Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Mitja Rizvič

# Avtomatsko odkrivanje zanimivih šahovskih problemov

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matej Guid

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Razvijte računalniški program, s katerim bi si ljudje lahko pomagali pri reševanju in ustvarjanju zanimivih šahovskih problemov. Šahovski problem običajno zahteva premikanje figur na šahovnici z uporabo klasičnih šahovskih pravil in ni nujno povezan z matiranjem nasprotnikovega kralja. Z vidika umetne inteligence so še poseben izziv t.i. konstrukcijske naloge, ki so lahko brez šahovskega diagrama in tipično vsebujejo samo nalogo, npr. konstruirajte igro z določenimi lastnostmi (npr. najhitrejši mat s promocijo v lovca ali skakača). Te naloge namreč zahtevajo premagovanje izjemne kombinatorične kompleksnosti. Hkrati so tovrstni problemi zanimivi za širšo javnost, saj zahtevajo le poznavanje osnovnih šahovskih pravil in lahko predstavljajo privlačen itziv za reševalce. Razviti računalniški program naj temelji na hevrističnem preiskovanju. Razvijte in eksperimentalno ovrednotite vsaj dve različni hevristiki, ki naj služita usmerjanju preiskovanja k danim ciljem. Z dodatnimi mehanizmi za premagovanje kombinatorične kompleksnosti poskrbite, da bo program lahko odkrival zanimive šahovske probleme tudi na osebnih računalnikih.

# Izjava o avtorstvu diplomskega dela

Spodaj podpisani Mitja Rizvič sem avtor diplomskega dela z naslovom:

*Avtomatsko odkrivanje zanimivih šahovskih problemov* (angl. *Automatic discovery of interesting chess compositions*)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Mateja Guida,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 7. septembra 2016 Podpis avtorja:

# Kazalo

# Seznam uporabljenih kratic

| **kratica** | **angleško** | **slovensko** |
|---|---|---|
| BFS | Breadth-first search | Iskanje v širino |
| DFS | Depth-first search | Iskanje v globino |
| A* | A-star search algorithm | Preiskovalni algoritem A zvezdica |
| PGN | Portable Game Notation | Prenosljivi zapis igre |
| FEN | Forsyth Edwards Notation | Forsyth-Edwardsov zapis |

# Povzetek

Cilj diplomske naloge je bil razviti računalniški program, s katerim bi si ljudje lahko pomagali pri reševanju in ustvarjanju zanimivih šahovskih problemov. Šahovski problem je uganka na šahovnici, ki reševalcu predstavlja neko nalogo. Tovrstna naloga običajno zahteva premikanje figur na šahovnici z uporabo klasičnih šahovskih pravil in ni nujno povezana z matiranjem nasprotnikovega kralja. Poznamo več vrst šahovskih problemov: probleme direktnega mata, pomožne mate, samomate, serijske probleme, šahovske študije, retrogradno analizo itd. Reševalcu lahko postavimo dodatne zahteve, kot so na primer predpisana zadnja poteza ali pa figura, s katero matiramo. Poseben tip šahovskih problemov so konstrukcijske naloge, ki so lahko brez diagrama in tipično vsebujejo samo nalogo, npr. postavite določeno pozicijo ali konstruirajte igro z določenimi lastnostmi. Primer zanimive konstrukcijske naloge bi bil sestaviti najkrajšo šahovsko partijo, ki se konča z matiranjem nasprotnega kralja s kmetom, ki je ravno promoviral v skakača. Tovrstni problemi so lahko zanimivi ne le za šahiste, ampak tudi za širšo javnost, saj zahtevajo le poznavanje osnovnih šahovskih pravil.

Igra šaha je kombinatorično zelo zahtevna, kar pomeni, da imamo pri vsakem premiku figure na voljo veliko različnih možnosti. Posledično je pri več zaporednih potezah število možnih kombinacij premikov lahko ogromno. Za ilustracijo: iz začetne šahovske pozicije lahko le štiripotezno partijo (štiri poteze belega in štiri poteze črnega) odigramo na skoraj 85 milijard različnih načinov. Ravno ta kombinatorična eksplozija je pomemben razlog, da je reševanje šahovskih problemov in še zlasti konstrukcijskih nalog lahko iz-

jemno težavno ne le za človeka, ampak tudi za sodobne računalnike. Pri slednjih se lahko dokaj enostavno prepričamo, da samo preiskovanje s t. i. “surovo silo“ ne bo obrodilo sadov. Potreben je pametnejši pristop oz. vpeljava algoritmov umetne inteligence.

V okviru diplomske naloge smo razvili računalniški program, s katerim je mogoče računalnik usposobiti za premagovanje omenjene kombinatorične kompleksnosti ter ga uporabiti za reševanje različnih neobičajnih šahovskih problemov, še zlasti konstrukcijskih nalog. Program temelji na uporabi hevrističnega preiskovanja in namenskih hevristik, katerih naloga je preiskovanje čim bolj učinkovito usmerjati k danemu cilju. Za spopadanje s kombinatorično eksplozijo smo uporabili dodatne mehanizme, kot so zgoščevalne tabele, možnost omejevanja aktivnosti figur in iskanje več rešitev hkrati. Poleg reševanja že znanih problemov pa program, ki je v odprtokodni različici dostopen na spletu in torej na voljo širši javnosti, lahko služi tudi kot pripomoček za iskanje novih zanimivih šahovskih problemov.

**Ključne besedee**

*reševanje problemov, šah, šahovski problem, hitri mat, konstrukcijska naloga, preiskovalni algoritmi, hevristično preiskovanje, algoritem A*, hevristika, kombinatorična zahtevnost*

# Abstract

The aim of the thesis was to develop a computer program to help creating and dealing with interesting chess problems. Chess problem, also called a chess composition, is a puzzle on a chessboard, which represents a task to be solved. Task usually requires moving the chess pieces on the chessboard using classic chess rules and is not necessarily related to checkmating the opponent. We know several types of chess problems: directmates, helpmates, selfmates, serialmovers, chess studies, retrograde analysis etc. We can set additional requirements to a player, for instance prescribed last move of the figure to checkmate the opponent. Special type of chess problems are construction tasks, that may be without diagram and typically contain just a task such as "set a certain position" or "construct a game with certain properties". For example, an interesting construction task would be to construct the shortest game of chess ending with checkmating opponent with a pawn, which was just promoted to a knight. Such problems may be of an interest not only for chess players, but also for the general public, as they require only knowledge of basic chess rules.

Chess is a very demanding game in terms of complexity, which means that many different options are available for every chess piece movement. Consequently, in successive moves number of possible combinations can be enormous. For illustration: from starting chess position we can finish a four-move game (four moves by each player) in almost 85 billion different combinations. This combinatorial explosion is the major reason that chess problems and especially construction tasks can be extremely difficult - not

only for humans but also for modern computers. We can easily verify with computer that brute force search will not bring results. Smarter approach respectively the introduction of artificial intelligence algorithms is needed.

Within the thesis we have developed a computer program by which a computer can overcome this combinatorial complexity and can be used to solve a variety of unusual chess problems, particularly construction tasks. The program is based on the use of heuristic search and advanced heuristics, whose task is to guide the search towards a given goal. We used additional mechanisms, such as hash tables, possibility to deactivate chess pieces and search for several solutions at the same time, to cope with the combinatorial explosion. In addition to solving of already known problems, the program, which is available online in the open source version and therefore to the general public, can also serve as a tool to discover new interesting chess compositions.

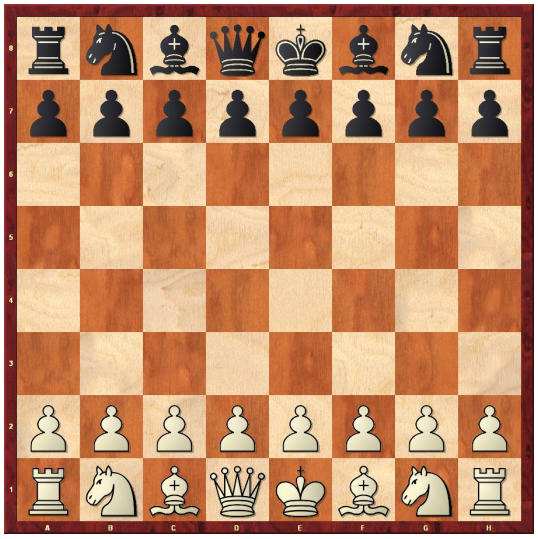
**Keywords**

*problem solving, chess, chess composition, quickmate, construction task, search algorithms, heuristic search, A* algorithm, heuristic, combinatorial complexity*

# Poglavje 1

# Uvod

Slika 1.1 prikazuje začetno pozicijo na šahovnici, besedilo pod njo pa problem, ki je bil leta 1999 objavljen na spletni strani ChessBase [4]. Potrebno je bilo



Slika 1.1: Beli igra potezo 1.e4. Kako v peti potezi skakač vzame trdnjavo in s tem matira nasprotnika?

najti le ustrezno zaporedje legalnih potez, ki ustreza zgornjemu navodilu. Iz definicije problema nam ni znano, s katerim skakačem vzamemo katero trdnjavo niti katerega kralja je potrebno matirati. Izkazalo se je, da je to izjemno

zahtevno. Vrsta problemov je postala priljubljena ne le med šahisti, temveč tudi v širši šahovski javnosti, saj za reševanje ne zahteva veliko šahovskega predznanja. Tako vsako leto na spletni strani objavijo nove probleme te vrste pod naslovom "ChessBase Christmas Puzzle" [5].

Na tem mestu predstavimo še tri sorodne šahovske probleme, ki nam jih je uspelo odkriti s pomočjo razvitega računalniškega programa. V vseh treh primerih začnemo iz začetne pozicije, navodila pa so naslednja:

- v peti potezi kmet promovira v skakača in matira nasprotnega kralja;
- v peti potezi kmet promovira v lovca in matira nasprotnega kralja;
- v šesti potezi eden od skakačev "zamenja barvo" in matira nasprotnega kralja.

Pri tretjem problemu skakač zamenja barvo tako, da kmet vzame skakača nasprotne barve in v isti potezi promovira v skakača svoje barve. S to potezo pa mora zadati tudi mat nasprotnemu kralju.

## 1.1 Vrste šahovskih problemov

Obstaja več vrst šahovskih problemov. Navedimo nekatere najbolj znane [2]:

- **direktni mati** (ang. *directmates*) – beli mora matirati črnega v določenem številu potez ne glede na igro črnega;
- **pomožni mati** (ang. *helpmates*) – problemi, pri katerih obe strani sodelujeta pri matiranju kralja;
- **samomati** (ang. *selfmates*) – beli mora s svojo igro prisiliti črnega, da ga matira, pri čemer se črni poskuša temu upreti;
- **serijski problemi** (ang. *seriesmovers*) – pri tem tipu problemov igra samo ena stran z namenom, da doseže predpisani cilj. Vsi zgornji problemi so lahko serijski;

- **šahovske študije** (ang. *studies*) – naloge, pri katerih beli začne igro in na koncu zmaga ali remizira ne glede na igro nasprotnika. Za to vrsto problemov obstaja tudi računalniški program *Chesthetica*, ki ga je razvil prof. Azlan Iqbal [6]. Prvotno je bil namenjen ocenjevanju estetike šahovske igre, kar je za računalnik zelo težko, saj mora znati oceniti, kaj je estetsko za človeka. Za ta namen so razvili tudi posebne algoritme [23] [24]. Kasneje so program nadgradili, da je znal poleg ocenjevanja tudi komponirati probleme oz. šahovske študije [12];

- **retrogradna analiza** (ang. *retrograde analysis problems*) – pri tej vrsti problemov je tipično podana šahovska pozicija in vprašanje, kako smo prišli do nje. Od reševalca je zahtevano, da poišče potezo oz. poteze, ki so privedle do nje. S to vrsto problemov se ukvarjata knjigi "Šahovske skrivnosti Sherlocka Holmesa" in "Šahirazada" avtorja Raymonda M. Smullyana [27] [28];

- **konstrukcijske naloge** (ang. *construction tasks*) – problemi, pri katerih mora reševalec konstruirati šahovsko pozicijo ali igro z določenimi lastnostmi. Znan problem takega tipa je problem osmih dam [7];

- **rekonstrukcijske naloge** (ang. *reconstruction tasks)* – podana je končna pozicija ter zgolj pomanjkljive informacije glede zaporedja premikov, ki pripelje do nje. Npr. premiki so lahko podani v koordinatnem zapisu, kar posledično pomeni, da figure, ki izvedejo premike niso znane. Naloga je določiti začetno pozicijo [3] [1].

V diplomskem delu se bomo osredotočili še zlasti na dve vrsti problemov; in sicer na konstrukcijske naloge in pomožne mate [17].

Kot omenjeno gre pri pomožnih matih za probleme, kjer obe strani sodelujeta oziroma si pomagata pri matiranju kralja. Pomožni mat označujemo s črko $h$ in številom potez. S takim zapisom smo omejeni le na cele poteze. Zapis *h4* namreč pomeni, da iščemo mat v štirih potezah oziroma v osmih polpotezah. Ena poteza pri šahu predstavlja premik belega in črnega,

medtem ko je polpoteza premik posameznega igralca. Prav tako z omenjenim zapisom ne moramo postaviti kompleksnejših pogojev, zato ciljni pogoj zapišemo z algebraično notacijo [16]:

*"**4.Qxf7#**" predstavlja mat v četrti potezi, ki ga zada bela dama s premikom na polje f7, pri čemer vzame nasprotnikovo figuro.*

Pri problemih, na katere smo osredotočeni v tem delu, lahko poleg številke poteze v ciljni pogoj vključimo dodatne zahteve, ki jih moramo pri reševanju upoštevati. Prav tako nismo omejeni le na cele poteze, temveč lahko iščemo mat belega ali pa črnega kralja. Več o uporabi algebraične notacije je napisano v poglavju 2.1.

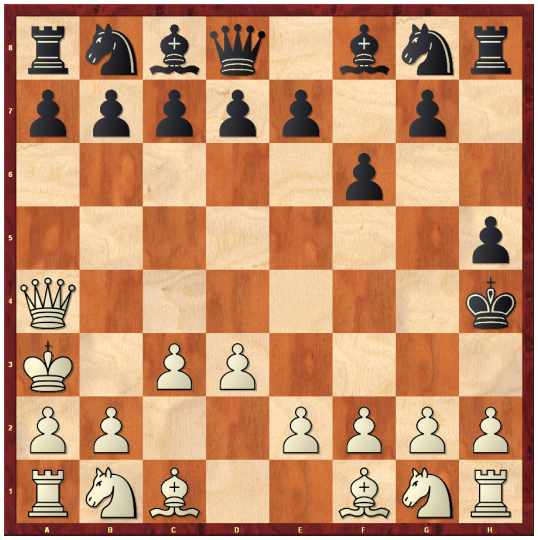
## 1.2 Iskanje hitrih matov

Z uporabo algebraičnega zapisa lahko definiramo tudi hitre mate (ang. *quickmates*). Z njimi se je prvi začel ukvarjati Stuart Rachels [15], ki jih je izumil in določil z naslednjo definicijo:

*Naloga je sestaviti najkrajšo možno partijo, ki se konča s poljubno potezo zapisano v algebraični notaciji.*

Pri tem velja poudariti, da to – če se v zadnji potezi na isto polje lahko premakneta dve enaki figuri (npr: Rad1 - premik iz a-linije na d1, Red1 - premik iz e-linije na d1) ne predstavlja dveh ločenih problemov, ampak se obravnava kot eden (Rd1 - premik na d1). Z upoštevanjem zgornje definicije lahko z uporabo algebraične notacije predstavimo vsaj 1456 različnih hitrih matov (tabela 1.1). Do tega števila pridemo tako, da za vsako polje na šahovnici izračunamo, koliko matov je na njem mogoče predstaviti z algebraično notacijo. Npr. vse mate s figurami lahko izračunamo na naslednji način:

$$640 = 64(\texttt{polj}) * 5(\texttt{figur}) * 2(\texttt{barvi figur})$$



Slika 1.2: Slika prikazuje končno pozicijo problema, kjer beli v sedmi potezi matira nasprotnika s premikom kralja s polja b4 na a3.

| Tip mata | Št različnih matov |
|---|---|
| S figurami brez jemanja | 640 |
| S figurami z jemanjem | 640 |
| S promocijami brez jemanja | 64 |
| S promocijami z jemanjem | 112 |

Tabela 1.1: Tabela predstavlja število različnih tipov hitrih matov.

Pet figur, s katerimi lahko matiramo, zajema vse figure razen kmetov. Torej vključno s kraljem, saj s premikom le-tega lahko prav tako matiramo nasprotnika. Primer takega mata je prikazan na sliki 1.2.

Problem pri hitrih matih je z gotovostjo vedeti, če je rešitev, ki jo najdemo, res najkrajša. Da bi z gotovostjo trdili, da do rešitve ne moremo priti v manj potezah, bi lahko npr. preverili vse smiselne kombinacije premikov figur. Ker pa imamo pri vsaki potezi na voljo veliko različnih premikov, število možnih načinov za odigranje igre in s tem število premikov, ki jih moramo preveriti, zelo hitro narašča. Če izhajamo iz začetne šahovske postavitve, ima beli na voljo 20 različnih premikov. Po dva za vsakega kmeta

in po dva za vsakega skakača. Tudi črni ima v prvi potezi na voljo enako število premikov. Posledično je prvo potezo (premik belega in črnega) mogoče odigrati na 400 različnih načinov. Število zaradi velike kombinatorične zahtevnosti zelo hitro narašča in število različnih načinov, ki jih moramo preveriti, postane tako veliko, da jih ročno tako rekoč ni mogoče pregledati.

Tu postane zelo zanimiv pristop z računalniškim programom. Računalnik namreč lahko za razliko od človeka v eni sekundi preveri več sto tisoč ali pa celo milijonov različnih potez, kar močno skrajša čas preiskovanja. Vendar pa je, kot vidimo iz tabele 1.2, kombinatorična zahtevnost šaha tako velika, da niti najsodobnejši računalniki niso kos številu različnih načinov, na katere lahko odigramo igro, ko se število potez povečuje. Francois Labelle je razvil program, s katerim je poskušal poiskati rešitve za probleme, ki so določeni z zadnjo potezo, med katere spadajo tudi hitri mati. S programom je preiskal 12 polpotez (6 potez) in našel 733 hitrih matov [9].

Osrednji cilj diplomskega dela je bil razviti računalniški program, s katerim bi si uporabnik lahko pomagal pri reševanju nekaterih šahovskih problemov. Namen programa ni izčrpno preiskovanje vseh možnih kombinacij premikov, ampak hitro iskanje, saj je mišljen kot pripomoček pri reševanju ali pa odkrivanju šahovskih problemov. Za ta namen smo v programu uporabili algoritem A*, ki se od požrešnega algoritma razlikuje po tem, da upošteva hevristično znanje, ki ga imamo na voljo. To znanje algoritem usmerja k preikovanju tistih potez, ki bolj verjetno vodijo do rešitve. Tak pristop zmanjša število premikov, ki jih moramo preveriti, preden najdemo rešitev, kar posledično pomeni hitrejše preiskovanje ter možnost iskanja daljših rešitev kot je to mogoče z uporabo požrešnega algoritma.

Algoritem A* se je že izkazal kot zelo uporaben za hitro iskanje matov. V magistrskem delu *Razvoj programa za igranje 1-2-3 šsaha* je omenjeni algoritem uporabljen za iskanje matov pri 1-2-3 šahu (ang. *Progressive chess*) [25] [26]. Gre za vrsto šaha, pri kateri se namesto da bi vsak igralec odigral eno potezo, sekvenca zaporednih potez povečuje. Tako beli začne z eno potezo, črni odgovori z dvema, nato beli igra tri poteze itd. Čeprav

je igra bistveno drugačna od klasičnega šaha, so principi iskanja mata z računalnikom podobni in smo si z njimi pomagali pri razvoju našega programa.

## 1.3 Pregled diplomskega dela

V poglavju 2.1 je najprej predstavljenih nekaj zapisov za predstavitev šahovske igre. Nekateri od njih so namenjeni za predstavitev igre človeku, drugi računalniku. Poznavanje zapisov je pomembno pri definiciji šahovskih problemov ter implementaciji računalniškega programa za njihovo reševanje. V poglavju 2.2 je nato predstavljenih nekaj preiskovalnih algoritmov, ki so primerni za implementacijo preiskovanja s poudarkom na algoritmu A*, ki je bil uporabljen v programu. Prav tako sta predstavljeni dve hevristiki, ki smo ju v programu uporabili. V poglavju 3 je opisan razvit program tako s programerskega kot uporabniškega vidika. V poglavju 4 so predstavljeni rezultati, ki smo jih pridobili s testiranjem programa na različnih primerih. V poglavju 5 smo podali zaključke, v dodatku pa tabelo najdenih hitrih matov in prispevek, ki smo ga objavili v reviji Šahovska misel.

| Št. polpoteze | Št raličnih iger |
|---|---|
| 1 | 20 |
| 2 | 400 |
| 3 | 8902 |
| 4 | 197281 |
| 5 | 4865609 |
| 6 | 119060324 |
| 7 | 3195901860 |
| 8 | 84998978956 |
| 9 | 2439530234167 |
| 10 | 69352859712417 |
| 11 | 2097651003696806 |
| 12 | 62854969236701747 |
| 13 | 1981066775000396239 |
| 14 | 61885021521585529237 |

Tabela 1.2: Tabela prikazuje, kako z večanjem števila polpotez narašča število možnih načinov, na katere lahko igro odigramo [14].

# Poglavje 2

# Metode

## 2.1 Zapis šahovske igre

### 2.1.1 Zapisovanje potez

Najbolj razširjen zapis, ki se danes uporablja za zapis premikov pri šahu, je *algebraični zapis*. Njegov namen je predstaviti premike figur na človeku razumljiv način.

Za opis polj na šahovnici se uporablja kombinacije črk in številk. Vsak stolpec polj je definiran s svojo črko od a do h, vsaka vrstica pa s svojo številko od 1 do 8. Poljubno polje na šahovnici tako lahko predstavimo kot presečišče stolpca in vrstice. Če za primer pogledamo začetno postavitev figur (slika 1.1), se bela dama nahaja na polju d1. Za označevanje figur se uporabljajo črke, kot je to razvidno iz tabele 2.2. Zapis premika je sestavljen iz kombinacije figure in ciljnega polja. Poleg klasičnih premikov je potrebno paziti tudi na posebne primere: jemanja, promocije, "en passant" ter rokada. Za jemanje se pred ciljno polje vstavi črka x. Pri promocijah je potrebno zapisati, v katero figuro promoviramo kmeta, kar naredimo z znakom = in oznako figure. Pri "en passant" premiku je tako kot pri ostalih premikih s kmeti potrebno označiti, katerega kmeta premaknemo. Rokada pa ima posebno oznako; in sicer 0-0 za malo rokado ter 0-0-0 za veliko rokado. V

tabeli 2.1 je prikazanih nekaj primerov premikov, zapisanih v algebraični notaciji.

| Algebraični zapis | Opis premika |
|---|---|
| Be5 | lovec na e5 |
| Nf3 | skakač na f3 |
| Nxf3 | skakač na f3 z jemanjem |
| a4 | kmet na a4 |
| exd6 | kmet iz linije e na d6 z jemanjem |
| Nd1+ | skakač na d1 šah |
| Nd1# | skakač na d1 mat |
| 0-0 | mala rokada |
| 0-0-0 | velika rokada |

Tabela 2.1: Primeri premikov v algebraičnem zapisu.

| Ime figure | Angleška oznaka | Slovenska oznaka |
|---|---|---|
| kralj | K | K |
| dama | Q | D |
| trdnjava | R | T |
| skakač | N | S |
| lovec | B | L |

Tabela 2.2: Oznake v angleškem in slovenskem jeziku, ki se uporabljajo za označevanje figur pri algebraičnem zapisu.

### 2.1.2 Zapisovanje partij

Z algebraičnim zapisom zapišemo posamezne poteze. Za zapis celotne partije pa uporabimo druge vrste zapisa, med katerimi je najbolj razširjen zapis PGN (ang. *Portable Game Notation*) [13].

Zapis poteze se začne s številko poteze, ki ji sledi pika, če je na potezi beli igralec, oziroma tri pike, če je na potezi črni igralec. Temu sledi še zapis premika v standardni algebraični notaciji. Tako dobimo zapis, ki je razumljiv človeku, poleg tega pa je primeren za uporabo z računalniškim programom.

### 2.1.3 Zapisovanje pozicij

Tako PGN zapis kot algebraična notacija sta namenjena predstavitvi igre z zaporedjem premikov od začetka do konca. Poleg tega poznamo tudi zapise, ki so namenjeni predstavitvi trenutnega stanja igre, npr. zapis FEN [8].

Primer zapisa FEN za začetno postavitev, ki je prikazana na sliki 1.1:

*rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1*

Zapis vsebuje 6 polj.

- Postavitev figur na šahovnici. Zapis se začne z osmo vrstico s poljem *a8*. Za vsako polje v vrstici z oznako figure, vzeto iz standardne algebraične notacije, označimo, katera figura se na njem nahaja. V primeru praznih polj to označimo s številko, ki predstavlja število takih polj med dvema figurama. Celotno šahovnico predstavimo vrstico za vrstico ločenimi z znakom /.
- Barva na potezi - W/B.
- Možnosti rokade. Če rokada ni možna, se uporabi znak - sicer pa se zapiše, katere rokade so še mogoče K,Q,k,q. Npr. K pomeni, da lahko beli rokira na kraljevo stran.
- "En passant". Če poteza ni možna, se uporabi znak - sicer pa zapis polja v algebraični notaciji.
- Števec polpotez od zadnjega jemanja oziroma premika kmeta.
- Števec potez, ki se začne z 1 in se poveča po vsakem premiku črnega.

### 2.1.4 Podatkovne strukture za predstavitev šahovske igre

Da bi z računalniškim programom lahko reševali šahovske probleme, moramo igro šaha najprej predstaviti na računalniku. Preiskovalni algoritmi, ki jih obravnavamo v diplomskem delu, delujejo nad grafi, zato je potrebno šahovsko igro predstaviti kot graf oziroma drevo. Drevo je posebna vrsta grafa, ki ima v izhodišču koren, ki v našem primeru predstavlja začetno stanje igre. Z vsakim premikom, ki ga lahko naredimo iz začetnega stanja, dobimo novo stanje igre in s tem novo vozlišče drevesa, ki je naslednjik korena. Če predpostavimo, da igro začnemo iz standardne postavitve, imamo na voljo 20 različnih premikov. Dva različna premika lahko naredimo z vsakim kmetom in dva z vsakim skakačem. Posledično imamo po enem premiku belega možnih 20 različnih stanj igre. Črni lahko belemu odgovori prav tako na 20 različnih načinov. Tako dobimo po eni celi potezi 400 različnih stanj igre.

Če tako nadaljujemo do konca igre, dobimo drevo, ki vsebuje vse možne pozicije v šahovki partiji. Tako drevo imenujemo *drevo igre* (ang. *game tree*) in vsebuje približno $10^{46}$ vozlišč [18]. Tako velikega drevesa ni mogoče predstaviti na računalniku, zato preiskovalni algoritmi preiščejo le majhen del. Del drevesa, ki ga med preiskovanjem izgradi preiskovalni algoritem, imenujemo *iskalno drevo* (ang. *search tree*) in je bistveno manjše od celotnega drevesa igre.

Vsako vozlišče v drevesu predstavlja stanje igre (šahovsko pozicijo), kar pomeni, da moramo v vsakem vozlišču poznati:

- položaj figur na šahovnici,
- igralca na potezi,
- možnosti rokade,
- polje za “en passant“,
- število polpotez od zadnjega jemanja oziroma premika kmeta,

- številko poteze.

Po pregledu zgornjega seznama lahko opazimo, da vse zahtevane informacije lahko podamo z notacijo FEN, kar pomeni, da lahko vsako vozlišče drevesa opišemo z njo. Vendar pa se pri tem pojavi problem, saj je velikost pomnilnika na računalniku omejena in je posledično bolje uporabiti bolj učinkovito metodo zapisa. Eden od načinov je, da v vsakem vozlišču hranimo le premik. Vse ostale informacije pa lahko sproti po potrebi izračunamo tako, da se sprehodimo po drevesu iz korena do željenega vozlišča in tako "simuliramo" igro, ki nas pripelje do njega. Na ta način lahko prihranimo pri porabi pomnilnika, vendar v zameno za nekaj procesorskega časa, ki ga program porabi za izračun zahtevanih informacij.

## 2.2 Preiskovalni algoritmi

Problem, ki ga želimo reševati, nam določi drevo igre in cilj. Kako bomo prišli od korena drevesa do cilja, pa je naloga preiskovalnega algoritma. Programi za igranje šaha navadno uporabljajo algoritem *minimaks* (ang. *minimax*). Gre za algoritem, namenjen igram z dvema igralcema. Temelji na dejstvu, da kar je dobro za enega igralca, mora biti slabo za drugega [21]. Tak pristop je dober pri klasični igri šaha, kjer vsak igralec igra zase in poteze izbira tako, da vodijo k njegovi zmagi. Pri problemih, ki smo jih omenili v uvodu, pa imata oba igralca skupni cilj, zato se morajo poteze izbirati tako, da so dobre za oba igralca oziroma dobre za dosego skupnega cilja.

### 2.2.1 Neinformirani preiskovalni algoritmi

Neinformirani preiskovalni algoritmi poskušajo najti pot od korena do cilja s preiskušanjem različnih akcij oziroma v našem primeru premikov. Pri tem ne upoštevajo nobenih informacij, s katerimi bi lahko iskanje usmerili proti cilju, temveč akcije izbirajo na slepo po vnaprej določenem zaporedju, dokler ne najdejo cilja.

Prvi algoritem, ki ga bomo predstavili, je preiskovanje v globino (ang. *Depth-first search*, v nadaljevanju DFS). Delovanje algoritma je prikazano na sliki 2.1. Vozlišče na vrhu je koren drevesa in predstavlja začetek preiskovanja. Če predpostavimo, da algoritem sosednja vozlišča preiskuje od leve proti desni, je na sliki s številkami označeno zaporedje vozlišč, po katerem jih program preiskuje. Kot že ime pove, algoritem graf preiskuje v globino, kar pomeni, da se vozlišča preiskujejo vzdolž posamezne veje, dokler program ne pride do cilja oziroma lista drevesa. Ko iskanje doseže list drevesa in le-ta ne predstavlja cilja, algoritem zamenja vejo preiskovanja. Pri tem se pomika navzgor po preiskani poti in izbere prvo alternativno vozlišče, ki je še na voljo [19]. Preiskovanje v globino je primerno, če:

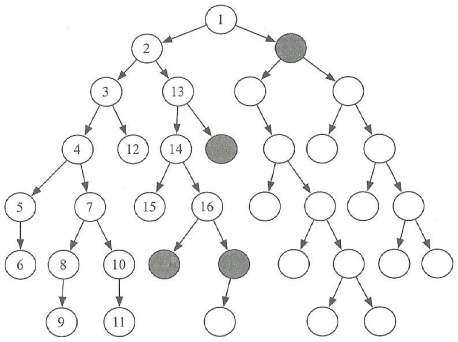
- imamo omejen prostor v pomnilniku, saj moramo v določenem trenutku hraniti le vozlišča, ki ležijo na trenutni poti;
- obstaja več rešitev, le te pa ležijo globoko v drevesu.

Preiskovanje v globino ni primerno:

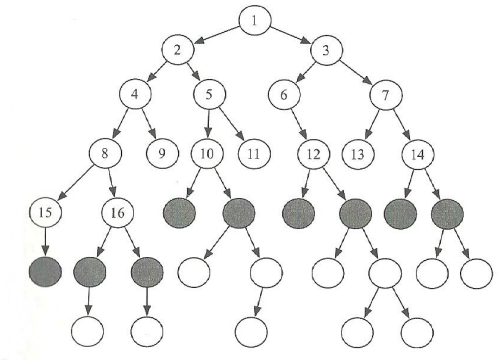
- če graf vsebuje cikle, saj se algoritem lahko ujame v neskončno zanko;
- v primeru, ko rešitve ležijo na majhni globini, saj v tem primeru lahko algoritem določeno vejo preiskuje do velike globine, čeprav je rešitev v drugi veji na majhni globini;
- ko želimo najti najkrajšo rešitev, ker lahko zaradi zgornjega razloga najprej najdemo rešitev, ki leži globje.

Pri iskanju v širino (ang. *Breadth-first search*, v nadaljevanju BFS) namesto preiskovanja posamezne veje algoritem najprej preišče vsa sosednja vozlišča, šele nato se iskanje nadaljuje proti naslednikom. Na sliki 2.2 je prikazano zaporedje, po katerem se vozlišča v drevesu preiskujejo, če predpostavimo, da se sosednja vozlišča preiskujejo od leve proti desni. Iskanje v širino je uporabno, ko:

- prostor v pomnilniku ne predstavlja ovire;




Slika 2.1: Primer iskanja v globino, kjer se sosednja vozlišča preiskujejo od leve proti desni. S številkami je označeno zaporedje, po katerem se razvijajo vozlišča, s sivo barvo pa ciljna vozlišča [19].




Slika 2.2: Primer iskanja v širino, kjer se sosednja vozlišča preiskujejo od leve proti desni. S številkami je označeno zaporedje, po katerem se razvijajo vozlišča, s sivo barvo pa ciljna vozlišča [19].

- želimo vedno najti najkrajšo rešitev;
- rešitve ležijo na manjši globini;
- kadar imamo poti neskončne dolžine ali cikle, saj se algoritem ne more ujeti v zanko.

Slabost iskanja v širino je, da ima veliko prostorsko kompleksnost, zato se ne uporablja pogosto, še posebej kadar imamo za določen problem na voljo hevristično znanje [19].

### 2.2.2 Informirani preiskovalni algoritmi

Oba algoritma opisana v prejšnjem poglavju sta neinformirana, kar pomeni, da vsa vozlišča obravnavata enakovredno. Pri iskanju torej ne upoštevata nobenih informacij o tem, kje leži cilj. Za razliko od tega informirano preiskovanje uporablja hevristično oceno za usmerjanje preiskovanja proti tistim vozliščem, ki bolj verjetno vodijo do rešitve. Hevristično oceno izračunamo s pomočjo hevristične funkcije $h(n)$, ki za vozlišče $n$ izračuna pozitivno celo število. To število predstavlja oceno poti od vozlišča $n$ do cilja.

#### Algoritem A*

Eden najbolj široko uporabljenih algoritmov v umetni inteligenci, ki uporablja hevristično znanje, je algoritem A*. Namenjen je iskanju najkrajše poti od izhodišča do cilja, ki je v našem primeru matna pozicija. Za razliko od neinformiranega preiskovanja, kjer se vozlišča razvijajo po vrsti, algoritem A* najprej razvije vozlišče $n$ z najmanjšo oceno $f(n)$:

$$f(n) = g(n) + h(n)$$

kjer $g(n)$ predstavlja dolžino poti od izhodišča do vozlišča $n$, $h(n)$ pa hevristično oceno vozlišča $n$.

Problemi, ki jih obravnavamo v diplomskem delu, imajo lastnost, da je dolžina rešitve vnaprej določena, kar je v nasprotju z idejo algoritma A*, ki vedno išče najkrajše rešitve. Posledično algoritem porabi veliko časa za preiskovanje vozlišč blizu korena, čeprav se rešitev nahaja globje v drevesu. Temu se lahko izognemo tako, da ceno poti $g(n)$ do vozlišča $n$ zanemarimo. Tako dobimo namesto algoritma A* algoritem, ki v vsakem koraku razvije vozlišče z najboljšo hevristično oceno ( ang. *Best-first search* ). Kljub naivnosti se algoritem dobro obnese in je bil uporabljen v končni verziji programa.

#### Hevristike

Implementacija hevristične funkcije je odvisna od vrste poblema, ki ga rešujemo. Zaželeno je, da je hevristična funkcija:

- optimistična, kar pomeni, da dejanska cena poti od vozlišča do cilja nikoli ni manjša od ocenjene;

- konsistentna oziroma monotona, kar pomeni, da za poljuben par vozlišč $m, n$ velja: $h(m) <= k(m,n) + h(n)$ kjer, $k(m,n)$ predstavlja najkrajšo pot od $m$ do $n$;

- hitra za izračun.

Cilj preiskovanja pri problemih, s katerimi se ukvarjamo v diplomskem delu, je najti mat. Zato je za implementacijo hevristične funkcije potrebno najprej razmisliti, kako iz danega vozlišča oziroma šahovke postavitve oceniti, koliko nam manjka do mata. Izkaže se, da je za mat potreben pogoj pokritost vseh polj okoli kralja, ki ga želimo matirati, vključno s poljem, na katerem stoji kralj. Ta polja imenujemo matni kvadrat in je prikazan na sliki 2.3. Poleg omenjenega pogoja je za mat potrebno tudi to, da figura, ki napada kralja, ne more biti vzeta ali blokirana. Vendar pa tega pogoja pri izračunu hevristične ocene navadno ne upoštevamo, saj je zahteven za izračun. Zaradi zgornjega pogoja je smiselno, da se hevristična funkcija ukvarja s tem, kako najhitreje pokriti matni kvadrat. V nadaljevanju si bomo ogledali dve hevristiki za usmerjanje programa pri iskanju matov.

Prva hevristika je *manhattanska razdalja*. Le-ta sešteje manhattansko razdaljo vseh figur brez kmetov do matnega kvadrata okoli nasprotnikovega kralja.

$$h(n) = \sum_{f \in figure} manhattanskaRazdalja(f, MatniKvadrat)$$

Hevristika je relativno enostavno izračunljiva in iskanje vodi tako, da prioritizira tiste pozicije, kjer so naše figure bližje nasprotnikovemu matnemu kvadratu. Primer izračuna za sliko 2.3:

$$h(n) = 2(\texttt{trdnjava}) + 2(\texttt{skakač})$$

Naslednja hevristika je *pokritost* ( ang. *cover* ) in šteje, koliko polj v matnem kvadratu je pokritih:

$$h(n) = \sum_{p \in MatniKvadrat} \begin{cases} 1, & \text{Polje p je pokrito} \\ 0, & \text{Polje p ni pokrito} \end{cases}$$

Polje je pokrito, ko se kralj, ki je v sredini matnega kvadrata, nanj ne more premakniti. Torej v primeru, ko je le-to zasedeno s figuro enake barve kot kralj, je zasedeno z nasprotnikovo figuro, ki je kralj ne more vzeti ali pa je z nasprotnikovo figuro napadeno. Hevristika je hitra za izračun in daje zelo dobre rezultate. Predstavljena je bila v magistrskem delu Vita Janka [25] in je bila med vsemi opisanimi hevristikami najboljša pri iskanju mata za 1-2-3 šah. V svojem delu je Vito Janko pri izračunu upošteval tudi večkratno pokritost posameznega polja, kar je pri igri 1-2-3 šah še izboljšalo njeno delovanje [26]. V našem primeru večkratno upoštevanje pokritosti ni izboljšalo preiskovanja in zaradi dodatne kompleksnosti izračuna ni bilo uporabljeno v končnem programu. Primer izračuna za sliko 2.3:

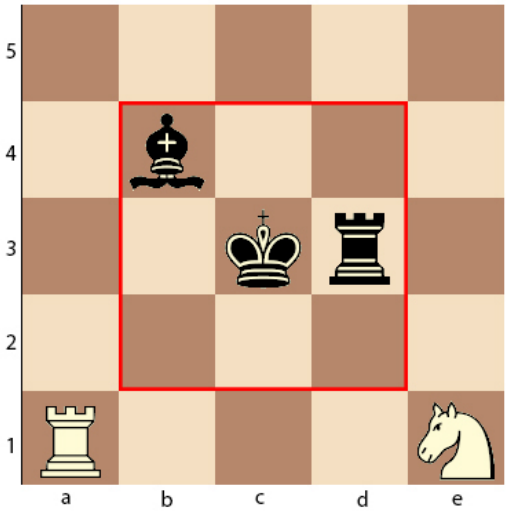
$$h(n) = 1(\texttt{b4}) + 1(\texttt{c2}) + 1(\texttt{d3})$$

## 2.3 Spopadanje s kombinatorično kompleksnostjo

### 2.3.1 Transpozicije

Pri šahu lahko do določene postavitve pridemo na več različnih načinov. Ker imamo pri iskanju matov vnaprej določeno dolžino rešitve, so za nas enake postavitve le tiste, do katerih pridemo v enakem številu potez. Za primer si poglejmo sliko 2.4. Če predpostavimo, da smo do postavitve, ki je prikazana, porabili dve potezi, potem lahko do nje pridemo na naslednje načine:

- "1. Na3 Na6 2. Nb5 Nb4".

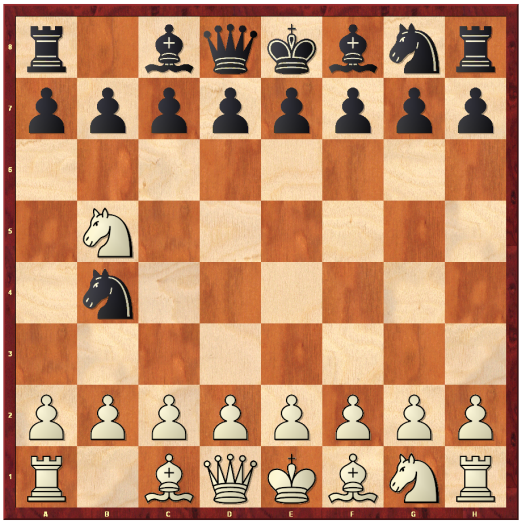



Slika 2.3: Slika predstavlja del šahovnice, na kateri je z rdečo barvo označen matni kvadrat okoli črnega kralja. Matni kvadrat vključuje vsa polja znotraj rdečega kvadrata, vključno s poljem, na katerem stoji kralj.

- "1. Na3 Nc6 2. Nb5 Nb4".
- "1. Nc3 Na6 2. Nb5 Nb4".
- "1. Nc3 Nc6 2. Nb5 Nb4".

Z vsakim od naštetih zaporedij premikov pridemo do enakega stanja igre. Temu pravimo transpozicija (ang. *transposition*). Če bi imeli na razpolago večje število potez, bi bilo takih transpozicij še več.

Transpozicije predstavljajo pomemben problem pri preiskovanju. Ker preiskovalni algoritmi ne hranijo zgodovine preiskanih vozlišč, nimajo informacije o tem, če so določeno stanje igre že preiskali ali ne. Ko pride do transpozicije, algoritem vozlišče obravnava kot novo in iskanje nadaljuje tako, da generira njegove naslednike. Še posebej pri šahu je to lahko velik problem. Ker je razvejanost drevesa zelo velika, vsako tako vozlišče pomeni ogromno število naslednikov, ki se bodo generirali, še posebej če do transpozicije pride blizu korena drevesa. In kadar nasledniki originalnega vozlišča ne vsebujejo rešitve, je ne bodo vsebovali niti nasledniki transpozicij. Če takih trans-



Slika 2.4: Slika postavitve figur, do katere lahko pridemo na več različnih načinov.

pozicij ne odstranimo, veliko procesorskega časa zapravimo za preiskovanje neplodnih vozlišč.

Za reševanje transpozicij potrebujemo način hranjenja zgodovine preiskanih vozlišč. Šahovski programi za ta namen uporabljajo transpozicijske tabele (ang. *transposition tables*). Gre za podatkovno strukturo, kamor program shranjuje že pregledana vozlišča. Vsakič ko program nato generira novo vozlišče, najprej preveri, ali je transpozicija tako, da pogleda v transpozicijsko tabelo. Če je, ga zavrže, če ni, se iskanje nadaljuje. Problema, ki se pojavita pri tem pristopu, sta dva. Število vozlišč, ki jih lahko hranimo, je odvisno od količine pomnilnika, ki ga imamo na voljo. Prav tako z večanjem prostora, ki ga dodelimo hranjenju zgodovine vozlišč, zmanjšamo količino prostora v pomnilniku, ki ga ima na voljo program za izgradnjo iskalnega drevesa. Poleg tega pa moramo za učinkovito iskanje transpozicij implementirati hitro iskanje po zgodovini vozlišč. Najbolje je uporabiti podatkovno strukturo s takojšnjim dostopom do elementov, saj lahko v nasprotnem primeru iskanje

transpozicij vzame toliko procesorskega časa, da ga ni smiselno implementirati.

### Zgoščevalne tabele

Podatkovna struktura, ki rešuje oba problema, je zgoščevalna tabela. Osnovna ideja zgoščevalne tabele je, da s pomočjo posebne funkcije, ki jo imenujemo zgoščevalna funkcija, vozlišču, ki ga želimo shraniti v tabelo, izračunamo numerično vrednost. Ta vrednost nato predstavlja naslov v tabeli oziroma v pomnilniku, kamor bomo vozlišče shranili. Ko želimo kasneje preveriti, če vozlišče že obstaja v tabeli (ali je transpozicija), enostavno izračunamo njegovo vrednost z zgoščevalno funkcijo in pogledamo na dobljen naslov v tabelo. Za izračun naslova lahko uporabimo različne zgoščevalne funkcije. Namen vseh je vsako vozlišče, ki v našem primeru predstavlja šahovsko postavitev, predstaviti z numerično vrednostjo. Zaželeno je, da so numerične predstavitve vozlič unikatne, saj lahko v nasprotnem primeru dva vozlišča, ki nista enaka, dobita isti naslov v tabeli. Če zgoščevalna funkcija ne zagotavlja unikatnih vrednosti, potem moramo – v primeru da se na izračunanem naslovu v tabeli že nahaja neko drugo vozlišče – enakost dodatno preveriti. Šele potem lahko z gotovostjo trdimo, da je obravnavano vozlišče zares transpozicija. Zgoščevalna funkcija, ki smo jo uporabili v našem programu, je *Zobrist Hashing* [29].

Namen zgoščevalne funkcije Zobrist je vsako šahovsko postavitev predstaviti s skoraj unikatnim številom pod dodatnim pogojem, da dve podobni postavitvi data čimbolj različna števila. To je pomembno zato, da se vozlišča, ki jim izračunamo numerično predstavitev, enakomerno razporejajo v zgoščevalno tabelo. Funkcija deluje tako, da ob inicializaciji generira polje naključnih števil, kjer vsako število predstavlja informacijo, ki jo želimo vključiti v izračun. Pri šahu navadno definiramo naključna števila, predstavljena v nadaljevanju.

- Po eno število za vsako kombinacijo polja in figure na njem. Npr. trdnjava na “a1“ dobi svojo naključno vrednost, trdnjava na “a2“ dobi

drugo naključno vrednost itd. Podobno se naredi za vse ostale figure.

- Število, ki predstavlja igralca na potezi - beli/črni.
- Štiri števila, ki predstavljajo pravice za rokado - po dve rokadi za vsakega igralca (mala in velika).
- Osem števil, ki predstavljajo stolpec, na katerem lahko izvedemo "en passant" premik, če obstaja možnost.

Po želji lahko v izračun vključimo tudi dodatne informacije, kot je npr. številka poteze, vendar za zaznavanje transpozicij zadošča zgornji seznam. Ideja je ta, da vsa števila, ki pripadajo lastnostim, ki v dani poziciji veljajo, med seboj združimo z operatorjem XOR. Kot rezultat dobimo število, ki je skoraj unikatno in predstavlja numerično reprezentacijo šahovske pozicije. Dobra lastnost operatorja XOR je, da sam sebi predstavlja inverz, kar pomeni, da za poljuben par X in Y velja:

$$Z = X \oplus Y \Rightarrow X = Y \oplus Z \wedge Y = X \oplus Z$$

Kot primer vzemimo začetno šahovsko postavitev, ki smo ji izračunali zobrist vrednost. Če beli naredi premik 1.a4, lahko novo zobrist vrednost izračunamo tako, da izhajamo iz prejšnje vrednosti in upoštevamo le spremembe. Ker je beli premaknil kmeta s polja a2 na a4, originalno zobrist vrednost najprej z operatorjem XOR združimo s številom, ki predstavlja kmeta na polju a2. S tem razveljavimo, da kmet stoji na polju a2. Nato je potrebno dobljeno vrednost z operatorjem XOR združiti s številom, ki predstavlja kmeta na polju a4. Ker v drevesu vozlišč, ki ga gradimo z iskanjem, vedno poznamo starševsko vozlišče, lahko zobrist vrednosti vedno računamo inkrementalno in tako prihranimo pri procesorskem času.

Ker imamo v praksi na voljo veliko manj pomnilnika, kot je potrebnega za hranjenje celotne zgodovine preiskanih vozlišč, moramo velikost tabele omejiti. To storimo tako, da vrednost, ki jo izračunamo z zgoščevalno funkcijo, delimo po modulu in s tem "velikost" naslova omejimo z velikostjo deljitelja.

Problem, ki se pri tem pojavi, je, da sedaj lahko različna vozlišča dobijo enako numerično vrednost. Zato je pri zadetku v tabeli potrebno še dokončno preveriti, če sta vozlišča zares enaka.

Če je pri shranjevanju novega vozlišča v zgoščevalno tabelo naslov, kamor ga želimo shraniti, že zaseden, potem obstajata dve možnosti.

- Vozlišča nista enaka in sta enak naslov dobila zaradi deljenja po modulu. V tem primeru se je potrebno odločiti, če v tabeli obdržimo staro vozlišče ali ga zamenjamo z novim. Tretja možnost je, da na isto mesto v tabeli shranimo več vozlišč, povezanih v seznam, vendar ta možnost ni optimalna, saj je potrebno pri preverjanju, če vozlišče že obstaja v tabeli, seznam preiskati po elementih, kar pa predstavlja potrato časa.

- Vozlišča sta enaka, kar pomeni, da je novo vozlišče transpozicija že shranjenega, zato ga lahko zavržemo.

Poznamo več scenarijev, ki se jih pri zamenjavi vozlišča v tabeli lahko držimo. Najboljši so tisti, ki upoštevajo tudi globino vozlišča, saj s tem upoštevajo tudi količino dela, vloženega v iskanje [20]. Vendar pa v našem primeru taki scenariji ne pridejo prav, saj imamo vnaprej določeno dolžino rešitve. Posledično dve šahovski postavitvi, ki sta enaki, vendar do njih pridemo v različnem številu potez, za nas ne predstavljata transpozicije. Zato smo se odločili, da v programu uporabimo scenarij, kjer staro vozlišče vedno zamenjamo z novim, saj je enostaven za implementacijo in dokaj učinkovit.

### 2.3.2 Omejevanje aktivnosti figur

Kot smo omenili že v uvodu, je kombinatorična zahtevnost šaha zelo velika, kar predstavlja težavo pri računalniškem preiskovanju. Algoritem A*, ki smo ga uporabili za preiskovanje, zahteva eksponentno mnogo prostora glede na dolžino rešitve, ki jo iščemo. Ker smo v praksi omejeni s količino pomnilnika, ki ga imamo na voljo, je potrebno količino vozlišč, ki se generirajo, nekako omejiti, saj v nasprotnem primeru ne moremo iskati daljših rešitev.

Količino generiranih vozlišč najbolj učinkovito omejimo tako, da zmanjšamo razvejanost drevesa. V našem primeru to pomeni, da je potrebno zmanjšati število različnih premikov, ki so na voljo, v vsakem koraku preiskovanja. To najlažje storimo tako, da omejimo figure, ki jih program lahko uporabi pri iskanju rešitve. Že pred začetkom preiskovanja lahko povemo programu, katere figure naj uporablja med preiskovanjem in katere mora pustiti pri miru. Slabost tega pristopa je, če je izbrana napačna kombinacija aktivnih figur, program rešitve ne bo našel, čeprav je mogoča z neko drugo kombinacijo. Kljub temu pa je pristop dober, saj lahko v večini primerov določene figure deaktiviramo brez tveganja. Na primer, če iščemo mat s promocijo belega kmeta v peti potezi, potem je med belimi figurami lahko aktiven le en kmet, saj le-ta porabi vseh pet potez, preden doseže osmo vrsto in lahko promovira. Druga slabost pa je v tem, da program potrebuje človeško pomoč pri izbiri aktivnih figur.

Obe pomanjklivosti lahko delno zaobidemo tako, da programu naročimo, naj namesto z vsemi figurami poskusi rešitev najti z različnimi podmnožicami figur. Podmnožice se lahko izbirajo avtomatsko in pri tem ne potrebujemo interakcije uporabnika. Ta pristop sicer dobro deluje za nekatere probleme, kjer je število smiselnih kombinacij figur dovolj majhno, da jih je mogoče preiskati v doglednem času. V drugih primerih, kjer je takih kombinacij preveč, pa ne preostane drugega, kot da vnaprej določimo figure, s katerimi želimo poiskati rešitev.

### 2.3.3 Iskanje več rešitev hkrati

Program, ki smo ga razvili, je zasnovan tako, da je cilj preiskovanja mogoče določiti z različnimi parametri. To nam omogoča, da iščemo rešitev za zelo specifičen problem, lahko pa ciljni pogoj sestavimo zelo ohlapno. To pride v poštev še zlasti, ko iščemo več rešitev za podobne probleme, saj lahko ciljni pogoj sestavimo tako, da zajame vse probleme, katerih rešitve so predmet preiskovanja. Na tak način lahko iščemo rešitve za več problemov istočasno. Če npr. želimo najti vse mate, ki so mogoči z različnimi figurami v določeni

potezi, lahko to storimo tako, da najprej poiščemo vse mate, ki so mogoči s kraljico, nato s trdnjavo, skakačem itd. Lahko pa enostavno iščemo vse mate, ki so mogoči v določeni potezi, in pri tem ne omejimo figure, ki mat zada. Rezultat takega preiskovanja je datoteka z vsemi rešitvami, ki jih je program našel. Če želimo iz nje razbrati rešitve za posamezne probleme, je potrebno rešitve med seboj ločiti, kar pa je veliko hitreje kot iskanje matov za vsako figuro posebej.

# Poglavje 3

# Opis programa

Program je napisan v programskem jeziku C++ in je namenjen izvajanju v linux okolju. Tako smo se odločili zaradi hitrosti in pa množice odprtokodnih generatorjev potez, ki so na voljo. V nadaljevanju je opisan generator potez, ki smo ga v programu uporabili. Sledi tudi opis programa s programerskega in uporabniškega vidika.

## 3.1 Generator potez

Program temelji na odprtokodnem generatorju potez sachista-chess, ki je prosto dostopen na spletni strani GitHub [10]. Omenjeni generator je napisan v programskem jeziku C++ in je v obliki knjižnice, ki jo lahko enostavno uporabimo v programu.

Generator potez za predstavitev šahovnice uporablja bitno polje (ang. *bit-board*), kar omogoča hitre operacije in optimalno porabo pomnilnika. Bitno polje je podatkovna struktura, ki je v našem primeru implementirana kot 64-bitna vrednost in si jo lahko predstavljamo kot polje velikosti 8 x 8, kjer vsak element predstavlja polje na šahovnici in ima lahko vrednosti 0 ali 1. S tako strukturo lahko na šahovnici predstavimo poljubne lastnosti, kot so npr: polja, na katerih se nahajajo bele figure, vse legalne premike določene figure, aktivna polja itd. Kot primer si oglejmo sliko 3.1, na kateri stoji beli

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

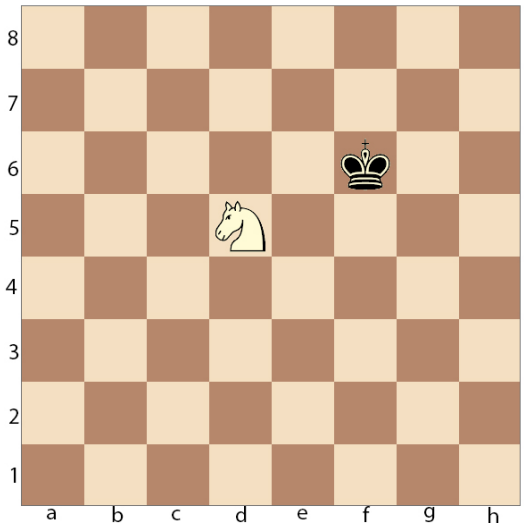
Tabela 3.1: Prikaz zasedenosti polj za sliko 3.1 z bitboard predstavitvijo.

skakač na polju d5 in črni kralj na polju f6. Če želimo zapisati, katera polja na šahovnici so zasedena, lahko to storimo, kot je prikazano v tabeli 3.1 in dobimo binarno število:

0000000000000000000010000010000000000000000000000000000000000000

Tako število lahko v programu enostavno predstavimo kot 64-bitno celo število, ki je implementirano s podatkovnim tipom "uint64". Podobno lahko storimo tudi za ostale lastnosti, ki jih želimo predstaviti: polja, na katera se lahko premakne skakač, polja, na katerih se nahajajo bele figure, napadi s skakačem itd. Operacije nad bitnimi polji so veliko hitrejše, kot če bi šahovnico implementirali s poljem (tabelo) figur. Posledično je taka implementacija generatorja hitrejša in jo uporablja večina boljših generatorjev.

Generator, ki smo ga uporabili v našem programu, omogoča generiranje psevdolegalnih potez. Psevdolegalne poteze so tiste, ki so v skladu s trenutno postavitvijo na šahovnici ter s klasičnimi šahovskimi pravili. Vse psevdolegalne poteze niso nujno tudi legalne. Generator potez namreč ne preveri, če igralec po premiku izpostavi lastnega kralja šahu oziroma šaha ne prepreči v primeru, če je bil kralj v šahu že pred premikom. Da lahko generirane poteze uporabimo za izgradnjo iskalnega drevesa, moramo preveriti še dodatna dva pogoja, kar smo naredili z implementacijo funkcije, ki za vsak psevdolegalni




Slika 3.1: Slika prikazuje postavitev figure, za katero je iz tabele 3.1 razvidno, katera polja so zasedena.

premik dodatno preveri, če je tudi legalni. Delovanje funkcije je razvidno iz algoritma 1. Funkcija za trenutno vozlišče, ki predstavlja trenutno šahovsko pozicijo, izračuna vse možne premike. S pomočjo generatorja potez se najprej generirajo vsi psevdolegalni premiki, ki se shranijo v polje oziroma tabelo. Z zanko se nato sprehodimo skozi celotno polje psevdolegalnih premikov ter za vsakega simuliramo, kakšno pozicijo bi dobili, če bi izvedli premik. Iz dobljene pozicije lahko nato razberemo, če je nasprotnikov kralj v šahu. Če je po izvedenem premiku kralj ostal v šahu ali pa je šah nastal s premikom, je premik nelegalen. V nasprotnem primeru je premik legalen, zato ga dodamo v tabelo legalnih premikov.

Zaradi velike kombinatorične zahtevnosti smo, kot smo omenili v poglavju 2.3, omogočili deaktivacijo posameznih figur. Zato moramo pri generiranju premikov preveriti, če se lahko s figuro, za katero je generator generiral premik sploh premaknemo. Deaktivacija figur deluje tako, da programu na začetku povemo, na katerih poljih se nahajajo figure, ki jih želimo vključiti v preiskovanje. Program nato sproti ažurira aktivna polja glede na to, kam se figure na njih premaknejo. Tak način implementacije je dober, saj nam

**Algoritem 1** Funkcija za generiranje legalnih potez

```
1: procedure LEGALMOVES
2:     trenutnaPozicija ← vozlisce.pozicija()
3:     psevdoLegalniPremiki[] ← generator.generirajPremike()
4:     legalniPremiki = []
5:     for premik in psevdoLegalniPremiki[] do
6:         if premik.izhodiscnoPolje ∈ aktivnaPolja then
7:             nslednjaPozicija ← trenutnaPozicija.narediPotezo(premik)
8:             if naslednjaPozicija.nasprotnikovKraljNiVSahu() then
9:                 legalniPremiki[]+ = premik
       return legalniPremiki
```

omogoča, da aktivna polja hranimo kot bitno polje, kar je priročno, saj so tudi premiki, ki jih generira generator, predstavljeni z bitnim poljem. Tako lahko ob vsakem opravljenem premiku aktivna polja posodobimo z enostavnim izračunom; in sicer:

$$\texttt{(bitboard aktivnih polj)} \wedge \neg\texttt{(bitboard izhodiščnega polja premika)} \vee \texttt{(bitboard ciljnega polja premika)} \quad (3.1)$$

Najprej deaktiviramo polje, na katerem je stala figura pred premikom. To naredimo z operacijo AND med bitboard predstavitvijo aktivnih polj in inverzom bitboard predstavitve izhodiščnega polja premika (število z ničlo, kjer je stala figura pred premikom in enicami drugje). Nato z operacijo OR med dobljeno vrednostjo in bitboard predstavitvijo ciljnega polja premika (število s samimi ničlami in enico, ki predstavlja ciljno polje) dobimo posodobljeno vrednost, ki predstavlja nova aktivna polja. Preostane nam le še to, da za vsak premik, ki ga generira generator potez preverimo, če je veljaven; torej če je izhodiščno polje premika vsebovano med aktivnimi polji. Kot je razvidno iz algoritma 1, je le-to implementirano v isti funkciji kot preverjanje legalnosti premika.

## 3.2 Implementacija preiskovanja

Kot smo omenili v poglavju 2.2 program za preiskovanje uporablja algoritem A*. Program je namenjen uporabi preko ukazne vrstice. Ob zagonu se prebere konfiguracijska datoteka, ki smo jo podali na vhod. Če med tolmačenjem datoteke ni prišlo do napake, program vstopi v glavno zanko, ki jo izvaja, dokler ne najde rešitve ali pa ne preišče vseh možnih potez do globine, ki je nastavljena v konfiguracijski datoteki. Shematski prikaz delovanja programa je razviden iz algoritma 2.

**Algoritem 2** Glavna zanka programa - Algoritem A*

```
1: vrsta ← Prioritetna vrsta urejena po f ocenah vozlišč
2: preiskana ← Seznam že preiskanih vozlišč
3: while vrsta ni prazna do
4:     vozlisce ← Prvi element iz vrste
5:     if vozlisce ==resitev then
6:         Vrni pot do vozlišča
7:     legalniPremiki ← Seznam legalnih premikov, glej algoritem 1
8:     while seznam legalnih premikov ni prazen do
9:         premik ← Prvi element iz seznama legalnih premikov
10:        novoVozlisce ← vozlišče.naredi Potezo(premik)
11:        if novoVozlisce == transpozicija then
12:            Zavrži vozlišče in nadaljuj od začetka zanke
13:        vrsta ← Dodaj novo vozlišče v vrsto
14:        tabelaTranspozicij ← Dodaj novo vozlišče v tabelo transpozicij
15:    preiskana ← Dodaj obdelano vozlišče v seznam preiskanih
```

Pred vstopom v zanko se v vrsti nahaja začetno vozlišče, ki predstavlja začetno postavitev na šahovnici. Vsako vozlišče vsebuje kazalec na starša, premik, s katerim pridemo od starša do njega in hevristično oceno. Vse ostale lastnosti lahko po potrebi izračunamo, kar nam omogoča prihranek pomnil-

nika v zameno za nekaj procesorskega časa, ki ga porabimo pri izračunu.

V vsaki iteraciji zanke program iz vrste vzame prvo vozlišče in preveri, če je vozlišče končno; torej če smo prišli do želenega cilja. Ker je vrsta po definiciji urejena podatkovna struktura, v našem primeru po naraščujočih hevrističnih ocenah, je prvo vozlišče v vrsti vedno tisto z najmanjšo oceno. Če vozlišče predstavlja rešitev, se zanka zaključi in program izpiše rezultat. V nasprotnem primeru pa se – če globina vozlišča ne presega omejitve – generirajo vsi nasledniki vozlišča. Pri generiranju vozlišča program najprej preveri, če je novo vozlišče transpozicija in ga – če je to res – zavrže. V nasprotnem primeru program izračuna njegovo hevristično oceno in ga na podlagi le-te doda na pravo mesto v vrsto. Poleg tega se generirano vozlišče doda v tabelo transpozicij, tako da lahko v prihodnje, ko pridemo do iste pozicije, to tudi zaznamo. Tabela transpozicij je implementirana z zgoščevalno tabelo in zgoščevalno funkcijo zobrist tako, kot je predstavljeno v poglavju 2.3. Na koncu zanke obdelano vozlišče dodamo v seznam pregledanih, saj ga bomo, če leži na poti do rešitve, še potrebovali.

## 3.3 Uporabniški vmesnik

Program, ki smo ga razvili, uporabniku omogoča reševanje različnih med seboj povezanih šahovskih problemov in tudi iskanje novih. Zasnovan je tako, da omogoča nastavljanje parametrov iskanja, zato lahko z njim rešujemo različne tipe problemov. Program je namenjen izvajanju v ukazni vrstici in vse parametre prejme preko konfiguracijske datoteke. Uporabnik lahko v to datoteko zapiše parametre, predstavljene v nadaljevanju.

- SEARCHALL – Naročimo programu, če naj po prvi najdeni rešitvi zaključi iskanje ali naj poskuša najti še kakšno rešitev.
- PROBLEM – Izberemo, če naj program išče najkrajšo pot ali pot fiksne dolžine. V primeru poti fiksne dolžine parameter MOVES predstavlja dolžino le-te, v nasprotnem primeru pa predstavlja omejitev globine preiskovanja.

- MOVES – Število potez, ki naj jih preišče program.
- WINCOLOR – Barva figure, ki zaključi igro (matira nasprotnika).
- H – Izbira hevristike. Na voljo za izbiro so naslednje možnosti: brez hevristike, cover, manhatenska razdalja ter kombinacija obeh. Poleg omenjenih hevristik pa program ponuja še prilagojeno hevristiko manhattan za iskanje pozicije.
- START – FEN zapis začetne pozicije na šahovnici. Če je ne določimo, je privzeta vrednost standardna začetna postavitev.
- TRANSP_TABLE – Velikost transpozicijske tabele.
- END – FEN zapis iskane končne pozicije na šahovnici.
- STARTMOVE – Določimo prvi premik. Torej premik, s katerim začnemo igro.
- ENDMOVE – Določimo zadnji premik. Torej premik, s katerim zaključimo igro.
- ENDPIECE – Določimo figuro, ki naredi zadnji premik.
- CAPTURE – Določimo, če je zadnja poteza jemanje ali ne.
- CAPTURED – Figura, ki jo v zadnji potezi vzamemo.
- PROMOTIONPIECE – Figura, v katero v zanji potezi promoviramo.
- FMOVENUMBER, FMOVE, FMOVEPIECE – S temi parametri lahko po želji določimo katerikoli vmesni premik.
- ACTIVE – Določimo aktivna polja.

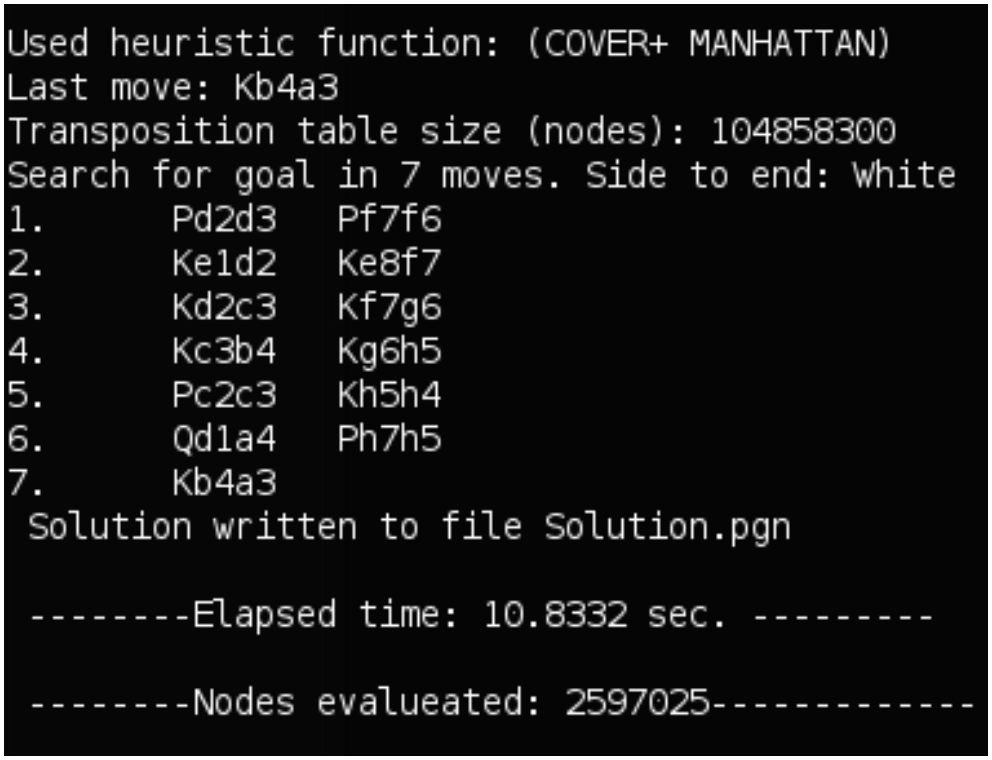
Katere parametre bo uporabnik uporabil, je odvisno od problema, ki ga želi reševati. Poleg parametrov, s katerimi opišemo vrsto problema, so med zgornjimi tudi taki, s katerimi zmanjšamo kompleksnost preiskovanja.

Najpomembnejši med njimi je parameter ACTIVE, s pomočjo katerega aktiviramo le figure, ki jih želimo vključiti v iskanje. Tako zmanjšamo število generiranih potez, kar je nujno za iskanje daljših rešitev.

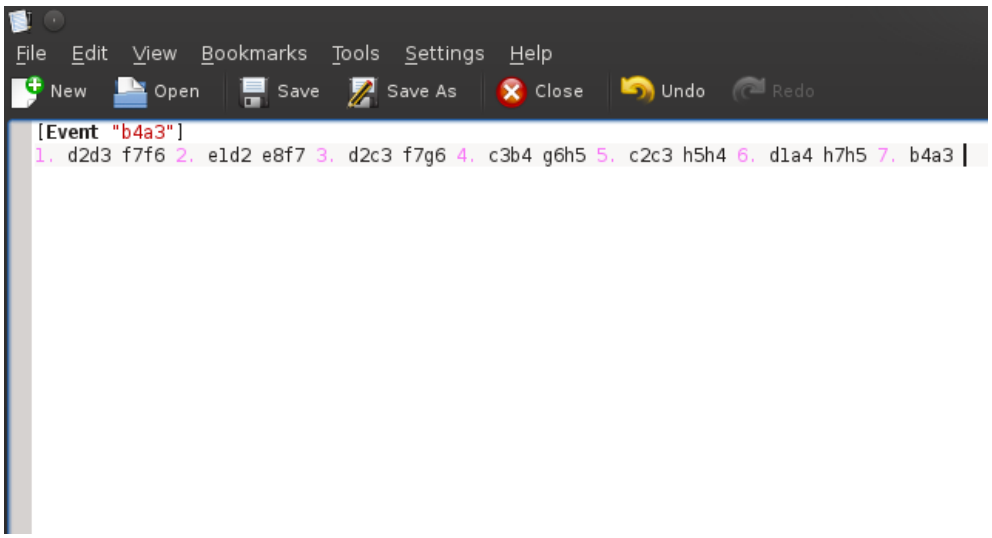
Na sliki 3.2 je za mat s kraljem, ki je predstavljen v uvodu (slika 1.2), prikazan primer zagona programa. Uporabljena je bila naslednja konfiguracija:

- PROBLEM=1
- H=3
- MOVES=7
- WINCOLOR=W
- ENDMOVE=b4a3
- ENDPIECE=K
- TRANSP_TABLE=104858300
- ACTIVE=d2,f7,e1,e8,c2,d1,h7

Izhod programa je poleg izpisa rešitve v terminal datoteka .pgn, v katero se izpišejo vse najdene rešitve. Primer datoteke za zgornjo konfiguracijo je prikazan na sliki 3.3. Datoteko lahko nato za lažje pregledovanje odpremo s katerim od programov, kot je npr. program ChessBase.




Slika 3.2: Slika prikazuje primer zagona programa za problem, ki je prikazan na sliki 1.2 v uvodu.




Slika 3.3: Slika prikazuje primer izhodne PGN datoteke za problem, ki je prikazan na sliki 1.2 v uvodu.

# Poglavje 4

# Rezultati

## 4.1 Metode testiranja programa

Delovanje programa smo preizkusili na množici šahovskih problemov. Ker je število različnih problemov zelo veliko, smo testno množico omejili na hitre mate, ki smo jih omenili že v uvodu. Poleg tega smo program omejili le na iskanje matov, ki se zgodijo s promocijo. Za tak pristop smo se odločili, ker je testna množica obvladljivo majhna, poleg tega pa lahko število takih hitrih matov tudi natančno izračunamo.

Spomnimo se definicije hitrih matov, ki pravi, da je naloga sestaviti najkrajšo igro, ki se konča s potezo, zapisano v algebraični notaciji. Če za vsako polje, na katerem lahko promoviramo, določimo, na koliko načinov lahko to storimo, pri čemer upoštevamo, da sta različna načina le tista, ki ju lahko različno zapišemo z algebraično notacijo, potem lahko izračunamo, da za vsakega igralca obstaja:

- 32 različnih matov brez jemanja - osem različnih stolpcev, kjer v vsakem lahko promoviramo v eno od štirih figur (kraljica, trdnjava, skakač, tekač);

- 56 različnih matov z jemanjem - upoštevati moramo, da kmet lahko jemlje “v levo“ ali “v desno“, kar predstavlja dva različna premika.

Katero figuro s premikom vzamemo, pa ni pomembno, saj ta informacija ni vsebovana v algebraičnem zapisu, zato dva premika, kjer v zadnji potezi vzamemo različni figure, ne predstavljata dveh različnih hitrih matov. Tako kot pri matih brez jemanja lahko promoviramo v štiri različne figure.

Skupaj torej obstaja 176 hitrih matov s promocijami (tabela 1.1). V kolikih potezah je kateri izmed njih izvedljiv, pa je odvisno od posameznega primera.

Pri testiranju smo uporabili vse pristope za spopadanje s kombinatorično zahtevnostjo, ki smo jih opisali v poglavju 2.3.

- Transpozicijska tabela velikosti 1048583 elementov.
- Deaktivacija figur. Ker smo iskali mate s promocijo, lahko za vsak posamezni primer določimo, katere figure je smiselno deaktivirati. Npr. če iščemo mat s promocijo v petih potezah, potem ima lahko igralec, ki mora matirati nasprotnika, aktivnega največ enega kmeta in nobene druge figure, saj le-ta porabi vseh pet potez, preden lahko promovira. Če iščemo mat v šestih potezah, lahko poleg kmeta aktiviramo še eno figuro, saj imamo eno potezo več, kot jo potrebujemo, da kmet promovira. Nasprotno pa na nasprotnikovi strani ne vemo, katere figure lahko varno izklopimo. Proces smo avtomatizirali z uporabo skripte in iskanje ponovili z vsemi možnimi kombinacijami aktivnih figur za določen problem.
- Iskanje več rešitev hkrati. Ciljni pogoj smo postavili zelo ohlapno; in sicer tako, da smo omejili le dolžino rešitve in določili, da zadnji premik predstavlja promocijo. Na tak način smo dosegli, da je program s posamezno kombinacijo aktivnih figur preiskal vse možne mate s promocijo. Kasneje smo rezultate različnih kombinacij aktivnih figur med seboj združili in za vsak hitri mat iz tabele A.1 med njimi poiskali najkrajšo rešitev.

Skripta, ki smo jo napisali za testiranje programa, omogoča avtomatsko

iskanje matov s promocijami brez posredovanja uporabnika. Ob zagonu je potrebno skripti podati parametre, predstavljene v nadaljevanju.

- Konfiguracijska datoteka, ki vsebuje navodila za program (glej poglavje: 3). Določili smo iskano dolžino rešitve, zadnjo potezo (promocija) in uporabljeno hevristiko.
- Število vzporednih procesov. Skripta omogoča pohitritev iskanja za večjederne procesorje tako, da zažene več vzporednih procesov, ki iščejo rešitev z različnimi kombinacijami aktivnih figur.
- Količina pomnilnika v GB. Skupna količina pomnilnika, ki ga ima skripta na voljo in se enakomerno razdeli med vse vzporedne procese.
- Zaporedna številka kombinacije, od katere nadaljujemo iskanje v primeru, da smo iskanje prekinili, preden je program preveril vse kombinacije aktivnih figur.

Skripta najprej generira vse smiselne kombinacije aktivnih figur za problem, ki ga rešujemo. Nato za vsako posamezno kombinacijo zažene program s konfiguracijsko datoteko, ki smo jo podali kot parameter. Pri vsakem klicu programa se uporabi skripta *timeout* [11], ki izvajanje programa prekine, ko le-ta preseže določeno količino pomnilnika. To je potrebno, ker je pri daljših rešitvah drevo igre preveliko, da bi ga lahko shranili v pomnilnik računalnika in je potrebno program prekiniti, preden izčrpa celoten pomnilnik, ki je na voljo. V nadaljevanju so predstavljene nastavitve, ki smo jih uporabili pri iskanju.

- Iskali smo mat fiksne dolžine. Začeli smo z globino petih potez in poskušali najti vse možne mate tako za belega kot za črnega igralca. Globino smo nato povečevali do globine sedmih potez in v vsakem koraku shranili vse najdene mate.
- Uporabili smo hevristiko *cover* in transpozicijsko tabelo velikosti 1048583 elementov.

- Določili smo, da skripta zažene šest vzporednih procesov. Program je tekel na šestjedrnem procesorju "Intel i7 5820K".
- Skripti smo dodelili 32 GB pomnilnika (4 GB/proces).

Na koncu smo rezultate, ki jih je našel program z različnimi kombinacijami aktivnih figur, združili v skupno datoteko, v kateri smo nato za vsak hitri mat s promocijo poiskali najkrajšo najdeno rešitev.

## 4.2 Reševanje in odkrivanje šahovskih problemov

Celotna tabela hitrih matov s promocijami (A.1) je zapisana v dodatku A. V skrajno levem stolpcu so zapisani vsi možni hitri mati s promocijo. Pri tistih, za katere nam je s programom uspelo najti rešitev, je le-ta zapisana poleg.

V nadaljevanju si bomo podrobneje ogledali, kako izbira hevristike vpliva na hitrost delovanja programa. Prav tako bomo predstavili, kako pomembno je izklapljanje figur. Za konec pa smo s programom poskusili najti rešive za probleme, ki od igralca ne zahtevajo iskanja mata. S tem smo pokazali, da je program mogoče uporabiti za različne vrste problemov.

### 4.2.1 Iskanje hitrih matov

Program uporabniku omogoča, da ob zagonu izbere, katero hevristiko želi uporabiti pri preiskovanju. Na izbiro so naslednje možnosti:

- brez hevristike,
- pokritost oz *cover*,
- manhattanska razdalja.

V nadaljevanju si bomo na nekaterih primerih iz tabele A.1 ogledali hitrost delovanja programa z različnimi hevristikami.

V tabeli 4.1 je prikazanih nekaj primerov hitrih matov različnih dolžin. Za vsakega smo zagnali program z različnimi hevristikami ter zabeležili število razvitih vozlišč ter čas iskanja. Polja, označena z znakom "/", pomenijo, da s posamezno hevristiko nismo našli rešitve, preden je program izčpal 16 GB pomnilnika in smo ga prekinili.

Iz tabele je razvidno, da je hevristika *cover* na splošno dajala najboljše rezultate. Pri daljših rešitvah še posebej opazimo, da je izbira hevristike zelo pomembna, saj smo le z njo uspeli najti navedene rešitve dolžine 7 potez.

Manhattanska razdalja je pri krajših rešitvah delovala zelo dobro, pri daljših pa je imela težave. Zanimivo pa je, da je pri uspešnih primerih, četudi so bili daljši, manhattanska razdalja vodila do rešitve hitreje kot hevristika *cover*.

Za primerjavo smo v tabelo vključili tudi stolpec, ki prikazuje delovanje programa brez hevristike. V tem primeru iskanje postane klasično iskanje v širino (BFS). Čeprav je pri krajših rešitvah tak pristop deloval, je uporaba katerekoli hevristike izboljšala delovanje za približno faktor 5. Pri daljših rešitvah pa brez uporabe hevristike nismo našli nobenih rešitev.

### 4.2.2 Vpliv števila aktivnih figur na hitrost iskanja

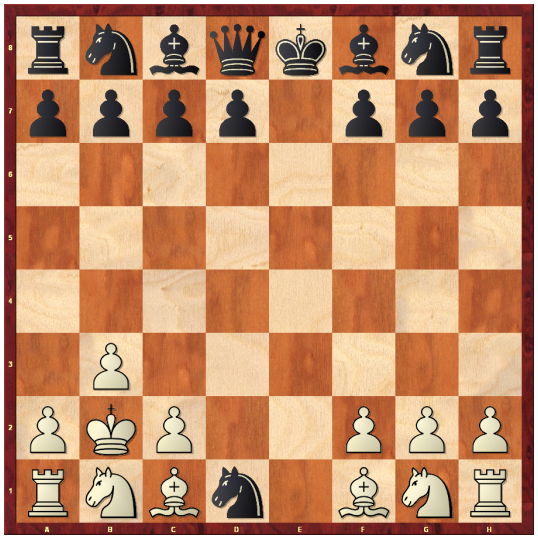
V nadaljevanju so podrobneje predstavljeni trije primeri, ki so bili objavljeni tudi v reviji *Šahovska misel* [22]. Besedilo prispevka je na voljo v dodatku B, problemi, ki smo jih v njem predstavili, pa so naslednji:

- v peti potezi kmet promovira v skakača in matira nasprotnega kralja (slika 4.1);
- v peti potezi kmet promovira v lovca in matira nasprotnega kralja (slika 4.2);
- v šesti potezi eden od skakačev "zamenja barvo" in matira nasprotnega kralja (4.3);

Na vsakem od zgornjih treh primerov smo preizkusili, kako število aktivnih figur vpliva na hitrosti iskanja. Iskanje smo začeli z minimalnim naborom

| Hitri mat | Aktivna | Število preverjenih vozlišč | | |
|---|---|---|---|---|
| | | Brez | Cover | Manhattan |
| 5...e2e1Q | c7 | 3939518 (9,22s) | 1043469 (2,71s) | 4159938 (10,30s) |
| 5...d2d1R | b7 | 4418998 (9,66s) | 612456 (1,70s) | 25641 (0,06s) |
| 5...b2c1Q | a7 | 957902 (5,45s) | 102018 (0,41s) | 69705 (0,18s) |
| 5...c2d1Q | a7 | 974003 (5,49s) | 64556 (0,20s) | 64991 (0,17s) |
| 5...b2c1R | a7 | 712310 (3,90s) | 84787 (0,33s) | 59735 (0,12s) |
| 6...e2e1R | b7,c7 | / | 35948328 (121,54s) | 5188256 (17,40s) |
| 6...f2f1R | h7,d8 | / | 7336201 (28,18s) | / |
| 6...d2d1N | e7,h7 | / | 313654 (1,22s) | / |
| 6...e2e1B | d7,g7 | / | 7915870 (28,76s) | / |
| 6...b2a1Q | a7 | / | 35113637 (116,90s) | 14894897 (47,63s) |
| 7...b2b1N | b7,f7 | / | 8319005 (32,43s) | / |
| 7...g2g1B | h7,d8 | / | 24639883 (107,32s) | / |
| 7...a2b1Q | a7,a8 | / | 6583697 (42,1s) | / |
| 7...a2b1R | a7,a8 | / | 6583697 (42,1s) | / |
| 7...a2b1N | f7,a7 | / | 12951813 (58,28s) | / |

Tabela 4.1: Tabela za posameni hitri mat prikazuje število vozlišč, ki jih je program preveril, preden je našel rešitev z različnimi hevristikami. V oklepaju je naveden tudi čas izvajanja programa na procesorju “Intel i7 5820K“. Pri iskanju so bile aktivne figure, ki so navedene v tabeli vključno z vsemi nasprotnikovimi figurami. Program je imel na voljo 16 GB pomnilnika.



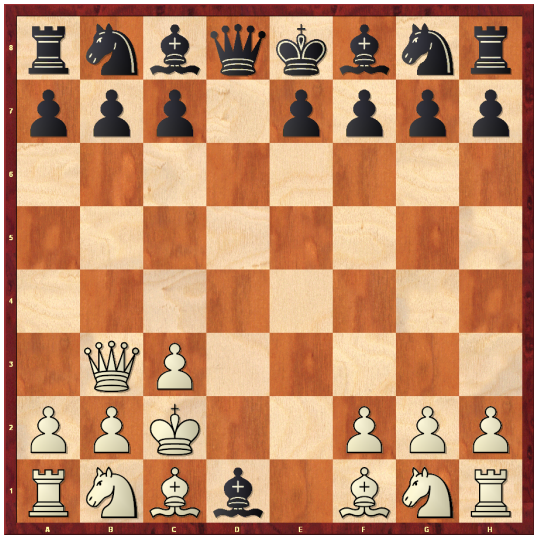
Slika 4.1: Slika končne pozicije za mat s promocijo v skakača v peti potezi. Rešitev: 1. d2d3 e7e5 2. e1d2 e5e4 3. d2c3 e4xd3 4. b2b3 d3xe2 5. c3b2 e2xd1N#.

| Mat | Število aktivnih figur | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| e2xd1N | 0.001 | 0.02 | 0.08 | 0.10 | 0.71 | 4.05 | 12.21 | / | / | / | / |
| d2d1B | / | 0.005 | 0.02 | 0.22 | 0.13 | 0.27 | 1.77 | 5.26 | 18.99 | 36.66 | 55.66 |
| c2xb1N | / | / | / | 0.09 | 0.49 | 68.20 | / | / | / | / | / |

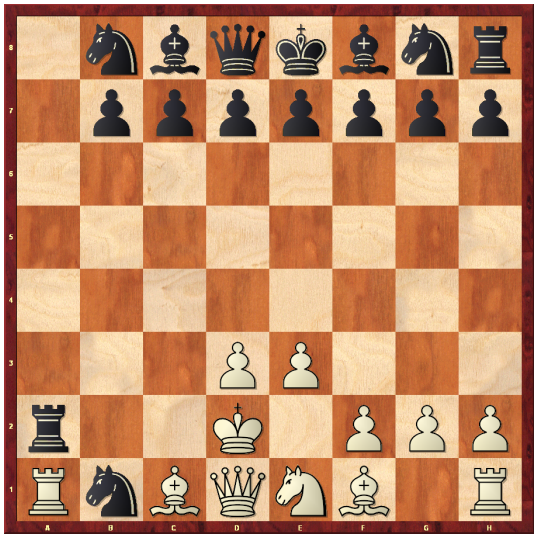
Tabela 4.2: Tabela prikazuje čas iskanja hitrih matov v sekundah v odvisnosti od števila aktivnih figur.

aktivnih figur, nato pa smo nabor povečevali toliko časa, da program ni več našel rešitve. V vsakem koraku smo iskanje ponovili z 20 naključno izbranimi kombinacijami aktivnih figur in tako dobili povprečje. Na sliki 4.4 je prikazano, kako število aktivnih figur vpliva na hitrost iskanja. Izmerjene vrednosti so prikazane tudi v tabeli 4.2.

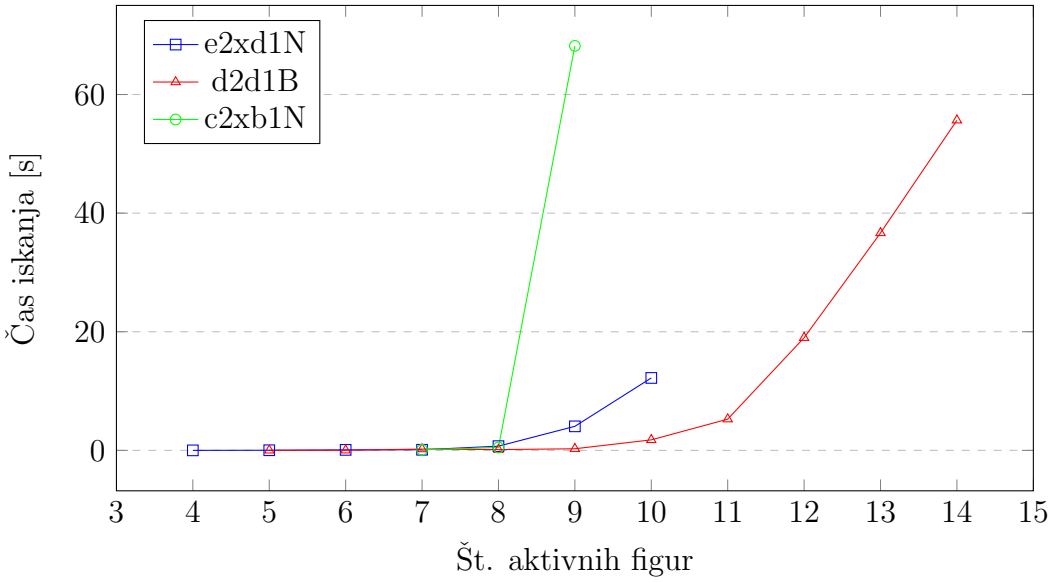
Izvajanja programa nismo časovno omejili. Ko je program porabil ves dodeljeni pomnilnik (16GB) se je njegovo izvajanje avtomatsko prekinilo s pomočjo skripte *timeout* [11].



Slika 4.2: Slika končne pozicije za mat s promocijo v tekača v peti potezi. Rešitev: 1. c2c3 d7d5 2. e2e4 d5xe4 3. d1b3 e4e3 4. e1d1 e3xd2 5. d1c2 d2d1B#.



Slika 4.3: Slika končne pozicije za mat, kjer skakač v šesti potezi zamenja barvo. Rešitev: 1. b2b4 a7a5 2. d2d3 a5xb4 3. e1d2 a8xa2 4. e2e3 b4b3 5. g1f3 b3xc2 6. f3e1 c2xb1N#.




Slika 4.4: Čas iskanja v odvisnosti od števila aktivnih figur.

### 4.2.3 Reševanje konstrukcijskih nalog

Program poleg iskanja matov omogoča tudi reševanje drugačnih vrst problemov, npr. konstrukcijskih nalog. Z naborom parametrov, ki jih ponuja (poglavje 3), lahko določimo različne ciljne pogoje. Program smo testirali na iskanju legalnega zaporedja potez, ki vodi do vnaprej določene šahovske pozicije v podanem številu potez. Gre torej za konstrukcijske naloge, kjer moramo v določenem številu potez odigrati partijo, ki se konča v izbrani poziciji. Program smo preizkusili na naslednjih dveh primerih:

- po štirih potezah nastane šahovska pozicija, ki je razvidna na sliki 4.5;
- po štirih potezah nastane šahovska pozicija, ki je razvidna na sliki 4.6.

Hevristika, ki smo jo uporabili pri iskanju, je nekoliko prilagojena manhattanska razdalja. Namesto da hevristika izračuna manhattansko razdaljo figur do nasprotnikovega kralja, le-ta izračuna razdaljo od trenutnega položaja figure do položaja figure v iskani poziciji. Razdalja se izračuna za vse figure, razen za kmete. Če imamo na šahovnici dve enaki figuri (npr. dva skakača),

| Problem | Aktivne figure | Rešitev | Število vozlišč | Čas iskanja |
|---|---|---|---|---|
| Slika 4.5 | b8,g8,g1,d7 | 1. g1f3 d7d5 2. f3d4 g8f6 3. d4c6 f6d7 4. c6xb8 d7xb8 | 7710 | 0,02s |
| Slika 4.6 | e2,f1,c7,e7,e8, d7,d8,b8,c8 | 1. e2e4 e7e6 2. f1b5 e8e7 3. b5xd7 c7c6 4. d7e8 e7xe8 | 709955 | 1,84s |

Tabela 4.3: Tabela prikazuje rešitvi za problema na slikah 4.5 in 4.6. Program je tekel na procesorju "Intel i7 5829K".

se izračuna povprečje razdalj med vsemi kombinacijami trenutne in končne pozicije le-teh.

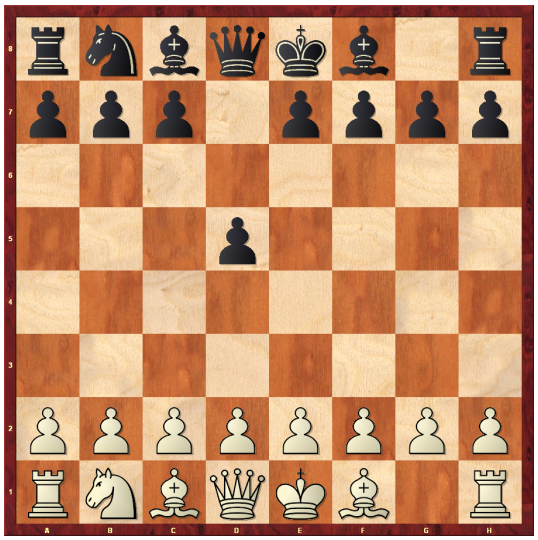
Iz tabele 4.3 sta razvidni rešitvi za problema iz slik 4.5 in 4.6. Poleg rešitve je zapisano, katere figure so bile aktivne pri iskanju. Prav tako je razvidno tudi število pregledanih vozlišč in čas, ki ga je program uporabil za preiskovanje.

## 4.3 GitHub povezava

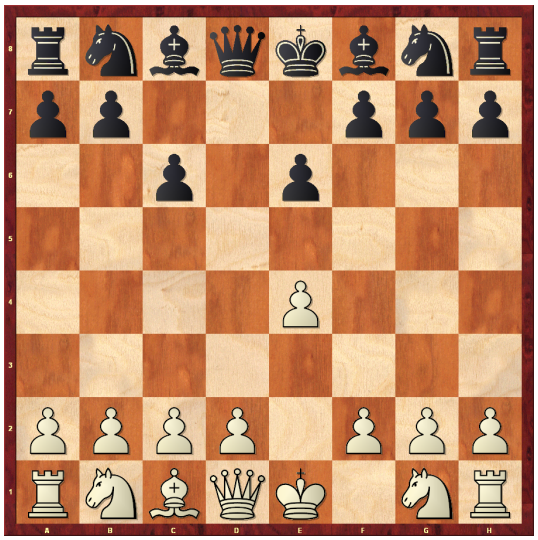
Program je skupaj z vsemi skriptami in testnimi primeri prosto dostopen na naslednji povezavi:

*https://github.com/mitjari/Avtomatsko-odkrivanje-zanimivih-sahovskih-problemov.git*

K programu so priloženi primeri konfiguracijskih datotek za vse probleme, obravnavane v diplomskem delu. Prav tako je priložena datoteka z opisom vseh parametrov, ki so uporabniku na voljo.



Slika 4.5: Slika končne pozicije, ki jo moramo doseči po štirih potezah belega in štirih potezah črnega iz začetne šahovske pozicije.



Slika 4.6: Slika končne pozicije, ki jo moramo doseči po štirih potezah belega in štirih potezah črnega iz začetne šahovske pozicije.

# Poglavje 5

# Zaključek

Ključni cilj diplomske naloge je bil izdelati program, ki bi uporabniku pomagal pri reševanju in odkrivanju zanimivih šahovskih problemov. Uporabniku smo omogočili nastavljanje različnih parametrov iskanja in s tem iskanje raznolikih rešitev. Poleg odkrivanja matov program omogoča tudi reševanje drugačnih problemov, kot so na primer konstrukcijske naloge.

Ker je kombinatorična zahtevnost šaha prevelika, tako rekoč ne moremo preiskati vseh možnosti, ki jih imamo na voljo pri iskaju rešitve. Zato smo v programu uporabili algoritem A*, ki iskanje s pomočjo hevrističnih ocen usmerja proti rešitvi in s tem poveča možnosti, da rešitev tudi najdemo. V končni verziji programa smo implementirali dve funkciji za izračun hevrističnih ocen; in sicer pokritost (ang. *cover*) in manhattansko razdaljo. Poleg preiskovalnega algoritma A* smo za premagovanje kombinatorične zahtevnosti uporabili še nekaj tehnik, ki so izboljšale delovanje programa. Uporabniku smo tako omogočili izbiranje figur, s katerimi želi iskati rešitev. Na ta način se zmanjša razvejanost iskalnega drevesa in posledično izboljša delovanje programa. Poleg tega smo implementirali zaznavanje transpozicij. Torej vozlišč, do katerih smo prišli po različnih poteh, ne preiskujemo večkrat, kar spet zmanjša razvejanost drevesa in pripomore k hitrosti preiskovanja ter poveča možnosti za iskanje daljših rešitev. Hranjenje transpozicij smo implementirali z zgoščevalno tabelo ter zgoščevalno funkcijo Zobrist, kar

programu omogoča zelo hitro iskanje po zgodovini transpozicij.

Program smo preizkušali na množici primerov in pri tem primerjali različne hevristike. Kot smo pričakovali, se je hevristika *cover* odnesla najbolje, zato smo jo uporabili tudi pri iskanju hitrih matov s promocijo. Tri izmed najdenih problemov smo v sklopu nagradnje igre predstavili tudi v prispevku, objavljenem v reviji Šahovska misel. Poleg problemov, kjer je potrebno poiskati mat, pa smo s programom rešili tudi dve konstrukcijski nalogi.

Za enostavnejšo uporabo program potrebuje grafični vmesnik, kar bi lahko bil prvi korak pri nadaljnjem delu. Tak vmesnik bi uporabniku omogočal izbiro aktivnih figur, vrsto problema itd., kar bi omogočalo enostavnejšo uporabo. Poleg tega bi lahko uporabniku najdeno rešitev tudi grafično predstavili. Preiskovanje samo bi lahko izboljšali tako, da bi zmanjšali razvejanost iskalnega drevesa, saj velika razvejanost in posledično velika kombinatorična kompleksnost predstavljata glavni problem pri iskanju daljših rešitev. To bi najlažje storili z uporabo boljših hevrističnih ocen, kar bi lahko bila tudi iztočnica za nadaljnje delo.

# Literatura


[1] Azlan iqbal: Recomposition contest result. http://en.chessbase.com/post/azlan-iqbal-recomposition-contest-result. Dostop: 7. 9. 2016.

[2] Chess problem. https://en.wikipedia.org/wiki/Chess_problem. Dostop: 16. 8. 2016.

[3] Chessbase chrismas puzzles 2015 (5). http://en.chessbase.com/post/chessbase-chrismas-puzzles-2015-5. Dostop: 7. 9. 2016.

[4] Chessbase.com. http://en.chessbase.com/post/the-infamous-1999-chebase-christmas-puzzle-160813. Dostop: 1. 4. 2016.

[5] Chessbase.com. http://en.chessbase.com/post/fifteen-years-of-chessbase-christmas-puzzles. Dostop: 18. 5. 2016.

[6] Chesthetica. http://www.chesthetica.com/. Dostop: 6. 9. 2016.

[7] Eight queens puzzle. https://en.wikipedia.org/wiki/Eight_queens_puzzle. Dostop: 16. 8. 2016.

[8] Forsyth-Edwards notation. https://chessprogramming.wikispaces.com/Forsyth-Edwards+Notation. Dostop: 13. 7. 2016.

[9] Francois Labellee homepage. http://wismuth.com/chess/chess.html. Dostop: 20. 5. 2016.

[10] Github-sachista-chess. https://github.com/dsaiko/sachista-chess. Dostop: 22. 8. 2016.

[11] Github-timeout. https://github.com/pshved/timeout. Dostop: 29. 8. 2016.

[12] A machine that composes chess problems. http://en.chessbase.com/post/a-machine-that-composes-chess-problems. Dostop: 6. 9. 2016.

[13] Portable game notation. https://chessprogramming.wikispaces.com/Portable+Game+Notation. Dostop: 13. 7. 2016.

[14] Statistics on chess games. http://wismuth.com/chess/statistics-games.html. Dostop: 20. 5. 2016.

[15] Stuart Rachels homepage. http://www.jamesrachels.org/stuart/. Dostop: 20. 5. 2016.

[16] Wikipedia: Algebraic notation. https://en.wikipedia.org/wiki/Algebraic_notation_(chess). Dostop: 20. 5. 2016.

[17] Wikipedia: Helpmate. https://en.wikipedia.org/wiki/Helpmate. Dostop: 19. 5. 2016.

[18] Shirish Chinchalkar. An upper bound for the number of reachable positions. *ICCA Journal, Vol. 19, No. 3*, 1996.

[19] Alan K. Mackworth David L. Poole. *Artificial inteligence.* CAMBRIDGE UNIVERSITY PRESS, 2010.

[20] J.W.H.M. Uiterwijk D.M. Breuker and H.J. van den Herik. Replacement schemes for transposition tables.

[21] Matej Guid. *Search and Knowledge for Human and Machine Problem Solving.* PhD thesis, University of Ljubljana, Slovenia, 2010.

[22] Matej Guid and Mitja Rizvič. Računalniško generirani šahovski problemi. *Šahovska misel*, 2016.

[23] Azlan Iqbal, Matej Guid, Simon Colton, Jana Krivec, Shazril Azman, and Boshra Haghighi. *The Digital Synaptic Neural Substrate: A New Approach to Computational Creativity*, volume 3. Springer, 2016.

[24] Azlan Iqbal, Harold van der Heijden, Matej Guid, and Ali Makhmali. Evaluating the aesthetics of endgame studies: a computational model of human aesthetic perception. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(3):178–191, 2012.

[25] Vito Janko. Razvoj programa za igranje 1-2-3 šaha. Master's thesis, Univerza v Ljubljani Fakulteta za matematiko in fiziko, Fakulteta za računalništvo in informatiko.

[26] Vito Janko and Matej Guid. A program for progressive chess. *Theoretical Computer Science*, 644:76–91, 2016. Recent Advances in Computer Games.

[27] Raymond M. Smullyan. *Šahovske skrivnosti Sherlocka Holmesa*. Državna založba Slovenije, 1986.

[28] Raymond M. Smullyan. *Šahirazada*. Državna založba Slovenije, 1992.

[29] Albert L Zobrist. A new hashing method with application for game playing. *ICCA journal*, 13(2):69–73, 1970.

# Dodatek A

# Tabela hitrih matov

Spodnja tabela (A.1) prikazuje vse najdene hitre mate, ki smo jih našli s programom po postopku opisanem v poglavju 5. Vsaka vrstica vsebuje zaključno potezo, katera določa mat, dolžino rešitve in zaporedje premikov, ki reši problem. Znak "/" pomeni, da program rešitve za posamezni mat ni našel.

| Zadnja poteza | Dolžina rešitve | Rešitev |
|---|---|---|
| a7a8Q | / | / |
| b7b8Q | / | / |
| c7c8Q | 5 | 1. c2c4 d7d5 2. c4c5 d8d6 3. c5xd6 c8h3 4. d6xc7 h3xg2 5. c7c8Q |
| d7d8Q | 5 | 1. d2d4 c7c5 2. d4xc5 e7e6 3. c5c6 d8h4 4. c6xd7 e8e7 5. d7d8Q |
| e7e8Q | 6 | 1. a2a3 d7d5 2. e2e4 d8d6 3. e4e5 e8d8 4. e5xd6 c8h3 5. d6xe7 d8c8 6. e7e8Q |
| f7f8Q | 6 | 1. b2b3 e7e5 2. g2g4 f8a3 3. g4g5 c7c5 4. g5g6 d8c7 5. g6xf7 e8d8 6. f7f8Q |
| g7g8Q | 6 | 1. a2a3 g7g5 2. e2e4 h7h5 3. e4e5 g8f6 4. e5xf6 f8g7 5. f6xg7 h8h6 6. g7g8Q |
| h7h8Q | / | / |
| a7a8R | / | / |

| | | |
|---|---|---|
| b7b8R | / | / |
| c7c8R | 6 | 1. a2a3 d7d5 2. e2e4 g8f6 3. e4e5 d8d6 4. e5xd6 c8h3 5. d6xc7 f6d7 6. c7c8R |
| d7d8R | 6 | 1. a2a3 d7d5 2. e2e4 g7g5 3. e4e5 d8d6 4. e5xd6 f8g7 5. d6d7 e8f8 6. d7d8R |
| e7e8R | / | / |
| f7f8R | / | / |
| g7g8R | 6 | 1. a2a3 g7g5 2. e2e4 h7h5 3. e4e5 g8f6 4. e5xf6 f8g7 5. f6xg7 h8h6 6. g7g8R |
| h7h8R | / | / |
| a7a8N | / | / |
| b7b8N | / | / |
| c7c8N | / | / |
| d7d8N | / | / |
| e7e8N | / | / |
| f7f8N | / | / |
| g7g8N | / | / |
| h7h8N | / | / |
| a7a8B | / | / |
| b7b8B | / | / |
| c7c8B | / | / |
| d7d8B | 6 | 1. d2d4 c7c5 2. f2f3 d8b6 3. d4xc5 e8d8 4. c5c6 d8c7 5. c6xd7 b6c6 6. d7d8B |
| e7e8B | 6 | 1. d2d4 c7c5 2. e2e4 f7f6 3. d4d5 e8f7 4. d5d6 f7g6 5. d6xe7 d8b6 6. e7e8B |
| f7f8B | / | / |
| g7g8B | / | / |
| h7h8B | / | / |
| a2a1Q | / | / |
| b2b1Q | / | / |

| | | |
|---|---|---|
| c2c1Q | 5 | 1. b2b3 a7a5 2. c2c3 a5a4 3. c1a3 a4xb3 4. d1c2 b3xc2 5. a3b4 c2c1Q |
| d2d1Q | 5 | 1. e2e3 b7b5 2. f1c4 b5xc4 3. d2d3 c4c3 4. d1d2 c3xd2 5. e1f1 d2d1Q |
| e2e1Q | 5 | 1. d2d4 c7c5 2. d1d3 c5c4 3. c1e3 c4xd3 4. e1d2 d3xe2 5. d2c1 e2e1Q |
| f2f1Q | 5 | 1. e2e4 d7d5 2. f1e2 d5xe4 3. e2h5 e4e3 4. d1g4 e3xf2 5. e1d1 f2f1Q |
| g2g1Q | 5 | 1. g2g3 e7e5 2. h2h3 e5e4 3. g1f3 e4xf3 4. f1g2 f3xg2 5. h1h2 g2g1Q |
| h2h1Q | / | / |
| a2a1R | / | / |
| b2b1R | / | / |
| c2c1R | 5 | 1. b2b3 a7a5 2. c2c3 a5a4 3. c1a3 a4xb3 4. d1c2 b3xc2 5. a3b4 c2c1R |
| d2d1R | 5 | 1. e2e3 b7b5 2. f1c4 b5xc4 3. d1f3 c4c3 4. e1f1 c3xd2 5. g1e2 d2d1R |
| e2e1R | 6 | 1. d2d3 c7c5 2. c1e3 c5c4 3. e1d2 c4xd3 4. d2c1 d3xe2 5. d1d3 b7b5 6. b1d2 e2e1R |
| f2f1R | 6 | 1. c2c3 h7h5 2. g2g3 h5h4 3. f1h3 h4xg3 4. h3xd7 d8xd7 5. d1c2 g3xf2 6. e1d1 f2f1R |
| g2g1R | 5 | 1. g2g4 h7h5 2. h2h4 h5xg4 3. g1h3 g4xh3 4. f1g2 h3xg2 5. h1h2 g2g1R |
| h2h1R | / | / |
| a2a1N | / | / |
| b2b1N | 7 | 1. d2d3 b7b5 2. e1d2 b5b4 3. a2a3 f7f5 4. b1c3 f5f4 5. a1a2 b4xa3 6. d1e1 a3xb2 7. c3d1 b2b1N |

| | | |
|---|---|---|
| c2c1N | / | / |
| d2d1N | 6 | 1. d2d3 e7e5 2. e1d2 e5e4 3. d2c3 e4e3 4. b2b3 h7h5 5. d1d2 e3xd2 6. c3b2 d2d1N |
| e2e1N | 6 | 1. a2a3 d7d5 2. c2c4 d5xc4 3. d1b3 d8d4 4. d2d3 c4xd3 5. e1d2 d3xe2 6. d2c2 e2e1N |
| f2f1N | / | / |
| g2g1N | / | / |
| h2h1N | / | / |
| a2a1B | / | / |
| b2b1B | / | / |
| c2c1B | / | / |
| d2d1B | 5 | 1. c2c3 d7d5 2. e2e4 d5xe4 3. d1b3 e4e3 4. e1d1 e3xd2 5. d1c2 d2d1B |
| e2e1B | 6 | 1. c2c4 d7d5 2. e2e4 d5xe4 3. f2f3 e4e3 4. d1c2 e3e2 5. e1f2 g7g5 6. f2g3 e2e1B |
| f2f1B | / | / |
| g2g1B | 7 | 1. f2f3 h7h5 2. e1f2 h5h4 3. d2d3 h4h3 4. c1f4 h3xg2 5. f4xc7 d8xc7 6. g1h3 c7xh2 7. d1e1 g2g1B |
| h2h1B | / | / |
| a7xb8Q | / | / |
| b7xa8Q | / | / |
| b7xc8Q | 5 | 1. a2a4 b7b5 2. a4a5 b5b4 3. a5a6 c8b7 4. a6xb7 d8c8 5. b7xc8Q |
| c7xb8Q | / | / |
| c7xd8Q | 5 | 1. a2a4 b7b5 2. a4xb5 e7e5 3. b5b6 f8a3 4. b6xc7 e8f8 5. c7xd8Q |
| d7xc8Q | 6 | 1. a2a3 d7d5 2. e2e4 g7g5 3. e4e5 d8d6 4. e5xd6 f8g7 5. d6d7 e8f8 6. d7xc8Q |

| | | |
|---|---|---|
| d7xe8Q | 6 | 1. d2d4 c7c5 2. e2e4 f7f6 3. d4xc5 e8f7 4. c5c6 f7g6 5. c6xd7 d8e8 6. d7xe8Q |
| e7xd8Q | 5 | 1. d2d4 c7c5 2. d4xc5 d7d5 3. c5xd6 a7a5 4. d6xe7 a5a4 5. e7xd8Q |
| e7xf8Q | 6 | 1. f2f4 c7c5 2. f4f5 d8b6 3. f5f6 b6xb2 4. f6xe7 b2xb1 5. c1a3 c5c4 6. e7xf8Q |
| f7xe8Q | 6 | 1. a2a3 d7d5 2. e2e4 c8h3 3. e4e5 e8d7 4. e5e6 d7c8 5. e6xf7 d8e8 6. f7xe8Q |
| f7xg8Q | / | / |
| g7xf8Q | 5 | 1. d2d4 e7e5 2. d4xe5 d8f6 3. e5xf6 a7a5 4. f6xg7 e8d8 5. g7xf8Q |
| g7xh8Q | 5 | 1. e2e4 g7g5 2. e4e5 g8f6 3. e5xf6 f8g7 4. f6xg7 g5g4 5. g7xh8Q |
| h7xg8Q | 6 | 1. b2b3 g7g6 2. g2g4 h7h5 3. g4g5 h8h6 4. g5xh6 f8g7 5. h6h7 g7xa1 6. h7xg8Q |
| a7xb8R | / | / |
| b7xa8R | / | / |
| b7xc8R | 5 | 1. a2a4 b7b5 2. a4a5 b5b4 3. a5a6 c8b7 4. a6xb7 d8c8 5. b7xc8R |
| c7xb8R | / | / |
| c7xd8R | 5 | 1. a2a4 b7b5 2. a4xb5 g7g5 3. b5b6 f8g7 4. b6xc7 e8f8 5. c7xd8R |
| d7xc8R | 6 | 1. a2a3 d7d5 2. e2e4 g7g5 3. e4e5 d8d6 4. e5xd6 f8g7 5. d6d7 e8f8 6. d7xc8R |
| d7xe8R | / | / |
| e7xd8R | 6 | 1. a2a3 e7e6 2. e2e4 g8f6 3. e4e5 f8e7 4. e5xf6 e8f8 5. f6xe7 f8g8 6. e7xd8R |
| e7xf8R | 6 | 1. f2f4 c7c5 2. f4f5 d8b6 3. f5f6 b6xb2 4. f6xe7 b2xb1 5. c1a3 c5c4 6. e7xf8R |
| f7xe8R | / | / |

| | | |
|---|---|---|
| f7xg8R | / | / |
| g7xf8R | 5 | 1. e2e4 c7c5 2. e4e5 f7f5 3. e5xf6 d8c7 4. f6xg7 e8d8 5. g7xf8R |
| g7xh8R | 5 | 1. e2e4 g7g5 2. e4e5 g8f6 3. e5xf6 f8g7 4. f6xg7 g5g4 5. g7xh8R |
| h7xg8R | 6 | 1. b2b3 g7g6 2. g2g4 h7h5 3. g4g5 h8h6 4. g5xh6 f8g7 5. h6h7 g7xa1 6. h7xg8R |
| a7xb8N | / | / |
| b7xa8N | / | / |
| b7xc8N | / | / |
| c7xb8N | / | / |
| c7xd8N | 6 | 1. d2d4 b7b6 2. f2f4 f7f5 3. d4d5 g8f6 4. d5d6 e8f7 5. d6xc7 f7e6 6. c7xd8N |
| d7xc8N | / | / |
| d7xe8N | / | / |
| e7xd8N | 6 | 1. a2a4 b7b6 2. e2e4 d7d5 3. e4xd5 e8d7 4. d5d6 d7c6 5. d6xe7 c6b7 6. e7xd8N |
| e7xf8N | / | / |
| f7xe8N | / | / |
| f7xg8N | / | / |
| g7xf8N | / | / |
| g7xh8N | / | / |
| h7xg8N | / | / |
| a7xb8B | / | / |
| b7xa8B | / | / |
| b7xc8B | / | / |
| c7xb8B | / | / |
| c7xd8B | / | / |
| d7xc8B | / | / |

| | | |
|---|---|---|
| d7xe8B | 6 | 1. d2d4 c7c5 2. e2e4 f7f6 3. d4xc5 e8f7 4. c5c6 f7g6 5. c6xd7 d8e8 6. d7xe8B |
| e7xd8B | 6 | 1. d2d4 d7d5 2. e2e4 d5xe4 3. d4d5 c7c6 4. d5d6 e8d7 5. d6xe7 d7c7 6. e7xd8B |
| e7xf8B | / | / |
| f7xe8B | / | / |
| f7xg8B | / | / |
| g7xf8B | / | / |
| g7xh8B | / | / |
| h7xg8B | / | / |
| a2xb1Q | 7 | 1. b2b4 a7a5 2. d2d3 a5xb4 3. e1d2 a8xa2 4. c1a3 a2b2 5. d2c1 b4b3 6. a1a2 b3xa2 7. c2c3 a2xb1Q |
| b2xa1Q | 6 | 1. b2b4 a7a6 2. e2e3 a6a5 3. b1c3 a5xb4 4. c1b2 b4xc3 5. d1e2 c3xb2 6. e1d1 b2xa1Q |
| b2xc1Q | 5 | 1. a2a3 a7a5 2. b2b4 a5xb4 3. c2c3 b4xc3 4. c1b2 c3xb2 5. d1c1 b2xc1Q |
| c2xb1Q | 6 | 1. a2a3 e7e5 2. d2d3 f8b4 3. c1d2 e5e4 4. c2c4 e4xd3 5. d1c2 d3xc2 6. a1a2 c2xb1Q |
| c2xd1Q | 5 | 1. b2b4 a7a5 2. c2c3 a5xb4 3. e2e3 b4xc3 4. f1d3 c3c2 5. e1f1 c2xd1Q |
| d2xc1Q | 5 | 1. d2d3 c7c5 2. g2g3 c5c4 3. d1d2 c4c3 4. f1g2 c3xd2 5. e1f1 d2xc1Q |
| d2xe1Q | 6 | 1. a2a3 e7e5 2. d2d3 f8b4 3. d1d2 e5e4 4. e1d1 e4xd3 5. d2e1 d3d2 6. b2b3 d2xe1Q |
| e2xd1Q | 5 | 1. d2d3 c7c5 2. g1f3 c5c4 3. f3e5 c4xd3 4. e5xd7 d3xe2 5. d7c5 e2xd1Q |

| | | |
|---|---|---|
| e2xf1Q | 6 | 1. e2e3 h7h5 2. d1g4 h5xg4 3. g1f3 g4xf3 4. f1e2 a7a5 5. e1g1 f3xe2 6. g1h1 e2xf1Q |
| f2xe1Q | 5 | 1. d2d4 c7c5 2. c1e3 c5xd4 3. e1d2 d4xe3 4. d2c1 e3xf2 5. d1e1 f2xe1Q |
| f2xg1Q | 7 | 1. a2a3 d7d5 2. b2b3 c8g4 3. e2e4 d5xe4 4. h2h3 e4e3 5. h1h2 d8d4 6. f1c4 e3xf2 7. e1f1 f2xg1Q |
| g2xf1Q | 5 | 1. a2a3 h7h5 2. b2b3 h5h4 3. c2c3 h4h3 4. d1c2 h3xg2 5. e1d1 g2xf1Q |
| g2xh1Q | 5 | 1. g2g4 h7h5 2. h2h4 h5xg4 3. g1h3 g4xh3 4. f1g2 h3xg2 5. b1a3 g2xh1Q |
| h2xg1Q | 5 | 1. g2g4 h7h5 2. h2h4 h5xg4 3. h1h2 g4g3 4. f1g2 g3xh2 5. g2h1 h2xg1Q |
| a2xb1R | 7 | 1. b2b4 a7a5 2. d2d3 a5xb4 3. e1d2 a8xa2 4. c1a3 a2b2 5. d2c1 b4b3 6. a1a2 b3xa2 7. c2c3 a2xb1R |
| b2xa1R | 6 | 1. b2b4 a7a6 2. e2e3 a6a5 3. b1c3 a5xb4 4. c1b2 b4xc3 5. d1e2 c3xb2 6. e1d1 b2xa1R |
| b2xc1R | 5 | 1. a2a3 a7a5 2. b2b4 a5xb4 3. c2c3 b4xc3 4. c1b2 c3xb2 5. d1c1 b2xc1R |
| c2xb1R | 6 | 1. a2a3 e7e5 2. d2d3 f8b4 3. c1d2 e5e4 4. c2c4 e4xd3 5. d1c2 d3xc2 6. a1a2 c2xb1R |
| c2xd1R | 5 | 1. b2b4 a7a5 2. c2c3 a5xb4 3. g2g3 b4xc3 4. f1g2 c3c2 5. e1f1 c2xd1R |
| d2xc1R | 5 | 1. d2d3 c7c5 2. g2g3 c5c4 3. d1d2 c4c3 4. f1g2 c3xd2 5. e1f1 d2xc1R |

| | | |
|---|---|---|
| d2xe1R | 6 | 1. a2a3 e7e5 2. d2d3 f8b4 3. d1d2 e5e4 4. e1d1 e4xd3 5. d2e1 d3d2 6. b2b3 d2xe1R |
| e2xd1R | 5 | 1. e2e3 c7c5 2. f1d3 c5c4 3. g1e2 c4xd3 4. e1f1 d3xe2 5. f1g1 e2xd1R |
| e2xf1R | 6 | 1. e2e3 h7h5 2. d1g4 h5xg4 3. g1f3 g4xf3 4. f1e2 a7a5 5. e1g1 f3xe2 6. g1h1 e2xf1R |
| f2xe1R | 6 | 1. a2a3 e7e5 2. d2d3 f8b4 3. d1d2 e5e4 4. f2f3 e4xf3 5. e1d1 f3f2 6. d2e1 f2xe1R |
| f2xg1R | 7 | 1. a2a3 d7d5 2. b2b3 c8g4 3. e2e4 d5xe4 4. h2h3 e4e3 5. h1h2 d8d4 6. f1c4 e3xf2 7. e1f1 f2xg1R |
| g2xf1R | 5 | 1. a2a3 h7h5 2. b2b3 h5h4 3. c2c3 h4h3 4. d1c2 h3xg2 5. e1d1 g2xf1R |
| g2xh1R | 5 | 1. g2g4 h7h5 2. h2h4 h5xg4 3. g1h3 g4xh3 4. f1g2 h3xg2 5. b1a3 g2xh1R |
| h2xg1R | 5 | 1. g2g4 h7h5 2. h2h4 h5xg4 3. h1h2 g4g3 4. f1g2 g3xh2 5. g2h1 h2xg1R |
| a2xb1N | 7 | 1. d2d3 f7f5 2. e1d2 f5f4 3. a2a3 a7a5 4. b2b4 a5xb4 5. g1f3 b4b3 6. a1a2 b3xa2 7. f3e1 a2xb1N |
| b2xa1N | 6 | 1. b2b4 a7a5 2. d2d3 a5xb4 3. e1d2 b4b3 4. c2c3 b3b2 5. d2c2 a8xa2 6. e2e3 b2xa1N |
| b2xc1N | / | / |
| c2xb1N | 6 | 1. b2b4 a7a5 2. d2d3 a5xb4 3. e1d2 a8xa2 4. e2e3 b4b3 5. g1f3 b3xc2 6. f3e1 c2xb1N |

| | | |
|---|---|---|
| c2xd1N | 6 | 1. b2b4 a7a5 2. d2d3 a5xb4 3. e1d2 b4b3 4. f2f3 a8a4 5. d2e3 b3xc2 6. b1d2 c2xd1N |
| d2xc1N | 7 | 1. a2a3 d7d5 2. c2c4 d5xc4 3. b2b3 d8d3 4. f2f3 c4c3 5. e2e3 d3c2 6. e1e2 c3xd2 7. d1e1 d2xc1N |
| d2xe1N | 7 | 1. d2d3 c7c5 2. e1d2 c5c4 3. d2e3 g7g5 4. g2g3 c4xd3 5. e3f3 g5g4 6. f3g2 d3d2 7. d1e1 d2xe1N |
| e2xd1N | 5 | 1. d2d3 e7e5 2. e1d2 e5e4 3. d2c3 e4xd3 4. b2b3 d3xe2 5. c3b2 e2xd1N |
| e2xf1N | / | / |
| f2xe1N | / | / |
| f2xg1N | / | / |
| g2xf1N | 7 | 1. d2d3 b7b5 2. e1d2 b5b4 3. a2a3 f7f5 4. g1f3 f5f4 5. f3e1 f4f3 6. h2h3 f3xg2 7. h1h2 g2xf1N |
| g2xh1N | 6 | 1. f2f3 h7h5 2. e1f2 h5h4 3. e2e3 h4h3 4. f2g3 h3xg2 5. d1e1 h8xh2 6. g3f2 g2xh1N |
| h2xg1N | 7 | 1. a2a3 d7d5 2. c2c4 d5xc4 3. e2e3 g7g5 4. e1e2 g5g4 5. g1f3 g4g3 6. f3e1 g3xh2 7. h1g1 h2xg1N |
| a2xb1B | / | / |
| b2xa1B | 6 | 1. d2d3 a7a5 2. e1d2 a5a4 3. d2c3 a4a3 4. b2b3 a8a4 5. c1b2 a3xb2 6. d1d2 b2xa1B |
| b2xc1B | / | / |
| c2xb1B | / | / |

| | | |
|---|---|---|
| c2xd1B | 6 | 1. d2d3 e7e5 2. e1d2 e5e4 3. d2c3 b7b5 4. c3b3 e4xd3 5. c1d2 d3xc2 6. d2c3 c2xd1B |
| d2xc1B | 7 | 1. d2d3 c7c5 2. e1d2 c5c4 3. d2c3 a7a5 4. d1d2 a5a4 5. c3b4 c4c3 6. b4a3 c3xd2 7. b2b4 d2xc1B |
| d2xe1B | 6 | 1. a2a3 d7d5 2. c2c4 d5xc4 3. f2f4 d8d5 4. e1f2 c4c3 5. f2g3 c3xd2 6. d1e1 d2xe1B |
| e2xd1B | 6 | 1. d2d3 e7e5 2. e1d2 e5e4 3. d2c3 b7b5 4. c3b3 e4xd3 5. c2c3 d3xe2 6. f2f3 e2xd1B |
| e2xf1B | / | / |
| f2xe1B | / | / |
| f2xg1B | / | / |
| g2xf1B | / | / |
| g2xh1B | / | / |

Tabela A.1: Tabela prikazuje vse možne mate s promocijami ter rešitve za tiste primere za katere smo jih našli s programom.

# Dodatek B

# Prispevek v reviji Šahovska misel

Naslednji prispevek je bil objavljen kot nagradna igra v reviji šahovska misel št. 2016/3 [22]:

***RAČUNALNIŠKO GENERIRANI ŠAHOVSKI PROBLEMI***

*Naš cilj je bil razviti računalniški program s katerim bi lahko odkrili nove šahovske probleme, podobne kot problem, ki je bil leta 1999 objavljen na spletni strani ChessBase:“ V začetni poziciji beli igra 1.e4. Partija se konča v peti potezi ko skakač, vzame trdnjavo z matom“. “Vse“ kar je bilo potrebno storiti, je bilo najti zaporedje legalnih potez, ki bo zadostilo pogojem, kar pa se je izkazalo za izjemno zahtevno nalogo. S pomočjo računalniškega programa lahko sedaj poleg reševanja takih problemov odkrijemo tudi nove sorodne probleme. Po nam dostopnih informacijah je naš program edini tovrstni program na svetu. Vsi problemi predstavljeni v nadaljevanju, se začnejo iz začetne postavitve figur, dovoljene pa so le legalne šahovske poteze.*

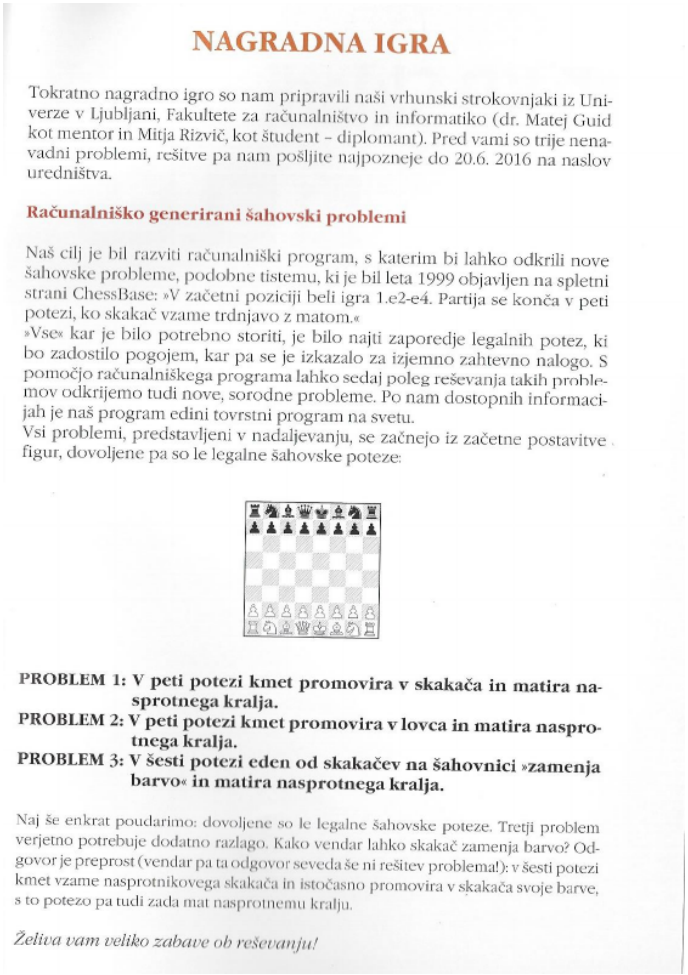
- *PROBLEM 1: V peti potezi kmet promovira v skakača in matira nasprotnega kralja,*

- *PROBLEM 2: V peti potezi kmet promovira v lovca in matira nasprotnega kralja,*

- *PROBLEM 3: V šesti potezi eden od skakačev "zamenja barvo" in matira nasprotnega kralja.*

*Naj še enkrat poudarimo: dovoljene so le legalne šahovske poteze. Tretji problem verjetno potrebuje dodatno razlago. Kako vendar lahko skakač zamenja barvo? Odgovor je preprost (vendar pa ta odgovor seveda še ni rešitev problema!): v šesti potezi kmet vzame nasprotnikovega skakača in istočasno promovira v skakača svoje barve, s to potezo pa tudi zada mat nasprotnemu kralju.*

*Želiva vam veliko zabave pri reševanju!*



## NAGRADNA IGRA

Tokratno nagradno igro so nam pripravili naši vrhunski strokovnjaki iz Univerze v Ljubljani, Fakultete za računalništvo in informatiko (dr. Matej Guid kot mentor in Mitja Rizvič, kot študent – diplomant). Pred vami so trije nenavadni problemi, rešitve pa nam pošljite najpozneje do 20.6. 2016 na naslov uredništva.

### Računalniško generirani šahovski problemi

Naš cilj je bil razviti računalniški program, s katerim bi lahko odkrili nove šahovske probleme, podobne tistemu, ki je bil leta 1999 objavljen na spletni strani ChessBase: »V začetni poziciji beli igra 1.e2-e4. Partija se konča v peti potezi, ko skakač vzame trdnjavo z matom.«
»Vse« kar je bilo potrebno storiti, je bilo najti zaporedje legalnih potez, ki bo zadostilo pogojem, kar pa se je izkazalo za izjemno zahtevno nalogo. S pomočjo računalniškega programa lahko sedaj poleg reševanja takih problemov odkrijemo tudi nove, sorodne probleme. Po nam dostopnih informacijah je naš program edini tovrstni program na svetu.
Vsi problemi, predstavljeni v nadaljevanju, se začnejo iz začetne postavitve figur, dovoljene pa so le legalne šahovske poteze:

**PROBLEM 1: V peti potezi kmet promovira v skakača in matira nasprotnega kralja.**
**PROBLEM 2: V peti potezi kmet promovira v lovca in matira nasprotnega kralja.**
**PROBLEM 3: V šesti potezi eden od skakačev na šahovnici »zamenja barvo« in matira nasprotnega kralja.**

Naj še enkrat poudarimo: dovoljene so le legalne šahovske poteze. Tretji problem verjetno potrebuje dodatno razlago. Kako vendar lahko skakač zamenja barvo? Odgovor je preprost (vendar pa ta odgovor seveda še ni rešitev problema!): v šesti potezi kmet vzame nasprotnikovega skakača in istočasno promovira v skakača svoje barve, s to potezo pa tudi zada mat nasprotnemu kralju.

*Želiva vam veliko zabave ob reševanju!*

Slika B.1: Slika prispevka objavljenega v reviji Šahovska misel.