MINIMUM GRAPH COLORING

Racunarksa inteligencija

Matematicki Fakultet

Stefan Mitrovic 350/2020

Lazar Rankovic 107/2019

Opis problema

Uvod

Minimum graf bojenje je problem u teoriji grafova koji se odnosi na bojenje čvorova grafa tako da dva susedna čvora nemaju istu boju, koristeći pri tome najmanji mogući broj boja. Ovaj problem je ključan za razumevanje mnogih kompleksnih struktura i ima široku primenu u različitim oblastima kao što su računarstvo, telekomunikacije, logistika i mnoge druge.

Formalna definicija

Dati graf $G=(V,E)$, gde je V skup čvorova, a E skup grana, cilj je odrediti funkciju bojenja $c: V \rightarrow \{1,2,...,k\}$ tako da $c(u) \neq c(v) c(u)$ za svaku granu $(u,v) \in E$, pri čemu je k minimalan broj boja. Broj k se naziva hromatski broj grafa G, označen kao $\chi(G)$.

Praktične primene

Problem minimum graf bojenje ima brojne praktične primene, uključujući:

- **Raspoređivanje zadataka**: Efikasno alociranje resursa tako da nema konflikta u vremenskim terminima ili resursima.
- **Dodela frekvencija**: Dodela frekvencija u telekomunikacionim mrežama tako da ne dolazi do interferencije.

- **Mapiranje registara**: Optimizacija upotrebe registara u kompajlerima za smanjenje broja potrebnih registara.

Algoritmi:

Primena grube sile (Brute Force)

Metoda grube sile (brute force) je jednostavan, ali neefikasan pristup za rešavanje ovog problema, koji podrazumeva ispitivanje svih mogućih kombinacija bojenja kako bi se našlo optimalno rešenje.

Metoda grube sile funkcioniše tako što generiše sve moguće načine bojenja čvorova grafa i proverava koji od tih načina zadovoljava uslove problema, pri čemu koristi minimalan broj boja. Postupak je sledeći:

1. **Generisanje svih mogućih bojenja**: Za dati graf sa n čvorova i k boja, potrebno je generisati k^n mogućih kombinacija bojenja. Svaka kombinacija predstavlja jedno potencijalno rešenje.
2. **Provera validnosti bojenja**: Za svaku kombinaciju, proverava se da li je bojenje validno, odnosno da li dva susedna čvora nemaju istu boju.

3. **Odabir optimalnog rešenja**: Od svih validnih kombinacija bojenja, bira se ona koja koristi najmanji broj boja.

```python
def brute_force_node_coloring(graph, start_time):
    n = len(graph.nodes())
    iters = 0

    for coloring in product(range(1, n + 1), repeat=n):
        iters += 1
        if time.time() - start_time >= 300:
            return -1, -1
        if is_valid_node_coloring(graph, coloring):
            return {node: coloring[i] for i, node in enumerate(graph.nodes())}, iters

    return None, iters
```

Kod funkcije brute_force_node_coloring

Funkcija brute_force_node_coloring pokušava da oboji čvorove grafa metodom grube sile, isprobavajući sve moguće kombinacije bojenja sa do n boja (gde je n broj čvorova). Za svaku kombinaciju, proverava validnost bojenja tako da nijedna dva susedna čvora nemaju istu boju. Ako pronađe validno bojenje, vraća rečnik sa bojama po čvorovima i broj iteracija. Ukoliko funkcija traje duže od 300 sekundi, vraća -1, -1. Ako ne pronađe validno bojenje nakon ispitivanja svih kombinacija, vraća None i broj iteracija.

```python
def initialize_pheromones(g):
    for u, v in g.edges():
        g[u][v]['pheromones'] = 1.0

    for node in g.nodes():
        g.nodes[node]['pheromones'] = 0.1  # Setting a small initial value for nodes

def construct_solution(g):
    coloring = {}
    uncolored_nodes = list(g.nodes())

    for node in uncolored_nodes:
        used_colors = set(coloring.get(neighbor, None) for neighbor in g.neighbors(node))
        available_colors = [c for c in range(len(uncolored_nodes)) if c not in used_colors]
        chosen_color = min(available_colors)
        coloring[node] = chosen_color

    return coloring

def evaluate_coloring(g, coloring):
    conflicts = 0
    for u, v in g.edges():
        if coloring[u] == coloring[v]:
            conflicts += 1
    return conflicts / 2  # Each conflict is counted twice
```

```python
def map_colors(coloring):
    color_map = {}
    next_color = 0
    for node in coloring:
        color = coloring[node]
        if color not in color_map:
            color_map[color] = next_color
            next_color += 1
    return [color_map[color] for color in coloring.values()]

def aco_graph_coloring(g, num_ants, rho, num_iterations):
    initialize_pheromones(g)
    best_coloring = None
    smallest_conflicts = float('inf')
    conflicts_history = []

    for _ in range(num_iterations):
        for _ in range(num_ants):
            coloring = construct_solution(g)
            conflicts = evaluate_coloring(g, coloring)
            if conflicts < smallest_conflicts:
                best_coloring = deepcopy(coloring)
                smallest_conflicts = conflicts
        conflicts_history.append(smallest_conflicts)

        # Evaporation
        for u, v in g.edges():
            g[u][v]['pheromones'] *= rho

        # Update pheromones
        for u, v in g.edges():
            if best_coloring[u] != best_coloring[v]:
                g[u][v]['pheromones'] += 1.0 / (smallest_conflicts + 1)

    return best_coloring, smallest_conflicts
```

Kod funkcije aco_graph_coloring sa svojim metodama.

ACO (Ant Colony Optimization) je metaheuristički algoritam inspirisan ponašanjem mrava u potrazi za hranom. Osnovna ideja je simulacija kolektivnog ponašanja mrava koji koriste tragove feromona da pronađu najkraći put do hrane. Algoritam radi tako što mravi postepeno grade rešenje, ostavljajući tragove feromona na putanji kojom su prošli, što privlači druge mrave da preferiraju istu putanju. Isparavanje feromona i ažuriranje tragova pomaže algoritmu da istražuje različite mogućnosti i izbegava lokalne optimume, težeći ka

globalnom optimumu. ACO se primenjuje na širok spektar problema optimizacije, uključujući rutiranje, raspoređivanje resursa, bojenje grafova i druge kombinatorne probleme gde je potrebno pronaći najbolje moguće rešenje u velikom prostoru mogućnosti.

- **Inicijalizacija feromona:**
- Feromoni na granama grafa postavljeni su na početnu vrednost od 1.0, što znači da su sve grane u početku jednako privlačne za mrave.
- Feromoni na čvorovima postavljeni su na malu početnu vrednost od 0.1, što pomaže u inicijalnoj konstrukciji rešenja.
- **Konstrukcija rešenja:**
- Svaki mrav konstruira rešenje (bojenje grafa) na sledeći način:
    - Čvorovi se boje jedan po jedan.
    - Za svaki čvor, mrav bira najmanju dostupnu boju koja se ne koristi kod njegovih suseda.
    - Ovaj proces se ponavlja za sve čvorove dok svi ne budu obojeni.
- **Evaluacija bojenja:**
- Broj konflikata se računa za svako bojenje. Konflikti nastaju kada dva susedna čvora imaju istu boju.
- Formula za broj konflikata je jednostavna: svaki konflikt se računa kao 1, ali se deli sa 2 da bi se izbeglo dupliranje (jer se svaki konflikt računa dva puta, jednom za svaki čvor u konfliktu).
- **Ažuriranje najboljeg rešenja:**
- Ako novo rešenje ima manje konflikata od trenutnog najboljeg rešenja, algoritam ažurira najbolje rešenje i broj konflikata.
- Ovim se osigurava da se tokom vremena najbolje rešenje konstantno poboljšava.
- **Isparavanje feromona:**
- Feromoni na svim granama se smanjuju množenjem sa faktorom isparavanja ρ. Ovim se modelira prirodno isparavanje feromona i sprečava da se mravi previše oslanjaju na stare tragove.
- Ovo pomaže algoritmu da istraži nove mogućnosti i izbegne lokalne minimume.
- **Ažuriranje feromona:**
- Feromoni na granama između čvorova koji imaju različite boje u najboljem rešenju se povećavaju.
- Time se ojačavaju tragovi feromona na granama koje vode ka dobrim rešenjima, povećavajući verovatnoću da će budući mravi koristiti te grane.

Ulazni parametri:

- num_ants: Broj mrava koji se koriste za kreiranje rešenja u svakoj iteraciji. Više mrava može značiti bolju pokrivenost mogućih rešenja, ali i duže vreme izvršavanja.
- rho: Faktor isparavanja feromona. Manja vrednost znači brže isparavanje i agresivnije istraživanje novih rešenja, dok veća vrednost znači sporije isparavanje i više eksploatacije trenutnih tragova.
- num_iterations: Broj iteracija algoritma. Više iteracija omogućava algoritmu da bolje istraži prostor rešenja i potencijalno pronađe bolje bojenje.

PSO

PSO (Particle Swarm Optimization) je metaheuristički algoritam inspirisan socijalnim ponašanjem roja ptica ili riba u potrazi za hranom. Osnovna ideja PSO-a je simulacija kretanja čestica u prostoru rešenja, pri čemu svaka čestica predstavlja moguće rešenje problema optimizacije. Svaka čestica se kreće kroz prostor pretrage u skladu sa svojom trenutnom pozicijom i brzinom, težeći da nađe najbolju poziciju (rešenje) koja minimizuje ili maksimizuje funkciju cilja. U procesu optimizacije, čestice se međusobno "informišu" o svojim najboljim pozicijama i globalno najboljoj poziciji koju je pronašla bilo koja čestica u roju. Ovaj algoritam se često koristi za rešavanje različitih problema optimizacije, kao što su optimizacija funkcija, problemi u mašinskom učenju, raspoređivanje resursa i slično.

```python
class ParticleColoring:
    def __init__(self, graph):
        self.graph = graph
        self.num_nodes = len(graph)
        self.colors = {node: None for node in graph}  # Početno, nijedan čvor nije obojen
        self.personal_best_colors = None
        self.personal_best_fitness = float('inf')  # Početna vrednost za najbolju ocenu

    def calculate_fitness(self):
        unique_colors = len(set(self.colors.values()))
        return unique_colors

    def update_position(self):
        for node in self.graph:
            neighbor_colors = {self.colors[neighbor] for neighbor in self.graph[node] if self.colors[neighbor] is not None}
            if self.colors[node] is None:
                available_colors = set(range(len(neighbor_colors) + 1)) - neighbor_colors
                self.colors[node] = min(available_colors)
                print("Čvor", node, "dodeljena boja:", self.colors[node])



    @classmethod
    def update_global_best(cls, personal_best_fitness, personal_best_colors):
        if personal_best_fitness < cls.swarm_best_fitness:
            cls.swarm_best_fitness = personal_best_fitness
            cls.swarm_best_colors = personal_best_colors.copy()
```

```python
def pso_coloring(graph, swarm_size, num_iters):
    ParticleColoring.swarm_best_colors = None
    ParticleColoring.swarm_best_fitness = float('inf')

    swarm = [ParticleColoring(graph) for _ in range(swarm_size)]

    for i in range(num_iters):
        print("Iteracija", i + 1)
        for particle in swarm:
            particle.update_position()
            fitness = particle.calculate_fitness()
            if fitness < particle.personal_best_fitness:
                particle.personal_best_fitness = fitness
                particle.personal_best_colors = particle.colors.copy()
                ParticleColoring.update_global_best(fitness, particle.personal_best_colors)

        print("Best colors:", ParticleColoring.swarm_best_colors)
        print("Best fitness:", ParticleColoring.swarm_best_fitness)

    return ParticleColoring.swarm_best_colors, ParticleColoring.swarm_best_fitness
```

Primer klase ParticleColoring sa svojim metodama.

1. **ParticleColoring klasa**
    - Ova klasa predstavlja česticu u PSO algoritmu. Svaka čestica odgovara jednom čvoru grafa.
    - **__init__(self, graph)**: Konstruktor inicijalizuje česticu sa zadatim grafom. Svaki čvor počinje sa None vrednošću boje.

- **calculate_fitness(self)**: Metoda računa fitness vrednost čestice. U kontekstu bojenja grafova, fitness se definiše kao broj jedinstvenih boja koje su dodeljene čvorovima.
- **update_position(self)**: Metoda ažurira poziciju (boje čvorova) čestice. Za svaki čvor se određuje najmanja raspoloživa boja koja ne dovodi do konflikta sa susednim čvorovima.
- **update_global_best(cls, personal_best_fitness, personal_best_colors)**: Klasna metoda za ažuriranje globalno najboljeg rešenja. Ažurira se samo ako je trenutna čestica pronašla bolje rešenje od trenutnog globalnog najboljeg.

2. **pso_coloring(graph, swarm_size, num_iters) funkcija**
   - Ova funkcija implementira glavnu logiku PSO algoritma za bojenje grafova.
   - **swarm_size**: Broj čestica (čvorova) u roju.
   - **num_iters**: Broj iteracija algoritma.
   - **Inicijalizacija roja čestica**: Kreira se swarm sa swarm_size čestica, gde svaka čestica odgovara jednom čvoru grafa.
   - **Glavna petlja iteracija**: Algoritam prolazi kroz num_iters iteracija.
     - Svaka čestica ažurira svoje pozicije (boje čvorova) pozivom update_position().
     - Računa se fitness vrednost za svaku česticu pozivom calculate_fitness().
     - Ako čestica ima bolji fitness od svog dosadašnjeg najboljeg rešenja (personal_best_fitness), ažurira se njeno najbolje rešenje.
     - Ažurira se globalno najbolje rešenje (swarm_best_fitness i swarm_best_colors) ako je pronađeno bolje rešenje od trenutno najboljeg.
   - **Ispis rezultata**: Na kraju svake iteracije se ispisuju trenutno najbolje boje (swarm_best_colors) i pripadajući fitness (swarm_best_fitness).

Detaljno objašnjenje algoritma:

- **Inicijalizacija čestica**: Svaka čestica u početku nema dodeljenu boju, predstavljenu kao None.
- **Ažuriranje pozicija (boja)**: Za svaki čvor se određuje boja tako da se minimizira broj konflikata sa susednim čvorovima.
  - Ako je boja čvora None, čestica bira najmanju raspoloživu boju koja ne dovodi do konflikta.
- **Fitness funkcija**: Računa se kao broj jedinstvenih boja koje su dodeljene čvorovima grafa. Cilj je minimizirati ovu vrednost, što znači koristiti što manje boja.

- **Ažuriranje najboljih rešenja**: Svakom iteracijom se ažurira najbolje poznato rešenje (swarm_best_fitness i swarm_best_colors) ako se pronađe bolje rešenje od trenutnog.
- **Kraj algoritma**: Algoritam se zaustavlja nakon određenog broja iteracija (num_iters), a rezultati se ispisuju i mogu se koristiti dalje za analizu.

Simulirano kaljenje

Opšteno simulirano kaljenje je metaheuristički optimizacioni algoritam inspirisan procesom kaljenja metala, gde se materijal postepeno zagreva, zatim hladi kako bi se postigla optimalna struktura. U kontekstu optimizacije, algoritam počinje sa slučajnim rešenjem i iterativno ga poboljšava, prihvatajući lošija rešenja sa određenom verovatnoćom kako bi izbegao zaglavljenje u lokalnim optimumima. Proces se sastoji od postepenog smanjenja temperature (verovatnoće prihvatanja lošijih rešenja) tokom iteracija, što omogućava algoritmu da pretražuje širi prostor rešenja u ranim fazama i fokusira se na lokalno poboljšanje u kasnijim fazama. Na kraju, algoritam konvergira ka optimalnom ili suboptimalnom rešenju, zavisno od parametara kao što su temperatura i strategije prihvatanja novih rešenja. Simulirano kaljenje se često koristi za rešavanje raznovrsnih problema optimizacije gde je teško pronaći globalni optimum metodama grube sile ili lokalne pretrage.

```python
def simulated_annealing(graph, num_iters, cooling_rate):
    solution = initialize(graph)
    value = calc_solution_value(solution, graph)
    best_solution = copy.deepcopy(solution)
    best_value = value

    values = [None for _ in range(num_iters)]
    for i in range(1, num_iters + 1):
        new_solution = make_small_change(solution, graph)
        new_value = calc_solution_value(new_solution, graph)
        delta_E = new_value - value

        if delta_E < 0 or random.random() < math.exp(-delta_E / cooling_rate):
            value = new_value
            solution = copy.deepcopy(new_solution)
            if new_value <= best_value:
                best_value = new_value
                best_solution = copy.deepcopy(new_solution)

        values[i - 1] = value

        num_colors_used = len(set(best_solution.values()))
        print(f"Iteration: {i}, Num colors used: {num_colors_used}, Conflict: {value}")


    print("Best Solution:")
    for node, color in best_solution.items():
        print(f"Node {node}: Color {color}")
    print("Best Conflict:", best_value)

    num_colors_used = len(set(best_solution.values()))
    print("Number of colors used:", num_colors_used)

    G = nx.Graph()
    for node in graph:
        for neighbor in graph[node]:
            G.add_edge(node, neighbor)

    pos = nx.spring_layout(G)
    node_colors = [best_solution[node] for node in graph]

    plt.figure(figsize=(8, 8))
    nx.draw(G, pos, with_labels=True, node_color=node_colors, cmap=plt.cm.rainbow, node_size=500)
    plt.title('Graph Coloring')
    plt.show()

    return best_solution, best_value
```

Prikaz koda funkcije simulirano kaljenje

Ova funkcija simulated_annealing implementira simulirano kaljenje za rešavanje problema bojenja grafova. Evo detaljnog objašnjenja koraka i strukture funkcije:

1. Inicijalizacija početnog rešenja:

- solution = initialize(graph): Inicijalizuje se početno rešenje. Ova funkcija initialize(graph) verovatno dodeljuje slučajne boje čvorovima grafa graph.

- value = calc_solution_value(solution, graph): Računa se vrednost (konflikt) početnog rešenja pozivom funkcije calc_solution_value(solution, graph) koja verovatno računa broj konflikata (susedni čvorovi obojeni istom bojom)

2. Pamćenje najboljeg rešenja:

- best_solution = copy.deepcopy(solution): Inicijalizuje se najbolje poznato rešenje kao početno rešenje.
- best_value = value: Inicijalizuje se vrednost najboljeg rešenja kao početna vrednost početnog rešenja.

3. Glavna petlja simuliranog kaljenja:

- Iterira se num_iters puta, što predstavlja broj iteracija algoritma.
- Za svaku iteraciju:
  - new_solution = make_small_change(solution, graph): Generiše se novo rešenje new_solution tako što se napravi mala promena u trenutnom rešenju solution.
  - new_value = calc_solution_value(new_solution, graph): Računa se vrednost novog rešenja.
  - delta_E = new_value - value: Računa se razlika u vrednosti između trenutnog i novog rešenja.

4. Prihvatanje ili odbacivanje novog rešenja:

- Ako je delta_E < 0 ili ako je slučajna vrednost manja od exp(-delta_E / cooling_rate), novo rešenje se prihvata kao trenutno rešenje.
- Ako je novo rešenje bolje (manje konflikata), ažurira se best_solution i best_value.

5. Praćenje vrednosti tokom iteracija:

- values[i - 1] = value: Čuva se vrednost trenutnog rešenja tokom svake iteracije.
- Ispisuju se informacije o svakoj iteraciji, uključujući iteraciju, broj korišćenih boja i trenutni konflikt.

6. Kraj algoritma:

- Na kraju se ispisuje najbolje pronađeno rešenje (best_solution) i pripadajući konflikt (best_value).

Genetski Algoritam

Genetski algoritam je metoda optimizacije i pretrage inspirisana prirodnom selekcijom, deo evolucione biologije. Koristi populaciju potencijalnih rešenja koja se razvijaju kroz generacije. Svaka generacija prolazi kroz proces selekcije, ukrštanja (crossover) i mutacije kako bi se stvorila nova populacija. Cilj je maksimizirati ili minimizirati funkciju fitnesa, koja meri kvalitet svakog rešenja.

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

    for _ in range(num_generations):
        population.sort(key=lambda x: x.fitness)
        # print(population[0].fitness)
        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)
            if random.random() < mutation_prob:
                mutation(child)
            offspring.append(child)

        population = elites + offspring

    best_solution = min(population, key=lambda x: x.fitness)
    if is_feasible(best_solution):
        return best_solution, True
    else:
        return best_solution, False
```

Glavna funkcija ga implementira genetski algoritam. Kreira početnu populaciju, sortira je po fitnes vrednosti, selektuje najbolje jedinke (elitizam), ukršta i mutira ostatak populacije, te na kraju vraća najbolje rešenje.

```
def calc_fitness(self):
    conflicts = 0
    num_colors = max(self.colors.values())

    for node in self.graph:
        for neighbor in self.graph[node]:
            if self.colors[node] == self.colors[neighbor]:
                conflicts += 1

    fitness = conflicts + (num_colors / self.max_degree)
    return fitness
```

Funkcija za računanje fitnesa (calc_fitness) određuje kvalitet rešenja tako što računa broj konflikata, gde su konflikti definisani kao parovi susednih čvorova koji imaju istu boju. Takođe, uzima u obzir i broj korišćenih boja. Fitnes vrednost je kombinacija broja konflikata i broja korišćenih boja normalizovanih maksimalnim stepenom grafa. Manja fitnes vrednost znači bolje rešenje.

```
def crossover(parent1, parent2):
    child_colors = {}
    for node in parent1.colors:
        child_colors[node] = parent1.colors[node] if random.random() < 0.5 else parent2.colors[node]
    child = Individual(parent1.graph)
    child.colors = child_colors
    child.fitness = child.calc_fitness()
    return child
```

Funkcija za ukrštanje (crossover) kreira novu jedinku tj individual (dete) kombinovanjem boja dva roditelja. Za svaki čvor u grafu, boja deteta se nasumično bira iz boje jednog od roditelja. Ovo omogućava kombinovanje karakteristika oba roditelja u novom rešenju, što može dovesti do boljih rešenja.

```
def mutation(individual):
    node = random.choice(list(individual.colors.keys()))
    neighbor_colors = {individual.colors[neighbor] for neighbor in individual.graph[node]}
    available_colors = set(range(1, len(individual.graph) + 1)) - neighbor_colors
    if available_colors:
        individual.colors[node] = random.choice(list(available_colors))
        individual.fitness = individual.calc_fitness()
```

Funkcija za mutaciju (mutation) menja boju jednog nasumično odabranog čvora. Nova boja se bira tako da se izbegne boja susednih čvorova, ako je to moguće. Ova operacija dodaje varijabilnost populaciji i pomaže u izbegavanju lokalnih minimuma, poboljšavajući tako sposobnost algoritma da pronađe globalno optimalno rešenje.

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

Funkcija za turnirsku selekciju (tournament_selection) bira roditelje za ukrštanje tako što nasumično odabira podskup populacije i bira individualca sa najboljom fitnes vrednošću iz tog podskupa. Turnirska selekcija osigurava da se najbolji individualci češće biraju za reprodukciju, dok još uvek daje šansu manje fit individualcima da doprinesu genetskoj varijabilnosti.

Poredjenje:

Kada analiziramo broj boja u rešenju i vreme izvršavanja algoritama za minimalno bojenje grafova, ulazimo u suptilni domen gde se susrećemo sa mnogim nijansama u efikasnosti i tačnosti ovih algoritama.

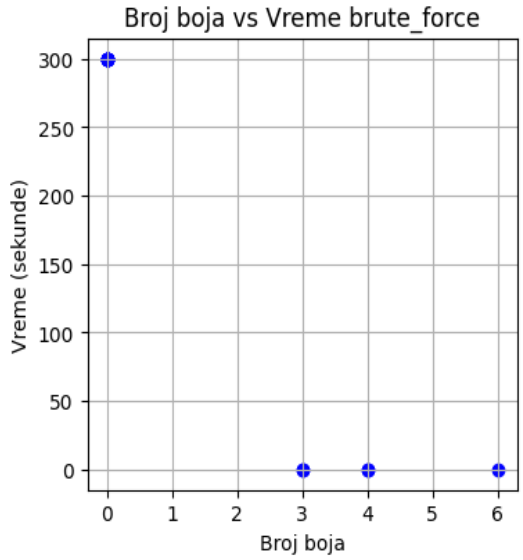
Algoritam grube sile, uprkos svojoj tačnosti, pokazuje značajan rast vremena izvršavanja sa povećanjem broja čvorova u grafu. Iako je ograničen na 300 sekundi, nakon što pređe 6 čvorova, ne uspeva da pronađe rešenje u zadatom vremenskom intervalu, što ga čini nepraktičnim za veće grafove i ne upotrebljivim iako bi posle nekog odredjenog vremena dosao do resenja.

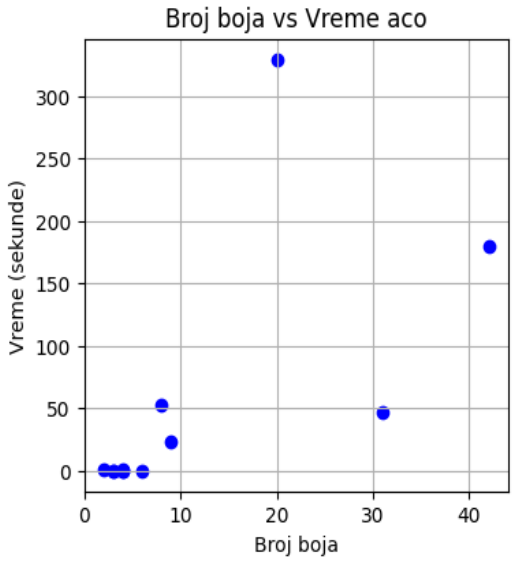
ACO i PSO algoritmi su vrlo slični i efikasni u pronalaženju rešenja, ali zahtevaju nešto više vremena. Za male grafove su izuzetno efikasni kako u pogledu vremena tako i tačnosti. Međutim, kako raste broj čvorova, raste i vremensko ograničenje. Najveći

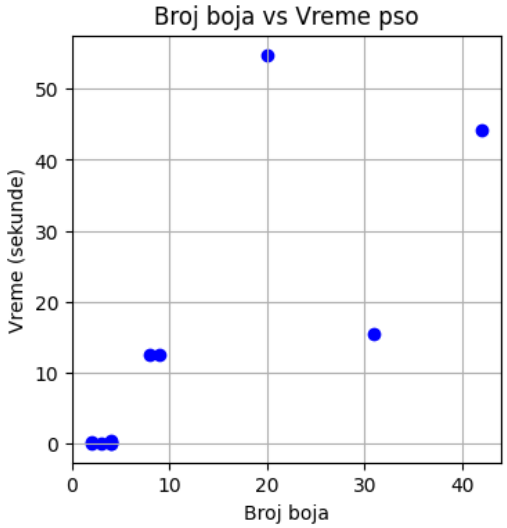
izazov predstavlja povezanost čvorova, gde graf sa visokom povezanošću čvorova radi znatno sporije.

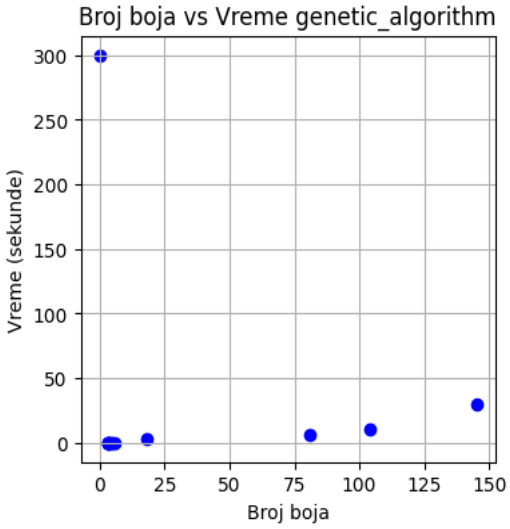
Genetski algoritam može postati vremenski efikasan ako povećavamo njegove parametre, ali što se tiče pronalaženja rešenja, nije efikasan za velike grafove jer često vraća loša rešenja.

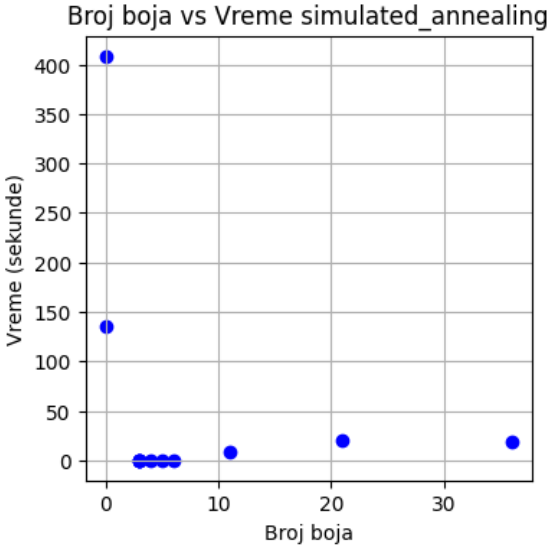
Simulirano kaljenje se pokazalo odličnim za male grafove sa brzim vremenom pronalaska rešenja. Međutim, kako se povećava broj čvorova, njegova efikasnost opada, često dovodeći do sporijeg pronalaženja rešenja ili čak do situacija gde ne uspeva da pronađe rešenje uopšte.


Broj boja vs Vreme brute_force
Vreme (sekunde)
Broj boja


Broj boja vs Vreme aco
Vreme (sekunde)
Broj boja


Broj boja vs Vreme pso
Vreme (sekunde)
Broj boja

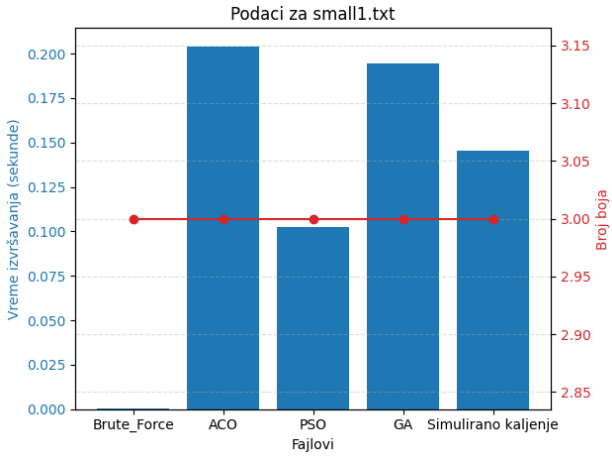

Broj boja vs Vreme genetic_algorithm
Vreme (sekunde)
Broj boja

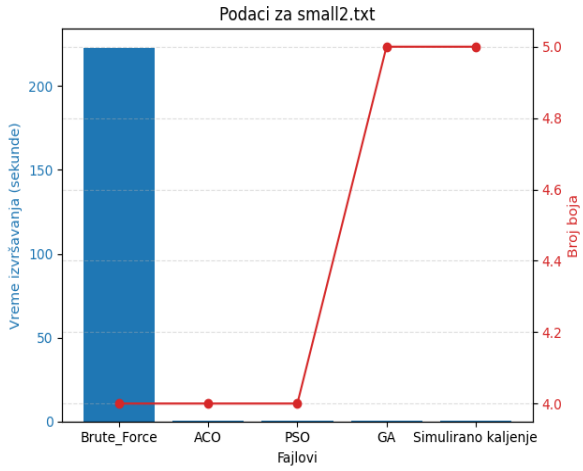

Broj boja vs Vreme simulated_annealing
Vreme (sekunde)
Broj boja

Odnos broj boje i vremena:

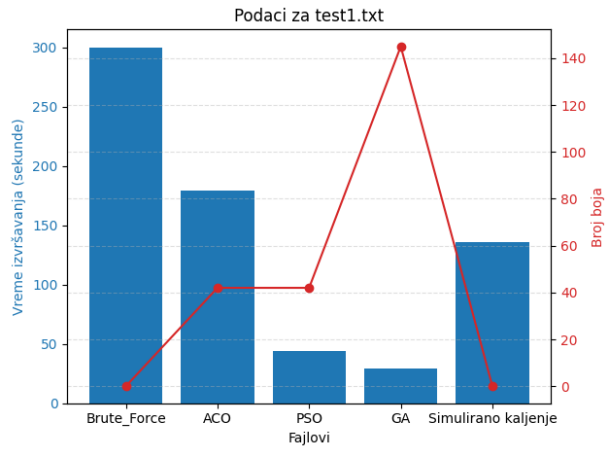
Kada se bavimo poredjenjem svakog algoritma sa svakim grafom u kontekstu minimalnog bojenja grafova, ulazimo u kompleksan domen gde se različiti algoritmi mogu ponašati različito u zavisnosti od specifičnosti grafa. Svaki algoritam ima svoje prednosti i ograničenja u efikasnosti i preciznosti rešenja, što čini važno istraživanje kako bi se razumela njihova performansa u različitim scenarijima.

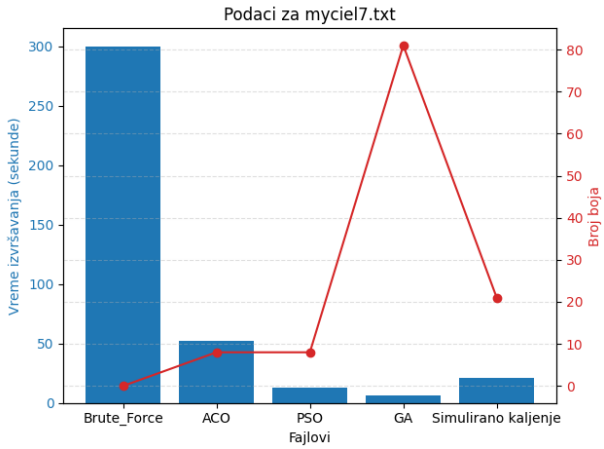
Ovde vidimo (small1.txt) da brute_force algoritam je najbolji sto se tice grafova do 3 posle toga situacija se drasticno menja na gore. Ostali su dosta slicni i nijanse odlucuju.





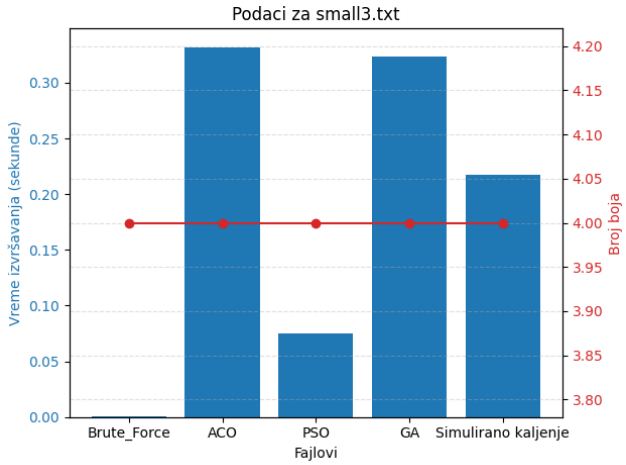
Za graf small2.txt vidimo vec taj veliki rast za brute_force sto se vremena tice dok su ostali algorimi isto sto se vremena tice ali GA i simulirano kaljenje ne pronalazi optimalno resenje.





Za nesto ozboljnije i vece grafove kao sto su ova dva ali i ostali vidimo da Brute_force prelazi ogranicenje od 300sec i ne dolazi do resenja kao i simulirano kaljenje ali to se

desava dosta ranije. Genetski algoritam dolazi do veoma loseg resenja ali na efikasan nacin. ACO i PSO jedini dolaze do tacnog resenja gde je PSO malo brzi.



Jos jedna primer manjeg grafa gde vidimo da je brute_force najbolji i da je tu negde i PSO a zatim svi ostali.

Primena:

Minimalno bojenje grafova ima široku primenu u raznim oblastima. Jedna od najpoznatijih primena je u raspoređivanju zadataka, gde se koristi za dodelu minimalnog broja resursa poput vremena ili prostorija tako da se izbegnu konflikti. U telekomunikacijama, minimalno bojenje grafova može optimizovati raspodelu frekvencija kako bi se smanjile interferencije između različitih predajnika.

U kompjuterskoj nauci, koristi se za alokaciju registara u kompajlerima, čime se efikasnije koristi memorija prilikom izvršavanja programa. Takođe, u mrežnim protokolima, minimalno bojenje pomaže u planiranju bežičnih mreža kako bi se smanjile smetnje među čvorovima mreže.

U društvenim mrežama, može se koristiti za grupisanje korisnika na osnovu interesa tako da su povezani korisnici različito obojeni, olakšavajući analizu strukture mreže. U biologiji, minimalno bojenje grafova može pomoći u analizi genetskih mreža, identifikaciji funkcionalnih modula i predikciji protein-protein interakcija.

Takođe, u transportnoj logistici, minimalno bojenje grafova može se koristiti za optimizaciju ruta i rasporeda, osiguravajući da se resursi poput vozila i putničkih linija

koriste što efikasnije. U industriji zabave, može se primeniti na raspoređivanje filmskih projekcija u bioskopima, kako bi se izbeglo preklapanje filmova koji ciljaju sličnu publiku.

Sve ove primene pokazuju kako minimalno bojenje grafova može biti moćan alat za optimizaciju i rešavanje problema u različitim domenima.

Zakljucak:

Kada se bavimo poređenjem svakog algoritma sa svakim grafom u kontekstu minimalnog bojenja grafova, ulazimo u kompleksan domen gde se različiti algoritmi mogu ponašati različito u zavisnosti od specifičnosti grafa. Svaki algoritam ima svoje prednosti i ograničenja u efikasnosti i preciznosti rešenja, što čini važno istraživanje kako bi se razumela njihova performansa u različitim scenarijima.

Algoritam grube sile:

Algoritam grube sile, iako precizan, pokazuje eksponencijalni rast vremena izvršavanja kako se povećava broj čvorova grafa. Ograničen je na oko 300 sekundi i nakon što pređe 6 čvorova, ne uspeva da pronađe rešenje u zadatom vremenskom intervalu, što ga čini nepraktičnim za veće grafove.

ACO i PSO algoritmi:

ACO i PSO algoritmi su vrlo slični i efikasni u pronalaženju rešenja, ali zahtevaju nešto više vremena. Za male grafove su izuzetno efikasni kako u pogledu vremena tako i tačnosti. Međutim, kako raste broj čvorova, raste i vremensko ograničenje. Najveći izazov predstavlja povezanost čvorova, gde graf sa visokom povezanošću čvorova radi znatno sporije.

Genetski algoritam:

Genetski algoritam može postati vremenski efikasan ako povećavamo njegove parametre, ali što se tiče pronalaženja rešenja, nije efikasan za velike grafove jer često vraća loša rešenja.

Algoritam simuliranog kaljenja:

Simulirano kaljenje se pokazalo odličnim za male grafove sa brzim vremenom pronalaženja rešenja. Međutim, kako se povećava broj čvorova, njegova efikasnost opada, često dovodeći do sporijeg pronalaženja rešenja ili čak do situacija gde ne uspeva da pronađe rešenje uopšte.

Sve u svemu, izbor algoritma za minimalno bojenje grafova zavisi od specifičnih zahteva zadatka, kao što su veličina i povezanost grafa, kao i od prihvatljivih kompromisa između vremena izvršavanja i tačnosti rešenja.

Literatura:

1) MINIMUM GRAPH COLORING https://www.csc.kth.se/~viggo/wwwcompendium/node15.html
2) https://github.com/MATF-RI/Materijali-sa-vezbi
3) https://mat.tepper.cmu.edu/COLOR/instances.html#XXREGhttps://mat.tepper.cmu.edu/COLOR/instances.html#XXREG