

Minimum Vertex Cover

Nikola Labus

02/09/2024

Sadrzaj:

1 Uvod:	3
2 Resenje:	3
Brute-Force	4
S-Metaheuristike	5
SK(Simulirano Kaljenje):	6
VNS(Variable Neighbourhood Search):	8
Genetski Algoritam:	10
3 Eksperimenti:	13
Eksperimenti zaključak:	20
4 Zaključak:	21
5 Literature:	22

1 Uvod:

-Problem Minimum Vertex Cover (MVC) predstavlja jedan od klasičnih NP-teških problema u teoriji grafova kao i u kombinatornoj optimizaciji. Cilj problema je pronaći najmanji skup čvorova u grafu takav da svaka grana u grafu bude incidentan barem jednom čvoru iz tog skupa. MVC se nalazi u srži mnogih praktičnih problema.

-Definicija:

Ako je graf dat pomocu $G = (V, E)$ gde je V skup čvorova, a E skup grana cilj je naci "pokrivač" čvorova za G , gde je $V' \subseteq V$ tako da, za svaku granu $(u, v) \in E$, barem jedan od čvorova u ili v pripadaju V' .

-Rešavanje problema Minimum Vertex Cover privuklo je veliku pažnju istraživača, sa mnogim pristupima koji su predloženi tokom godina. Klasični algoritmi poput aproksimativnih metoda, pretraga B&B, kao i heuristički pristupi, pružili su osnovu za istraživanje MVC problema. U skorije vreme, evolucioni algoritmi, uključujući genetske algoritme, pokazali su se kao efikasni alati za pronalaženje rešenja u velikim i složenim grafovima.

-Doprinosi drugih istraživača u ovom polju uključuju razvoj novih heuristika i metaheuristika koje omogućavaju efikasnije pretraživanje prostora rešenja. Na primer, implementacija adaptivnih strategija crossovera i selekcije u genetskim algoritmima značajno je unapredila performanse algoritama u kontekstu MVC problema ([Wang et al., 2018](#)). Dalji razvoj ovih metoda, kao i primena na specifične vrste grafova, predstavljaju ključne doprinose savremene nauke u ovom domenu.

2 Resenje:

Od gore navedenih tehnika, na prvi pogled su mi privukle pažnju S-metaheuristike, jer su problemi poput problema trgovca i još nekoliko koji se mogu modelovati grafovima bili veoma dobro rešeni korišćenjem S-metaheuristika. Takođe, neka nova "eksperimentalna" primena, kao što su genetski algoritmi, može vrlo dobro poslužiti u pronalaženju rešenja, dok brute force tehnika može biti korisna za probleme sa do 20 čvorova radi provere optimalnosti. Možemo početi sa brute force algoritmom, jer on predstavlja osnovnu tačku za rad na ovom problemu.

Brute-Force

Brute Force metod je jednostavan, ali neefikasan pristup rešavanju ovog problema, za male grafove je okej. Međutim, kako broj čvorova raste, vreme izvršavanja značajno se povećava, što čini ovaj metod nepraktičnim za veće grafove.

-Primer:

Za graf sa n čvorova, broj mogućih podskupova čvorova je 2^n , što znači da brute force algoritam mora da pregleda 2^n potencijalnih rešenja.

Vremenska složenost

Za $n=20$, algoritam bi morao da pregleda $2^{20} \approx 1,048,576$ podskupova.

Za $n=21$, broj podskupova raste na $2^{21} \approx 2,097,152$

Za $n=30$, broj podskupova je $2^{30} \approx 1,073,741,824$.

Pretpostavimo da jedan ciklus provere iznosi $t = 1\text{mkrsek}$ $T(n) = t * 2^n$,

Sto se može aproksimirati kao:

Za $n=20$, $T(20) = 1,048,576\text{mikrosek}$, sto iznosi **1s...**

Za $n=25$ $2^{25} = 33,554,432 \rightarrow T(25) = 33,554,432\text{mikrosek}$ sto iznosi **33 s**

Za $n=30$ to iznosi pribлизно **1100s**, sto je oko **18m**.

-Sto je kao što možemo da vidimo veoma loše vremensko rešenje odnosno eksponencijalno.

Algoritam

1: Inicijalizacija podataka

- Algoritam prvo uzima skup svih čvorova u grafu G i skladišti ih u listu *vertices*. Takođe, algoritam izračunava broj čvorova n u grafu.

2: Prolazak kroz sve moguće kombinacije čvorova

- Algoritam koristi dve ugnježdene petlje da bi prošao kroz sve moguće kombinacije čvorova. Spoljašnja petlja prolazi kroz sve veličine podskupova čvorova, počevši od veličine 1, pa do veličine n. Unutrašnja

petlja koristi funkciju ***combinations*** iz Python-ove biblioteke ***itertools*** da generiše sve podskupove čvorova veličine r (trenutne veličine podskupa).

3: Provera validnosti vertex cover-a

- Za svaki generisani podskup čvorova, algoritam proverava da li je taj podskup validan vertex cover, koristeći funkciju ***is_graph_minimum_vertex_cover***. Ova funkcija prolazi kroz sve grane u grafu i proverava da li je barem jedan od krajnijih čvorova svake grane u vertex cover-u. Ako postoji grana čiji oba krajnja čvora nisu u vertex cover-u, funkcija vraća False, što znači da podskup nije validan.

```
#BRUTE_FORCE
def is_graph_minimum_vertex_cover(G, vertex_cover):
    for u in G:
        for v in G[u]:
            if u not in vertex_cover and v not in vertex_cover:
                return False
    return True

def brute_force(G):
    vertices = list(G.nodes())
    n = len(vertices)

    for r in range(1, n+1):
        for subset in combinations(vertices, r):
            vertex_cover = set(subset)
            if is_graph_minimum_vertex_cover(G, vertex_cover):
                return vertex_cover, len(vertex_cover)

    return None, None
```

S-Metaheuristike

-Kada govorimo o **S-metaheuristikama**, nećemo se fokusirati na običnu lokalnu pretragu, već na njene modifikovane implementacije, kao što su ***Simulirano Kaljenje*** i **VNS** (Variable Neighborhood Search). Takođe, nećemo se previše baviti vremenskim ograničenjima, jer se u nastavku bavimo optimizacionim metodama. Najviše će zavisiti od parametara koje izaberemo.

Algoritmi

Pre nego što krenemo sa algoritmima malo da prokometarisemo inicijalizaciju, racunanje fitnesa i provere da li je vertex.

```

#S-METAHEURISTIC
#-----
#SHARED
def initial_solution(graph):
    num_nodes = len(graph.nodes())
    solution_list = [False] * num_nodes

    while not is_valid_cover(solution_list, graph):
        remaining_nodes = [i for i in range(num_nodes) if not solution_list[i]]
        random_node = random.choice(remaining_nodes)
        solution_list[random_node] = True

    return solution_list

def calculate_cost(vertex_cover, graph):
    return sum(vertex_cover)

def is_valid_cover(vertex_cover, graph):
    for u, v in graph.edges():
        if not vertex_cover[u] and not vertex_cover[v]:
            return False
    return True

```

Iinicijalizacija:

Iinicijalizacija za **VNS** (Variable Neighborhood Search) i **SK** (Simulirano Kaljenje) je ista i funkcioniše na sledeći način: nasumično se biraju čvorovi dok se ne formira validan vertex cover. Skup se smatra validnim kada se uključi poslednji čvor potreban da bi skup postao vertex cover. U ovom procesu koristi se funkcija kao što je **is_valid_cover**, koja proverava da li je dati vertex cover validan.

Fitness:

Fitness funkcija može se implementirati na dva načina: sabiranjem broja čvorova ili kao $\frac{1}{\epsilon + \text{suma_čvorova}}$. Oba pristupa su validna, ali sam se odlučio za sabiranje broja čvorova. Ovaj pristup je prikladan za S-metaheuristike, ali se pokazao kao manje efikasan za genetske algoritme, što ćemo detaljnije razmotriti kasnije.

SK(Simulirano Kaljenje):

Algoritam Simuliranog Kaljenja (SK):

1. Iinicijalizacija:

- Početno rešenje se generiše pomoću funkcije **initial_solution**. Početni trošak rešenja se izračunava pomoću **calculate_cost**.

2. Iteracija:

- U svakoj iteraciji, generiše se novo rešenje koristeći funkciju **get_neighbor**.

```

def get_neighbor(vertex_cover, graph):
    num_nodes = len(vertex_cover)
    new_solution = deepcopy(vertex_cover)
    random_idx = random.randrange(len(vertex_cover))

    new_solution[random_idx] = not new_solution[random_idx]

    neighbor = new_solution

    if not is_valid_cover(neighbor, graph):

        uncovered_edges = [(u, v) for u, v in graph.edges() if not neighbor[u] and not neighbor[v]]
        while uncovered_edges:
            u, v = random.choice(uncovered_edges)
            if not neighbor[u]:
                neighbor[u] = True
            if not neighbor[v]:
                neighbor[v] = True
            uncovered_edges = [(u, v) for u, v in graph.edges() if not neighbor[u] and not neighbor[v]]


    return neighbor

```

Get_Neighbor:

1. Nasumično se bira indeks čvora iz trenutnog rešenja I on se flipuje (ako je True postaje False I obrnuto)
 2. Proverava se da li je resenje validno I ako jeste to je to
 3. Ako nije listaju se sve grane koje nisu pokrivene I nasumicno se bira cvor za svaku granu I on se menja dok skup ne postane validan.
- o Trošak novog rešenja se računa, a napredak u trošku se beleži. Ako je novo rešenje bolje, ono se postavlja kao trenutno rešenje. Ako je lošije, postoji verovatnost da će biti prihvaćeno na osnovu temperature koja se smanjuje sa brojem iteracija.

3. Kraj:

- o Najbolje pronađeno rešenje i njegov trošak se vraćaju, zajedno sa napretkom troška tokom iteracija.

```

def simulated_annealing(graph, max_iter, k):
    current_solution = initial_solution(graph)
    current_cost = calculate_cost(current_solution, graph)
    best_solution = current_solution
    best_cost = current_cost
    cost_progress = [best_cost]

    for i in range(1, max_iter):
        neighbor = get_neighbor(current_solution, graph)
        neighbor_cost = calculate_cost(neighbor, graph)
        cost_progress.append(neighbor_cost)

        if neighbor_cost < current_cost:
            current_cost = neighbor_cost
            current_solution = neighbor
            if neighbor_cost < best_cost:
                best_solution = neighbor
                best_cost = neighbor_cost
        else:
            p = 1 / i ** k
            q = random.random()
            if q < p:
                current_cost = neighbor_cost
                current_solution = neighbor

    return best_solution, best_cost, cost_progress

```

Kao što možemo da vidimo, ovakav pristup uvek će težiti ka najboljem lokalnom rešenju zbog provere validnosti. Pomoću temperature i parametra **k** koji se prosleđuje, može doći do nasumičnog biranja lošijeg rešenja, što može doprineti pronalaženju globalnog optimuma. Još jedna implementacija koja je mogla da se uradi je da se, umesto provere validnosti, taksuje graf ako nije validni pokrivač tako što bismo na sumu čvorova u fitnesu dodali i broj nedostajućih grana puta 2 (dva čvora na jednu granu), ali to nije bilo potrebno.

VNS(Variable Neighbourhood Search):

```
def vns(graph, max_iters, k_max, move_prob):
    solution = initial_solution(graph)
    value = calculate_cost(solution, graph)
    cost_progress = [value]

    for i in range(1, max_iters):
        for k in range(1, k_max + 1):
            new_solution = shaking(solution, k, graph)
            new_solution = local_search_best_improvement(new_solution, graph)
            new_value = calculate_cost(new_solution, graph)
            if new_value < value or (new_value == value and random.random() < move_prob):
                value = new_value
                solution = new_solution
                break
        cost_progress.append(new_value)

    return solution, value, cost_progress
```

1.Inicijalizacija:

- Generiše se početno rešenje pomoću funkcije `initial_solution`. Početni trošak rešenja se izračunava pomoću `calculate_cost`.

2.Iteracija:

U svakoj iteraciji, za svaki nivo k od 1 do `k_max`, generiše se novo rešenje koristeći funkciju `shaking`.

```
def shaking(solution, k, graph):
    num_nodes = len(solution)

    k = min(k, num_nodes)

    chosen_indices = random.sample(range(num_nodes), k)

    new_solution = solution.copy()

    for idx in chosen_indices:
        new_solution[idx] = not new_solution[idx]

    if not is_valid_cover(new_solution, graph):

        uncovered_edges = [(u, v) for u, v in graph.edges() if not new_solution[u] and not new_solution[v]]

        while uncovered_edges:
            u, v = random.choice(uncovered_edges)
            if not new_solution[u]:
                new_solution[u] = True
            if not new_solution[v]:
                new_solution[v] = True

            uncovered_edges = [(u, v) for u, v in graph.edges() if not new_solution[u] and not new_solution[v]]
```

Shaking:

1. Nasumično se bira k čvorova iz trenutnog rešenja i flipuje se njihov status (ako je True postaje False i obrnuto).
 2. Ako novo rešenje nije validno, listaju se sve grane koje nisu pokrivenе i nasumično se bira čvor za svaku granu kako bi se dodao u pokrivač dok skup ne postane validan.
- Nakon generisanja novog rešenja, primenjuje se lokalna pretraga sa najboljim poboljšanjem koristeći funkciju local_search_best_improvement.

```
def local_search_best_improvement(solution, graph):  
    improved = True  
  
    while improved:  
        improved = False  
        best_solution = solution.copy()  
        best_value = calculate_cost(best_solution, graph)  
  
        for node in range(len(solution)):  
            if best_solution[node]:  
                temp_solution = best_solution.copy()  
                temp_solution[node] = False  
                if is_valid_cover(temp_solution, graph):  
                    temp_value = calculate_cost(temp_solution, graph)  
                    if temp_value < best_value:  
                        best_value = temp_value  
                        best_solution = temp_solution  
                        improved = True  
                elif random.random() < 0.1:  
                    best_solution = temp_solution  
                    best_value = temp_value  
                    improved = True  
  
        uncovered_edges = [(u, v) for u, v in graph.edges() if not best_solution[u] and not best_solution[v]]  
        for u, v in uncovered_edges:  
            temp_solution = best_solution.copy()  
            if not temp_solution[u]:  
                temp_solution[u] = True  
            if not temp_solution[v]:  
                temp_solution[v] = True  
            temp_value = calculate_cost(temp_solution, graph)  
            if temp_value < best_value:  
                best_value = temp_value  
                best_solution = temp_solution  
                improved = True  
  
        solution = best_solution  
  
    return solution
```

Local Search - Best Improvement:

1. Proverava se poboljšanje rešenja tako što se pokušavaju različite promene čvorova (uključivanje/isključivanje) i proverava se da li se trošak poboljšava.
2. Ako novo rešenje poboljšava trošak, postavlja se kao trenutno najbolje rešenje.

3. Takođe, proverava se pokrivač za nedostajuće grane i pokušava se poboljšanje dodavanjem potrebnih čvorova.
 - Ako je novo rešenje bolje ili ako je novo rešenje isto kao trenutno najbolje ali se prihvata sa verovatnoćom move_prob, ažurira se trenutno rešenje i trošak.
- 3.Kraj:**
- Vraća se najbolje pronađeno rešenje, njegov trošak i napredak troška tokom iteracija.

-Kao što možemo da vidimo, VNS pristup koristi varijaciju u okviru pretrage rešenja kako bi izbegao lokalna optima. Pomoću funkcije shaking nasumično se biraju i flipuju čvorovi u trenutnom rešenju, čime se istražuju različiti delovi prostora rešenja. Ako novo rešenje nije validno, dodatno se obrađuje kako bi postalo validno, što može doprineti pronalaženju boljeg rešenja.

-Implementacija lokalne pretrage sa najboljim poboljšanjem osigurava da se nastavi sa poboljšanjem trenutnog rešenja, dok je takođe moguće da se rešenje prihvati čak i ako nije značajno bolje, uz određenu verovatnoću move_prob. Ovaj pristup omogućava istraživanje različitih rešenja, čime se povećava šansa za pronalaženje globalnog optima. Takodje moglo se implementirati first_improvent, ali sa best_improvent testiranjem se video da daje bolja rešenja. Alternativno, mogla je biti primenjena dodatna taksacija za nevalidne pokrivače, ali to nije bilo potrebno u ovoj implementaciji.

Genetski Algoritam:

Kao što možemo da vidimo, Genetski Algoritam koristi princip evolucije za pretragu prostora rešenja. Početna populacija se sastoji od nasumično generisanih jedinki, koje se evoluiraju kroz više generacija. Selekcija, crossover i mutacija su ključni procesi u ovom algoritmu.

```

def genetic_algorithm(G, population_size, generations, mutation_rate, elitism_size_ratio, cross, selection):
    num_nodes = len(G.nodes())
    edges = list(G.edges())
    elitism_size = int(population_size * elitism_size_ratio)
    if elitism_size % 2 == 1:
        elitism_size = elitism_size + 1
    population = [Individual(num_nodes, edges) for _ in range(population_size)]
    new_population = deepcopy(population)

    cost_progress = []
    hof = HallOfFame()

    for generation in range(generations):

        population = sorted(population, key=lambda ind: ind.fitness, reverse=False)
        cost_progress.append(population[0].fitness)

        hof.update(population[0])

        new_population[:elitism_size] = population[:elitism_size]

        for i in range(elitism_size, population_size, 2):
            if(selection == "tournament"):
                parent1 = tournament_selection(population, population_size // 10)
                parent2 = tournament_selection(population, population_size // 10)
            elif(selection == "roulette_wheel"):
                parent1 = inverted_roulette_wheel_selection(population)
                parent2 = inverted_roulette_wheel_selection(population)

            if(cross == "single_point"):
                single_point_crossover(parent1, parent2, new_population[i], new_population[i+1])
            elif(cross == "uniform"):
                uniform_crossover(parent1, parent2, new_population[i], new_population[i+1])
            mutation(new_population[i], mutation_rate)
            mutation(new_population[i+1], mutation_rate)

            new_population[i].fitness = new_population[i].calc_fitness()
            new_population[i+1].fitness = new_population[i+1].calc_fitness()

        population = new_population.copy()

    best_individual = hof.best_individual
    best_solution = best_individual.code
    best_cost = best_individual.fitness

    return best_solution, best_cost, cost_progress

```

- **Selekcija:** Algoritam koristi različite metode selekcije, kao što su turnirska selekcija i obrnuta rulet selekcija, kako bi odabralo najbolje jedinke za reprodukciju. Turnirska selekcija nasumično bira grupu jedinki i bira najbolju iz te grupe, dok obrnuta rulet selekcija koristi obrnuto proporcionalnu verovatnoću fitnessa za odabir jedinki. Kod ruletske selekcije koristi se invertovana selekcija, jer u ovoj implementaciji najmanji fitness zapravo označava najbolje rešenje. Kada se koristi invertovana selekcija, najmanji fitness se tretira kao najveći fitness u obrnutoj proporcionalnosti, što omogućava pravilno poređenje i selekciju jedinki.

```

def tournament_selection(population, tournament_size):
    chosen = random.sample(population, min(tournament_size, len(population)))
    return min(chosen, key=lambda x: x.fitness)

```

```

def inverted_roulette_wheel_selection(population):
    epsilon = 1e-6
    fitness_values = [1 / (ind.fitness + epsilon) for ind in population]

    total_fitness = sum(fitness_values)
    probabilities = [f / total_fitness for f in fitness_values]
    cumulative_probabilities = [sum(probabilities[:i+1]) for i in range(len(probabilities))]

    r = random.random()
    for i, individual in enumerate(population):
        if r <= cumulative_probabilities[i]:
            return individual

```

- **Crossover:** Dve metode crossovera se koriste: jednokretni crossover i uniformni crossover. Kod jednokretnog crossovera, slučajan presečni položaj se bira i roditeljski hromozomi se kombinuju na osnovu tog preseka. Kod uniformnog crossovera, svaki gen se nasumično bira iz jednog od roditelja, omogućavajući veću raznovrsnost u potomstvu.

```

def uniform_crossover(parent1, parent2, child1, child2, crossover_prob=0.5):
    for i in range(len(parent1.code)):
        if random.random() < crossover_prob:
            child1.code[i] = parent1.code[i]
            child2.code[i] = parent2.code[i]
        else:
            child1.code[i] = parent2.code[i]
            child2.code[i] = parent1.code[i]

def single_point_crossover(parent1, parent2, child1, child2):
    random_pos = random.randrange(0, len(parent1.code))

    child1.code[:random_pos] = parent1.code[:random_pos]
    child1.code[random_pos:] = parent2.code[random_pos:]

    child2.code[:random_pos] = parent2.code[:random_pos]
    child2.code[random_pos:] = parent1.code[random_pos:]

```

- **Mutacija:** Mutacija se primenjuje na potomstvo sa određenom verovatnoćom kako bi se očuvala raznovrsnost u populaciji i omogućilo istraživanje novih rešenja. Ovo pomaže u izbegavanju lokalnih optimuma i pronalaženju boljih globalnih rešenja.

```

def mutation(individual, mutation_prob):
    for i in range(len(individual.code)):
        if random.random() < mutation_prob:
            individual.code[i] = not individual.code[i]

```

- **Elitizam:** Najbolje jedinke iz prethodne generacije se direktno prenose u novu generaciju, što osigurava da se najbolja rešenja ne izgube tokom evolucije.

Implementacija može koristiti različite selekciione i crossover metode koje mogu značajno uticati na performanse algoritma. Alternativno, mogla bi se koristiti drugačija strategija mutacije ili dodatni mehanizmi za očuvanje raznovrsnosti u populaciji, ali to nije bilo potrebno u ovoj implementaciji.

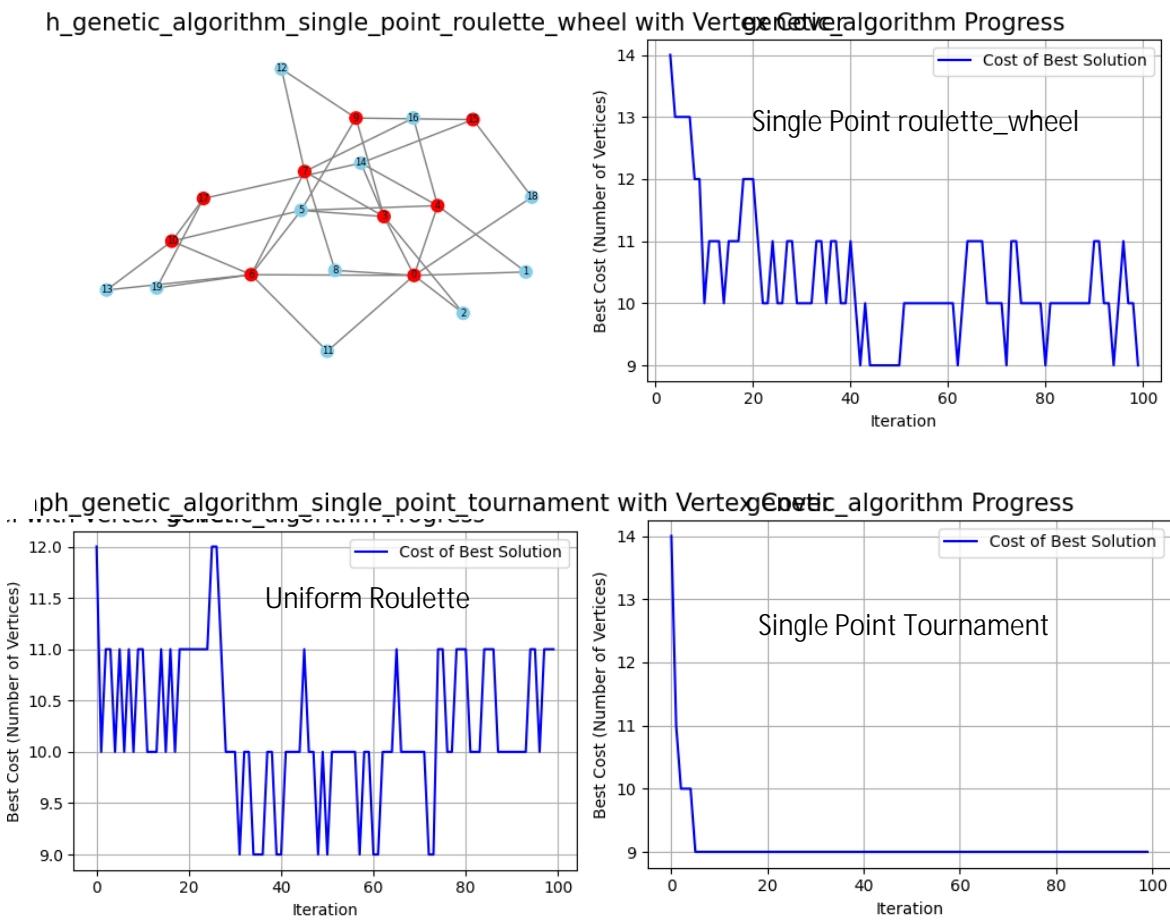
-U ovoj implementaciji se koristi kazneni fitness, gde se na fitness dodaje 2^* broj nedostajućih grana kako bi se rešio problem kod velikih broja čvorova (npr. 100), jer svi čvorovi mogu postati nevalidni i fitness može postati beskonačan. Ovo otežava selekciju i može dovesti do gubitka rešenja. Takođe, **Hall_of_Fame** je dodat za praćenje najboljeg mogućeg rešenja. Moglo bi se koristiti praćenje kroz populaciju, ali se ponekad gube rešenja i algoritam ne konvergira ka najboljem rešenju, posebno pri manjim populacijama (npr. do 1000 jedinki).

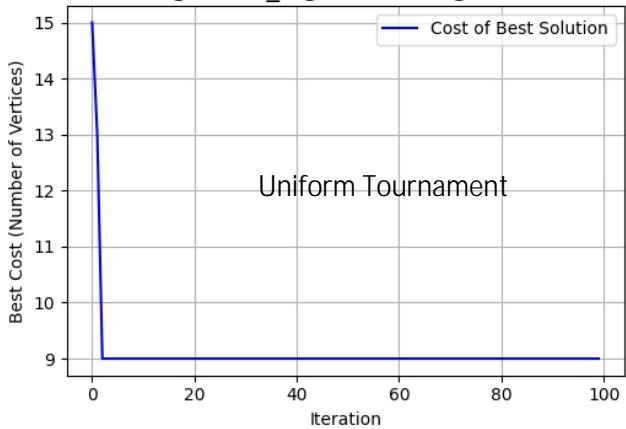
3 Eksperimenti: *Optimalnost*

Provera optimalnosti navedenih algoritama mozemo veoma lako da uradimo na malim grafovima, tako što ćemo proveriti rad algoritama na 10 nasumicno generisanih grafova (manjih od 20 cvorova) i rezultati su sledeći:

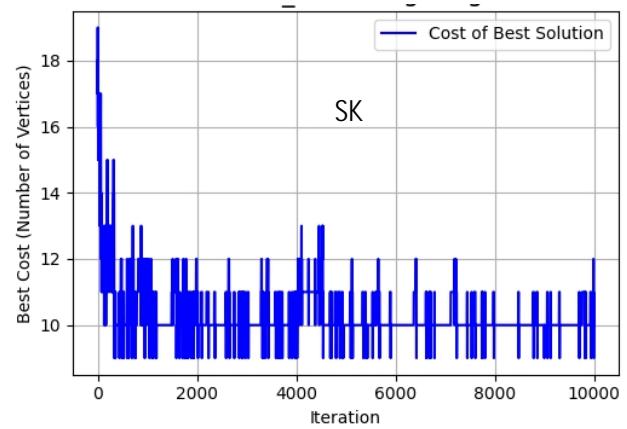
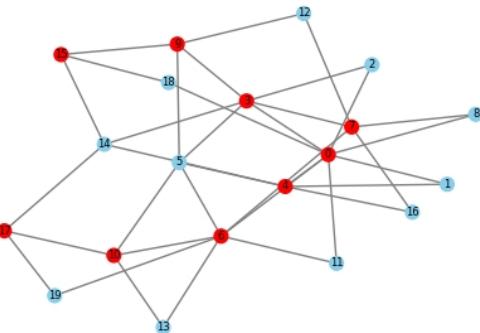
(Radi globuznosti necu prikazati svih 10 testiranje, imate sve na [githubu](#))

Primer 1: nx.barabasi_albert_graph(20, 2)

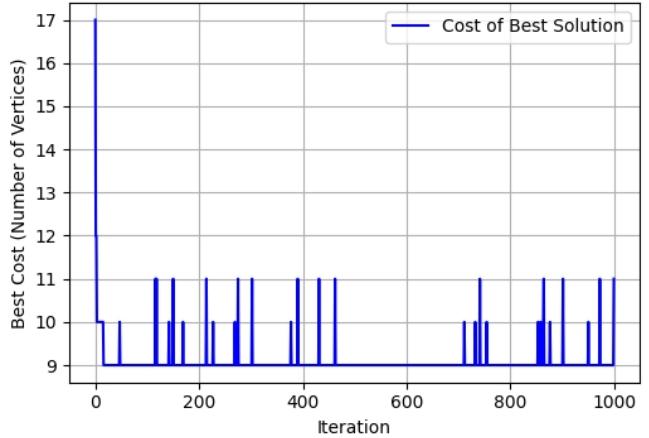




Barabási-Albert_Graph_vns with Vertex Cover



vns Progress



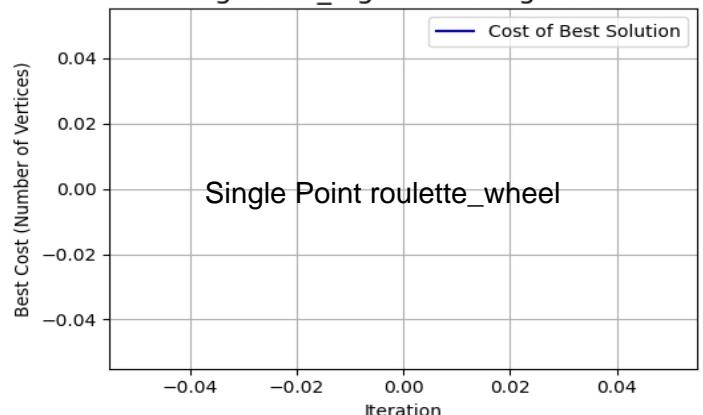
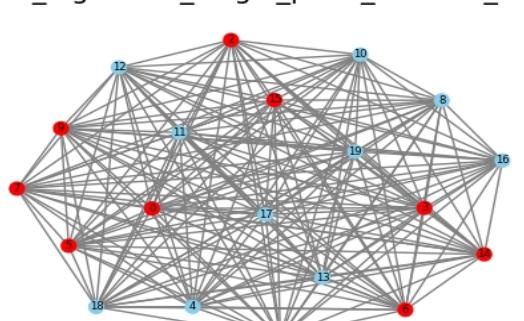
Možemo primetiti da kod genetskog algoritma turnirska selekcija veoma brzo teži ka rešenju u poređenju sa ruletskom selekcijom, dok ruletska selekcija pokazuje bolju sposobnost za istraživanje prostora rešenja, kao da se teže zaglavljuje u lokalnim optimumima. Uniformni crossover se takođe pokazuje efikasnijim od single-point crossovera, jer brže pronalazi rešenje.

S druge strane, algoritmi Simuliranog Kaljenja (SK) i Varijacionih Susedskih Struktura (VNS) pokazuju dobru sposobnost za istraživanje prostora rešenja i značajno su brži u pronalaženju optimalnog rešenja u poređenju sa genetskim algoritmom.

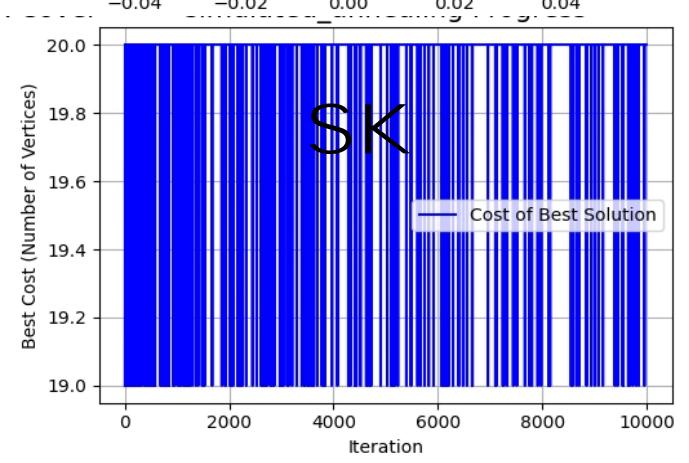
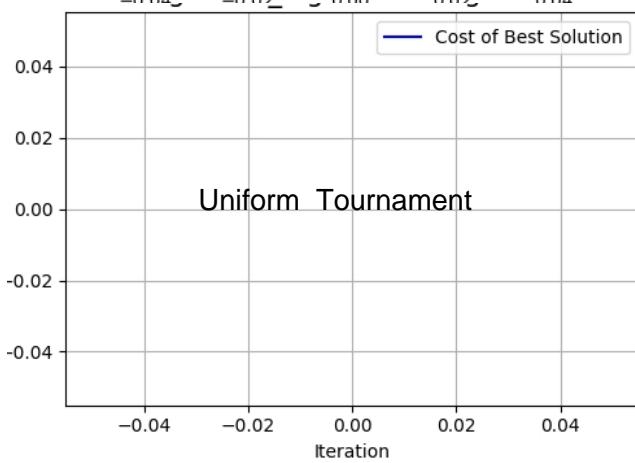
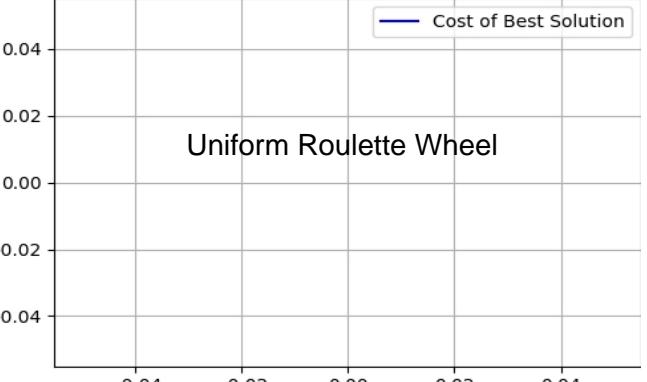
```
Graph: Barabási-Albert
GA SINGLE_POINT tournament Vertex Cover Found: 9 in 0.47432780265808105 seconds
GA UNIFORM tournament Vertex Cover Found: 9 in 0.5562789440155029 seconds
GA SINGLE_POINT roulette Vertex Cover Found: 9 in 1.9468615055084229 seconds
GA UNIFORM roulette Vertex Cover Found: 9 in 1.6083428859710693 seconds
BRUTE Vertex Cover Found: 9 in 0.47740650177001953 seconds
SIM Vertex Cover Found: 9 in 0.2867293357849121 seconds
VNS Vertex Cover Found: 9 in 0.447037935256958 seconds|
```

Primer2: `nx.complete_graph(20)`

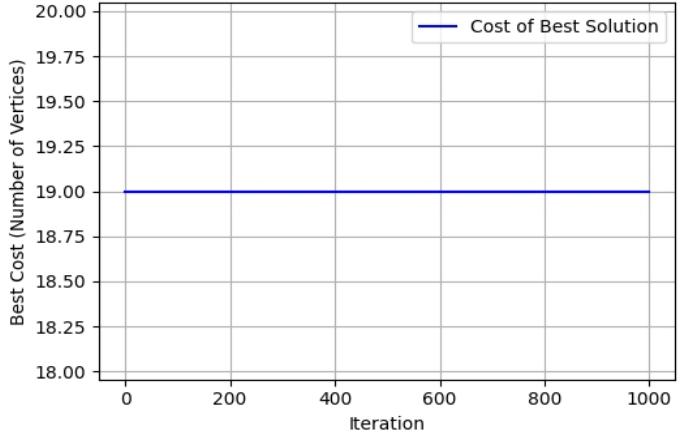
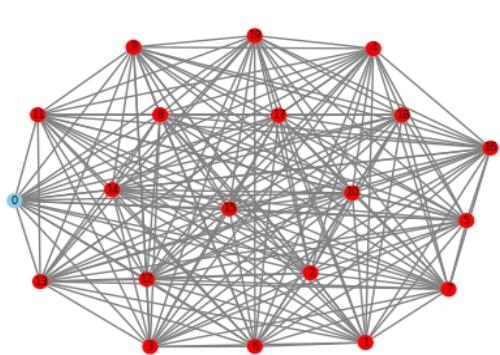
Genetic_algorithm_single_point_roulette_wheel with Vertex Cover Genetic_algorithm Progress



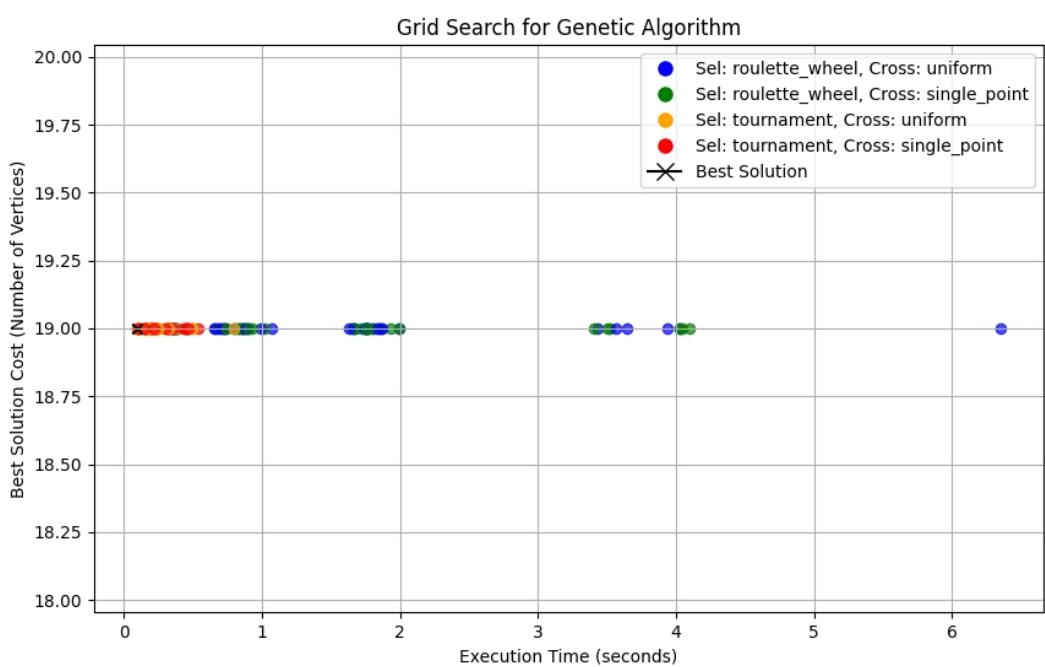
t with vertex Cover Genetic_algorithm Progress



Complete_Graph_vns with Vertex Cover



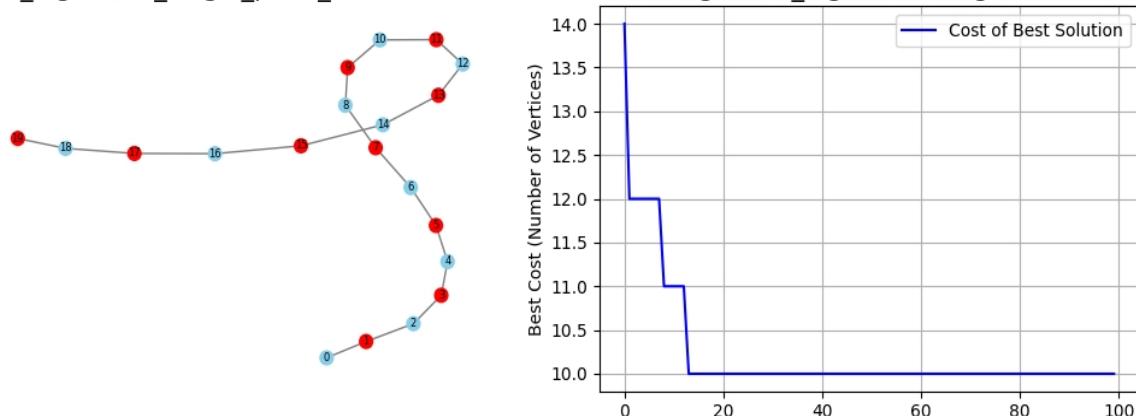
Ovde je napravljenja krucijalna Greska, jer nisam radio sa kaznenim poenima za fitness I GA nije mogao da nadje Gresku radeci samo sa inf jedinkama.



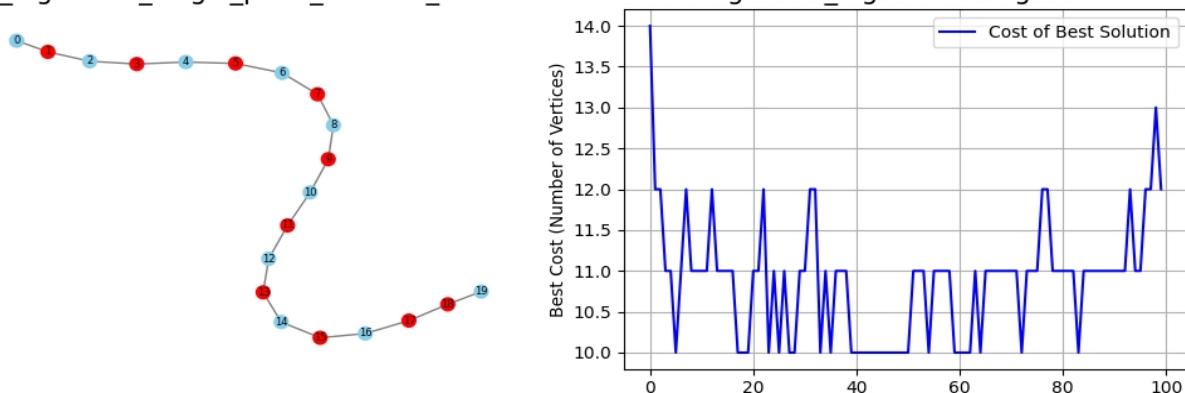
Ovde možete videti za isti graf, sa popravljenim fitnesom gde je dodat kazneni fitnes, kako je algoritam uspeo da pronađe optimalno rešenje. U ovom slučaju, dodavanje kaznenog fitnesa omogućilo je algoritmu da efikasnije izbegne lokalne minimume i da se približi globalnom optimumu. U grafu su korišćeni različiti parametri, pa ako vas zanima više, pogledajte kod za detalje.

Primer 3: nx.path_graph(20)

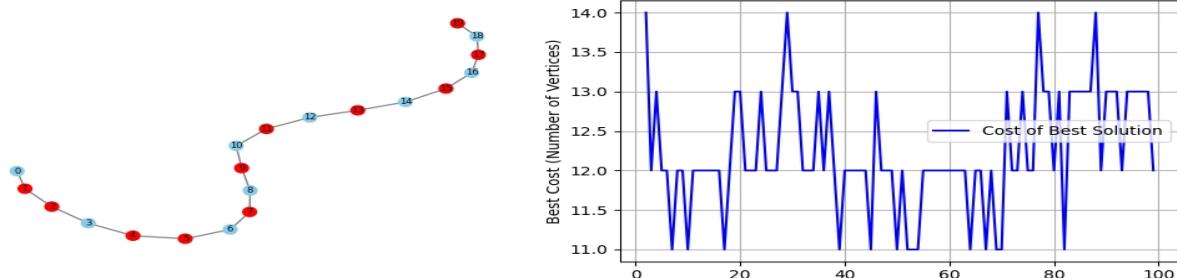
genetic_algorithm_single_point_tournament with Vertex Cover



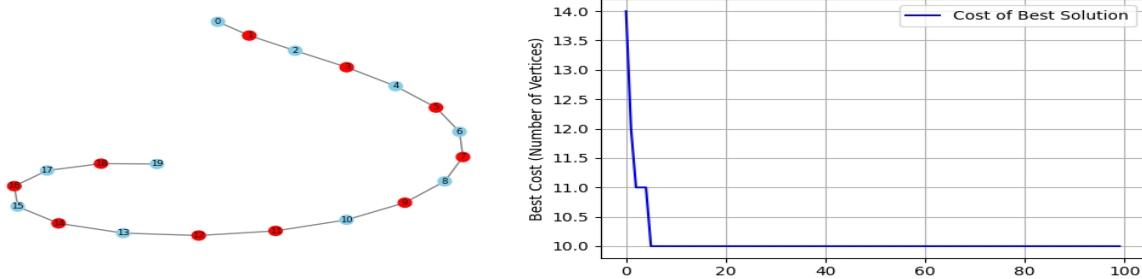
genetic_algorithm_single_point_roulette_wheel with Vertex Cover



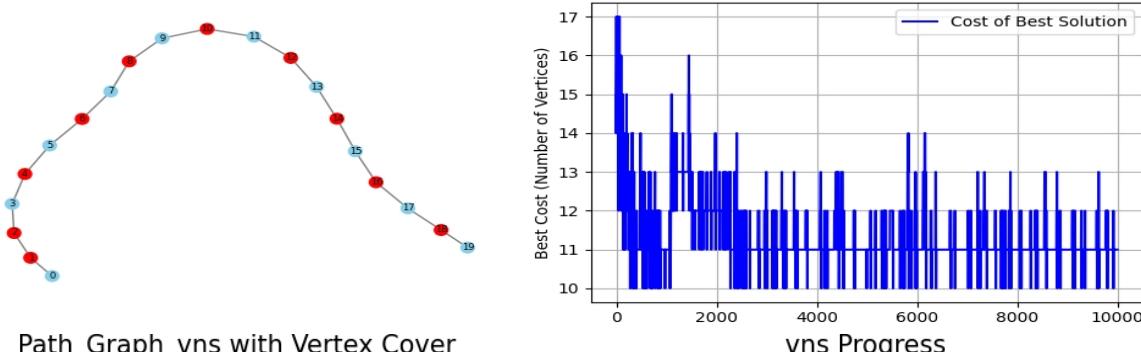
genetic_algorithm_uniform_roulette_wheel with Vertex Cover



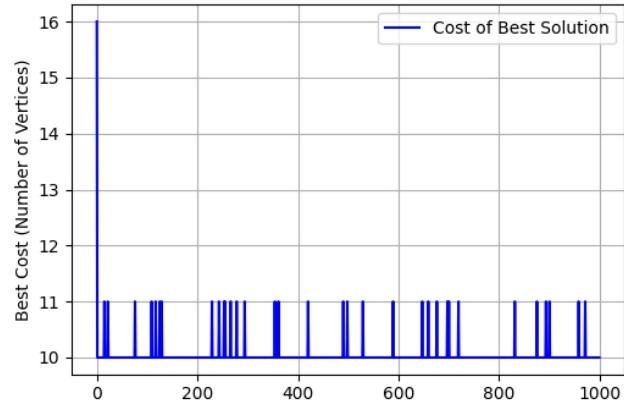
genetic_algorithm_uniform_tournament with Vertex Cover



:h_Graph_simulated_annealing with Vertex Cover



Path_Graph_vns with Vertex Cover



Graph: Path

```

GA SINGLE_POINT tournament Vertex Cover Found: 10 in 0.480405330657959 seconds
GA UNIFORM tournament Vertex Cover Found: 10 in 0.489290714263916 seconds
GA SINGLE_POINT roulette Vertex Cover Found: 10 in 1.7314262390136719 seconds
GA UNIFORM roulette Vertex Cover Found: 11 in 1.706796407699585 seconds
BRUTE Vertex Cover Found: 10 in 1.018876552581787 seconds
SIM Vertex Cover Found: 10 in 0.38988542556762695 seconds
VNS Vertex Cover Found: 10 in 0.3648841381072998 seconds

```

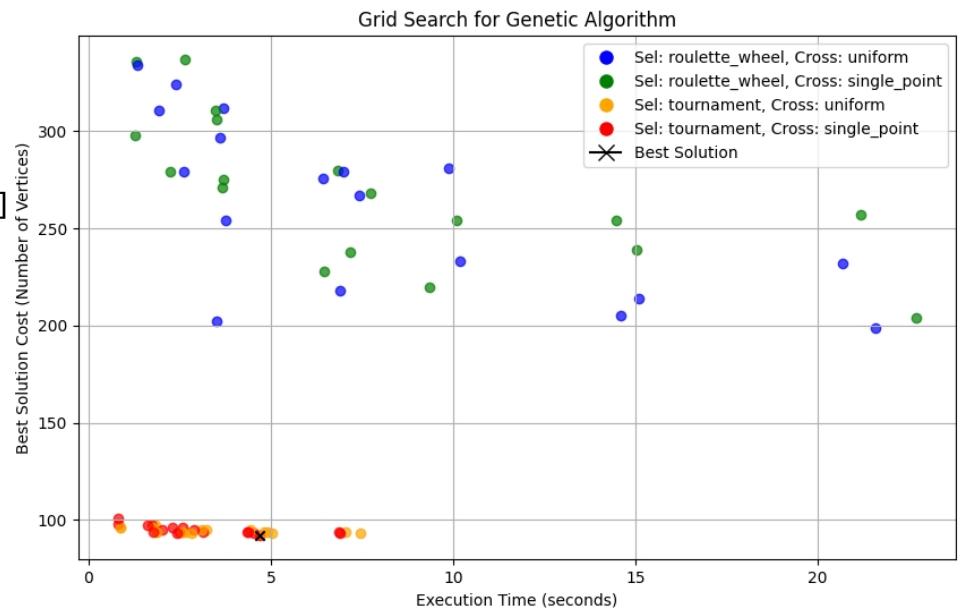
Ovde možemo videti još jedan primer u kojem su svi algoritmi, osim kombinacije uniformnog krosovanja i ruletske selekcije, pronašli tačno rešenje. Kao što sam ranije pomenuo, uniformno krosovanje u kombinaciji sa ruletskom selekcijom dovodi do veće eksploracije

prostora rešenja i nasumičnog biranja elemenata, što može rezultirati time da algoritam ne pronađe globalni maksimum.

Hajde da pogledamo neka poređenja koja sam radio da vidimo kako će se ruletska selekcija pokazati pri većem broju čvorova.

```
nx.erdos_renyi_graph(100, 0.5)
'population_size': [100, 200, 300],
'generations': [50, 100, 150],
'mutation_rate': [0.04],
'elitism_size_ratio': [0.05, 0.1],
'cross': ["uniform", "single_point"],
'selection': ["roulette_wheel", "tournament"]]
```

Best Parameters: {'population_size': 300, 'generations': 100, 'mutation_rate': 0.04, 'cross': 'single_point', 'selection': 'tournament', 'elitism_size_ratio': 0.05}
GA Best Cost: 92 in 4.6951 seconds
SIM Vertex Cover found: 91 in 7.78 seconds
VNS Vertex Cover found: 92 in 27.93 seconds

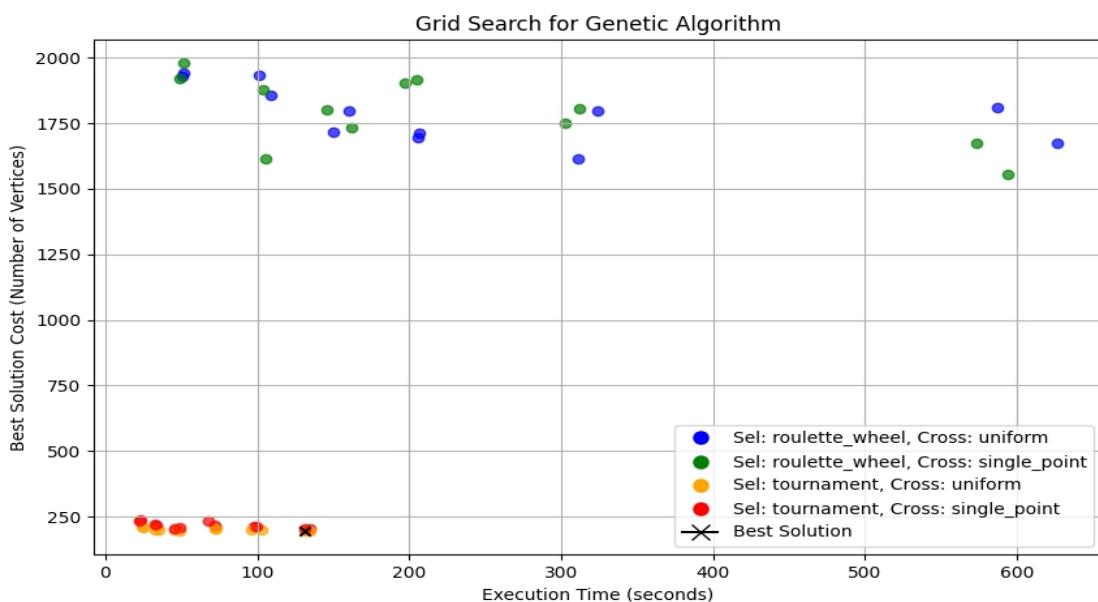


```

nx.erdos_renyi_graph(200, 0.5)
'population_size' : [100, 200, 300],
'generations' : [50, 100, 150],
'mutation_rate' : [0.04],
'elitism_size_ratio' : [0.05, 0.1],
'cross' : ["uniform", "single_point"],
'selection' : ["roulette_wheel", "tournament"]

```

Best Parameters:
'population_size': 300, 'generations': 100,
'mutation_rate': 0.04, 'cross': 'uniform',
'selection': 'tournament', 'elitism_size_ratio': 0.1
GA Best Cost: 211 in 16.3802 seconds
SIM Vertex Cover found: 191 in 31.06 seconds
VNS Vertex Cover found: 191 in 76.35 seconds

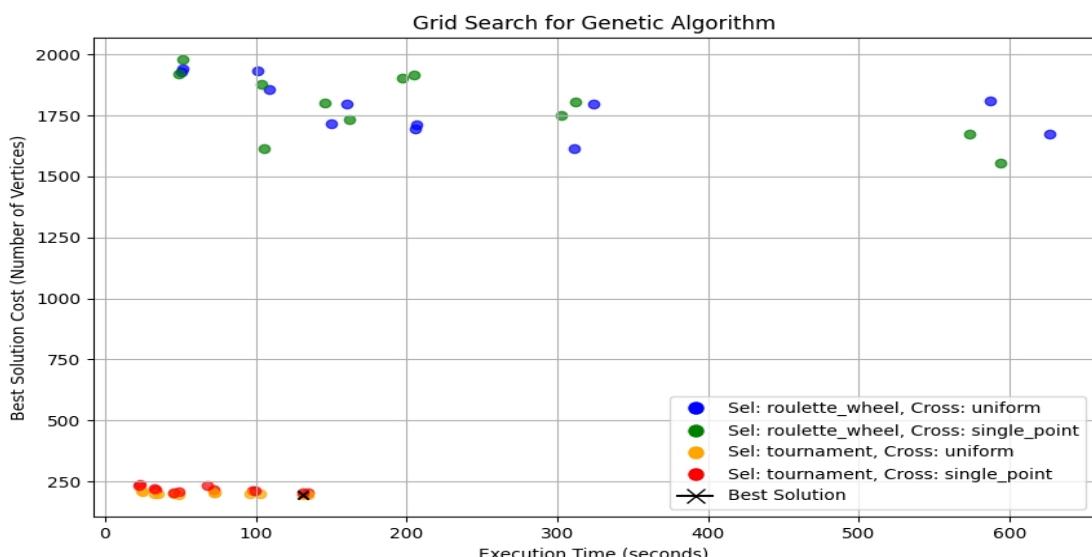


```

nx.erdos_renyi_graph(200, 0.5)
'population_size' : [400, 600, 800],
'generations' : [100, 300],
'mutation_rate' : [0.04],
'elitism_size_ratio' : [0.05, 0.1],
'cross' : ["uniform", "single_point"],
'selection' : ["roulette_wheel", "tournament"]

```

Best Parameters:
population_size': 800, 'generations': 300,
'mutation_rate': 0.04, 'cross': 'uniform',
'selection': 'tournament', 'elitism_size_ratio': 0.05
GA Best Cost: 195 in 131.4965 seconds
SIM Vertex Cover found: 192 in 32.28 seconds
VNS Vertex Cover found: 191 in 81.53 seconds



Mislio sam da će ruletska selekcija pronaći bolje rešenje i pokazati se efikasnijom sa povećanjem populacije i broja generacija, ali to se nije desilo. Čak ni turnirska selekcija nije uspela da nadmaši rešenja koja su pronašla S-metaheuristike. Pošto smo uočili da ruletska selekcija sa povećanjem populacije teži ka lošijim rešenjima, odlučujemo da je više ne koristimo.

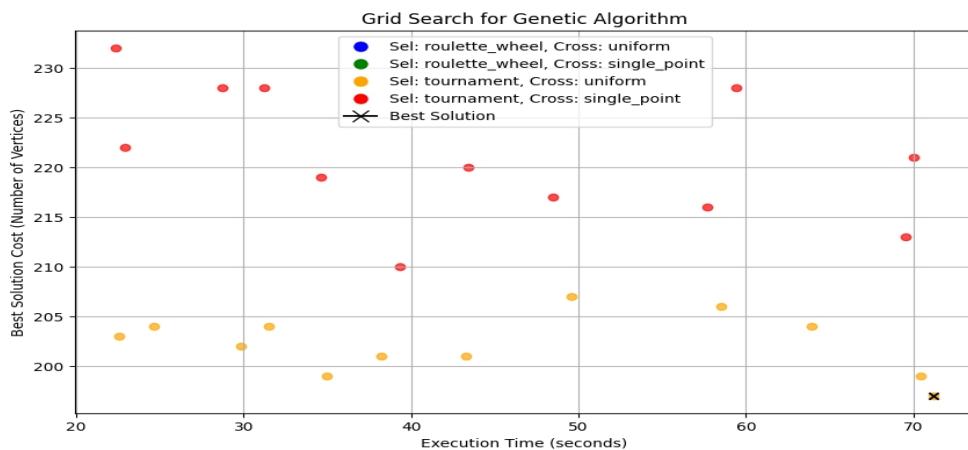
S druge strane, S-metaheuristike dva puta za redom daju isto rešenje sa 191 čvorom, što potvrđuje da dobro funkcionišu sa zadatim parametrima.

Eksperimenti zaključak:

Kao što sam ranije naglasio, na [GitHubu](#) možete pronaći još primera poređenja, ali ću ukratko izneti zaključke eksperimenata.

Optimalnost je potvrđena poređenjem algoritama sa brute force metodom na početku za 10 različitih grafova, pa o tome ne moramo dalje pričati. Što se tiče poređenja, odabrao sam metaheuristike kao osnovu za poređenje jer su pokazale izuzetno dobre rezultate. Nakon mnogih testiranja, parametri za metaheuristike su optimizovani, i odabrao sam one koji su davali najbolje rešenje (pogledajte kod na GitHubu).

Cilj eksperimenta je bio da se ispita da li se genetski algoritam može pokazati kao dobro rešenje. Odabrao sam dve selekcije i dva crossover-a, a nakon testiranja selekcija, dokazao sam da ruletska selekcija nije efikasna (možda je problem u fitness funkciji, ali smatram da je ona korektna). Što se tiče crossover-a, prikazaču još jedan primer gde sam testirao crossover-e na 200 čvorova kako bih utvrdio koji je bolji.



Generalno, kod manjih grafova (oko 100 čvorova) parametri poput 100 populacije i 100 generacija delovali su optimalno. Međutim, kako se broj čvorova i grana povećava, potrebno je povećati i broj populacije i generacija. Kada sam testirao sa 400, 600, i 800 populacije i 100 ili 300 generacija, najbolje rešenje je došlo iz populacije od 800 i 300 generacija, što mi sugerira da je potrebno još više populacije i generacija za optimalno rešenje (izvršavanje je trajalo 2 sata).

Nasuprot tome, heuristike su pronašle najbolje rešenje za 32 i 82 sekunde, što jasno pokazuje da su heuristike znatno efikasnije. Nakon ovih eksperimenata, zaključujem da su heuristike superiornije u odnosu na genetski algoritam za problem Minimum Vertex Cover. Iako genetski algoritam može pružiti dobro rešenje ako imate dovoljno vremena, heuristike su ipak bolje rešenje.

4 Zaključak:

U ovom radu sprovedeni su brojni eksperimenti i analize različitih algoritama za rešavanje problema Minimalnog Pokrivača Čvorova (**Minimum Vertex Cover**). Iako su genetski algoritmi često korišćeni za rešavanje ovakvih problema, rezultati eksperimenata pokazuju da se oni u ovom slučaju nisu pokazali kao najefikasniji pristup, posebno u poređenju sa metaheuristikama kao što su Simulated Annealing (**SA**) i Variable Neighborhood Search (**VNS**).

Genetski algoritmi, uprkos prilagođavanju parametara kao što su veličina populacije, broj generacija, selekcija i crossover metoda, nisu uspeli da pronađu bolje rešenje od metaheurističkih metoda. Uočeno je da, čak i sa povećanjem populacije i broja generacija, genetski algoritmi nisu postigli značajno poboljšanje u rešenju, dok su metaheuristike, sa pažljivo podešenim parametrima, konzistentno pružale optimalna rešenja u znatno kraćem vremenskom okviru.

Ovo nas dovodi do zaključka da, iako genetski algoritmi imaju potencijal za dobru eksploraciju rešenja, oni zahtevaju značajno više vremena i resursa da bi postigli rezultate slične ili blizu onih koje postižu metaheuristike. Sa druge strane, metaheurističke metode su se pokazale kao superiorniji izbor za rešavanje problema Minimalnog Pokrivača Čvorova u datim uslovima.

Pravci za dalje unapređenje:

- **Optimizacija fitnes funkcije:** Iako je fitnes funkcija korišćena u genetskom algoritmu bila korektna, moguće je istražiti alternativne metode penalizacije kako bi se dodatno poboljšala sposobnost algoritma da izbegne lokalne minimume.
- **Kombinacija metoda:** Uvođenje hibridnih pristupa koji kombinuju genetske algoritme sa metaheuristikama moglo bi da rezultira boljim rešenjima, uz zadržavanje prednosti obje metode..
- **Dodatna testiranja na kompleksnijim grafovima:** Proširenje eksperimenata na još veće i kompleksnije grafove, kao i upotreba različitih distribucija, moglo bi pružiti dodatne uvide u performanse analiziranih algoritama.

Ovim radom potvrđena je efikasnost metaheurističkih metoda za rešavanje problema Minimalnog Pokrivača Čvorova, dok su genetski algoritmi pokazali potrebu za daljim prilagođavanjem i optimizacijom kako bi dostigli slične rezultate.

5 Literature:

1. <https://www.csc.kth.se/~viggo/wwwcompendium/node10.html#3021>
2. [https://www.researchgate.net/publication/272182126 A GA based Approach to Find Minimal Vertex Cover](https://www.researchgate.net/publication/272182126_A_GA_based_Approach_to_Find_Minimal_Vertex_Cover)
3. <https://visualgo.net/en/mvc>
4. <https://core.ac.uk/download/pdf/82772116.pdf>
5. https://en.wikipedia.org/wiki/Vertex_cover#Approximate_evaluation
6. <https://github.com/sangyh/minimum-vertex-cover>
7. <https://github.com/sedgwickc/VertexCoverSearch/blob/main/VertexCover.py>
8. https://github.com/MATF-RI/Materijali-sa-vezbi/tree/master/2022_2023/live