Minimum metric dimension

Projakt izRačunarske inteligencije
Univerzitet u Beogradu
Matematički fakultet

Irina Marko
mi17243@alas.matf.bg.ac.rs

Anita Jovanović
mi17227@alas.matf.bg.ac.rs

28. septembar 2024.

Sadržaj

1 Definicija problema

Za dati graf G, problem minimalne metričke dimenzije je nalaženje najmanjeg mogućeg skupa čvorova S koji diferencira sve parove čvorova u grafu. To znači da svaka dva različita čvora u i v postoji barem jedan čvor w u skupu S tako da je rastojanje od u do w različito od rastojanja od v do w.

2 Primena grube sile

Metoda grube sile(eng. *Brute-force*) za nalaženje metrike dimenzije grafova podrazumeva ispitivanje svih mogućih podskupova čvorova i proveru da li svaki podskup zadovoljava uslov rezolventnog skupa. Ovaj pristup je teoretski moguć za manje grafove, ali postaje neefikasan za veće grafove. Sledeći kod implementira pristup grube sile.

```
def brute_force_metric_dimension(G, shortest_paths, start_time):
  n= len(G.nodes)
  for r in range(1,n+1):
    if time.time() - start_time >= 500:
      return -1

    for nodes in combinations(G.nodes, r):
      if is_resolving_set(G, nodes, shortest_paths):
        return nodes
  return None
```

Slika 1: Kod funkcije brute_force

Ovaj kod prolazi kroz sve kombinacije čvorova pomoću funkcije *combinations* iz biblioteke itertools i provera da li dati čvorovi ispunjavaju uslove rezultujućeg skupa.

```
def is_resolving_set(G, nodes, shortest_paths):
    for u, v in combinations(G.nodes, 2):
        distances_u = [shortest_paths[u][node] for node in nodes]
        distances_v = [shortest_paths[v][node] for node in nodes]

        if not any(dist_u != dist_v for dist_u, dist_v in zip(distances_u, distances_v)):
            return False
    return True
```

Slika 2: Kod funkcije is_resolving_set

Funkcija *is_resolving_set* proverava da li dati skup čvorova može da razlikuje sve parove čvorova u grafu na osnovu njihovih udaljenosti. Ovo je ključni deo algoritma, jer određuje da li je skup čvorova validan rešavajući skup za problem minimalne metricke dimenzije.

3 Optimizacija

3.1 Genetski algoritam

Genetski algoritam, inspirisan prirodnom evolucijom, započinje sa nasumično generisanom populacijom rešenja, od kojih svako ima svoju "prilagođenost"ili "fitnes". Bolja rešenja imaju veću verovatnoću da se reprodukuju i kombinuju,

simulirajući ukrštanje genetskog materijala, dok nasumične promene predstavljaju genetsku mutaciju. Ovaj proces se ponavlja kroz generacije s ciljem poboljšanja populacije.

Rezultat genetskog algoritma je tačno ili približno rešenje problema optimizacije. Reprezentacija jedinke naziva se hromozom ili genotip, a cilj je pronaći ekstremum funkcije cilja. U svakoj generaciji se procenjuje kvalitet jedinki pomoću funkcije prilagođenosti, pri čemu se kvalitetnije jedinke biraju za reprodukciju. Nakon selekcije, primenjuju se operatori ukrštanja i moguća mutacija.

Elitizam predstavlja strategiju u genetskim algoritmima koja omogućava da se najbolje prilagođene jedinke direktno prenesu u sledeću generaciju, bez izmena ili ukrštanja. Ovaj pristup pomaže u očuvanju najefikasnijih rešenja tokom evolucije populacije, sprečavajući da se izgube ili izmene tokom procesa reprodukcije. Elitizam takođe doprinosi bržoj konvergenciji algoritma ka boljim rešenjima.

Algoritam se završava kada se dostigne zadati broj generacija, željeni nivo kvaliteta populacije ili neki drugi uslov. Početna populacija se nasumično generiše.

```
def genetic_algorithm(G, population_size=100, generations=1000, mutation_rate=0.1, selection_strategy='roulette'):
    start_time = time.time()

    nodes_list = list(G.nodes)
    shortest_paths = precompute_shortest_paths(G)
    population = [random.sample(nodes_list, random.randint(1, len(nodes_list))) for _ in range(population_size)]

    best_solution = min(population, key=lambda nodes: fitness(G, nodes, shortest_paths))


    time_limit= 0.02

    for g in range(generations):

        if (time.time() - start_time) > time_limit:
              print(f"Time limit of {time_limit} seconds reached.")
              print("Best result set:", best_solution)
              print("Metric dimension:", len(best_solution))
              break

        fitness_values = [fitness(G, nodes, shortest_paths) for nodes in population]

        parents = select_two_parents(population, fitness_values, g, generations)
        if len(parents) != 2:
            raise ValueError(f"Expected 2 parents, but got {len(parents)}")

        parent1, parent2 = parents
        child = crossover(parent1, parent2)

        if random.random() < mutation_rate:
            child = mutate(child, nodes_list)

        population.append(child)
        population = sorted(population, key=lambda nodes: fitness(G, nodes, shortest_paths))[:population_size]


        current_best = min(population, key=lambda nodes: fitness(G, nodes, shortest_paths))
        if fitness(G, current_best, shortest_paths) < fitness(G, best_solution, shortest_paths):
            best_solution = current_best
    return best_solution
```

Slika 3: Kod funkcije *genetic_ algorithm*

Elitizam - Na početku svake iteracije identifikuju se elitisti, odnosno najbolje jedinke.

Selekcija - Iz populacije se biraju jedinke koje će preživeti i imati priliku za reprodukciju, pri čemu bolje prilagođene jedinke imaju veće šanse za izbor.

Ukrštanje - Odabrane jedinke se kombinuju kako bi se stvorile nove jedinke. Ovaj proces predstavlja simulaciju ukrštanja genetskog materijala koji se dešava kod živih organizama.

```python
def crossover(parent1, parent2):
    method_choice = random.randint(0, 2)

    set1, set2 = set(parent1), set(parent2)

    if method_choice == 0:

        result = list(set1.union(set2))
    elif method_choice == 1:

        result = list(set1.intersection(set2))
    else:

        combined = list(set1.union(set2))
        result = random.sample(combined, random.randint(1, len(combined)))

    return result
```

Slika 4: Kod funkcije *crossover*

Mutacija - U određenim situacijama, nova jedinka može doživeti mutaciju, što podrazumeva nasumične promene u njenom genetskom materijalu. Ova operacija doprinosi očuvanju raznolikosti unutar populacije, pri čemu nove jedinke zamenjuju prethodne.

```python
def mutate(nodes, nodes_list):

    mutation_choice = random.randint(0, 1)

    if mutation_choice == 0:

        if nodes:
            nodes.remove(random.choice(nodes))
        nodes.append(random.choice(nodes_list))
    else:

        if nodes:
            nodes[random.randint(0, len(nodes) - 1)] = random.choice(nodes_list)

    return nodes
```

Slika 5: Kod funkcije *mutatiom*

Svako rešenje u genetskom algoritmu predstavlja skup čvorova u grafu. Ova rešenja se nazivaju jedinkama ili hromozomima. Svaka jedinka može sadržati odabrane čvorove koji čine potencijalno rešenje. Funkcija prilagođenosti (fitness function) ocenjuje kvalitet svake jedinke na osnovu toga da li je odabrani skup čvorova resolving set. Ako jeste, funkcija vraća veličinu skupa; ako nije, vraća beskonačnost. Ova funkcija omogućava algoritmu da identifikuje koja rešenja su bolja od drugih.

```python
def fitness(G, nodes, shortest_paths):
    if is_resolving_set(G, nodes, shortest_paths):
        return len(nodes)
    else:
        return float('inf')
```

Slika 6: Kod funkcije *fitness*

Dva roditelja se biraju iz populacije korišćenjem različitih strategija selekcije (npr. turnirska selekcija ili rulet selekcija). Ovo osigurava da se bolje prilagođene jedinke imaju veće šanse za reprodukciju.

```python
def select_two_parents(population, fitness, iteration, max_iterations):

    if iteration < max_iterations // 2:

        return [tournament_selection(population, fitness, k=3) for _ in range(2)]
    else:

        return [roulette_wheel_selection(population, fitness) for _ in range(2)]

def tournament_selection(population, fitness, k):

    selected = random.sample(list(zip(population, fitness)), k)
    winner = max(selected, key=lambda x: x[1])
    return winner[0]

def roulette_wheel_selection(population, fitness):

    finite_population = [individual for individual, fit in zip(population, fitness) if fit != float('inf')]
    finite_fitness = [fit for fit in fitness if fit != float('inf')]

    if not finite_fitness:
        raise ValueError("No finite fitness values found in population")

    total_fitness = sum(finite_fitness)
    probabilities = [f / total_fitness for f in finite_fitness]
    return random.choices(finite_population, probabilities)[0]
```

Slika 7: Kod funkcije *select_two_parents*

3.2 Variable neighborhood search (VNS)

VNS (Variable Neighborhood Search) je metaheuristička metoda namenjena rešavanju kombinatornih optimizacionih problema. Osnovna ideja ove tehnike je istraživanje prostora rešenja kroz različite "okoline", čime se obezbeđuje šira pretraga i mogućnost pronalaženja boljih rešenja.

Jedna od ključnih karakteristika VNS-a je korišćenje različitih okolina, pri čemu svaka okolina predstavlja poseban način modifikacije rešenja. Ova raznolikost omogućava istraživanje različitih delova prostora rešenja.

VNS dinamički menja okruženja tokom pretrage kako bi se sprečilo zaglavljivanje u lokalnim optimumima. Kada se u jednoj okolini postigne određeni nivo uspeha, prelazi se na drugu kako bi se nastavilo sa pretragom.

Ova metoda kombinuje intenzivnu pretragu (usmerenu na lokalna poboljšanja) s diverzifikacijom (istraživanje novih rešenja) s ciljem pronalaženja optimalnog rešenja.

```python
def VNS(G, max_iterations=1000, time_limit=0.2):
    start_time = time.time()
    nodes_list = list(G.nodes)
    shortest_paths = precompute_shortest_paths(G)
    current_solution = generate_initial_solution(nodes_list)
    best_solution = current_solution

    for iteration in range(max_iterations):
        if (time.time() - start_time) > time_limit:
            print(f"Time limit of {time_limit} seconds reached.")
            print("Best result set:", best_solution)
            print("Metric dimension:", len(best_solution))
            break

        new_solution = shaking(current_solution, nodes_list)
        improved_solution = local_search(new_solution, nodes_list, G, shortest_paths)

        if fitness(G, improved_solution, shortest_paths) < fitness(G, best_solution, shortest_paths):
            print('New best solution', improved_solution)
            best_solution = improved_solution
            current_solution = improved_solution
        else:
            current_solution = improved_solution

    return best_solution
```

Slika 8: Kod funkcije *vns*

VNS u ovom kodu koristi se za rešavanje problema pronalaženja minimalne metricke dimenzije grafova. Evo kako funkcioniše proces:
-Glavna petlja VNS-a: glavna petlja se izvršava do definisanog broja iteracija (*max_iterations*) ili dok ne istekne vremensko ograničenje (*time_limit*)
-Shaking: zamućivanje rešenja-funkcija shaking nasumično menja trenutno rešenje dodavanjem, uklanjanjem ili zamenom čvora. Ovo omogućava istraživanje novog dela prostora rešenja.
-Lokalna pretraga: unapređenje rešenja-funkcija local_search se koristi za optimizaciju novog rešenja. Ova funkcija koristi tri operacije:
-dodavanje čvora
-uklanjanje čvora
-zamena čvora
-Upoređivanje rešenja: nakon svake iteracije, novo rešenje se upoređuje s trenutnim najboljim rešenjem. Ako je novo rešenje bolje (ima manju metricku dimenziju), postaje novo najbolje rešenje.

Nakon što sve iteracije završe (ili se postigne vremensko ograničenje), funkcija vraća najbolje rešenje pronađeno tokom pretrage. Kroz kombinaciju nasumične promene rešenja (shaking) i intenzivne lokalne pretrage, VNS dinamički istražuje različite delove prostora rešenja i osigurava da se ne zaglavi u lokalnim optimumima. Na taj način, efikasno traži minimalnu metricku dimenziju grafova.

```
def local_search_add(solution, G, nodes_list, shortest_paths):
    best_local_solution = solution[:]
    best_local_fitness = fitness(G, solution, shortest_paths)
    candidates = list(set(nodes_list) - set(solution))
    for node in candidates:
        new_solution = solution + [node]
        new_fitness = fitness(G, new_solution, shortest_paths)
        if new_fitness < best_local_fitness:
            best_local_solution = new_solution
            best_local_fitness = new_fitness
    return best_local_solution

def local_search_remove(solution, G, shortest_paths):
    best_local_solution = solution[:]
    best_local_fitness = fitness(G, solution, shortest_paths)
    for i in range(len(solution)):
        new_solution = solution[:i] + solution[i+1:]
        new_fitness = fitness(G, new_solution, shortest_paths)
        if new_fitness < best_local_fitness:
            best_local_solution = new_solution
            best_local_fitness = new_fitness
    return best_local_solution

def local_search_swap(solution, G, nodes_list, shortest_paths):
    best_local_solution = solution[:]
    best_local_fitness = fitness(G, solution, shortest_paths)
    for i in range(len(solution)):
        for node in nodes_list:
            if node not in solution:
                new_solution = solution[:]
                new_solution[i] = node
                new_fitness = fitness(G, new_solution, shortest_paths)
                if new_fitness < best_local_fitness:
                    best_local_solution = new_solution
                    best_local_fitness = new_fitness
    return best_local_solution

def local_search(solution, nodes_list, G, shortest_paths):
    best_solution = solution[:]
    improved = True

    while improved:
        improved = False
        for operation in ['add', 'remove', 'swap']:
            if operation == 'add':
                new_solution = local_search_add(best_solution, G, nodes_list, shortest_paths)
            elif operation == 'remove':
                new_solution = local_search_remove(best_solution, G, shortest_paths)
            elif operation == 'swap':
                new_solution = local_search_swap(best_solution, G, nodes_list, shortest_paths)

            if fitness(G, new_solution, shortest_paths) < fitness(G, best_solution, shortest_paths):
                best_solution = new_solution
                improved = True

    return best_solution
```

Slika 9: Kod funkcije lokalne pretrage za pronalazak najboljeg rešenja

4 Optimizacija parametara

Jedna od najčešće korišćenih tehnika za optimizaciju parametara je *grid search*. Ova metoda uključuje definisanje skupa vrednosti za svaki hiperparametar, a zatim sistematsko isprobavanje svake moguće kombinacije. Cilj je minimizovati ili maksimizovati određeni kriterijum, kao što je tačnost, preciznost ili vreme izvršavanja.

U našem istraživanju, optimizacija parametara je primenjena na genetski algoritam i VNS kako bismo identifikovali optimalne vrednosti koje poboljšavaju performanse u pronalaženju minimalne metricke dimenzije velikih grafova. Ova strategija nam je omogućila da poboljšamo efikasnost i tačnost naših algoritama, što je dovelo do boljih rešenja u kraćem vremenskom periodu.

4.1 Primena optimizacije parametara na genetski algoritam

Problem minimalne metričke dimenzije se fokusira na pronalaženje minimalnog razrešavajućeg skupa čvorova koji može jedinstveno identifikovati sve preostale čvorove u grafu na osnovu najkraćih putanja.

Funkcije poput generate_connected_erdos_renyi, generate_connected_watts_strogatz, itd., generišu različite tipove povezanih grafova, koji se koriste za testiranje algoritma.

Funkcija optimize_parameters isprobava različite kombinacije parametara (veličina populacije, broj generacija, stopa mutacije, strategije selekcije) da bi se našla najbolja kombinacija za dati graf.

U glavnoj petlji: run_genetic_algorithm_with_params(G, population_size, generations, mutation_rate, selection_strategy, time_limit) pokreće genetski algoritam, iterira kroz generacije, selektuje roditelje, vrši crossover, mutira potomstvo, i održava najbolju soluciju tokom procesa.

```
def optimize_parameters(G, population_sizes, generations_list, mutation_rates, selection_strategies, time_limit):

    best_solution_overall = None
    best_fitness_overall = float('inf')
    best_params_overall = None

    for population_size in population_sizes:
        for generations in generations_list:
            for mutation_rate in mutation_rates:
                for selection_strategy in selection_strategies:
                    print(f"Running with population: {population_size}, generations: {generations}, mutation: {mutatic


                    best_solution, best_fitness, elapsed_time = run_genetic_algorithm_with_params(
                        G, population_size, generations, mutation_rate, selection_strategy, time_limit
                    )


                    if elapsed_time is None:
                        continue


                    if best_fitness < best_fitness_overall:
                        best_solution_overall = best_solution
                        best_fitness_overall = best_fitness
                        best_params_overall = (population_size, generations, mutation_rate, selection_strategy)


                    print(f"Finished parameters with elapsed time {elapsed_time:.2f} seconds and fitness {best_fitnes:

    print(f"Best overall parameters: {best_params_overall} with fitness {best_fitness_overall}")
    return best_params_overall, best_solution_overall, best_fitness_overall
```

Slika 10: Kod koji prikazuje optimizaciju parametara

```python
def run_genetic_algorithm_with_params(G, population_size, generations, mutation_rate, selection_strategy, time_limit):

    if not nx.is_connected(G):
        print("Graph is not connected, skipping this graph.")
        return None, None, None

    nodes_list = list(G.nodes)
    shortest_paths = precompute_shortest_paths(G)
    population = [random.sample(nodes_list, random.randint(1, len(nodes_list))) for _ in range(population_size)]

    best_solution = min(population, key=lambda nodes: fitness(G, nodes, shortest_paths))

    start_time = time.time()

    for g in range(generations):

        if (time.time() - start_time) > time_limit:
            print(f"Time limit exceeded in generation {g}, stopping algorithm.")
            break

        print(f"Generation {g} / {generations}, population size: {len(population)}")

        fitness_values = [fitness(G, nodes, shortest_paths) for nodes in population]
        parents = select_two_parents(population, fitness_values, g, generations)

        parent1, parent2 = parents
        child = crossover(parent1, parent2)

        if random.random() < mutation_rate:
            child = mutate(child, nodes_list)

        population.append(child)
        population = sorted(population, key=lambda nodes: fitness(G, nodes, shortest_paths))[:population_size]

        current_best = min(population, key=lambda nodes: fitness(G, nodes, shortest_paths))
        if fitness(G, current_best, shortest_paths) < fitness(G, best_solution, shortest_paths):
            best_solution = current_best


        if (time.time() - start_time) > time_limit:
            print(f"Time limit exceeded after generation {g}, stopping algorithm.")
            break

    elapsed_time = time.time() - start_time
    return best_solution, fitness(G, best_solution, shortest_paths), elapsed_time
```

Slika 11: Pokretanje genetskog algoritma

4.2 Primena optimizacije parametara na VNS

Funkcija optimize_parameters(G, max_iterations_list, time_limits) testira različite kombinacije maksimalnog broja iteracija i vremenskih ograničenja kako bi pronašla najbolje parametre za VNS.

```python
def run_genetic_algorithm_with_params(G, population_size, generations, mutation_rate, selection_strategy, time_limit):

    if not nx.is_connected(G):
        print("Graph is not connected, skipping this graph.")
        return None, None, None

    nodes_list = list(G.nodes)
    shortest_paths = precompute_shortest_paths(G)
    population = [random.sample(nodes_list, random.randint(1, len(nodes_list))) for _ in range(population_size)]

    best_solution = min(population, key=lambda nodes: fitness(G, nodes, shortest_paths))

    start_time = time.time()

    for g in range(generations):

        if (time.time() - start_time) > time_limit:
            print(f"Time limit exceeded in generation {g}, stopping algorithm.")
            break

        print(f"Generation {g} / {generations}, population size: {len(population)}")

        fitness_values = [fitness(G, nodes, shortest_paths) for nodes in population]
        parents = select_two_parents(population, fitness_values, g, generations)

        parent1, parent2 = parents
        child = crossover(parent1, parent2)

        if random.random() < mutation_rate:
            child = mutate(child, nodes_list)

        population.append(child)
        population = sorted(population, key=lambda nodes: fitness(G, nodes, shortest_paths))[:population_size]

        current_best = min(population, key=lambda nodes: fitness(G, nodes, shortest_paths))
        if fitness(G, current_best, shortest_paths) < fitness(G, best_solution, shortest_paths):
            best_solution = current_best


        if (time.time() - start_time) > time_limit:
            print(f"Time limit exceeded after generation {g}, stopping algorithm.")
            break

    elapsed_time = time.time() - start_time
    return best_solution, fitness(G, best_solution, shortest_paths), elapsed_time
```

Slika 12: Kod funkcije *optimize_parameters*

VNS(G, max_iterations, time_limit) pokreće varijantno nasumično pretraživanje i beleži najbolje rešenje tokom iteracija

local_search(solution, nodes_list, G, shortest_paths) pokušava da poboljša trenutnu soluciju putem operacija dodavanja, uklanjanja ili zamene čvorova

```
def local_search(solution, nodes_list, G, shortest_paths):
    best_solution = solution[:]
    improved = True
    iteration_count = 0

    while improved:
        improved = False


        for operation in ['add', 'remove', 'swap']:
            iteration_count += 1
            print(f"Trying operation: {operation} on {best_solution}")
            if operation == 'add':
                new_solution = local_search_add(best_solution, G, nodes_list, shortest_paths)
            elif operation == 'remove':
                new_solution = local_search_remove(best_solution, G, shortest_paths)
            elif operation == 'swap':
                new_solution = local_search_swap(best_solution, G, nodes_list, shortest_paths)

            if fitness(G, new_solution, shortest_paths) < fitness(G, best_solution, shortest_paths):
                best_solution = new_solution
                improved = True


    return best_solution
```

Slika 13: Kod funkcije *local_ search*

5 Testiranje i rezultati

5.1 Brute force

Bruteforce metoda je veoma korisna za tačna rešenja, posebno na malim i jednostavnim grafovima. Međutim, zbog eksponencijalne složenosti, postaje nepraktična za veće grafove, što ukazuje na potrebu za razvojem efikasnijih algoritama za analizu metricke dimenzije.

Kako se broj čvorova u grafu povećava, vreme potrebno za pronalaženje rešenja se značajno povećava. Bruteforce pristup proverava sve moguće kombinacije čvorova, što može postati neizvodljivo za grafove sa više od 10-12 čvorova, zavisno od strukture grafa.

Bruteforce pristup garantuje pronalaženje optimalnog rešenja, jer proverava sve moguće kombinacije čvorova. To je prednost u odnosu na heurističke ili aproksimativne metode, koje možda ne bi mogle da garantuju optimalnost.

Za jednostavne grafove (npr. linijski ili ciklični) vreme izvršavanja može biti relativno kratko, dok su rezultati često lakši za analizu. Grafovi poput potpunih ili zvezdastih, koji imaju više veza i čvorova, mogu značajno povećati vreme izvršavanja, ali rezultati i dalje ostaju tačni.

5.2 Poređenje

Definisali smo različite vrste grafova koje ćemo koristiti za testiranje:
-linijski graf: jednostavni grafovi sa malim brojem čvorova
-potpuni graf: svi čvorovi su povezani, što dovodi do većih metrickih dimenzija
-ciklični graf: obezbeđuje povezanu strukturu sa ponavljanjem
-zvezdasti graf: jedan centralni čvor povezan sa svim ostalima
-nasumični graf: koristiće se Erdos-Rényi model, koji će obezbediti raznolikost u strukturi

-Brzina i efikasnost
VNS se pokazao bržim za manje grafove (do 20 čvorova) zbog svoje jednostavnosti i efikasnosti u pretrazi lokalnog prostora.
Genetski algoritam je bio konkurentniji na većim grafovima, posebno kada je potrebno istražiti raznolike rešenja, ali je zahtevao više vremena za konvergenciju.
-Kvalitet rešenja
VNS je često pronalazio rešenja sa manjim brojem čvorova u metrickim dimenzijama za jednostavnije strukture, kao što su linijski i ciklični grafovi. *GA* je uspeo da pronađe dobar balans između brzine i kvaliteta rešenja u složenijim strukturama poput potpunih i zvezdastih grafova.
-Stabilnost
VNS je imao manje varijabilnosti u rešenjima kada je testiran više puta na istim grafovima, dok su rezultati *GA* pokazali veću raznolikost, što može biti pozitivno u potrazi za globalno optimalnim rešenjem.
-Ponašanje sa različitim grafovima
VNS se bolje snalazio u grafovima sa jasnim lokalnim minimumima (npr. ciklični grafovi). *GA* je bio efikasniji u kompleksnijim strukturama gde su potrebne raznolike strategije pretrage.
-Praktična upotreba
Za manje i jednostavne grafove, preporučuje se korišćenje *VNS*-a zbog brzine i efikasnosti.
Za veće i kompleksnije grafove, *GA* bi mogao biti bolji izbor zbog svoje sposobnosti da istražuje širi prostor rešenja.

5.3 Veliki grafovi

Kada pričamo o velikim grafovima i optimizaciji parametara, *VNS* može bolje raditi na manje kompleksnim grafovima zbog svoje jednostavne strukture pretrage. *GA* može bolje raditi na grafovima sa više složenosti (npr. Barabási-Albert) zbog svoje sposobnosti da istražuje veći prostor rešenja.

U većim i složenijim grafovima, GA bi mogao biti bolji zbog svoje sposobnosti da pronađe raznovrsna rešenja kroz mutaciju i ukrštanje. U manje složenim ili jednostavnim grafovima, VNS bi mogao pružiti brža rešenja.

6 Zaključak

Ovaj rad istražuje različite pristupe za rešavanje problema pronalaženja minimalne metričke dimenzije. Svaka od ovih metoda ima svoje prednosti i nedostatke, koji se manifestuju u zavisnosti od karakteristika ulaznog grafa i zahteva aplikacije.

Bruteforce metoda se pokazala kao najpouzdaniji pristup za male grafove, jer garantuje pronalaženje optimalnog rešenja. Međutim, njena vremenska složenost brzo postaje neizvodljiva za veće grafove, što je značajan nedostatak.

VNS pruža efikasniji okvir za srednje velike grafove, omogućavajući brže konvergiranje ka dobrom rešenju uz razumnu garantiju kvaliteta. Njena sposobnost da se prilagodi raznim operacijama pretraživanja čini je fleksibilnim alatom za različite vrste grafova.

Genetski algoritam nudi visoku fleksibilnost i može se koristiti za rešavanje kompleksnih problema u velikim grafovima. Iako ne garantuje optimalnost, njegov kapacitet za pronalaženje dobrih rešenja u razumnom vremenu čini ga pogodnim za širok spektar aplikacija. Međutim, zahteva pažljivo podešavanje parametara, što može biti izazovno.

Na kraju, izbor metode zavisi od specifičnih potreba problema. Za male grafove, bruteforce ostaje najbolji izbor. Za srednje velike grafove, VNS pruža optimalnu ravnotežu između brzine i tačnosti. Dok je GA najprikladniji za velike i složene grafove, posebno kada je vreme izvršavanja kritično. Dalja istraživanja mogla bi se usmeriti na optimizaciju ovih algoritama i njihovu kombinaciju kako bi se postigli još bolji rezultati.

Literatura

[1] Nenad Mladenović, Jozef Kratica, Vera Kovačvić -Vujčić, Mirjana Čangalović- Variable neighborhood search for metric dimension and minimal doubly resolving set problems