# 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, 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 $V1, V2, ..., VK$ tako da suma težina grana koje povezuju čvorove iz različitih podskupova bude maksimalna.

# Pristupi rešenju

- 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 <= 100) {
                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::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;
}
```

# Genetski algoritam

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

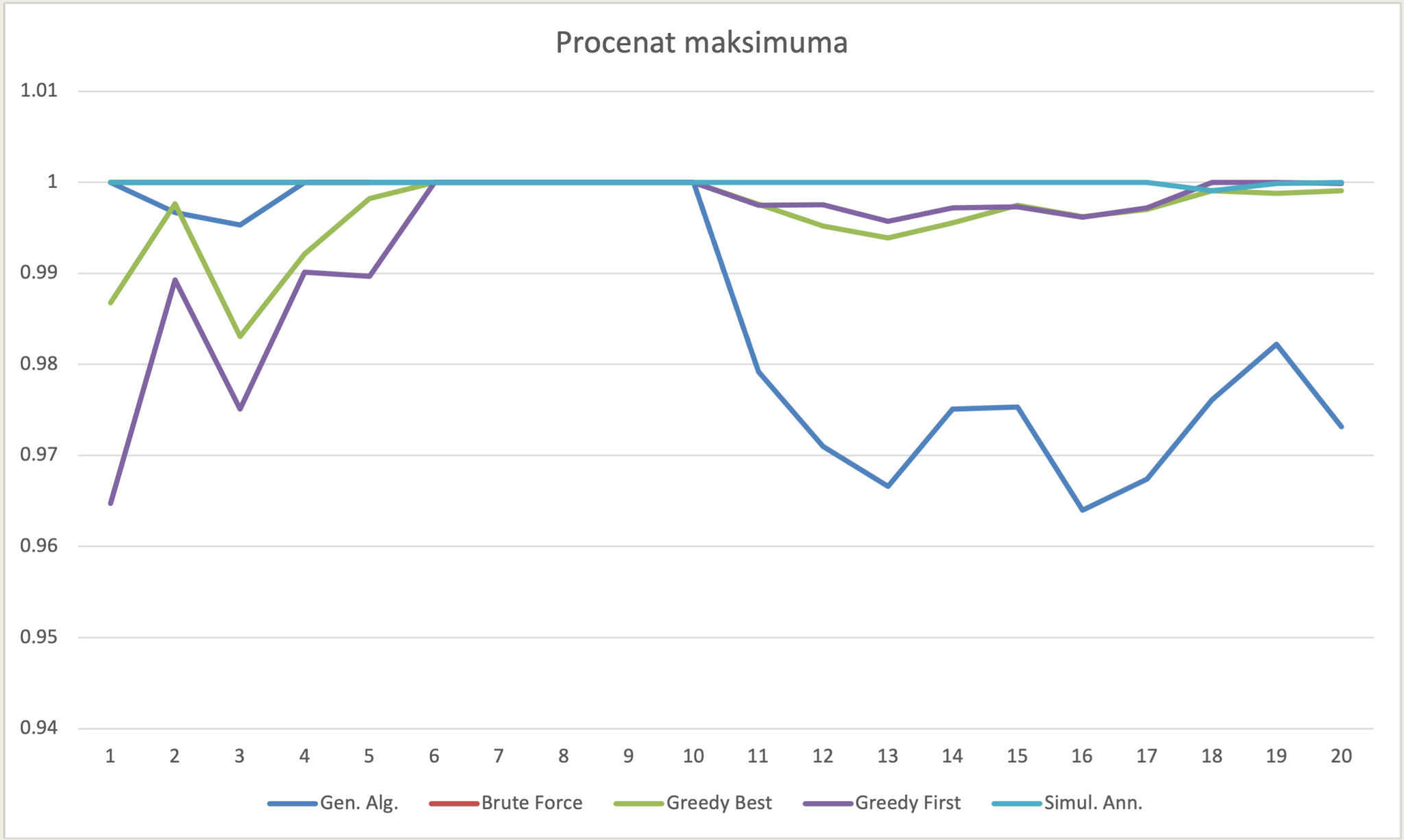
- Za svrhe testiranja algoritama generisano je 20 test primera sa različitim svojstvima:
  - *1 − 5:* mali grafovi na kojima se u razumnom vremenu završava brute force*, $10 \leq n \leq 20$, $2 \leq k \leq 5$.*
  - *6 − 10:* grafovi sa većim brojem čvorova, ali malim brojem grana*, $120 \leq n \leq 130$, $190 \leq m \leq 200$, $2 \leq k \leq 20$.*
  - *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

| | 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

# Poređenje


Procenat maksimuma
1.01
1
0.99
0.98
0.97
0.96
0.95
0.94
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Gen. Alg.
Brute Force
Greedy Best
Greedy First
Simul. Ann.

# Zaključak

- Značajno različiti rezultati u zavisnosni od veličine i tipa grafa i broja grupa.
- Na malim grafovima postoji značajna varijacija u rešenjima, pri čemu 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 potencijalno mogla smanjiti drugim tipovima selekcija I mutacija.
- Medu 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. Takode, različiti parametri algoritama za različite vrste grafova.

# HVALA NA PAŽNJI!