Minimum Edge Coloring

Računarska inteligencija
Matematički fakultet

Petra Ignjatović i Anđela Jovanović
mi20063@alas.matf.bg.ac.rs
mi20205@alas.matf.bg.ac.rs

Sažetak

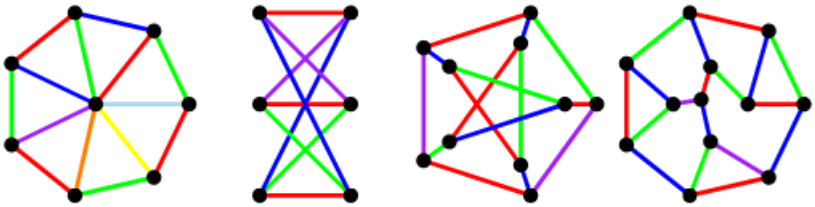
U ovom dokumentu ćemo prikazati kako smo uspešno rešili izazov minimalnog bojenja grafa, kao i koje smo strategije primenili kako bismo optimizovali proces pronalaženja rešenja. Projekat možete pogledati ovde.

Sadržaj

1 Definicija problema

U teoriji grafova, pravilno bojenje grafa označava graf obojen na takav način da nikoje dve susedne grane nemaju dodeljenu istu boju. Dve grane grafa su susedne ukoliko imaju zajednički čvor koji ih spaja. Ukoliko je za neki graf upotrebljeno k boja, onda se za taj graf kaže da je *k-obojen*.

Dodatno, minimalno bojenje grafa označava najmanji broj boja kojim se može ispravno obojiti zadati graf, to jest, traži se minimalno k.



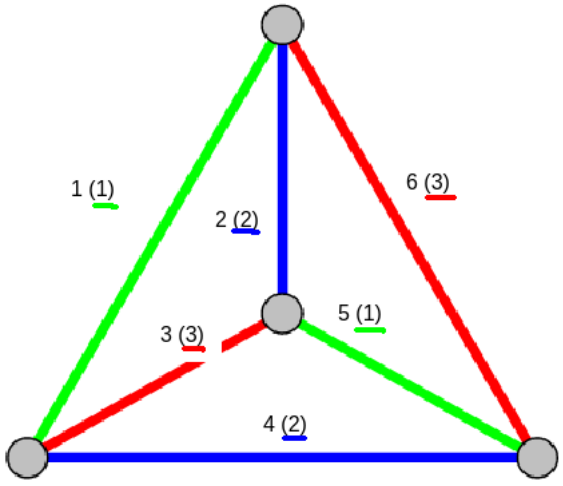
Slika 1: Primeri ispravno obojenih grafova

Najmanji broj kojim se može obojiti graf naziva se hromatski indeks $\chi'(G)$.

2 Reprezentacija grafova u kodu i provera ispravnosti

Radi lakšeg rukovanja grafovima i dodeljenim bojama, svako bojenje grafa je predstavljeno putem permutacija – svaka grana grafa je implicitno numerisana (brojevima od 0 do n-1, gde n označava ukupan broj grana), i svakoj grani je dodeljen prirodan broj koji označava boju. Svaki prirodan broj predstavlja tačno jednu boju.

Na ovaj način, svako bojenje grafa može biti ispisano kao permutacija brojeva od 0 do n-1 (ili od 1 do n)



Grane: 123456

Bojenje: 123213

=

z – p - c – p – z - c

Slika 2: Prikaz obojenih grana putem brojeva

Funkcija *is_valid_coloring* proverava ispravnost bojenja: za svaku granu proverava da li je obojena drugom bojom u odnosu na susedne grane.

```
def is_valid_coloring(graph, coloring):
    edge_color = {edge: coloring[i] for i, edge in enumerate(graph.edges())}

    for edge in graph.edges():
        u, v = edge

        u_edges = neighbor_edges_of_node(graph, u)
        v_edges = neighbor_edges_of_node(graph, v)

        u_edges.remove(edge)
        v_edges.remove(edge)

        for i in range(len(u_edges)):
            if edge_color[u_edges[i]] == edge_color[edge]:
                return False

        for j in range(len(v_edges)):
            if edge_color[v_edges[j]] == edge_color[edge]:
                return False
    return True
```

Slika 3: Kod funkcije *is_valid_coloring*

Prvo se kreira rečnik *edge_color* koji mapira svaku granu grafa na njenu boju iz datog bojenja. Ovo se radi kako bi se lakše pristupalo bojama grana tokom provere. Zatim se prolazi kroz svaku granu grafa.

- Za svaku granu se uzimaju susedne grane čvorova koje ta grana povezuje.
- Iz susednih grana se uklanja trenutna grana kako ne bi bila uzeta u obzir prilikom provere.
- Za svaku susednu granu se proverava da li ima istu boju kao i trenutna grana. Ako se pronađe susedna grana sa istom bojom, to znači da bojenje nije ispravno i funkcija vraća False.

Ovaj proces se ponavlja za sve grane grafa. Ako se ni za jednu granu ne pronađe susedna grana sa istom bojom, funkcija vraća True, što znači da je bojenje ispravno.

3 Primena grube sile

Brute force pristup podrazumeva iscrpnu pretragu svih mogućih bojenja grana grafa kako bi se pronašlo ono koje zahteva najmanji broj boja. Ovaj pristup je teoretski moguć za manje grafove, ali postaje neefikasan za veće grafove zbog eksponencijalnog rasta broja mogućih bojenja sa brojem grana. Za graf sa n granova, broj mogućih bojenja je k^n, gde je k broj boja. Ovo može dovesti do veoma dugog vremena izvršavanja za veće grafove i/ili veliki broj boja.

Koristeći reprezentaciju bojenja putem permutacija, metoda grube sile samo proverava svaku moguću permutaciju brojeva od 1 do n.

```
def brute_force_edge_coloring(graph, start_time):
    n = len(graph.edges())
    iters = 0

    for coloring in product(range(1, n + 1), repeat=n):
        iters += 1
        if time.time() - start_time >= 390:
            return -1, -1
        if is_valid_coloring(graph, coloring):
            return coloring, iters

    return None
```

Slika 4: Kod funkcije brute_force_edge_coloring

funkcija product iz modula itertools generiše sve moguće permutacije zadate dužine, pri čemu se svaki element permutacije može kretati od 1 do n (uključujući n) u ovom slučaju

Da bi se sprečilo predugo izvršavanje programa, uvedeno je vremensko ograničenje koje, u slučaju prekoračenja, vraća rezultat (-1, -1) što se može smatrati neuspehom.

Ovu kompleksnost možemo pokušati da rešimo pomoću metaheuristika i drugih metoda optimizacija.

4 Optimizacije

4.1 Variable neighborhood search (VNS)

VNS (Variable Neighborhood Search) je metaheuristička tehnika za rešavanje kombinatornih optimizacionih problema. Ideja je da se pretraga prostora rešenja vrši kroz različite "okoline" (neighborhoods) kako bi se istražio veći deo prostora rešenja i potencijalno pronašlo bolje rešenje.

Ključna karakteristika VNS-a su različite okoline koje se koriste za pretragu. Svaka okolina predstavlja drugačiji način menjanja rešenja kako bi se istražio novi deo prostora.

VNS dinamički menja okolinu tokom pretrage kako bi se izbeglo zaglavljivanje u lokalnom optimumu. Kada se postigne određeni uspeh u jednoj okolini, prelazi se na drugu okolinu radi dalje pretrage.

VNS kombinuje intenzivnu pretragu (fokusiranje na lokalno poboljšanje) sa diverzifikacijom (istraživanje novih rešenja) kako bi se pronašlo optimalno rešenje.

```python
def vns(graph, vns_params: dict):
    start_time = perf_counter()
    coloring = initialize(graph)
    value = calculate_value(graph, coloring)
    iter = 0
    best_iter = 0

    while perf_counter() - start_time < vns_params['time_limit']:
        for k in range(vns_params['k_min'], vns_params['k_max']):

            new_coloring = shaking(graph, coloring, k)
            new_value = calculate_value(graph, new_coloring)

            new_coloring, new_value, iter_local = local_search_invert_first_improvement(graph, new_coloring, new_value, iter)

            iter = iter_local

            if new_value < value or (new_value == value and random.random() < vns_params['move_prob']):
                if is_valid_coloring(graph, new_coloring):
                    if new_value < value:
                        best_iter = iter
                    value = new_value
                    coloring = deepcopy(new_coloring)

    return coloring, best_iter
```

Slika 5: Kod funkcije vns

1. **Glavna petlja**: Algoritam se nalazi u glavnoj petlji koja se izvršava dok nije dostignuto vreme izvršavanja definisano u `vns_params['time_limit']`.
2. **Okoline**: Za svaku vrednost k u opsegu od `vns_params['k_min']` do `vns_params['k_max']`, vrši se `"shaking"` okolina kako bi se generisalo novo bojenje *new_coloring*.
3. **Intenzifikacija**: Nakon generisanja novog bojenja, vrši se lokalna pretraga u okviru funkcije `local_search_invert_first_improvement` kako bi se pokušalo poboljšati bojenje. Ova faza se naziva intenzifikacija.
4. **Diverzifikacija**: U slučaju da lokalna pretraga ne pronađe bolje bojenje, ili se slučajno odabere da se prihvati isto rešenje (`new_value` == `value` i `random.random() < vns_params['move_prob']`), algoritam prelazi na sledeću vrednost k radi diverzifikacije pretrage.
5. **Kriterijum za prihvatanje novog rešenja**: Novo bojenje se prihvata ako ima manju vrednost od trenutnog bojenja ili ako ima istu vrednost ali je slučajno odabrano da se prihvati (`random.random() < vns_params['move_prob']`).
6. **Uslov za završetak**: Algoritam se zaustavlja kada dostigne vreme izvršavanja ili kada nema poboljšanja u određenom broju iteracija.

Funkcija ***local_search_invert_first_improvement***

```
def local_search_invert_first_improvement(graph, coloring, value, iter):

    new_coloring = deepcopy(coloring)
    improved = True
    best_iter = iter

    while improved:
        improved = False
        unbiased_order = list(range(len(new_coloring)))
        random.shuffle(unbiased_order)

        for i in unbiased_order:
            iter += 1
            old = new_coloring[i]
            new_coloring[i] = random.randrange(len(graph.edges))
            new_value = calculate_value(graph, new_coloring)

            if new_value < value:
                value = new_value
                improved = True
                best_iter = iter
                break
            else:
                new_coloring[i] = old

    return new_coloring, value, best_iter
```

Slika 6: Kod funkcije *local_search_invert_first_improvement*

Algoritam se nalazi u petlji koja se izvršava dok se ne postigne lokalno poboljšanje. Za svaku iteraciju, generiše se nasumičan redosled indeksa grana *unbiased_order* kako bi se izbeglo pristrasnost pretrage ka određenim granama.

Za svaki indeks grane u `unbiased_order`, pokušava se promeniti boja te grane na nasumično odabranu boju. Ako se novim bojenjem dobije bolja vrednost funkcije cilja (manja vrednost *value*), nova boja se zadržava i postavlja se *improved=True*, što ukazuje da je postignuto lokalno poboljšanje.

Ako je postignuto poboljšanje, ažuriraju se vrednosti *value* i *best_iter* (indeks iteracije u kojoj je postignuto poboljšanje). Petlja se prekida ako nije bilo poboljšanja u bojenju u trenutnoj iteraciji.

4.2 Simulirano kaljenje

Simulirano kaljenje (eng. **simulated annealing**) je metoda optimizacije inspirisana procesom kaljenja metala. Ideja je simulirati proces kaljenja metala kako bi se pronašlo globalno optimalno rešenje problema optimizacije. Proces kaljenja metala podrazumeva zagrevanje metala do visoke temperature, zatim postepeno hlađenje, što dovodi do promene strukture metala i smanjenja energetskih stanja.

U kontekstu optimizacije, simulirano kaljenje funkcioniše tako što se rešenje problema postepeno poboljšava tokom iteracija. U svakoj iteraciji, algoritam bira novo rešenje koje može biti bolje ili lošije od prethodnog rešenja, s tim što je verovatnoća prihvatanja lošijeg rešenja veća na početku (kao što je temperatura visoka u kaljenju metala) i postepeno opada kako se iteracije nastavljaju (kao što temperatura pada tokom kaljenja metala). Ovaj proces omogućava algoritmu da izbegne zaglavljenje u lokalnom optimumu i istraži šire područje prostora rešenja.

Simulirano kaljenje se često koristi za rešavanje problema optimizacije koji imaju veliki broj lokalnih optimuma ili kada je funkcija cilja "šumovita"ili nedifirencijabilna.

```python
def simulated_annealing(graph, max_iter=1000, initial_temperature=100.0, cooling_rate=0.95):
    coloring_result = initialize(graph)
    value = calculate_value(graph, coloring_result)

    best_coloring = None
    best_value = float('inf')

    current_temperature = initial_temperature
    iter_found = 0

    for i in range(max_iter):

        if current_temperature < 0.1:
            break

        if is_valid_coloring(graph, coloring_result):
            # reduce the number of colors by one
            new_coloring = make_small_change_colors(coloring_result)
            new_value = calculate_value(graph, new_coloring)
        else:
            # change one random position to a different color
            new_coloring = make_small_change_shuffle(graph, coloring_result)
            new_value = calculate_value(graph, new_coloring)

        if new_value < value:
            coloring_result = deepcopy(new_coloring)
            value = new_value
            if new_value < best_value:
                if is_valid_coloring(graph, new_coloring):
                    best_coloring = deepcopy(new_coloring)
                    best_value = new_value
                    iter_found = i
        else:
            delta = new_value - value
            if delta < 0 or random.random() < pow(2.71828, delta / current_temperature):
                coloring_result = deepcopy(new_coloring)
                value = new_value

        current_temperature *= cooling_rate

    return best_coloring, iter_found
```

Slika 7: Kod funkcije *simulated_annealing*

Algoritam pravi varijaciju trenutnog bojenja. Ako su sve grane različito obojene, algoritam pokušava da smanji broj korišćenih boja. U suprotnom, menja boju jedne nasumične grane.

Ako je novo bojenje bolje od trenutnog, uvek ga prihvata. Ako nije, novo bojenje se prihvata sa određenom verovatnoćom, koja zavisi od razlike u kvalitetu bojenja i trenutne temperature. Nakon svake iteracije, temperatura se postepeno smanjuje kako bi se smanjila verovatnoća prihvatanja lošijih rešenja.

Algoritam pamti najbolje pronađeno bojenje. Algoritam se zaustavlja kada temperatura postane dovoljno niska ili kada se dostigne maksimalan broj iteracija. Simulirano kaljenje omogućava algoritmu da istražuje različita rešenja, čak i lošija, kako bi pronašao globalno najbolje bojenje grana grafa.

4.3 Genetski algoritam

Genetski algoritam je simulacija evolucije u prirodi. Počinjemo sa populacijom nasumično generisanih rešenja. Svako rešenje ima svoju "prilagođenost"ili "fitnes"koji odražava koliko je dobro u rešavanju problema. Zatim, bolja rešenja imaju veću šansu da se reprodukuju i kombinuju međusobno, simulirajući ukrštanje genetskog materijala. Ponekad nova rešenja mogu imati i nasumične promene, simulirajući genetsku mutaciju. Ovaj proces se ponavlja kroz generacije, sa nadom da će se populacija poboljšavati vremenom. Algoritam se zaustavlja kada dostigne određeni broj iteracija ili kada postigne zadovoljavajuće rešenje.

Elitizam je koncept u genetskom algoritmu gde se najbolje prilagođene jedinke direktno prenose u sledeću generaciju, bez ikakvih promena ili ukrštanja. Ovaj koncept pomaže u očuvanju najboljih rešenja tokom evolucije populacije, sprečavajući ih da budu izgubljena ili promenjena u procesu reprodukcije. Elitizam doprinosi konvergenciji algoritma ka boljim rešenjima.

```python
def genetic_algorithm(graph, population_size, num_generations, tournament_size, elitism_size, mutation_prob):

    population = [Individual(graph) for _ in range(population_size)]
    new_population = population.copy()

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

        for j in range(elitism_size, population_size, 2):
            parent1 = selection(population, tournament_size)
            parent2 = selection(population, tournament_size)

            crossover(parent1, parent2, child1=new_population[j], child2=new_population[j+1])

            mutation(new_population[j], mutation_prob)
            mutation(new_population[j+1], mutation_prob)

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

        population = new_population.copy()

        best_individual = max(population, key=lambda x: x.fitness)

        idx = population.index(best_individual)

        if is_valid_coloring(graph, best_individual.coloring):
            return best_individual, idx
        else:
            return -100, -100
```

Slika 8: Kod algoritma *genetic_algorithm*

Početna populacija je inicijalizovana nasumično.

Elitizam - Na početku svake iteracije, izdvajaju se elitisti – najbolje jedinke.

Selekcija- Iz populacije se biraju jedinke koje će preživeti i reprodukovati se. Bolje prilagođene jedinke imaju veću šansu da budu izabrane.

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

Slika 9: Kod funkcije *selection*

Ukrštanje - Odabrane jedinke se kombinuju (ukrštaju) kako bi se generisale nove jedinke. Ovaj korak simulira ukrštanje genetskog materijala kod organizama.

```
def crossover(parent1, parent2, child1, child2):
    random_pos = random.randrange(0, len(parent1.coloring))

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

    child2.coloring[:random_pos] = parent2.coloring[:random_pos]
    child2.coloring[random_pos:] = parent1.coloring[random_pos:]
```

Slika 10: Kod funkcije *crossover*

Ovde je takođe dobro došla reprezentacija bojenja putem permutacija, jer jednostavnim ukrštanjem levog dela permutacije A i desnog dela permutacije B možemo dobiti novo bojenje.

Mutacija - U nekim slučajevima, nova jedinka može da mutira, tj. da ima nasumične promene u svom genetskom materijalu. Ova operacija pomaže u očuvanju diverziteta populacije. Nove jedinke zamenjuju stare u populaciji.

```
def mutation(individual, mutation_prob):
    for i in range(len(individual.coloring)):
        if random.random() < mutation_prob:
            new_color = random.randrange(1, individual.num_of_edges)
            while new_color == individual.coloring[i]:
                new_color = random.randrange(1, individual.num_of_edges)
            individual.coloring[i] = new_color
```

Slika 11: Kod funkcije *mutation*

Za svaku granu u bojenju, postoji određena verovatnoća mutation_prob da će se boja promeniti. Ako je ta verovatnoća zadovoljena, boja se menja nasumično na neku drugu boju, osim na trenutnu boju grane.

Algoritam se završava ako pronađe bojenje koje je validno za dati graf ili ako dostigne zadati broj generacija. U slučaju da nije pronađeno validno bojenje, algoritam vraća vrednosti (-100, -100) kao indikaciju da nije uspeo da pronađe rešenje.

4.4 Optimizacija rojem čestica

PSO je inspirisan ponašanjem jata ptica ili roja insekata. U PSO-u, rešenje problema predstavljeno je kao čestica koja se kreće kroz prostor pretrage rešenja. Svaka čestica ima svoju poziciju i brzinu koja joj pomaže da se kreće kroz prostor. Cilj je da čestice sarađuju i pretraže prostor kako bi pronašle najbolje rešenje.

Osnovna ideja je da čestice prate svoju najbolju poziciju do sada (lokalno najbolje rešenje) i najbolju poziciju koju je pronašla bilo koja čestica u jatu (globalno najbolje rešenje). Kroz iteracije algoritma, čestice se kreću ka svojoj najboljoj poziciji i ka globalno najboljoj poziciji, simulirajući proces pretrage i konvergencije ka optimalnom rešenju.

PSO je efikasan za probleme optimizacije koji imaju kontinuirani prostor pretrage rešenja, gde se rešenja mogu predstaviti kao vektori brojeva. Algoritam je jednostavan za implementaciju i često brzo konvergira ka optimalnom rešenju. Međutim, može imati izazove u situacijama sa više lokalnih optimuma.

U okviru klase Particle, imamo sledeće bitne funkcije:

- `__init__`: Konstruktor koji inicijalizuje bojenje čestice nasumično, postavlja brzinu čestice i računa vrednost bojenja. Ako je ovo prva čestica ili je vrednost bojenja bolja od globalno najbolje vrednosti, ažurira globalno najbolje bojenje.
- `calculate_conflicts`: Ova funkcija računa broj konflikata u bojenju čestice. Za svaku granu grafa, proverava da li je boja te grane ista kao boja neke susedne grane. Ako je boja ista, povećava se broj konflikata. Na kraju, funkcija vraća broj konflikata podeljen sa 2 (jer svaki konflikt se broji dva puta).
- `update_colors`: Ova funkcija ažurira bojenje čestice i proverava da li je novo bojenje bolje od lično najboljeg bojenja i globalno najboljeg bojenja. Ako je novo bojenje bolje, ažurira lično najbolje bojenje i globalno najbolje bojenje. Ako novo bojenje nije validno (tj. nije zadovoljen uslov validnosti bojenja), funkcija vraća vrednost -100.
- `update_velocity`: Ova funkcija ažurira brzinu čestice koristeći formulu optimizacije rojem čestica. Brzina se ažurira kombinacijom kognitivne i socijalne komponente. Kognitivna komponenta predstavlja razliku između lično najboljeg bojenja čestice i trenutnog bojenja, dok socijalna komponenta predstavlja razliku između globalno najboljeg bojenja u roju i trenutnog bojenja. Funkcija koristi nasumične vrednosti **r_p** i **r_s** kako bi se izbeglo zaglavljenje u lokalnom minimumu.

Implementacija samog algoritma:

```
def pso(graph, swarm_size, num_iters, c_i, c_p, c_s):
    num_nodes = len(graph.nodes())
    num_edges = len(graph.edges())
    num_colors = num_nodes  # Inicijalno postavljamo broj boja na broj čvorova
    swarm = [Particle(graph, num_colors, c_i, c_p, c_s) for _ in range(swarm_size)]
    for i in range(num_iters):
        for p in swarm:
            p.update_velocity(i + 1)  # Povećavamo faktor c_i kako se iteracije povećavaju
            p.update_colors()
        # Ažuriramo broj boja ako smo pronašli bolje bojanje
        if Particle.swarm_best_value < num_colors:
            num_colors = Particle.swarm_best_value
            for p in swarm:
                p.num_colors = num_colors
    return Particle.swarm_best_coloring, Particle.swarm_best_value, num_colors
```

Slika 12: Kod algoritma *Particle Swarm Optimization*

Prvo se računaju broj čvorova i ivica u grafu, što će se koristiti kasnije u algoritmu. Inicijalno se broj boja postavlja na broj čvorova. Zatim se kreira roj čestica, gde svaka čestica ima svoje bojenje, brzinu i vrednost bojenja. Za svaku česticu u roju, ažuriraju se brzina i bojenje. Proverava se da li je pronađeno bolje bojenje u roju. Ako jeste, ažurira se broj bojai svi česticama se dodeljuje novi broj boja. Na kraju se vraćaju globalno najbolje bojenje, njegova vrednost i broj boja.

```python
class Particle:
    swarm_best_coloring = None
    swarm_best_value = None

    def __init__(self, graph, num_colors, c_i, c_p, c_s):
        self.c_i = c_i
        self.c_p = c_p
        self.c_s = c_s

        self.graph = graph
        self.num_colors = num_colors
        self.colors = np.random.randint(0, num_colors, len(graph.edges()))
        self.velocity = np.random.uniform(-1, 1, len(graph.edges()))

        self.personal_best_coloring = self.colors.copy()
        self.value = self.calculate_conflicts()
        self.personal_best_value = self.value
        if Particle.swarm_best_value is None or self.value < Particle.swarm_best_value:
            Particle.swarm_best_value = self.value
            Particle.swarm_best_coloring = self.colors.copy()

    def calculate_conflicts(self):
        conflicts = 0
        edge_color = {edge: self.colors[i] for i, edge in enumerate(self.graph.edges())}
        for edge in self.graph.edges():
            u, v = edge
            u_edges = neighbor_edges_of_node(self.graph, u)
            v_edges = neighbor_edges_of_node(self.graph, v)
            u_edges.remove(edge)
            v_edges.remove(edge)

            for i in range(len(u_edges)):
                if edge_color[u_edges[i]] == edge_color[edge]:
                    conflicts += 1

            for j in range(len(v_edges)):
                if edge_color[v_edges[j]] == edge_color[edge]:
                    conflicts += 1
        return conflicts // 2

    def update_colors(self):
        self.colors = np.clip(self.colors + self.velocity, 0, self.num_colors - 1)
        self.value = self.calculate_conflicts()
        if self.value < self.personal_best_value and is_valid_coloring(self.graph, self.colors):
            self.personal_best_value = self.value
            self.personal_best_coloring = self.colors.copy()
            if self.value < Particle.swarm_best_value:
                Particle.swarm_best_value = self.value
                Particle.swarm_best_coloring = self.colors.copy()
        else:
            return -100

    def update_velocity(self, iteration):
        cognitive_velocity = self.personal_best_coloring - self.colors
        social_velocity = Particle.swarm_best_coloring - self.colors
        r_p = np.random.random(len(cognitive_velocity))
        r_s = np.random.random(len(social_velocity))
        self.velocity = (self.c_i / iteration) * self.velocity + \
                        self.c_p * r_p * cognitive_velocity + \
                        self.c_s * r_s * social_velocity
```

Slika 13: Klasa *Particle*

4.5 Kolonija mrava

Ant colony optimization (ACO) je inspiriran ponašanjem mrava u potrazi za hranom. Mravi koriste feromone kako bi komunicirali i označavali puteve do hrane. U ACO algoritmima, simulira se ovo ponašanje kroz graf. Mravi, krećući se po grafu ostavljaju feromone, a ostali mravi biraju puteve na osnovu feromona i heuristike. Kroz iteracije, feromoni se osvežavaju prema uspešnosti pronađenih rešenja, te se algoritam usmerava prema boljim rešenjima.

```python
def ant_colony_optimization(graph, num_ants, num_iterations, alpha, beta, evaporation_rate, pheromone_deposit):
    num_nodes = len(graph.nodes())
    num_edges = len(graph.edges())
    num_colors = num_nodes  # Initially set the number of colors to the number of nodes

    # Initialize pheromone matrix
    pheromone_matrix = np.ones((num_edges, num_colors))

    # Initialize best coloring and its value
    best_coloring = np.zeros(num_edges, dtype=int)
    best_value = float('inf')

    for iteration in range(num_iterations):
        # Initialize ant solutions
        ant_solutions = np.zeros((num_ants, num_edges), dtype=int)

        # Construct ant solutions
        for ant in range(num_ants):
            for edge in range(num_edges):
                probabilities = np.zeros(num_colors)
                if graph.has_node(edge):  # Check if edge is a valid node
                    for color in range(num_colors):
                        if is_valid_coloring(graph, ant_solutions[ant]):
                            probabilities[color] = (pheromone_matrix[edge][color] ** alpha) * ((1 / (graph.degree(edge) + 1)) ** beta)

                sum_probabilities = np.sum(probabilities)
                if sum_probabilities == 0:
                    probabilities = np.ones_like(probabilities) / len(probabilities)
                else:
                    probabilities /= sum_probabilities
                if num_colors > 0:
                    ant_solutions[ant][edge] = np.random.choice(range(num_colors), p=probabilities)



        # Update pheromone matrix
        pheromone_matrix *= (1 - evaporation_rate)
        for ant in range(num_ants):
            if is_valid_coloring(graph, ant_solutions[ant]):
                value = calculate_conflicts(graph, ant_solutions[ant])
                if value < best_value:
                    best_value = value
                    best_coloring = ant_solutions[ant]
                for edge in range(num_edges):
                    pheromone_matrix[edge][ant_solutions[ant][edge]] += pheromone_deposit / (value + 1e-10)


        # Update number of colors if a better coloring is found
        if best_value < num_colors:
            num_colors = best_value

    return best_coloring, best_value, len(np.unique(best_coloring))
```

Slika 14: Kod funkcije ant_colony_optimization

Objašnjenja parametara:

- **Alpha:** Parametar koji kontroliše uticaj feromona na izbor boje. Veće vrednosti alpha povećavaju verovatnoću da mrav izabere boju sa većim feromonima na bridu.
- **Beta:** Parametar koji kontroliše uticaj heuristike (trenutna boja, susedne boje itd.) na izbor boje. Veće vrednosti beta povećavaju verovatnoću da mrav izabere boju koja je dobro povezana sa trenutnim bridom.
- **Evaporation rate:** Stopa isparavanja feromona sa grana. Kontroliše koliko brzo feromoni nestaju sa vremenom. Veće vrednosti uzrokuju brže nestajanje feromona.
- **Pheromone deposit:** Količina feromona koju mrav ostavlja na bridu kad pređe preko njega. Ova vrednost obično zavisi o kvalitetu rešenja koje je pronašao mrav. Bolja rešenja ostavljaju više feromona.

Koristimo matricu feromona kako bismo modelirali trag koji mravi ostavljaju dok traže rešenje. Ova matrica se obično koristi za praćenje feromona na svakoj grani i za svaku boju. Kada mrav pređe preko grana, ostavlja feromone koji mogu uticati na izbor boje za sledeći put.

Razlog za korišćenje matrice feromona je specifičnost problema bojenja grana. Za razliku od nekih drugih problema u kojima mravi tragaju za najkraćim putem ili optimalnim rešenjem, kod problema bojenja grana, bitno je da mravi ostavljaju tragove koji se međusobno poništavaju. Drugim rečima, ako jedan mrav oboji granu crvenom bojom, drugi mrav ne bi trebao koristiti isti put ili boju ukoliko to nije optimalno rešenje.

Dakle, matrica feromona omogućava mravima da komuniciraju indirektno putem feromona, što je ključno za konvergenciju algoritma ka boljim rešenjima kod problema bojenja grana.

Algoritam, korak po korak:

1. **Inicijalizacija**:

   Definišu se brojevi čvorova i grana.

   - `num_colors = num_nodes`: Na početku, broj boja se postavlja na broj čvorova. Ovo će se kasnije ažurirati ako se pronađe bolje bojenje.
   - `pheromone_matrix = np.ones((num_edges, num_colors))`: Kreira se matrica feromona dimenzija `num_edges x num_colors`. Svaki red u ovoj matrici će odgovarati jednoj grani u grafu, a svaka kolona će odgovarati jednoj boji. Početna vrednost feromona se postavlja na 1 za sve elemente matrice.
   - `best_coloring = np.zeros(num_edges, dtype=int)`: Inicijalno se postavlja niz `best_coloring` dužine `num_edges` koji će sadržati boje za svaku granu. Početno se sve boje postavljaju na 0 (ili bilo koju drugu početnu vrednost).
   - `best_value = float('inf')`: Inicijalno se postavlja `best_value` na beskonačno, što označava da još uvek nije pronađeno bolje bojenje. Ova vrednost će se ažurirati kako algoritam bude napredovao.

2. **Glavna petlja**:

   - `ant_solutions = np.zeros((num_ants, num_edges), dtype=int)`: Za svaku iteraciju, inicijalizuje se matrica `ant_solutions` dimenzija `num_ants x num_edges`. Svaki red u ovoj matrici će predstavljati jedno rešenje (bojenje) koje konstruiše jedan mrav.
   - Za svakog mrava `(for ant in range(num_ants):)` se konstruiše rešenje za bojenje svake grane grafa `(for edge in range(num_edges):)`.
   - `probabilities = np.zeros(num_colors)`: Za svaku granu se inicijalizuje niz verovatnoća za svaku boju.
   - Petlja za izračunavanje verovatnoća za svaku boju:
     `probabilities[color] = (pheromone_matrix[edge][color] ** alpha) * ((1 / (graph.degree(edge) + 1)) ** beta)`: Za svaku boju se izračunava verovatnoća na osnovu formule koja kombinuje uticaj feromona i heuristike

3. **Normalizacija verovatnoća**:

   - `sum_probabilities = np.sum(probabilities)`: Računa se ukupna verovatnoća za sve boje.
   - Ako je suma verovatnoća jednaka nuli, postavljaju se sve verovatnoće na jednake vrednosti:
     `if sum_probabilities == 0:`
     `probabilities = np.ones_like(probabilities) / len(probabilites)`

     Inače, verovatnoće se normalizuju: `probabilities /= sum_probabilities.`
   - Odabir boje na osnovu verovatnoća:
     `ant_solutions[ant][edge] = np.random.choice(range(num_colors), p=probabilities)`
     Boja za trenutnu granu se bira nasumično na osnovu verovatnoća.
   - Normalizacija verovatnoća se radi kako bi se osiguralo da sve verovatnoće sumiraju u 1. To je važno jer verovatnoće predstavljaju distribuciju verovatnoća za izbor boje, pa njihove vrednosti treba prilagoditi tako da zbir svih verovatnoća bude 1 (jer je to osnovni princip verovatnoća).

4. **Ažuriranje matrice feromona**

- Isparavanje feromona: `pheromone_matrix *= (1 - evaporation_rate)`
  Smanjuje se količina feromona na svakom bridu množenjem matrice feromona sa iznosom `1 - evaporation_rate`.
  Ovo je korak isparavanja feromona, gde se smanjuje nivo feromona kako bi se postepeno zaboravljale loše putanje i omogućilo istraživanje novih putanja.
- Ažuriranje feromona za svakog mrava:
  - `if is_valid_coloring(graph, ant_solutions[ant])`: Proverava se da li je bojenje koje je pronašao trenutni mrav validno, tj. da li su zadovoljeni uslovi bojenja (da susedne grane imaju različite boje).
  - `value = calculate_conflicts(graph, ant_solutions[ant])`: Računa se broj sukoba u bojenju koje je pronašao mrav. Ovaj broj će se koristiti za evaluaciju kvaliteta pronađenog rešenja.
  - `if value < best_value:` Proverava se da li je pronađeno bojenje bolje od najboljeg do sada pronađenog. Ako jeste, ažuriraju se najbolje bojenje i njegova vrednost: `best_value = value` i `best_coloring = ant_solutions[ant]`.
- Ažuriranje feromona na osnovu kvaliteta pronađenog bojenja:
  `pheromone_matrix[edge][ant_solutions[ant][edge]] += pheromone_deposit / (value + 1e-10)`

  Ako je bojenje validno, povećava se količina feromona na bridu za boju koja je korišćena u bojenju. Ovde se koristi `pheromone_deposit` kako bi se odredilo koliko feromona treba dodati, pri čemu se uzima u obzir i kvalitet bojenja (manji broj sukoba vodi ka većem dodavanju feromona). Dodata je i mala vrednost (1e-10) kako bi se izbeglo deljenje nulom u slučaju da je value jednak nuli.

Algoritam se zaustavlja nakon određenog broja iteracija `(num_iterations)`

5 Poređenje

5.1 Iteracije naspram broja grana

Kada se suočavamo s problemom minimalnog bojenja grafa, imamo na raspolaganju širok spektar algoritama koji pružaju različite pristupe i performanse. Svaki od ovih algoritama nosi sa sobom jedinstvene karakteristike, prednosti i mane, čineći ih relevantnim izborom u zavisnosti od specifičnih zahteva problema.

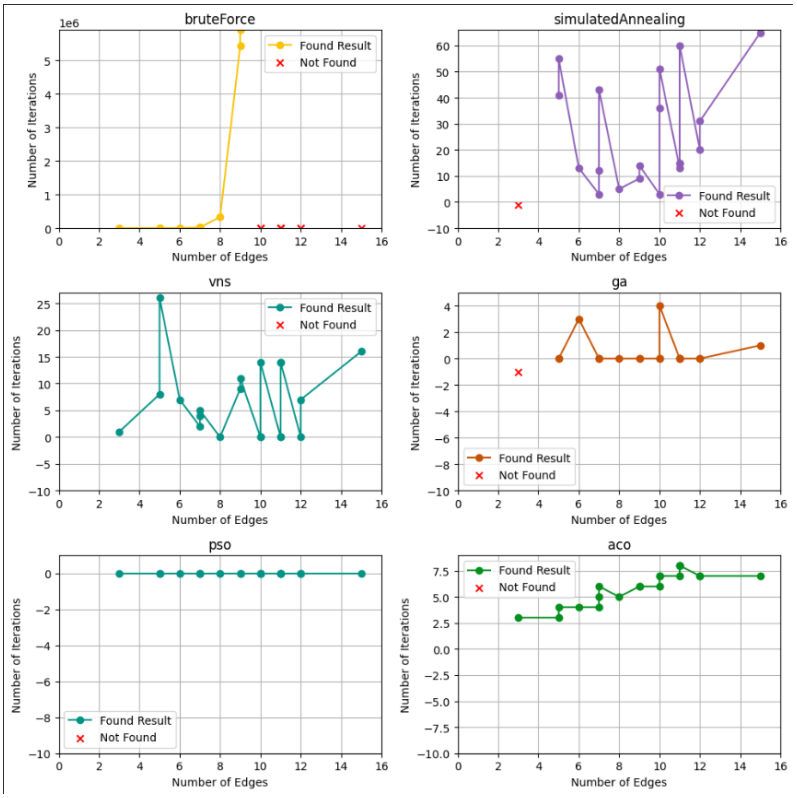
Prvo, analizirajmo primenu **grube sile**. Ovaj pristup je poznat po svojoj preciznosti - sposoban je da pronađe tačna rešenja. Međutim, njegova glavna mana leži u vremenskoj složenosti, koja eksponencijalno raste sa povećanjem broja čvorova grafa. To ga čini neefikasnim za veće grafove, gde pretraga svih mogućih kombinacija postaje neprihvatljivo dugotrajna.

Sa druge strane, **simulirano kaljenje** nudi fleksibilnost u rešavanju ovog problema. Njegova probabilistička priroda omogućava mu da istražuje različite delove prostora rešenja, što može dovesti do dobrih rezultata za različite veličine grafova. Ipak, baš zbog takve prirode, može doći do varijacija u rezultatima u zavisnosti od inicijalnih parametara ili iteracija.

Variable Neighborhood Search (VNS) je još jedan prilagodljiv algoritam koji koristi različite `"okoline"` u pretrazi prostora rešenja. Ovo ga čini sposobnim da pronađe raznovrsna rešenja, ali takođe može dovesti do varijacija u performansama, s obzirom na broj okolina i redosled njihove primene.

Genetski algoritmi (GA) i **Particle Swarm Optimization (PSO)** su algoritmi koji pokazuju stabilne performanse za različite veličine grafova. Ovi algoritmi su efikasni i pružaju konzistentne rezultate.

Na kraju, **Ant Colony Optimization (ACO)** je algoritam koji se izdvaja po svom sporijem rastu performansi za veće grafove, ali može biti koristan za srednje velike grafove. ACO simulira ponašanje kolonije mrava u pretrazi prostora rešenja, čime omogućava pronalaženje dobrih rešenja za određene klase problema bojenja grafova.



Slika 15: Grafovski prikaz poredjenja svih algoritama

5.2 Potrebno vreme u odnosu na broj grana

Kada uporedimo ove algoritme za minimalno bojenje grafova u pogledu vremena izvršavanja i broja grana, dolazimo do zanimljivih zaključaka.

Algoritam grube sile, iako precizan, pokazuje spor rast vremena izvršavanja kako se povećava broj čvorova grafa. Ograničen je na oko 180 sekundi i nakon što pređe 9 čvorova ne uspeva da pronađe rešenje u zadatom vremenskom intervalu, što ga čini nepraktičnim za veće grafove.

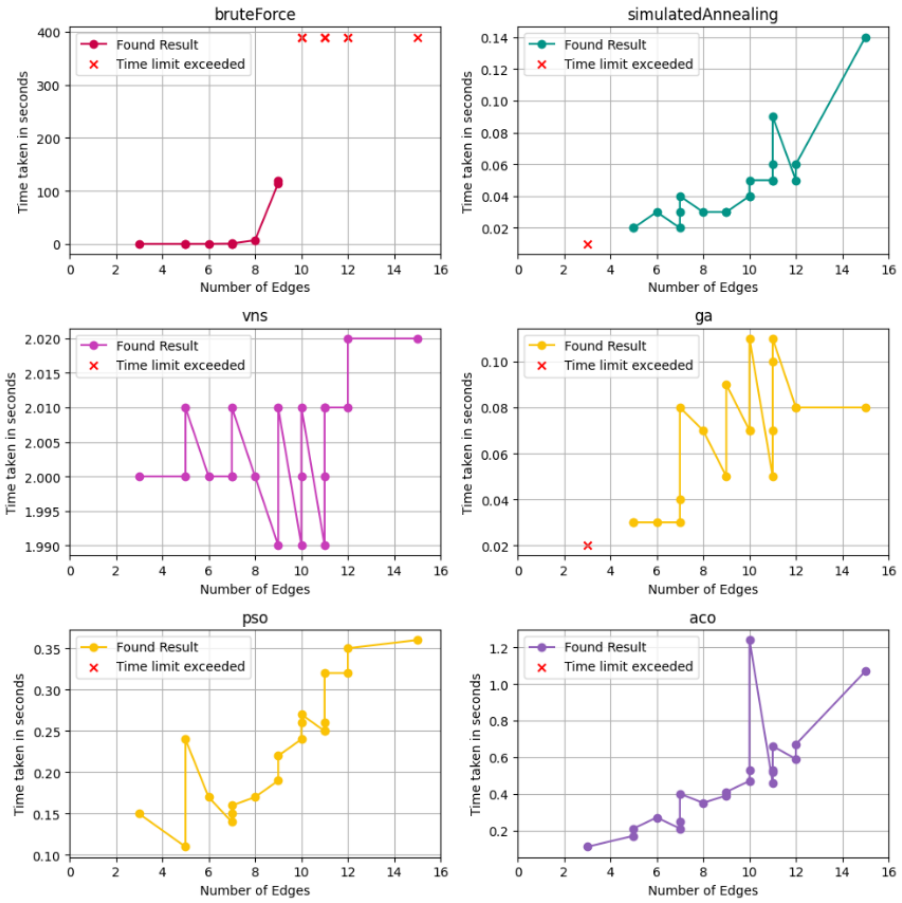
Simulirano kaljenje takođe ima postepen rast vremena izvršavanja, između 0.02 i 0.14 sekundi, što ga čini bržim znatno bržim od brute force-a, ali moramo uzeti u obzir da ne daje uvek najoptimalnije rešenje.

Variable Neighborhood Search (VNS) pokazuje velike varijacije u vremenu izvršavanja, ali optimalne rezultate često postiže u vremenskom rasponu od 2 do 2.010 sekundi, što ga čini dobrom opcijom za srednje velike grafove.

Genetski algoritmi imaju varijabilno vreme izvršavanja, ali primećuje se spor rast između 0.03 i 0.10 sekundi, što ih čini stabilnim i efikasnim za različite veličine grafova.

Particle Swarm Optimization (PSO) i **Ant Colony Optimization (ACO)** pokazuju slične obrasce rasta u vremenu izvršavanja. PSO raste između 0.1 i 0.35 sekundi, dok ACO polako raste između 0.2 i 1.2 sekunde sa povećanjem broja grana.

Uzimajući u obzir ove rezultate, Genetski algoritmi i VNS mogu biti dobar izbor za različite veličine grafova zbog relativno stabilnih performansi u vremenskom smislu. Brute Force je precizan ali sporo raste, dok PSO i ACO pokazuju dosta malo vreme izvršavanja, posebno za veće grafove, ali na uštrb preciznosti.



Slika 16: Grafovski prikaz poredjenja svih algoritama

5.3 Broj boja naspram potrebnog vremena za izvršavanje

Kada poredimo broj boja u rešenju sa vremenom izvršavanja algoritama za minimalno bojenje grafova, ulazimo u suptilniji domen gde se susrećemo s mnogim nijansama u efikasnosti i preciznosti ovih algoritama.

Brute Force algoritam, iako precizan, često se suočava s ograničenjima kada je reč o vremenskom izvršavanju. On može pronaći najbolja rešenja sa samo 2-4 boje za manje grafove, ali njegova vremenska složenost eksponencijalno raste sa povećanjem veličine grafa te postaje neefikasan i često ne pronalazi rešenja.

Simulated Annealing je algoritam koji se ističe u pronalaženju dobrog balansa između brzine izvršavanja i kvaliteta rešenja, naročito za srednje velike grafove. Njegova probabilistička priroda omogućava mu da istražuje različite delove prostora rešenja, što rezultira često prihvatljivim brojem boja u rešenju.

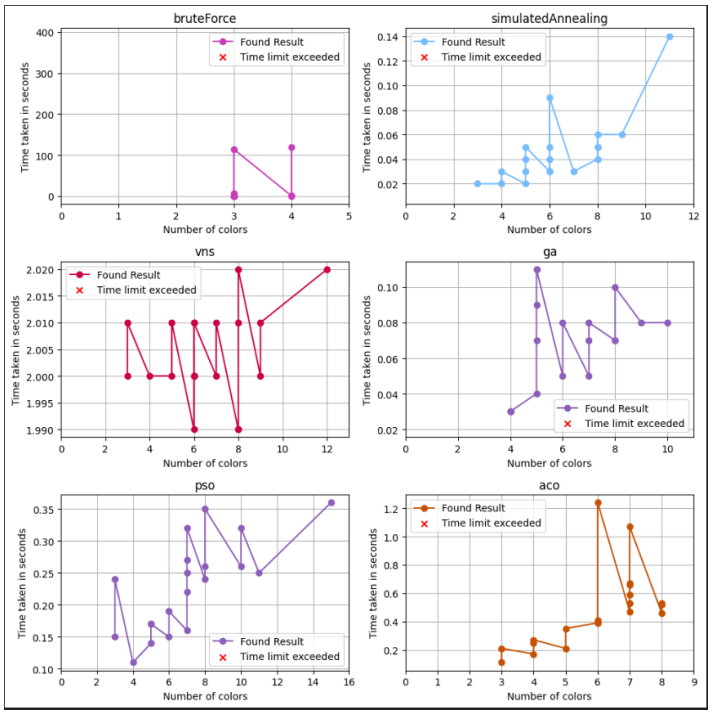
Variable Neighborhood Search (VNS) može imati varijacije u broju boja u zavisnosti od veličine grafa i strukture samog grafa. Ovaj algoritam je fleksibilan i može pronaći dobra rešenja, ali njegova efikasnost može varirati.

Genetski algoritmi su poznati po svojoj stabilnosti i sposobnosti pronalaženja dobrih rešenja u razumnim vremenskim intervalima. Oni često pružaju rešenja sa relativno malim brojem boja i mogu biti pouzdan izbor za različite veličine grafova.

Particle Swarm Optimization (PSO) može pokazati tendenciju povećanja broja boja sa povećanjem broja grana, što može biti kompromis između vremena izvršavanja i kvaliteta rešenja.

Ant Colony Optimization (ACO), iako brz za manje grafove, može postati prosečan u odnosu na druge algoritme kada se suočava s većim grafovima. Njegova efikasnost može opasti kako raste složenost grafa.

U krajnjem, ako je prioritet minimalan broj boja, algoritmi poput Brute Force-a, Genetskih algoritama ili Simulated Annealing-a mogu biti korisni. Za brzinu izvršavanja, algoritmi poput PSO-a ili VNS-a mogu biti bolji izbori.



Slika 17: Grafovski prikaz poredjenja svih algoritama

5.4 Celokupno poređenje

Kada posmatramo ova tri poredjenja - iteracije naspram broja grana, potrebno vreme u odnosu na broj grana, i broj boja naspram potrebnog vremena za izvršavanje, primećujemo nekoliko značajnih paralela i zaključaka.

Prvo, algoritmi kao što su Brute Force, Simulirano kaljenje i Genetski algoritmi pokazuju visoku preciznost u rešavanju problema bojenja grafova, što se vidi kroz njihovu sposobnost da pronađu rešenja sa malim brojem boja. Međutim, ovi algoritmi često imaju vremenske i resursne ograničenja, naročito za veće grafove. Sa druge strane, algoritmi poput PSO-a, VNS-a i ACO-a imaju tendenciju da budu brži u izvršavanju, ali može doći do kompromisa u preciznosti rešenja, što se vidi kroz povećanje broja boja u rešenju.

Druga paralela koju možemo izvući je da su algoritmi poput Simuliranog kaljenja i VNS-a često bolji izbor za srednje velike grafove, jer uspevaju da pronađu balans između brzine izvršavanja i optimalnosti rešenja. Sa druge strane, Genetski algoritmi i PSO mogu biti stabilni izbori za različite veličine grafova, pružajući konzistentne performanse u vremenskom smislu.

Kada je reč o broju boja u rešenju u odnosu na potrebno vreme za izvršavanje, možemo primetiti da postoji trade-off između efikasnosti i kvaliteta rešenja. Algoritmi koji su brži u izvršavanju, poput PSO-a i ACO-a, često imaju tendenciju da pronađu rešenja sa većim brojem boja, dok algoritmi koji su precizniji, poput Simuliranog kaljenja i Genetskih algoritama, mogu zahtevati više vremena za izvršavanje ali daju bolja rešenja sa manjim brojem boja.

6 Praktična primena

Bojenje grana potpunih grafova može se koristiti za raspoređivanje round-robin turnira u što manje rundi kako bi svaki par takmičara igrao jedan protiv drugog u jednoj od rundi; u ovoj primeni, vrhovi grafa odgovaraju takmičarima u turniru, grane odgovaraju utakmicama, a boje grana odgovaraju rundama u kojima se igraju utakmice. Slične tehnike bojenja mogu se koristiti i za raspoređivanje drugih sportskih parova koji nisu svi-protiv-svih; na primer, u Nacionalnoj fudbalskoj ligi, parovi timova koji će igrati jedan protiv drugog u datoj godini određuju se na osnovu rezultata timova iz prethodne godine, a zatim se algoritam bojenja grana primenjuje na graf formiran skupom parova kako bi se utakmice rasporedile za vikende u kojima se igraju.

Open shop raspoređivanje je način organizovanja proizvodnje gde se svaki predmet koji treba da se napravi sastoji od više koraka ili operacija. Svaki od ovih koraka mora da se obavi na određenoj mašini ili radnoj stanici. Ključna karakteristika open shop raspoređivanja je fleksibilnost u izboru redosleda izvršavanja operacija za svaki predmet. Ova fleksibilnost omogućava efikasno korišćenje resursa, kao što su mašine i radnici, kako bi se minimiziralo vreme potrebno za proizvodnju svih predmeta.

Ako su svi zadaci iste dužine, tada se ovaj problem može formalizovati kao problem bojenja grana bipartitnog multigrafa, u kojem vrhovi sa jedne strane bipartitne particije predstavljaju objekte koji treba da se proizvedu, vrhovi sa druge strane bipartitne particije predstavljaju proizvodne mašine, grane predstavljaju zadatke koji moraju da se obave, a boje predstavljaju vremenske korake u kojima svaki zadatak može da se obavi.

U komunikacijama putem optičkih vlakana, problem bojenja putanja je problem dodele boja (frekvencija svetlosti) parovima čvorova koji žele da komuniciraju međusobno, i putanjama kroz mrežu optičkih vlakana za svaki par, uz ograničenje da nijedne dve putanje koje dele segment vlakna ne koriste istu frekvenciju svetlosti. Putanje koje prolaze kroz isti komunikacioni prekidač, ali ne kroz isti segment vlakna, mogu koristiti istu frekvenciju. Kada je komunikaciona mreža organizovana kao zvezdana mreža, sa jednim centralnim prekidačem povezanim posebnim vlaknima sa svakim čvorom, problem bojenja putanja može biti modelovan tačno kao problem bojenja grana grafa ili multigrafa, u kojem čvorovi koji komuniciraju čine vrhove grafa, parovi čvorova koji žele da komuniciraju čine grane grafa, a frekvencije koje se mogu koristiti za svaki par čine boje problema bojenja grana. Za komunikacione mreže sa opštijom topologijom drveta, lokalna rešenja bojenja putanja za zvezdane mreže definisane svakim prekidačem u mreži mogu biti povezana zajedno kako bi se formiralo jedinstveno globalno rešenje.

7 Zaključak

Izbor odgovarajućeg algoritma za minimalno bojenje grafova zahteva pažljivo balansiranje između preciznosti rešenja, vremena izvršavanja i resursnih zahteva. Svaki od navedenih algoritama ima svoje prednosti i mane u ovim aspektima, te je važno prilagoditi izbor algoritma specifičnim zahtevima problema kako bi se postiglo optimalno rešenje. Kombinacija više algoritama ili optimizacija parametara može biti korisna strategija u rešavanju složenih problema bojenja grafova.

Prvo, veličina grafa je od suštinskog značaja. Kao što smo videli Brute Force, iako precizni, postaju nepraktični za velike grafove, dok algoritmi poput Genetskog algoritma i PSO-a mogu biti efikasniji za velike grafove, pružajući stabilne performanse s povećanjem broja čvorova.

Preciznost rešenja je takođe bitan faktor. Ako nam je potrebno tačno rešenje, Brute Force može biti izbor, ali ako je prihvatljiva određena mera aproksimacije, algoritmi poput Simuliranog kaljenja ili VNS-a mogu pružiti zadovoljavajuće rezultate uz manje vremenske troškove. Takođe, PSO ili Genetskih algoritama imaju tendenciju da budu brži, čineći ih prikladnijim u situacijama gde je brzina izvršavanja važna.

U krajnjem, pažljiv izbor algoritma za minimalno bojenje grafova zavisiće od specifičnih zahteva problema. Ako su preciznost i tačnost ključni, algoritmi poput Brute Force-a ili Genetskih algoritama mogu biti prioritet. Za veće grafove gde je brzina važna, PSO, Simulated Annealing ili VNS mogu pružiti dobar balans između efikasnosti i tačnosti. Važno je takođe imati na umu da kombinacija različitih algoritama ili optimizacija parametara može biti korisna strategija u rešavanju složenih problema bojenja grafova.

Literatura

[1] Minimum edge coloring,
online at: https://www.csc.kth.se/ viggo/wwwcompendium/node18.html