Univerzitet Singidunum

Veštačaka inteligencija

Problem osam kraljica

dr. Milan M. Milosavljević

Anđelković Miloš 2016/203282

Beograd 2017.

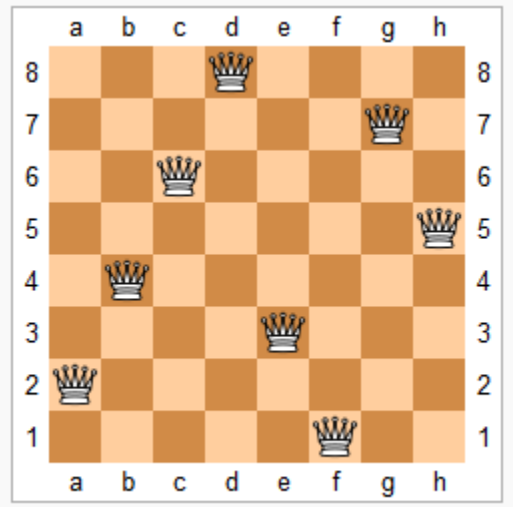
Sadržaj

1 Uvod

Problem osam kraljica predstavlja problem raspoređivanja sam kraljica na šahovsku tablu tako da ni jedna kraljica ne napada drugu. Ovo znači da dve kraljice ne mogu da se nađu u isom redu, koloni ili dijagonali. Primer osam kraljica je u stvari primer problema n kraljica, odnosno raspoređivanja n kraljica na nxn tablu tako da svi navedeni uslovi budu ispunjeni.

Maks Bazel je 1848. objavio zagonetku osam kraljica a prvo rešenje je objavio Frans Nauk 1850. Nauk je takođe proširio zagonetku na problem n kraljica. Od tada, mnogi matematečari, uključujući i Karla Frederika Gausa, su radili na problemu osam kraljica i generalizovanu verziju n kraljica. 1874. S. Gunter je predložio metod za pronalaženje rešenja koji je J. V. L. Glešer preradio.

1972. Edsger Dijkstra je koristio ovaj problem za ilustraciju moći strukturnog programiranja. Objavio je vrlo detaljan opis backtracking algoritma u dubinu.



Slika 1.1.:*Jedno od rešenja problema osam kraljica*

2 Razvoj rešenja

S obzirom na to da postoji 4 426 165 3668 mogućih rasporeda osam kraljica na tabli veličine 8x8, od kojih je samo 92 tačno, ovaj problem može da bude veoma zahtevan. Međutim uvođenjem prostih pravila kao što je da u jednoj koloni ili redu može da se nalazi samo jedna kraljica broj mogućih kombinacija se svodi na 16 777 216. Generisanje premutacija ovih rasporeda dalje smanjuje broj mogućnosti na 40 320 koje se dalje proveravaju za dijagonalen napada.

Martin Ričards je objavio program koji računa broj rešenja problema n kraljica korišćenjem operacija na nivou bita.

Problem osam kraljica ima 92 rešenja. Međutim ako se rešenja koja se razlikuju samo u simetriji, odnosno rotiranjem table može se od jednog dobiti drugo, računaju kao jedno, onda ovaj problem ima 12 rešenja. Ova rešenja se nazivaju fundamentalna rešenja.

Svako fundamentalno rešenje u glavnom ima osam variacija koje se dobijaju rotiranjem za 90, 180 ili 270 stepeni a zatim reflektovanjem svake od variacija u ogledalu koje se nalazi na fiksnoj poziciji. U koliko je rešenje isto kao i njegva rotacija od 90 stepeni, što je slučaj kod 5 kraljica, takvo rešenje ima samo četiri variacije. Takođe u koliko je rešenje jednako svojoj rotaciji od 180 stepeni ali ne i rotaciji od 90, imaće četiri variacije. U koliko je broj kraljica veći od jedan nije moguće da rešenje bude ekvivalentno sa svojom refleksijom jer bi to značilo da se dve kraljice nalaze u istoj koloni. U slučaju problema osam kraljica, od dvanaest fundamentalnih rešenja, jedno je jednako svojoj rotaciji od 180 stepeni, i ni jedno nije jednako svojoj rotaciji od 90 stepeni. Šta znači broj rešenja je $11 \times 8 + 1 \times 4 = 92$.

Brute force algoritmi za računanje broj rešenja su računarski izvodljivi za $n = 8$ ali su neizvodljivi za probleme gde je $n \geq 20$ jer $20! = 2.433 \times 10^{18}$. U koliko je cilj da se pronađe samo jedno rešenje eksplicitna rešenja postoje za svako $n \geq 4$, i ne zahtevaju nikakvo pretraživanje. Za ovo može da se koristi sledeća formula:

- Neka je (i, j) kvadrat u koloni *i* i redu j na tabli $n \times n$, neka je *k* ceo broj.
- Ako je *n* paran broj i $n \neq 6k + 2$, onda se kraljica postavalja na $(i, 2i)$ i $\left(\frac{n}{2} + i, 2i - 1\right)$ za $i = 1,2, \dots, n/2$.
- Ako je n parana broj i $n \neq 6k$ onda se kraljice postavljaju na $\left(i, 1 + \left(2i + \frac{n}{2} - 3(mod\ n)\right)\right)$ i $\left(n + 1 - i, n - \left(2i + \frac{n}{2} - 3(mod\ n)\right)\right)$ za $i = 1,2, \dots, n/2$.
- Ako je n neparana, onda se koristi jedan od navedenih obrazaca za $n - 1$ i dodaje se kraljica na (n, n).

Postoje i drugi načini za rešavanje ovog.

2.1 Broj rešenja

U nastavku je data tabela s brojem rešenja za problem n kraljica na $n \times n$ tabli za vrednosti $n = 1,2, \ldots,10,24,25,26,27$. Naveden je broj fundamentalinih rešanja kao i svih.

n:	1	2	3	4	5	6	7	8	9	10
Fundamentalna:	1	0	0	1	2	1	6	12	46	92
Sva:	1	0	0	2	10	4	40	92	352	724

Tabela 2.1.1. *Broj rešenja za n=1,2,...,10*

n:	24	25	26	27
Fundamentalna:	28 439 272 956 934	275 986 683 743 434	2 789 712 466 510 289	29 363 791 967 678 199
Sva:	227 514 171 973 736	2 207 893 435 808 352	22 317 699 616 364 044	234 907 967 154 122 528

Tabela 2.1.2.: *Broj rešenja za n=24,25,26,27*

3 Algoritmi za rešavanje

Ovaj problem predstavlja jedan dobar primer jednostavnog ali netrivijalnog problema. Iz tog razloga se često koristi kao primerz za mnoge tehnike programiranja, uključujući i netradicionalne pristupe kao što su logično programiranje ili genetski algoritmi. Naječešće se koristi kao primer koji može da se reši rekurzivnim algoritmom, tako što se u svakom pozivu funkcije dodaje jedna kraljica na tablu.

Ova tehnika je mnogo efikasnija od naivnog algoritam za pretragu grubom silom, koja uzima u obzir svih $64^8 = 2^{48}$ = 281 474 976 710 656 mogućih kombinacija osm kraljica, a zatim ih filtrira tako da se uklone sve kombinace koje stavljaju dve kraljice ili na istom polju (ostavljajući samo 64! / 56! = 178,462,987,637,760 mogućih kombinacija) ili položaje u kojima se međsobno napadaju. Ovaj veoma loš algoritam će, između ostalog, da proizvode iste rezultate iznova i iznova u svim različitim permutacija postavljanja osam kraljica, kao i da ponavlja iste proračune iznova i iznova za različite podgrupe svaki rešenje. Bolji algoritam grubom silom stavlja jednu kraljicu na svaki red, što je dovodi do svega $8^8 = 2^{24}$ = 16 777 216 slepih rasporeda.

Moguće je da se uradi mnogo bolje od ovoga. Jedan algoritam rešava problem osam topa generisanjem permutacije brojeva 1 do 8 (kojih ima 8! = 40 320), i koristi elemente svake

permutacija kao indikaciju da postavi kraljicu u svakom redu. Onda odbacuje te rasporede na kojima se kraljice dijagonalno napadaju. Backtrack prvo u dubinu pretraživanje, sa jednim poboljšanjem u odnosu na metode permutacija, konstruiše stablo za pretraživanje uzimanjem u obzir jedanog reda tabe u jednom trenutku, eliminišući još u ranoj fazi rasporede koji ne vode do rešenja. Zato što odbacuje topa i dijagonalne napada čak i na nepotpunim rasporedima, ispituje samo 15 720 mogućih rasporeda kraljica. Dalji napredak, koji ispituje samo 5 508 mogućih rasporeda kraljica, je kombinacija metoda zasnovanog na permutacijama sa metodom ranog odbacivanja: permutacije se generišu prvo u dubinu i prostor pretraživanja je smanjen ako delimična permutacija proizvodi dijagonale napad.

Alternativa za iscrpljujuće pretrage je algoritam "iterativane prepravke", koji obično počinje sa svim kraljicama na tabli, na primer, sa jednim kraljicom po koloni. Onda prebrojava sukobe (napade), i koristi heuristiku da se utvrdi kako poboljšati postavljanje kraljice. Heuristika "Minimum-konflikata" - pomeranje kraljice sa najvećim brojem sukoba na kocku u istoj koloni, gde je broj sukoba najmanji - posebno je efikasan: nađe rešenje za problem sa 1 000 000 kraljica za manje od 50 koraka u proseku. Ovo podrazumeva da je početna konfiguracija 'razumno dobra' - ako milion kraljice sve počinje u istom redu, to će očigledno biti potrebno najmanje 999 999 koraka da to popravi. A "prilično dobara" polazna tačka može biti, na primer stavljajnje svaku kraljice u svom redu i koloni, tako da je raspored sa najmanjm brojem kraljica u sukobu već na ploči kraljice.

Treba imati na umu da "iterativan popravka", za razliku od "backtrack" pretrage, ne garantuje rešenje: kao i svi algoritami sa usponom (tj, pohlepnih), može da se zaglavi na lokalnom Optimum (u kom slučaju algoritam može biti ponovo pokrenut sa drugačijim početnim rasporedom). S druge strane, to može da reši probleme koji su nekoliko redova veličine izvan domašaja algoritama pretrage u dubinu.

4 Rešenje u programskom jeziku C#

Za izradu rešenja je korišćen backtrack algoritam u dubinu realizovan u programskom jeziku C#. Kao razvojno okruženje je korišćen VisualStudio 2015.

Korišćene su dve klase PositionNode i PositionTree a main funkcija samo pravi instancu bjekta klase PostionTree koja dalje radi sve šta treba, a zatim štampa broj rešenja i vreme potrebno za izračunavanje.

```
private class PositionNode
        {
            public char X { get; set; }
            public int Y { get; set; }
            public List<PositionNode> next { get; set; }
            public PositionNode()
            {
                next = new List<PositionNode>();
            }
         }
```

Kod 4.1.: *Klasa PositionNode*

Klas PositionNode je pomoćna kalas. Instance ove klase predstavljaju čvorove u stablu. Svaki čvor ima svoju poziciju na tabli i listu potomaka, pošto je moguće za jednu poziciju da postoji više mogućih kombinacija.

```
private PositionNode root ;
public Int64 numberOfSolutions { get; internal set;}
private int depth;
private List<PositionNode> soFar;
private StreamWriter sw;
private bool saveTree;
private bool writeToFile;
```

Kod 4.2.: *Promenljive klase PositionTree*

Deklaracija svih potrebnih promenljivih.

- root predstavlja korieni čvor od kojeg počinje izgradnja stabla.
- numberOfSolutions predstavlja broj rešenja, ovde su u pitanju sva rešenja a ne samo fundamentalana. Ovaj properti je jedini javan, s' tim da van klase može samo da se uzme njegova vrednost, za štampanje, a može da se menja samo iz unutrašnjosti klase (internal set).
- depth predstavalj dubinu stabla, odnosno broj kraljica.
- soFar je pomoćna lista koja pamti dosadašnji put u stablu. Ovo olakšava proveru.
- sw se koristi za upis kombinacaija u tekstualnu datoteku.
- saveTre je bool koji govori da li se čuva stablo ili ne.
- writeToFile je bool koji govori o tome da li se kombinacije upisuju u datoteku ili ne.

Pre nego što se pređe na glavni deo koda prvo ćemo da prođemo kroz pomoćne metode.

```
private bool check(PositionNode a, PositionNode b)
        {
            if (a.X == b.X)
            {
                return false;
            }
            if (Math.Abs(a.X - b.X) == Math.Abs(a.Y - b.Y))
            {
                return false;
            }
            return true;
         }
```

Kod 4.1.: *Metoda check*

Metoda check je metoda koja proverava da li se kraljice koje se nalaze na određenim pozicijama napadaju. Kao argumente prihvata dva PositionNod-a a kao rezultat vraća true ako nema sukoba i false ako ima. Prvo se proverava da li se kraljice nalaze u istoj koloni, a zatim se proverava dijagonala. Napada po dijagonali se detektuje tako što se poredi razlika po X i Y koordinati (ako zamislimo šahovsku tabu kao koordinatni sistem onda je kolona X koordinata a red Y).

```
private void writePath(PositionNode p)
        {
            sw.Write((numberOfSolutions) + ":{");
            foreach (PositionNode po in soFar)
            {
                sw.Write( po.X + "" + po.Y + " ; ");
            }
            sw.Write( p.X + "" + p.Y + "}" + Environment.NewLine);
        }
```

Kod 4.2.: *Metoda writePath*

Metoda writePath služi za upisivanje kombinacije u fajl. Kao argument prima PositionNode koji predstavlja poslednji čvor u putanji, onaj koji se ne nalazi na listi soFar.

```
private bool addNode(PositionNode p, int level)
        {
            for (int i = 1; i <= depth; i++)
            {
                bool flag = false;
                PositionNode temp = new PositionNode();
                temp.X = (char) (96+i);
                temp.Y = level;
                foreach (PositionNode po in soFar)
                {
                    if (!(check(po, temp)))
                    {
```

```
                flag = true;
                break;
            }
        }
        if (flag)
        {
            continue;
        }

        p.next.Add(temp);

        if (level == depth)
        {
            ++numberOfSolutions;
            if (writeToFile)
            {
                writePath(temp);
            }

            return true;
        }

        soFar.Add(temp);

        if ((addNode(temp, level + 1)) && saveTree)
        {
            soFar.RemoveAt(soFar.Count - 1);
            continue;
        }

        p.next.RemoveAt(p.next.Count - 1);

        soFar.RemoveAt(soFar.Count - 1);
    }
    if (p.next.Count == 0 && level != depth)
    {
        return false;
    }

    return true;
}
```

Kod 4.3.: *Metoda addNode*

Metoda addNode je ta u kojoj se nalazi glavan logika programa. Ona pokušava da na određeni nivo postavi kraljicu. Pozicija se generiše onoliko puta koliko je široka tabla. Prvo se proverava da li na toj pozicija postoji sukob sa nekom već postavljenom kraljicom, za šta se koristi lista soFar i metoda check. U koliko prođe ova prover pozicija se dodaje na stablo. Zatim u koliko se stiglo do dubnie stabla onda povećava broj kombinacija i vraća se true. U suprotnom čvor se dodaje na listu soFar i poziva se ponovo ova metoda i pokušava se da se doda sledeći čvor, odnosno čvor na sledećoj dubini. U koliko ne uspe da se doda čvor ili je izabrano da se ne čuva stablo čvor se uklanja iz stabal. Ova metoda se poziva rekurzivno najviše onoliko puta koliko je dubok stablo, odnosno koliki je broj kraljca.

```
public PositionTree(int numblerOfQueens, bool saveTree, bool writeToFile)
        {
            numberOfSolutions = 0;
            depth = numblerOfQueens;
            soFar = new List<PositionNode>();

            this.writeToFile = writeToFile;
            this.saveTree = saveTree;

            if (writeToFile)
            {
                sw = new StreamWriter(@"Log\Queens_" + numblerOfQueens + "_" +
DateTime.Now + ".txt");
            }

            root = new PositionNode();
            root.X = (char)96; root.Y = -1;

            for (int i = 1; i <= depth; i++)
            {
                PositionNode temp = new PositionNode();
                temp.Y = 1;
                temp.X = (char) (96+i);
                soFar = new List<PositionNode>();
                soFar.Add(temp);
                root.next.Add(temp);
                if (!(addNode(temp, 2)))
                {
                    root.next.RemoveAt(root.next.Count - 1);
                }

            }
            if (writeToFile)
            {
                sw.Close();
            }
        }
```

Kod 4.4.:*Konstruktor*

U konstrukotoru se postavljaju sve vrednosti, generiše se root koji predstavlja koren stabal, što je neki nepostojeći čvor. Generišu se mogući čvorovi za prvi red table i pokušavaju da se dodaju ostali.

5 Analiza rešenja

Dva glavna kriterijum pri analizi rešnja jesu vreme potrebno za pronalaženje rešenja i količna iskorišćene memorije.

Kako ovo nije optimizovan algoritam vreme potrebno za pronalaženje rešenja je daleko veće od optimizovanijih verzija. Tabla vremena izvršavanja za vrednosti $n = 8,9, \dots ,15$ data je u nastavku.

n:	Vreme:
8	00:00:00.0055265
9	00:00:00.0162534
10	00:00:00.0836201
11	00:00:00.4634629
12	00:00:02.8191371
13	00:00:17.9501538
14	00:02:03.7459231
15	00:14:42.2197656

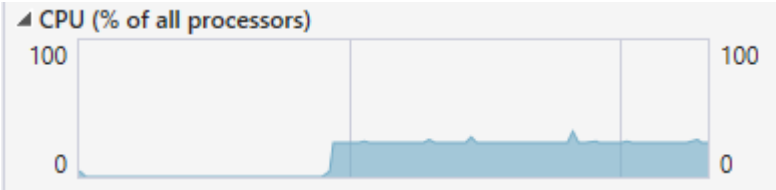
Tabela 5.1.: *Tabela vrmena izvšavanja*

Kao što se iz ove tabele vidi vreme izvršavanja eksponenicijalno raste sa porastom n. Treba imati na umu da je procesor na kojem je pokrenut ovaj progam Intel i7-5500U 2.40GHz, što čini ova vremena ogromnim (neprihvatljivim). Isti program pokrenut na Intel i7-6700HQ 2.60GHz procesoru je dao sledeće rezultate:

n:	Vreme:
8	00:00:00.0059579
9	00:00:00.0184450
10	00:00:00.0790715
11	00:00:00.4477711
12	00:00:02.6901206
13	00:00:17.136669
14	00:01:54.7540925
15	00:13:26.9627058

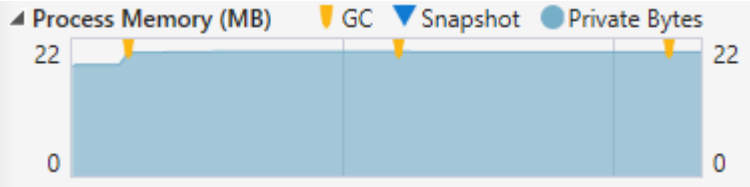
Tabela 5.2.: *Tabela vreman izvršavanja*

Kod ovog razmatranje treba takođe uzeti u obzir i iskorišćenost procesora. Kao što se iz grafika vidi, ne računiajući manje skokove, ona je veoma niska. Razlog tome je taj što iako konkretno navedeni procesor može da radi sa do 4 niti u ovom programu se sve odvija na jendoj. Eventulano uvođenje višeprocesnog programiranja bi moglo da dovede do povećanja performansi.

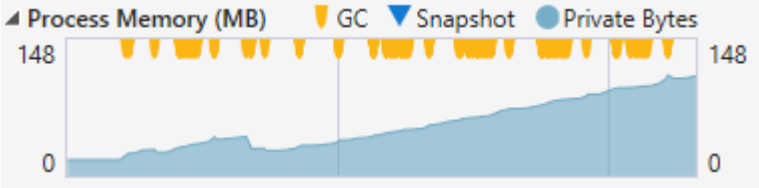


Graf 4.1.: *Iskorišćenost procesora*

Sledeći bitan resurs je ram memorija. Koliko memorije je potrebno programu zavisi od toga da li se stablo pamti ili ne. U koliko se stablo ne pamti potrebno je vrlo malo memorije i količina je konstantna za sve vrednosti n, u ovom slučaju jedini kritačn resurs jeste vreme. Međutim u koliko se pamti celo stablo, kao i vreme potrebno za izvršavanje, i potrebna memorija eksponenicijalno raste sa porastom n. Na sledećim grafovima se vidi količina potrebne memorije za sličaj kada se na pamti stablo i kada se pamti.



Graf 4.2.: *Količina iskorišćene ram memorije kada se ne pamti stablo*



Graf 4.2.: *Količina iskorišćene ram memorije kada se pamti stablo*

Oba grafa predstavljaju pkretanje aplikcaije sa vrednostima $n = 8,9, \dots ,13$. Treba primetiti žute trouglove na vrhu grafa. Oni predstavljaju trenutke kada je pokrenut Garbage Collector, koji oslobađa memoriju koja je zauzeta a ne koristi se. Očigledno je da u dugrom primeru mnogo čeće postoji potreba za oslobađanjem nepotrebno zauzete memorije kako bi program mogao da je koristi. Takođe pad vredsnoti na grafu je u trenucima kada počne da se formira novo stablo pa GC oslobodi memoriju, bez mehanimzma GC graf bi bio linearno rastući.

6 Zaključak

Problem osam, ili n kraljica, je odličan primer kako jedan, donekle prost, zadatak, može, u stvari, da bude veoma zahtevan. Pored odličnog primera za pokazivanje razlike između brut force i sufisticiranijih algoritama, može da se koristi i kao primer za memory management i multithreading.

Prikazano rešenje nije baš najsrećnije jer veoma neefikasno po pitanju oba resursa. Mnogo puta je sporije od najbržeg rešenja i u koliko se pokrene sa vrednošću 15 za *n* dobija se OutOfMemory Exception. Ali i dalje može da se koristi kao primer backtracking algoritam kao i razvijanja stabla.

Jedno od najboljih rešenja se može maći na sledećem linku:

http://www.jsomers.com/nqueen_demo/nqueens.html

Program je rađen u C programskom jeziku, manipuliše bitima i pronalazi pola rešenja pa množi taj broj sa dva, pošto sva rešenja mogu jednostvano zarotiraju da bi se dobila druga. Iz svih tih razloga je očigledno zbog čega je ovo rešenje toliko brže.

7 Literatura

[1] https://en.wikipedia.org/wiki/Eight_queens_puzzle

[2] http://www.jsomers.com/nqueen_demo/nqueens.html