Minimum Precedence Constrained Sequencing With Delays

Matematički Fakultet Univerziteta u Beogradu

Relja Pešić 73/2019 Pavle Dušanić 287/2019

September 1, 2024

Pregled tema

Definicija problema

Rasporediti zadatake tako da se poštuje odredjeni redosled kojm se zadaci izvršavaju i kašnjenja izmedju istih, a da se pritom vreme završetka svih zadataka minimizuje

Definicija problema

- $G = (T, E)$ - usmereni aciklični graf koji definiše redosled izvršavanja zadataka koji rešenje mora ispoštovati
- $d(t)$ - kašnjenje definisano za svaki zadatak $t \in T$ kao pozitivan ceo broj $0 \leq d(t) \leq D$
- $S : T \to \mathbb{Z}^+$ - injektivna funkcija za koju važi naredni uslov: $S(t_j) - S(t_i) > d(t_i)$ za svaku granu $(t_i, t_j) \in E$
- $\max\limits_{t \in T} S(t)$ - vreme završetka svih poslova

Rešenje algoritmom grube sile

- Iscrpna pretraga svih mogućih permutacija rasporeda
- Dobro rešenje za male grafove

```
def is_valid_schedule(schedule, edges):
    task_to_index = {task: index for index, task in enumerate(schedule)}
    for u, v in edges:
        if task_to_index[u] > task_to_index[v]:
            return False
    return True
```

Figure: Provera valjanosti rasporeda

```python
def calculate_S(permutation, graph, delay, predak):
    S = {t: 0 for t in permutation}
    for node in permutation:
        max_S = S[node]
        for pred in predak[node]:
            max_S = max(S[node], S[pred] + delay[pred] + 1)

        #ako postoji node sa istim S, uvecamo ga za 1
        while max_S in S.values():
            max_S += 1
        S[node] = max_S

    return S, max(S.values())
```

Figure: Računanje vremena završetka posla

```
def brute_force_alg(tasks, edges, delays, max_seconds):
    start_time = time.time()
    signal.signal(signal.SIGALRM, timeout_handler)
    signal.alarm(max_seconds)

    try:
        graph, predak = inicialize_graph(edges)
        min_S = float('inf')
        for permutation in schedule_permutations(tasks):
            if is_valid_schedule(permutation, edges):
                S, maximum_S = calculate_S(permutation, graph, delays, predak)
                min_S = min(maximum_S, min_S)
                best_permutation = permutation

        end_time = time.time()
        time_taken = end_time - start_time
        print("Best order of tasks:", best_permutation)
        print("S:", S)
        print("Minimal S:", min_S)
        print("Time taken to find the solution:", time_taken)
        return best_permutation, min_S, time_taken
    except TimeoutException as e:
        return str(e)
    finally:
        signal.alarm(0)
```

Figure: Algoritam grube sile

Genetski algoritam

- Proces započinje populacijom nasumično generisanih rešenja.
- Jedinku karakterišu fitness i schedule
- Bolja rešenja imaju veću šansu da se pojave u narednim generacijama i kreiraju nova
- Ponekad nova rešenja mutiraju, čime se menja njihov kvalitet.
- Proces koji obuhvata sve ove operacije se ponavlja kroz više generacija

```
class Individual:
    def __init__(self, schedule, edges, delay):
        self.schedule = schedule
        self.fitness = self.calc_fitness(edges, delay)

    #ako ne zadovoljava topsort fitness->inf inace izracunaj max(S)
    def calc_fitness(self, edges, delay):
        if not is_valid_schedule(self.schedule, edges):
            return float('-inf')
        graph, predak = inicialize_graph(edges)
        return -calculate_S(self.schedule, graph, delay, predak)[1]

    def __lt__(self, other):
        return self.fitness < other.fitness
```

Figure: Način predstavljanja rešenja

```python
def create_initial_population(size_of_population, tasks, edges, delays):
    population = []
    selected_permutations = list(islice(permutations(tasks), size_of_population))
    random.shuffle(selected_permutations)

    for schedule in selected_permutations:
        individual = Individual(list(schedule), edges, delays)
        population.append(individual)

    return population
```

Figure: Generisanje inicijalne populacije

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

Figure: Selekcija

Ukrštanje jedinki

- Ukrštanje prvog reda
- Partially mapped crossover
- Cycle crossover

```python
def crossover(parent1, parent2):
    idx1, idx2 = sorted(random.sample(range(len(parent1.schedule)), 2))
    child = [None] * len(parent1.schedule)
    child[idx1:idx2+1] = parent1.schedule[idx1:idx2+1]

    current_pos = 0
    for task in parent2.schedule:
        if task not in child:
            while child[current_pos] is not None:
                current_pos += 1
            child[current_pos] = task
    return child
```

Figure: Implementacija ukrštanja prvog reda

Mutacija jedinki

- Mutacija zasnovana na zameni
- Mutacija zasnovana na inverziji
- Mutacija zasnovana na mešanju
- Višestruka primena mutacije zasnovane na zameni
- Kombinacija mutacije zasnovane na zameni i inverziji

```python
def mutate(individual):
    idx1, idx2 = random.sample(range(len(individual)), 2)
    individual[idx1], individual[idx2] = individual[idx2], individual[idx1]
```

Figure: Implementacija mutacije zasnovane na zameni

```python
def genetic_algorithm(mutation_func, crossover_func, population_size, num_generations,
                      tournament_size, elitism_size, mutation_prob,
                      tasks, edges, delays):
    population = create_initial_population(population_size, tasks, edges, delays)
    new_population = []
    start_time = time.time()

    for i in range(num_generations):
        population = sorted(population, reverse=True)
        new_population[:elitism_size] = population[:elitism_size]
        for j in range(elitism_size, population_size):
            parent1 = selection(population, tournament_size)#nz dal se razlikuju
            parent2 = selection(population, tournament_size)

            child1, child2 = [], []
            if crossover_func.__name__ == 'crossover':
                child1 = crossover_func(parent1, parent2)
                child2 = crossover_func(parent1, parent2)
            else:
                child1, child2 = crossover_func(parent1, parent2)

            if random.random() < mutation_prob:
                mutation_func(child1)
            if random.random() < mutation_prob:
                mutation_func(child2)

            new_population.append(Individual(child1, edges, delays))
            new_population.append(Individual(child2, edges, delays))

        population = new_population.copy()

    end_time = time.time()
    time_taken = end_time - start_time
    best_individual = max(population)
    print(f'solution: {best_individual.schedule}, cost: {-best_individual.fitness}, time taken: {time_taken}')
    return best_individual.schedule, -best_individual.fitness, time_taken
```

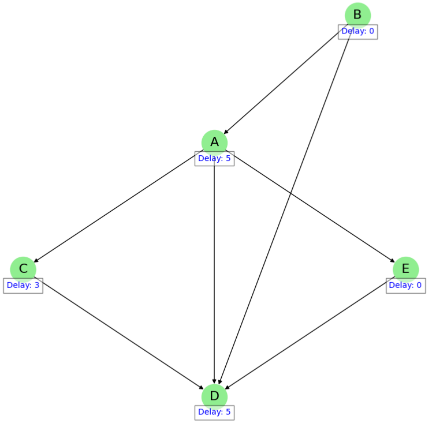

Figure: Primer manjeg grafa na kom su testirani algoritmi

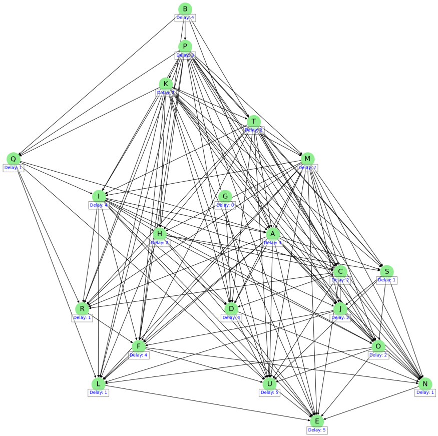

Figure: Primer većeg grafa na kom su testirani algoritmi

Kombinacija genetskog algoritma i simuliranog kaljenja

- Sekvencijalna primena genetskog algoritma, potom simuliranog kaljenja
- Sekvencijalna primena simuliranog kaljenja, potom genetskog algoritma
- Paralelno izvršavanje oba algoritma

```
def GASA(population_size, num_generations, tournament_size, elitism_size, mutation_prob, tasks, edges, delays):
    start_time = time.time()

    best_individual = genetic_algorithm(
                                        population_size=40,
                                        num_generations=40,
                                        tournament_size=9,
                                        elitism_size=7,
                                        mutation_prob=0.5,
                                        tasks=tasks,
                                        edges=edges,
                                        delays=delays
                                        )
    best_individual = simulatedAnnealing(best_individual, edges, delays)

    end_time = time.time()
    time_taken = end_time - start_time

    print(f'solution: {best_individual.schedule}, cost: {-best_individual.fitness}, time taken: {time_taken}')
    return best_individual.schedule, -best_individual.fitness, time_taken
```

Figure: Sekvencijalna primena genetskog algoritma, potom simuliranog kaljenja

Rezultati i zaključak

- Svi algoritmi pronalaze optimalno rešenje za male grafove
- Vreme izvršavanja algoritma grube sile raste srazmerno faktorijelu broja čvorova
- Vreme izvršavanja ostalih algorirama ne zavisi od broja čvorova
- Grafovski prikaz rezultata

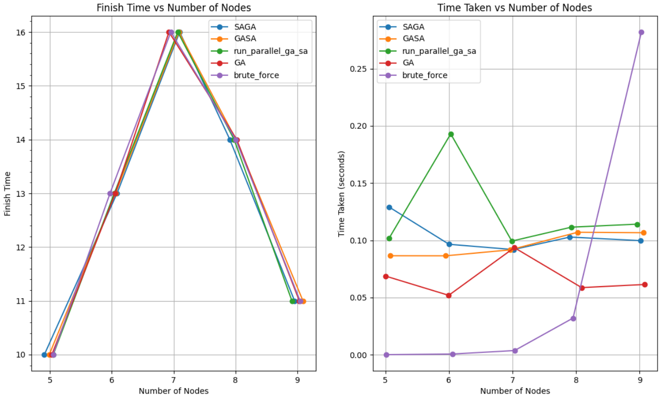

Figure: Rezultati algoritama primenjenih nad malim podacima

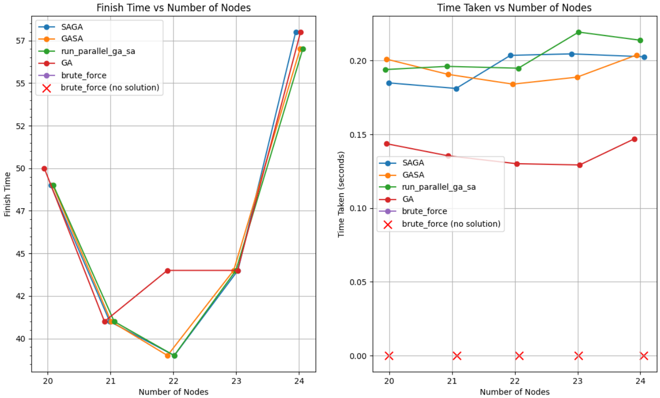

Figure: Rezultati algoritama primenjenih nad velikim podacima

Hvala na pažnji!