Rješavanje problema skakača

Herceg, Ivana

SVEUČILIŠTE U SPLITU

PRIRODOSLOVNO MATEMATIČKI FAKULTET

ZAVRŠNI RAD

RJEŠAVANJE PROBLEMA SKAKAČA

Ivana Herceg

Split, rujan 2022.

Temeljna dokumentacijska kartica

Završni rad

Sveučilište u Splitu

Prirodoslovno-matematički fakultet

Odjel za informatiku

Ruđera Boškovića 33, 21000 Split, Hrvatska

RJEŠAVANJE PROBLEMA SKAKAČA

Ivana Herceg

SAŽETAK

Poopćenje problema skakača je problem pronalaska minimalnog broja poteza za bilo koju početnu i ciljnu šahovsku ploču. Kretanje skakača na šahovskoj ploči može se prikazati pomoću grafa. Pozicije na šahovskoj ploči smo prikazali kao vrhove, a povezanost između vrhova smo prikazali bridovima. Za rješavanje problema koristili smo najjednostavniji algoritam pretrage, algoritam iscrpnog pretraživanja. Algoritam je implementiran u programskom jeziku Python.

Ključne riječi: Problem skakača, algoritam iscrpnog pretraživanja, metoda razotkrivanja grafa.

Rad sadrži: 30 stranica, 12 grafičkih prikaza, 1 tablicu i 6 literaturnih navoda. Izvornik je na hrvatskom jeziku

Mentor: **izv. prof. dr. sc. Branko Žitko**

Ocjenjivači: **izv. prof. dr. sc. Branko Žitko**

izv. prof. dr. sc. Ani Grubišić

Lucija Bročić, *asistentica*

Rad prihvaćen: rujan, 2022. godine

Basic documention card

Thesis

University of Split
Faculty of Science
Department of Informatics
Ruđera Boškovića 33, 21000 Split, Croatia

SIX KNIGHTS

Ivana Herceg

ABSTRACT

A generalization of six knights' problem is finding the minimal amount of moves for any starting and final chess position. A knight's movement can be modeled with a graph. Each vertex in the modelling graph represents a chess position, white the edges represent possible knight's moves that mutate the game state. In this paper, the problem was approached using a naive search algorithm which was implemented in Python.

Keywords: Six Knights, brute force algorithm, method of unraveling graph

Thesis consists of : 30 pages, 12 figures, 1 table and 6 references. Original language: Croatian

Mentor: **Branko Žitko, Ph.D.** *Associate Professor*

Reviewers: **Branko Žitko, Ph.D.** *Associate Professor*

Ani Grubišić, Ph.D. *Associate Professor*

Lucija Bročić, *Assistant*

Thesis accepted: September, 2022

SADRŽAJ

1. Uvod

Problem skakača (engl. *Six knights*) je logička zagonetka u kojoj je cilj zamijeniti skakače u minimalan broj poteza, tako da ni u jednom trenutku ne budu dva skakača na istoj poziciji. Ova slagalica je dobro poznata kao proširenje Guarinijeve slagalice (engl. *Guarini's puzzle*). Slagalica se sastoji od šest skakača koji su postavljeni na šahovsku ploču dimenzija 3x4: tri crna skakača u prvom redu, te tri bijela skakača u zadnjem, tj. četvrtom redu. Ova zagonetka pripada vrsti problema koji se mogu učinkovito riješiti korištenjem tripartitnog grafa. Glavna ideja koju dijele sve verzije Guarinijeve zagonetke je zamjena pozicija skakača na šahovskoj ploči uz minimalan broj poteza. Ploča je predstavljena grafom, a zatim se graf vizualizira kako bi se dobila jasnija slika problema. Numeriranjem polja šahovske ploče, pozicije će predstavljati vrhove grafa, a mogući potezi skakača povezati bridovima. Također, postoje i različiti problemi koji su slični Guarinijevoj slagalici, kao što su problem prelazak skakača preko Dunava (engl. *Knights Crossing over thee Danube*).

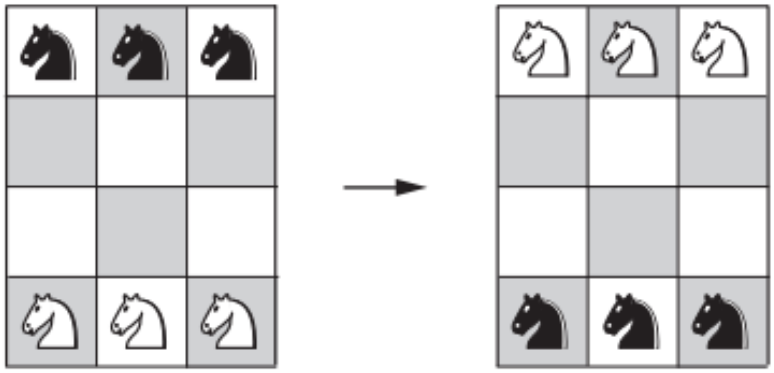
Dakle, cilj rada je opisati postupak pronalaženja rješenja, ako rješenje zagonetke postoji, u minimalan broj poteza za bilo koje početno stanje i ciljno stanje na šahovskoj ploči.

U uvodnom poglavlju ćemo se ukratko osvrnuti na povijest problema skakača, kako je nastao problem, te tko ga prvi put spominje. Nakon toga, spomenut ćemo probleme koji su slični problemu skakača i po čemu se oni razlikuju od ovog problema. Nadalje, govorit ćemo o tome kako se problem pokušao riješiti i kako se riješio. Zatim ćemo upoznati temeljito osnovu koja je temeljena na diskretnoj matematici. Na kraju prvog poglavlja opisan je algoritam koji se koristi pri rješavanju ovog problema.

U drugog poglavlju se pristupa rješavanju problema, algoritmu koji vodi ka rješenju. Prvo se oblikuje problem sa svojim ulaznim i izlaznim parametrima. Matematički se problem formalizira i oblikuju se algoritmi za rješavanje ovog problema. Zatim slijedi implementacija oblikovanih algoritama u programskom jeziku Python, njihovo pojašnjenje i izvršavanje. Rezultati dobiveni izvršavanjem se analiziraju nad određenim algoritmima.

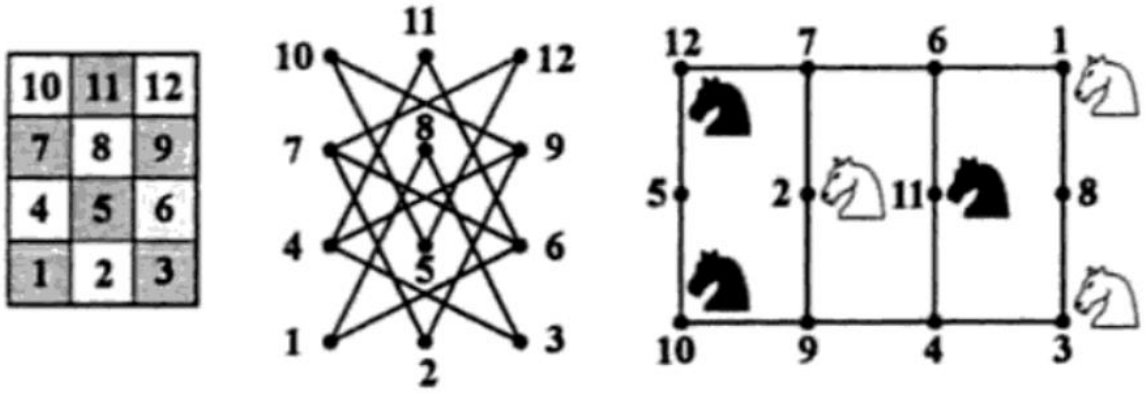
1.1. Povijest problema

Guarinijev problem jedan je od prvih zagonetki koje su zadane na šahovskoj ploči koja datira iz 1512. Ovaj problem je se pripisuje *Paolu Guariniju di Forli* po komu je problem i dobio ime, ali zapravo je pronađen u arapskim šahovskim rukopisima koji datiraju oko 840. godine. Guarinijev klasični problem koristi šahovsku ploču 3x3, dva bijela skakača u gornjim kutnim poljima i dva crna skakača u donjim kutnim poljima. Zadatak je u minimalnom broju poteza pomaknuti bijele skakače na mjesta koja su na početku zauzimali crni skakači i obrnuto. Skakači se mogu pomjerati bilo kojim redoslijedom, bez obzira na njihovu boja. Naravno, niti jedan od njih ne može zauzeti poziciju koja je zauzeta od strane drugog skakača. Henry Ernest Dudeney je dao rješenje 1917. Problem je opisao u kontekstu žaba koje skakuću između lopoča, te on u svom rješenju pripisuje zasluge Guariniju.



Slika 1.1 Problem skakača

Problem skakača koji ćemo obraditi u ovom radu je jedna od varijacija Guarinijevog problema. Dakle, sva pravila su ista samo je razlika u broju skakača i dimenziji šahovske ploče. Ova varijanta Guarinijevog problema prvi put je objavljena u Journal of Recreational Mathematics 1974. i 1975. ali s netočnim odgovorom. Kasnije je objavljeno u Scientific Americanu 1978. i 1979. Posljednje izdanje uključivalo je rješenje koje se sastojalo od 16 poteza što je minimalni broj poteza za riješiti ovaj problem. Dudeney je postavio metodu kako problem skakača lako riješiti primjenom takozvane „metode razotkrivanja grafa“ koja je prikazana sljedećom slikom.

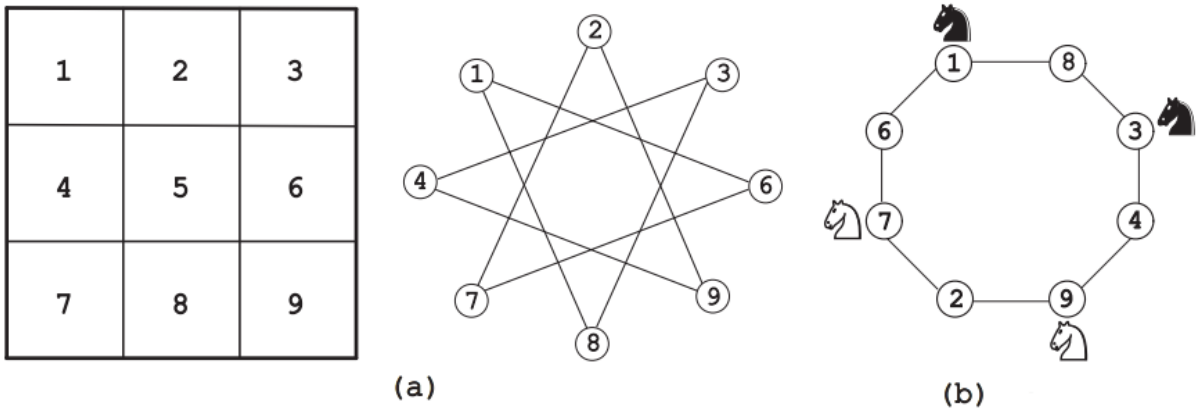


Slika 1.2 Graf mogućih poteza

Na gornjoj slici je prikazana šahovska ploča gdje su polja predstavljena uzastopnim cijelim brojevima. Iz lijevog grafa koji je prikazan na slici možemo vidjeti da su polja predstavljena vrhovima, a mogući potezi skakača između odgovarajućih polja predstavljeni su bridovima. Lijevi graf smo dobili postavljanjem vrhova na način koji oponaša položaje kvadrata na ploči. Ovakav graf nam ne pomaže puno u rješavanju problema i zbog toga koristimo takozvanu „*metodu razotkrivanja grafa*“. Ako postavljamo vrhove duž opsega redoslijedom kojim se može doći iz vrha 1 potezima skakača, dobit ćemo puno jasniju sliku koja je prikazana desnim grafom. Ova dva grafa sa slike 1.2 su ekvivalentna, sačuvana je topološka struktura i povezanost.

1.2. Slični problemi

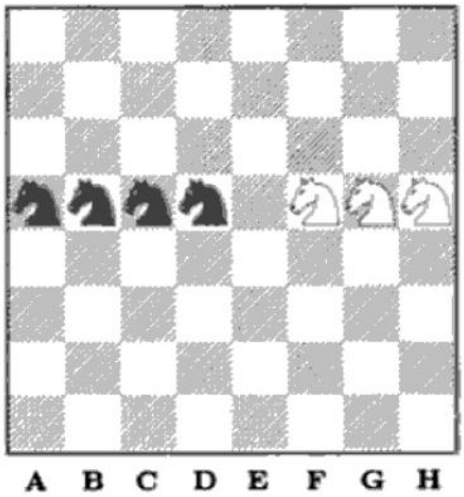
Problem koji je najsličniji problemu skakača je klasični Guarinijev problem. O kojem smo već govorili u uvodnom i povijesnom dijelu. Koristeći Dueneyjevu poznatu „metodu razotkrivanja grafa“ (engl. *method of unraveling graph*) krenuvši od bilo kojeg čvora, na slici ispod graf a) se može oblikovati u ekvivalenti graf b), što nam daje dosta jasniju i jednostavniju sliku za analiziranje.



Slika 1.3 Graf za Guarinijevu klasičnu slagalicu

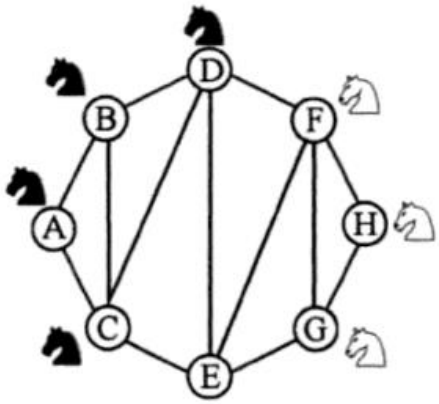
Kod Gaurinijeve klasične slagalice imamo četiri skakača na šahovskoj ploči 3x3: dva bijela skakača su u dva donja kuta (pozicija 7 i 9), a dva crna skakača u dva gornja kuta (pozicija 1 i 3).Cilj je zamijeniti skakače u minimalan broj poteza. Slika 1.3 sadrži prikaz šahovska ploča za klasičnu Gaurinijevu slagalicu, gdje su polja numerirana od 1 do 9. Graf *a)* predstavlja graf kojem su vrhovi polja, a bridovi kojima su polja povezana predstavljaju mogućnost poteza skakača s jedne na drugu poziciju. Takav dobiveni graf je planaran i ako primijenimo metodu razotkrivanja grafa dobit ćemo ciklički graf (*b)*). Sada imamo puno jasniju sliku za analizu. Može se uočiti da svaki potez skakača čuva njihov relativni poredak u smjeru kazaljke na satu, odnosno suprotnom smjeru kazaljke na satu. Stoga postoje samo dva načina za rješavanje ovog problema: pomicati skakače po vrhovima u smjeru kazaljke na satu dok ne dobijemo sliku ploče kakva nam je potrebna .

Prelazak skakača preko Dunava (engl. *Knights Crossing over the Danube*) je problem koji se pojavio 1877. godine. Autor problema je Sam Loyd koji je smatrao ovaj problem jednim od svojih najtežih problema. Kod ovog problema imamo slučaj s četiri crna i tri bijela skakača koji su postavljeni na standardnu šahovsku ploču u petu horizontalu. Ideja problema je zamijeniti crne skakače na a, b, c i d vertikalama s bijelim skakačima na f, g, i h okomicama. Vertikala koja nije zauzeta u početnom stanju, koja dijeli crne i bijele skakače, predstavlja rijeku Dunav i nju moraju prijeći svi skakači. Pri zamjeni skakača ne može se postaviti više od jednog skakača na svakoj vertikali.



Slika 1.4 Početno stanje problema Knights crossing the Dunabe

Na slici 1.4 je prikazano početno stanje na ploči za problem prelazak skakača preko Dunava. Dudeneyjeva metoda razotkrivanja grafa koju smo već spomenuli, može se učinkovito primijeniti u rješavanju problema prelaska skakača preko Dunava. Vrhovi grafa prikažemo kao okomice šahovske ploče. Bridovi povezuju vrhove koji odgovaraju vertikalama između kojih se skakač može kretati. Skakač se kreće bridovima do slobodnog vrha, tj. pozicije. Budući da polja na bilo kojoj vertikali gdje se kreću nisu bitna, rješenje se može bilježiti samo po početnoj i ciljnoj vertikali. Dobiveni graf za rješavanje ovog problema je prikazan slijedećom slikom.



Slika 1.5 Rješenje za problem prelaska skakača preko Dunava koristeći graf

Dakle, možemo reći da je ovaj problem dosta sličan našem problemu. Cilj je isti, a to je da se zamjene pozicije skakača.

1.3. Složenost problema

U teoriji računalne složenosti, NP (engl. *non-polynomial problems*) je klasa složenosti koja se koristi za klasifikaciju problema odlučivanja. NP je skup problema čije rješenje se može provjeriti na deterministočkom Turingovom stroju u polinomalnom vremenu t = N * k (N – dimenzija problema, k- konstanta, t-vrijeme). Problem skakača pripada ovoj vrsti problema. Dakle, lako je provjeriti rješenje zagonetke, ali je teško doći do njega.

Najveći problem će biti pronalazak najkraćih putanja za svakog skakača do svake ciljne pozicije. Iscrpnim pretraživanjem (engl. *brute force*) će se uzeti svaki podskup skupova vrhova i provjeriti je li on vodi do ciljne pozicije, te kolika je njegova duljina putanje, postoji li kraća putanja. Za svakog skakača treba provjeriti koja mu je najkraća putanja do cilja, te je li ta ciljna pozicija zauzeta od strane drugog skakača. Provjeravat ćemo sve moguće permutacije putanja svakog skakača.

1.4. Teorijska podloga

Problem skakača koristi definicije i teoreme diskretne matematike, odnosno teoriju grafova za postavljanje matematičke podloge problema. Prvo ćemo definirati graf koji nam je u ovom problemu ključan za rješavanje.

Definicija 1. Graf je uređeni par $\boldsymbol{G} = (V, E)$, gdje je $\emptyset \neq V = V(G)$ skup **vrhova** (engl. vertex), E = E(G) skup **bridova** (engl. edge) disjunktni s V, a svaki brid $e \in E$ spaja dva vrha $u, v \in V$ koji se zovu krajevi od e[1].

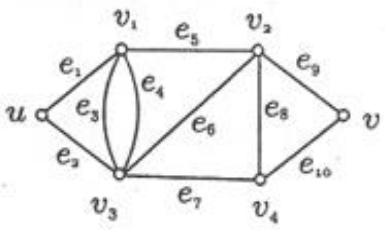
Definicija 2. Jednostavan graf u kojem je svaki par vrhova spojen bridom zove se **potpun graf.**

Definicija 3. Graf G je **bipartitan** (ili **dvodijelni**) ako mu se skup vrhova može particionirati u dva skupa X i Y tako da svaki brid ima jedan kraj u X, a drugi u Y. Particija (X,Y) zove se tada **biparticija** grafa.

Definicija 4. Graf G je **r – partitan graf,** gdje je $r \geq 0$ i čiji skup vrhova možemo podijeliti u r međusobno disjunktnih skupova tako da ne postoji brid između sva vrha u istom podskupu.

Definicija 5. Šetnja u grafu G je niz $W := v_0\ e_1 v_1\ e_2 \ldots\ e_k v_k$, čiji su članovi naizmjence vrhovi v_i i bridovi e_i, tako da su krajevi brida e_i, vrhovi v_{i-1} i v_i, gdje je $1 \leq i \leq k$. Kažemo da je v_0 **početak**, a v_k **kraj** šetnje W te da je W šetnja od v_0 do v_k .

Definicija 5. Ako su svi bridovi $e_1, e_2, \ldots\ e_k$ šetnje W međusobno različiti, onda se W zove **staza**, a ako su na stazi i svi vrhovi $v_0, \ldots, v_k$ međusobno različiti, onda se zove **put**.



(u, v)-šetnja: $ue_1v_1e_3v_3e_7v_4e_7v_3e_6v_2e_9v$
(u, v)-staza: $ue_2v_3e_3v_1e_4v_3e_7v_4e_{10}v$
(u, v)-put: $ue_2v_3e_4v_1e_5v_2e_9v$
$d(u, v) = 3 = \text{diam } G$, struk $G = 2$.

Slika 1.6 Primjer šetnje, staze i puta

1.5. Algoritmi pretraživanja

U informatici, algoritam pretraživanja je algoritam koji obično uključuje i mnoge druge specifične algoritme koji rješavaju problem pretraživanja. Algoritmi pretraživanja rade na dohvaćanju informacija koje su pohranjene u neke baze podataka ili izračunate u prostoru pretraživanja domene problema, s diskretnim ili kontinuiranim vrijednostima.

Za rješavanje problema skakača korišten je algoritam iscrpnog pretraživanja (engl. Brute-force). Algoritam na ulazu prima niz koji pretvara u rječnik u kojem je upisan svaki vrh koji predstavlja ključ rječnika, a vrijednosti su nizovi vrhova koji su povezani s tim vrhom. Problem skakača se može opisati putevima na grafu. Graf je jedna od najčešćih matematičkih struktura za opisivanje prostora stanja pa možemo koristiti algoritme pretraživanja.

1.5.1.Iscrpno pretraživanje

Brute force algoritam, tj. algoritam iscrpnog pretraživanja je najjednostavnija tehnika rješavanja problema u kojoj se ispituju svi mogući načini ili sva moguća rješenja zadanog problema. Dakle korištenjem ovog algoritma sigurno ćemo doći do rješenja ako ono postoji. Algoritam iscrpnog pretraživanja je idealan za rješavanje manjih i jednostavnijih problema, ali je isto tako i neefikasan kada su u pitanju složeniji problemi, kada raste veličina problema.

Algoritam iscrpnog pretraživanja generira sve moguće kombinacije objekata i provjerava jesu li valjani. Sljedećim pseudokodom prikazan je algoritam iscrpnog pretraživanja.

Algoritam: Iscrpno pretraživanje

Input: problem P

Output: kandidati c

```
1 c ← first (P);
2 while c ≠ λ do
3     if valid (P, c) then output (P, c);
4     c ← next(P, c);
5 end while
```

Za implementaciju algoritma iscrpnog pretraživanja potrebna nam je i implementacija procedura: *first, valid, output* i *next*. Navedene procedure kao ulazni parametar primaju problem *P* i rade sljedeće:

1. *first(P)* – generira prvog kandidata za problem *P*
2. *valid(P, c)* – provjerava je li kandidat *c* rješenje problema *P*
3. *output(P, c)* – ispisuje kandidata *c* kao rješenje problema *P*
4. *next(P, c)* – generira sljedećeg kandidata za problem *P* na temelju trenutnog kandidata *c*

Procedura *next* mora javiti kada nema više kandidata za problem *P* na temelju trenutnog kandidata. Najčešće je da vrati prazni kandidat koji je označen s λ. I u slučaju kada nemamo prvog kandidata, procedura *first* će vratiti λ.

Razmotrimo jedan jednostavan primjer iscrpnog pretraživanja, a to je traženje djelitelja broja *n*, instanca problema *P* je broj *n*. Onda će *first(P)* bi trebao vratiti 1 ako je $n \geq 1$, inače vraća λ; pozivanje procedure *next(P, c)* treba vratiti $c + 1$ ako je $c < n$, u suprotnom vraća λ; i procedura *valid(P, c)* treba vratiti *true* ako i samo ako je *c* djelitelj od *n*. Algoritam iscrpnog pretraživanja će pozvati izlaz za svakog kandidata koji je rješenje u danoj instanci *P*.

Ovaj algoritam se lako modificira tako da se zaustavi nakon pronalaženja prvog rješenja, ili određenog broja rješenja, ili nakon testiranja određenog broja kandidata, ili nakon što potroši određeno vrijeme obrade.Glavni nedostatak algoritma iscrpnog pretraživanja je što za mnoge probleme iz stvarnog svijeta broj kandidata c pretjerano velik.

Na primjer, u slučaju kada tražimo djelitelja broja n kao što smo opisali, broj testiranih kandidata bit će samo zadani broj n. Možemo zaključiti da ako n ima šesnaest decimalnih znamenki pretraga će zahtijevati izvršavanje najmanje 10^{15} instrukcija, a takvo izvršavanje programa bi trajalo predugo za standardno računalo. Ako pogledamo primjer traženje određenog rasporeda od 10 slova, u tom slučaju imamo 10! = 3.628.800 kandidata za razmatranje, koje tipično računalo može generirati i testirati za manje od jedne sekunde. No, dodavanjem još jednog slova povećavamo i veličinu podatka od samo 10 %, povećat ćemo broj kandidata na 11 i time povećanje od 1000 %. Nadalje, za 20 slova, broj kandidata je 20! što je 2,4 bilijuna i pretraživanje će trajati oko 10 godina. Ovakav učinak funkcija obično se naziva kombinatorna eksplozija.

Jedan od načina koji bi ubrzao brute-force algoritam je smanjivanje prostora pretraživanja, odnosno skupa mogućih rješenja, korištenjem heuristike specifične klase problema, ali u ovom završnom radu se nećemo baviti time.

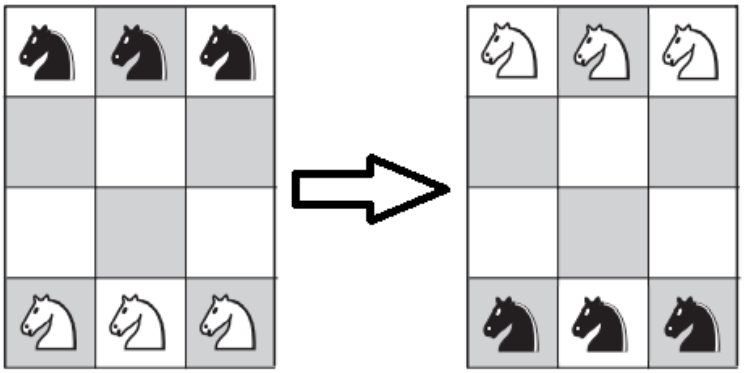
2. Algoritamsko rješavanje problema skakača

Na početku, rješavanju problema se pristupa algoritamski. Prvo se pojašnjava sami problem, tj. opisuju se ulazni i izlazni parametri problema. Uz pomoć teorije grafova opisuje se problem i tako dobivenim matematičkim formulacijama oblikovat ćemo rješenje problema i prilagoditi algoritmu iscrpnog pretraživanja za problem skakača. Algoritam iscrpnog pretraživanja je implementiran u programskom jeziku Python-u, te navedeni specifični koraci algoritma. Na kraju su izmjerene performanse: prikazana je prostorna, vremenska složenost implementiranog rješenja i stopa rasta složenosti.

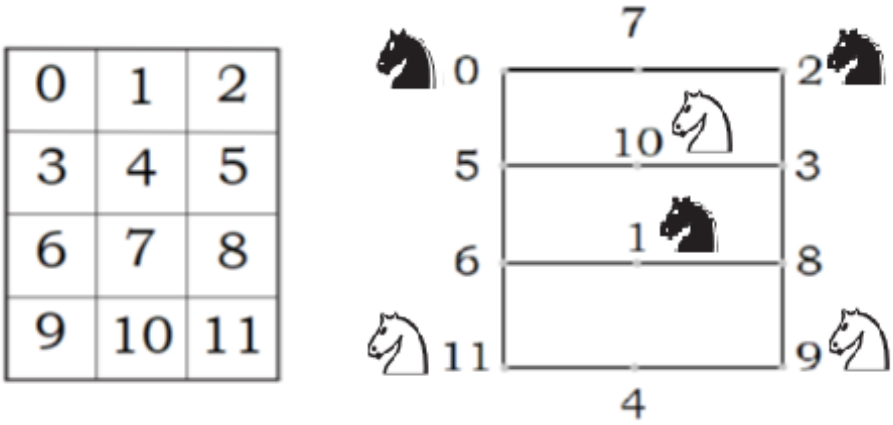
2.1. Razumijevanje problema skakača

Problem skakača je problem koji za danu početnu i ciljnu ploču na kojoj su postavljeni skakači vraća slijed poteza koji su doveli do rješenja u minimalan broj poteza, ako rješenje postoji.

Ulazni parametri su dva stringa, gdje prvi string predstavlja početnu situaciju na šahovskoj ploči i drugi string predstavlja ciljnu situaciju na šahovskoj ploči. Preko ulaznih parametara saznajemo informaciju o dimenziji šahovske ploče pa ćemo i pozicije na šahovskoj ploči označiti brojevima od 0 do n, u našem slučaju od 0 do 11. U ovoj slagalici na šahovskoj ploči postavljeni su skakači, šahovske figure koje se kreću po poljima u obliku slova „L“. Imamo ukupno dvanaest pozicija koje je potrebno povezani na način na koji se skakač kreće po šahovskoj ploči. Pozicije koje su označene brojevima prikazat ćemo kao vrhove, a bridom ćemo prikazati povezanost dviju pozicija. Stvorit ćemo rječnik čiji su ključevi vrhovi, a vrijednosti skup vrhova incidentnih s vrhom koji je ključ.



Slika 2.1 Početno i ciljno stanje na šahovskoj ploči



Slika 2.2 Numerirane pozicije i graf povezanih pozicija

Na slici 2.2. prikazana je tablica sa pozicijama i graf povezanosti pozicija. Možemo sa slike vidjeti kako se koji skakač može kretati.

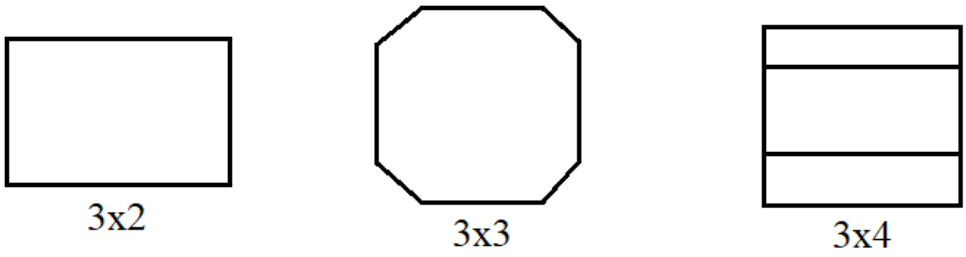
Na primjer, crni skakač na poziciji 0 može na poziciju 7 ili poziciju 5, naravno u uvjet da tu se pozicije slobodne. Slika 2.1. prikazuje vizualizaciju početne i ciljne šahovske ploče. Izlaz iz problema će biti niz poteza koji su doveli rješenja u minimalan broj poteza.

U slučaju kada bismo krenuli na primjer s crnim skakačem na poziciji 2 i premjestili ga na poziciju 3 tada nam sljedeći potez ne smije biti vraćanje spomenutog skakača na prethodnu poziciju. Dakle, ideja rješavanja će biti neponavljanje bilo koje ploče dva ili više puta.

2.2. Matematička formulacija problema

Kod rješavanja problema skakača sada imamo rječnik koji nam predstavlja neusmjeren graf $G = (V, E)$ gdje je V skup vrhova (engl. *Vertex*) i E skup bridova (engl. *Edge*) grafa G.

Graf problema naravno ovisi o ulazu, odnosno o dimenzijama ulazne šahovske ploče. Na slici su prikazani neki od mogućih grafova kao ulaz u problem.



Slika 2.3 Mogući grafovi ulaznih parametara u problem skakača s obzirom na dimenzije šahovskih ploča

Neka je $\{v_0, v_1, \ldots, v_n\}$ skup vrhova, te neka su dvočlani skupovi oblika $\{u, v\}$ elementi skupa E gdje je $u, v \in V$.

Rješenje problema za graf G ulaznih parametara, koji sadrži vrh kao ključ i niz vrhova s kojima je povezan taj vrh, stvorit će niz uređenih parova (c_i, c_j) koji predstavljaju poteze. Prvi član c_i označava poziciju na kojoj se trenutno nalazi skakač, a c_j predstavlja poziciju koja je slobodna i na koju će se skakač pomjeriti. Za uređeni par (c_i, c_j) vrijedi sljedeće:

- $c_i \in V$ – vrh na kojoj se nalazi skakač je u skupu pozicija (vrhova)
- (c_i, c_j) – dva susjedna vrha povezana bridom, predstavljaju potez skakača
- $c_i \neq c_j$ vrh na kojem se nalazi skakač mora biti različit od vrha na koji ide skakač

2.3. Oblikovanje rješenja problema skakača

Rješenje problema skakača je algoritam koji za dani neusmjereni graf vraća niz poteza koji vode do rješenja u minimalan broj poteza. S obzirom da su algoritmi za pretraživanje pogodni za pronalaženje optimalnog rješenja, oblikovat ćemo rješenje jednim od ove vrste algoritama, algoritmom iscrpnog pretraživanja. Ovaj algoritam je najjednostavniji algoritam za pretraživanje, on generira sve moguće kandidate za rješenje, ali prije implementiranja algoritma iscrpnog pretraživanja moramo implementirati algoritme kojima ćemo preoblikovati ulazne parametre, pronaći kretnje skakača, definirati klase sa atributima i metodama, te na kraju vizualizacija rješenja u koracima.

2.3.1.Prikaz šahovske ploče

Budući da radimo na konkretnom primjeru problema skakača ulazni parametri će biti početna ploča i ciljna ploča koje će biti prikazane stringovima "BBB\n…\n…\nWWW" i "WWW\n…\n…\nBBB". Zadani stringovi sadrže znakove „B“, „W“, „\n“ i „.“. Znak „B“ (black) označava crnog skakača na određenoj poziciji, znak „W“ (white) predstavlja bijelog skakača, string „\n“ označava novi red i znak „.“ predstavlja poziciju koja je prazna, tj. slobodna.

Ideja je preoblikovati prikaz situacije na šahovskoj ploči tako da umjesto znakova „B“, „W“ i „.“ uvedemo brojeve 0, 1 i 2. Sada će nam 0 predstavljati bijele skakače, 1 će predstavljati crne skakače i 2 će označavati poziciju koja je prazna. Dakle, implementirat ćemo funkciju koja će kao ulaz primati ploču u obliku *"BBB\n…\n…\nWWW"* i kao rezultat vratiti ploču koja će biti u obliku *"111222222000"*.

```
1 def transform(board_str):
2    result = ''
3    for line in bord_str.strip().split('\n'):
4        for item in line:
5            if item == 'B':
6                result = result + '1'
7            elif item == 'W':
8                result = result + '0'
9            elif item == '.':
10               result = result + '2'
11   return result
```

Kod 2.1 Implementacija funkcije za preoblikovanje situacije na šahovskoj ploči

Kod 2.1. koristimo neke od ugrađenih funkcija:

- *.strip()* – funkcija u programskoj jeziku Pythonu koja vraća kopiju niza s uklonjenim znakovima na početku i na kraju
- *.split()* – funkcija prima jedan argument koji se naziva separator i dijeli string na niz stringova po zadanim separatorom

opisat ćemo na radnom primjeru gdje je ulazni parametar izgled početne situacije na šahovskoj ploči.

- 1: definirali smo funkciju *transform* koja prima kao ulazni parametar ploču (board_str)
- 2: definiramo prazan string *result* u koji ćemo dodavati znakove
- 3: uzima se svaka linija iz *board_str.strip().split("\n")*. Prva linija će biti *"BBB"*
- 4: uzima se svaki znak u liniji. Prvi znak je *"B"*.
- 5: provjerava se je li znak je
- 6: ako je varijabli *result* dodaj znak "1" . Ova linija će se izvršiti jer znaš trenutni znak zadovoljava uvjet i varijabla *result* će sada imat vrijednost "1".
- 7: ako prethodni uvjet nije zadovoljen, provjerava se li je znak jednak *"W"*
- 8: ako je varijabli *result* dodaj "0".
- 9: ako prethodni uvjeti nisu ispunjeni provjeri je li znak jednak "."
- 10: ako je varijabli *result* dodaj "2"

Sada se vraćamo na instrukciju broj 4 i provjerava se sljedeći znak, a to je *"B"*. Kada se izvrši provjera nad zadnjim znakom iz linije *result* će imati vrijednost "111".Sada se vraćamo na instrukciju 3 i prelazi se na sljedeću liniju ("…"). Postupak se ponavlja sve dok se ne izvrši provjera nad zadnjim znakom zadnje linije.

- 11: na kraju vrati string *result.* U ovom slučaju će to biti "111222222000".

Trebat će nam i funkcija koja radi suprotno od funkcije *transform.* Implementirat ćemo kod koji kao ulaz prima string u obliku "111222222000" i vraća rezultat u obliku *"BBB\n…\n…\nWWW"*. Razlika je još što će nam ova funkcija trebati dimenzije šahovske ploče pa ćemo izraditi i funkciju *dimension* koja će kao parametar primati početnu ploču i vratit će nam broj redaka i stupaca.

```
1 def dimension(board_str):
2     row = 0
3     col = 0
4     for line in board_str.strip().split('\n'):
5         row += 1
6         col = len(line)
7     return row,col
```

Kod 2.2 Implementacija funkcije koja na vraća broj redaka i stupaca za zadanu šahovsku ploču

Pozovemo li funkciju *dimension* i kao ulazni parametar stavimo početnu ploču koja treba biti u obliku *"BBB\n…\n…\nWWW"* kao rezultat će nam vratiti 3 i 4, što odgovara broju redaka i stupaca.

```
1 def transform2(board_str,i):
2     rows, col = dimension(board_str)
3     board = []
4     for j in range(rows):
5         row = ''
6         for k in range(col):
7             if i[k] == '1':
```

```
8                   row = row + 'B'
9               elif i[k] == '0':
10                  row = row + 'W'
11              else:
12                  row = row + '.'
13          i = i[col:]
14          board.append(row)
15      return board
```

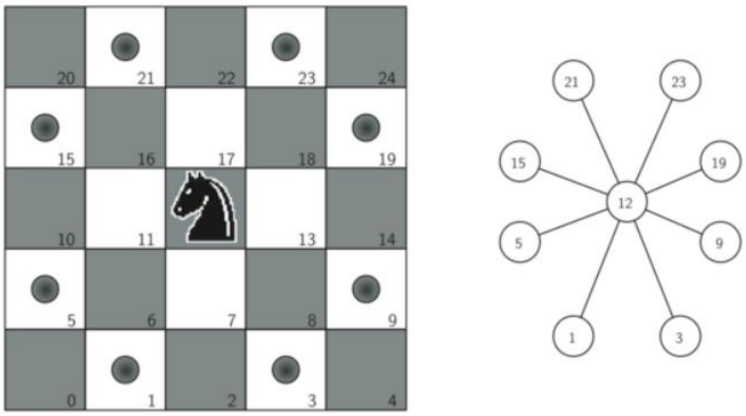
Kod 2.3: Implementacija funkcije za preoblikovanje situacije na šahovskoj ploči

Dakle, funkcija *transform2* radi suprotno od funkcije *transform*, ali će kao rezultat vratiti niz stringova. Za ulaz *"111222222000"* vratiti će *['BBB', '...', '...', 'WWW']*.

2.3.2. Algoritam za pronalaženje poteza skakača

Kako bismo predstavili problem skakača kao graf, koristimo dvije ideje:

- svako polje na šahovskoj ploči može se prikazati kao vrh u grafu
- svaki ispravan potez skakača s jedne pozicije na drugu poziciju može se prikazati kao brid koji spaja vrhove



Slika 2.4 Mogući potezi skakača na polju 12 i odgovarajući graf

```
1 def initialize(board_str):
2     idx2coord = {}
3     coord2idx = {}
4     counter = 0
5     jumps = {}
6
7     i, j = 0, 0
8     for line in board_str.strip().split('\n'):
9         for item in line:
10            idx2coord[counter] = (i, j)
11            coord2idx[i, j] = counter
12            j += 1
13            counter += 1
14        i += 1
15        j = 0
16
17    for idx1, (i, j) in idx2coord.items():
18       for di, dj in [(-2, -1), (-2, +1), (+2, -1), (+2,+1),
19                      (-1, -2), (+1, -2), (-1, +2), (+1,+2)]:
20           ni = i + di
21           nj = j + dj
22           if (ni, nj) in coord2idx:
23               idx2 = coord2idx[ni, nj]
24               if idx1 not in jumps:
25                   jumps[idx1] = []
26               jumps[idx1].append(idx2)
27    return jumps
```

Kod 2.4: Implementacija funkcije koja vraća rječnik kojem su vrhovi ključevi, a vrijednosti niz vrhova s kojima je povezan

Implementacijom funkcije *initialize* koja prima početnu situaciju na ploči dobili smo rječnik sa ključevima koji predstavljaju pozicije na ploči kao i niz pozicija s kojima su povezani s obzirom na pravilo kretanja skakača u šahu.

Opisat ćemo kako funkcija *initialize(board_str)* prikazana kodom 2.4. radi za ulazni parametar "BBB\n...\n...\nWWW". Prvo smo stvorili prazne rječnike (*idx2coord*, *coord2ix* i *jumps)*, postavili brojač na 0 i varijablama *i* i *j* smo dodijelili jednakost 0. Uzimamo svaku liniju u ulaznom parametru kojem smo uklonili bilo kakve znakove ili razmake na početku i kraju ako ih ima (funkcijom *.strip()*) i podijelili na stringove separatorom *"\n"* (funkcijom *.split("\n")*). Prva linija će biti *"BBB"*. Sljedeće što moramo napravit je uzeti svaki znak u liniji, a prvi znak je *"B"*. U rječnik *idx2coord* spremat ćemo koordinate pozicija s ploče. Prvi znak nalazi se na koordinati *(i, j)*, a njihova vrijednost je *(0, 0)*. Pod ključem brojača koji je 0 (što označava poziciju) spremit ćemo vrijednost koordinate (*idx2coord[counter] = (i, j)*). A u rječnik *coord2ix* spremat ćemo vrijednost *counter*-a pod ključem koji odgovara koordinatama (*coord2ix[i, j] = counter)*. Nakon toga *j* i *counter* uvećavamo za 1 jer prelazimo na sljedeću poziciju koja se nalazi na sljedećoj koordinati. Sljedeći znak je *"B"* i sljedeća koordinata je *(0, 1)*, a *counter* je jednak 1. Vrijednosti ćemo spremiti u rječnike (*idx2coord[1] = (0, 1)* i *coord2ix[0, 1] = 1)* i uvećati *j* i *counter* za 1. Tako će se nastaviti niz izvršavanja kada dođemo do zadnjeg znaka u liniji i spremimo vrijednosti u rječnike uvećat ćemo *i* za 1 i *j* postaviti na 0 jer prelazimo u novi red. Kada dođemo do zadnjeg znaka zadnje linije tada će *counter* imati vrijednost 11, a koordinata će biti *(3,2)*, onda prelazimo na sljedeću *for* petlju. S obzirom na to da se skakača kreće na ploči u obliku slova L, moramo povezati pozicije na isti način. Uvest ćemo niz koordinata koje će nam pomoći pri povezivanju pozicija. Sa slike 2.4. možemo primijetiti da su potezi skakača sa trenutne pozicije formirani na sljedeće načine:

- dva polja lijevo i jedno polje dolje što ćemo koordinatama prikazati (-2, -1)
- dva polja lijevo i jedno polje gore (-2, +1)
- jedno polje lijevo i dva polje gore (-1, +2)
- jedno polje lijevo i dva polje dolje (-1, -2)
- dva polja desno i jedno polje gore (+2, +1)
- dva polja desno i jedno polje dolje (+2, -1)
- jedno polje gore i dva polja lijevo (+1, -2)
- jedno polje gore i dva polja desno (+1, +2)

Uzimamo sve ključeve (*idx1)* i vrijednosti (*i, j)* iz rječnika *idx2coord*. Uzet ćemo sve koordinate (označene s *di, dj)* iz niza koji smo formirali za kretanje skakača i provjeriti nalazi li se izračunata koordinata kao ključ u rječniku *coord2ix*. Prve vrijednosti će biti *idx1 = 0* , *(i, j) = (0, 0), di = -2* i *dj =-1*. Deklarirat ćemo nove varijable *ni = i + di* i *nj = j + dj* koje će predstavljati nove koordinate po kojima se skakač može kretati s koordinate *(i, j)*. Prve

vrijednosti će biti *ni = 0 - 2 = -2* i *nj = 0 – 1 =-1*. Sljedeći korak je provjeriti je li se koordinata *(ni, nj)* nalazi u *coord2ix*, ako je uvjet ispunjen u varijablu *idx2* spremamo vrijednosti *coord2ix[ni, nj]* i ako *idx1* nije u rječniku *jumps* onda stvaramo ključ u rječniku i vrijednost mu je prazan niz (*jumps[idx1] = []*). Nakon što smo varijabli *idx2* dodijelili vrijednost i provjerili uvjet u rječnik *jumps* u vrijednost ključa *idx1* dodajemo vrijednost *idx2 (jumps[idx1].append(idx2)*. Vrijednost trenutne koordinate *(ni, nj)* nije u rječniku *coord2ix* pa prelazimo na sljedeće vrijednosti *di* i *dj*. Za *idx1 = 0* ćemo dobiti pozicije 5 i 7 s kojima je povezan. Na ovaj način ćemo dobiti za svaku poziciju kao ključ rječnika *jumps* niz pozicija kao vrijednosti. Konačni rezultat će biti rječnik *jumps ({0: [7, 5], 1: [6, 8], 2: [7, 3], 3: [10, 2, 8], 4: [9, 11], 5: [10, 0, 6], 6: [1, 5, 11], 7: [0, 2], 8: [1, 3, 9], 9: [4, 8], 10: [3, 5], 11: [4, 6]})*

2.3.3.Definiranje potrebnih klasa

Objektno-orijentirano programiranje koristi objekte i njihove interakcije za izgradnju računalnih programa. Na taj način se dobije razumljiv model nekog područja ili problema. Tehnike objektno-orijentiranog programiranja uključuju pojmove kao što su:

- enkapsulacija – skrivanje nepotrebnih detalja klasa radi jasnijeg i jednostavnijeg rada
- nasljeđivanje – omogućuje klasama nasljeđivanje polja i metoda od nadređenih klasa
- apstrakcija – izdvajanje bitnih svojstava objekta, omogućuje nam rad s objektom bez da znamo njegov unutarnji izgled
- polimorfizam – različito ponašanje

Klasa definira apstraktne karakteristike objekta, a svaki objekt je instanca samo jedne klase. Danas je veliki broj programskih jezika objektno-orijentiran, te kako je objektno-orijentirano programiranje (OOP) blisko načinu razmišljanja čovjeka, upotreba takvog jezika trebala bi olakšati rješavanje problema, odnosno modeliranje stvarnog svijeta. Klase se definira ključnom riječi *class* iza koje slijedi naziv ili ime klase. Osnovni elementi klase su:

- deklaracija klase – linija u kojoj deklariramo naziv klase
- tijelo klase
- konstruktor – za stvaranje novih objekata
- polja (engl. fields) – varijable koje su deklarirane unutar klase
- metode – imenovani blokovi koda koji obavljaju određene klase

2.3.3.1. Klasa Board

```
1 class Board():
2     def __init__(self, board: str):
3         self.current_board = board
4         self.history: list[str] = list()
5 
6     def __eq__(self, other):
7         return self.current_board == other.current_board and
8                 len(self.history) == len(other.history)
9 
10    def __le__(self, other):
11        return self.current_board == other.current_board and
12                len(self.history) <= len(other.history)
13 
14    def make_move(self, home: int, landing: int):
15        self.history.append(self.current_board)
16        board = list(self.current_board)
17        board[home], board[landing] = board[landing],
18                                      board[home]
19        self.current_board = "".join(board)
20 
21    def is_empty(self, landing: int) -> bool:
22        if self.current_board[landing] == "2":
23            return True
```

Kod 2.5 Definirana klasa Board sa svojim konstruktorom i metodama

Kao što smo već rekli klasu smo definirali ključnom riječi *class* i nakon toga slijedi naziv klase, u našem slučaju to je *Board*. Nakon što smo definirali klasu stvorili smo konstruktor koji služi za inicijalizaciju objekta. Metoda *"__init__"* je konstruktor, suprotno onome što taj naziv implicira, konstruktor ništa ne konstruira ili kreira, on isključivo služi za inicijalizaciju novonastalog objekta. Poput metoda, konstruktor također sadrži niz instrukcija koje se izvode u trenutku stvaranja objekta. Pokreće se čim se instancira objekt klase. Varijable objekta kao što imamo u klasi *Board current_board* (trenutna ploča) i *history* (povijest poteza) nazivaju se

poljima ili atributima. Konstruktorom najčešće inicijaliziramo polja objekta iako on može sadržavati bilo kakav kod.

U *Python*-u postoji skup posebnih funkcija koje predstavljaju određene operatore. U našoj klasi imamo funkcije *"__eq__"* i *"__le__"*. Funkcija *"__eq__"* (engl. *equality*) predstavlja operaciju jednakosti. Time bi izraz $a == b$ zapravo bio poziv metode "*__eq__"*, to jest *a.__eq__(b)*. Ovo načelo po kojem u *Python*-u definiramo operatore za nove (neugrađene)tipove podataka. Operator jednakosti za klasu *Board* definira se kako je prikazano u kodu. Za izraz $a == b$ parametar *self* sadržavao bi objekt *a*, a parametar *other* objekt *b*. S obzirom da relacijski operatori trebaju vratiti logičku vrijednost kao rezultat tako i metoda "*__eq__*" vraća rezultat logičkog izraza kojim uspoređujemo vrijednosti *self.current_board == other.current_board* i vrijednosti duljina *len(self.history) == len(other.history)*. Metoda "*__le__*" poziva metode "*__lt__*" (za operator "<") i "*__eq__*" (za operator "=="). U ovom slučaju koristimo operator za manje ili jednako (engl. *less-or-equal)*. Ova metoda nam vraća rezultat logičkog izraza kojim uspoređujemo vrijednosti trenutnih ploča i duljinu povijesti poteza (*self.current_board == other.current_board* i *len(self.history) <= len(other.history))*.

Sljedeća metoda je *make_move* (napravi potez) koja u povijesti poteza, odnosno stanja na ploči dodaje i trenutnu ploču (*self.history.append(self.current_board)*) i varijabli *board* pridružuje trenutnu ploču koja je pretvorena u niz stringova (za self.*current_board = "111222222000" board = ['1', '1', '1', '2', '2', '2', '2', '2', '2', '0', '0', '0']*). Ova metoda kao ulazne parametre osim *self* prima i dvije varijable vrste *integer*. Prvi broj je trenutna pozicija skakača (*home*), a drugi broj je slobodna pozicija na koju će skakač doći (*landing)*. U listi *board* izmjenit ćemo vrijednosti tih dviju pozicija kao da je skakač napravio potez. Na primjer, uzet ćemo da je *current_board = "111222222000"* i parametri *home=7 landing= 2*. Tada će se u *history* dodati *current_board* i *board će imati vrijednost = ['1', '1', '1', '2', '2', '2', '2', '2', '2', '0', '0', '0']*. *Board[home] = '2'* i *board[landing] = '1'* te dvije vrijednosti ćemo zamjeniti i *current_board* će sada imati vrijednosti *"112222212000"*.

Zadnja metoda u klasi *Board* je *is_empty* koja prima *self, landing* i vraća bool vrijednost. Ulazni parametar *landing* predstavlja poziciju na ploči koju provjeravamo je li slobodna, ako je metoda vraća *True*. Ako bismo pozvali funkciju *is_empty("111222222000", 7)* funkcija bi nam vratila *True* jer je *self[7] = "2"*, a *"2"* predstavlja slobodnu poziciju.

2.3.3.2. Klasa Game

```
1 class Game():
2     def __init__(self, initial_board: Board, goal: str,
3                  edges: dict[int, list]):
4         self.boards: deque[Board] = deque([initial_board])
5         self.goal = goal
6         self.edges = edges
7         self.history: deque[Board] = deque()
8 
9     @staticmethod
10    def make_move(board: Board, home: int, landing: int) ->
11                  Board:
12        board = deepcopy(board)
13        board.make_move(home, landing)
14        return board
15 
16    def check_goal(self, board: Board) -> bool:
17        if board.current_board == self.goal:
18            return True
19 
20    def is_valid_move(self, new_board: Board):
21        return not (
22            any(
23                new_board == board
24                for board in self.history
25            )
26            or any(
27                new_board <= board
28                for board in self.boards
29            )
30        )
31    def bfs(self):
```

Kod 2.6 Definirana klasa Game sa svojim konstruktorom i metodama

Definirali smo klasu *Game* i stvorili smo konstruktor "*__init__"* koji ima svoje atribute. Atribut *self.boards* je kao vrsta podatka *deque. Deque (Doubly Ended Queue)* u *Python*-u je implementiran pomoću modula kolekcije. Bolje je koristiti *deque* u odnosu na listu u slučajevima kada su nam potrebne brže operacije dodavanja i izbacivanja s oba kraja spremnika. Budući da *deque* pruža O(1) vremensku složenost za operacije dodavanja i izbacivanja u usporedbi s listom koja pruža O(n) vremensku složenost. Polje *self.goal* je vrijednost ciljne ploče, *self.edge* su svi ključevi u rječniku koje predstavljaju vrhove (pozicije) i *self.history* je također *deque* i u njega se sprema stanje ploče kao povijest.

Metode koje u svom prvom parametru *self* ne prihvaćaju instancu te klase nazivaju se *statičkim metodama.* Statičkim, zato jer ne rade s instancama klase (koje su kreirane dinamički, odnosno u toku izvršavanja programa) nego sa samom klasom. Zbog toga statičke metode nemaju pristup instancama svoje klase pa nemaju ni pristup podacima tih instanci. Takve metode obično upotrebljavaju za sljedeće:

- Kao funkcije koje trebaju pristup statičkim poljima i statičkim metodama klase u kojoj su definirane, ali koje ne trebaju pristup instancama te klase.
- Kao pomoćne funkcije kojima klasa u kojoj su definirane služi samo kao imenski prostor.

U klasi *Game* imamo statističku metodu *make_move* i vidimo da ona nema *self* kao prvi parametar, nema pristup instanci klase. U ovoj metodi ćemo koristiti *deepcopy,* odnosno duboko kopiranje.

Duboko kopiranje je proces u kojem se proces kopiranja odvija rekurzivno. To znači prvo konstruirati novi objekt zbirke, a zatim ga rekurzivno popuniti kopijama određenog objekta u orginalu. U slučaju duboke kopije, kopija objekta se kopira u drugi objekt, što znači da se sve promjene napravljene na kopiji objekta ne odražavaju na izvorni objekat. U *Python*-u se to implementira pomoću funkcije *deepcopy().* Duboku kopiju ćemo napraviti od varijable *board* i nakon toga pozvati pozvati metodu *make_move,* ali onu koja se nalazi u klasi *Board* i poslati ulazne parametre. Kao rezultat statičke metode dobit ćemo ploču koja je izmijenjena metodom *make_move.*

Sljedeća metoda je *check_goal* koja vraća bool vrijednost. Ona provjerava je li trenutna ploča jednaka onoj ploči koja je zadana kao ciljna, ako je onda metoda vraća *True.*

Ostala nam je zadnja metoda *is_valid_move,* prije algoritma za iscrpno pretraživanje koje ćemo u sljedećem poglavlju detaljnije razmotriti. Metoda *is_valid_move* provjerava je li potez ispravan. Pregledava se cijela povijest do tad spremljenih ploča i metoda vraća *not* ako nova ploča, koja je zadana kao ulazni parametar već postoji u povijesti ploča (*history*).

2.3.4.Iscrpno pretraživanje za problem skakača

Kod iscrpnog pretraživanja, generiraju se sve moguće permutacije vrhova zadanog povezanog grafa i za svaku se provjerava je li rješenje. U algoritmu se koriste permutacije bez ponavljanja, jer se ne smije ponoviti ista ploča dva puta. Algoritam iscrpnog pretraživanja uzima prvog skakača koji mu je na redu te generira moguće poteze za tog skakača s te pozicije i generira moguće ploče koje će nastati kada se skakač pomjeri s pozicije na kojoj se nalazi. Nakon toga provjerava se je li dobivena ploča jednaka ciljnoj ploči, ako je izbacuju se ploče koje su se izmijenile to rješenja. A u slučaju da dobivena ploča ne odgovara ciljnoj ploči, ploča se sprema i na osnovu te ploče dalje se generiraju novi potezi i nove ploče. Ovaj postupak se ponavlja sve dok se ne dođe do rješenja.

Do sada smo implementirali sve klase i funkcije koje su nam potrebne za implementaciju zadnjeg algoritma. Iscrpno pretraživanje će biti metoda unutar klase *Game* i ta metoda će primati *self* (*bfs(self)).*

```
1  def bfs(self):
2          start_time = time.perf_counter()
3          while self.boards:
4              board = self.boards.popleft()
5              if self.check_goal(board):
6                  print(f"Pronađeno rješenje u
7                        {len(board.history)}poteza")
8                  return board.history
9              self.history.append(board)
10             #print(len(board.history))
11            for home in [i for i, piece in
```

```
                    enumerate(board.current_board)
                    if (int(piece) == 0 or int(piece) ==
                        1)]:
            for landing in self.edges[home]:
                if board.is_empty(landing):
                    new_board = Game.make_move(board,
                                home, landing)
                    if self.is_valid_move(new_board):
                        self.boards.append(new_board)
```

Kod 2.7: Implementacija algoritma za iscrpno pretraživanje

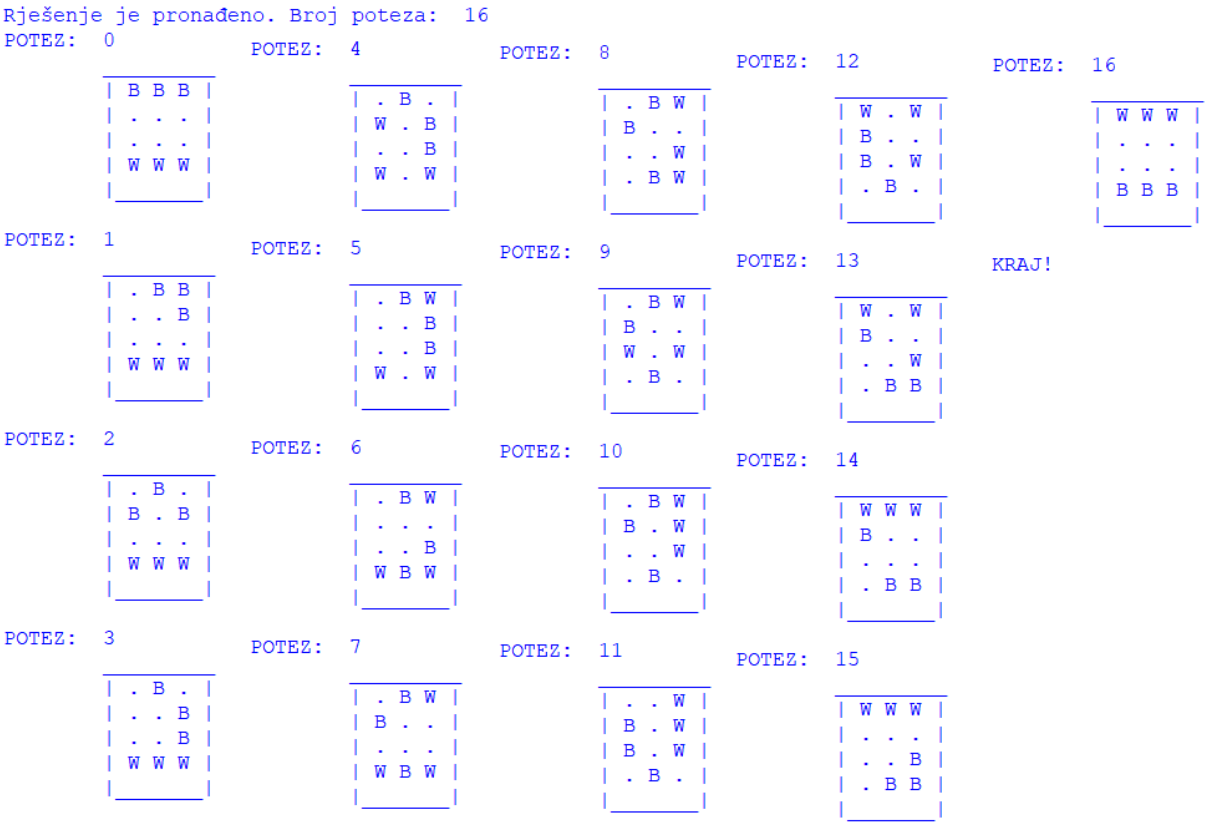
Već smo pokazali da kasa *Game* ima atribute *self.boards, self.goal, self.edges* i *self.history*. Objasnili što što predstavlja i svaki atribut i kakve je vrste podatka. U metodi *bfs* koristit ćemo i metode *check_goal(), is_empty(), make_move()* i *is_valid()* koje smo također sve objasnili kako rade i koji rezultat vraćaju. Također ćemo koristiti i klasu *Board* sa svojim atributima i metodama.

Kada pozovemo metodu *bfs()*, metoda se izvršava sve dok *self.boards* nije prazan. Definiramo varijablu *board* i dodjeljujemo joj vrijednosti prvog lijevog elementa iz deque-a *self.boards*, a to ćemo napraviti uz pomoć funkcije *.popleft()*. Dakle ako je naša početna ploča "*111222222000"* koja se nalazi u *self.boards*, pošto ona jedina u *self.boards* onda će i varijabla *board* imati vrijednost početne ploče sa svojim atributima. Postavljamo uvjet je li trenutna ploča jednaka ciljnoj ploči i ako je vrati *history*, ako nije dodaj ploču u *history*. U sljedećoj for petlji uzimamo svaki *home* iz niza u kojem su pozicije na kojima se nalaze skakači na trenutnoj ploči.

Niz smo dobili tako što smo uzeli svaki indeks i vrijednost koja je na mjestu indeksa na trenutnoj ploči i provjerili je li ta vrijednost 1 ili 0, što označava skakače i ako je dodaje se u niz njihov indeks. Pa ćemo za trenutnu ploču "111222222000" za prvi *i* i *piece* imati *tuple* (0,"1") i *int(piece) == 1* pa ćemo u niz dodati vrijednost *i*, odnosno 0. Istu stvar ćemo napraviti za svaki znak i na kraju dobiti niz *[0,1,2,9,10,11]*. Zatim, za svaki *landing* iz skupa vrhova koji su povezani s vrhom *home* (prvi *home* će biti 0 i prvi skup vrhova *self.edge[home] = [7,5]*). Dalje provjeravamo je li uzeta pozicija *landing* na trenutnoj ploči slobodna pozicija (*board.is_empty(landing)*). Ako je uvjet zadovoljen neka nova ploča bude trenutna ploča kojoj su vrijednosti *home* i *landing* zamijenjene. Moramo provjeriti je li se nova ploča već spremila u *history*, a to ćemo provjeriti metodom *.is_valid_move(new_board)*. Ako metoda vraća *True*

onda u *self.boards* dodajemo novu ploču. Postupak se ponavlja sve dok *self.boards* nije prazan ili ako smo došli do rješenja.

Rješenje koje smo dobili pomoću iscrpnog algoritma pretraživanja smo vizualizirali i dobili rješenje koje je prikazano na slici ispod.



Slika 2.5 Rješenje problema skakača u minimalan broj poteza

2.4. Analiza izvršavanja implementiranog algoritma

Algoritam implementiran za rješavanje problema skakača je algoritam pretrage i samim time jasno je da će se povećavanjem broja skakača na ploči i povećavanjem dimenzija ploče povećati i broj mogućih rješenja, broj mogućih poteza, vrijeme izvršavanja algoritma i složenost.

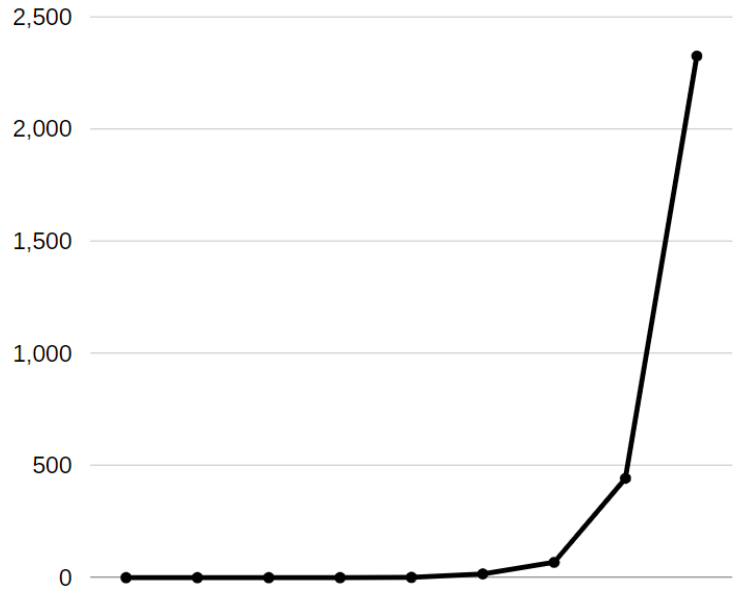
Sljedećom tablicom prikazano je kako na ploči dimenzija 3x4 s povećanjem broja skakača raste broj mogućih poteza i vrijeme utrošeno na pronalaženje prvog rješenja. Vrijeme izmjereno će se odnositi na raspored gdje su skakači jedne boje u gornjem redu i skakači druge boje u zadnjem redu, te na slučajeve kada imamo skakače samo jedne boje u prvom redu koji moraju doći na dno ploče.

Broj skakača jedne boje	Broj skakača druge boje	Broj mogućih ploča	Vrijeme utrošeno na pronalazak prvog rješenja u minimalan broj poteza
1	0	12	0.039 sekundi
2	0	66	0.048 sekundi
1	1	132	0.074 sekundi
3	0	220	0.171 sekundi
1	2	660	1.451 sekundi
1	3	1980	16.752 sekundi
2	2	2970	68.509 sekundi
2	3	7920	443.168 sekundi
3	3	18480	2325.753 sekundi

Tablica 2.1 Prikaz rezultata mjerenja za algoritam iscrpnog pretraživanja

U tablici 2.1 prikazani su rezultati mjerenja vremenske složenosti algoritma iscrpnog pretraživanja. Za slučajeve kada imamo skakače samo jedne boje vidimo da vremenska složenost ne raste naglo i da su takve ploče lakše i brže rješive. Problem se pojavljuje kada uvodimo skakače druge boje. Složenost u tom slučaju raste i vrijeme izvođenja se povećava, te se na rješenje čeka već po par sekundi. Slučaj postane još složeniji kada imamo 4 skakača (2 crna i 2 bijela) i tad se program izvršava duže od minute što je već previše, a najgori slučaj

imamo je kada su na ploči šest skakača (3 crna i 3 bijela) gdje smo na rješenje čekali više od 38 minuta. Dakle, možemo zaključiti da što je veći broj skakača s manjom razlikom između broja bijelih i broja crnih skakača to je veći broj mogućih ploči koje će se pojaviti bez ponavljanja. Iz tablice možemo iščitati i slijedeće: što je veći broj mogućih ploča time i raste vremenska složenost i postaje sve očitija.



Slika 2.6 Graf rasta vremenske složenosti povećanjem broja skakača

Graf na slici 2.6 prikazuje vremensku složenost za algoritam iscrpnog pretraživanja, s kojeg se može bolje vidjeti kako vremenska složenost raste s povećavanjem broja skakača.

3. ZAKLJUČAK

Tema ovog rada je bila opisati problem i način rješavanja problema skakača. Problem skakača je poznat stoljećima. Usko je vezan s teorijom grafova iz matematičkog aspekta pa je problem dosta zanimljiv i ne samo matematičarima već i računalnim znanstvenicima.

U ovom radu je opisan postupak rješavanja problema skakača za konkretan slučaj kada imamo tri crna i crni bijela skakača. Skakači su raspoređeni na šahovskoj ploči dimenzija 3x4 tako da su crni skakači u prvom redu i bijeli skakači u zadnjem redu. Algoritam koji smo koristili za rješavanje ovog problema je iscrpno pretraživanje. Iscrpno pretraživanje je najjednostavniji od algoritama pretrage i rješenje je zagarantirano, naravno ako rješenje postoji.

Rješenje problema skakača je algoritam koji za dani neusmjereni graf vraća niz poteza koji vode do rješenja u minimalan broj poteza. Implementirali smo algoritam u programskom jeziku *Python.* Također smo u programskom kodu deklarirali sve potrebne klase, metode za pronalazak poteza skakača i prikaz ploče.

Na kraju smo napravili analizu implementiranog algoritma i prikazali vremensku složenost.

LITERATURA

1. Darko Veljan. *Kombinatorna i diskretna matematika.* Algoritam, Zagreb, 2001.
2. Wikipedia. Brute-force search, 2022. URL Brute-force search. - Wikipedia
3. Miodrag Petković, Ph.D. *Matematics and Chess: 110 Entertaining Problems and Solutions.* Dover Pulications, Inc. Mineola, New York, 2003.
4. RUNESRONE ACADEMY. Building the Knight's Tour Graph, 20222. URL https://runestone.academy/ns/books/published/cppds/Graphs/BuildingtheKnightsTourGraph.html
5. Aleksandar Stojanović, Željko Kovačević. *Uvod u programski jezik Python.* Tehničko veleučilište u Zagrebu, Zagreb, 2022.
6. Anany Levitin, Maria Levitin. *Algorithmic puzzle.* Oxford University Press, New York,2011.