MAXIMUM K-CUT

Mladen Puzić, 18/2020
Matematički fakultet

Opis problema

- Dat je graf $G = (V, E)$, gde je V skup čvorova, a E skup grana, pričemu svaka grana $e \in E$ ima pridruženu težinu $w(e)$.
- Cilj Maximum K-Cut problema je pronaći particiju skupa V na K podskupova $V1, V2, ..., VK$ tako da suma težina grana koje povezuju čvorove iz različitih podskupova bude maksimalna.

Pristupi rešenji

- Pet algoritama:
  - *Gruba sila*
  - *Pohlepni algoritam po prvoj boljoj promeni*
  - *Pohlepni algoritam po najboljoj promeni*
  - *Simulirano kaljenje*
  - *Genetski algoritam*

Predstavljanje grafa i rešenja

```
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;
};
```

```
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;
};
```

Gruba sila

- Isprobavanje svih mogućnosti podele grafa rekurzijom
- Kreće od nulta podele i generiše sve kombinacije, nakon čega ih zapamti ukoliko su najbolje do sad

```
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();
}
```

Pohlepni algoritmi

```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);
}
```

```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);
}
```

Simulirano kaljenje

```
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 == 1) {
                cur = nw;
            }
        }
    }
    return opt.getFitness();
}
```

GAIndividual

```
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;
};
```

Selekcija

- Selekcija u formi turnira
- Nasumičan poredak, pa najbolji iz prefiksa
- Opcija za isključenje jedinke, da bi se izbegli duplikati

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

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

Genetski algoritam

- populationSize = 200
- numGenerations = 100
- elitismSize = 20
- tournamentSize = 10
- mutationP rob = 0.1

```
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;
}
```

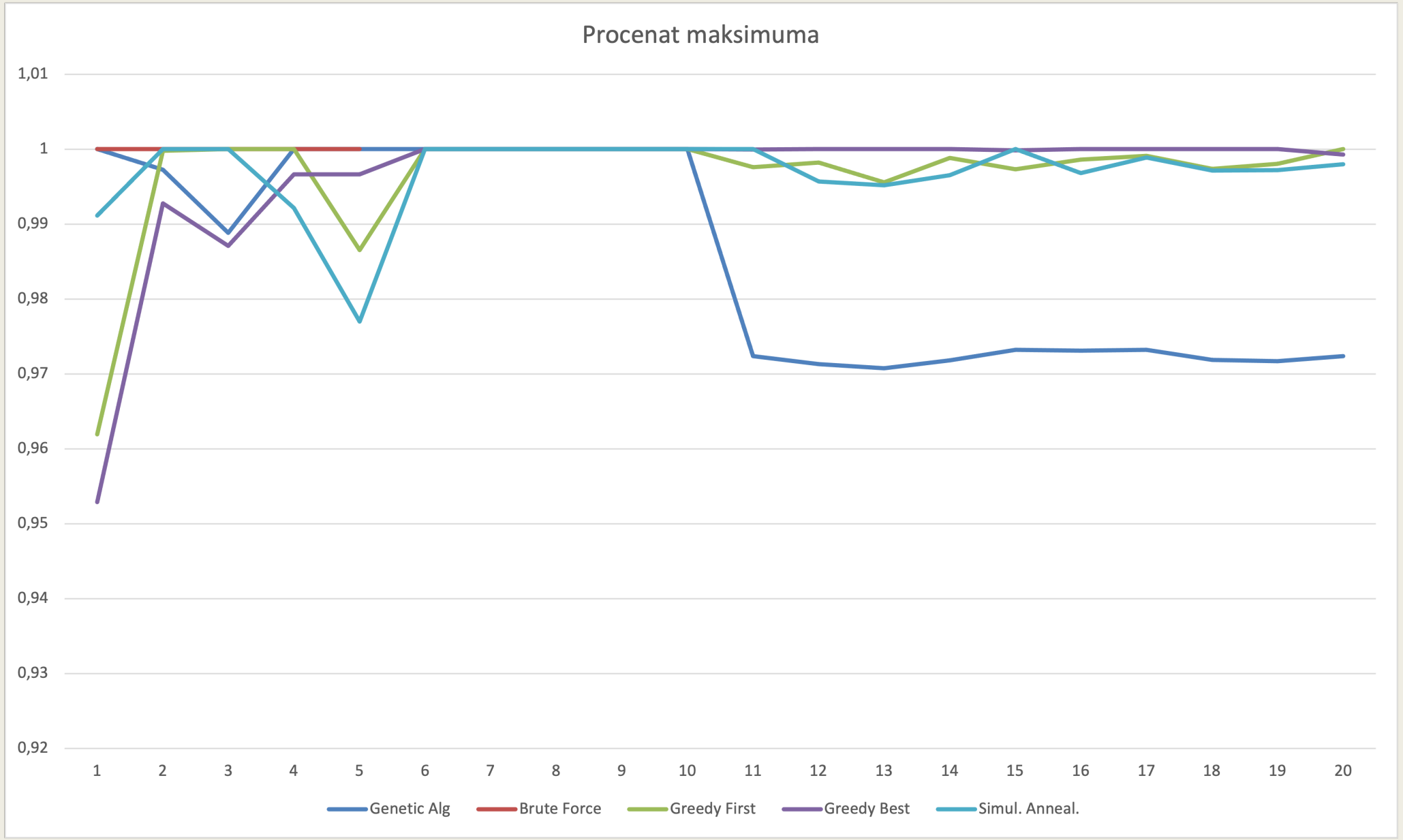
Testovi

- 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.

Rezultati

Test	Genetic Alg	Brute Force	Greedy First	Greedy Best	Simul. Anneal.
1	5414	5414	5208	5159	5366
2	4410	4422	4421	4390	4422
3	2220	2245	2245	2216	2245
4	1781	1781	1781	1775	1767
5	1781	1781	1757	1775	1740
6	998373	DNF	998373	998373	998373
7	998373	DNF	998373	998373	998373
8	998373	DNF	998373	998373	998373
9	998373	DNF	998373	998373	998373
10	998373	DNF	998373	998373	998373
11	64081911	DNF	65742163	65897081	65900744
12	63984576	DNF	65759174	65875541	65592592
13	64001325	DNF	65640470	65930720	65612981
14	63991546	DNF	65768969	65847171	65618374
15	64082178	DNF	65669440	65834406	65844752
16	73863753	DNF	75800299	75904130	75661330
17	73849310	DNF	75816071	75882576	75798196
18	73927324	DNF	75869469	76068139	75849645
19	73738216	DNF	75737432	75886053	75674536
20	73843991	DNF	75942209	75887695	75789954

Poređenje


Procenat maksimuma
1,01
1
0,99
0,98
0,97
0,96
0,95
0,94
0,93
0,92
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Genetic Alg
Brute Force
Greedy First
Greedy Best
Simul. Anneal.

Zaključak

- Značajno različiti rezultati u zavisnosni od veličine i tipa grafa i broja grupa.
- Na malim grafovima postoji značajna varijacija u rešenjima, pričemu uspešnost algoritma zavisi od njegove sposobnosti da pronade 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 verovatno mogla smanjiti većim brojem generacija, po ceni vremena izvršavanja.
- Medu ispitivanim algoritmima, pohlepan algoritam sa najboljim promenama se pokazao 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. Takode, različiti parametri algoritama za različite vrste grafova.

HVALA NA PAŽNJI!