Minimum Edge Coloring

Računarska inteligencija

Petra Ignjatović i Anđela Jovanović

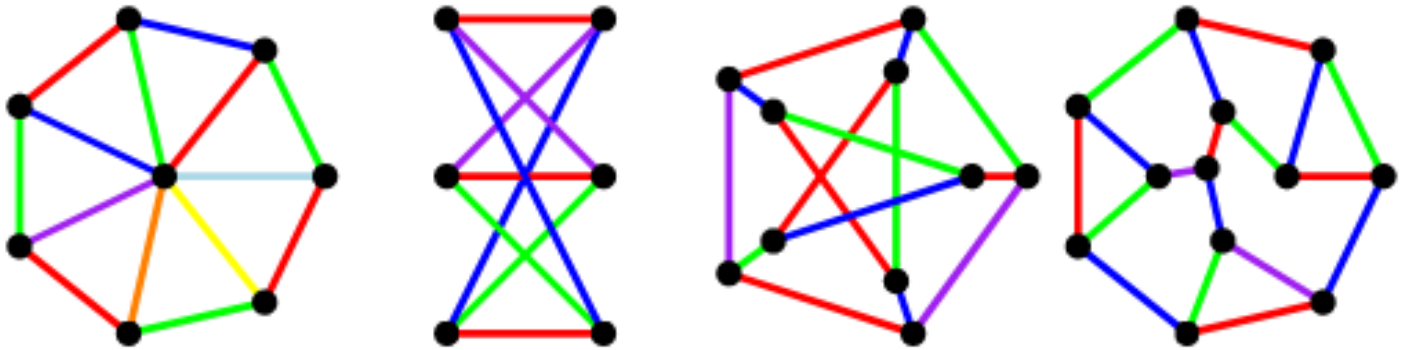
Matematički fakultet
Univerzitet u Beogradu

Beograd, 2024.

Pregled

Definicija problema

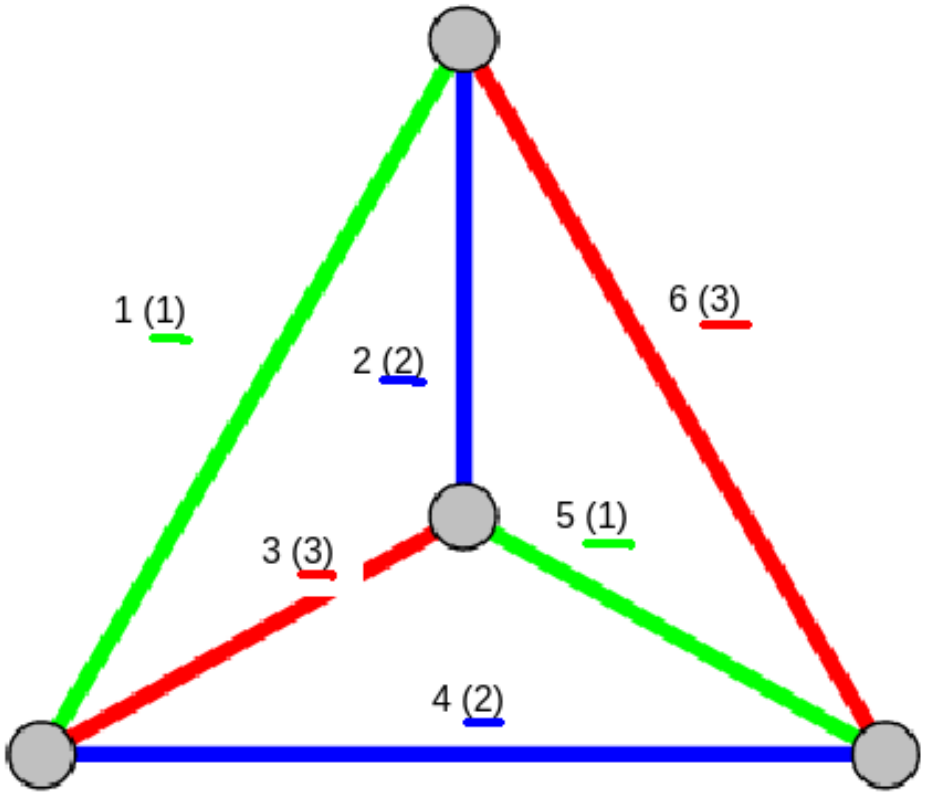
- Pravilno bojenje grafa: svake dve susedne grane nemaju istu boju
- Minimalno bojenje: traži se najmanji broj boja za ispravno bojenje
- Kako da pronađemo tačno i efikasno rešenje?



Slika: Primeri ispravno obojenih grafova

Reprezentacija grafova i provera bojenja

- Bojenje predstavljeno permutacijama brojeva od 0 do n-1
- `is_valid_coloring` proverava ispravnost bojenja



Primena grube sile

- Brute force: iscrpna pretraga svih mogućih bojenja
- Vreme izvršavanja eksponencijalno raste sa brojem grana

```
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
```

Variable neighborhood search (VNS)

- Metaheuristička tehnika za rešavanje kombinatornih optimizacionih problema

```
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
```

Simulirano kaljenje

- Metoda optimizacije inspirisana procesom kaljenja metala

```
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
```

Genetski algoritam

Simulacija evolucije u prirodi koja počinje sa populacijom nasumično generisanih rešenja.

- Svako rešenje ima svoju "prilagođenost" ili "fitnes" koji odražava koliko je dobro u rešavanju problema.
- 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.

Genetski algoritam - Kod

```
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
```

Genetski algoritam - Inicijalizacija populacije

- Početna populacija je inicijalizovana nasumično
- Elitizam - na početku svake iteracije, izdvajaju se elitisti – najbolje jedinke.

Genetski algoritam - Selekcija

- Selekcija je proces biranja jedinki koje će preživeti i reprodukovati se. Bolje prilagođene jedinke imaju veću šansu da budu izabrane.

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

Slika: Kod funkcije *selection*

Genetski algoritam - 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: Kod funkcije *crossover*

Genetski algoritam - 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.

```
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: Kod funkcije *mutation*

Genetski algoritam - Kriterijumi za završetak

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 odgovarajuću vrednost kao indikaciju da nije uspeo da pronađe rešenje.

Optimizacija rojem čestica

- PSO je inspirisan ponašanjem jata ptica ili roja insekata
- Kao kvalitet jedinki smo koristili funkciju koja racuna konflikte

```
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
```

Slika: Kod funkcije *calculate_conflicts*

Optimizacija rojem čestica

- `update_colors` ažurira bojenje čestice i proverava da li je novo bojenje bolje od lično najboljeg bojenja i globalno najboljeg bojenja
- `update_velocity` ažurira brzinu kombinacijom kognitivne i socijalne komponente. Kognitivna komponenta predstavlja razliku izmedu lično najboljeg bojenja čestice i trenutnog bojenja, dok socijalna komponenta predstavlja razliku izmedu globalno najboljeg bojenja u roju i trenutnog bojenja.

Kolonija mrava

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.
U svakoj iteraciji, svaki mrav konstruiše svoje rešenje koristeći verovatnoću prelaska na sledeći čvor zasnovanu na tragovima feromona i heuristici. Nakon što svaki mrav izgradi svoje rešenje, ažuriraju se tragovi feromona na osnovu kvaliteta pronađenih rešenja.

Kolonija mrava - depozit feromona

- Za svakog mrava u populaciji, i za svaku ivicu grafa, računa se verovatnoća da se odabere određena boja za tu ivicu. Verovatnoća se računa na osnovu tragova feromona na ivici i heurističke informacije o bojama susednih čvorova

```
# 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)
```

Kolonija mrava - isparavanje feromona

- Prvo se smanjuje količina feromona na svakoj ivici grafa kako bi se postiglo isparavanje feromona.
- Zatim se prolazi kroz svakog mrava i ažurira se najbolje bojenje i vrednost.
- Na kraju, ažuriraju se tragovi feromona na svakoj ivici na osnovu boje ivice u rešenju mrava, pri čemu se dodaje određena količina feromona proporcionalna kvalitetu rešenja.

```
# 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)
```

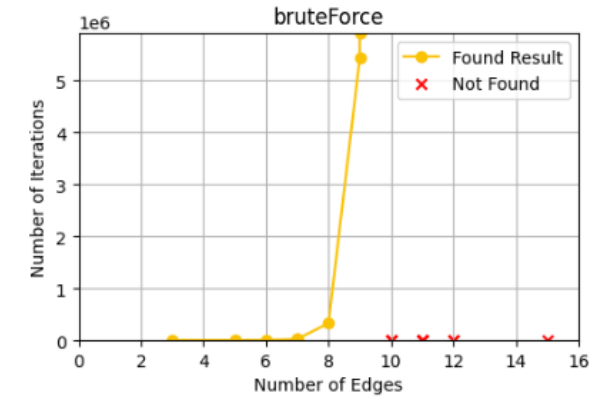
Uvod

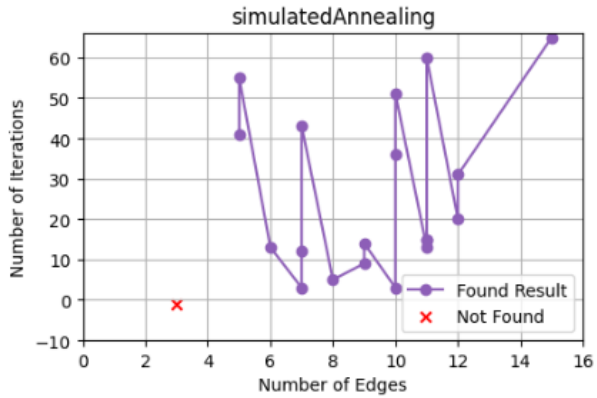
Kada se suočavamo s problemom minimalnog bojenja grafa, važno je razumeti različite pristupe i performanse algoritama. Ovo poređenje analizira nekoliko ključnih tačaka u kontekstu iteracija, vremena izvršavanja i broja boja u rešenju.

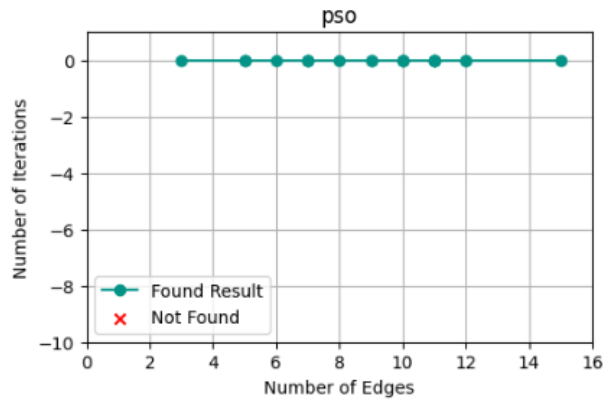
Iteracije naspram broja grana

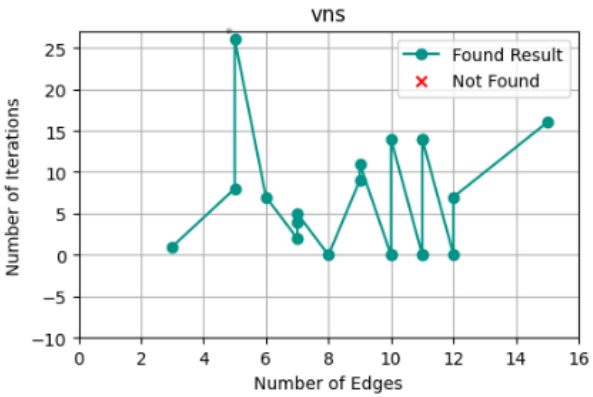
- **Brute Force** - precizan ali neefikasan za veće grafove
- **Simulirano Kaljenje** - fleksibilan, ali varira u optimalnosti rešenja
- **Variable Neighborhood Search (VNS)** - prilagodljiv, s varijacijama u performansama
- **Genetski Algoritmi (GA)** i **Optimizacija rojem čestica (PSO)** - stabilni i efikasni
- **Kolonija mrava (ACO)** - sporiji za veće grafove, ali koristan za srednje velike grafove

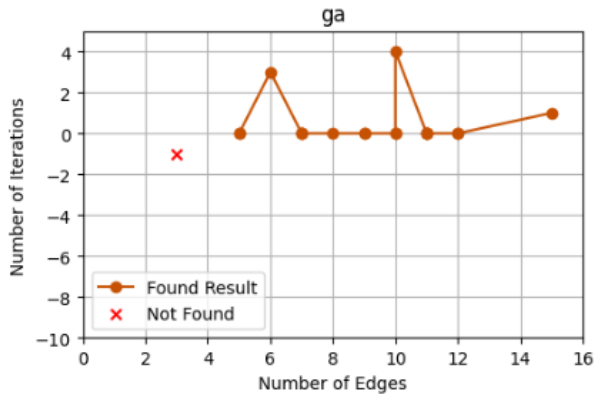
Iteracije naspram broja grana (nastavak)

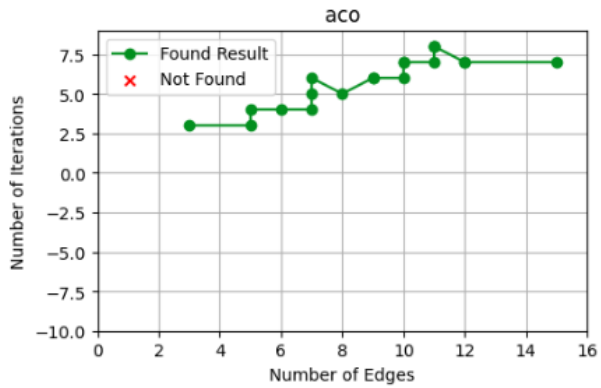








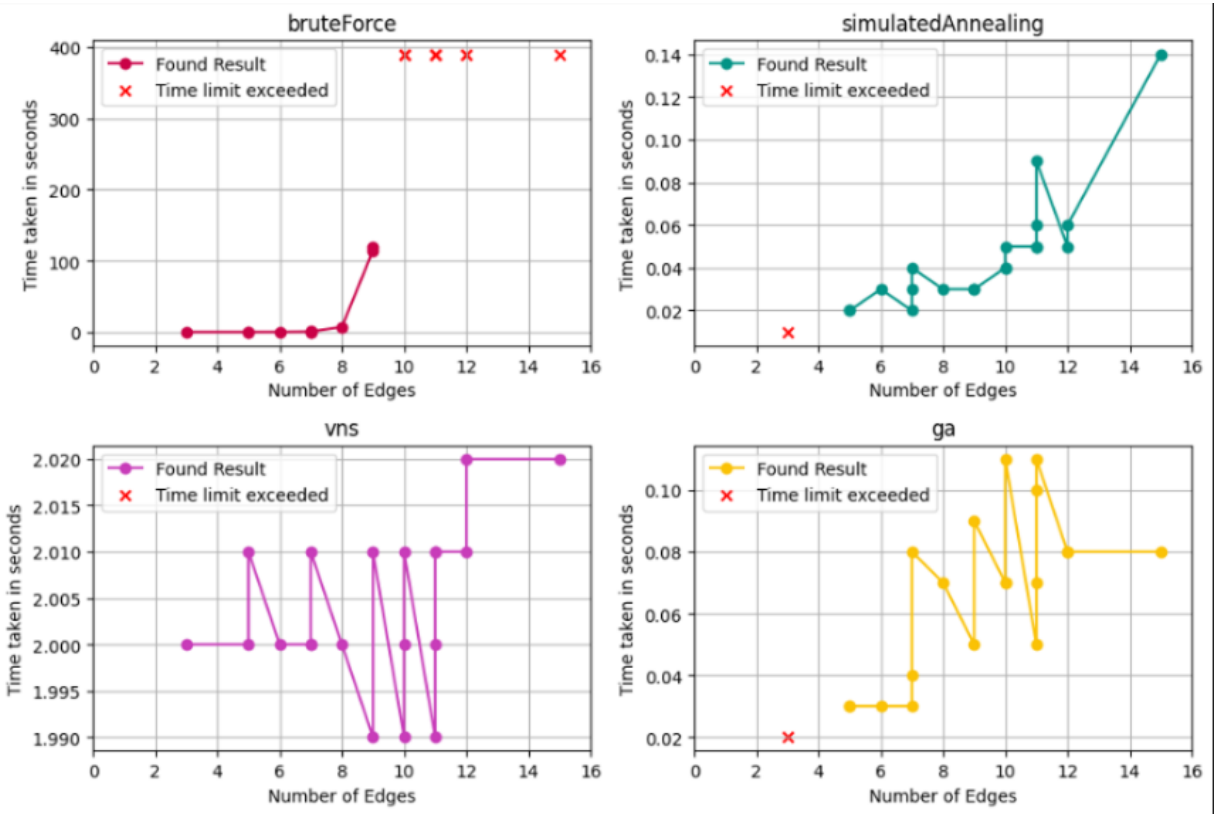


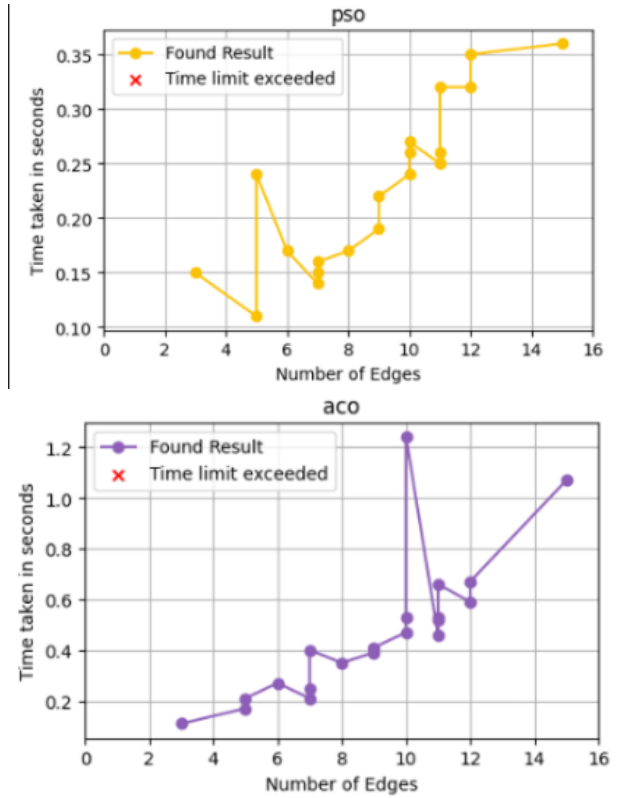


Potrebno vreme u odnosu na broj grana

- **Brute force** - sporo rastuće vreme izvršavanja za veće grafove
- **Simulirano Kaljenje** - postepeno rastuće vreme izvršavanja
- **VNS** - velike varijacije, ali često optimalni rezultati za srednje velike grafove
- **Genetski Algoritmi** - stabilni i efikasni za različite veličine grafova
- **PSO** i **ACO** - slični obrasci rasta u vremenu izvršavanja

Potrebno vreme u odnosu na broj grana (nastavak)

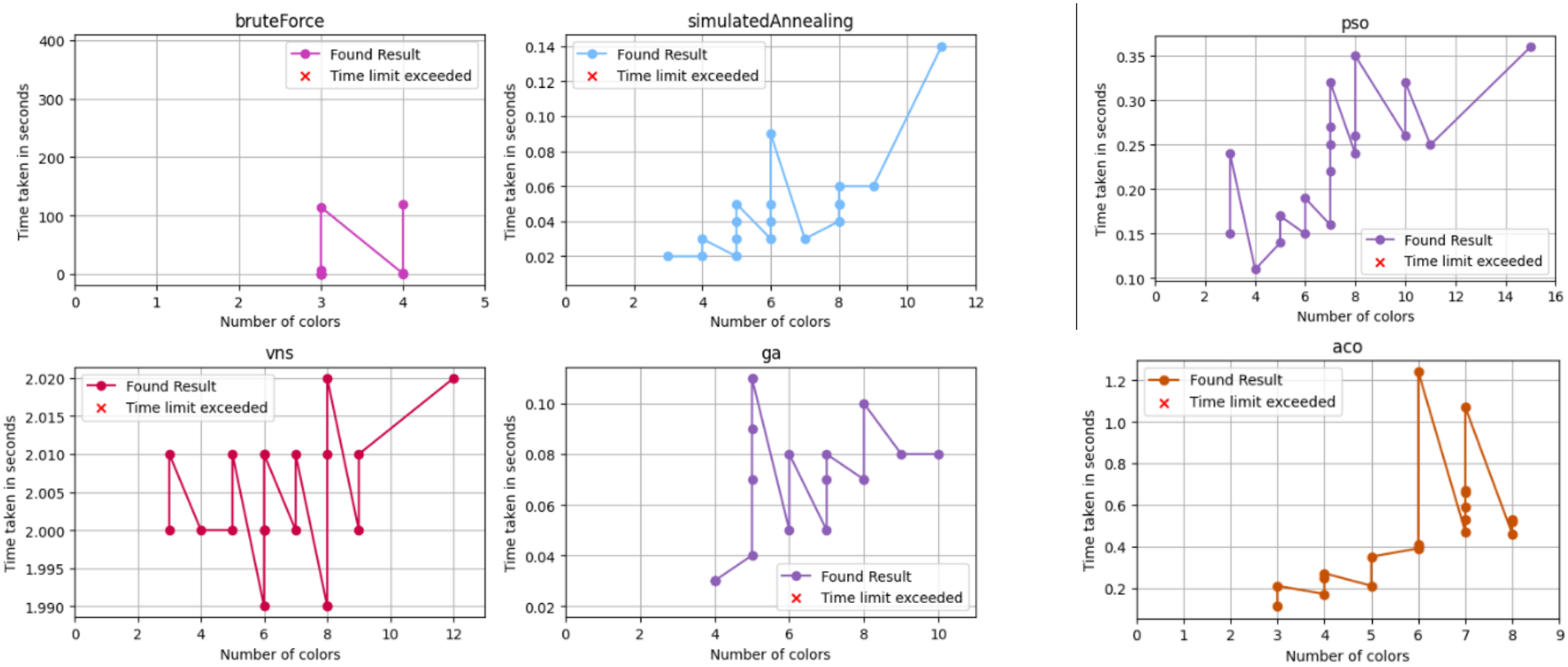




Broj boja naspram potrebnog vremena za izvršavanje

- **Brute Force** - najoptimalniji ali neefikasan za veće grafove
- **Simulirano Kaljenje** - dobro balansiranje između brzine i kvaliteta rešenja
- **VNS** - varira u broju boja u zavisnosti od grafa
- **Genetski Algoritmi** - stabilni i često sa malim brojem boja
- **PSO** - tendencija povećanja broja boja sa povećanjem grana
- **ACO** - brz za manje grafove, prosečan za veće

Broj boja naspram potrebnog vremena za izvršavanje (nastavak)



Zaključak poređenja

Izbor odgovarajućeg algoritma za minimalno bojenje grafova zahteva balansiranje između preciznosti, vremena izvršavanja i resursnih zahteva. Svaki algoritam ima svoje prednosti i mane, te je važno prilagoditi izbor algoritma specifičnim zahtevima problema.

Praktična primena

Bojenje grana grafa ima široku praktičnu primenu, uključujući:

- Raspoređivanje round-robin turnira
- Open shop raspoređivanje u proizvodnji
- Bojenje putanja u komunikacijama putem optičkih vlakana

Raspoređivanje turnira

- Bojenje grana potpunih grafova za raspoređivanje round-robin turnira
- Vrhovi grafa su takmičari, grane su utakmice, boje su runde
- Slično se može primeniti i za raspoređivanje drugih sportskih parova

Open Shop Raspoređivanje

- Open shop organizacija proizvodnje omogućava fleksibilnost u izboru redosleda operacija
- Bojenje grana bipartitnog multigrafa za organizaciju operacija na mašinama
- Efikasno korišćenje resursa za minimiziranje vremena proizvodnje

Komunikacije putem optičkih vlakana

- Bojenje putanja u komunikacionim mrežama
- Problemi bojenja putanja u zvezdanim mrežama i druge topologije
- Održavanje frekvencijske nezavisnosti u optičkim mrežama

Zaključak

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


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

Hvala na pažnji! :))