Sveučilište u Zagrebu
Prirodoslovno - matematički fakultet
Matematički odsjek

Umjetna inteligencija - projektni zadatak

Osam kraljica, skakači, generalizacije

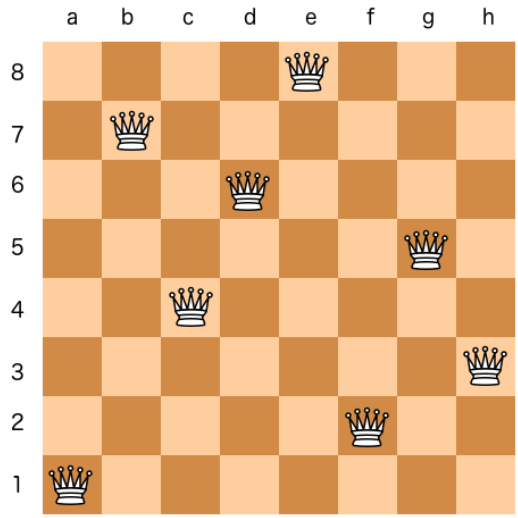
Roberto Grabovac

Zagreb, prosinac 2022.

Sadržaj

1 Uvod

Problem *Osam kraljica* izložio je njemački šahist Max Bezzel 1848. godine, a definirao ga je na sljedeći način: potrebno je smjestiti osam kraljica na šahovsku ploču dimenzije 8×8 tako da se niti jedan par kraljica ne napada. Prva rješenja ovog problema objavio je Franz Nauck 1850. godine te je uz to predložio generalizirani problem: potrebno je smjestiti n kraljica na šahovsku ploču dimenzije $n \times n$ tako da se niti jedan par kraljica ne napada. Naravno, ubrzo su se brojni matematičari pozabavili rješavanjem tog zadatka, uključujući i sveprisutnog Carl Friedrich Gaussa. Jedno od predloženih matematičkih rješenja bilo je korištenje determinanti, no 1972. godine Edsger Dijkstra je odlučio na tom primjeru prikazati moć strukturiranog programiranja, odnosno pretraživanja u dubinu.



Slika 1: Prikaz jednog mogućeg rješenja
(slika je preuzeta iz [6])

2 Pojedinosti problema

Prije samog rješenja, promotrit ćemo nekoliko važnih značajki - postojanje rješenja, asimptotsko ponašanje te pojam fundamentalnog rješenja. Prethodno ćemo razraditi na polaznom i generaliziranom problemu.

2.1 Postojanost rješenja

Zbog načina djelovanja kraljice u šahu, jasno je da nećemo izgubiti niti jedno rješenje ako svaku kraljicu postavimo u zaseban redak. Upravo će to predstavljati svojevrsnu optimizaciju pretraživanja u dubinu koje će biti izloženo u nastavku. U skladu s tim, možemo zaključiti kako postoji 8^8, odnosno n^n različitih položaja kraljica na šahovskoj ploči. Stoga primjenom *brute force* tehnike programiranja (generirati sve moguće položaje kraljica te odabrati onaj raspored u kojem se niti jedan par kraljica ne napada) nećemo efikasno riješiti problem jer već za $n = 8$ imamo 16777216 različitih položaja kraljica. Naravno, znamo da dvije kraljice ne mogu biti niti u istom stupcu pa bi stoga prostor rješenja bio 8!, odnosno $n!$. Kako god, *brute force* tehnika ne dolazi u obzir jer je funkcija složenosti u tom slučaju nadeksponencijalnog rasta.
Rješenje ovog problema postoji za sve $n \in \mathbb{N}$, osim za $n \in \{2, 3\}$. Za $n = 2$ je očito da rješenje ne postoji, a za $n = 3$ također. Slika ispod prikazuje nemogućnost postavljanja treće kraljice. Isti ishod se dobiva i variranjem položaja prve i druge kraljice.

Slika 2: Nepostojanje rješenja
(slika je preuzeta iz [7])

2.2 Asimptotsko ponašanje rješenja

U prethodnom odjeljku vidjeli smo da rješenje postoji za gotovo sve prirodne brojeve n. No, postojanje nekog rješenja je posve različit problem od otkrivanja svih mogućih rješenja. Točan broj rješenja poznat je za sve $n \leq 27$, dok $(0.143 \times n)^n$ predstavlja asimptotski rast broja rješenja.

2.3 Fundamentalno rješenje

U svrhu objašnjavanja pojma fundamentalnog rješnja, izostavit ćemo generalizirani te se fokusirati na radni problem osam kraljica. U nastavku ćemo pokazati da postoje 92 različita rješenja, no neka od njih su ekvivalentna. Smatramo da su dva rješenja ekvivalentna ako jedno možemo dobiti iz drugog primjenom rotacije za $0°, 90°, 180°, 270°$ i/ili osne simetrije. Predstavnik pojedine klase ekvivalencije će stoga predstavljati jedno fundamentalno

rješenje. Može se pokazati da za naš radni problem postoji 12 fundamentalnih rješenja.

3 Rješenje problema - pretraživanje u dubinu

Uvidom u broj rješenja, lako smo došli do zaključka da *brute force* tehnika nije efikasna te nam po tom pitanju treba nešto bolje - pretraživanje u dubinu. Implementacija je napravljena u programskom jeziku C++, a kako su autoru jako drage rekurzije, koristit ćemo rekurzivnu verziju DFS-a. U nastavku će cjelokupan kôd biti objašnjen postepeno, dio po dio.

3.1 Inicijalizacija

Koristit ćemo dvodimenzionalno polje *ploca* koje predstavlja stanje šahovske ploče željene dimenzije N, dok ćemo sve njegove elemente postaviti na 0. Nadalje, kako bismo zaustavili rekurziju, bit će nam potrebna (globalna) varijabla *br_kraljica*, a da bismo pamtili broj rješenja, koristit ćemo (globalnu) varijablu *br_rjesenja*. Pitanje je kako razaznati koje je polje na šahovskoj ploči slobodno, a koje nije. Odgovor je jednostavan: ako $ploca[i][j] = 1$, onda je ta pozicija zauzeta, odnosno na njoj se nalazi kraljica. Inače, tj. ako $ploca[i][j] = 0$, pripadna pozicija je slobodna te smo u mogućnosti na nju postaviti kraljicu.

```
int** ploca;
int N, i, j;
cout << "Unesite broj kraljica > "; cin >> N;
ploca = new int*[N];
for (i = 0; i < N; i++)
    ploca[i] = new int[N];

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        ploca[i][j] = 0;
```

Slika 3: Inicijalizacija dvodim. polja *ploca* u main-u

```
#include <iostream>

using namespace std;

int br_rjesenja;
int br_kraljica;
```

Slika 4: Globalne varijable

3.2 Pomoćna funkcija

Prema prethodno navedenom, kraljicu možemo postaviti na poziciju (i, j) ako $ploca[i][j] = 0$. No, problem je što to mjesto, iako slobodno, možda nije sigurno. Naime, može se dogoditi da kraljica koja je već postavljena na neko

drugo mjesto može napasti poziciju (i, j). Znamo da kraljica napada sve pozicije u njezinom stupcu i retku, ali i sve one koje se nalaze na jednoj od (maksimalno) četiri dijagonale koje se sijeku točno u poziciji na kojoj se nalazi odgovarajuća kraljica. Sada smo dobili konačan kriterij postavljanja kraljice na poziciju (i, j) - pozicija (i, j) je sigurna ako i samo ako $ploce[i][j] = 0$ te niti jedna od trenutno postavljenih kraljica ne napada tu poziciju. Argumenti funkcije *provjeri* su redom *ploca*, veličina ploče N i pozicija (x, y) za koju ispitujemo je li sigurna. Provjera nalazi li se (x, y) u retku, odnosno stupcu već postavljenih kraljica je trivijalna. Pitanje je kako provjeriti nalazi li se na jednoj od četiri dijagonale neke postavljene kraljice. No, pogledom na vezu indeksa i dijagonala u matrici, lako vidimo da to vrijedi ako $|i - x| = |j - y|$ gdje je (i, j) lokacija postavljene kraljice. Funkcija vraća 1 ako je pozicija sigurna, a 0 inače.

```
int provjeri(int **ploca, int N, int x, int y) {
    int i, j;
    for (i = 0; i < N; i++)
        if (ploca[x][i] == 1)
            return 0;

    for (i = 0; i < N; i++)
        if (ploca[i][y] == 1)
            return 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            if (ploca[i][j] == 1 && (aps(i - x) == aps(j - y)))
                return 0;

    return 1;
}
```

Slika 5: Pomoćna funkcija *provjeri*

3.3 Pretraživanje u dubinu

Zanemarimo na kratko bazni slučaj, odnosno uvjet $if(br\_kraljica == N)$ te promotrimo dvije ugniježđene *for* petlje. Njihova svrha je jednostavna: prolazi čitavom šahovskom pločom, ako je moguće postavi kraljicu na odgovarajuću poziciju i ponovno pozovi istu funkciju. Uočimo kako funkcija *dfs_N_queens* u argumentu dobiva cijeli (točnije nenegativan) broj *row*. On upravo određuje početak prve *for* petlje, odnosno skraćuje vrijeme izvođenja algoritma jer znamo da nova kraljica nikako ne može biti u istom retku kao prethodno postavljena. Zbog toga pri rekurzivnom pozivanju umjesto *row*,

šaljemo $row + 1$. Nadalje, ako smo zaključili da je odabrana pozicija sigurna, onda stavimo $ploca[i][j] = 1$ što, prisjetimo se, označava da je postavljena kraljica na poziciji (i, j). Potom povećamo varijablu $br\_kraljica$ za 1 jer ona označava koliko je trenutno postavljeno kraljica na šahovskoj ploči. Kada smo gotovi? Upravo onda kada $br\_kraljica = N$ te upravo to detektira uvjet kojeg smo na početku zanemarili. Ako je uvjet ispunjen, ispisat ćemo stanje na ploči, odnosno rješenje u kojem q predstavlja kraljicu, dok . označava da pripadna pozicija nije ispunjena te varijablu $br\_rjesenja$ povećati za 1. Jednom kada smo došli do linije 54, ispitali smo sva moguća rješenja ako je kraljica postavljena na poziciju (i, j) pa sada želimo isto napraviti za sljedeću sigurnu poziciju na šahovskoj ploči (ovdje uočavamo karakteristiku zbog koje ovaj algoritam doista predstavlja rekurzivno pretraživanje u dubinu). No, prije toga je potrebno šahovsku ploču i ukupan broj kraljica vratiti u prethodno stanje, a to upravo činimo u linijama 54 i 55.

```
33  void dfs_N_queens(int **ploca, int N, int row) {
34      int i, j;
35      if (br_kraljica == N) {
36          for (i = 0; i < N; i++) {
37              for (j = 0; j < N; j++)
38                  if (ploca[i][j] == 1) cout << "q ";
39                  else cout << ". ";
40              cout << endl;
41          }
42          br_rjesenja++;
43          cout << endl;
44          return;
45      }
46
47      for (i = row; i < N; i++) {
48          for (j = 0; j < N; j++)
49              if (ploca[i][j] == 0) {
50                  if (provjeri(ploca, N, i, j)) {
51                      ploca[i][j] = 1;
52                      br_kraljica++;
53                      dfs_N_queens(ploca, N, row + 1);
54                      br_kraljica--;
55                      ploca[i][j] = 0;
56                  }
57              }
58      }
59
60      return;
61  }
```

Slika 6: Pretraživanje u širinu - funkcija *dfs_N_queens*

Konačno, funkciju pozivamo s *dfs_N_queens(ploca, N, 0)*[1] koja ispisuje sva moguća rješenja problema N kraljica.

U nastavku je priložena tablica koja prikazuje broj svih, ali i fundamentalnih rješenja za neke $n \in \mathbb{N}$:

Broj kraljica	Broj fundamentalnih rješenja	Broj svih rješenja
1	1	1
2	0	0
3	0	0
4	1	2
5	2	10
6	1	4
7	6	40
8	12	92
9	46	352
10	92	724
11	341	2680
12	1787	14200
13	9233	73712
14	45752	365596
15	285053	2279184
16	1846955	14772512
17	11977939	95815104

4 Rješenje problema - *hill climbing*

Hill climbing ili algoritam uspona na vrh predstavlja heurističko pretraživanje čija učinkovitost uvelike ovisi o izboru heurističke funkcije. Radi jednostavnosti, u nastavku ćemo opisati rješenje za osam kraljica, dok implementacija algoritma podržava i generalizirani problem N kraljica. Algoritam[2] vraća (ispisuje) jednu valjanu konfiguraciju kraljica, no prije samog prelaska na implementaciju, potrebno je definirati heurističku funkciju i pojmove vezane uz algoritam.

[1] datoteka *dfs_queens.cpp*
[2] datoteka *hill_climbing.cpp*

4.1 Heuristička funkcija, ključni pojmovi

Definiramo heuristiku **h** = broj kraljica koje se napadaju. Dakle, ciljno stanje bit će trenutak kada h = 0 nakon kojeg algoritam ispisuje pripadnu konfiguraciju kraljica i završava s radom. **Stanje** će biti bilo koja konfiguracija 8 kraljica, a kako bismo reducirali prostor rješenja, dodat ćemo uvjet da u jednom stupcu može biti samo jedna kraljica. Za prikaz stanja, koristit ćemo jednodimenzionalno polje *stanje* gdje $stanje[i] = j$ govori da imamo kraljicu u i-tom stupcu i j-tom retku. **Susjedno stanje** (susjedstvo) bit će stanje koje se od postojećeg stanja razlikuje samo po poziciji jedne kraljice (ta kraljica može biti postavljena bilo gdje u svojem stupcu). Dakle, svako stanje na 8×8 ploči imat će 56 susjednih stanja.

4.2 Opis algoritma

Algoritam možemo jednostavno opisati u sljedećih 5 koraka:

1. započni s proizvoljnim (nasumično odabranim) stanjem, tj. konfiguracijom kraljica

2. pregledaj sve moguće susjede stanja i prijeđi u ono kod kojega je heuristička funkcija najmanja. Ako je heuristička funkcija u svim susjednim stanjima veća ili jednaka od vrijednosti u postojećem stanju, nasumično odaberi susjedno stanje

3. ponavljaj korak 2 sve dok se vrijednost heurističke funkcije ne smanji

4. pronađeno stanje je ili lokalni ili globalni optimum. Ako je riječ o lokalnom optimumu, ponovno nasumično odabiremo susjedno stanje (korak 2) ili pak krećemo ispočetka u algoritmu s novom konfiguracijom (korak 1)

5. ispiši valjanu konfiguraciju

Dakle, cilj je prelaziti iz jednog stanja u susjedno tako da optimiziramo funkciju cilja h, odnosno da pri svakom prijelazu smanjujemo njezinu vrijednost. Kao što je navedeno, može se dogoditi da zaglavimo u lokalnim optimumima što rješavamo nasumično odabranim susjedstvom ili potpuno novom konfiguracijom. Kako ovaj problem nema velik broj lokalnih optimuma, koristit ćemo nasumično odabrano susjedstvo jer je ta tehnika puno brža nego restart algoritma.

4.3 Pomoćne funkcije

Implementacija algoritma uspona na vrh je poprilično jednostavna i lako razumljiva zbog velikog broja pomoćnih funkcija. Naravno, imena funkcija ukratko opisuju njihov "posao" kako bi glavna funkcija *hill_climbing* bila što razumljivija. Kao što je već prethodno navedeno, implementiran je velik broj pomoćnih funkcija pa ćemo samo navesti njihov popis i ono što rade, dok su detalji njihove implementacije obrazloženi u obliku komentara u samom kôdu.

Popis pomoćnih funkcija i njihov "posao":

- ***nasumicnaKonfiguracija*** : kreira proizvoljan raspored kraljica na ploči te služi za korak 1 u algoritmu.
- ***ispisiPlocu*** : ispisuje ploču nakon zaustavljanja algoritma.
- ***usporediStanja*** : uspoređuje dva stanja (vidi 4.1) te vraća 1 ako su jednaka, a 0 inače.
- ***ispuniPlocu*** : ispunit će dvodimenzionalno polje koje reprezentira šahovsku ploču varijablom *vrijednost* koja joj je proslijeđena u argumentu (u algoritmu je u svakom njezinom pozivu *vrijednost* = 0).
- ***h*** : heuristička funkcija koja vraća broj napada između postavljenih kraljica na šahovskoj ploči. Postoji mnoštvo načina za njezinu implementaciju, no većina njih prati već izloženu vezu indeksa matrice i četiri dijagonale iz kraljice (funkcija *provjeri* u algoritmu pretraživanja u dubinu).
- ***kreirajZadanuPlocu*** : kreira šahovsku ploču tako da raspored kraljica odgovara varijabli *stanje* koja se prosljeđuje kao argument funkcije.
- ***kopirajStanja*** : kopira jedno stanje u drugo.
- ***opt_susjedstvo*** : traži optimalno susjedstvo, odnosno susjedno stanje. Njezina implementacija je zapravo (uz korištenje prethodno navedenih funkcija) ekvivalentna implementaciji pronalaženja minimalnog elementa u polju brojeva.

4.4 *Hill climbing*

U nastavku je priložena slika 7 na kojoj se vidi implementacija glavne funkcije *hill_climbing*. Možemo uočiti kako imamo jednu *while* petlju koja se izvršava beskonačno mnogo puta, osim ako se ne prekine nekim break-om. Upravo

se to događa ako je zadovoljen prvi *if* uvjet, odnosno došli smo do stanja koje je globalno optimalno na skupu svih susjednih stanja. No, to ne znači nužno da je to stanje dobro jer ovaj algoritam nije optimalan. Upravo tome služi provjera u liniji 189, dok se inače ispisuje stanje na šahovskoj ploči jer je u tom slučaju konfiguracija kraljica valjana. Autora je zanimalo u koliko slučajeva ovaj algoritam vraća optimalno rješenje, odnosno valjan raspored kraljica na šahovskoj ploči. Nakon što je program pokrenut ukupno 100 puta, utvrđena je efikasnost od 95 %.

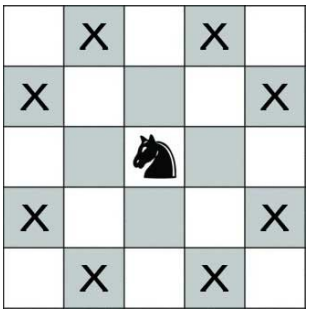
```
176  void hill_climbing(int ploca[][N], int* stanje) {
177      int susjednaPloca[N][N];
178      int susjednoStanje[N];
179      kopirajStanja(susjednoStanje, stanje);
180      kreirajZadanuPlocu(susjednaPloca, susjednoStanje);
181      while (1) {
182          kopirajStanja(stanje, susjednoStanje);
183          kreirajZadanuPlocu(ploca, stanje);
184
185          opt_susjedstvo(susjednaPloca, susjednoStanje);
186          //ako je optimalno susjedno stanje jednako trenutnom stanju
187          if (usporediStanja(stanje, susjednoStanje)) {
188              int fja_h = h(susjednaPloca, susjednoStanje);
189              if (fja_h > 0)
190                  cout << "Nismo pronasli rjesenje!" << endl;
191              else
192                  ispisiPlocu(ploca);
193              break;
194          } //ako else if prodje, to znaci da smo zapeli u lokalnom optimumu ili "ramenu"
195          else if (h(ploca, stanje) == h(susjednaPloca, susjednoStanje)) {
196              //rekli smo da cemo za bjezanje iz lokalnog optimuma koristiti
197              //nasumicno odabrano susjedstvo
198              susjednoStanje[rand() % N] = rand() % N;
199              kreirajZadanuPlocu(susjednaPloca, susjednoStanje);
200          }
201      }
202  }
```

Slika 7: Algoritam uspona na vrh - funkcija *hill_climbing*

Konačno, u funkciji main prvo definiramo dvodimenzionalno polje $ploca[N][N]$ koje reprezentira šahovsku ploču i jednodimenzionalno polje $stanje[N]$ čiju smo svrhu već spomenuli nekoliko puta (vidi 4.1). Potom kraljice proizvoljno (s dodatnim uvjetom da je u svakom stupcu točno jedna kraljica) raspo-redimo na šahovskoj ploči pozivom pomoćne funkcije *nasumicnaKonfigura-cija(ploca, stanje)*. Sada možemo pozvati glavnu funkciju *hill_climbing* s *hill_climbing(ploca, stanje)* te u velikoj većini slučajeva dobiti rješenje.

5 Dodatak - skakači

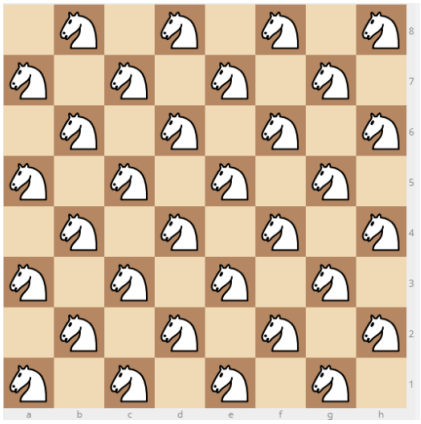
Postoji niz problema koji se tiču rasporeda šahovskih figura pod određenim uvjetima. Jedan takav jest raspored skakača na šahovskoj ploči tako da se niti jedan par međusobno ne napada. Dakle, problem je analogan problemu osam kraljica pa će u skladu s tim njegova analiza biti znatno kraća kako bi se izbjeglo ponavljanje već rečenih stvari. Jasno, jedina razlika proizlazi iz njihovih kretnji. Zbog toga ne možemo iskoristiti pretpostavku da se u jednom retku i stupcu mora nalaziti točno jedan skakač, kao što je to bio slučaj kod kraljica. Samim time ne možemo suziti prostor rješenja što utječe na vrijeme izvršavanja algoritma. Dosad smo detaljno analizirali pretraživanje u dubinu i algoritam uspona na vrh. Ovdje ćemo ponovno izložiti algoritam pretraživanja u dubinu. No, implementacija je praktički identična onoj u problemu osam kraljica. Jedina ključna razlika je jasno u funkciji *provjeri*, no ona i dalje ima istu zadaću - provjerava možemo li postaviti skakača na poziciju (x, y). Autor namjerno nije htio smišljati novu implementaciju kako bi pokazao da se svi problemi tipa "postavi šahovske figure tako da se nikoje dvije međusobno ne napadaju" mogu potpuno analogno riješiti. Kako bismo ipak vidjeli nešto novo, omogućit ćemo unos različitih dimenzija ploče i proizvoljnog broja skakača koje na nju treba postaviti te ćemo promotriti vrijeme izvršavanja algoritma za neke od parametara.



Slika 8: Osam različitih kretnji skakača
(slika je preuzeta iz [8])

Primjer 1. *Uzmimo standardnu šahovsku ploču dimenzija 8×8. Zanimljivo pitanje je koliko maksimalno možemo postaviti skakača tako da se niti jedan par međusobno ne napada. Pogledamo li sliku 9, uočavamo da skakač može napasti samo ona polja koja su bojom različita od onog na kojem se on nalazi. Tako lako dolazimo do jednog mogućeg rješenja koje uključuje 32 skakača - odaberemo proizvoljnu boju šahovske ploče i na polja obojana tom bojom postavimo skakače. Ispostavlja se da je taj broj upravo i maksimalan broj skakača koje možemo postaviti na šahovsku ploču te dimenzije.*

Općenito, može se dokazati da za dimenziju $N \geq 3$ vrijedi da je maksimalan broj skakača jednak $\lceil \frac{N^2}{2} \rceil$ ako je N paran, a $\lceil \frac{N^2+1}{2} \rceil$ inače. Dokaz koristi teorijske rezultate problema Knight's tour koji izlazi van okvira ovog rada pa dokaz nećemo provoditi. Više o tome može se pronaći na [9].



Slika 9: Rješenje s 32 skakača
(slika je preuzeta iz [10])

5.1 Implementacija

Implementacija[3] je napravljena u programskom jeziku C++. Kao što je već navedeno, implementacija je potpuno analogna onoj u problemu osam kraljica (algoritam pretraživanja u dubinu), a glavne promjene vidljive su tek u nekoliko kozmetičkih detalja i u funkciji *provjeri*. Zbog toga će opis onoga što je ostalo isto (analogno) biti kraći.

5.1.1 Globalne varijable, inicijalizacija

Korisnik na početku programa učitava dimenziju šahovske ploče (N) i broj skakača koje treba postaviti (globalna varijabla *broj_skakaca*). Globalna varijabla *br_rjesenja* predstavlja ukupan broj rješenja za odgovarajuću dimenziju ploče i broj skakača, dok varijabla *ukupno_skakaca* ima vrijednost ukupnog broja skakača postavljenih na šahovsku ploču u određenom trenutku te služi kao uvjet za zaustavljanje u glavnoj funkciji *dfs_knights*. Šahovska ploča reprezentirana je dvodimenzionalnim poljem *polje*, tj. na potpuno isti način kao i prije.

[3] datoteka *dfs_knights.cpp*

5.1.2 Pomoćna funkcija

Koristimo već spomenutu pomoćnu funkciju *provjeri* koja provjerava možemo li skakača postaviti na poziciju (x, y). Pogledamo li sliku 9, vidimo da polje (x, y) može biti napadnuto s točno 8 pozicija te stoga imamo toliko if uvjeta u samoj funkciji koji naprosto ispituju sve kretnje skakača.

```
int provjeri(int **ploca, int N, int x, int y) {
    if ((x + 2) < N && (y - 1) >= 0 && ploca[x + 2][y - 1] == 1)
        return 0;
    if ((x - 2) >= 0 && (y - 1) >= 0 && ploca[x - 2][y - 1] == 1)
        return 0;
    if ((x + 2) < N && (y + 1) < N && ploca[x + 2][y + 1] == 1)
        return 0;
    if ((x - 2) >= 0 && (y + 1) < N && ploca[x - 2][y + 1] == 1)
        return 0;
    if ((x + 1) < N && (y + 2) < N && ploca[x + 1][y + 2] == 1)
        return 0;
    if ((x - 1) >= 0 && (y + 2) < N && ploca[x - 1][y + 2] == 1)
        return 0;
    if ((x + 1) < N && (y - 2) >= 0 && ploca[x + 1][y - 2] == 1)
        return 0;
    if ((x - 1) >= 0 && (y - 2) >= 0 && ploca[x - 1][y - 2] == 1)
        return 0;

    return 1;
}
```

Slika 10: Pomoćna funkcija *provjeri*

5.1.3 Pretraživanje u dubinu

Rekurzivnu verziju pretraživanja u dubinu pokrećemo pozivom funkcije $dfs\_knights$. Ona se od funkcije dfs_N_queens razlikuje samo po tome što ima jedan parametar više. Taj parametar je *col* koji predstavlja početak prve for petlje u idućem rekurzivnom pozivu funkcije. Ta varijabla, zajedno s *row* služi za ubrzanje algoritma i sprječavanje ponavljanja rješenja.

5.1.4 Mjerenje vremena

U glavnoj *main* funkciji korišten je mjerač vremena izvršavanja funkcije $dfs\_knights$. Da bismo doista mogli izmjeriti vrijeme, potrebno je uključiti *std* :: *chrono* biblioteku u kojoj možemo pronaći tri različite vrste sata. Odabran je *high_resolution_clock* koji je najpreciznji, a izmjereno vrijeme može biti izraženo u proizvoljnim mjernim jedinicama. Jednom kada smo uključili tu biblioteku, u varijable *start* i *stop* upisujemo vrijeme neposredno prije, odnosno nakon poziva funkcije $dfs\_knights$. Konačno, razlika tih dvaju

vremena predstavlja trajanje algoritma pretraživanja u dubinu izraženo u sekundama.

5.2 Rezultati mjerenja vremena izvršavanja

Testiranja su provedena nad raznim parametrima na prijenosnom računalu s 16 GB RAM-a i procesorom *11th Gen Intel(R) Core(TM) i7-1165G7, 2.80GHz*.
Donja tablica sadrži broj rješenja i vrijeme izvršavanja u ovisnosti o uređenom paru (N, k) gdje N predstavlja dimenziju šahovske ploče, a k broj skakača. Vrijeme će biti izraženo u mikrosekundama, sekundama ili minutama, ovisno o zahtjevnosti uređenog para.

(Dimenzija šahovske ploče, broj skakača)	Broj rješenja	Vrijeme izvršavanja
(3,3)	36	139217 μs
(3,5)	2	6476 μs
(4,4)	412	1073634 μs
(4,8)	6	28883 μs
(5,5)	9386	79.24 s
(5,8)	8526	62 s
(5,13)	1	13379 μs
(6, 4)	26133	102 s
(6,6)	257318	25.48 min
(6,15)	2560	10 s
(6,18)	2	199226 μs

Već na ulazu $(6, 6)$ uočavamo kako je potrebna velika količina vremena da se dođe do svih rješenja. Zanimljiv je klasičan problem postavljanja 32 skakača na šahovskoj ploči dimenzije 8×8. Lako se može naslutiti da je potrebna ogromna količina vremena za obuhvaćanje cijelog prostora rješenja. Naravno, algoritam je ispitan na tom primjeru, ali jasno da je njegovo izvršavanje prekinuo autor jer bi bili potrebni dani, a možda i mjeseci, da se pronađu sva rješenja. Radi stjecanja dojma o potrebnom vremenu, da bi se pronašlo jedno jedino rješenje potrebno je 6.3 minute.

6 Zaključak

Problem osam kraljica ili pak njegova generalizacija na N kraljica predstavlja jako dobar primjer toga da se isplati stati, razmisliti i konstruirati kvalitetno pretraživanje ili heurističku funkciju, nego krenuti *brute force* tehnikom programiranja koja možda može proći za prvih par prirodnih brojeva, no takav algoritam se u asimptotskoj analizi uopće ne broji kao jedno moguće rješenje ovog problema zbog izrazite neefikasnosti. *Depth first search* ili pretraživanje u dubinu koje je korišteno kao prvo rješenje ovog problema predstavlja vrlo smislen i prirodan pristup ovom problemu. S druge strane, *hill climbing* ili algoritam uspona na vrh je općenito dosta zanimljiv algoritam jer je potrebno dosjetiti se efikasne heuristike, a sve njegove karakteristike mogli smo vidjeti u ovom problemu. Iako je kôd nešto duži, algoritam je poprilično razumljiv zbog jednostavne ideje nastale iz prirodnog odabira heurističke funkcije. Vrijedi napomenuti da ima raznih varijanti definirane heuristike, a uspješnost od 95 % odabrane varijante je poprilično zadovoljavajuća za ovaj tip problema. Na kraju smo vidjeli sličnu varijantu problema u kojem je potrebno rasporediti skakače, umjesto kraljica, tako da se niti jedan par međusobno ne napada. Kako je sama formulacija zadaće analogna problemu osam kraljica, jasno da će im i rješenja biti u suštini identična. To smo mogli uočiti u jako jednostavnoj modifikaciji koja nam je bila potrebna da kôd za raspored kraljica u potpunosti prilagodimo rasporedu skakača na šahovskoj ploči. Time je ujedno i utvrđeno razmišljanje kod ovakvih tipova problema - sve što treba napraviti jest prilagoditi funkciju provjere postavljanja na mjesto (x, y) šahovske ploče u ovisnosti o kretnjama figure zadane u problemu.

Literatura

[1] Eight queen puzzle, https://en.wikipedia.org/wiki/Eight_queens_puzzle#cite_note-rouse_ball_1960-1, 10.12.2022.

[2] 8-Queen Problem, https://www.cukashmir.ac.in/cukashmir/User_Files/imagefile/DIT/StudyMaterial/DAA/DAA_UNIT-III_6th-Sem_StudyMaterial.pdf, 10.12.2022.

[3] Hill Climbing, https://www.youtube.com/watch?v=vEpPMIiTSDI&ab_channel=FranciscoIacobelli, 10.12.2022.

[4] Introduction to Hill Climbing, https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/?fbclid=IwAR36DjscOyS1NXL05FHOPzFxBkqntuIe-Et-hHBiJOOm7IK9Lx_O0s_160g, 11.12.2022.

[5] N-Queen Problem (Local Search using Hill climbing with random neighbour), https://www.geeksforgeeks.org/n-queen-problem-local-search-using-hill-climbing-with-random-neighbour/?fbclid=IwAR0nOy5eZdwC4czUyjNGxLn-IyjfMCcvdxAjQzj4JsAVRcmS-80v_v-muAo, 11.10.2022.

[6] Genetic Algorithm: 8 Queens Problem, https://medium.com/nerd-for-tech/genetic-algorithm-8-queens-problem-b01730e673fd, 10.12.2022.

[7] N-Queens: Describing the Problem, https://medium.com/@jtfeliciano/n-queens-problem-describing-the-problem-12dbd5c1508e, 10.12.2022.

[8] Possible chess knights movements, https://diatomenterprises.com/ruby-and-recursion-find-out-all-possible-chess-knights-movements-using-minimax-algorithm/, 13.12.2022.

[9] Knight's tour, https://en.wikipedia.org/wiki/Knight%27s_tour, 13.12.2022.

[10] Maximum number of knights, https://www.quora.com/What-is-the-maximum-number-of-knights-that-can-be-placed-on-a-chess-board-such-that-no-knight-attacks-another, 13.12.2022.