aby zišiel dostatočne hlboko do stromu, a aby mohol klientom generovať reálne vyriešiteľné úlohy
- používa usekávanie podstromu hľadania po asynchrónnom získaní výsledkov od klienta

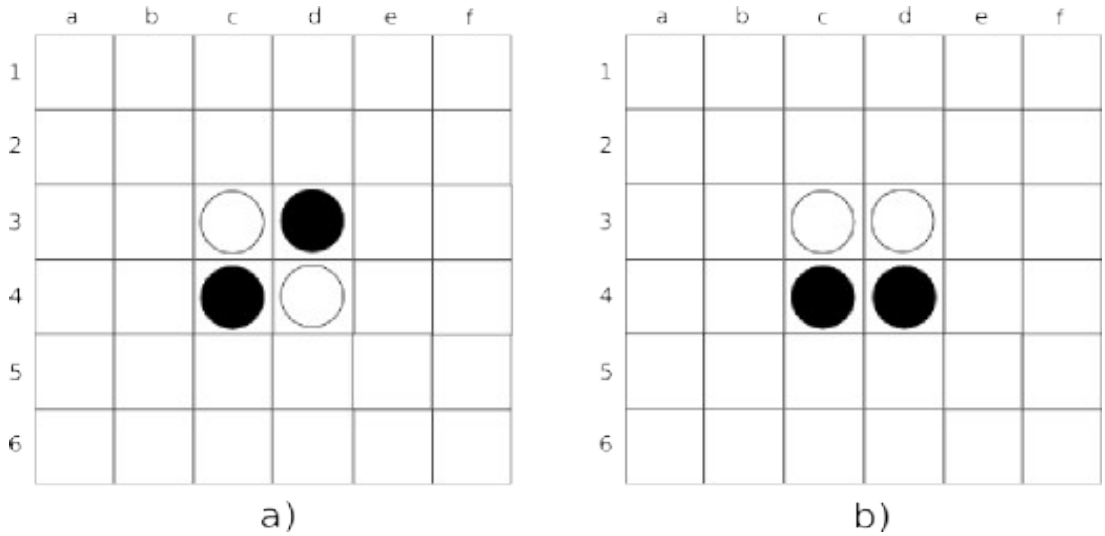
Ako poukazuje Liang [7], použitie konkrétneho algoritmu v klientskej časti je otázkou experimentálneho vyskúšania. Podľa autora sa nedá povedať, ktoré z uvedených vylepšených riešení je lepšie. Keďže charakteristiky stromu prehľadávania sú rozdielne pre každú hru, pre každú sa môže ukázať iný algoritmus ako vhodný.

6 Existujúce riešenia

V tejto časti sú popísané existujúce riešenia symetrických hier reversi a GO. Tieto riešenia boli dosiahnuté na jednom počítači, na rozdiel od nášho projektu, kde budeme riešiť problémy distribuované.

6.1 Reversi

Dr. Joel Feinstein v roku 1993 vyriešil problém reversi pre veľkosť hracej plochy 6x6. [5] Dva týždne počítačového času mu trvalo, aby sa dopracoval k výsledku. Výsledok znel, ak obaja hráči hrajú perfektne, tak 20:16 vyhrá hráč, ktorý ide ako druhý v poradí (20 krát vyhrá druhý hráč, zatiaľ čo prvý hráč vyhrá len 16 krát). Počas behu programu bolo vygenerovaných okolo 40 mld. pozícií. Feinstein tiež spočítal trvanie výpočtu pre hraciu plochu veľkosti 8x8. Podľa jeho výpočtov by to trvalo 10^{14}, približne 3,8 milióna rokov. Riešenie reversi 6x6 s alternatívnou štartovacou pozíciou (Obr. 23b) trvalo Feistaeinovi spočítať 5 týždňov a vygenerovalo sa okolo 100 mld. pozícií. Tu bol pomer 19:17 v prospech druhého hráča. Feinstein robil výpočty na jednom počítači.



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

Pri riešení použil prehľadávanie hrubou silou za pomoci algoritmu alfa-beta usekávania s usporiadaním ťahov podľa jeho ohodnocovacej funkcie. Program používal 70 KB pamäti.

6.2 GO

Dňa 19. októbra 2002 sa podarilo Erik van der Werf vyriešiť hru GO pre veľkosť hracej plochy 5x5. [17] Program pracoval tak, že prvý ťah bol v strede hracej plochy. Víťazom bol čierny hráč (prvý v poradí ťahania). Konečné skóre bolo 25 pre čierneho, celá hracia plocha bola obsadené čiernymi kameňmi.

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

1) transpozičná tabuľka – mala 2 x 2^{24} položiek
2) rozšírené transpozičné osekávanie
3) symetria bola vyhľadávaná v transpozičnej tabuľke
4) 2 killer moves
5) historické heuristiky
6) Bensonov algoritmus
7) heuristické ohodnotenie pre pozície, ktoré neboli určené Bensonovým algoritmom

Riešenie bolo nájdené v 22. úrovni (pre prázdnu plochu v 23. úrovni). Výpočet prebiehal na počítači P4 2.0Ghz. Počas výpočtu bolo vygenerovaných približne 4,5 mld. uzlov. Čas trvania tohto výpočtu bol asi 4 hodiny.

Nadôležitejším pri riešení bol Bensonov algoritmus, ktorý zmenšil hĺbku prehľadávaného 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 vyhnúť všetkým dlhým cyklom.

V decembri 2002 Erik van der Werf vylepšil svoj program na riešenie GO 5x5, nazval ho MIGOS (Mini GO Solver). Ďalej rozšíril program o riešenie hier, ktoré nie sú otvorené v strede hracej plochy. Program bol schopný riešiť akékoľvek 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 políčko môže mať 3 rôzne stavy (čierny, 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 - ďalšie šachovnice, ktoré sú od aktuálnej šachovnice vzdialené 1 ťah (teda + 1 ťah). Šachovnica má zároveň ohodnotenie (nemá ho pri vytvorení, ale získa ho až pri ohodnocovaní stromu). To môže mať 3 hodnoty - vyhral biely, čierny, remíza.

7.2 Veľkosť uložených dát

Strom celej hry reversi $8x8$ má maximálnu hĺbku 64. Faktor vetvenia sa zo začiatku pohybuje okolo 6 (podrobnejšie kapitola 9.1.2 Odhad faktoru vetvenia) a celý strom obsahuje približne $2,2716 x10^{53}$ uzlov. Z toho bude časť uložená na klientovi a časť 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 výpočet nie je ale správny, lebo nezahŕňa informácie o hierarchii ani o ohodnotení jednotlivých uzlov. Preto môžeme použiť takúto dátovú štruktúru (Tab. 1), ktorá implementuje aj hierarchiu a ohodnotenie:

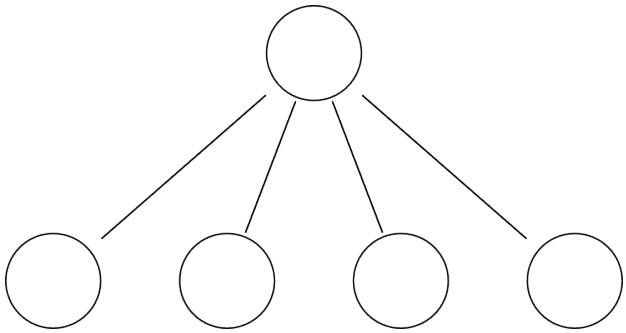
Uzol	
Stav hry	128b
Rodičovský stav	128b
Ohodnotenie	2b

Tab. 1: Pamäťové nároky uzla

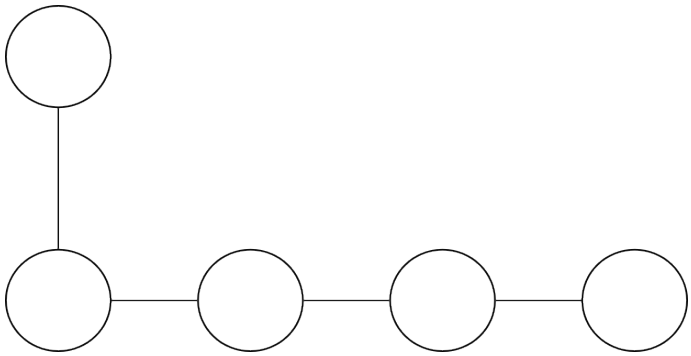
Potom by sme potrebovali na uloženie 1 záznamu 258b, čo je 33 B, teda na disk by sa nám zmestilo $1,627 \times 10^{10}$ uzlov. To zodpovedá celému stromu do hĺbky 11 (podľa 5.3 Veľkosť stromu hľadania).

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 potrebovať jednoducho zistiť nasledovníka a predchodcu daného uzla, preto je podľa mňa prvá možnosť pre nás výhodnejšia.

7.4 Konkrétne možnosti ukladania stromu na disk

7.4.1 Existujúci systém súborov

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

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

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

V súčasnosti existujú FS, ktorým nerobí problém tých cca 3×10^{10} súborov. Hlavným problémom je minimálna veľkosť bloku. Veľkosť sektora disku neumožňuje mať menšiu najmenšiu veľkosť bloku ako 512 B. Ďalším problémom je, že na väčšine FS sa pri tak malom bloku nedá vytvoriť dostatočne veľký oddiel. Z dostupných FS pre bežné operačné systémy by z hľadiska veľkosti oddielu pri minimálnej veľkosti bloku vyhovoval napr. ReiserFS4 [18], ReiserFS [19] (*nix) alebo ZFS (Sun Microsystems). Tie sú ale nevhodné kvôli obmedzenému maximálnemu počtu súborov na disku (2^{32}) . Väčšina používaných FS, napr. FAT, NTFS, nespĺňa 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, čo je 32 krát viac).

7.4.2 Vlastný FS

Týmto riešením by sme dokázali obísť vyššie spomenuté limity. Realizácia by bola pravdepodobne pomocou priameho zápisu na disk, napr. do jedného veľkého súboru. Problém je však v implementácii riešenia. Tá sa zdá byť príliš zložitá a nerealizovateľná v danom čase.

7.4.3 Databáza

Podľa [1] ju použili na ukladanie sekvencií genómu. Preto som testoval, či by bola použiteľná aj v našom prípade.

V mojich pokusoch som predpokladal trojice parent - child - ohodnotenie_child. Použil som PostgreSQL 8.1 (ďalej len PG). Postupne som generoval databázu s cca $2{,}6 \times 10^6$ záznamov (na disku zaberá zhruba 500 MB). Počas generovania som meral efektivitu využívania diskového miesta (koľkokrát 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 stĺpce parent a child. Select podľa stavu hry pracoval rýchlo (rádovo ms). Merania som robil vždy v jednej a tej istej tabuľke po spustení VACUUM FULL. VACUUM je pomerne časovo náročná 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.

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

Tab. 2: Zavislosť efektivity od počtu záznamov v DB

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

Max. veľkosť DB	neobmedzená
Max. veľkosť tabuľky	32 TB
Max. veľkosť riadku	1,6 TB
Max. veľkosť poľa	1 GB
Max. počet riadkov v tabuľke	neobmedzený
Max. počet stĺpcov v tabuľke	250 – 1600, podľa typu
Max. počet indexov v tabuľke	neobmedzený

Tab. 3: Obmedzenia DB

7.5 Záver k ukladaniu dát

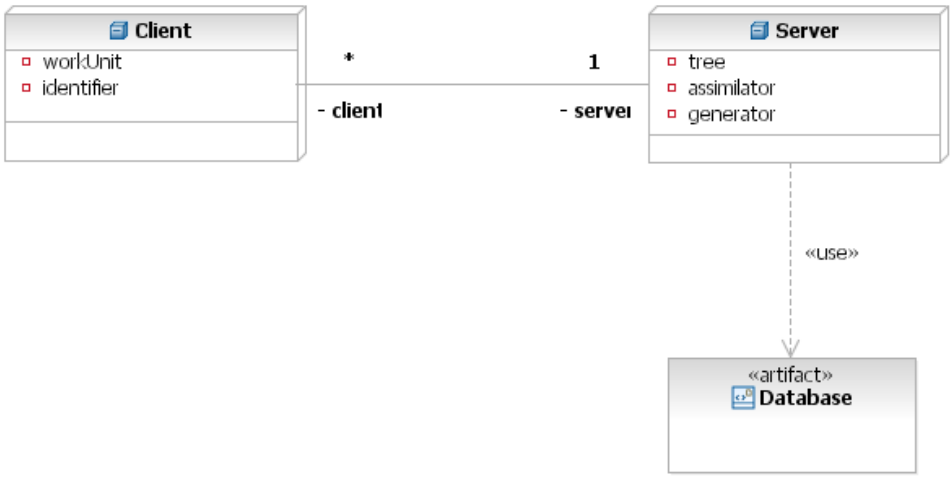
Ako najefektívnejší a najjednoduchší spôsob na ukladanie dát sa javí databáza. Bolo by možno dobré otestovať, ako sa budú správať na tomto type dát iné databázy, napr. MySQL, SQlite, Firebird.

8 Návrh systému

Táto kapitola sa zaoberá návrhom systému nášho projektu. Hneď na začiatku je tu otázka, či vôbec ide o systém. Nájsť veľmi slabé riešenie hry je v podstate len jeden algoritmus, prispôsobený naším konkrétnym podmienkam. Na druhej strane, keďže sa jedná o využitie existujúceho systému na paralelné počítanie - BOINC (pozri kapitola 3), ktoré obsahuje pojmy ako server a klient, tak je namieste hovoriť o architektúre - návrhu aj keď značne odlišného od veľkých softvérových produktov.

8.1 Návrh paralelného systému

Využitím systému BOINC sa nám do problému dostalo isté ohraničenie, ktoré definuje prvotnú architektúru klient - server. Časť servera sa ešte delí na dve hlavné komponenty (časti), a to generátor a asimilátor (Obr. 26), ktoré sú bližšie popísané v kapitole 3.2Démoni bežiaci na serveri. Keby sa to malo zhrnúť, tak generátor sa stará o generovanie zadaní pre klientské počítače, tzv. *workunit,* a asimilátor je zodpovedný za spracovanie výsledku od jednotlivých klientov.



Obr. 26: Celkový pohľad na systém

Systém je navrhnutý čo najviac modulárny, tak ako dovoľuje jazyk C. Dôvodom je použitie pre rôzne typy distribuovaných hier. Ako bolo práve spomenuté, naším implementačným jazykom bude obyčajný jazyk C, hlavne a jedine kvôli rýchlosti, pretože časová náročnosť riešenia je značne veľká. Problém je riešený prehľadaním celého priestoru riešení použitím distribuovaného prehľadávanie 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 značne prispôsobený. Architektúra nebude klasická čo 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 hlavičkové 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 funkčnosti bude použitý klasický sekvenčný diagram UML, s tým že jeho vykonávatelia sú komponenty alebo rozhrania realizované komponentom. Celý návrh vrátane jeho častí komponent a rozhraní bude kvôli znovu použiteľnosti pomenovaný anglickými výrazmi, ktoré sú súčasťou projektového slovníka (pozri Príloha A – Slovník pojmov) a jednotlivé prvky diagramov sú presnejšie opísané v Príloha B – Použitá notácia diagramov.

Pod pojmom zoznam budeme rozumieť lineárne jednosmerne zreťazený zoznam, klasickú dátovú štruktúru. Prvý hráč je označenie pre hráča, ktorý začína partiu hry, a existuje práve jeden ďalší - druhý hráč. Rozhranie nenavrhované nami, inak povedané externé - definované v BOINC, bude na diagramoch farebne oddelené. Tieto pojmy budú ďalej bez odvolávania sa používané.

Spomínaná modularita má výhodu aj v existencii spoločných častí pre klient a server, na strane servera spoločných pre generátor a asimilátor. V nasledujúcich kapitolách budú tieto časti podrobne opísané.

8.2 Návrh spoločných častí

Prvky v tejto kapitole sú potrebné pre klienta a server, opisujú stavový priestor hry, čiže aj musia byť zhodné. Prehľadávanie s využitím usekávania je naopak jedinečné, to je popísané v kapitole 5 Teoretický základ pre riešenie hier. Prvky tejto, ale aj iných častí, budeme nazývať komponentmi. V prípade jazyka C sa väčšinou bude jednať o jedno rozhranie, ktoré predstavuje hlavičkový 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 potrebovať štruktúry na uchovanie, a funkcie na manipuláciu so šachovnicou. Táto je pre každú hru iná, najmä čo sa týka rozmeru, hracích figúrok (kameňov), keďže 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 pozerať na jednotlivú situácie v hre. Toto rozhranie je špecifikované súborom .h, obsahuje potrebné typy a funkcie. Na nasledujúcom obrázku (Obr. 27) je vidieť 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 kameňov. Má presne definované rozmery a kódovanie pre jednotlivé možné stavy políčka šachovnice (napr. kameň prvého hráča predstavuje číslo jedna).
- `Move` - popisuje možný ťah hráča ktorý je práve na ťahu. V zásade obsahuje jednu alebo viac súradníc šachovnice z ktorými stavmi políčok je vykonaná zmena v dôsledku ťahu hráča.

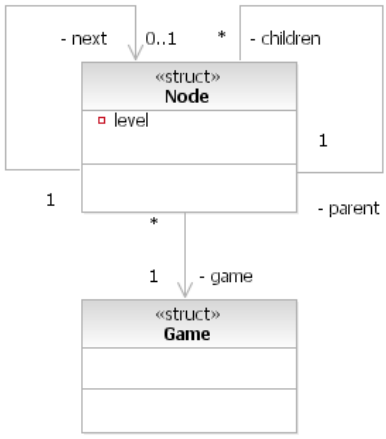
Funkcie

- `getLegalMoves(Game)` - vráti zoznam možných ťahov pre konkrétnu hru podľa pravidiel zahrnutých v implementácii tohto rozhrania. Keďže sa nejedná o hľadanie stratégie, ale (len) o hľadanie veľmi ťahkého riešenia, môžu byť pravidlá hry takto oddelené od ostatných častí.
- `getGame(Game, Move)` - vráti novú inštanciu štruktúry hra, po spravení zadaného ťahu na zadanej hre.
- `evaulateGame(Game)` - ohodnotí zadanú konečnú (neexistuje ťah pre hráča na rade) konfiguráciu hry jedným číslom, ktoré určuje výsledok tejto hry. V našom prípade hrajú vždy dvaja hráči, nula znamená remízu, kladné číslo čím väčšie tým vyššia výhra prvého hráča, analogicky záporná hodnota.
- `initGame()` - vráti počiatočne nastavenú hru s "vynulovanou" šachovnicou.
- `makeMove(Move)` - podobne ako getGame(), len teraz vykoná zadaný ťah na danej hre.
- `heuristicEvaulation()` - vráti ohodnotenie hry na základe špecifickej heuristiky definovanej pre konkrétnu hru.

8.2.2 Komponent Uzol

Prehľadávaný priestor je graf, presnejšie strom. Možnosť opakovania sa rovnakej konfigurácie v grafe bude riešená porovnávaním na strane servera, na strane klienta bude riešená len interne, alebo nebude riešená. Podrobnejšie túto záležitosť opisuje kapitola 5.10 Transpozičná tabuľka. Graf je dvojica uzly a hrany. Budeme reprezentovať uzol ako dátovú štruktúru, a hrany ako referencie uzla na iné. Nasledujúce obrázky popisujú rozhranie *node.h* (Obr. 28) a znázorňuje prepojenie štruktúr typu Game a typu Node (Obr. 29).

Obr. 28: Rozhranie node.h



Obr. 29: Vzťah node so štruktúrou game

Štruktúry

- `Node` - predstavuje uzol grafu, obsahuje referenciu na potomkov a na rodiča. Pre rýchle zistenie hĺbky v strome obsahuje aj tento atribút - level, z ktorého a vrátane aktuálnej konfigurácie hry, bude možné zistiť, kto z hráčov je na ťahu.

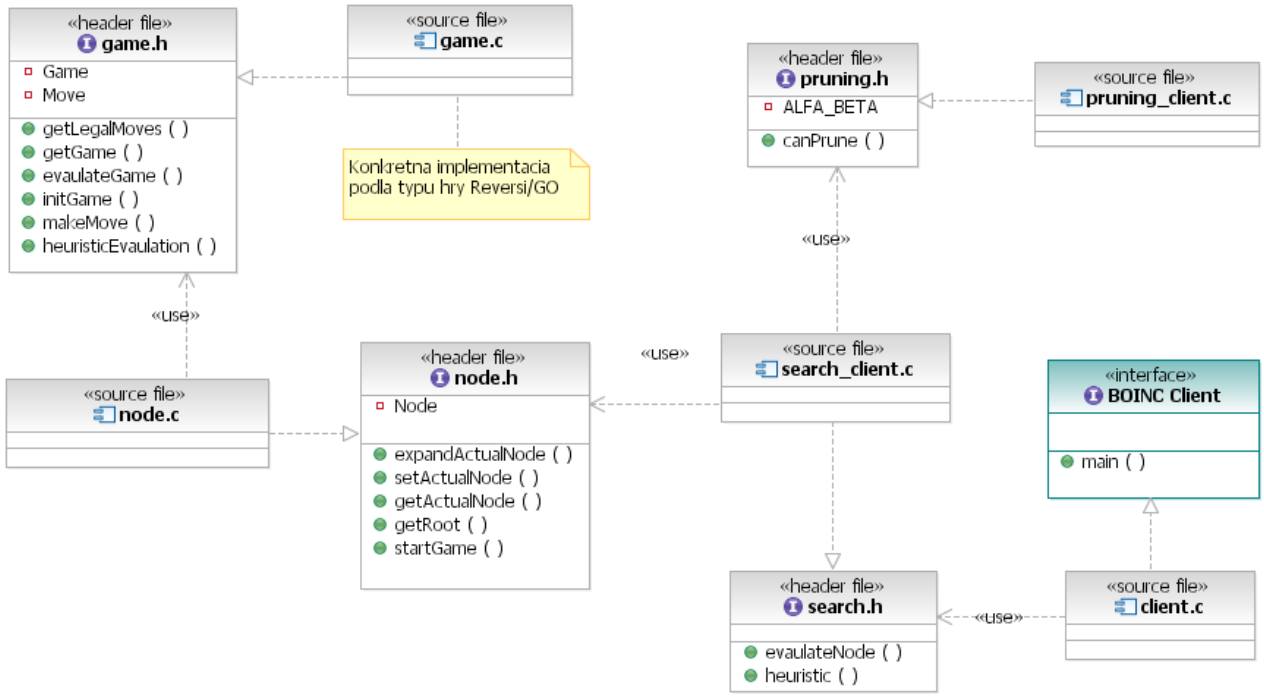
Funkcie

- `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. Sekvenčný 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 prehľadávanie stromu.
- `getRoot()` - vráti referenciu na špeciálny uzol - koreň stromu.
- `startGame()` - vytvorí koreňový uzol s počiatočnou hernou konfiguráciou - inštanciou typu hra.

8.3 Návrh klienta

Úlohou klienta je ohodnotiť časť stromu použitím algoritmu *minimax* (kapitola 5.2 MiniMax) spolu s usekávaním. Jednoducho prejde časť stromu a vráti jedno číslo, istý opis komplexného výsledku hry v danej časti stromu. Vstupom pre klienta, ako bolo povedané v úvode návrhu, je tzv. *workunit*, ktorý v podstate opíše vstupný uzol, ktorým má klient pri prehľadávaní začať, t.j. berie ho ako koreň

stromu. Chceli sme docieliť nezávislosť na hre, na výbere algoritmu osekávania a na výbere heuristiky. Heuristika je súčasťou už opísaného rozhrania *game.h*. Architektúra klienta je zhrnutá na nasledujúcom obrázku (Obr. 30), ktorý obsahuje hlavné časti - komponenty a ich závislosti.

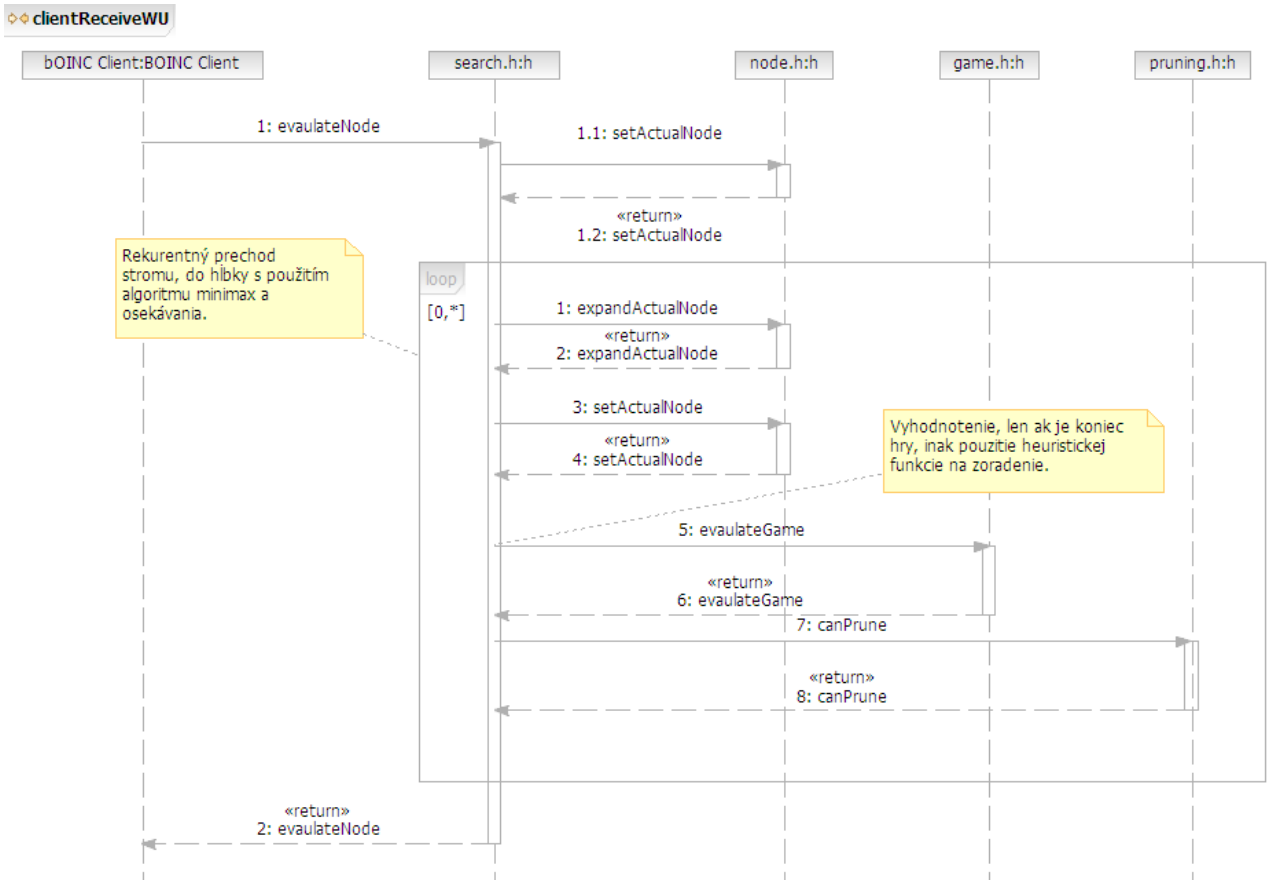


Obr. 30: Architektúra klientskej strany

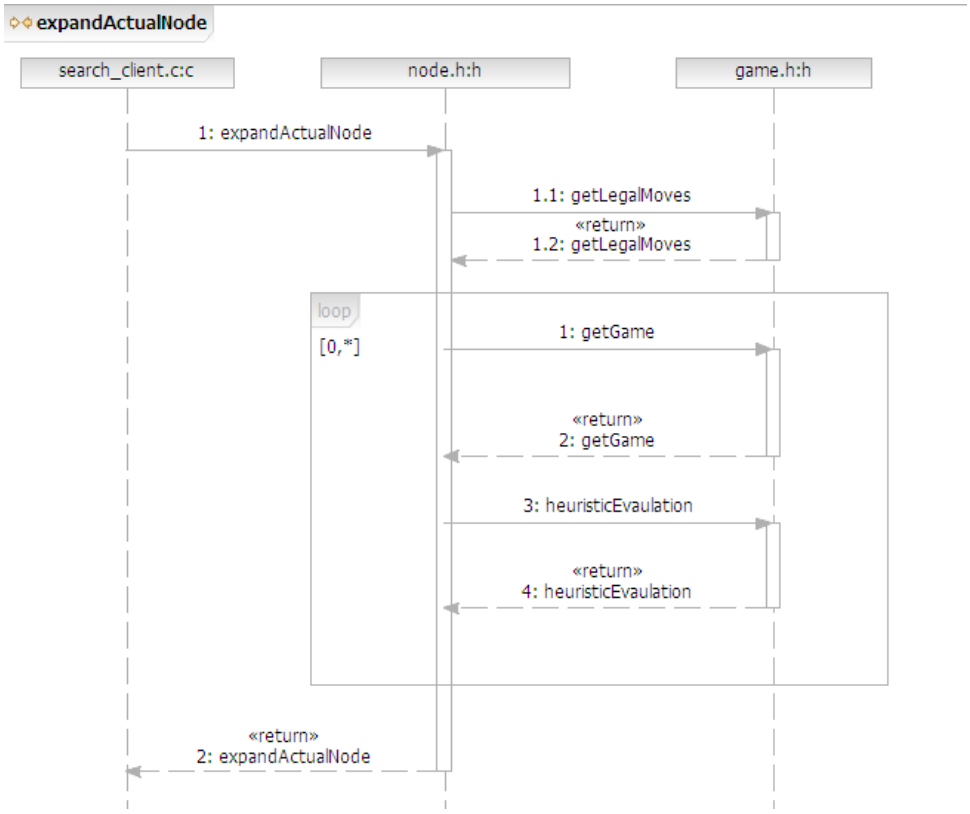
Rozhrania *game.h* a *node.h* boli už podrobnejšie popísané. Komponent **game.c** je konkrétna implementácia rozhrania *game.h* a popisu pravidla hry, v našom prípade Reversi a Go (pozri kapitolu 2.4). Podobne komponent node.c, ktorá implementuje manipuláciu s uzlami stromu a samotným stromom. Je zodpovedný za referencie na prehľadávaný strom, najmä koreň stromu. Popis ďalších častí 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 byť striktne alfa-beta osekávanie.
- *search.h* - v podstate hlavné rozhranie pre klienta, t.j. pre komponentu client.c. Zapúzdruje operácie a atribúty potrebné pre prehľadávanie stavového stromu. Je spúštané funkciou `evaulateNode(Node)`, ktorá má vstupný parameter uzol určený na vyhodnotenie klientom. Vracia číselnú hodnotu pre výsledok hry začínajúcej konfiguráciou, ktorú obsahuje tento uzol.
- **search_client.c** - konkrétna implementácia predchádzajúceho rozhrania, vyžaduje popis stromu a jeho inštancii (node.h), akým spôsobom bude osekávať nepotrebné vetvy (pruning.h).

- **client.c** - vykonateľný klientský kód u klienta, stručne povedané, tu sa všetko pre klienta začína a končí, od prijatia *workunit* (počiatočného uzla) po odoslanie výsledku (ohodnotenie uzla). Tento komponent sa podriaďuje pravidlám BOINC-u, na diagrame je to znázornené implementáciou externého rozhrania. Sekvenčný diagram (Obr. 31) opisuje udalosti pri prijatí *workunit*-u klientom. Časťou tohto diagramu je funkcia `expandActualNode()`, ktorá nieje triviálna a jej bližší popis poskytuje sekvenčný diagram (Obr. 32).



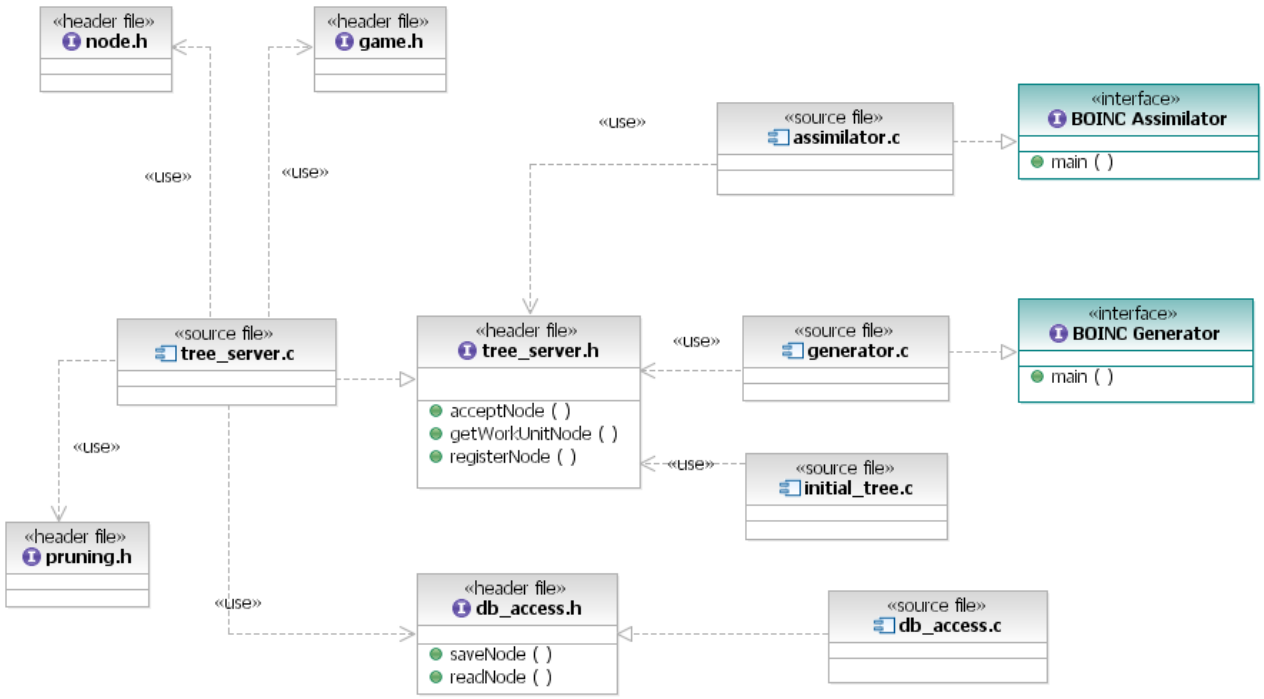
Obr. 31: Sekvenčný diagram popisujúci prijatie práce (workunit) klientom



Obr. 32: Sekvenčný diagram pre expandovanie jedného uzla

8.4 Návrh server

Ako už bolo spomenuté na začiatku návrhu, server je delený na dve hlavné časti 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.

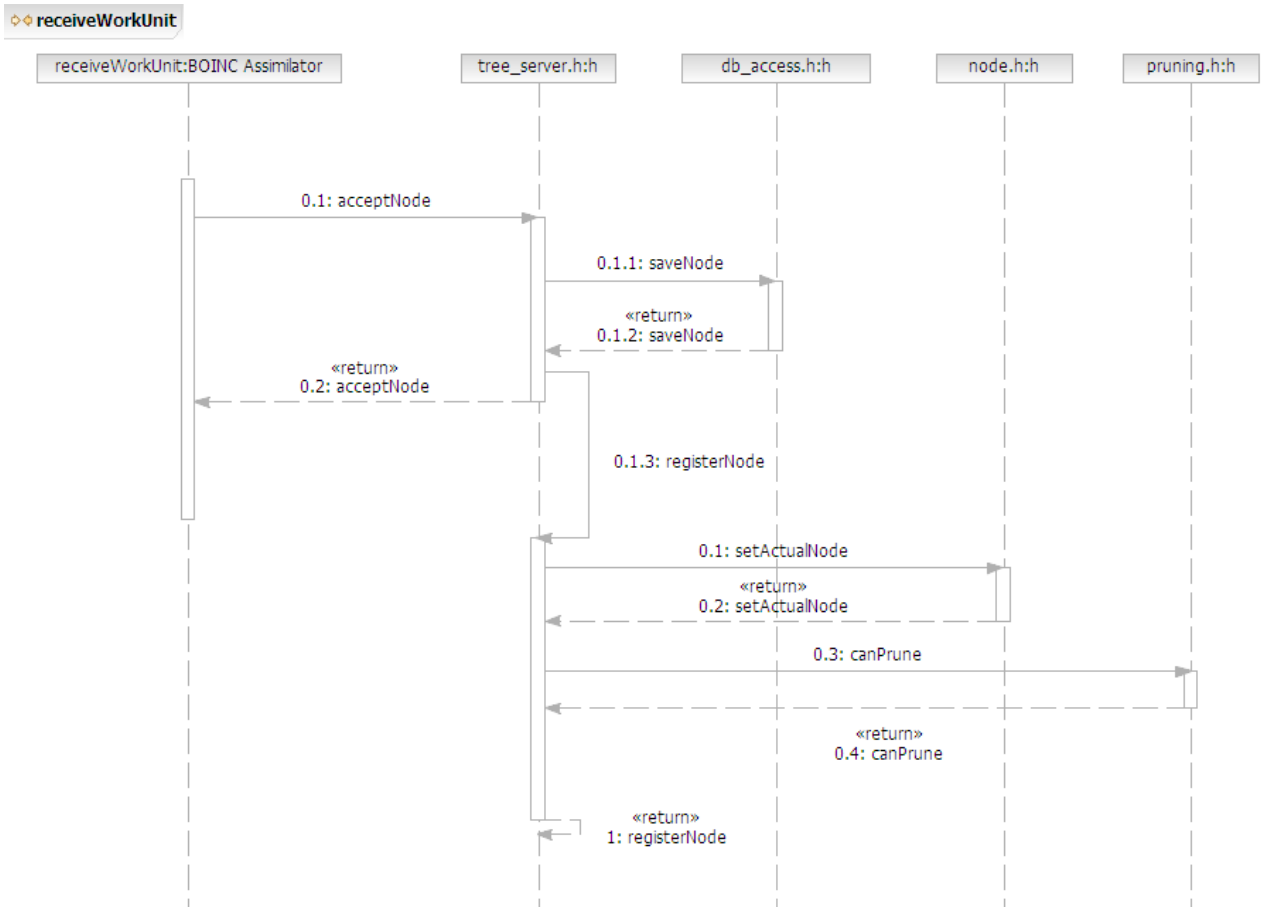


Obr. 33: Diagram komponentov pre server

Nasleduje popis jednotlivých častí servera. Rozhrania na komunikáciu s klientom node.h a game.h boli opísané v predchádzajúcich častiach. Ďalšie časti servera:

- *tree_server.h* - komplexné rozhranie pre celý server, popisujúce prehľadávaný strom. K jeho zodpovednosti patrí prijatie ohodnoteného uzla od klienta (`acceptNode()`) a jeho zaregistrovanie (`registerNode()`), čo je lokalizácia v strome, uloženie, prípadne spustenie usekávania. Heuristika na vyžiadanie generátora vráti klientovi podľa preferencií uzol na prehľadanie, inak povedané, hlavnú časť *workunit*-u (`getWorkUnitNode()`).
- **tree_ ser ver.c** - implementácia rozhrania *tree_server.h*, dá sa povedať, že táto komponenta je akési srdce celého servera. Uchováva informácie o práve skúmaných uzloch, vie, ktorá časť celého stavového priestoru bola preskúmaná a ohodnotená. Registruje ohodnotené uzly od klientov, to môže spôsobiť osekávanie istej časti stromu, z toho dôvodu má k dispozícii rozhranie *pruning.h*.
- *pruning.h* - rozhranie osekávania, ktoré môže, ale nemusí, mať takú istú implementáciu ako klient.
- *db_access.h* - rozhranie pre prístup k disku, konkrétne k databáze uložených uzlov. Poskytuje funkcie na čítanie 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 operačnej 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.

- **init ial_t ree. c** - jednoduchá a priamočiara časť, pred spustením samotnej BOINC komunikácie klient-server je vygenerovaná istá časť stromu, ktorej koncové uzly sú posielané klientom.
- **genera tor .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 prehľade systému BOINC (kapitola 3.2 Démoni bežiaci na serveri). Názorný popis prijatia výsledku od klienta je na Obr. 34.



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

9 Prototyp

Rozhodli sme sa vytvoriť prototypy v dvoch zatiaľ oddelených častiach:

- Prototyp prehľadávania stavového priestoru na jednom stroji zatiaľ iba pre hru *reversi*, pre získanie informácie o veľkosti stavového priestoru o závislosti faktoru vetvenia od hĺbky v strome, t.j. od čísla ťahu hry.
- Prototyp pre systém BOINC, triviálny program pre otestovanie funkčnosti a získanie informácií o vytváraní programu s paralelným počítaním.

Preto je náš prototyp odlišný od klasického prototypu informačného systému, sú to v podstate dva oddelené programy. Výsledný systém bude predstavovať spojenie týchto častí.

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 byť základ pre ďalšie programovanie samotnej aplikácie.

Najprv bol implementovaný mechanizmus pre definovanie hry (v našom prípade *reversi*), to spočívalo 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 hĺbky, odhad faktoru vetvenia.

9.1.1 Celý strom do istej hĺbky

Pokus o vygenerovanie čo najhlbšieho stromu na jednom počítači v rozumnom čase (tým je myslené niekoľko minút). Toto bolo dosiahnuté do hĺbky deväť s výsledkami, ktoré popisuje Tab. 4. Obsahuje počet uzlov v jednotlivej hĺbke stromu a k tomu dopočítaný faktor vetvenia v danej hĺbke ako pomer počtu uzlov v susedných hĺbkach.

Ako vidieť, v hĺbke deväť je potrebné uchovať si takmer tri a pol milióna uzlov, čo pre klasický počítač je relatívne dosť. Cieľom tejto časti bolo najmä odhadnúť rozsah serverom uchovávanej časti stromu. Rátame aj s tým, že v nasadenej aplikácii ešte minimalizujeme množstvo pamäti pre uchovanie jedného uzla.

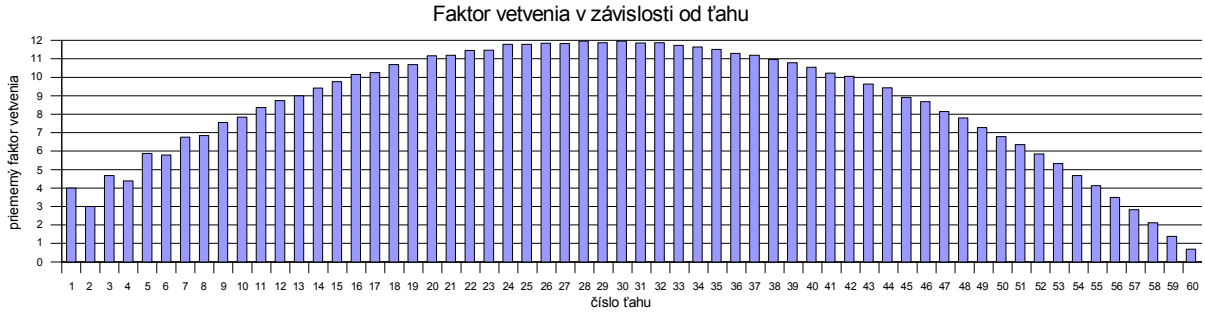
3 http://www.eclipse.org/cdt/
4 http://www.mingw.org/

Hĺbka	Počet 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 dosiahnuteľných hĺbkach

9.1.2 Odhad faktoru vetvenia

Druhý experiment a cieľ prototypu bolo získať faktor vetvenia pre každú hĺbku, a tým nepriamo odhad veľkosti celého stavového priestoru pre hru Reversi. Toto bolo riešené náhodným prechodom stavového priestoru až do najnižšej hĺbky - koniec hry. Po niekoľkonásobnom zopakovaní (minimálne tisíc krát) bol vyrátaný priemerný faktor vetvenia pre každú hĺbku, t.j. každý ťah. Výsledky tohto pokusu sú znázornené na Obr. 35



Obr. 35: Graf závislosti faktoru vetvenia od čísla ťahu

Ako vidieť z grafu, faktor vetvenia sa najprv zvyšuje, maximum dosahuje asi v strede hry, pri ťahu okolo tridsať. Tu ešte chceme poukázať na faktor vetvenia na konci hry, je menší ako jeden, čo je logicky podložené tým, že ku koncu môže nastať situácia, že hráč na ťahu nemá podľa pravidiel možný ťah. Ďalším poznatkom z tohto pokusu je odhad veľkosti stavového priestoru hry Reversi, zrátaním sumy pre všetky hĺbky použitím zistených faktorov vetvenia sme dospeli k číslu $2{,}27x10^{53}$, čo je dosť veľké číslo, ale bolo očakávané.

9.2 Prototyp aplikácie pre BOINC

Platformu BOINC si bolo potrebné pred samotnými úvahami o spôsobe riešenia úlohy „ohmatať“. Na našom serveri sme teda okrem podporných nástrojov ako dotProject nainštalovali i funkčný BOINC server a vytvorili si projekt test1. Projekt bežal bez incidentov, bolo sa možné k nemu prihlásiť, ale neobsahoval žiadnu klientsku aplikáciu a ani žiadne úlohy. Bolo potrebné sa s touto

platformou bližšie zoznámiť a preto sme si v rámci analýzy vyskúšali skompilovať a zverejniť klientsku aplikáciu. Pokusnou aplikáciou bola ukážková aplikácia „uppercase“, ktorej úlohou bolo prijať ako úlohu akýkoľvek textový vstup a transformovať ho na výstup s pôvodným znením, avšak so všetkými znakmi „veľkými“. Táto ukážková aplikácia pre svoje spomalenie počítala po každom transformovanom znaku sumu zo sínus pí, krát číslo iterácie v cykle s miliónom krokov. Ďalej táto aplikácia používala „checkpointing“ (pozri Príloha A – Slovník pojmov) a previazanie na grafickú časť. Aplikáciu sa nám podarilo úspešne skompilovať i zverejniť na BOINC projektovom serveri. Funkčnosť sme overili pripojením dvoch BOINC klientov a vypočítaním niekoľko desiatok úloh. Body sa pripočítavali 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 časti 5 Teoretický základ pre riešenie hier sa na klientovi použije
- na základe hĺbky stromu, ktorý klient dokáže prehľadať, ktorý mechanizmus bude potrebné použiť pri generovaní stromu hľadania na serveri a vytváraní úloh pre klientov.

Do úvahy prichádzajú nasledovné prístupy generovania stromu na serveri:

- vygenerovať na začiatku strom na serveri do určitej pevne danej hĺbky
- dynamicky generovať strom , nevygenerovať celú šírku stromu na každej hĺbke, ale zísť do stromu hlbšie, aby klientom ostal zvládnuteľný menší strom hľadania

Nakoniec, po konzultácií s vedúcim projektu, sme sa však rozhodli, že prototyp bude predstavovať zjednodušenú verziu celého systému riešenia a nebudeme sa sústrediť na jednú špecifickú časť. Z toho sme stanovili, že cieľom prototypu je vytvoriť BOINC projekt, ktorý bude schopný vyriešiť distribuovaným spôsobom hru reversi. Prototyp má takto overiť principiálnu funkčnosť navrhnutého prístupu. Keďže sa v tejto časti nebudeme sústrediť na žiadne optimalizácie ani pokročilé algoritmy, systém bude podľa odhadov schopný vyriešiť hru reversi do rozmerov 4x4. Riešenie načrtnutých otázok bude presunuté do nasledujúceho semestra.

Vrámci prototypu vytvárame nasledovné časti 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 určila na ukladanie dát databáza. Sú v nej uložené iba údaje priamo sa týkajúce riešenej úlohy. Réžijné údaje spojené s mechanizmom rozdeľovania 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 dostať
- ohodnotenie konfigurácie
- ktorý hráč je v danom ťahu na rade

Nami zvolené kódovanie hernej pozície je takéto:

Každé políčko môže mať 3 rôzne ohodnotenia:

- kameň bieleho hráča (01)
- kameň čierneho hráča (00)
- prázdne políčko (10)

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

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

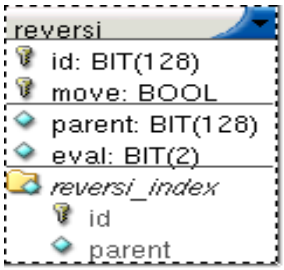
Pole informujúce o hráčovi, ktorý je na rade, môže mať 2 rôzne hodnoty, preto na jeho kódovanie používame typ boolean.

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

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

Na kódovanie ohodnotenia preto používame 2 bity.

Fyzický model databázy



Obr. 36: Fyzický model databázy

Primárnym kľúčom sú polia id a move. Pole id samo o sebe ako primárny kľúč nestačí, dve rovnaké pozície na hernej ploche sa totiž od seba môžu líšiť tým, ktorý hráč je na ťahu.

Keďže potrebujeme vyhľadávať v DB podľa aktuálnej pozície a podľa rodičovskej pozície, tieto dve polia sú indexované.

Databázu tvorí jediná tabuľka 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 aplikačné programátorské rozhranie pre aplikácie napísané v jazyku C, ktoré je priamo súčasťou 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, načítanie uzla z databázy a načítanie potomkov uzla z databázy.

9.3.2 Klient

Pod klientom sa v tejto časti myslí klientska aplikácia. Teda aplikácia, ktorá je za pomoci BOINC systému distribuovaná klientom, ktorí ju následne spúšťajú a vracajú jej výsledky na server. Implementácia klienta spočívala prakticky v upravení prvého prototypu, ktorý počítal priemerný faktor vetvenia tak, aby počítal celú hru a nie iba priemerný faktor vetvenia. Po tejto úprave bolo nutné doplniť 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 častí knižníc je v nasledujúcich podkapitolách.

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

Algoritmus riešenia hry

Tak, ako bolo spomenuté, klientska časť prototypu sa nesústredí na žiadne optimalizácie ani pokročilé algoritmy a techniky ohodnocovania stromu hľadania symetrickej hry. Za algoritmus na strane klienta sme preto zvolili jednoduchý a ľahko implementovateľný MiniMax algoritmus. Použitie tohto algoritmu automaticky obmedzilo triedu riešiteľných problémov na problémy s malým stavovým priestorom. Na túto skutočnosť je potrebné pri testovaní prototypu myslieť, nakoľko MiniMax algoritmus musí navštíviť každý uzol stromu prehľadávania.

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 starať nemusíme, zabezpečuje ju BOINC mechanizmus. Štruktúru súborov sme stanovili nasledovne:

- vstupný súbor – obsahuje stav pre ktorý sa bude počítať ohodnotenie a druhá hodnota je ktorý hráč je na ťahu. Je generovaný generátorom úloh a je vstupom pre počítanie výsledku na klientovi.
- výstupný súbor – obsahuje údaj pre ktorú úlohu sa daný výsledok vypočítal 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 „ručné“ 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 zabezpečí 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 počet flopov (flops) potrebných pre výpočet daného workunitu. Ak je tento parameter nastavený na malé číslo, tak výpočet skončí hláškou o prekročení časového limitu. Výpočet vhodného horného ohraničenia si ponechávame na letný semester.

Druhý spôsob generovania úloh je vygenerovať ich programovo, skopírovať ich na vhodné miesto a zaregistrovať. Táto programová metóda bola implementovaná, ale zatiaľ nebola využívaná. Bolo potrebné použiť niekoľko BOINC knižníc a ich volania. Pri generovaní úloh si ukladáme celý stavový priestor hry do databázy, až po úroveň, na ktorej generujeme z listov stromu úlohy. Teda

prehľadávame stavový priestor hry do hĺbky a ukladáme si do našej databázy všetky vygenerované stavy. V prípade, že hĺbka aktuálneho uzla je rovná číslu 3, tak daný uzol uložíme na disk ako úlohu a nevykonáme generovanie nasledovníkov.

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

Tretím spôsobom je generovanie úloh (workunit-ov) na požiadanie. Teda, keď klesne počet neodoslaných úloh pod nejakú hranicu, „dogeneruje“ sa daný počet úloh na požiadanie. Tento spôsob zatiaľ neplánujeme využiť.

9.3.4 Asimilátor

Asimilátor je jednou z dvoch hlavných súčastí servera. V prototype sme sa sústredili na vytvorenie jednoduchého asimilátora, ktorý bude ukladať prijaté výsledky od klientov do databázy. Implementovanie asimilátora spočíva v napísaní jednej konkrétnej funkcie, ktorá je BOINC-om volaná pri obdržaní výsledku. Hlavička funkcie má tvar:

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

Kde typy WORKUNIT a RESULT sú súčasťou BOINC knižnice. Pre nás je dôležité, že RESULT obsahuje referenciu (cestu) k súboru, ktorý bol vytvorený klientom a obsahuje výsledok. Po načítaní hry (game) z tohto súbora je hra nájdená v databáze. Ak je výsledok relevantný, tak v databáze musí byť 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.

Cieľom prototypu bolo iba otestovať funkčnosť a napojenie na databázu. Výsledný produkt bude samozrejme v spomínanej asimilačnej funkcii obsahovať viac logiky, napr. vyhodnotenie uzla aj v zmysle možného osekávania susedných uzlov.

10 Zhodnotenie

V zimnom semestri sme sa venovali analýze problémovej oblasti, ktorá zahrňuje popis BOINC systému, algoritmami riešenia hier, analýzou hier reversi, go a ich existujúcimi riešeniami. Ďalej diskutujeme možnosti uchovávania údajov.

Následne uvádzame hrubý návrh systému, ktorým určujeme architektúru systému, jednotlivé programové moduly a ich vzájomné väzby.

Zavŕšením práce v zimnom semestri predstavuje prototyp. Vytvorili sme ho v dvoch fázach. Prvý prototyp predstavoval program pre jeden počítač, ktorým sme odhadli faktor vetvenia stromu hľadania. Druhý prototyp je implementácia BOINC projektu na distribuované riešenie symetrickej hry. Toto riešenie neobsahuje žiadne optimalizácie ani zložitejšie techniky a umožňuje teda iba riešenie jednoduchých hier.

V letnom semestri je potrebné zamerať sa práve na vylepšenie klientskej časti riešenia, aby bolo schopné využiť pokročilejšie algoritmy na riešenie hier a tak umožniť riešenie reversi 8x8.

11 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 prehľadávania hry. Vylepšenie algoritmu MiniMax založené na useknutí uzlov, ktoré nemôžu ovplyvniť výsledok. (angl. *alpha beta pruning*)

Canonical Result – Je to výsledok (result), ktorý vytvorí validátor (alebo za neho označí nejaký existujúci) pri validácii prišlých result-ov. Je to akýsi zástupca výsledku. Podľa aplikácie ním môže byť napríklad priemer všetkých validných výsledkov, alebo prvý výsledok doručený na server.

Dobrovolník – Užívateľ vlastniaci minimálne jeden počítač, ktorého výkon chce poskytnúť pre BOINC projekty.

Faktor vetvenia – priemerný počet detí uzla v strome hľadania

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

GO – stolová hra pre dvoch hráčov, v ktorej sa hráči snažia ovládnuť čo najväčšie územie

Heuristika – odporúčanie, ktoré má viesť k dobrému výsledku

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

Checkpointing - metóda, pri ktorej si klientska aplikácia z času načas (určuje BOINC klient) ukladá na disk aktuálny stav výpočtu s cieľom pokračovať z aktuálneho bodu po vypnutí a následnom zapnutí aplikácie.

Klient – BOINC klient, teda aplikácia, za pomoci ktorej sa prihlasuje na PC dobrovoľníka do projektov. Táto aplikácia sťahuje z projektových serverov klientske aplikácie, workunit-y a manažuje celú komunikáciu i zdieľanie prostriedkov viacerými projektami.

Klientska aplikácia – je výpočtovo náročná aplikácia, ktorú distribuuje BOINC server a na počítač dobrovoľníka sa dostane za pomoci BOINC klienta

MiniMax – základný algoritmus na ohodnotenie stromu prehľadávania hry

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

MTD(f) – vylepšený algoritmus ohodnotenia stromu prehľadávania hry založený na hľadaní s minimálnym oknom

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

Okno hľadania – dvojica (α,β) ktorá pri algoritmoch zalozených na alfa beta usekávaní určuje, ktoré uzly má zmysel prehľadávať (angl. *search window*)

Othelo – obdoba hry reversi, pri ktorej je však jednoznačne určená začiatočná pozícia štyroch kameňov na hracej ploche

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

Reversi – stolová hra pre dvoch hráčov, v ktorej si hráči navzájom preberajú figúrky.

Strom hľadania – strom, ktorý sa vytvára pri prehľadávaní stavového priestoru.

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

Transpozičná tabuľka – tabuľka, v ktorej sa ukladajú ohodnotenia už prehľadaných uzlov stromu hľadania

Veľmi slabé riešenie – je určenie, ktorý z hráčov hru vyhrá *(*angl. *ultra-weak solution)*

Vyhladenie – stav, keď jeden z hráčov už nemá žiadne kamene

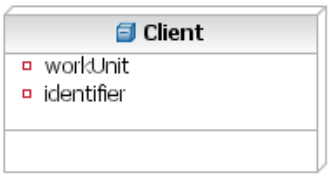
Workunit – súbor, ktorý obsahuje úlohu pre klientsku aplikáciu. Niekedy sa takto nazýva i samotná úloha.

UML – Unified Modeling Language - štandardný špecifikačný jazyk

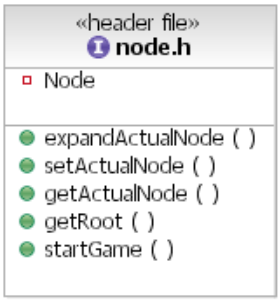
CRUD - štyri základné databázové funkcie (skratka z angl. Create, Read, Update, Delete) popisuje možné operácie s databázou.

Príloha B – Použitá notácia diagramov

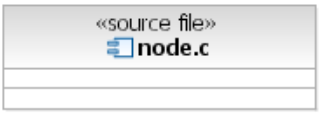
Z dôvodu implementácie projektu čisto v jazyku C, bez použitia objektových štruktúr, je použitá notácia istou nadstavbou štandardu UML.
Objekty a asociácie v diagramoch



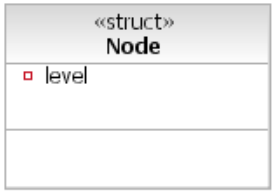
- jednotka systému, opisuje jeden fyzický stroj - počítač
- má svoje vlastnosti (▫), v našom prípade súčasti na najvyššej úrovni abstrakcie



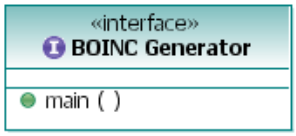
- rozhranie (*interface*), pri C programovaní (náš prípad) predstavuje jeden hlavičkový súbor
- obsahuje mená štruktúr (▫) v jazyku C
- v spodnej časti prototypy funkcií (●)



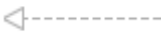
- notácia pre zdrojový súbor v jazyku C



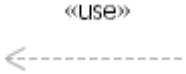
- notácia pre klasickú C štruktúru
- štruktúra obsahuje premenné (▫)



- notácia pre vzdialené rozhranie, predstavuje pravidlá pre konkrétnu implementáciu



- realizácia, v našom prípade istý zdrojový súbor implementuje daný hlavičkový súbor



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