Maximum K-Cut

Mladen Puzić

8. septembar 2024.

Sadržaj

1 Uvod

Maximum K-Cut problem je klasičan problem optimizacije u teoriji grafova, koji se odnosi na pronalaženje najboljeg načina za podelu skupa čvorova grafa u K disjunktnih podskupova (particija) tako da se maksimizuje težina isečaka (cutova) između različitih podskupova. Isečak između dva podskupa se definiše kao skup grana koje povezuju čvorove iz jednog podskupa sa čvorovima iz drugog podskupa.

1.1 Formalna definicija problema

Dat je graf $G = (V, E)$, gde je V skup čvorova, a E skup grana, pri čemu svaka grana $e \in E$ ima pridruženu težinu $w(e)$. Cilj Maximum K-Cut problema je pronaći particiju skupa V na K podskupova $V_1, V_2, \ldots, V_K$ tako da suma težina grana koje povezuju čvorove iz različitih podskupova bude maksimalna.

1.2 Važnost i primene

Maximum K-Cut problem ima brojne primene u raznim oblastima, uključujući klasterovanje, dizajn mreža, bioinformatiku, mašinsko učenje, kao i u raznim optimizacionim problemima. Na primer, u klasterovanju, ovaj problem može biti korišćen za grupisanje podataka na način koji maksimizuje sličnosti unutar grupa, a minimizuje sličnosti između različitih grupa.

1.3 Kompleksnost

Maximum K-Cut problem je NP-težak, što znači da ne postoji poznat algoritam koji rešava ovaj problem za sve ulazne grafove u polinomijalnom vremenu. Zbog toga se često koriste aproksimativni algoritmi i heuristike za rešavanje ovog problema u praksi.

2 Pristupi rešenju

Za projekat je odabran C++ iz dva razloga, delom zbog svoje efikasnosti, delom zbog potrebe da se projekat koristi i za svrhe predmeta Alati za Razvoj Softvera. Svako rešenje je implementirano kao zasebna klasa, uz nekoliko dodatnih klasa radi lakše organizacije programa. Odabrana rešenja su:

- Gruba sila
- Pohlepni algoritmi (sa prvom promenom, sa najboljom promenom)
- Simulirano kaljenje
- Genetski algoritam

Pogledajmo detaljnije različite aspekte implementacije projekta.

2.1 Predstavljanje grafa

Na slici 1 vidi se implementacija grafa koja se koristi u svim rešenjima. Ona takođe definiše i strukturu grana. U grafu čuvamo broj čvorova i sve grane u vektoru grana. Implementirani su *getteri*, kao i funkcija za dodavanje grana. Broj čvorova se postavlja kroz konstruktor.

```
class Graph {
   public:
    Graph(int nodeCount = 0) {
        m_nodeCount = nodeCount;
    }
    struct Edge {
        int src, dst, w;
    };
    void addEdge(int x, int y, int w);
    int getNodeCount();
    const std::vector<Edge>& getEdges();

   private:
    int m_nodeCount;
    std::vector<Edge> m_edges;
};
```

Slika 1: Definicija grafa

2.2 Predstavljanje rešenja

Klasa *Individual* je dizajnirana za modelovanje jedinke u kontekstu genetskih algoritama, ali je kasnije uopštena za primenu u svim rešenjima, dok je napravljena podklasa za svrhe genetskog algoritma. Klasa omogućava predstavljanje rešenja koje sadrži particiju grafa na određeni broj grupa, kao i procenu kvaliteta te particije kroz funkciju fitnesa.

Za svaki čvor pamtimo kojoj grupi od 0 do $K-1$ pripada. Konstruktor omogućava generisanje nula podele (svi elementi pripadaju grupi 0) ili generisanje nasumične podele. Čuvamo pokazivač na graf, broj grupa i trenutni fitnes. Pruža *getter* za fitness, koji daje i mogućnost da ga izračuna

od 0. Fitnes podrazumeva zbir težina grana koje se nalaze između različitih grupa. Ovo se radi u složenosti $O(N)$.

Takođe, pruža mogućnost za sitnu promenu (nasumična promena grupe jednog nasumičnog čvora), koja će se koristiti za simulirano kaljenje. Implementacija se može naći na slici 2.

```
class Individual {
   public:
    Individual() {}
    Individual(Graph *g, int groups, bool rnd)
        : m_fitness(0),
          m_groups(groups),
          m_graph(g),
          m_chooseGroup(std::uniform_int_distribution<int>(a: 0, b: groups - 1)) {
        if (rnd) {
            generateSplit();
        } else {
            m_split = std::vector<int>(n: m_graph->getNodeCount());
        }
    }

    void smallChange();
    long long getFitness(bool recalc = false);
    std::vector<int> m_split;

   protected:
    void generateSplit();
    void updateFitness();

    long long m_fitness;
    int m_groups;
    Graph *m_graph;
    std::uniform_int_distribution<int> m_chooseGroup;
};
```

Slika 2: Predstavljanje rešenja

2.3 Gruba sila

Implementacija brute force algoritma pretražuje sve moguće načine za particionisanje čvorova grafa u grupe. Funkcija run() pokreće pretragu, dok rekurzivna funkcija ispituje svaku moguću particiju. Na kraju, algoritam vraća najbolje rešenje koje je pronašao. Čuvamo trenutno i najbolje rešenje kroz *m_opt* and *m_cur*. Implementacija se može videti na slici 3.

2.4 Pohlepni algoritmi

Implementirane su dve verzije pohlepnog algoritma - po najboljoj promeni i po prvoj boljoj promeni. Oba algoritma pokušavaju da poboljšaju trenutnu particiju grafa tako što premeštaju čvorove između grupa i procenjuju da li to poboljšava fitnes funkciju.

2.4.1 Po najboljoj promeni

- Počinje sa nasumično generisanom particijom grafa.
- Zatim iterativno ispituje sve moguće premene pojedinačnih čvorova u druge grupe kako bi pronašao onu koja donosi najveće poboljšanje u fitnesu.
- Zatim iterativno ispituje sve moguće premene pojedinačnih čvorova u druge grupe kako bi pronašao onu koja donosi najveće poboljšanje u fitnesu.
- Ako se pronađe poboljšanje, primenjuje se najbolja promena i proces se ponavlja dok se više ne može poboljšati rezultat ili dok se ne iscrpe iteracije.

```cpp
class BruteForce {
  public:
   BruteForce(int k = 1, Graph* g = nullptr) {
       m_groups = k;
       m_graph = g;
   }

   long long run();

  private:
   void checkAllOptions(int idx);
   int m_groups;
   Individual m_cur;
   Individual m_opt;
   Graph* m_graph;
};
```

```cpp
void BruteForce::checkAllOptions(int idx) {
    if (idx == m_graph->getNodeCount()) {
        long long curScore = m_cur.getFitness(recalc: true);
        if (curScore > m_opt.getFitness()) {
            m_opt = m_cur;
        }
        return;
    }
    for (int opt = 0; opt < m_groups; opt++) {
        m_cur.m_split[idx] = opt;
        checkAllOptions(idx + 1);
    }
}

long long BruteForce::run() {
    m_cur = Individual(m_graph, m_groups, rnd: false);
    m_opt = m_cur;
    checkAllOptions(idx: 0);
    return m_opt.getFitness();
}
```

Slika 3: Implementacija grube sile

2.4.2 Po prvoj boljoj promeni

- Takođe počinje sa nasumično generisanom particijom grafa.
- U svakom koraku nasumično se određuje redosled čvorova za premenu.
- Algoritam zatim prolazi kroz čvorove i traži prvu promenu koja poboljšava fitnes. Kada je pronađe, odmah je primenjuje i počinje novu iteraciju.
- Proces se nastavlja dokle god se može naći poboljšanje ili dok ne istekne broj iteracija.

Oba algoritma vraćaju konačnu vrednost fitnes funkcije nakon što više nije moguće postići poboljšanje ili nakon što se iscrpe iteracije. Korišćeno je 50000 iteracija. Implementacije se mogu videti na slici 4.

```cpp
long long Greedy::runFirstImprovement(int iter) {
    Individual cur = Individual(m_graph, m_groups, rnd: true);
    long long curFitness = cur.getFitness();
    bool improved = true;
    while (improved && iter--) {
        improved = false;
        auto perm = std::vector<int>(n: m_graph->getNodeCount());
        for (int i = 0; i < perm.size(); i++) {
            perm[i] = i;
        }
        std::shuffle(first: perm.begin(), last: perm.end(), [>>] m_rnd);
        for (auto idx:int : perm) {
            int st = cur.m_split[idx];
            for (int nw = 0; nw < m_groups; nw++) {
                if (st != nw) {
                    cur.m_split[idx] = nw;
                    long long nwFitness = cur.getFitness(recalc: true);
                    if (nwFitness > curFitness) {
                        curFitness = nwFitness;
                        improved = true;
                        break;
                    }
                }
            }
            if (improved) {
                break;
            }
            cur.m_split[idx] = st;
        }
    }
    return cur.getFitness(recalc: true);
}
```

```cpp
long long Greedy::runBestImprovement(int iter) {
    Individual cur = Individual(m_graph, m_groups, rnd: true);
    long long curFitness = cur.getFitness();
    bool improved = true;
    while (improved && iter--) {
        improved = false;
        int mx_idx = 0, mx_grp = cur.m_split[0];
        for (int idx = 0; idx < m_graph->getNodeCount(); idx++) {
            int st = cur.m_split[idx];
            for (int nw = 0; nw < m_groups; nw++) {
                if (st != nw) {
                    cur.m_split[idx] = nw;
                    long long nwFitness = cur.getFitness(recalc: true);
                    if (nwFitness > curFitness) {
                        curFitness = nwFitness;
                        mx_idx = idx;
                        mx_grp = nw;
                    }
                }
            }
            cur.m_split[idx] = st;
        }
        if (cur.m_split[mx_idx] != mx_grp) {
            cur.m_split[mx_idx] = mx_grp;
            improved = true;
        }
    }
    return cur.getFitness(recalc: true);
}
```

Slika 4: Implementacija pohlepnog algoritma

2.5 Simulirano kaljenje

Implementacija simuliranog kaljenja radi tako što kombinuje lokalno pretraživanje sa kontrolisanim slučajnim promenama kako bi se izbeglo zaglavljivanje u lokalnim optimumima. Implementacija se može videti na slici 5.

Neki detalji implementacije:

- Algoritam počinje sa nasumično generisanom particijom grafa (*cur*), koja se takođe postavlja kao trenutno najbolje rešenje (*opt*).
- Tokom zadatog broja iteracija (*iter*), algoritam pravi malu promenu u trenutnom rešenju (*cur*) kako bi dobio novo rešenje (*nw*).
- Ako novo rešenje nije bolje, postoji mala šansa (koja opada tokom iteracija) da će algoritam prihvatiti to lošije rešenje, što omogućava izlazak iz lokalnih optimuma.
- Na kraju, algoritam vraća fitnes najboljeg rešenja (*opt*) koje je pronađeno tokom pretrage.

```
long long SimmulatedAnnealing::run(int iter) {
    Individual cur = Individual(m_graph, m_groups, rnd: true);
    Individual opt = cur;

    for (int idx = 1; idx <= iter; idx++) {
        Individual nw = cur;
        nw.smallChange();
        if (nw.getFitness() > cur.getFitness()) {
            cur = nw;
            if (nw.getFitness() > opt.getFitness()) {
                opt = nw;
            }
        } else {
            int rnd = std::uniform_int_distribution<int>( a: 1, b: idx)( [&] m_rnd);
            if (rnd <= 100) {
                cur = nw;
            }
        }
    }
    return opt.getFitness();
}
```

Slika 5: Implementacija simuliranog kaljenja

Odrađeno je 50000 iteracija. U prvoj verziji nije davao sjajne rezultate, što sam popravio povećanjem broja iteracija, kao i šanse da se prihvati lošije rešenje.

2.6 Genetski algoritam

2.6.1 GAIndividual

Za svrhe genetskog algoritma napravljena je odvojena podklasa klase *Individual*, *GAIndividual*. Njena definicija se može videti na slici 6.

Dodatno čuva verovatnoću mutacije svakog čvora. Implementira dve funkcije:

- člana *mutate* koji prolazi kroz niz podele i mutira svakog člana (promeni mu grupu) sa verovatnoćom *m_mutationProb*
- statičku funkciju *crossover* koja uzima prvih *position* elemenata iz *a*, a ostatak iz *b*

2.6.2 Selekcija

Na slici 7 se može videti implementacija selekcije:

- Ova funkcija implementira selekciju jedinki iz populacije koristeći pristup turnira.

```
class GAIndividual : public Individual {
   public:
    GAIndividual(Graph *g, int groups, double mutationProbability = 0.05)
        : Individual(g, groups, rnd: true),
          m_mutationProb(mutationProbability),
          m_doesMutate(std::uniform_real_distribution<double>(a: 0, b: 1)) {}
    static GAIndividual crossover(GAIndividual a, GAIndividual b, int position);
    void mutate();

   private:
    double m_mutationProb;
    std::uniform_real_distribution<double> m_doesMutate;
};
```

Slika 6: Definicija GAIndividual

- Populacija se najpre nasumično promeša.
- Zatim se bira najbolja jedinka iz nasumično odabranog podskupa populacije, koji ima veličinu *tournamentSize*.
- Ako se prosledi pokazivač na jedinku koja treba biti isključena (*excl*), ona se neće uzeti u obzir tokom selekcije - radi izbegavanja duplikata.
- Na kraju, funkcija vraća jedinku sa najvišim fitnesom iz tog podskupa kao izabranu jedinku za sledeće korake genetskog algoritma.

Takođe, tu je i implementacija poređenja individua po fitnesu, koja će kasnije biti korisna.

```
bool GeneticAlgorithm::fitnessCmp(GAIndividual a, GAIndividual b) {
    return a.getFitness() > b.getFitness();
}

GAIndividual GeneticAlgorithm::selection(std::vector<GAIndividual> &population,
                                         int tournamentSize,
                                         GAIndividual *excl) {
    std::vector<int> p;
    for (int i = 0; i < population.size(); i++) {
        p.push_back(i);
    }
    std::shuffle(first: p.begin(), last: p.end(), [>>] m_rnd);
    GAIndividual max_individual = population[p[0]];
    long long max_fitness = -1;
    for (int idx = 0; idx < tournamentSize; idx++) {
        int el = p[idx];
        if (&population[el] != excl &&
            population[el].getFitness() > max_fitness) {
            max_fitness = population[el].getFitness();
            max_individual = population[el];
        }
    }
    return max_individual;
}
```

Slika 7: Implementacija selekcije

2.6.3 Implementacija algoritma

Implementacija algoritma se može videti na slici 8.

- Algoritam počinje sa inicijalizacijom populacije od *populationSize* jedinki (*GAIndividual*), gde svaka jedinka predstavlja potencijalno rešenje problema.

- Algoritam zatim prolazi kroz zadati broj generacija (numGenerations), gde se u svakoj generaciji populacija evoluira kroz selekciju, reprodukciju, i mutaciju:
  - Sortiranje - populacija se sortira prema fitnesu korišćenjem funkcije *fitnessCmp*, kako bi jedinke sa boljim fitnesom bile na vrhu.
  - Elitizam - najbolje jedinke iz trenutne populacije (*elitismSize*) se direktno prenose u sledeću generaciju, osiguravajući da se dobra rešenja ne izgube.
  - Selekcija i reprodukcija - za ostatak populacije, jedinke se biraju putem turnirskog izbora (*selection*) kako bi se formirali parovi roditelja. Svaki par roditelja prolazi kroz proces ukrštanja (*crossover*), gde se deliće svojih rešenja kombinuju kako bi formirali dva potomka.
  - Mutacija - potomci zatim prolaze kroz proces mutacije (*mutate*), koji uvodi nasumične promene u rešenja sa određenom verovatnoćom (*mutationProb*), čime se omogućava istraživanje novih delova prostora rešenja.
  - Ažuriranje populacije - nakon reprodukcije i mutacije, nova generacija zamenjuje staru populaciju.

Na kraju evolucije, algoritam prolazi kroz finalnu populaciju i vraća najviši fitnes pronađen među jedinkama, što predstavlja najbolje rešenje koje je algoritam pronašao.

Ovaj genetski algoritam koristi selekciju na osnovu turnira, elitizam, ukrštanje, i mutaciju kako bi kroz više generacija evoluirao populaciju rešenja i pronašao optimalno ili blizu optimalno rešenje problema particionisanja grafa. Odabrano je koristiti sledeće parametre:

- $populationSize = 200$
- $numGenerations = 250$
- $elitismSize = 20$
- $tournamentSize = 20$
- $mutationProb = 0.1$

3 Rezultati

3.1 Testovi

Za svrhe testiranja algoritama generisano je 20 test primera sa različitim svojstvima:

- $1 - 5$: mali grafovi na kojima se u razumnom vremenu završava brute force, $10 \leq n \leq 20$, $2 \leq k \leq 5$.
- $6 - 10$: grafovi sa većim brojem čvorova, ali malim brojem grana, $120 \leq n \leq 130$, $190 \leq m \leq 200$, $2 \leq k \leq 20$.
- $11 - 15$: veći, gusti grafovi sa malim brojem grupa, $190 \leq n \leq 200$, $2 \leq k \leq 5$.
- $16 - 20$: veći, gusti grafovi sa većim brojem grupa, $190 \leq n \leq 200$, $6 \leq k \leq 20$.

Težine grana su u svim test primerima nasumični brojevi, u prvoj grupi do 100, u ostalim do 10000.

Ovakvim test primerima možemo da ispitamo uspeh algoritama na različitim klasama grafova.

3.2 Rezultati

Na tabeli 1 možemo videti apsolutne vrednosti rezultata svih algoritama na svim primerima. Svaki test primer je pokrenut triput i svi rezultati su uzeti kao prosek tri rezultata.

Kao što je očekivano, gruba sila je završila u razumnom vremenu samo na prvih pet test primera. Ne dostižu svi algoritmi vrednosti koje je pronašla gruba sila:

- Genetski algoritam pronalazi optimalno rešenje u 3 od 5 testova u sva tri pokušaja
- Pohlepni algoritmi ne pronalaze ni na jednom test primeru optimalno rešenje u sva tri pokušaja

```cpp
long long GeneticAlgorithm::run(int populationSize, int numGenerations,
                                int elitismSize, int tournamentSize,
                                double mutationProb) {
    std::vector<GAIndividual> population(
        populationSize, x: GAIndividual(m_graph, m_groups, mutationProb));
    std::vector<GAIndividual> new_population = population;

    for (int gen = 0; gen < numGenerations; gen++) {
        std::sort(first: population.begin(), last: population.end(), fitnessCmp);
        for (int el = 0; el < elitismSize; el++) {
            new_population[el] = population[el];
        }
        for (int j = elitismSize; j < populationSize; j += 2) {
            GAIndividual parent1 = selection([&] population, tournamentSize);
            GAIndividual parent2 =
                selection([&] population, tournamentSize, &parent1);

            int randomPos = std::uniform_int_distribution<int>(
                a: 0, b: m_graph->getNodeCount())([&] m_rnd);
            new_population[j] =
                GAIndividual::crossover(a: parent1, b: parent2, randomPos);
            new_population[j + 1] =
                GAIndividual::crossover(a: parent2, b: parent1, randomPos);

            new_population[j].mutate();
            new_population[j + 1].mutate();
        }
        population = new_population;
    }
    long long max_fitness = 0;
    for (auto ind :GAIndividual : population) {
        max_fitness = std::max(a: max_fitness, b: ind.getFitness());
    }
    return max_fitness;
}
```

Slika 8: Implementacija genetskog algoritma

- Simulirano kaljenje je svaki put pronašlo optimalno rešenje na ovoj grupi test primera.

Odavde možemo videti da se simulirano kaljenje najbolje snalazilo na malim grafovima, dok se pohlepni algoritam sa prvom boljom promenom snašao ubedljivo najgore.

Možemo pogledati i rešenja prvih 5 primera i grafički na slici 9.

3.3 Poređenje

Pošto je teško porediti apsolutne vrednosti s obzirom na njihovu veličinu, u tabeli 2 se mogu videti vrednosti skalirane tako da najbolja na testu iznosi 1 - odnosno procenat maksimuma koji je pronađen.

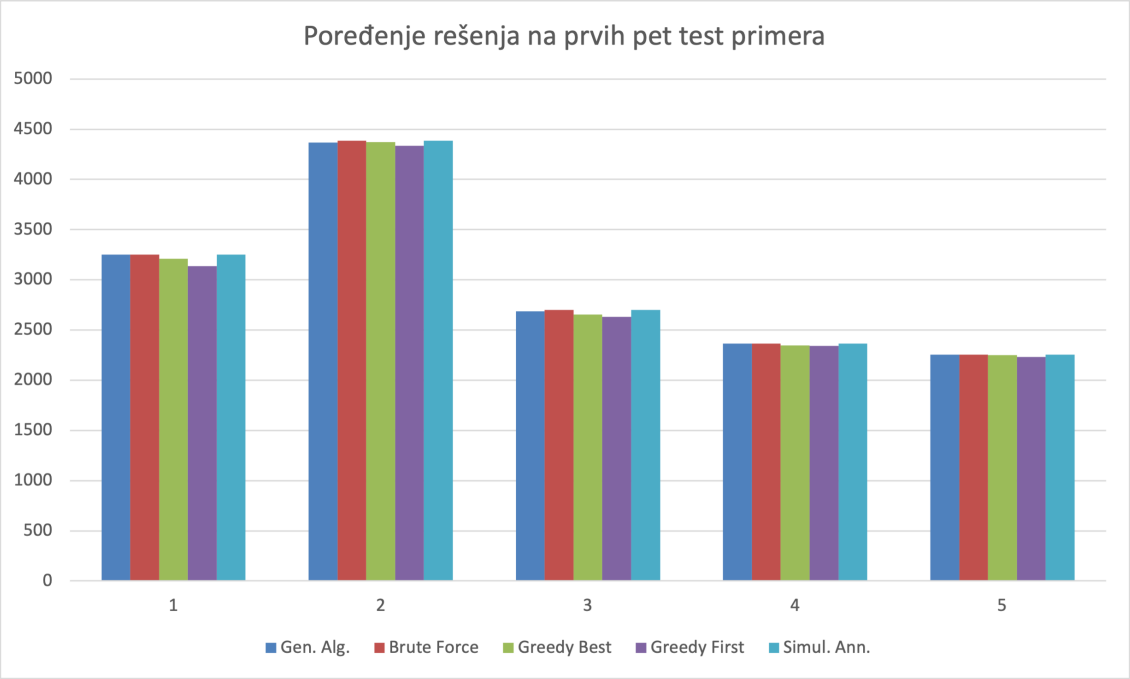
Takođe možemo da pogledamo neke grafike, jer je sada lakše primetiti razlike. Pogledati slike 10 i 11.

Na osnovu ovoga možemo da zaključimo dosta korisnih informacija:

- Na prvih pet primera postoji najviše relativne varijacije - ovo ima smisla iz dva razloga - za ovu grupu znamo optimalno rešenje zbog grube sile, tako da se rešenja porede sa optimalnim, takođe apsolutno rešenje je manje, pa manje apsolutne razlike prave veće procentualne razlike.
- Na drugih pet primera svi algoritmi su našli isto rešenje (možemo da pretpostavimo, ali ne možemo dokazati, da je u pitanju maksiumum). Ovo ima smisla, jer manji broj grana dovodi do manje kompleksnosti zadatka.
- Rezultati na poslednjih 10 primera su dosta slični. Vidimo da je na osam od deset primera najbolje simulirano kaljenje. Pohlepni algoritmi nikad nisu za više od 1% lošiji od maksimuma,

	Gen. Alg.	Brute Force	Greedy Best	Greedy First	Simul. Ann.
1	3252	3252	3209	3137.33	3252
2	4368.33	4383	4372.67	4336	4383
3	2687.33	2700	2654.33	2632.67	2700
4	2366	2366	2347.33	2342.67	2366
5	2256	2256	2252	2232.67	2256
6	996891	0	996891	996891	996891
7	1031925	0	1031925	1031925	1031925
8	930954	0	930954	930954	930954
9	958266	0	958266	958266	958266
10	1044532	0	1044532	1044532	1044532
11	73671101.33	0	75054876	75046486.67	75237405
12	60357970.67	0	61864732	62010305.33	62163160.33
13	40389686	0	41529287.67	41605705.67	41784411.33
14	62874405.33	0	64192367.67	64297072	64479045
15	68979307	0	70546065	70531228.67	70723287.67
16	47093345.67	0	48666094.33	48663464.67	48851941.67
17	50979280.67	0	52540229.67	52548734.67	52697060.67
18	58760693	0	60142682.67	60198172.67	60141240.67
19	77788289.67	0	79098211	79195419	79183478.67
20	56413792.67	0	57916404.67	57962376.33	57971552

Tabela 1: Poređenje rezultata algoritama nad test primerima



Slika 9: Rešenja na prvih pet test primera

dok se genetski pokazao kao najgori sa ispod 97.5% na većini primera. Ovo potencijalno naznačava potrebu za različitim parametrima genetskog algoritma u zavisnosti od veličine grafa.

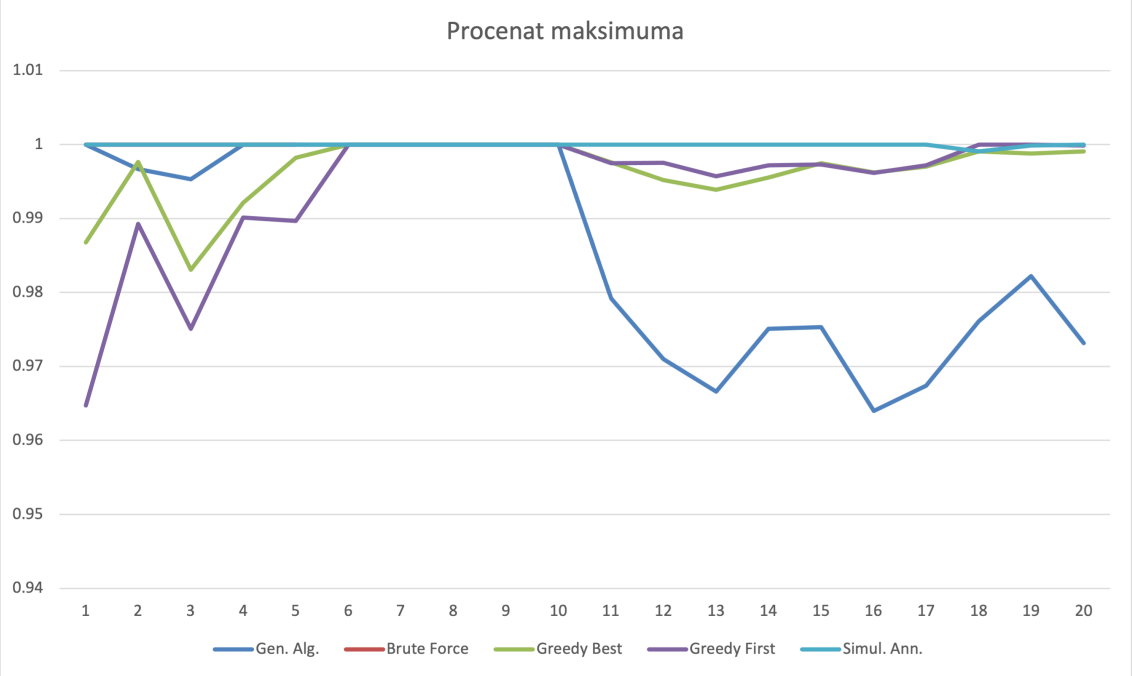
- Prednost simuliranog kaljenja je veća za manje K - ovo ima smisla, jer značajno smanjuje broj mogućih opcija za malu promenu.

4 Zaključak

Maximum K-Cut problem, iako NP-težak, pokazuje različite karakteristike u zavisnosti od veličine i strukture grafa. Na malim grafovima postoji značajna varijacija u rešenjima, pri čemu

	Gen. Alg.	Brute Force	Greedy Best	Greedy First	Simul. Ann.
1	1	1	0.986777368	0.964739647	1
2	0.996653738	1	0.997642406	0.989276751	1
3	0.995308642	1	0.98308642	0.975061728	1
4	1	1	0.992110454	0.990138067	1
5	1	1	0.99822695	0.98965721	1
6	1	0	1	1	1
7	1	0	1	1	1
8	1	0	1	1	1
9	1	0	1	1	1
10	1	0	1	1	1
11	0.979181849	0	0.997573959	0.997462455	1
12	0.970960459	0	0.995199273	0.997541068	1
13	0.966620917	0	0.993894286	0.99572315	1
14	0.975113781	0	0.995553946	0.997177796	1
15	0.975340786	0	0.99749414	0.99728436	1
16	0.964001513	0	0.996195702	0.996141873	1
17	0.967402736	0	0.997023914	0.997185308	1
18	0.976120875	0	0.999078211	1	0.999054257
19	0.982232188	0	0.998772555	1	0.999849229
20	0.973128901	0	0.999048717	0.999841721	1

Tabela 2: Poređenje rezultata kao procenat najboljeg rešenja

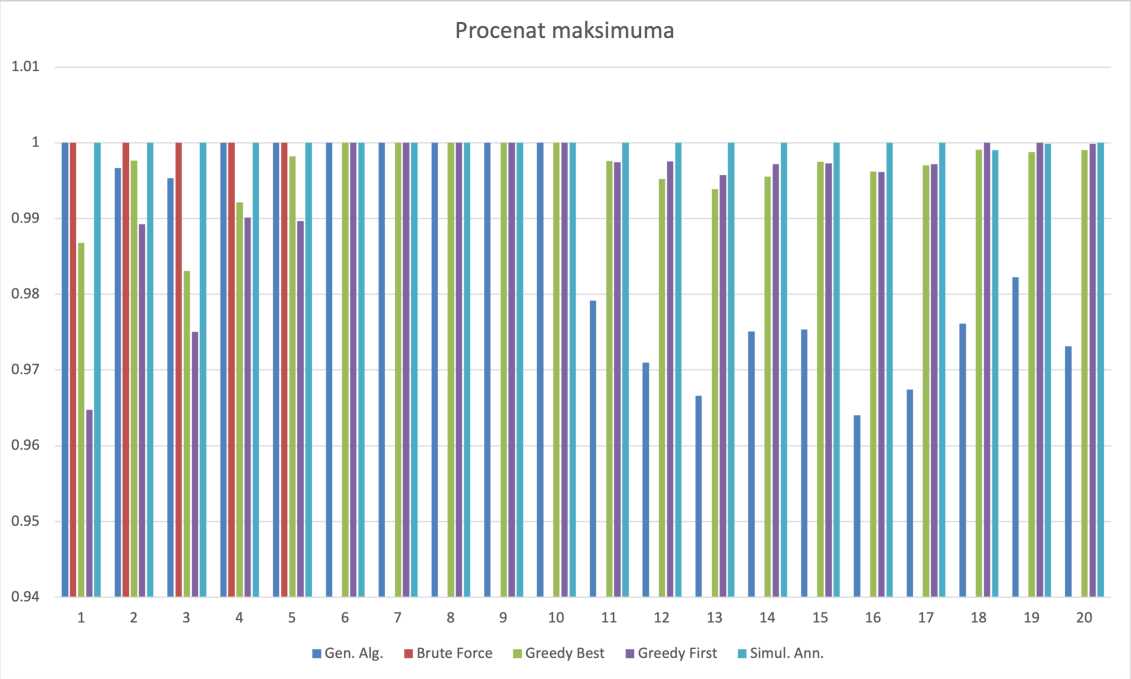


Slika 10: Grafik procenta od maksimuma 1

uspešnost algoritma zavisi od njegove sposobnosti da pronađe konkretno najbolje rešenje. Sa druge strane, kod grafova sa malim brojem grana, svi pristupi obično dovode do istog rešenja, jer su mogućnosti za podelu ograničene.

Na velikim i gustim grafovima, većina algoritama daje rešenja koja se razlikuju za najviše 1%, osim genetskog algoritma, koji zaostaje za oko 3% u odnosu na najbolje rešenje. Ova razlika bi se potencijalno mogla smanjiti drugim parametrima ili tipovima selekcije/mutacije. Među ispitivanim algoritmima, simulirano kaljenje se pokazalo kao najefikasniji na velikim grafovima, pružajući najbolje rezultate u većini slučajeva.

Za dalje istraživanje bi trebalo uraditi detaljnu analizu razliku pojedinačnih algoritama u zavisnosti od njihovih parametara, kao i dodatni testovi koji podrazumevaju posebne tipove grafova, poput kompletnih, bipartitnih i sličnih grafova. Takođe, različiti parametri algoritama za različite



Slika 11: Grafik procenta od maksimuma 2

vrste grafova.

5 Literatura

[1] Maximum K-Cut, online at: https://www.csc.kth.se/~viggo/wwwcompendium/node88.html