IMPLEMENTACIJA IGRE I AGENTA POMOĆU MINIMAX ALGORITMA

Radak, Frane

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

FRANE RADAK

Z A V R Š N I R A D

IMPLEMENTACIJA IGRE I AGENTA POMOĆU MINIMAX ALGORITMA

Split, rujan 2023.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

Predmet: Strukture podataka i algoritmi

Z A V R Š N I R A D

Kandidat: Frane Radak

Naslov rada: Implementacija igre i agenta pomoću minimax algoritma

Mentor: dr. sc Toma Rončević, v. pred.

Split, rujan 2023

Sadržaj

Sažetak

Tema ovog rada je implementacija igre šah i agenta koji igra šah korištenjem minimax algoritma. Cilj rada je prikazati teoretske i praktične korake kreiranja šahovskog motora (engl.*chess engine*). Rad obuhvaća sve ključne korake implementacije samog motora potkrijepljene teoretskim osnovama same problematike. Kako se tema klasificira kao algoritamski problem, bilo je potrebno proučiti osnovne principe rada sljedećih algoritama: osnovni algoritmi pretrage, binarna pretraga, minimax algoritam te alfa-beta podrezivanje. Također, bilo je potrebno proučiti praktičnu upotrebu navedenih algoritama te njihovu adaptaciju unutar šahovske problematike, uzevši u obzir da su spomenuti algoritmi generički. Rad je u cijelosti pisan u C++ jeziku, te je zbog iznimne algoritamske složenosti problema bilo potrebno napredno poznavanje samog programskog jezika.

Ključne riječi: C++, minimax, optimizacija, šah

Summary

Implementation of game and agent using minimax algorithm

The topic of this paper is the implementation of the chess game and the chess-playing agent using the minimax algorithm. The aim of the work is to show the theoretical and practical steps of creating a chess engine. The paper covers all the key steps of implementing the engine itself, supported by the theoretical foundations of the problem itself. As the topic is classified as an algorithmic problem, it was necessary to study the basic working principles of the following algorithms: basic search algorithms, binary search, minimax algorithm, and alpha-beta pruning. It was also necessary to study the practical use of the mentioned algorithms and their adaptation within chess problems, given that the mentioned algorithms are generic. The work was written entirely in C++ language, and due to the exceptional algorithmic complexity of the problem, an advanced knowledge of the used technology was required.

Keywords: C++, chess, minimax, optimization

1. Uvod

1.1 Šah

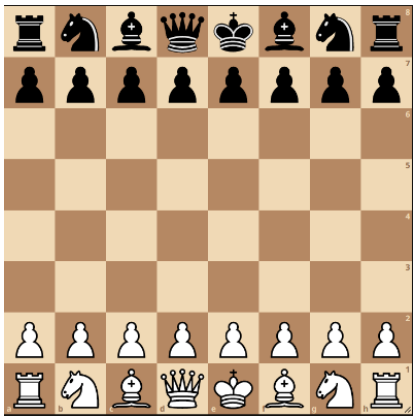
Prvo pojavljivanje šaha zabilježeno je u Indiji 600.g novog doba, pod imenom chaturanga (ind.*chaturanga – 4 odvojene vojske*). Nakon pojavljivanja, šah se do 15.stoljeća širi duž Europe i Kine, te postaje jako popularan.

Antički šah se uvelike razlikovao od šahovske igre koju poznajemo danas. Naime, u Indijskom šahu figure su imale drugačija imena od onih koje danas poznajemo, te se tadašnja reprezentacija pješaka nije mogla pomicati za 2 polja naprijed, što je danas slučaj. Međutim, iako je od inicijalnog kreiranja šaha prošlo više od 1000 godina, i dan danas se u raznim literaturama mogu pronaći korijeni izvorne igre u imenima pojedinih varijanti otvaranja (kraljeva-indijska obrana, damina-indijska obrana, staro-indijska obrana itd…)

Šah ili *kraljevska igra* ili *igra milijuna,* kako je popularno nazvan, u svojim početcima klasificirao se kao igra za bogate, sve do 18. stoljeća kada su pravila šaha definirana na način koji danas poznajemo, što je uvelike pridonijelo popularizaciji same igre.

Šah je igra na 64 (8 x 8) polja od kojih su 32 polja bijele bolje, dok su ostala 32 crne boje. Šahovska ploča sastoji se od redova (1, 2, 3, 4, 5, 6, 7, 8) i linija (a, b, c, d, e, f, g, h), te je potpuno simetrična. Na šahovskoj ploči, u početnom trenutku svaka strana raspolaže sa 16 figura, od kojih je 8 pješaka, 2 lovca, 2 topa, 2 skakača, 1 dama i 1 kralj. Cilj igre je protivniku *dati* "šah mat". "Šah mat" ili *kralj je mrtav* stanje je u kojem je protivnički kralj napadnut i nema niti jedno slobodno polje za bijeg, niti se može zaštiti drugom figurom iste boje. Početno stanje šahovske ploče prikazano je na slici 1.

Najveća popularizacija šaha nastala je za vrijeme “Hladnog rata”, odnosno dvoboja za svjetskog prvaka između Američkog velemajstora Bobbya Fischera i Ruskog velemajstora Borisa Spasskya.



Slika 1: Početno stanje šahovske ploče

1.2 Šahovska računala

Kako se šah popularizirao tako su se dobre prakse i temeljni koncepti igre jako brzo mijenjali. Šahovska teorija jako se brzo razvijala, te su turnirske i profesionalne partije postale sve kompleksnije. Upravo zato, krajem 60-ih godina prošlog stoljeća dolazi do razvoja šahovskih računala.

Prvo moderno šahovsko računalo/program “Deep Blue” razvijeno je 1996. godine. Kako je “Deep Blue” predstavljao veliki iskorak u svijetu šahovskih računala tako je 1996. godine u Philadelphiji organiziran meč *Deep Blue – Gary Kasparov* (Gary Kasparov tada je bio formalni svjetski prvak u šahu, te najviše rangiran igrač prema FIDE ljestvici). Prvi meč završio je rezultatom 2 – 4, odnosno pobjedom Garya Kasparova. Ubrzo nakon, 1997.godine u New Yorku nadograđena inačica “Deep Blue” odigrala je uzvratni meč protiv Gary Kasparova. Uzvratni meč završio je rezultatom 3.5 – 2.5 u korist “Deep Blue", te je “Deep Blue” postalo prvo računalo u povijesti koje je pobijedilo svjetskog prvaka.

“Deep Blue” korišten je na IBM-ovom RS/6000 SP superračunalu pogonjeno PowerPC 604e čipom, te posebno kreiranim VLSI čipom koji je omogućavao paralelizaciju pri izvođenju alfa-beta podrezivanja. “Deep Blue-ova” funkcija evaluacije bila je poprilično generalizirana, odnosno oslanjala se na analizu pohranjenih partija, te je imala specijalne parametre za određene tipove pozicije. “Deep blue” koristio je knjigu otvaranja s preko 400 pozicija i bazu podataka koja sadržava preko 70 000 velemajstorskih partija. Mogao je obraditi do 200 milijuna pozicija po sekundi [1].

Problem ranih šahovskih računala bio je upravo generalizacija pozicije. Naime, zbog nedostatka procesorske snage i memorije bilo je jako teško unutar samog računala ugraditi sve potrebne principe, već su se isti uzimali zdravo za gotovo.

Motivacija za izradu šahovskog motora je prvenstveno ovladavanje složenim algoritamskim problemima. Glavni problem u implementaciji je obrada velikog broja pozicija u jako kratkom vremenu, uz ograničen memorijski prostor.

U nastavku ovog dokumenta detaljno će biti objašnjen proces kreiranja šahovskog motora, tako da se obuhvate sve komponente: teorijska podloga, prezentacija ploče i figura, implementacija funkcije evaluacije, implementacija minimax algoritma uz optimizaciju korištenjem alfa-beta podrezivanja, te povezivanje cijelog motora s grafičkim sučeljem.

2.Teorijski dio

2.1 Korišteni koncepti

2.1.1 Pojam i upotreba rekurzije

Rekurzija ili rekurzivna funkcija, prvenstveno je definirana kao matematički pojam. Točnije, rekurzija (lat.*recursio - rekurzija*), u logici i matematici, postupak je kojim se pomoću neke funkcije (rekurzivne funkcije) ili izraza (rekurzivne formule) opetovanim postupkom dolazi do jednostavnijeg oblika matematičkih izraza ili rješenja zadane jednadžbe [2].

Matematička rekurzija definirana je kao klasa funkcija koja djeluje nad skupom prirodnih brojeva [3]. Glavna karakteristika rekurzivnih funkcija je postojanje krajnjeg uvjeta, odnosno "*terminalnog stanja*". Terminalno stanje koristi se kako bi se mogao odrediti kraj djelovanja funkcije. Rekurzija se može prikazati pomoću primjera izračunavanja faktorijela prirodnog broja 5. Cijeli postupak možemo podijeliti u 3 koraka:

1. **Utvrđivanje terminalnog stanja**:
   Kako računamo faktorijel prirodnog broja terminalno stanje je $f(0) = 1$
2. **Izvršavanje rekurzije**

$$
\begin{aligned}
f(5) &= 5 \cdot f(4) \\
&= 5 \cdot \big(4 \cdot f(3)\big) \\
&= 5 \cdot \big(4 \cdot (3 \cdot f(2))\big) \\
&= 5 \cdot \Big(4 \cdot \big(3 \cdot (2 \cdot f(1))\big)\Big) \\
&= 5 \cdot \Big(4 \cdot \big(3 \cdot \big(2 \cdot (1 \cdot f(0))\big)\big)\Big)
\end{aligned}
$$

Kako smo utvrdili je da terminalno stanje $f(0) = 1$, korak 2 ovdje završava.

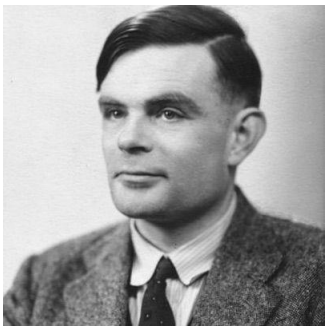
3. **Konačni izračun**

   U konačnici:

   $f(5) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

Prikazani koncept pogodan je za rješavanje brojnih matematičkih problema kao što su: Fibonaccijev niz, rekurzivne relacije, faktorijel, permutacije, kombinacije, teorija igara, itd….

Kada je riječ o praktičnoj uporabi rekurzivne funkcije, rekurzija se ponajviše upotrebljava unutar grane matematike po imenu "Teorija izračunljivosti". "Teoriju izračunljivosti" zasnovao je britanski znanstvenik Alan Turing (slika 2) 1930-ih godina pri izradi istoimene mašine, tzv. Turingova mašina.



Slika 2: Alan Turing

Teorija izračunljivosti dijeli se u 3 glavna područja:

1. **Teorija automata**

   Ova teorijska grana bitan je dio računalne znanosti, te je razvijena za potrebe kreiranja i razvitka apstraktne teorije računala. Osnovna svojstva automata su: stanje, ulazni podatak te izlazni podatak.

2. **Teorija izračunljivosti**

   Definira konačnost problema, tj. može li se problem riješiti bilo kojim apstraktnim strojem u konačnom vremenu.

3. **Teorija složenosti**

   Mjeri složenost algoritma, uključuje analizu kako bi se utvrdilo koliko je vremena potrebno za izvođenje algoritma [4].

Rekurzija, kao matematička formula, osnova je za rješavanje λ calculusa. λ calculus može biti korišten za predstavljanje proizvoljne "Turingove mašine". Turingova mašina dokazuje da ne postoji učinkovita metoda ili procedura za rješavanje *problema odluke*. *Problem odluke* u navedenom kontekstu bila bi odluka o prelasku u sljedeće stanje na osnovu ulaznih parametara, stanja, te algoritma odluke [5].

Kako bi bilo jasnije, tema ovog rada je jedan primjer problema odluke, rješavan rekurzivno. Cijeli rad može se promatrati kao logički automat sa svojim osnovnim svojstvima, gdje bi stanje bilo položaj figura unutar šahovske ploče, ulaz bi bio odigrani potez druge strane, dok bi izlaz bio najbolji potez generiran na osnovu algoritma koji bi se mogao definirati kao algoritam odluke.

Unutar računarstva, osnovna definicija rekurzivne funkcije bila bi *funkcija koja poziva samu sebe*. Glavna motivacija za korištenje rekurzivnih funkcija je mogućnost *razbijanja* složenih problema na niz manjih problema, te potom ponovno kombiniranje rješenja manjih problema u konačno rješenje. Primjer može biti uzet iz usporedbe algoritama za sortiranje. Konkretno, kao jednostavan primjer može biti uzet primjer usporedbe brzog sortiranja (engl.*Quick sort*) i mjehuričastog sortiranja (engl.*Bubble sort*).

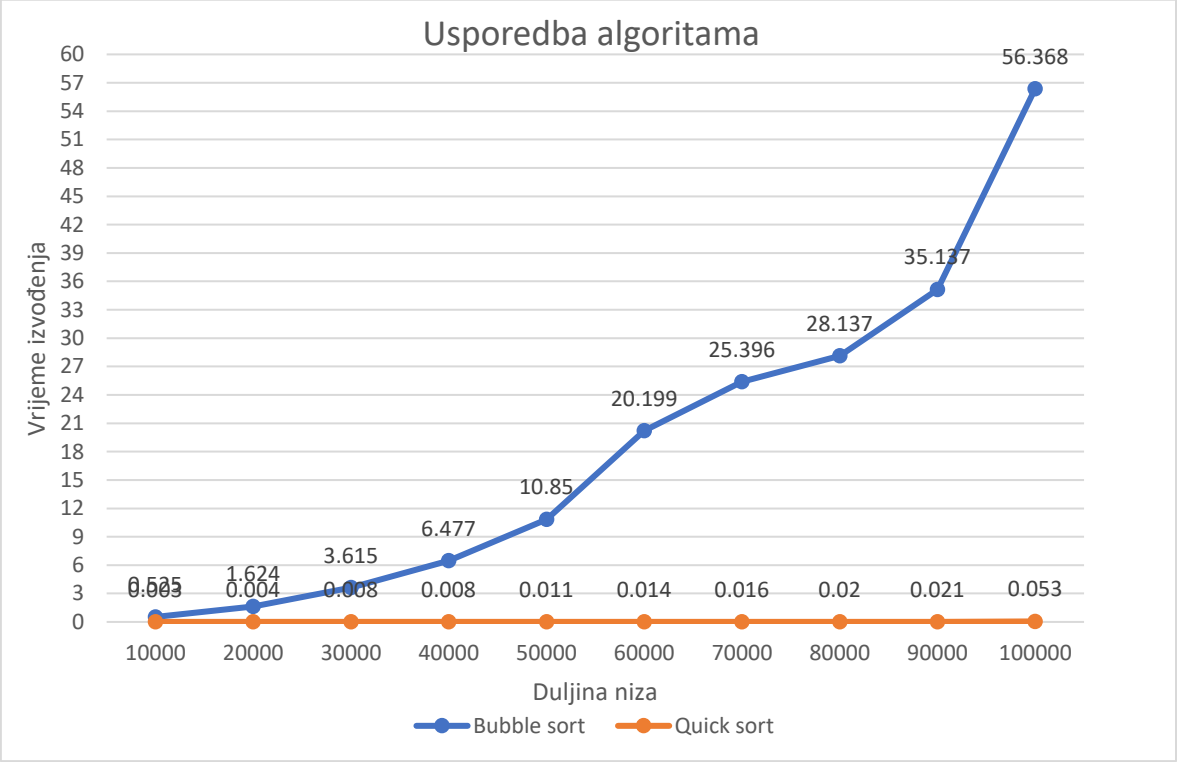
Generalno, u programiranju složenost algoritma mjeri se takozvanom "*O*" notacijom. "*O*" notacija je matematička notacija koja opisuje ograničavajuće ponašanje programa kada argument teži prema određenoj vrijednosti ili beskonačnosti [6]. Laički rečeno, što je vrijednost uz "*O*" veća, to je algoritam lošiji, odnosno pripada nižem razredu. Usporedbom brzog sortiranja (engl.*Quick sort*), te mjehuričastog sortiranja (engl.*Bubble sort*) može se doći do podataka:

Tablica 1: Usporedba kompleksnosti Quick sort I Bubble sort

	Quick sort	Bubble sort
najbolji slučaj	O(N*logN)	O(N)
prosječni slučaj	O(N*logN)	O(N*N)
najgori slučaj	O(N*N)	O(N*N)

Iz tablice 1 vidi se da je brzo sortiranje (engl.*Quick sort*) znatno bolje od mjehuričastog sortiranja (engl.*Bubble sort*). Kako bi stvar bila jasnija, dijagram 1 prikazuje usporedbu grafički.

Dijagram 1: Usporedba kompleksnosti Quick sort i Bubble sort



Dijagram 1 i tablica 1 dokazuju da je brzo sortiranje (engl.*Quick sort*) znatno brže u odnosu na mjehuričastog sortiranja (engl.*Bubble sort*). Bitno je napomenuti kako nepravilno korištenje rekurzivnih algoritama može dovesti do narušavanja performansi samog sustava. Naime, iterativna funkcija zahtijeva jedinstvenu alokaciju memorije na programskom stogu (engl.*stack*), dok rekurzivne funkcije u svakoj sljedećoj iteraciji alociraju novi blok memorije. Kada bi u gornjem mjerenju za duljine niza bile uzete puno manje vrijednosti jasno bi se vidjelo da bi mjehuričasto sortiranje (engl.*Bubble sort*) bilo brže od brzog sortiranje (engl.*Quick sort*). Rekurzivne funkcije sa sobom nose još jednu konsekvencu, a to je količina memorije koju zauzimaju. Kako je rečeno ranije, svaki poziv alocira novi memorijski blok na programskom stogu (engl.*stack*), te je zaključak da nije preporučljivo koristiti rekurzivne pozive na sustavima koji posjeduju malo memorije, a pogotovo na integriranim (engl.*embedded*) sustavima.

2.1.2 Algoritmi pretrage

Algoritmom pretrage smatra se bilo koji algoritam koji u skupu od *n* elemenata pronalazi jedan ili više elemenata koji zadovoljavaju postavljeni kriterij. Primjer: algoritam koji unutar skupa prirodnih brojeva traži prvo pojavljivanje broja prosljeđenog funkciji.

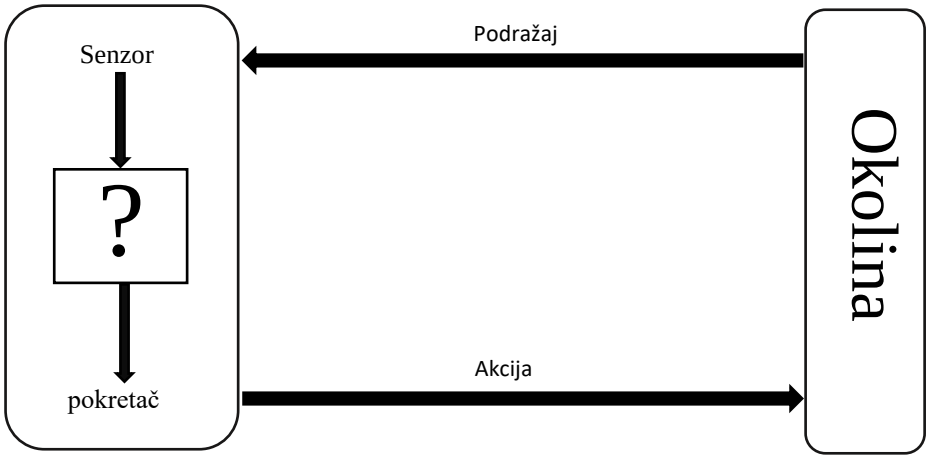
```
int pretrazi(const int* skup, int n, int trazeni_broj) {
    for (int i = 0; i < n; i++) {
        if (skup[i] == trazeni_broj) {
            return trazeni_broj;
        }
    }
    return -1;
}
int main(int argc, char*argv[])
{

    int skup[] = { 1,2,3,4,5,6,7,8,9,10 };
    //n odgovara broju elemenata skupa
    int n = sizeof(skup) / sizeof(skup[0]);
    int rezultat = pretrazi(skup, n, 5);
    if (rezultat >= 0) {
        std::cout << "Broj je pronaden" << std::endl;
    }
    else {
        std::cout << "Broj nije pronaden" << std::endl;
    }
    return 0;
```

Ispis 1: Algoritam pretrage

Ispis 1 tipični je primjer jednog sekvencijalnog algoritma pretrage. Sekvencijalni algoritmi pretrage imaju *O(N)* složenost, gdje bi *N* bila duljina skupa elemenata. Iz primjera je jasan problem ovog tipa pretrage. Naime, za pronalazak elementa potrebno je svaki put "*pregledati*" prosječno pola niza (traženi broj može biti na proizvoljnom indeksu).

2.1.3 Rekurzivno rješavanje problema

Rekurzivno rješavanje problema provodi se putem tzv. "*agenata*". Agent se može definirati kao logička cjelina koja podražaje i promjene koje se dogode u okolini detektira putem senzora, te pokretače (engl.*acutators*) koji odrađuju akcije i šalju odgovore natrag u okolinu. Agentove odluke o akciji donose se pomoću rekurzivnih algoritama. Agent u obliku robota kao senzore koristi kamere, mikrofone i razne ostale detektore, te određene elektro motore kao pokretače(engl.*acutators*). Ljudski agent također može biti uzet kao primjer, gdje bi oči, uši i nos bili senzori dok bi ruke noge i drugi organi bili pokretači (engl.*acutators*) [7]. Shema je prikazana na slici 3:



Slika 3: Shema agenta

Matematički gledano, može se reći da je agentovo ponašanje definirano funkcijom agenta koja pretvara razne sekvence podražaja u akcije. Zapravo, može se reći da akcija ovisi o sekvenci događaja (jednom ili više) koju agent primi putem senzora. Generalizirajući, da se zaključiti da svaki agent posjeduje tablicu sekvenca - akcija, te iz tablice za pripadnu sekvencu odradi pripadnu akciju. Jasno je da takva tablica može biti samo apstraktna zato što duljina sekvence mora biti ograničena, inače je tablica beskonačno velika. Takva tablica zove se eksterna karakterizacija agenta [7].

Funkcija agenta, unutar inteligentnog agenta bit će implementirana kao program agenta. Jako je važno uočiti da su spomenuta dva pojma različita. Funkcija agenta je apstraktna matematička funkcija, dok je program agenta implementacija funkcije agenta koja se izvršava na fizičkom sustavu koji je ograničen resursima [7].

Slika 4 prikazuje praktičnu uporabu spomenutih pojmova na primjeru. Kao primjer bit će korišten svijet usisivača. Riječ je o vrlo jednostavnom svijetu koji ima samo dvije lokacije: polje A i polje B [7].

Slika 4: Praktična ilustracija agenta

Funkcija agenta je takva da agent ovisno o čistoći polja odlučuje hoće li očistiti polje na kojem se nalazi, hoće se premjestiti na susjedno polje ili će čekati sljedeći podražaj. Funkcija je postavljena vrlo jednostavno na način da: ako je polje na kojem se agent nalazi prljavo, počisti ga, a inače se premjesti na susjedno polje. Ovisnost prikazuje tablica 2 [7].

Tablica 2: Eksterna Karakterizacija agenta

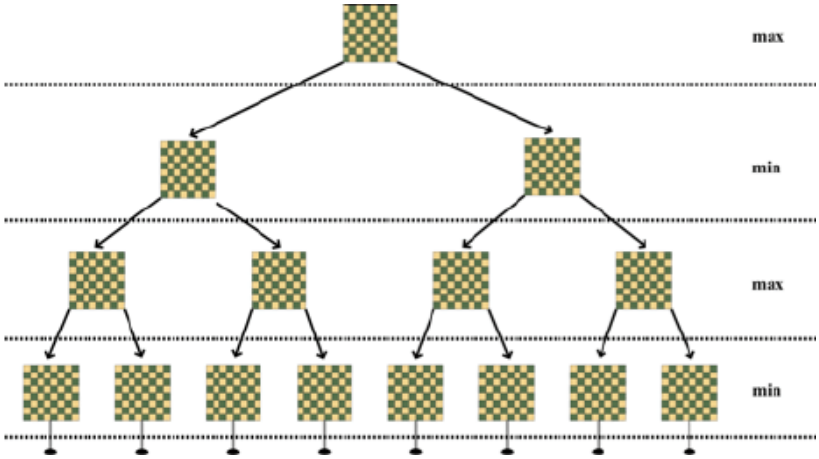
sekvenca	**akcija**
[A, čisto]	Desno
[A, prljavo]	Počisti
[B, čisto]	Lijevo
[B, prljavo]	Počisti
[A, čisto] [A, čisto]	Desno
[A, čisto] [A, prljavo]	Počisti
…	…
[A, čisto] [A, čisto] [A, čisto]	Desno
[A, čisto] [A, čisto] [A, prljavo]	Počisti
…	…

2.1.3 Minimax alogritam

Minimax algoritam je algoritam pretrage koji se koristi u igrama s dva igrača, tj. u više agentskom okruženju (engl.*multiagent environment*). Kod minimax algoritma, svaki agent mora razmotriti kako odluke protivničkog agenta utječu na tijek događaja. Koncept samog algoritma je vrlo jednostavan, svaki agent zasebno evaluira stanje $< -\infty, +\infty >$ te, što je evaluacija pozitivnija (MAX), to je prvi igrač u boljoj poziciji, dok s druge strane što je evaluacija negativnija (MIN), to je drugi igrač u boljoj poziciji. Važno je napomenuti da nepredvidljivost odluka protivničkog agenta značajno utječe na donošenje odluka.

Također, jako je bitna i dužina sekvence koju agent pregledava. Kako je ranije rečeno, nije moguće kreirati tablicu odluke baziranu na sekvenci koja ima beskonačnu duljinu već sekvenca mora biti limitirana. Duljina sekvence je najčešće povezana s fizičkim ograničenjima sustava na kojem se agent nalazi. Jedan način određivanja duljine sekvence je vremensko ograničenje agenta prilikom donošenja odluka (jedno od svojstava algoritma je konačnost !), dok je druga programsko ograničenje agenta prilikom pregledavanja sekvenci (najčešće se koristi kombinacija). Osim navedenih ograničenja, agent također završava pretragu ako prije kraja sekvence dosegne terminalno stanje. Primjer terminalnog stanja za igru šah bilo bi "*šah-mat*".

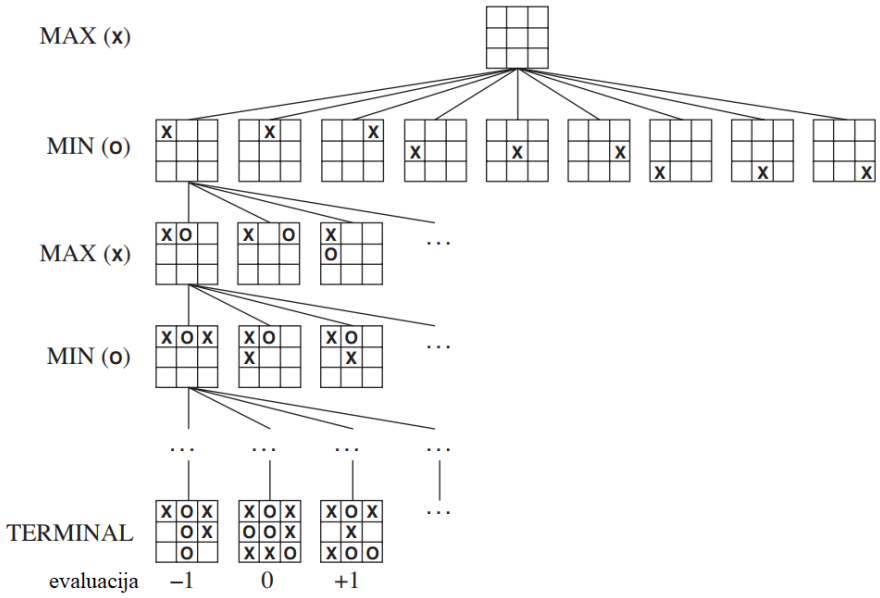
Agent odluku o prelasku u sljedeće stanje bira isključivo na temelju stanja koje se nalazi na kraju sekvence (osim ako ranije na ne naiđe na terminalno stanje). Kako se radi o igri u kojoj sudjeluju dva igrača jasno je da nakon poteza prvog slijedi potez drugog igrača. Uzeta je duljina sekvence 3 i prvi igrač je na potezu. Prvi igrač prvo izgenerira sve akcije čijim izvršavanjem može prijeći u druga stanja. Zatim, izgenerira sve akcije protivnika, te nakon toga ponovno izgenerira sve akcije tako da iz svakog stanja protivnika izgenerira sve moguće akcije. Kada se stvar raspiše, jasno je da se radi o jako velikom broju stanja čak i za jako jednostavnu igru. Kada se vizualno konstruira opisana podatkovna struktura jasno je da izgleda kao izokrenuto stablo. Takva struktura u programiranju zove se *n-arno* stablo pretrage. Primjer takvog stabla za igru šah prikazuje slika 5 [7].



Slika 5: Primjer stabla pretrage za igru šah

Formalno, cijela igra može biti opisana kao algoritam pretrage sa sljedećim stanjima:

1. S_0 – Početno stanje koje opisuje kako je igra postavljena.
2. $igrač(s)$ – Definira koji je igrač na potezu
3. $rezultat(s, a)$ – Tranzicijski model, definira rezultat akcije koju agent izvrši
4. $akcija(s)$ – Niz legalnih akcija pomoću kojih agent može prijeći u sljedeće stanje.
5. $test\ terminalnosti\ (s)$ – Provjerava je li igra gotova, odnosno je li agent dosegao terminalno stanje ili stanje koje predstavlja kraj sekvence.
6. $evaluacija(s)$ – Definira vrijednosti za igru koja je dosegla terminalno stanje.

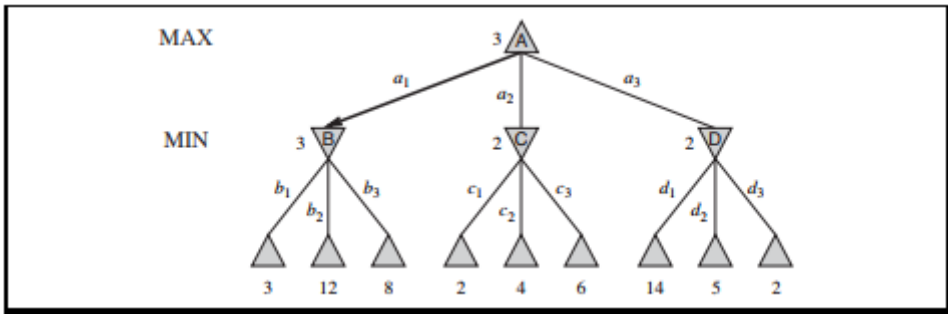


Slika 6:Primjer minimax algoritma

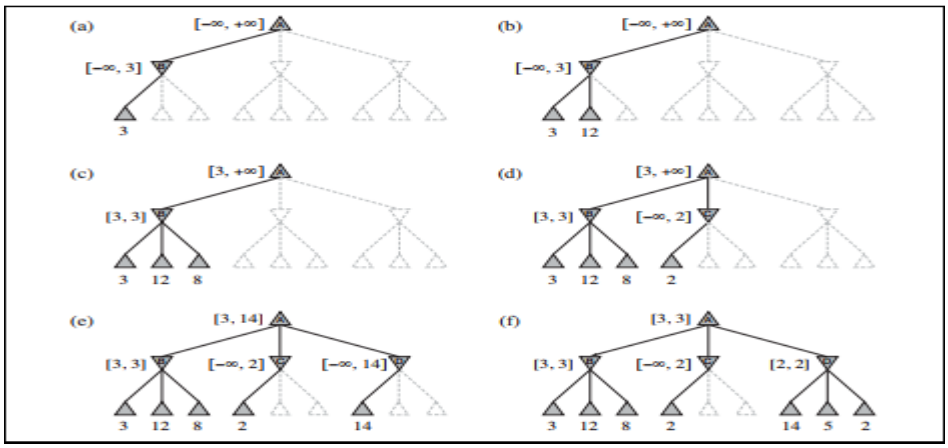
2.1.4 Alfa-beta podrezivanje

Problem minimax algoritma je eksponencijalna ovisnost broja stanja koje je potrebno ispitati i dubine stabla, tj. broj stanja raste eksponencijalno, kako se dubina povećava ($\sim (broj\ stanja)^{dubina}$). Nažalost, eksponencijalan rast ne može se eliminirati, ali može se umanjiti. Moguće je odrediti *najbolju* minimax odluku bez pregledavanja svih čvorova stabla, tako da se određeni dijelovi stabla *režu* i uklanjaju iz razmatranja.

Slika 7 prikazuje proces rada alfa-beta podrezivanja. Točnije, prikazuje izgenerirano stablo pretrage igre između dva igrača gdje Δ predstavlja MAX čvorove, dok ∇ predstavlja MIN čvorove unutar stabla. Terminalni čvorovi su evaluirani, te kraj sebe imaju vrijednosti za MAX (igrač 1 je na potezu) [7].

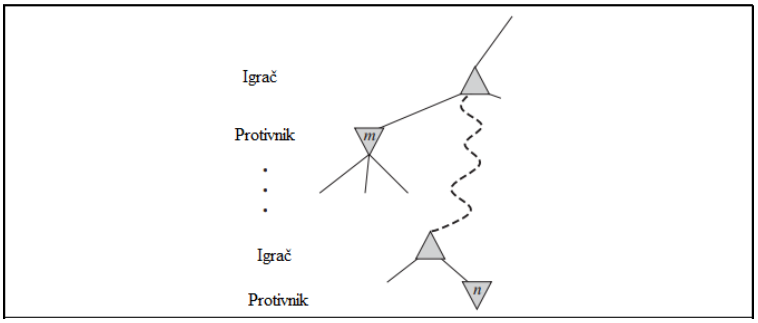


Slika 7: Grananje stabla pri minimax algoritmu



Slika 8: Grafički prikaz alfa beta podrezivanja pri minimax algoritmu

Slika 8 prikazuje korake odabira optimalne odluke za stablo pretrage. Prvi čvor ispod B ima vrijednost 3. Stoga, B, koji je MIN čvor ima vrijednost *najviše* 3. Drugi čvor ispod B ima vrijednost 12, te će stoga biti ignoriran od strane MIN funkcije, tako da je 3 i dalje *najveća vrijednost*. Treći čvor ispod B ima vrijednost 8 te će također biti ignoriran. Stoga, kako smo prošli sve "*dijete*" čvorove od B možemo reći da B ima vrijednost 3. Nadalje, možemo zaključiti da je vrijednost početnog stanja(engl.*root*) barem 3 zato što MAX ima izbor vrijedan 3, gledajući s vrha. Prvi čvor ispod C ima vrijednost 2. Stoga, C koji je MIN čvor ima vrijednost *najviše* 2, ali kako znamo da je B vrijedan 3, MAX nikad neće izabrati C. Stoga, nema potrebe gledati ostale čvorove ispod C. Prvi čvor ispod čvora D ima vrijednost 14, tako da je D vrijedan *najviše* 14. Sljedeći nasljednik ima vrijednost 5, tako da je ponovno potrebno daljnje istraživanje. Treći nasljednik je vrijedan 2 . Odluka za MAX bit će čvor B s pripadnom vrijednosti 3 [7].



Slika 9:Grafički prikaz praktične uporabe alfa-beta podrezivanja

Na slici 9 još je jasnije prikazana praktična uporaba alfa-beta podrezivanja. Naime, ako je čvor *m* bolji izbor od čvora *n*, čvor *n* u stvarnoj igri nikad neće biti dosegnut, te nije potrebno razmatrati njegove nasljednike, kao ni njega samog [7].

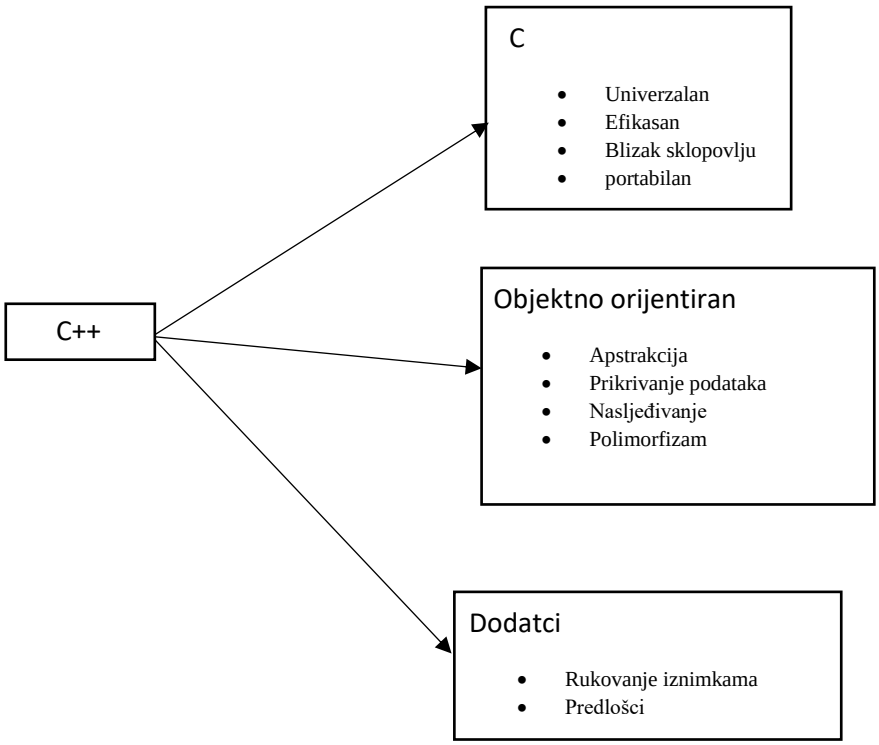
3. Praktični dio

3.1 Korištene tehnologije

3.1.1 Programski jezik C++

Programski jezik *C++* osmislio je i kreirao Bjarne Stroustrup, danski znanstvenik. Primarni razlog kreiranja *C++* jezika bio je poboljšanje tada široko korištenog programskog jezika *C*. Kao što mu samo ime kaže *C++* ili "*C s klasama*" nudi gotovo jednaku brzinu izvođenja kao *C*, ali uz puno veću razinu apstrakcije.

Pri kreiranju *C++* jezika, Bjarne Stroustrup bio je inspiriran programskim jezikom Simula, koji je tada bio korišten za kompleksne simulacije računalni sustava. Snaga *C++* jezika u odnosu na *C* bila je ta što je koristio sve aspekte jezika *C*, te u njih involvirao objektno orijentirani pristup kao i neke dodatke [8]. Struktura *C++* jezika prikazana je na slici 10.



Slika 10:Grafički prikaz strukture C++ jezika

3.1.2 Programsko okruženje Visual Studio

Visual studio, programsko je okruženje kreirano od strane kompanije Microsoft. Dolazi u nekoliko verzija dostupnih korisniku, vrlo lako se može preuzeti s Microsoft-ove web stranice, te je dostupan za Mac-os i Windows platformu. Visual studio dolazi u 3 različita izdanja: Community, Professional i Enterprise. Dok je za Professional i Enterprise verziju potrebno platiti licencu za korištenje, Community verzija je potpuno besplatna.

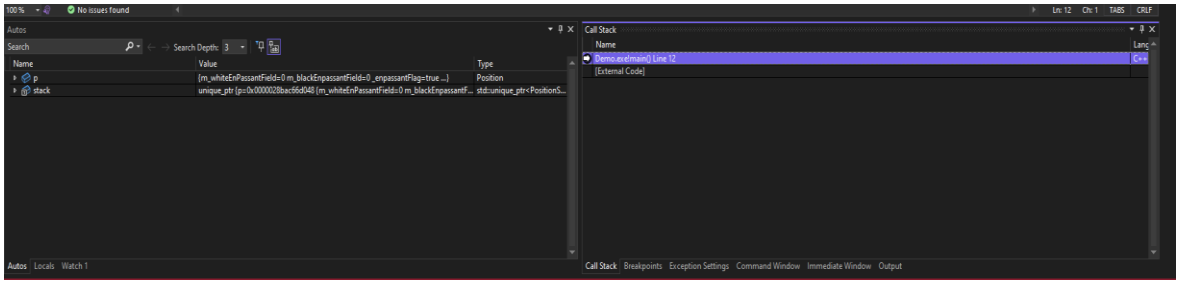
Nakon preuzimanja i instalacije programa, također je potrebno instalirati pakete koji će biti korišteni kao nadogradnje na sam Visual Studio. Naime, Visual Studio je programsko okruženje a ne uređivač teksta, kao na primjer Visual Studio Code.

Bitno je napomenuti da je razlika između uređivača teksta i programskog okruženja ta da se prilikom korištenja uređivača teksta (Visual Studio Code) korisnik sam mora pobrinuti oko instalacije prevoditelja (engl.*compiler*) i povezivanja istog s uređivačem dok se programsko okruženje samo pobrine za sve postavke i instalacije putem preuzimanja gore navedenih paketa.

Visual Studio, uz nadogradnje za programske jezike također omogućuje instalaciju brojnih proširenja kao što su proširenje za git itd… Potpuno je prilagođen korisniku tako da korisnik sam može mijenjati font slova, pozadinu, temu, izgled navigacijske trake itd…

Jedan od glavnih razloga uporabe Visual Studia bio je uporaba programa za ispravljanje nepravilnosti (engl.*debugger*). *Debugger* je podešen tako da su s lijeve strane vidljive vrijednosti varijabli, dok se s desne strane nalazi stanje stoga, pozivi funkcija itd…

Visual Studio također nudi proširivanje *debuggera* određenim alatima kao što su alat za profiliranje (engl.*profiling tool*) itd… Izgled sučelja *debuggera* prikazan je na slici 11.

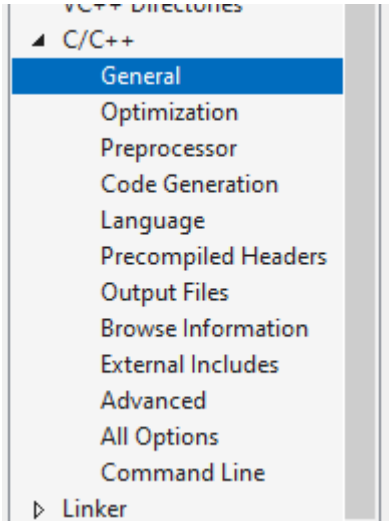


Slika 11: Izgled debuggera unutar Visual Studia

3.1.3 Prevoditelji

Kako je projektni zadatak pisan C++ jezikom, potreban je bio izbor prevoditelja (engl.*compiler*). Uzevši u obzir da je rad primarno bio kreiran u Visual Studio okruženju tako je bio korišten "MSVC" skup alata.

"MSVC" skup alata, integrirani je skup alata za prevođenje i otkrivanje nepravilnosti C++ koda, koji je integriran u Visual Studio. Prednost korištenja "MSVC" alata unutar Visual Studia je jednostavno pokretanje, kombinacijom tipki CTRL+ F5 za prevođenje ili samo tipkom F5 za otkrivanje nepravilnosti. Krasi ga vrlo dobro dizajnirano grafičko sučelje za konfiguriranje samog prevoditelja (slika 12).



Slika 12: Sučelje za konfiguraciju MVSC prevoditelja

U svrhu dodatne provjere i poboljšanje kvalitete koda uz “MSVC” korišten je i “GNU gcc/g++” prevoditelj. Gcc (engl.*GNU compiler collection*) skup je alata otvorenog koda koji omogućuju prevođenje koda, otkrivanje nepravilnosti, te detaljniju analizu samog koda. Primarna razlika između “MSVC” i Gcc je ta što je Gcc znatno *konzervativniji*, tj. puno strože pristupa određenim nepravilnostima unutar koda.

3.1.4 Winboard protokol

Kako je praktični dio rada zamišljen kao konzolna aplikacija bez upotrebe biblioteka za grafiku(prikaz šahovske ploče, figura i poteza), također bilo je potrebno kreirati grafičko sučelje ili iskoristiti postojeće sučelje otvorenog koda. Kreiranje novog grafičkog sučelja oduzelo bi previše vremena i zasigurno odmaklo fokus sa stvarne teme ovog rada, stoga je odlučeno koristiti postojeće sučelje otvorenog koda. Kao dva glavna izbora nametala su se grafička sučelja “Winboard” i “Arena” . U konačnici, zbog modernijeg izgleda, te više opcija za konfiguraciju izabrano je grafičko sučelje “Arena”.

Grafičko sučelje “Arena”, besplatno je sučelje za igru šah. Nudi mogućnost igranja partija između 2 igrača, igrač protiv računala ili pak sparivanje 2 računala na međusobni dvoboj, te kreiranje i praćenje turnira. Dostupna je za više platformi (Linux, Windows, MacOs), te nudi mogućnost povezivanja sa šahovskim satom, te praćenje šahovskih partija uživo (engl.*live stream*) što je izrazito korisno na šahovskim turnirima. “Arena” dolazi s nekoliko preinstaliranih motora(engl.*chess engine*) koji se mogu koristiti za igru ili za analizu. Neki od njih su: “AnMon”, “Hermann”, “Ruffian”, “Rybka”, “Stockfish”, itd…

Kako je “Arena” samo grafičko sučelje, potrebno je šahovski motor (engl.*chess engine*) *enkapsulirati* unutar samog sučelja. Prilikom pokretanja motora i grafičkog sučelja može se uočiti da su to dva odvojena procesa,te da svaki od njih ima zaseban adresni prostor (engl.*address space*), stoga za povezivanje motora i sučelja nužno je koristiti međuprocesnu komunikaciju.

Ideja je da sučelje prilikom svog pokretanja pokrene i motor, te da oni komuniciraju koristeći predefinirani jezik, tj.*protokol*. Ideja same komunikacije je vrlo dobra zato što kreatoru motora daje slobodu implementacije i prikaza šahovske ploče i poteza na proizvoljan način, s tim da je jedino bitno da prilikom *slanja* određenog poteza ili stanja prema sučelja poštuje protokol koji sučelje koristi.

Uzevši u obzir da je odabrano grafičko sučelje “Arena” u obzir su dolazila samo dva komunikacijska protokola: “Winboard/Xboard” i “UCI(Universal chess interface)”.
Zbog jednostavnosti u radu je korišten “Winboard/Xboard protokol”.

“Winboard” protokol dio je “GNU” projekta. Kako je ranije rečeno da se grafičko sučelje pokreće u odvojenom adresnom prostoru od samog motora potrebno je uspostaviti priključke (engl. *socket*) i putove (engl. *pipe*) između motora i sučelja. Kreator motora ne treba se brinuti za to, već se samo sučelje pobrine te uspostavi komunikaciju na način koji će biti opisan u nastavku.

Generalno, komunikacija između dva ili više procesa odvija se pomoću deskriptora datoteke (engl. *file descriptor*). Deskriptori datoteke jedinstveno identificiraju lokaciju na koju je potrebno poslati određenu informaciju ili mjesto s kojeg je potrebno pročitati određenu informaciju koristeći datoteke. Svakom kreiranom procesu automatski (operativni sustav) se dodjeljuju 3 deskriptora datoteke, a to su: standardni ulaz (STDIN), standardni izlaz (STDOUT) i standardna pogreška (STDERR). Također, bitno je napomenuti da je svaki deskriptor datoteke prikazan cijelim brojem koji se određenim algoritmima dobije iz imena ili pozicije datoteke koju pojedini deskriptor koristi. Spomenuti deskriptori za svaki proces

imaju predefinirane vrijednosti. Vrijednosti spomenutih deskriptora prikazuje tablica 3.

Tablica 3: Prikaz vrijednosti standardnih deskriptora datoteke

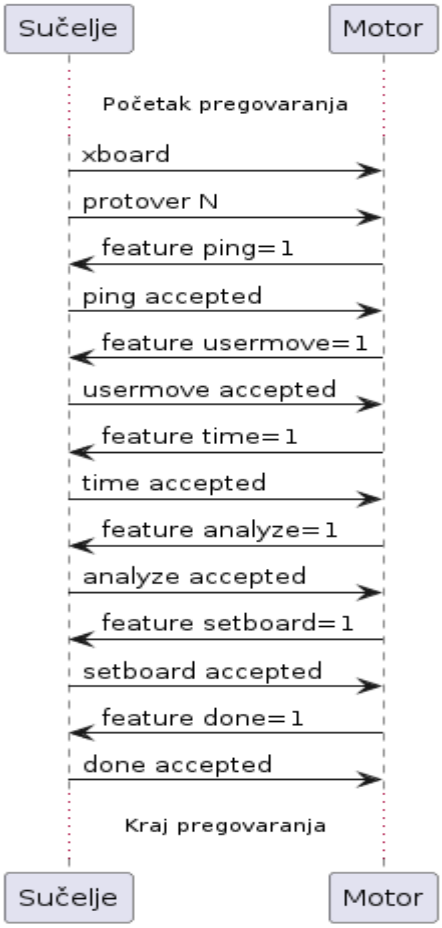
deskriptor	*vrijednost*
STDIN	0
STDOUT	1
STDERR	2

Winboard protokol u svom radu koristi gore navedene deskriptore za komunikaciju. STDIN za čitanje, STDOUT za pisanje i STDERR za javljanje poruke o pogrešci, bila ona nepravilan rad samog motora ili rušenja procesa (engl.*program crash*). Winboard zbog svoje implementacije ne može komunicirati s motorima koji su prevođeni koristeći "DOS" proširenja za 32-bitne adrese. Kako Winboard protokol dolazi u više verzija, te podržava različita proširenja i module od kojih su neki obvezni, prije same igre u pozadini se odvija procedura *pregovaranja* između motora i sučelja. Primjerice motor može, a i ne mora podržavati igru s kontrolom vremena, tj.ne mora pratiti vrijeme koje mu je potrebno za potez nego igra dok mu isto ne isteče. Winboard dolazi u dvi verzije (1, 2) stoga je motor prilikom pregovaranja dužan naglasiti koju verziju protokola koristi zato što su se kroz povijest same naredbe protokola mijenjale.

Radi pridržavanja samog protokola motor je dužan pri svom pokretanju prvi poslati proizvoljnu poruku na STDOUT deskriptor kako bi sučelje znalo da je motor aktivan i kako bi započelo proces *pregovaranja*. Proces pregovaranja počinje u trenutku kada sučelje motoru pošalje naredbu *xboard*.

Dijagram 2 prikazuje sekvencu *pregovaranja* između motora i sučelja. Bitno je napomenuti da je prikazani graf samo primjer jednog uspješnog tipa komunikacije. Komunikacija započinje naredbom "*xboard"*, nakon čega sučelje uvjetuje tip protokola koji koristi porukom " *Protover N* " gdje je $N \in [1, 2]$. Kao što je već rečeno tip protokola uvjetuje poruke koje će motor i sučelje razmjenjivati. Nakon definiranja tipa protokola motor izvještava sučelje o proširenjima koja podržava porukom "*feature proširenje = x*", gdje je $x \in [0, 1]$. Kako Winboard protokol podržava jako velik broj proširenja nije ih potrebno navesti sve već samo ona osnovna, tj.ona koja su prikazana na slici. Ukoliko je vrijednost parametra $x = 0$, to znači da motor ne podržava određeno proširenje, dok za $x = 1$ znači da podržava. Jako je bitno napomenuti da **motor ne mora podržavati sva osnovna proširenja**, ali mora se izjasniti o njihovom statusu inače pregovaranje neće biti uspješno. Proces pregovaranja završava proširenjem "*done*". Primjer proširenja prikazuje tablica 4.

Dijagram 2: Sekvenca pregovaranja između motora i sučelja



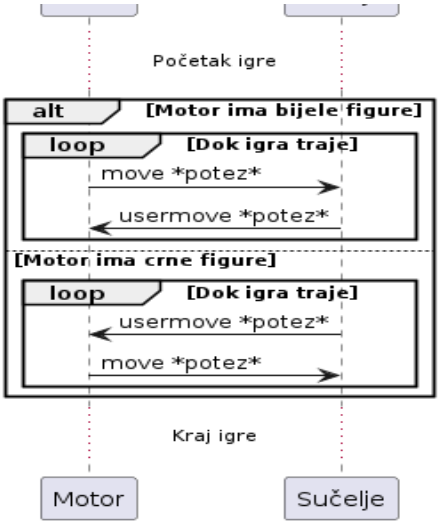
Tablica 4: Prikaz značenja osnovnih proširenja

Proširenje	***Značenje***
ping	Provjera aktivnosti motora
usermove	Potez koji odigra strana koja koristi sučelje, a šalje se motoru
time	Kontrola vremena
analyze	Prikaz stabla linija poteza od strane motora
setboard	Postavljanje stanja ploče iz FEN notacije
done	Kraj procesa pregovaranja

Naredba "*ping*" šalje se periodično od strane sučelja k motoru u svrhu provjere budnosti. Sučelje šalje "*ping N*", na što motor odgovara "*pong N*" , gdje je $N \in [\,1, +\infty >$. N u odgovoru mora biti isti N koji je primljen, inače komunikacija nije valjana.

Dijagram 3 prikazuje sekvencu kojom se igra odvija, te poruke koje se razmjenjuju između motora i sučelja. Svaku poruku o potezu koju motor primi reinterpretira u format koji sam motor razumije, odradi kalkulaciju te pošalje potez odgovora u predefiniranom formatu. Format u kojem se šalju potezi je standardni šahovski format (kratica imena figure + polazno polje + polje na koje se pomiče, primjer: Qd2d4 – Dama s polja d2 na polje d4). Također, sučelje može motoru poslati poruku "*go*" koja motoru nalaže da bez obzira koja strana je na potezu iskalkulira potez, te ga odigra.

Dijagram 3:Prikaz sekvence komunikacija motora i sučelja za vrijeme trajanja partije



3.1.5 Lichess poslužitelj

Kako je “Winboard/Xboard” protokol relativno star u usporedbi s “UCI protokolom” spomenutim ranije, tako su i sama sučelja koja podržavaju “Winboard” protokol zastarjela. Primarno, u svrhu testiranja motora, a potom i u svrhu poboljšanja korisničkog iskustva odlučio sam se za isporuku (engl. *deploy*) aplikacije na “lichess” poslužitelj. Naime, kako je riječ o šahu, gotovo je nemoguće testirati aplikaciju u potpunost,i stoga sam se odlučio za isporuku aplikacije na poslužitelj koji je aktivan 24 sata na dan kako bi moja aplikacija mogla biti testirana od strane drugih igrača. Sam postupak isporuke aplikacije je poprilično jednostavan, “lichess” pruža sučelje otvorenog koda u obliku “python” skripte, stoga je potrebno klonirati mapu koja se nalazi na službenoj lichess stranici te pratiti upute. Prije kloniranja same mape potrebno je napraviti “lichess” račun preko kojeg će motor igrati.

Nakon kloniranja mape, te instalacije virtualnog okruženja (venv) potrebno je prevedenu aplikaciju (“.exe” za Windows ili “binary” za Linux) postaviti u *engines* mapu. Finalni korak je postavljanje *config.yaml* datoteke u kojoj se postavljaju sve konfiguracije.

3.2 Implementacija

3.2.1 Struktura projekta

Kako je rad pisan C++ jezikom tako su korišteni principi objektno orijentiranog programiranja. Naime, kako je klasa prvenstveno logička cjelina struktura projekta sadrži tri klase.

1. **Position** – Prikazuje trenutno stanje šahovske pozicije. Uključuje stanje šahovske ploče, stanje rohada, en-passant polja, broj figura na ploči, informaciju u tome tko je na potezu. Kako se slijede principi objektno orijentiranog programiranja, tako navedena klasa ima sve metode koje definiraju njeno ponašanje, o kojima će biti govora kasnije. Strukturu atributa klase prikazuje ispis 2.

```
class Position {
private:
        enum {es, wP, wR, wB, wN, wQ, wK, bP, bR, bB, bN, bQ, bK};


        int m_whiteEnPassantField;
        int m_blackEnpassantField;
        bool m_turn;
        std::array <int, 144> m_board;
        bool m_whiteSmallCastle;
        bool m_whiteBigCastle;
        bool m_blackSmallCastle;
        bool m_blackBigCastle;
```

Ispis 2: Prikaz klase pozicije

2. **Move** – Prikazuje potez, te sve njegove atribute. Kako se šah odlikuje *posebnim* potezima (en-passant, rohada, promocija) tako je i klasu potez potrebno proširiti određenim zastavicama i vrijednostima radi lakšeg rukovanja potezima, o čemu će biti govora u nastavku. Strukturu atributa klase *Move* prikazuje ispis 3.

```
class Move {
public:
        int piece = 0;
        int toPosition = 0;
        int pieceToEat = 0;
        int fromPosition = 0;
        bool isEnpassant = false;
        bool isPromotion = false;
        int promotionPiece = 0;
```

Ispis 3: Prikaz klase potez

3. **PositionStack** – Predstavlja stog na koji se spremaju pozicije, služi za čuvanje *povijesti* odigranih poteza. Strukturu atributa klase prikazuje ispis 4

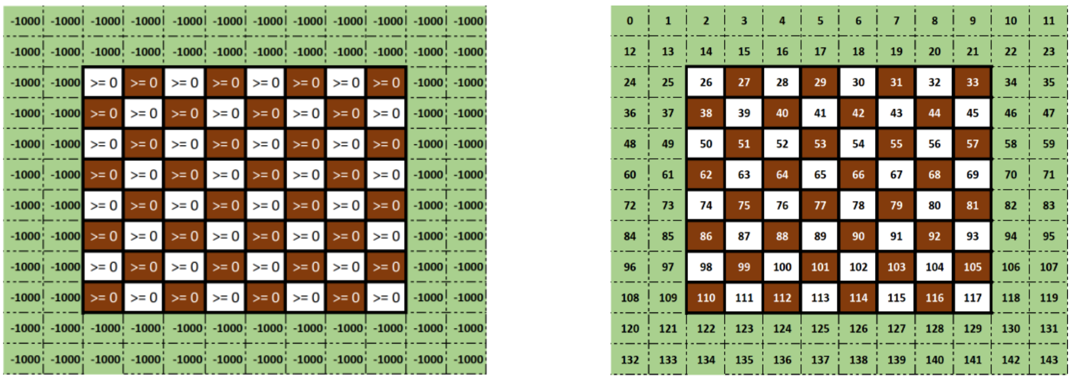
```
class PositionStack {
private:
        Position* m_history;
        int m_cursor;
public:
        PositionStack();
        void push(const Position& position);
        Position pop();
        ~PositionStack();
};
```

Ispis 4: Prikaz pozicijskog stoga

3.2.2 Prikaz šahovske ploče i figura

Šahovska ploča prikazana je nizom cjelobrojnih vrijednosti. Umjesto korištenja primitivnih nizova korištenih u C jeziku korišten je *standardni array*. *Standardni array* dio je standardne C++ biblioteke (STL), a odlikuje ga gotovo jednaka brzina pristupanja elementima kao kod primitivnog niza uz dodatne pogodnosti. Naime, *standardni array* alocira memoriju na stogu, jednako kao i primitivni niz, ali za razliku od primitivnog niza sadržava informaciju o svojoj veličini unutar privatne varijable *size*. *Standardni array* nudi mogućnost pristupanja elementu korištenjem [ ] operatora uz proširenje pristupanja elementu pomoću metode *at(indeks),* koja će u slučaju pokušaja pristupanja elementu koji je izvan granica zauzete memorije baciti iznimku, umjesto rušenja programa (engl.*program crash),* što bi se dogodilo u istom scenariju kod primitivnog niza. Metoda *at(indeks)* je samo dio javnog sučelja koju *standardni array* nudi.

U samom prikazu šahovske ploče korišten je niz duljine 144, što odgovara šahovskoj ploči dimenzija 12x12. Razlog korištenja ploče 12x12 je lakše generiranje poteza. Naime, rubovi *matrice* imaju visoko negativne vrijednosti, dok je samo područje šahovske ploče vrijednosti >= 0. Slika 13 prikazuje logički prikaz šahovske ploče. Naime, lijevi dio predstavlja polja s pripadnim vrijednostima dok su na desnom dijelu prikazani indeksi pojedinog polja.



Slika 13: Prikaz šahovske ploče

Svaka šahovska figura predstavljena je jedinstvenim identifikatorom u obliku pozitivne cjelobrojne vrijednosti uz pomoć privatnog enumeratora (ispis 5). Naime, u C++ jeziku enumerator je tip podatka koji služi za prikaz cjelobrojnih (engl. *integer*) vrijednosti, najčešće tako da se samo za prvu varijablu definira vrijednost, dok je vrijednost svake iduće varijable enumeratora jednaka vrijednosti prethodne uvećano za 1 ($var_n = var_{n-1} + 1$). Razlog ovakvog prikaza figura je taj što se figure moraju razlikovati (primjer: bijeli lovac i crni lovac), odnosno mora biti jasno na kojem se polju unutar šahovske ploče nalazi pojedina figura.

Tablica 5: Prikaz vrijednosti figura na šahovskoj ploči

Kratica figure	*figura*	*id*
es	prazno polje	0
wP	bijeli pješak	1
wR	bijeli top	2
wB	bijeli lovac	3
wN	bijeli skakač	4
wQ	bijela dama	5
wK	bijeli kralj	6
bP	crni pješak	7
bR	crni top	8
bN	crni skakač	9
bB	crni lovac	10
bQ	crna dama	11
bK	crni kralj	12

```
class Position {
private:
	enum {es, wP, wR, wB, wN, wQ, wK, bP, bR, bB, bN, bQ, bK};
```

Ispis 5: Implementacija figura

Iz tablice 5 može se primjetiti da je u figure svrstano i prazno polje. Razlog svrstavanja praznog polja među figure je iz konzistentonsti. Naime, ideja ovog enumeratora je prikaz stanja pojedinog polja unutar šahovske ploče, bila na njemu figura ili ne.

3.2.3 Implementacija minimax algoritma

"Minimax" algoritam predstavlja centralni dio praktičnog dijela rada. Kako je ranije rečeno, agentova odluka o prelasku u iduće stanje donosi se na osnovu algoritma koji je u ovom slučaju, rekurzivni, minimax algoritam. Također, iz minimax algoritma pozivaju se sve ostale metode koje su potrebne da bi se prešlo u sljedeće stanje, o kojima će biti govora u narednim poglavljima.

"Minimax" metoda se poziva u trenutku pretrage za najboljim potezom odgovora. Naime, metoda "search" generira sve dostupne poteze metodom za generiranje poteza (o kojoj će biti govora kasnije) te za svaki potez poziva "Minimax" metodu koja generira spomenuto stablo pretrage (slika 5). Ispis 6 prikazuje metodu "search".

```
Move moves[255];
generateMoves(moveCounter, moves);
for (int i = 0; i < moveCounter; i++) {
        this->makeMove(moves[i]);
        float evaluation = minimax(4,alpha,beta,line);
        this->undo_move();
        if (this->m_turn == white) {
                if (evaluation > alpha) {
                        alpha = evaluation;
                        moveIndex = i;
                }
        }
        else{
                if (evaluation < beta) {
                        beta = evaluation;
                        moveIndex = i;
                }
        }
```

Ispis 6: Metoda "search"

Kako je “minimax” rekurzivni algoritam, prvo je potrebno provjeriti je li stanje koje se trenutno obrađuje terminalno stanje. Broj terminalnih stanja ovisi o igri za koju se minimax implementira, a kako je riječ o šahu potrebno je provjeriti nekoliko terminalnih stanja.

```
if (move_counter == 0) {
        if (this->m_turn == white) {
                int kingIndex = findPiece(wK,*this);
                if (this->isAttacked(kingIndex)) {
                        return float(- 100000 - (depth - MAX_DEPTH));
                }
                else {
                        return 0;
                }
        }
        else {
                int kingIndex = findPiece(bK, *this);
                if (this->isAttacked(kingIndex)) {
                        return float(100000 + depth);
                }
                else {
                        return 0;
                }
        }
}
if (depth == 0) {
        return eval();
}
```

Ispis 7: Implementacija provjere terminalnih stanja

Ispis 7 prikazuje provjeru terminalnih stanja za igru šah. Kao što se može uočiti funkcionalnost se sastoji od 2 glavne provjere, uz 2 dodatne pod provjere.

1. Prvi slučaj terminalnog stanja događa se ako strana koja je na potezu nema mogućnost odigravanja niti jednog legalnog poteza. Takav scenarij može se objasniti samo na 2 načina: “šah-mat” ili “pat”. Šah mat je stanje u kojem je kralj napadnut od strane protivničke figure, te nema niti jedno polje za bijeg, niti se može zakloniti iza druge figure iste boje (predstavlja pobjedu strane koja je *dala* “šah-mat”, dok je “pat” sličan kao “šah-mat”, ali za razlika je u tome da polje na kojem se kralj nalazi nije napadnuto od strane protivničke figure (predstavlja neriješen ishod). Spomenuti scenariji provjeravaju se u pod provjerama.

2. Drugi slučaj provjere terminalnog stanje provjerava je li pretrage dosegnula kraj sekvence, te ako je, metodi *"search"* vraća se ocjena stanja na kraju sekvence.

Nakon provjere terminalnih stanja slijedi izvršavanje "minimax" algoritma sve dok se jedno od gore spomenutih stanja ne dosegne. Postupak izvođenja algoritma za bijele figure prikazuje ispis 8. Postupak je ekvivalentan i za crne figure.

```
if (this->m_turn==white) {
        int i;
        for (i = 0; i < move_counter; i++) {
                this->makeMove(moves[i]);
                float returnValue = minimax(depth - 1,alpha,beta,bestLine);
                this->undo_move();
                if (returnValue > alpha) {
                        alpha = returnValue;
                }
                if (alpha >= beta) {
                        return alpha;
                }
        }
        return alpha;
}
```

Ispis 8: Izvođenje minimax algoritma

3.2.4 Generiranje poteza

Generiranje poteza akcija je koja se događa u trenutku kada određena strana mora odigrati potez. Naime, da bi se pronašao najbolji potez potrebno je generirati sve moguće poteze koji se mogu odigrati iz trenutnog stanja. Prilikom generiranja svakog poteza kreira se objekt klase Move (ispis 4), te se atributi spomenutog niza popunjavaju u nekoliko koraka:

1. **Generiranje kretanja**

   Kako je ranije objašnjeno, svaka šahovska figura prikazana je jedinstvenim identifikatorom, dok je šahovska ploča dimenzija 12 x 12. Svaki par figura (pr. crni i bijeli lovac) posjeduje zasebne *delta* vrijednosti koja predstavlja vrijednost pomaka koj treba biti dodana na indeks polja na kojem se figura trenutno nalazi kako bi se ista pomakla na naredno polje (sukladno pravilima šaha). Ispis 9 prikazuje programsku implementaciju spomenute *delte* kretanja na primjeru para lovaca (crni i bijeli lovac).

```
static constexpr int diagonalDelta[4] = { 13,-13,11,-11 };
```

Ispis 9: Implementacija delte kretanja za lovački par

Kako je lovac figura koja se kreće samo dijagonalno, navedeni niz predstavlja potreban odmak (engl. *offset*) da se lovac pomakne na sljedeće dijagonalno polje (u sva 4 smjera). Konkretno, uzevši u obzir prezentaciju ploče prikazanu na slici 13, ako uzmemo primjer položaja lovca na polju br.89 to bi značilo da se lovac može pomaknuti na polja 102, 76, 100 i 78 što odgovara dijagonalama. Spomenuto uvećavanje indeksa se ponavlja dokle god je vrijednost polja na koje figura ide >= 0. Objašnjeni prikaz kretanja aplikabilan je za ostale sve figure unutar šahovske ploče.

2. Provjera legalnosti poteza

Nakon što su destinacijsko i odredišno polje objekta klase Move popunjeni, prije popunjavanja ostalih atributa potrebno je provjeriti smije li se (prema pravilima igre) odabrana figura pomaknuti s destinacijskog na odredišno polje. Prema pravilima šaha, potez nije legalan ako je kralj strane koja je na poteza napadnut od strane protivničke figure (suprotne boje) nakon odigranog poteza (ako je *pod šahom*).

```
if (isLegal(m)) {
        possible_moves[move_counter].setCurrent_Position(m.fromPosition);
        possible_moves[move_counter].setPosition(m.toPosition);
        possible_moves[move_counter].setPiece(m.piece);
        move_counter++;

}
```

Ispis 10: Provjera legalnosti poteza

Iz ispisa 10 vidljivo je da se potez dodaje u niz generiranih poteza tek nakon što se utvrdi da je legalan. Sama provjera legalnosti prikazana je u ispisu 11, te je kritičan dio ovog praktičnog zato što iziskuje preciznost i optimalnost u isto vrijeme. Naime, promjena položaja svake pojedine figure može ugroziti položaj kralja na više načina (otvoreni napadi, dvostruki šahovi, šah nakon rohade), stoga je potrebno nakon odigravanja svakog poteza krenuti od kralja u svim smjerovima da bi se ustvrdilo je li kralj strane koja je na potezu napadnut od strane protivničke figure.

```
if (position.diagonalAttack(searchedKingIndex)
                || position.horizontalAndVerticalAttack(searchedKingIndex)
                || position.KnightAttack(searchedKingIndex)
                || position.pawnAttack(searchedKingIndex)) {
                        return false;
        }
return true;
```

Ispis 11: Prikaz provjere legalnosti poteza

3. Prikaz posebnih atributa

Kako je šah igra koja u svojim pravilima definira nekoliko *posebnih poteza*, tako je u samu mehaniku igre bilo potrebno implementirati način identificiranja i odigravanja istih. Generalno, metoda za odigravanje poteza trebala bi biti poprilično jednostavna "s polja X na polje Y", međutim posebni potezi kompliciraju stvari, te su njihovi indikatori prikazani raznim zastavicama, kako bi funkcija odigravanja znala prepoznati poseban potez, te primijeniti posebno pravilo. Pod posebne poteze klasificiraju se: "rohada", "en-passant, te "promocija".

a. Rohada

"Rohada" je, u šahu, poseban potez u kojem se vrši zamjena položaja kralja i topa iste boje pod nekoliko uvjeta: Kralj i top moraju se nalaziti na početnim pozicijama (bez prethodnog pomicanja), kralj ne smije biti po šahom, sva polja između kralja i topa moraju biti prazna, te niti jedno prazno polje na kraljevoj putanji ne smije biti napadnuto od strane protivničke figure. Kako je iz prethodnog objašnjenja jasno da se prilikom generiranja poteza za kralja svaki put treba provjeriti može li kralj rohirati. Jasno je da provjera svih gore navedenih uvjeta uzima mnogo resursa i vremena (svaki put treba provjeriti je li početni položaj kralja ili topa mijenjan tokom partije). Stoga, kako bi doskočio problemu odlučio sam unutar same pozicije dodati zastavice koji služe kao indikator ima li određena strana pravo na rohadu, tako da je prilikom provjere rohade dovoljno provjeriti stanja zastavice, te provjeriti jesu li kritična polja na udaru protivničke figure. Zastavica je zadana na *true* prilikom instanciranja objekta klase *Position*, te se mijenja na *false* u trenutku kada se kralj ili top pomaknu s početnog polja.

b. **En-passant**

"En-passant" ili *uzimanje u prolazu*, specijalni je potez pješakom u šahu. Naime, kada se protivnički pješak određene boje pomakne za dva polja naprijed s početnog polja i stane paralelno s pješakom suprotne boje pravila dopuštaju da se pješak pojede tako da se pješaka pomakne dijagonalno naprijed (ako je polje prazno). Navedeno pravilo vrijedi isključivo prvi potez nakon pomicanja pješaka. Implementacija ovog pravila također iziskuje uporabu zastavice, ali ovaj put u obliku cjelobrojne vrijednosti (engl.*integer*). Naime, za svaku stranu postoji zastavica *en-passant-polje* čija je zadana vrijednost 0, te se postavi na pozitivnu cjelobrojnu vrijednost ako su svi gore navedeni uvjeti zadovoljeni. Zastavica se prilikom svakog odigravanja poteza suprotne resetira.

c. **Promocija**

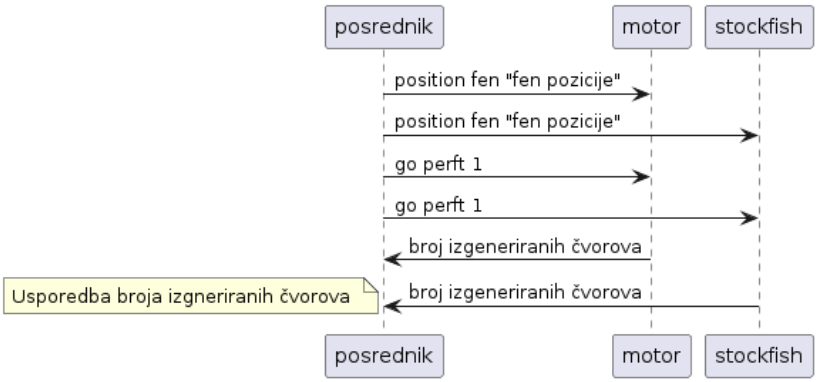
Pravilo "promocije" u šahu strani koja dođe do suprotnog ruba ploče s pješakom omogućuju zamjenu tog pješaka bilo kojom figurom iste boje. Strogo gledano, pomicanje pješaka na rubno polje u slučaju promocije klasificira se kao 4 zasebna poteza zato što se pješak može promovirati u skakača, lovca, damu ili topa. Implementacija ove mehaniku iziskuje dvije zastavice, jednu za indikaciju promocije (engl.*boolean*), dok je druga cjelobrojna (engl.*integer*) koja specificira promoviranu figuru. Također, uz zastavice za svaku stranu postoji programska implementacija niza figura koje se mogu promovirati.

3.2.4 Otkrivanje nepravilnosti prilikom generiranja poteza

Kako je generiranje poteza najsloženiji dio ovog rada, tako je i samo testiranje funkcionalnosti otkrivanja poteza poprilično složeno. Naime, kako se potezi generiraju unutar samog "minimax" algoritma, koji je rekurzivan, tako je i otkrivanje nepravilnosti teško zbog višestrukih rekurzivnih poziva na programskom stogu.

Nepravilnosti su otkrivane automatiziranom tehnikom komparacije. Naime, u svrhu samog praktičnog rada kreiran je poseban program posrednik (engl.*proxy software*) koji za isti ulaz uspoređuje izlazni rezultat testiranog motora s izlazom verificiranog motora (konkretno u ovom slučaju korišten je *stockfish*). Dovoljno je bilo izgenerirati velik broj pozicija u FEN (*Forsyth-Edwards notation)* formatu, te svaku od njih poslati put oba motora te ako se broj generiranih čvorova oba motora (na specificiranoj dubini) ne poklapa znači je da nepravilnost pronađena. Princip rada prikazuje dijagram 4.

Dijagram 4: Sekvenca rada programa za testiranje



Za izradu programa posrednika korišten je programski jezik Python, te modul Subprocess koji je bilo potrebno instalirati kao vanjsku biblioteku. Program radi na principu da oba motora pokrene kao pozadinski (engl.*deamon*) proces, te na njihove standardne ulaze šalje pozicije u FEN formatu i naredbu za izračunom broja čvorova, te s njihovih standardnih izlaza pročita rezultate te ih uspoređuje.

"FEN" format standardni je format prikaza šahovske pozicije, stanja "rohada", "en-passant" poteza i drugih bitnih parametara u tekstualnom obliku. Bitno je napomenuti da kreirani motor podržava uvoz pozicije iz "FEN" formata, te izvoz pozicije u "FEN" format.

3.2.5 Funkcija evaluacije

Kako je minimax generički algoritam, koji nije adaptiran samo za šah, već za većinu igara između dva igrača, stoga je bilo potrebno implementirati posebnu funkciju evaluacija na osnovu koje minimax donosi odluku o potezu. Kako je minimax korišten u igrama između dva igrača, konkretno za šah vrijedi sljedeće pravilo: pozitivne vrijednosti funkcije evaluacije indiciraju prednost igrača s bijelim figurama, dok negativne vrijednosti indiciraju prednost igrača s crnim figurama.

```
if (depth == 0) {
        return eval();
}
```

Ispis 12: Prikaz korištenja funkcije evaluacije

Kako je i ranije opisano, iz ispisa 12 vidljivo je da se evaluacija odvija tek na kraju sekvence, osim u slučajevima kada se ranije nađe terminalno stanje. Kreirani motor (engl.*engine*) koristi naprednu evaluaciju. Napredna evaluacija osim same vrijednosti figure u obzir uzima i položaj figure na ploči prema principima igre. Ispisi 13 i 14 prikazuju način evaluiranja pozicije.

```
int square = 0;
score += ConstValues::pieceValues[m_board[i]];
score += this->getScoreByPiecePosition(square, i);
```

Ispis 13: Funkcija evaluacije

```
float Position::getScoreByPiecePosition(int square,int boardIndex) {
        return m_board[boardIndex]>=MIN_WHITE_INDEX
            && m_board[boardIndex] <= MAX_WHITE_INDEX ?
               ConstValues::pieceOnSquareValues.
                   at(m_board[boardIndex] - 1).at(square)
               : float(-1) * ConstValues::pieceOnSquareValues.
               at(m_board[boardIndex] - 7).at(square);
}
```

Ispis 14: Dodatak vrijednosti na osnovu položaja figure

Vrijednosti dodatka na osnovu položaja figure dobiveni su na osnovu retrogardne analize i empirijskih formula, zato što je za šah još uvijek nisu strogo definirani principi zbog kompleksnosti same igre (primjer: u otvaranju principijelno je guranja pješaka k centru, dok u završnicama vrlo često rubni pješak dobiva ili primjer: u otvaranju principijelno je kralja "sakriti" u rohadu, dok je jedan od glavnih principa završnica kralj u centru ploče)

U svrhu spremanja vrijednosti figura, te dodataka kreirana je pomoćna klasa, međutim nikad se ne instancira, u svrhu uštede resursa sve vrijednosti unutar iste su **static constexpr.** Kako je konstantnih vrijednosti u ovom praktičnom radu puno korišten je **constexpr** identifikator kako bi se memorija sve konstante kreirale unutar vremena prevođenja (engl.*compile time*) umjesto za vrijeme trajanja programa (engl.*run time*).Primjer dodatka na evaluaciju za pješaka prikazuje ispis 15.

```
static constexpr
std::array<std::array<float, 144>, 7> pieceOnSquareValues = { {
{
        nn, nn, nn, nn, nn, nn, nn, nn, nn, nn, nn, nn,
        nn, nn, nn, nn, nn, nn, nn, nn, nn, nn, nn, nn,
        nn, nn, 0,  0,  0,  0,  0,  0,  0,  0, nn, nn,
        nn, nn, 50, 50, 50, 50, 50, 50, 50, 50, nn, nn,
        nn, nn, 10, 10, 20, 30, 30, 20, 10, 10, nn, nn,
        nn, nn, 5,  5, 10, 25, 25, 10,  5,  5, nn, nn,
        nn, nn, 0,  0,  0, 20, 20,  0,  0,  0, nn, nn,
        nn, nn, 5, -5,-10,  0,  0,-10, -5,  5, nn, nn,
        nn, nn, 5, 10, 10,-20,-20, 10, 10,  5, nn, nn,
        nn, nn, 0,  0,  0,  0,  0,  0,  0,  0, nn, nn,
        nn, nn, nn, nn, nn, nn, nn, nn, nn, nn, nn, nn,
        nn, nn, nn, nn, nn, nn, nn, nn, nn, nn, nn, nn
        },
```

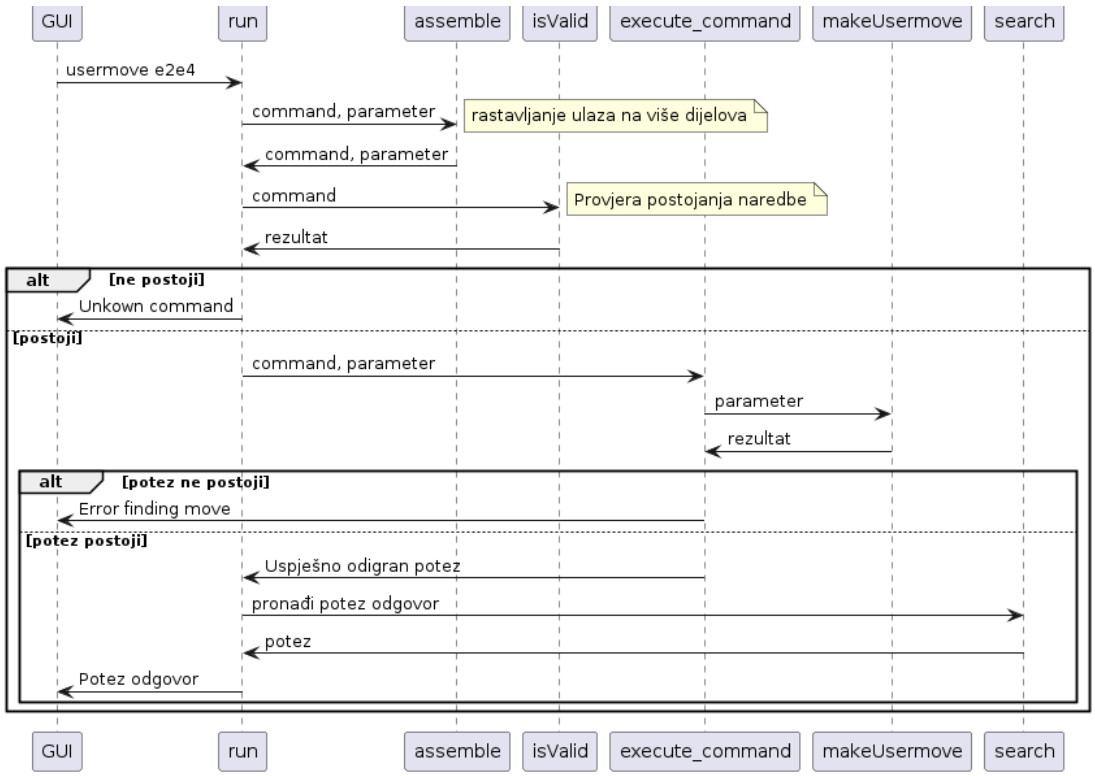
Ispis 15: Dodatak evaluaciji za pješaka

3.2.6 Implementacija sučelja za Winboard protokol

Kako je ranije rečeno, grafičko sučelje i motor su dva zasebna procesa koja komuniciraju preko predefiniranih deskriptora datoteke (engl.*file descriptor*). Kako su predefinirani deskriptori datoteke Winboard protokola zapravo isti oni koji se koriste prilikom ulaza s tipkovnice, bilo je dovoljno kreirati terminalno sučelje (engl.*TUI- Terminal user interface*). Sučelje se koristi za komunikaciju između grafičkog sučelja i samog motora, te za ručno postavljanje određenih postavki motora, otkrivanje nepravilnosti i testiranje.

Princip rada temelji se na čekanju događaja (engl.*event*) koji dođe iz vani. Događaj može biti ručni unos komande preko tipkovnice od strane korisnika ili pak poruka od strane grafičkog sučelja. Kada se motor pokrene, nakon ispisa inicijalne poruke potrebne za "Winboard" protokol, pokrene se beskonačna petlja (engl.*infinite loop*) koja čeka poruku iz van. Nakon što vanjska poruka dođe na STDIN deskriptor motor je detektira, validira, reinterpretira u oblik reprezentacije stanja koje motor podržava, generira odgovor, te na kraju generirani odgovor ponovno reinterpretira u standardni oblik. Primjer rada prikazuje dijagram 5.

Dijagram 5: Sekvenca rada terminalnog sučelja



4. Zaključak

U ovom radu dan je pregled osmišljavanja i izrade šahovskog motora (engl.*chess engine*) uz prikladnu teorijsku podlogu i objašnjenja. Prikazani šahovski motor ima mogućnost generiranja, obrade i evaluiranja ~150 000 čvorova po sekundi (engl.*nps- nodes per second*), kao i mogućnost uvoza i izvoza stanja u FEN (*Forsyth-Edwards notation*) formatu.

Sama izrada ovog završnog rada bitna je prvenstveno radi razumijevanja praktične uporabe algoritama. Praktični dio rada u potpunosti zaokružuje cijeli proces pristupa problemu, idejnog rješavanja problema, kreiranja algoritma na temelju idejnog rješenja te implementaciju samog rješenja korištenjem zadane tehnologije. Osobno, izrada ovog rada promijenila mi je pogled na algoritme pretrage, te sam naučio kako, generalno, takav tip algoritma primijeniti na neki realni životni problem. Nadalje, kako bi se priča upotpunila izrada motora iziskivala je duboko poznavanje strukture programske memorije, te načina interpretacije programskih instrukcija unutar samog procesora.

Bitno je spomenuti da u praktičnom dijelu rada nije riješen problem horizonta, te se sam način poništavanja poteza može poboljšati tako da se u samu mehaniku igre uvede prikaz pozicije pomoću "Zorbistovih ključeva".

Kako je ranije rečeno, kreirani motor objavljen je na "lichess" poslužitelju gdje svakodnevno, aktivno igra šah. Iskustva korisnika su pozitivna u smislu stabilnosti same aplikacije i brzine odigravanja poteza koja ne prelazi 7 sekundi. Do sad, motor od ukupnog broja odigranih partija ima 70% pobjeda.

5.Literatura

[1] IBM, “IBM službena stranica”, https://www.ibm.com/us-en, (posjećeno 04.05.2023).

[2] H. enciklopedija, Matematički pojam rekurzije, www.*enciklopedija.hr,* (posjećeno 04.05.2023).

[3] Stanford, Stanford Encyclopedia of Philosophy

[4] E. Omondi, Theory of computation

[5] Stanford, “Turing Machines”.

[6] f. c. camp, Big O notation.

[7] P. N. Stuart Russel, Artificial intelligence, a modern approach.

[8] B. Stroustrup, A history of C++, 1991.