Paralelizacija genetskog algoritma za problem N - kraljica

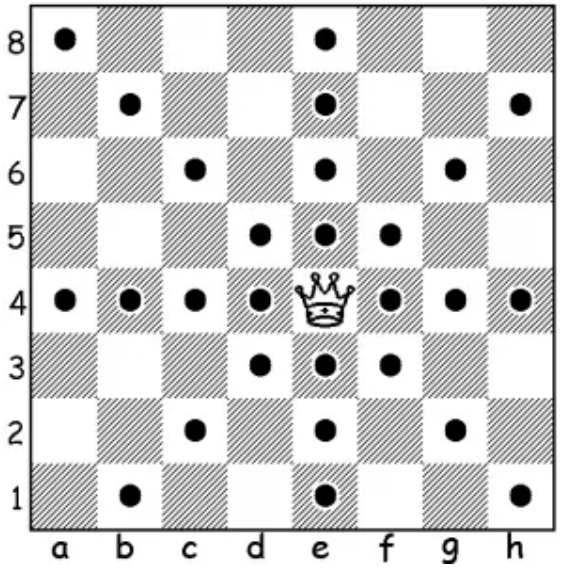
Vladimir Popov
Balša Bulatović

Jun, 2023

Sadržaj

1 Opis problema

Problem N - kraljica predstavlja klasičan kombinatorni problem u oblasti računarstva. Problem ima jednostavnu strukturu i definisan je na sledeći način. Na šahovsku tablu, veličine $N \times N$, treba postaviti N kraljica, tako da se nijedan par kraljica medjusobno ne napada. Dve kraljice se napadaju ukoliko se nalaze u istom redu, koloni ili na dijagonali (Slika 1).



Slika 1: Kretanje kraljice u šahu

Najjednostaviji primer ove vrste problema, za koji postoji rešenje, predstavlja problem 4 kraljice. U opštem slučaju, postoji $\binom{n^2}{n}$ različitih načina da se postavi N kraljica na tablu veličine $N \times N$, tako da za relativno mali problem od 10 kraljica postoji više od $1.73 * 10^3$ mogućnosti, što predstavlja velik prostor rešenja za pretraživanje, kod tako malog problema.

2 Osnovne karakteristike i serijska implementacija algoritma

Zarad rešavanja gorepomenutog problema, odlučili smo se za pristup genetskog algoritma. Njegovo generalno funkcionisanje, kao i konkretna implementacija za ovaj problem biće objašnjeni u narednim potpoglavljima.

2.1 Prikaz jedinke

Temelj svakog genetskog algoritma jesu jedinke, sastavljene od hromozoma, koje predstavljaju potencijalna rešenja. U našem slučaju svaka jedinka predstavlja niz od N elemenata, u kojem je vrednost svakog elementa tačno jedan ceo broj od 0 do $N-1$. Na ovaj način rešen je problem konflikta kraljica u istom redu. Element jedinke predstavlja indeks kolone gde se nalazi kraljica koja se nalazi u i-tom redu, gde je i indeks elementa u okviru jedinke. U nastavku je prikazan kod u programskom jeziku C++ funkcije koja generiše populaciju.

```
1   vector<vector<int>> init(int n, int population_size)
2   {
3     vector<vector<int>> population;
4     for (int i = 0; i < population_size; i++)
5     {
6       vector<int> individual;
7       for (int j = 0; j < n; j++)
8       {
9         int genome = get_random_int(0, n - 1);
10        individual.push_back(genome);
11      }
12
13      population.push_back(individual);
14    }
15    return population;
16  }
17
```

Kod 1: Funkcija za generisanje populacije

2.2 Ocena kvaliteta jedinke

Kvalitet jedinke (*fitness*) predstavlja ukupan broj kraljica koje se međusobno napadaju. Zbog samog načina prikaza jedinke, jedini sporan konflikt je na dijagonalama i u kolonama, pa kvalitet jedinke neposredno predstavlja broj kraljica koje se nalaze na istim dijagonalama i u istim kolonama. Za svaku kraljicu koja je u konfliktu sa nekom drugom, kvalitet opada za 1. Kako je u implementaciji programa cilj dostići nula konflikata, najkvalitetnija jedinka će imati ocenu nula. Sledi implementacija funkcije za računanje ocene kvaliteta jedinke u C++ programskom jeziku.

```
int fitness_score(int n, const vector<int> &individual)
{
  int res = 0;
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
    {
      if (i == j)
      {
        continue;
      }

      // two queens are attacking each other if they are:
      // 1 - in the same column
      // 2 - on the same diagonal

      int x1 = i;
      int x2 = j;
      int y1 = individual[i];
      int y2 = individual[j];

      if (y1 == y2)
      res++;
      else if (abs(x1 - x2) == abs(y1 - y2))
      res++;
    }
  }

  // we counted each occasion twice
  return res / 2;
}

```

Kod 2: Funkcija za računanje ocene kvaliteta jedinke

2.3 Ukrštanje i mutacija

Odabir jedinki za ukrštanje vrši se ruletskom selekcijom. Proces ruletske selekcije počinje tako što se prvo za svaku jedinku iz populacije odredi ocena kvaliteta, a nakon toga se dobijena ocena pomnoži sa nasumičnim realnim brojem $x \in (0, 1)$. Iz populacije se za ukrštanje uzimaju dve jedinke sa najvećim dobijenim rezultatom iz prethodnog postupka. Sledeći deo koda prikazuje proces ruletske selekcije na našem problemu.

```
vector<vector<int>> population = init(n, population_size);

vector<pair<vector<int>, int>> population_scores(population.
size());

transform(population.begin(), population.end(), back_inserter(
population_scores), [&](vector<int> &ind)
{ return make_pair(ind, fitness_score(n, ind)); });

```

```
    transform(population_scores.begin(), population_scores.end(),
    back_inserter(population_scores_roulette), [&](pair<vector<int>,
    int> &ind)
    { return make_pair(ind.first, ind.second * get_random_real(0.0,
    1.0)); });

    sort(population_scores_roulette.begin(),
    population_scores_roulette.end(), [](const pair<vector<int>,
    double> &ind1, const pair<vector<int>, double> &ind2)
    { return ind1.second < ind2.second; });

```

Kod 3: Proces ruletske selekcije

Samo ukrštanje dve jedinke odvija se tako što se prvo izabere nasumičan realan broj $x \in (0, 1)$. Nakon toga, roditelji se polove na x – toj poziciji, i nastaju deca spajanjem prvog dela prvog roditelja sa drugim delom drugog roditelja i obrnuto. Nakon toga postoji 70% šanse da će doći do mutacije kod dece, koja započinje ponovnim nasumičnim biranjem broja x u istom intervalu. Mutacija se vrši postavljanjem nasumične vrednosti od 0 do $N-1$ na poziciji x. Sledeći kodovi prikazuju implementaciju procesa ukrštanja i procesa mutacije.

```
    void mutate(vector<int> &child, int n)
    {
      int mutation_point = get_random_int(0, n - 1);
      child[mutation_point] = get_random_int(0, n - 1);
    }

    pair<vector<int>, vector<int>> crossover(int n, vector<int>
    parent1, vector<int> parent2)
    {
      // random pivoting point
      int crossover_point = get_random_int(0, n - 1);

      vector<int> child1(parent1.begin(), parent1.begin() +
    crossover_point);
      child1.insert(child1.end(), parent2.begin() + crossover_point
    , parent2.end());

      vector<int> child2(parent2.begin(), parent2.begin() +
    crossover_point);
      child2.insert(child2.end(), parent1.begin() + crossover_point
    , parent1.end());

      // mutation - 70% chance of a random change in the child
      if (get_random_int(1, 10) <= 7)
      mutate(child1, n);

      if (get_random_int(1, 10) <= 7)
      mutate(child2, n);

      return make_pair(child1, child2);
    }
```

```
27
```

Kod 4: Proces ukrštanja

2.4 Selekcija (prelazak u sledeću generaciju)

Nakon N ukrštanja, populacija se povećala za broj novonastale dece, i sada iznosi $2N$. S obzirom da je neophodno da kardinalnost populacije ostane ista, potrebno je izvršiti proces selekcije jedinki. Primenjuje se elitizam, gde se dopušta da 5% najboljih iz prethodne generacije predje u sledeću. Sortiranjem populacije na osnovu ocene kvaliteta svake jedinke, u narednu generaciju prelazi prvih N jedinki trenutne populacije. Naredni kod prikazuje implementaciju procesa selekcije i elitizam.

```
1     // we let 5% of the best parents live on - elitism
2     int elitism_deg = population_size * 0.05;
3
4     for (int i = 0; i < elitism_deg; i++)
5     {
6       if (children_scores[children_scores.size() - i - 1].second >
      population_scores[i].second)
7       {
8         swap(children_scores[children_scores.size() - i - 1],
      population_scores[i]);
9       }
10    }
11
12    transform(children_scores.begin(), children_scores.end(),
      population_scores.begin(), [](const pair<vector<int>, int> &ind)
13    { return ind; });
14
15    // clear children to avoid reallocating memory
16    children.clear();
17
```

Kod 5: Proces selekcije i elitizam (prelazak u sledeću generaciju)

2.5 Odabir parametara algoritma

Kako bismo sprečili da algoritam ostane u "lokalnom minimumu" šansa za mutaciju je poprilično velika i iznosi 70%, dok je stopa elitizma 5%. Na ovaj način omogućavamo algoritmu da u svaku generaciju uvodi veći broj noviteta i time brže konvergira. Podrazumevano, svaki od ovih parametara je podložan promenama i ukoliko primetimo da ne dolazi do promena u generacijama duži vremenski period, program se može zaustaviti i neki od parametara se može izmeniti u cilju brže kovergencije za dati slučaj.

3 Paralelna implementacija algoritma

Zbog prirode samog algoritma, koji se izvršava u nekoliko koraka kod kojih je bitan redosled izvršavanja, nije moguće parelelizovati celokupan program već samo određene delove. Zarad otkrivanja tih delova koristili smo alat poznat pod imenom *GNU gprof*. To je alat iz porodice *profiler*-a, koji nam omogućava da vidimo koje funkcije u kodu nam troše najviše vremena. Očekivano, najviše programskog vremena trajalo je ukrštanje jedinki, te smo se odlučili za paralelizovanje tog dela koda. U svrhe paralelizacije korišćena je apstrakcija zadataka (*task*) iz *OneTBB* biblioteke.

Početni korak predstavljalo je izdvajanje funkcije koja u sebi kombinuje ruletsku selekciju, ukrštanje i mutaciju nad delom populacije i na kraju vraća dobijenu decu, gde je svaka jedinka (dete) namapirana na svoju ocenu kvaliteta, kako taj posao ne bismo ponavljali kasnije. Kod opisane funkcije sledi u nastavku.

```
1    vector<pair<vector<int>, int>> next_generation(vector<pair<
     vector<int>, int>> population_scores, int start, int end, int n)
2    {
3      vector<vector<int>> children;
4      end = (population_scores.size() < end ? population_scores.
     size() : end);
5      for (int i = 0; i < (end - start) / 2; i++)
6      {
7        // sorting by adaptation - selection
8        vector<pair<vector<int>, double>>
     population_scores_roulette;
9        transform(population_scores.begin(), population_scores.end
     (), back_inserter(population_scores_roulette), [&](pair<vector<
     int>, int> &ind)
10       { return make_pair(ind.first, ind.second * get_random_real
     (0.0, 1.0)); });
11       sort(population_scores_roulette.begin(),
     population_scores_roulette.end(), [](const pair<vector<int>,
     double> &ind1, const pair<vector<int>, double> &ind2)
12       { return ind1.second < ind2.second; });
13
14       vector<int> child1, child2;
15       tie(child1, child2) = crossover(n,
     population_scores_roulette[0].first, population_scores_roulette
     [1].first);
16       children.push_back(child1);
17       children.push_back(child2);
18     }
19
20     vector<pair<vector<int>, int>> children_scores;
21     transform(children.begin(), children.end(), back_inserter(
     children_scores), [&](vector<int> &ind)
22     { return make_pair(ind, fitness_score(n, ind)); });
23     sort(children_scores.begin(), children_scores.end(), [](const
      pair<vector<int>, int> &ind1, const pair<vector<int>, int> &
     ind2)
24     { return ind1.second < ind2.second; });
```

```
25
26      // clear children to avoid reallocating memory
27      children.clear();
28
29      return children_scores;
30    }
31
```

Kod 6: Funkcija za računanje dela sledeće generacije

Nakon toga potrebno je definisati veličinu dela populacije koji će biti obrađivan nezavisno. Ta veličina nam ustvari služi kao *cutoff* parametar i predstavlja broj jedinki koji će biti obrađivan serijski u okviru jednog zadatka. Na izbor ove vrednosti utiču veličina šahovske table i veličina populacije. U narednim poglavljima biće prikazane performanse za neke konkretne odabrane vrednosti ovog parametra.

Paralelno pokretanje same funkcije *next generation*, uz pomoć *OneTBB* zadataka prikazano je sledećim kodom.

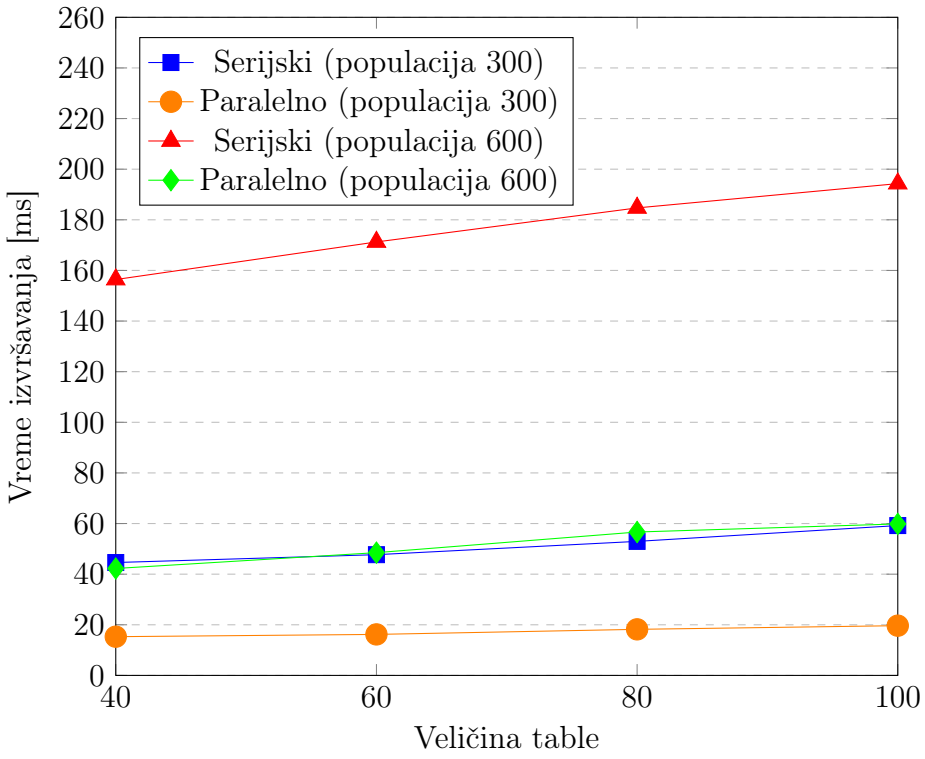
```
1     concurrent_vector<vector<pair<vector<int>, int>>> parallel_res;
2
3     for (int i = 0; i < population_size; i += group_size)
4     {
5       g.run([=, &parallel_res]
6       { parallel_res.push_back(next_generation(population_scores, i
    , i + group_size, n)); });
7     }
8
```

Kod 7: Paralelno pokretanje funkcije za računanje dela sledeće generacije

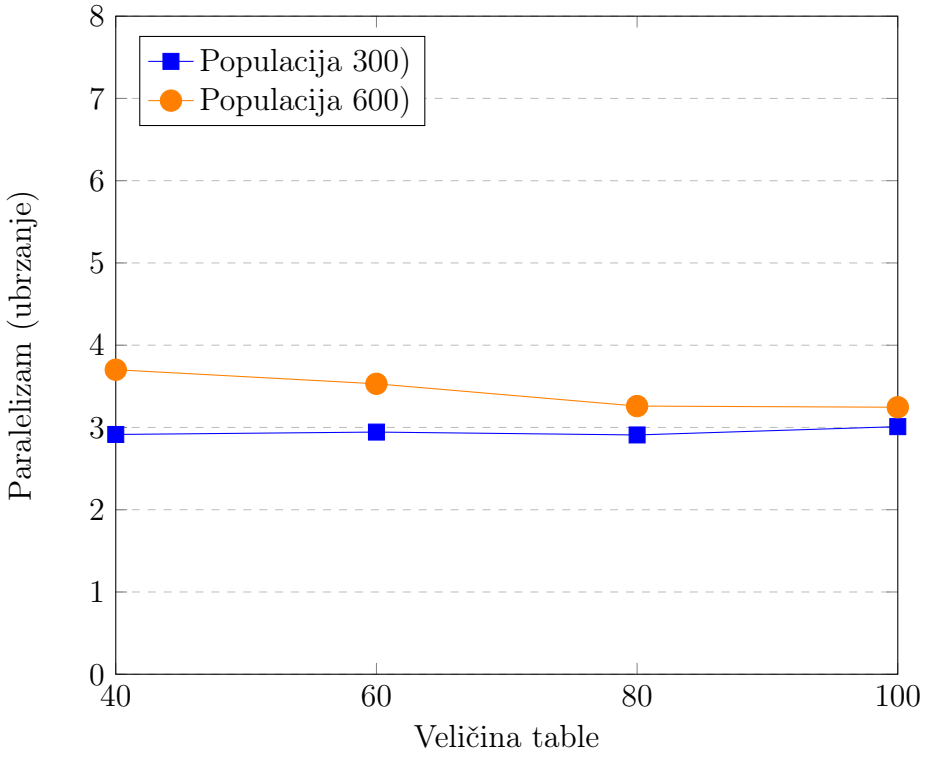
4 Analiza performansi

Nemoguće je predvideti koliko će generacija biti potrebno algoritmu da konvergira, u nekim slučajevima dovoljno je 100 generacija, ali taj broj može da naraste i do 8000. Zbog toga ukupno vreme izvršavanja algoritma ne predstavlja objektivnu meru performanse. Iz tog razloga odlučili smo da umesto ukupnog vremena prikažemo vreme koje je potrebno da bi se obradila jedna generacija (količnik ukupnog vremena i broja generacija).

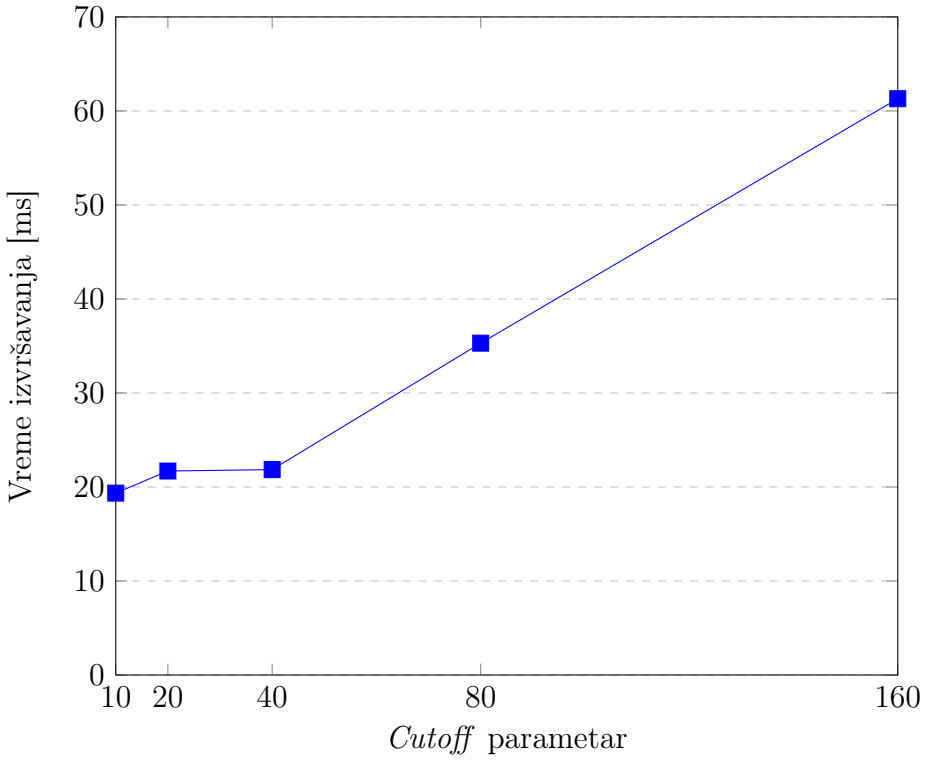
Rezultati testiranja programa uz promenu veličine šahovske table i populacije, prikazani su na grafikonima u nastavku. Za potrebe testiranja korišćen je procesor *AMD Ryzen* 7 4700u, sa 8 jezgara, zasnovan na *Zen* 2 procesorskoj mikroarhitekturi. Kod je kompajliran uz pomoć *g++* kompajlera verzije 11.3.0 na operativnom sistemu *Ubuntu* 22.04.



Grafik 1: Vreme izvršavanja algoritma, po generaciji, za različite veličine populacije, u zavisnosti od veličine šahovske table



Grafik 2: Ostvareni paralelizam (ubrzanje), za različite veličine populacije u zavisnosti od veličine šahovske table



Grafik 3: Vreme izvršavanja algoritma, po generaciji, za veličinu populacije 300 i veličinu šahovske table 100 u zavisnosti od *cutoff* parametra (veličine grupe)

Primećeno je da za *cutoff* parametar veći od 20, i veličine populacije i šahovske table koje odgovaraju onima sa prošlog grafika procesor nije dovoljno upošljen, tj. njegov load na svim jezgrima ne prelazi 90%. Imajući to u vidu odlučili smo se da cutoff za date dimenzije bude 10, dok smo za ostale učestale dimenzije koristili druge vrednosti parametra, koje su se isto empirijski pokazale kao najbolje.

5 Zaključak

Kao što je i poznato, genetski algoritam nema garancije pronalaženja rešenja, a razlog te nepredvidljivosti su u velikoj meri mutacije i ukrštanja. Takođe, empirijskom proverom zaključeno je da program koji radi sa manjom kardinalnosti populacije brže konvergira ka traženom rešenju, prilikom serijskog izvršavanja. Paralelizacijom dobijamo mogućnost da povećamo veličinu populacije, čime trošimo više resursa, ali ukupno vreme izvršavanja je kraće.

Paralelizacija programa pruža mogućnost efikasnijeg korišćenja dostupnih hardverskih resursa, što može rezultovati poboljšanjem performansi i smanjenjem vremena izvršavanja. Upotrebom odgovarajućih tehnika, moguće je rasporediti radne zadatke na više procesora ili jezgara i ostvariti istovremeno izvršavanje, čime se postiže ubrzanje izvršavanja. Ograničenja koja paralelizacija nosi su trke do podataka i veća kompleksnost koda, kao i teže otkrivanje grešaka. Pravilan odabir algoritama i strategija, kao i pažljivo upravljanje podacima i sinhronizaija istih predstavljaju ključne korake u postizanju optimalnih rezultata.

Uprkos navedenim izazovima, paralelizacija se pokazuje kao snažan alat za poboljšanje performansi programa. Ispravno implementirana paralelizacija može doneti značajno kraće vreme izvršavanja, ali i bolju upotrebu raspoloživih resursa.

6 Tabele

Veličina table	Serijski (populacija 300) [ms]	Paralelno (populacija 300) [ms]
40	44.6	15.3
60	47.7	16.2
80	52.95	18.2
100	59.2	19.65

Tabela 1: Tabela za grafik 1 (1. deo)

Veličina table	Serijski (populacija 600) [ms]	Paralelno (populacija 600) [ms]
40	156.4	42.25
60	171.25	48.5
80	184.7	56.65
100	194.3	59.85

Tabela 2: Tabela za grafik 1 (2. deo)

Veličina table	Populacija 300	Populacija 600
40	2.915	3.702
60	2.944	3.531
80	2.909	3.260
100	3.01	3.246

Tabela 3: Tabela za grafik 2

Cutoff	Vreme izvršavanja [ms]
10	19.35
20	21.7
40	21.85
80	35.3
100	61.3

Tabela 4: Tabela za grafik 3