Rešavanje Max-k-Cut problema

Jelena Ivanović
Računarska inteligencija — Matematički fakultet
Septembar 2025.

Uvod

Max-k-Cut je jedan od klasičnih problema kombinatorne optimizacije. Cilj je podeliti čvorove neusmerenog grafa na k grupa tako da se **maksimizuje zbir težina ivica** koje povezuju različite grupe.

Problem je NP-težak, pa nije moguće pronaći optimalno rešenje za veće grafove u razumnom vremenu. Zbog toga se koriste **heuristike i metaheuristike** koje daju rešenja dovoljno bliska optimumu.

Primene: analiza mreža, klasterovanje podataka, raspoređivanje zadataka i dizajn komunikacionih sistema.

Formalna definicija

Neka je $G = (V, E)$ neusmeren težinski graf sa funkcijom $w : E \to \mathbb{R}^+$. Potrebno je pronaći podelu $F = \{C_1, \ldots, C_k\}$ koja maksimizuje:

$$\max \sum_{i<j} \sum_{u \in C_i} \sum_{v \in C_j} w(\{u, v\})$$

Svaka ivica između različitih grupa doprinosi preseku. Za $k = 2$ problem se svodi na poznati **Maximum Cut**.

Izazov: broj mogućih podela raste eksponencijalno sa brojem čvorova.

Brute-Force pristup

Brute-Force isprobava **sve moguće kombinacije** raspodele čvorova u k grupa — ukupno k^n podela. Za svaku raspodelu računamo vrednost preseka i čuvamo *najbolje do sada*.

```
def brute_force_max_k_cut(graph, k, start_time, timeout=300):
    nodes = list(graph.nodes())
    n = len(nodes)
    iters = 0
    best_val = -1
    best_labels = None

    for assign in product(range(k), repeat=n):
        iters += 1
        if time.time() - start_time >= timeout:
            break
        labels = {nodes[i]: assign[i] for i in range(n)}
        val = cut_value(graph, labels)
        if val > best_val:
            best_val = val
            best_labels = labels
    return best_val, best_labels, iters
```

```python
def cut_value(graph, labels):
    total = 0
    for u, v, data in graph.edges(data=True):
        if labels[u] != labels[v]:
            total += data.get("weight", 1)
    return total
```

- Enumeracija svih k^n raspodela (*assignments*).
- Za svaku raspodelu vrednost se računa funkcijom `cut_value`.
- Ako je nova vrednost veća, ažurira se `best_val` i `best_labels`.

Složenost: evaluiramo $\Theta(k^n)$ raspodela; jedna evaluacija je $\mathcal{O}(m)$ (broj ivica), pa je ukupno $\mathcal{O}(k^n \cdot m)$.

	Broj čvorova (n)	Vreme izvršavanja (s)
0	4	0.0012
1	7	0.0252
2	10	0.6416
3	30	300.0001

Prednost: daje tačno i optimalno rešenje — pouzdano za male grafove gde je broj kombinacija izvodljiv.

Mana: eksponencijalni rast složenosti (k^n) znači da brzo postaje nepraktičan za veće grafove.

Zaključak: Brute-Force garantuje optimalno rešenje, ali njegova složenost (k^n) ograničava primenu na veoma male grafove. U praksi se koristi za proveru tačnosti heurističkih metoda.

Greedy algoritam

Greedy radi *u hodu*: za svaki čvor redom bira u koju će **klasu** (grupu) ići tako da se trenutno u tom potezu **maksimizuje povećanje vrednosti preseka**. Tj. ne isprobava sve kombinacije, već gradi podelu korak-po-korak, ažurirajući najboljeg „lokalnog" izbora za svaki čvor.

Prednost: vrlo brz i jednostavan — radi skoro instant za velike grafove; često daje dovoljno dobar rezultat u praksi.

Mana: može se zaglaviti u *lokalnom maksimumu* — nije garantovan globalni optimum.

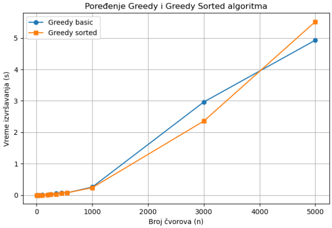
```python
def greedy_max_k_cut_basic(graph: nx.Graph, k: int):
    labels = {}
    for node in graph.nodes():
        best_group = 0
        best_value = float("-inf")
        for group in range(k):
            value = 0.0
            for neighbor, data in graph[node].items():
                w = data.get("weight", 1)
                if neighbor in labels and labels[neighbor] != group:
                    value += w
            if value > best_value:
                best_value = value
                best_group = group
        labels[node] = best_group
    return labels, cut_value(graph, labels)
```

```python
def greedy_max_k_cut_sorted(graph: nx.Graph, k: int):
    labels = {}
    sorted_nodes = sorted(graph.nodes(), key=lambda node: graph.degree(node), reverse=True)
    for node in sorted_nodes:
        best_group = 0
        best_value = float("-inf")
        for group in range(k):
            value = 0.0
            for neighbor, data in graph[node].items():
                w = data.get("weight", 1)
                if neighbor in labels and labels[neighbor] != group:
                    value += w
            if value > best_value:
                best_value = value
                best_group = group
        labels[node] = best_group
    return labels, cut_value(graph, labels)
```

Sortirani Greedy prvo izvrši **sortiranje čvorova** po kriterijumu značaja (npr. stepen čvora).

Ideja: čvorovi koji su *više povezani* imaju veći uticaj na vrednost preseka — zato im dajemo prioritet pri formiranju podela jer se nakon dodele za te čvorove uglavnom ne vraćamo unazad da menjamo njihov izbor.

Napomena: Sortiranje poboljšava stabilnost i kvalitet rešenja u većini testiranih instanci, dok je ,ukoliko i postoji, dodatno vreme za sortiranje zanemarljivo.



	Broj čvorova (n)	Vreme izvršavanja bez sortiranja(s)	Vreme izvršavanja sa sortiranjem(s)
0	4	0.0003	0.0002
1	7	0.0003	0.0002
2	30	0.0008	0.0006
3	50	0.0014	0.0009
4	100	0.0066	0.0024
5	200	0.0152	0.0135
6	250	0.0244	0.0263
7	350	0.0521	0.0288
8	450	0.0733	0.0606
9	550	0.0758	0.0771
10	1000	0.2577	0.2379
11	3000	2.9644	2.3567
12	5000	4.9291	5.5181

Genetski algoritam (GA)

Genetski algoritam inspirisan je **Darwinovom teorijom evolucije**: rešenja predstavljaju jedinke u populaciji koje se kroz generacije poboljšavaju selekcijom "jačih" rešenja, ukrštanjem i mutacijom. GA traži globalni maksimum kombinovanjem delova dobrih rešenja.

Glavne faze:

1) Inicijalizacija: Na početku pravimo početnu populaciju slučajnih rešenja – svaki čvor dobija neku od k grupa potpuno nasumično.

```
def initial_labels(self):
    return {node: random.randrange(self.k) for node in self.graph.nodes()}
```

2) Fitnes funkcija: kod nas je fitnes vrednost preseka: sabiramo težine svih ivica koje spajaju čvorove u različitim grupama. Pošto želimo da taj zbir bude što veći, naš cilj je maksimizacija fitnesa.

```python
def calc_fitness(self):
    total = 0
    for u, v, data in self.graph.edges(data=True):
        if self.labels[u] != self.labels[v]:
            total += data.get("weight", 1)
    return total
```

3) Selekcija: u kodu se koristila turnirska, gde nasumično izvučemo nekoliko jedinki i biramo onu koja ima najveći fitnes. Tako dajemo prednost boljim jedinkama (boljim rešenjima), ali i dalje ostavljamo šansu slabijima, kako bismo zadržali raznolikost

```python
def tournament_selection(population, tournament_size):
    tournament = random.sample(population, tournament_size)
    return max(tournament, key=lambda x: x.fitness)
```

3) Ukrštanje : u kodu se koristi ravnomerno, dete se pravi tako što svaki čvor ima jednaku šansu (50–50roditelja 2.

```
def crossover(parent1, parent2):
    child_labels = {}
    for node in parent1.labels:
        child_labels[node] = parent1.labels[node] if random.random() < 0.5 else parent2.labels[node]

    child = Individual(parent1.graph, parent1.k,child_labels)
    child.fitness = child.calc_fitness()

    return child
```

4) Mutacija: sa malom verovatnoćom da dolazi do mutacije, slučajno izaberemo jedan čvor i promenimo mu grupu. Ovo može dovesti do boljeg rešenja i povećava šansu da se pronađe globalni optimum.

```
def mutation(individual, mutation_prob):
    if random.random() < mutation_prob:
        node = random.choice(list(individual.labels.keys()))

        current_label = individual.labels[node]
        labels = list(range(individual.k))
        new_label = random.choice(labels)

        if new_label != current_label:
            individual.labels[node] = new_label


        individual.fitness = individual.calc_fitness()
```

```python
def ga(graph: nx.Graph, k, population_size, num_generations, tournament_size, elitism_size, mutation_prob):
    population = [Individual(graph, k) for _ in range(population_size)]

    for _ in range(num_generations):
        population.sort(key=lambda x: x.fitness, reverse=True)
        elites = population[:elitism_size]
        offspring = []

        for _ in range(population_size - elitism_size):
            parent1 = tournament_selection(population, tournament_size)
            parent2 = tournament_selection(population, tournament_size)
            child = crossover(parent1, parent2)
            mutation(child, mutation_prob)
            offspring.append(child)

        population = elites + offspring

    best_solution = max(population, key=lambda x: x.fitness)
    return best_solution
```

Prednost: GA-ovi su snažni u istraživanju prostora rešenja, često izlaze iz lokalnih maksimuma i kombinuju dobre konstruktivna rešenja.

Nedostatak: zahtevaju podešavanje parametara (veličina populacije, stopa mutacije, broj generacija) i mogu biti sporiji od prostih heuristika ako nisu pažljivo optimizovani.

Simulirano kaljenje (SA)

Simulirano kaljenje (*Simulated Annealing*) inspiriše se fizičkim procesom hlađenja metala.

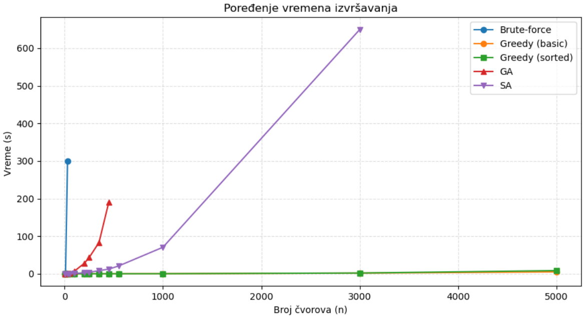
- Početna „temperatura" je visoka — sistem prihvata i lošije poteze.
- Temperatura se postepeno smanjuje — sistem se stabilizuje.
- Omogućava **izlazak iz lokalnih maksimuma**.

```python
def simulated_annealing_maxkcut(
    graph,
    k,
    initial_temperature: float = 1.0,
    cooling_rate: float = 0.995,
    stopping_temperature: float = 1e-4,
    max_iterations: int = 10000
):

    start = time.time()


    current_labels = initialize(graph, k)
    current_cut = cut_value(graph, current_labels)
    best_labels = current_labels
    best_cut = current_cut

    temp = initial_temperature


    for it in range(max_iterations):
        if temp < stopping_temperature:
            break

        neighbor_labels = change_label(current_labels, graph, k)
        neighbor_cut = cut_value(graph, neighbor_labels)

        delta = neighbor_cut - current_cut

        if delta > 0 or random.random() < math.exp(delta / temp):
            current_labels = neighbor_labels
            current_cut = neighbor_cut

            if current_cut > best_cut:
                best_labels = current_labels
                best_cut = current_cut


        temp *= cooling_rate

    elapsed = time.time() - start
    return best_labels, best_cut,elapsed
```

Prednosti: balans između brzine i kvaliteta.

Nedostatak: zahteva pažljivo podešavanje brzine hlađenja.

Poređenje algoritama



- **Brute-Force:** 100% tačno rešenje, ali neefikasno.
- **Greedy:** najbrži, ponekad manje tačan.
- **Sortirani Greedy:** bolji balans brzine i tačnosti.
- **Genetski i SA:** sporiji, ali daju najkvalitetnija rešenja.

Heuristike su pokazale da je moguće pronaći gotovo optimalne preseke čak i za velike grafove u odgovarajućem vremenu.

Zaključak

U radu su ispitane različite metode za rešavanje Max-k-Cut problema. **Brute-Force** je koristan za validaciju, ali ne i za veće instance. **Greedy** algoritmi su praktični i brzi, dok **Genetski** i **Simulirano kaljenje** daju kvalitetnija rešenja, naročito kod složenih grafova.

Glavni zaključak: Heuristički pristupi nude dobar balans između tačnosti i vremena izvršavanja. Izbor metode zavisi od veličine grafa i potrebne preciznosti.

Literatura

- Goemans, M. X. & Williamson, D. P. (1995). *Improved approximation algorithms for maximum cut*. JACM.
- Frieze, A. & Jerrum, M. (1997). *Improved approximation algorithms for MAX k-CUT and MAX BISECTION*. Algorithmica.
- Kann, V. (2000). *MAXIMUM K-CUT*. CSC KTH. `https://www.csc.kth.se/~viggo/wwwcompendium/node88.html`
- Newman, A. (2018). *Complex Semidefinite Programming and Max-k-Cut*. OASIcs–SOSA.
- Kodovi i eksperimenti sa predavanja i vežbi.