MAXIMUM K-CUT PROBLEM

Računarska inteligencija
Matematički fakultet

Jelena Ivanovic 365/19

September 2025

Sadržaj

1 Uvod **3**

2 Formalna definicija problema **4**

3 Algoritmi rešavanja **5**
3.1 Brute-force metod (metod grube sile) 5
3.2 Greedy (Pohlepni algoritam) 6
3.3 Genetski algoritam 7
3.4 Simulated Annealing (Simulirano kaljenje) 9

4 Poređenje algoritama **12**

5 Zaključak **18**

1 Uvod

Problem Maximum k-Cut predstavlja jedan od temeljnih problema u oblasti kombinatorne optimizacije i teorije grafova. Cilj je podeliti čvorove neusmerenog grafa na k disjunktnih skupova tako da se maksimizuje ukupan zbir težina ivica koje povezuju različite skupove. Kada je $k = 2$, problem postaje poznat kao Maximum Cut, za koji su Goemans i Williamson (1995) razvili revolucionarni pristup korišćenjem semidefinitnog programiranja (SDP), sa aproksimacionim faktorom od približno 0.87856.

Kasnije su Frieze i Jerrum proširili ovaj metod na slučaj $k > 2$, uvodeći efikasan algoritam zasnovan na SDP relaksaciji, koji daje bolja rešenja od nasumične podele. Dodatna poboljšanja dali su De Klerk i saradnici za različite vrednosti k, dok su Goemans i Williamson pokazali da se za $k = 3$ mogu koristiti kompleksne SDP tehnike sa najpreciznijim rezultatima .

Uprkos ovim naprecima, problem ostaje NP-težak za svako $k \geq 2$, što znači da nije poznat polinomski algoritam koji uvek daje tačno rešenje. Zbog toga se poseban fokus stavlja na razvoj heurističkih i aproksimativnih algoritama sa što boljim garancijama. Burer, Monteiro i Zhang (2002) predložili su heuristiku zasnovanu na rank-two relaksaciji za Max-Cut, koja koristi vektore u $\mathbb{R}^2$ umesto u $\mathbb{R}^n$. Ovaj pristup je značajno brži i skalabilniji, iako nema formalnu garanciju na aproksimacioni faktor, za razliku od klasičnog Goemans–Williamson algoritma.

Primene

- Klasterovanje podataka – grupisanje elemenata u klastere tako da se maksimizuje razdvajanje između grupa.
- Raspoređivanje zadataka – dodela poslova na više procesora ili radnih timova uz minimizaciju interne zavisnosti i maksimizaciju komunikacije između grupa.
- Dizajn komunikacionih mreža – segmentacija mreže u zone radi optimizacije protoka informacija i smanjenja interferencije.
- VLSI dizajn – fizičko razdvajanje komponenti čipova kako bi se smanjila dužina kritičnih veza između blokova.
- Analiza društvenih mreža – otkrivanje podgrupa korisnika koje su slabo povezane unutar, a snažno povezane van grupe.

2 Formalna definicija problema

Neka je dat neusmeren težinski graf $G = (V, E)$, gde je V skup čvorova ($|V| = n$), a E skup ivica sa funkcijom težina $w : E \to \mathbb{N}^+$ koja svakoj ivici $e \in E$ dodeljuje nenegativnu težinu. Takođe, dat je ceo broj k, $2 \leq k \leq |V|$.

Zadatak: pronaći particiju skupa čvorova V na k disjunktnih podskupova

$$F = \{C_1, C_2, \ldots, C_k\}$$

takvu da se maksimizuje zbir težina ivica koje povezuju čvorove iz različitih podskupova:

$$\max \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{\substack{u \in C_i \\ v \in C_j}} w(\{u, v\})$$

Drugim rečima, svaka ivica koja spaja različite skupove doprinosi vrednosti rešenja svojom težinom.

Složenost: Problem je NP-težak za svako $k \geq 2$. Za $k = 2$, problem je ekvivalentan Maximum Cut-u, a za opšte k povezan je sa problemom *Maximum k-Colorable Subgraph*.

3 Algoritmi rešavanja

U ovom poglavlju razmatramo različite pristupe rešavanju problema Maximum k-Cut. Metode koje ćemo prikazati su sledeće:

- **Metod grube sile (Brute-force)** – daje tačno rešenje, ali je primenljiv samo na male grafove zbog eksponencijalne složenosti.
- **Pohlepni algoritam (Greedy)** – heuristički pristup koji u svakoj iteraciji bira trenutno najbolje lokalno rešenje, bez garancije optimalnosti.
- **Genetski algoritam (GA)** – evolutivna heuristika inspirisana prirodnom selekcijom, koja koristi ukrštanje, mutaciju i elitizam da bi se došlo do boljih rešenja.
- **Simulirano kaljenje (Simulated Annealing)** – stohastička metoda zasnovana na analogiji procesa hlađenja metala, koja omogućava izlazak iz lokalnih optimuma prihvatanjem lošijih rešenja sa određenom verovatnoćom.

3.1 Brute-force metod (metod grube sile)

Brute-force (ili metod grube sile) predstavlja jednostavan, ali veoma neefikasan pristup rešavanju problema Maximum k-Cut. Ideja je da se isprobaju sve moguće podele čvorova u k grupa i za svaku dodelu izračuna vrednost preseka, a zatim se izabere najbolje rešenje.

Za dati graf sa n čvorova i k podela, ukupan broj kombinacija je k^n. Svaka kombinacija predstavlja potencijalno rešenje — tj. način na koji se čvorovi mogu rasporediti u grupe.

Nakon generisanja dodele za svaku ivicu gleda se da li spaja dva čvora iz različitih grupa, ako spaja, tada ta ivica doprinosi vrednosti preseka svojom težinom.

Na kraju, uzima se rešenje sa najvećom vrednošću preseka.

Prednosti: metod uvek daje tačno (optimalno) rešenje, pa je odlična osnova za poređenje sa heuristikama.

Mane: algoritam je eksponencijalno složen: za svaku od k^n dodela računa se suma po svim ivicama ($O(m)$), pa je ukupna složenost $O(m \cdot k^n)$. To znači da je metod primenljiv samo za male grafove (tipično za $n < 10$), jer eksponencijalni rast vrlo brzo postaje neizvodljiv.

```python
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
```

Slika 1: Brute-force Max-k-Cut. Enumeracija svih k^n dodela; čuva se najbolje rešenje (maksimalan presek).

```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
```

Slika 2: Funkcija vrednosti preseka. Sabira se težina svake ivice čiji su krajnji čvorovi u različitim grupama.

3.2 Greedy (Pohlepni algoritam)

```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)
```

Slika 3: Pohlepni algoritam.

```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)
```

Slika 4: Pohlepni algoritam sa sortiranjem.

Za razliku od brute-force metode koja ispituje sva moguća rešenja, pohlepni algoritam pokušava da brzo dođe do dobrog rešenja tako što u svakom koraku bira trenutno najbolju odluku. Kod našeg problema Maximum k-Cut to znači da čvorove smeštamo u onu grupu gde će trenutno najviše povećati vrednost preseka (zbir težina ivica koje idu ka drugim grupama).

Imamo dve varijante – osnovnu, gde čvorove obrađujemo redom kako ih graf daje, i poboljšanu, gde čvorove prvo sortiramo po stepenu (najpovezaniji čvorovi idu prvi). Ideja sortiranja je da se jako povezani čvorovi postave na pametnija mesta jer imaju najveći uticaj na presek. Na ovaj način se obezbeđuje da čvorovi sa najviše veza budu raspoređeni dok je raspodela još fleksibilna, jer kasnije više nema vraćanja i njihove odluke postaju ključne.

Prednost: greedy pristup je veoma brz i može da se koristi i na većim grafovima gde brute-force ne bi mogao ni da krene.

Mana: ne daje optimalno rešenje – lako može da zaglavi u lokalno najboljoj, ali ne i globalno najboljoj podeli. Zbog toga ga koristimo uglavnom kao poređenje ili kao dobar početni korak.

3.3 Genetski algoritam

Genetski algoritam je inspirisan Darvinovom idejom da preživljavaju najbolji, pa svako rešenje problema posmatramo kao jedinku koja ima svoj fitnes – broj koji pokazuje koliko je to rešenje dobro. Tokom svake generacije populacija se menja kroz nekoliko koraka: prvo ide selekcija gde se biraju najbolje jedinke koje će učestvovati u stvaranju nove generacije, zatim ukrštanje (crossover) gde od dva roditelja pravimo novu jedinku spajajući njihove dobre karakteristike, pa mutacija koja sa malom verovatnoćom menja neki deo rešenja kako bismo uneli raznolikost i možda došli do boljeg rešenja. Postoji i elitizam gde se nekoliko najboljih rešenja direktno prenosi u sledeću generaciju da ne bi bile izgubljene dobre karakteristike. Na ovaj način se kroz više generacija populacija unapređuje i približava sve boljem rešenju.

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

Slika 5: Inicijalizacija. Na početku pravimo početnu populaciju slučajnih rešenja – svaki čvor dobija neku od k grupa potpuno nasumično.

```
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
```

Slika 6: 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.

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

Slika 7: Selekcija (u kodu se koristila turnirska). Nasumično izvučemo nekoliko jedinki i biramo onu koja ima najveći fitnes. Tako dajemo prednost boljima, ali i dalje ostavljamo šansu slabijima, kako bismo zadržali raznolikost.

```
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
```

Slika 8: Ukrštanje (u kodu se koristi ravnomerno). U kodu, dete se pravi tako što svaki čvor ima jednaku šansu (50–50%) da nasledi grupu od roditelja 1 ili roditelja 2.

```python
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()
```

Slika 9: 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.

```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
```

Slika 10: Kod glavnog koda za genetski algoritam krećemo od inicijalizacije – nasumično napravimo početnu populaciju rešenja. Zatim u svakoj generaciji: prvo biramo jedinke koje će učestvovati u daljem razmnožavanju (selekcija), onda radimo ukrštanje (crossover) gde kombinujemo roditelje i pravimo novu jedinku, posle toga uz malu verovatnocu ide mutacija gde malo promenimo neko rešenje . Na kraju spojimo sve nove i najbolje iz prethodne generacije (elitizam) i tako dobijamo novu populaciju. Ovaj proces se ponavlja više puta dok ne dostignemo zadati kriterijum zaustavljanja – u našem kodu to je unapred definisan broj generacija.

3.4 Simulated Annealing (Simulirano kaljenje)

Simulirano kaljenje je algoritam inspirisan procesom kaljenja čelika. U toj tehnici čelik se najpre zagreva, pri čemu se njegove čestice pomeraju i menjaju kristalnu rešetku, a zatim se polako hladi. Tokom hlađenja čestice zauzimaju stabilnije pozicije, a sporije hlađenje dovodi do čvršće i otpornije strukture. Po analogiji, u algoritmu simuliranog kaljenja rešenje se u početku „zagreva“ dopuštanjem i lošijih poteza (kako bismo izbegli lokalne minimume), a zatim se postupnim „hlađenjem“ (smanjivanjem temperature) rešenje stabilizuje i približava globalnom optimumu.

```python
def initialize(graph, k) -> dict:
    return {node: random.randrange(k) for node in graph.nodes()}
```

Slika 11: Inicijalizacija: Na početku algoritma se uzima jedno proizvoljno (nasumično) rešenje, tj. svakom čvoru se dodeljuje neka grupa od 1 do k. Ovo rešenje obično nije optimalno, ali služi kao polazna tačka za dalje poboljšavanje kroz promene i poređenja vrednosti preseka.

```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
```

Slika 12: Funkcija vrednosti preseka. Sabira se težina svake ivice čiji su krajnji čvorovi u različitim grupama.

```python
def change_label(current_labels, graph, k):

    node = random.choice(list(graph.nodes()))
    current_label = current_labels[node]
    candidates = [g for g in range(k) if g != current_label]

    new_label = random.choice(candidates)
    new_labels = current_labels.copy()
    new_labels[node] = new_label
    return new_labels
```

Slika 13: Ova funkcija pravi novo susedno rešenje tako što nasumično izabere jedan čvor i promeni mu grupu.

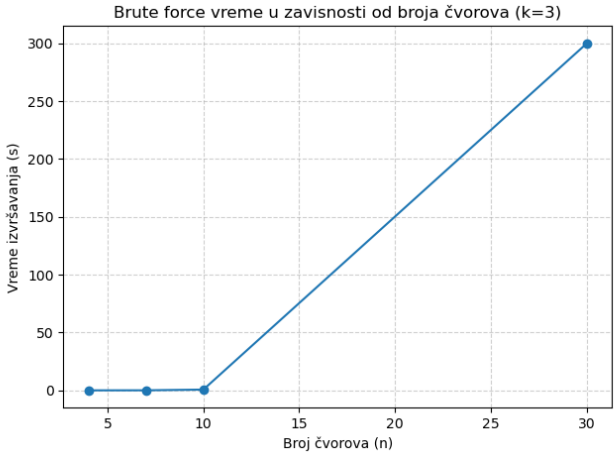
```python
def simulated_annealing_maxkcut(
    graph,
    k,
    initial_temperature: float = 1.0,
    cooling_rate: float = 0.995,
    stopping_temperature: float = 1e-4,
    max_iterations: int = 10000
):

    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
```

Slika 14: Ova funkcija implementira simulirano kaljenje za problem Maximum k-Cut. Algoritam kreće od početnog nasumičnog rešenja i računa njegovu vrednost preseka. U svakoj iteraciji pokušava da unapredi rešenje tako što menja grupu jednog slučajno izabranog čvora i računa novi presek. Ako novo rešenje donosi poboljšanje, prihvata se, a ako je lošije, može se prihvatiti sa određenom verovatnoćom koja zavisi od trenutne temperature. Tokom izvršavanja temperatura se smanjuje (hlađenje), pa je u početku algoritam sklon da prihvata i lošija rešenja da bi istražio prostor rešenja, dok kasnije postaje strožiji i zadržava uglavnom bolja. Na kraju vraća najbolje rešenje i njegovu vrednost preseka koje je pronašao u toku pretrage.

4 Poređenje algoritama

U ovom delu posvetićemo se analizom i upoređivanjem rezultata i složenosti različitih algoritama za rešavanje problema Maximum k-Cut. Na početku vidimo kratak pregled vremenske složenosti, a zatim prikazujemo i međusobno poređenje.

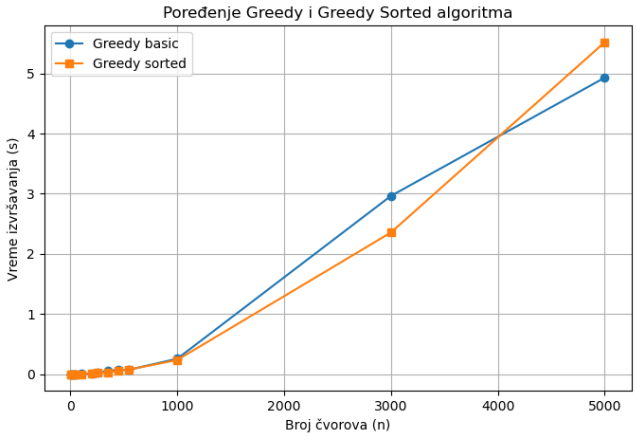
Brute-force algoritam sigurno daje tačno rešenje, jer prolazi kroz sve moguće podele čvorova u k grupa i bira optimalnu. Međutim, njegova najveća mana je izuzetno velika vremenska složenost i eksponencijalna eksplozija pri većim grafovima. Kao što smo videli u našim testovima, kod manjih primera od 4 ili 7 čvorova vreme izvršavanja je praktično zanemarljivo (skoro 0 sekundi), ali već na grafu od 30 čvorova vreme skače na oko 300 sekundi. U eksperimentalnom testiranju sa grafom od 10 čvorova dobili smo optimalno rešenje u vrlo kratkom vremenu, što potvrđuje da se ovaj algoritam primenjuje samo za male instance — tipično do otprilike 10 čvorova, jer za veće grafove postaje potpuno nepraktičan.



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

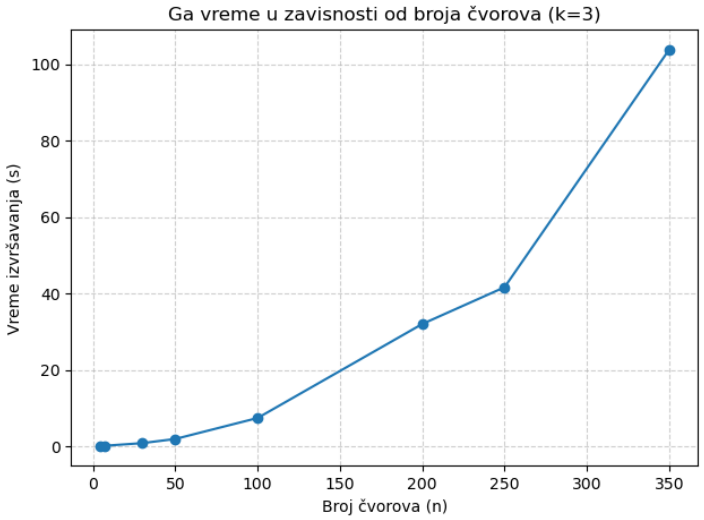
Greedy algoritam nam ne garantuje optimalno rešenje kao brute force, ali u praksi daje prilično dobra i približno optimalna rešenja u veoma razumnom vremenu. Dok brute force već kod grafa sa $n = 30$ postaje prezahtevan i ulazi u stotine sekundi zbog eksponencijalnog rasta, greedy se odlično ponaša i na grafovima sa nekoliko hiljada čvorova – na instancama do 5000 izvršava se za svega par sekundi.

Takođe, vidimo razliku između osnovne i sortirane varijante: sortirani greedy je napravljen da postavi „veće" čvorove (one sa većim stepenom) na značajnija mesta već na početku i time pokuša da dobije tačnije rešenje. Iako bi se moglo očekivati da dodatno sortiranje produži vreme izvršavanja, u praksi to nije nužno slučaj — sortirani greedy često radi podjednako brzo, pa čak i brže, jer unapređeni redosled obrade smanjuje broj nepotrebnih ažuriranja i omogućava efikasniji pristup podacima.



	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

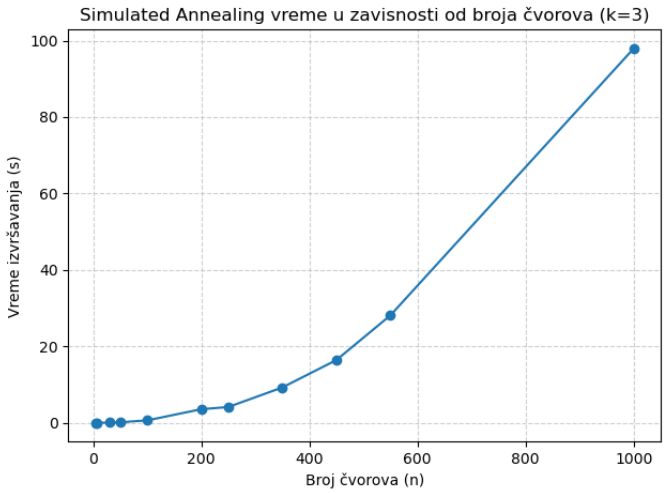
Genetski algoritam: Vidimo da se vreme izvršavanja genetskog algoritma postepeno povećava sa rastom broja čvorova. Kod malih instanci ($n = 4, 7$) vreme je praktično zanemarljivo (ispod 0.2 sekunde), a kod srednjih ($n = 30, 50, 100$) ostaje u veoma prihvatljivom opsegu od ispod 10 sekundi. Tek kod većih instanci ($n = 200$–350) primećujemo značajniji rast, ali i dalje daleko sporiji u poređenju sa brute-force metodom, jer genetski algoritam ne ispituje sve mogućnosti već koristi heurističko pretraživanje. Drugim rečima, eksponencijalna eksplozija kao kod brute-force metode se ne javlja, već rast vremena podseća više na polinomski – što omogućava da se testiraju i grafovi sa nekoliko stotina ili hiljada čvorova.



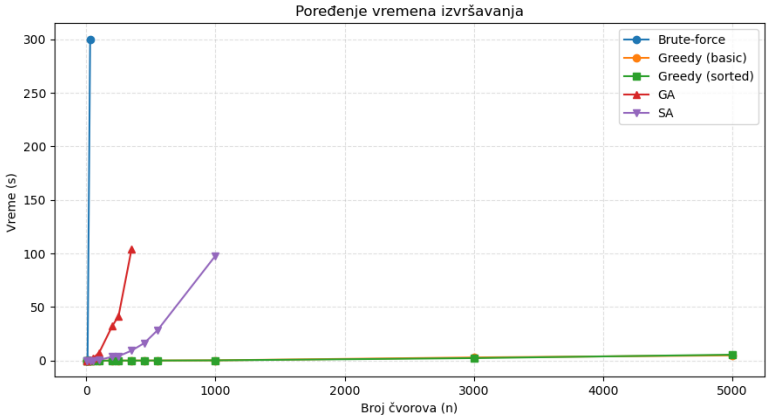
:

	Broj čvorova (n)	Vreme izvršavanja (s)
0	4	0.1894
1	7	0.2231
2	30	0.9192
3	50	2.0037
4	100	7.4947
5	200	32.0654
6	250	41.6805
7	350	103.7946

Simulated Annealing: Rezultati pokazuju da se simulirano kaljenje veoma dobro ponaša na manjim i srednjim instancama – kod grafa od $n = 100$ čvorova vreme izvršavanja ostaje ispod 1 sekunde, dok za $n = 200$–550 raste na desetak do dvadesetak sekundi. Tek kod velikih instanci, poput $n = 1000$, vreme prelazi u desetine, skoro stotine sekundi.



	Broj čvorova (n)	Vreme izvršavanja (s)
0	4	0.0328
1	7	0.0332
2	30	0.1104
3	50	0.1901
4	100	0.6824
5	200	3.6023
6	250	4.1755
7	350	9.2947
8	450	16.4635
9	550	28.1687
10	1000	97.9570



Slika 15: Vremena izvršavanja svih algoritama.

Na slici vidimo uporedni prikaz vremena izvršavanja svih algoritama. Kao što smo već pomenuli, brute-force metod uvek daje tačno (optimalno) rešenje, ali se njegova složenost eksponencijalno uvećava i postaje neprimenjiv već kod grafa sa $n = 30$, gde vreme prelazi 300 sekundi.

Sa druge strane, pohlepni algoritam (greedy) se pokazuje kao najefikasniji na našem problemu. On je sposoban da reši grafove i do 5000 čvorova za svega nekoliko sekundi, pa je daleko pogodniji za velike instance.

Genetski algoritam (GA) je svakako bolji od brute-force metode jer omogućava rešavanje većih instanci, ali u poređenju sa greedy i simuliranim kaljenjem pokazuje se kao vremenski skuplji. Već kod 350 čvorova vreme skače preko 100 sekundi, dok greedy i SA u tim uslovima i dalje rade znatno brže. Dakle, GA se po složenosti nalazi bliže brute-force metodu nego heuristikama koje se oslanjaju na lokalna pretraživanja.

Simulirano kaljenje (SA) zauzima srednju poziciju – brže je od GA, ali sporije od greedy-ja. Na primer, za graf sa 1000 čvorova vreme ide do oko 100 sekundi, dok GA već sa 350 dolazi do tog vremena.

7]:

	n	Greedy	Sorted Greedy	GA	SA
0	4	15	15	15	15
1	7	8	8	8	8
2	30	492	491	496	480
3	50	1097	1085	1118	1123
4	100	4295	4348	4372	4414
5	200	16996	16877	17082	17338
6	250	25753	25951	25486	26058
7	350	50169	50012	49316	50540

Slika 16: Poređenje vrednosti algoritama

Videli smo da Simulated Annealing (SA) ubedljivo daje najbolje rezultate, odnosno postiže najveći broj presečenih ivica između čvorova koji pripadaju različitim klasama. Jedini izuzetak je primer sa n = 30, gde je Genetski algoritam (GA) dao bolji rezultat, 496, dok je SA dao 480, što je razlika od 16

Očekuje se da postoji razlika između sortiranog i običnog Greedy pristupa, međutim vidi se da ona nije uvek prisutna, a kada postoji, najveća razlika se pojavila kod n = 100, gde je sortirani bio bolji od običnog algoritma za 53 (4348 naspram 4295).

Naravno, sve ove razlike zavise od parametara koje prosleđujemo algoritmima poput veličine populacije i kriterijuma zaustavljanja kod GA, brzine hlađenja i početne temperature kod SA ali i od same strukture grafova. Zbog toga se može desiti da bi, u nekoj drugačijoj strukturi grafa ili uz drugačije podešene parametre, neka druga metoda ostvarila bolje rezultate.

5 Zaključak

U ovom radu prikazani su različiti pristupi rešavanju problema Max-k-Cut. Prvo je razmotrena metoda grube sile koja uvek daje optimalno rešenje, ali je zbog svoje eksponencijalne složenosti neefikasna već za srednje i velike grafove. Zbog toga su primenjene heurističke i metaheurističke metode koje omogućavaju postizanje dovoljno dobrih rešenja u znatno kraćem vremenu.

Ispitani su **Greedy algoritam**, njegova **sortirana varijanta**, kao i **Genetski algoritam** i **Simulirano kaljenje**. Greedy pristup se pokazao kao najbrži, ali zbog lokalne prirode često ostaje zarobljen u lokalnim maksimumima i ne pronalazi globalno najbolje rešenje. Sortirana verzija greedy algoritma u većini slučajeva daje nešto tačnije rezultate, dok genetski algoritam i simulirano kaljenje zahtevaju duže vreme, ali uspešno pronalaze kvalitetnija rešenja i izbegavaju lokalne maksimume.

Eksperimentalni rezultati su pokazali da heuristički pristupi mogu uspešno da zamene metodu grube sile, posebno kod grafova sa većim brojem čvorova, jer postižu dobar kompromis između tačnosti i brzine. Na taj način potvrđeno je da heuristike predstavljaju praktično rešenje za složene optimizacione probleme gde je vreme izvršavanja ograničeno.

Literatura

[1] Frieze, A., & Jerrum, M. (1997). *Improved approximation algorithms for MAX k-CUT and MAX BISECTION*. Algorithmica, 18(1), 67–81.

[2] Goemans, M. X., & Williamson, D. P. (1995). *Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming*. Journal of the ACM, 42(6), 1115–1145.

[3] Newman, A. (2018). *Complex Semidefinite Programming and Max-k-Cut*. OASIcs–SOSA.

[4] Kann, Viggo. "MAXIMUM K-CUT." *Combinatorial Web Compendium*, CSC KTH, 2000-03-20, `https://www.csc.kth.se/~viggo/wwwcompendium/node88.html`.

[5] Kodovi sa predavanja i vežbi (interni materijali Fakulteta tehničkih nauka / kurs: Programiranje i optimizacija).