Minimum Precedence Constrained Sequencing With Delays

Matematički fakultet Univerziteta u Beogradu

Relja Pešić 73/2019
Pavle Dušanić 287/2019

August 2023

Sadržaj

Uvod

Ovaj rad se bavi optimizacijom rešenja problema: 'Minimum precedence constrained sequencing with delays' primenom različitih metaheuristika. Dalji tekst obuhvata detaljnan opis pristupa rešavanju problema, od inicijalne ideje do finalnih rešenja.

1.1 Definicija problema

Problem 'Minimum Precedence Constrained Sequencing with Delays' se bavi rasporedjivanjem zadataka, tako da se poštuje odredjeni redosled kojm se zadaci izvršavaju i kašnjenja izmedju istih. Cilj je minimizovati ukupno vreme koje je potrebno da se svi zadaci obave.

Skup zadataka je predstavljen kao usmeren aciklični graf, gde ivice grafova uslovljavaju redosled kojim se zadaci izvršavaju. Rešenje predstavlja raspored zadataka koji poštuje ova ograničenja i minimalizuje vreme završetka svih zadataka.

U nastavku koristimo sledeće oznake:

- T - skup zadataka
- E - skup grana izmedju zadataka
- $G = (T, E)$ - 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 $0 \leq d(t) \leq D$ gde D predstavlja gornje ograničenje
- $S : T \to \mathbb{Z}^+$ - injektivna funkcija za koju važi naredni uslov: $S(t_j) - S(t_i) > d(t_i)$ za svaku granu $(t_i, t_j) \in E$
- $\max_{t \in T} S(t)$ - vreme završetka svih poslova, koje je potrebno minimalizovati

Rešenje algoritmom grube sile

Kako je S zadato već navedenom formulom (1.1), sledećim izvodjenjem dobijamo:

$$S(\text{suc}) - S(\text{pre}) > d(\text{pre})$$
$$S(\text{suc}) > S(\text{pre}) + d(\text{pre})$$
$$S(\text{suc}) \geq S(\text{pre}) + d(\text{pre}) + 1$$

Gde su suc i pre oznake za čvorove naslednika i prethodnika.
Algoritam grube sile prolazi kroz sve permutacije rasporeda zadataka, proverava da li je zadovoljen kriterijum uslovljenosti i potom računa S za svaki čvor u rasporedu inkrementalno. Ukoliko dva čvora imaju istu vrednost za S, čvoru kome je kasnije dodeljen redosled izvršavanja se dodeljuje vrednost veća za 1.

```
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 2.1: Provera uredjenosti zadataka

```
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 2.2: Računanje vremena završetka posla

Najveći izazov ovakvom rešenju predstavljaju grafovi većih dimenzija, prilikom čije obrade se algoritam ne završava. Glavna ideja u optimizaciji ovog rešenja metaheuristikama je da se izbegne pretraga rešenja svih mogućih permutacija rasporeda, čime se dobija na efikasnosti.

```python
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 2.3: Algoritam grube sile

Genetski algoritam

Genetski algoritam je metoda optimizacije inspirisana prirodnom evolucijom. Proces započinje populacijom nasumično generisanih rešenja. Svako od tih rešenja ima svoju vrednost prilagodjenosti (fitness), koja pokazuje kvalitet jedinke. 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. Algoritam se zaustavlja nakon odredjenog broja generacija.

Provera uredjenosti zadataka kao i računanje vremena njihovog završetka S ostaju identične kao u prethodnom rešenju. Rešenja predstavljamo klasom Individual koja ima polja schedule (raspored zadataka) i fitness (merilo kvaliteta jedinke, obrnuto proporcijalno vremenu završetka zadataka S). Inicijalizacija početne populacije realizovana je generisanjem permutacija do željene veličine populacije. Za selekciju je korišćena turnirska selekcija dok je za crossover i mutation funkcije eksperimentisano sa nekoliko različitih metoda.

```python
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 3.1: 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 3.2: Generisanje inicijalne populacije

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

Figure 3.3: Selekcija

3.1 Crossover

Proverene metode ukrštanja jedinki:

- Ukrštanje prvog reda - nasumično se odedi segment iz prvog roditelja koji će se prekopirati u dete, nepopunjena mesta u rasporedu se dopunjuju elementima drugog roditelja, ukoliko se već ne nalaze u rasporedu
- Partially mapped crossover - nasumično se odredi segment koji će potomci naslediti, a ostali geni se preslikavaju odgovarajućim mapiranjem
- Cycle crossover - popunjava zadatke u rasporedu potomaka uspostavljanjem ciklusa koji formiraju rasporedi predaka

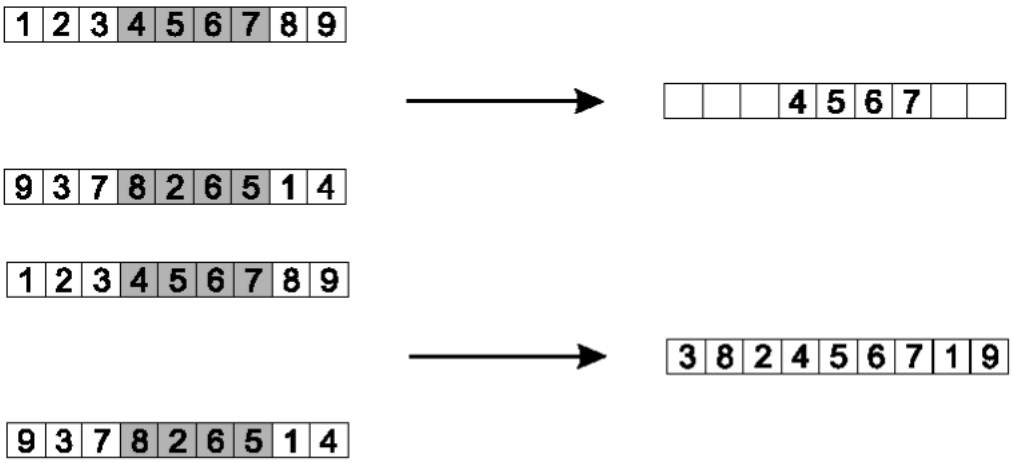

Figure 3.4: Demonstracija rada ukrštanja prvog reda - preuzeto sa slajdova prof. Vladimira Filipovića

```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 3.5: Implementacija ukrštanja prvog reda

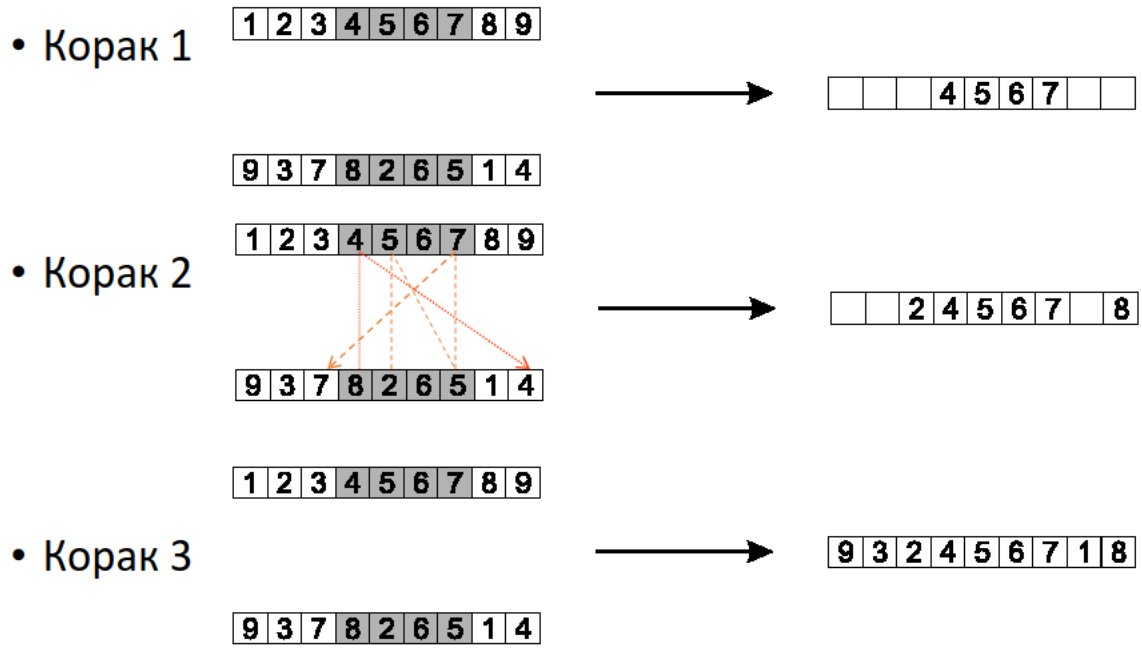

Figure 3.6: Demonstracija rada PMX algoritma - preuzeto sa slajdova prof. Vladimira Filipovića

3.2 Mutation

Proverene metode mutacije jedinki:

- Mutacija zasnovana na zameni - nasumično se biraju dva zadatka iz rasporeda, potom im se zamene mesta
- Mutacija zasnovana na inverziji - nasumično se odredi segment u rasporedu, potom se obrne redosled tih zadataka
- Mutacija zasnovana na mešanju - nasumično se odredi segment u rasporedu, potom se poslovi iz segmenta nasumično rasporede unutar segmenta
- Višestruka primena mutacije zasnovane na zameni
- Kombinacija mutacije zasnovane na zameni i inverziji

3.3 Odabir parametara

Algoritam je testiran kako za različite vrednosti parametara tako i za različite kombinacije funkcija ukrštanja i mutacije. Glavni uticaj na brzinu izvršavanja algoritma imali su parametri veličine populacije i broj generacija, dok su se kao najbolji izbor pokazale mutacija zasnovana na zameni i krštanje prvog reda. Bolje rezultate ostvarili su algoritmi sa manjom vrednosti veličine populacije i brojem generacija (izmedju 40 i 50) u poredjenju sa onim koji su imali vrednosti od 100 do 500 (Pretpostavka je da su algoritmi ostvarivali iste rezultate, ali je razlika u vremenu izvršavanja bila značajna), dok je za parametre koji odredjuju vrednosti elitizma i verovatnoće da dodje do permutacije važilo obrnuto. Validno rešenje je poželjno preneti u narednu generaciju, jer se iz njega mogu dobiti potencijalo bolja rešenja. Mutacija doprinosi većoj raznovrsti rešenja, što povećava šanse da se pronadje validno rešenje.

Za dalje unapredjenje rešenja fiksirani su sledeći parametri:

- veličina populacije = 40

```python
def partially_mapped_crossover(parent1, parent2):
    size = len(parent1.schedule)
    start, end = sorted(np.random.choice(size, 2, replace=False))

    offspring1 = [None] * size
    offspring2 = [None] * size

    offspring1[start:end+1] = parent1.schedule[start:end+1]
    offspring2[start:end+1] = parent2.schedule[start:end+1]

    mapping1 = {parent1.schedule[i]: parent2.schedule[i] for i in range(start, end+1)}
    mapping2 = {parent2.schedule[i]: parent1.schedule[i] for i in range(start, end+1)}

    def fill_offspring(offspring, mapping, parent):
        for i in range(size):
            if offspring[i] is None:
                gene = parent.schedule[i]
                while gene in mapping:
                    gene = mapping[gene]
                offspring[i] = gene

    fill_offspring(offspring1, mapping1, parent2)
    fill_offspring(offspring2, mapping2, parent1)

    return offspring1, offspring2
```

Figure 3.7: Implementacija PMX

- broj generacija = 40
- veličina turnira za turnirsku selekciju = 9
- elitism - broj najboljih jedinki iz prethodne generacije = 7
- verovatnoća da dodje do mutacije genoma = 0.5

3.4 Možemo li bolje?

Iako se algoritam uspešno završava prilikom obrade velikih grafova, ne dobijaju se uvek optimalna rešenja (što će biti pokazano u nastavku). Deluje da genetski algoritam nije otporan na lokalne minimume u pretrazi rešenja.

```python
def cycle_crossover(parent1, parent2):
    size = len(parent1.schedule)
    offspring1 = [None] * size
    offspring2 = [None] * size
    visited = [False] * size

    def find_cycle(start):
        cycle = []
        while not visited[start]:
            visited[start] = True
            cycle.append(start)
            start = parent1.schedule.index(parent2.schedule[start])
        return cycle

    for i in range(size):
        if not visited[i]:
            cycle = find_cycle(i)
            for pos in cycle:
                offspring1[pos] = parent1.schedule[pos]
                offspring2[pos] = parent2.schedule[pos]

    # Fill remaining genes
    for i in range(size):
        if offspring1[i] is None:
            offspring1[i] = parent2.schedule[i]
        if offspring2[i] is None:
            offspring2[i] = parent1.schedule[i]

    return offspring1, offspring2
```

Figure 3.8: Implementacija cycle crossover funkcije

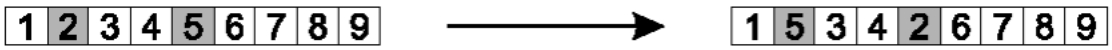

Figure 3.9: Demonstracija rada mutacije zasnovane na zameni - preuzeto sa slajdova prof. Vladimira Filipovića

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

Figure 3.10: Implementacija mutacije zasnovane na zameni

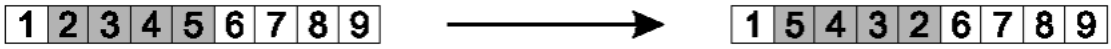

Figure 3.11: Demonstracija rada mutacije zasnovane na inverziji - preuzeto sa slajdova prof. Vladimira Filipovića

```python
def inversion_mutation(individual):
    size = len(individual)
    idx1, idx2 = np.sort(np.random.choice(size, 2, replace=False))

    individual[idx1:idx2+1] = individual[idx1:idx2+1][::-1]
```

Figure 3.12: Implementacija mutacije zasnovane na inverziji

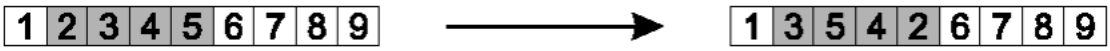

Figure 3.13: Demonstracija rada mutacije zasnovane na mešanju - preuzeto sa slajdova prof. Vladimira Filipovića

```python
def scramble_mutation(individual):
    size = len(individual)
    idx1, idx2 = np.sort(np.random.choice(size, 2, replace=False))

    subset = individual[idx1:idx2+1]
    np.random.shuffle(subset)
    individual[idx1:idx2+1] = subset
```

Figure 3.14: Implementacija mutacije zasnovane na mešanju

```python
def mutate_number_of_times(individual):
    num_swaps=2
    for _ in range(num_swaps):
        mutate(individual)
```

Figure 3.15: Implementacija višestruke mutacije

```python
def swap_and_invert_mutation(individual):
    mutate(individual)
    inversion_mutation(individual)
```

Figure 3.16: Implementacija mutacije zasnovane na zameni i inverziji

```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 3.17: Implementacija genetskog algoritma

```
Total combinations: 243
[{'population_size': 50, 'num_generations': 50, 'tournament_size': 3, 'elitism_size': 1, 'mutation_prob': 0.1}, {'population_size': 50, 'num_
generations': 50, 'tournament_size': 3, 'elitism_size': 1, 'mutation_prob': 0.2}, {'population_size': 50, 'num_generations': 50, 'tournament_
size': 3, 'elitism_size': 1, 'mutation_prob': 0.3}, {'population_size': 50, 'num_generations': 50, 'tournament_size': 3, 'elitism_size': 3, '
mutation_prob': 0.1}, {'population_size': 50, 'num_generations': 50, 'tournament_size': 3, 'elitism_size': 3, 'mutation_prob': 0.2}]
--------------------------
file:  test_file_22.json
Best Result: {'mutation_func': 'scramble_mutation', 'crossover_func': 'cycle_crossover', 'population_size': 50, 'num_generations': 100, 'tour
nament_size': 7, 'elitism_size': 3, 'mutation_prob': 0.3, 'cost': 39, 'time_taken': 0.6319921016693115}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_22.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_24.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 50, 'num_generations': 50, 'tournament_size': 7, '
elitism_size': 3, 'mutation_prob': 0.3, 'cost': 57, 'time_taken': 0.28307342529296875}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_24.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_20.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 50, 'num_generations': 50, 'tournament_size': 7, '
elitism_size': 3, 'mutation_prob': 0.3, 'cost': 49, 'time_taken': 0.23117971420288086}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_20.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_21.json
Best Result: {'mutation_func': 'mutate_number_of_times', 'crossover_func': 'cycle_crossover', 'population_size': 50, 'num_generations': 50, '
tournament_size': 7, 'elitism_size': 5, 'mutation_prob': 0.3, 'cost': 41, 'time_taken': 0.2384166717529297}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_21.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_23.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 50, 'num_generations': 50, 'tournament_size': 7, '
elitism_size': 5, 'mutation_prob': 0.2, 'cost': 44, 'time_taken': 0.26622629165649414}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_23.json.json' mode='w' encoding='UTF-8'>
```

Figure 3.18: Odredjivanje parametara

```
Total combinations: 48
[{'population_size': 40, 'num_generations': 40, 'tournament_size': 7, 'elitism_size': 3, 'mutation_prob': 0.3}, {'population_size': 40, 'num_
generations': 40, 'tournament_size': 7, 'elitism_size': 3, 'mutation_prob': 0.5}, {'population_size': 40, 'num_generations': 40, 'tournament_
size': 7, 'elitism_size': 5, 'mutation_prob': 0.3}, {'population_size': 40, 'num_generations': 40, 'tournament_size': 7, 'elitism_size': 5, '
mutation_prob': 0.5}, {'population_size': 40, 'num_generations': 40, 'tournament_size': 7, 'elitism_size': 7, 'mutation_prob': 0.3}]
--------------------------
file:  test_file_22.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 40, 'num_generations': 40, 'tournament_size': 7, '
elitism_size': 3, 'mutation_prob': 0.5, 'cost': 42, 'time_taken': 0.13059067726135254}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_22.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_24.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 40, 'num_generations': 40, 'tournament_size': 10,
'elitism_size': 7, 'mutation_prob': 0.3, 'cost': 57, 'time_taken': 0.16292834281921387}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_24.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_20.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 50, 'num_generations': 40, 'tournament_size': 7, '
elitism_size': 5, 'mutation_prob': 0.5, 'cost': 49, 'time_taken': 0.14260554313659668}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_20.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_21.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 40, 'num_generations': 40, 'tournament_size': 7, '
elitism_size': 7, 'mutation_prob': 0.5, 'cost': 41, 'time_taken': 0.12469148635864258}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_21.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_23.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 40, 'num_generations': 40, 'tournament_size': 10,
'elitism_size': 3, 'mutation_prob': 0.5, 'cost': 44, 'time_taken': 0.14411377906799316}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_23.json.json' mode='w' encoding='UTF-8'>
```

Figure 3.19: Odredjivanje parametara

```
Total combinations: 48
[{'population_size': 15, 'num_generations': 15, 'tournament_size': 9, 'elitism_size': 9, 'mutation_prob': 0.5}, {'population_size': 15, 'num_
generations': 15, 'tournament_size': 9, 'elitism_size': 12, 'mutation_prob': 0.5}, {'population_size': 15, 'num_generations': 15, 'tournament
_size': 9, 'elitism_size': 15, 'mutation_prob': 0.5}, {'population_size': 15, 'num_generations': 20, 'tournament_size': 9, 'elitism_size': 9,
'mutation_prob': 0.5}, {'population_size': 15, 'num_generations': 20, 'tournament_size': 9, 'elitism_size': 12, 'mutation_prob': 0.5}]
--------------------------
file:  test_file_22.json
Best Result: {'mutation_func': 'scramble_mutation', 'crossover_func': 'crossover', 'population_size': 25, 'num_generations': 30, 'tournament_
size': 9, 'elitism_size': 15, 'mutation_prob': 0.5, 'cost': 40, 'time_taken': 0.037431955337524414}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_22.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_24.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 15, 'num_generations': 30, 'tournament_size': 9, '
elitism_size': 9, 'mutation_prob': 0.5, 'cost': 58, 'time_taken': 0.017486095428466797}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_24.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_20.json
Best Result: {'mutation_func': 'scramble_mutation', 'crossover_func': 'crossover', 'population_size': 20, 'num_generations': 20, 'tournament_
size': 9, 'elitism_size': 12, 'mutation_prob': 0.5, 'cost': 49, 'time_taken': 0.016620159149169922}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_20.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_21.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'cycle_crossover', 'population_size': 20, 'num_generations': 15, 'tournament_size
': 9, 'elitism_size': 15, 'mutation_prob': 0.5, 'cost': 41, 'time_taken': 0.0061273574829101562}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_21.json.json' mode='w' encoding='UTF-8'>
--------------------------
file:  test_file_23.json
Best Result: {'mutation_func': 'mutate', 'crossover_func': 'crossover', 'population_size': 30, 'num_generations': 30, 'tournament_size': 9, '
elitism_size': 9, 'mutation_prob': 0.5, 'cost': 44, 'time_taken': 0.05941033363342285}
Data saved to <_io.TextIOWrapper name='results/big_data/GA_fine_tuning/ga_results_test_file_23.json.json' mode='w' encoding='UTF-8'>
```

Figure 3.20: Odredjivanje parametara

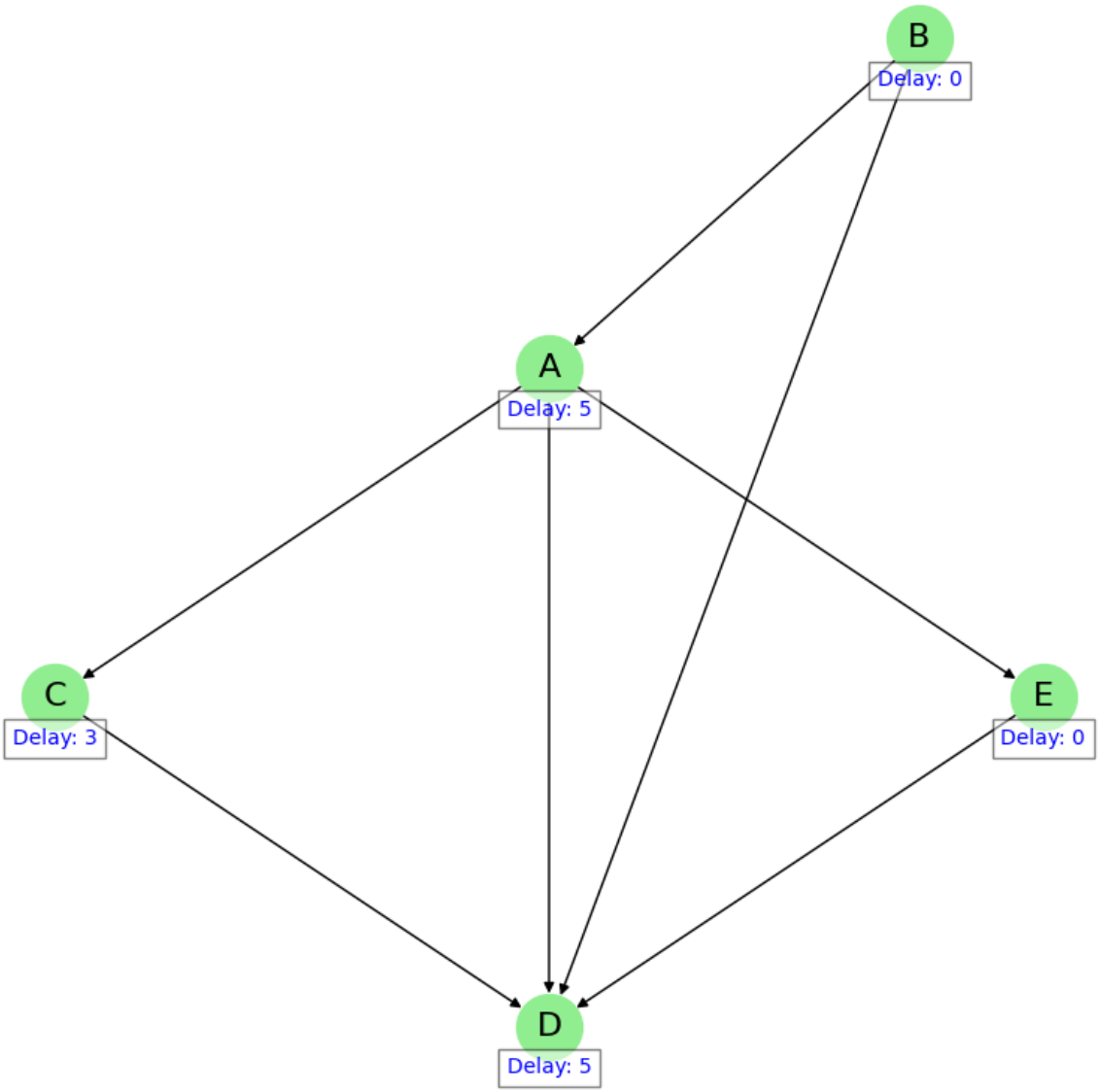

Figure 3.21: Primer manjeg grafa na kom su testirani algoritmi

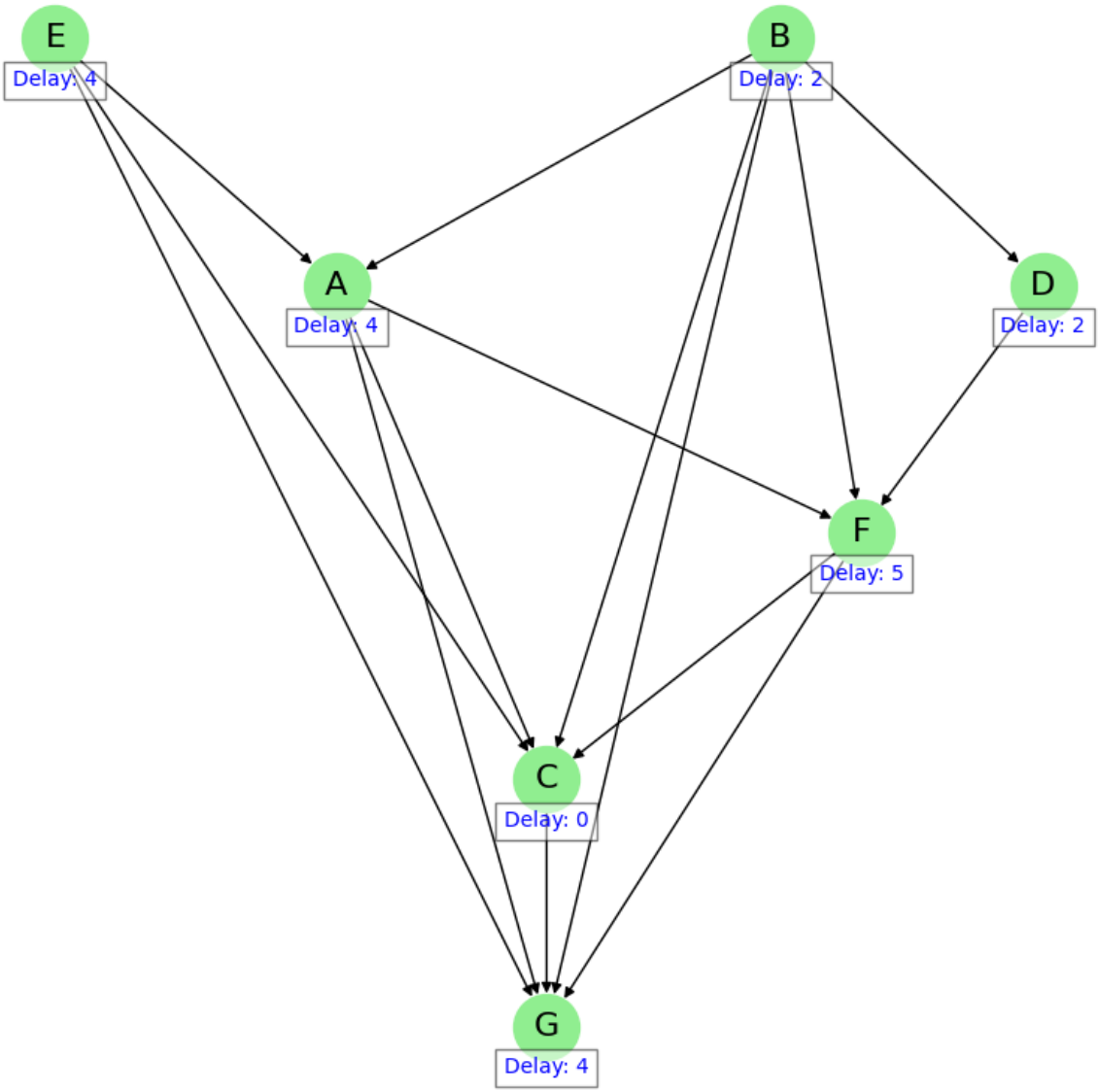

Figure 3.22: Primer manjeg grafa na kom su testirani algoritmi

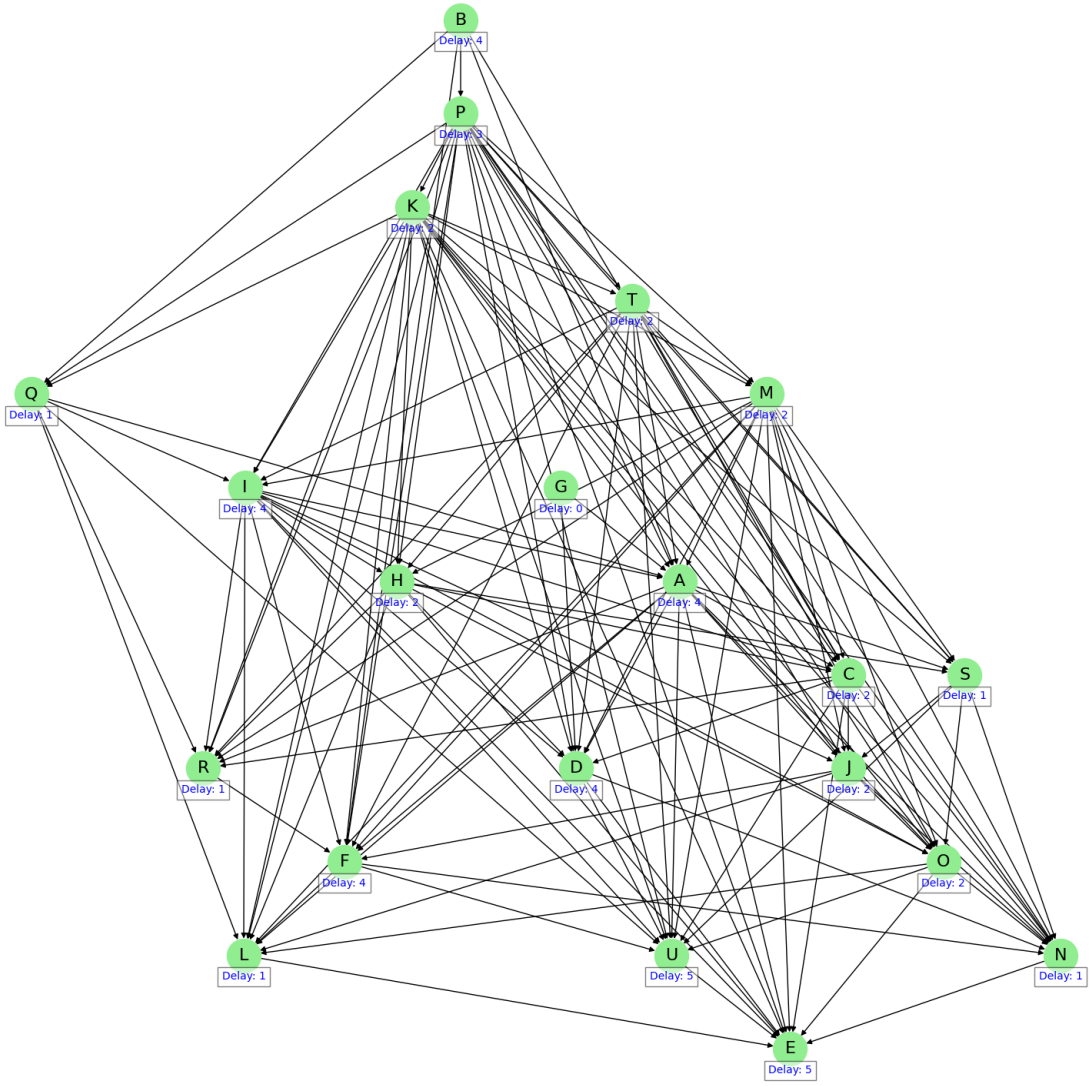

Figure 3.23: Primer većeg grafa na kom su testirani algoritmi

Kombinacija genetskog algoritma i simuliranog kaljenja

Simulirano kaljenje (Simulated Annealing) je heuristički algoritam za optimizaciju inspirisan procesom fizičkog kaljenja metala. Ovaj algoritam je koristan u slučajevima kada je prostor pretrage velik i kompleksan, jer može izbeći zaglavljivanje u lokalnim minimumima, što predstavlja jedan od razloga zbog kog je iskorišćen za unapredjenje prethodnog rešenja.

```python
def simulatedAnnealing(individual, edges, delays, iters=5000, initial_temp=1.0, alpha=0.99):
    best_individual = copy.deepcopy(individual)
    current_temp = initial_temp

    for i in range(1, iters + 1):
        original_schedule = copy.deepcopy(individual.schedule)
        original_fitness = individual.fitness

        # Perform a more substantial change (multi-step if necessary)
        idx1, idx2 = individual.invert()
        new_fitness = individual.calc_fitness(edges, delays)

        if new_fitness > individual.fitness:
            individual.fitness = new_fitness
        else:
            # Simulated annealing acceptance criteria
            delta_f = new_fitness - individual.fitness
            p = min(1.0, np.exp(delta_f / current_temp))
            q = random.uniform(0, 1)
            if p > q:
                individual.fitness = new_fitness
            else:
                # Revert to original schedule if the change isn't accepted
                individual.schedule = copy.deepcopy(original_schedule)
                individual.fitness = original_fitness

        # Update the best individual found
        if individual.fitness > best_individual.fitness:
            best_individual = copy.deepcopy(individual)

        # Gradually cool down
        current_temp *= alpha

    return best_individual
```

Figure 4.1: Implementacija algoritma simuliranog kaljenja

Algoritam simulirang kaljenja radi tako što se rešenje problema postupno unapredjuje kroz iteracije. U svakoj iteraciji, algoritam generiše novo rešenje koje može biti bolje ili lošije od prethodnog. Parametri temperatura i stepen hladjenja utiču na verovatnoću sa kojom se prihvata lošije rešenje. Ovaj pristup pomaže algoritmu da izbegne zaglavljenje u lokalnim minimumima time što neće uvek tražiti bolje rešenje u okolini trenutnog.

Ova dva algoritma iskombinovana su na tri sledeća načina:

- Sekvencijalna primena genetskog algoritma, potom simuliranog kaljenja. Zaključak iz prethodnog rešenja jeste da jedna permutacija dva elementa rasporeda stoji izmedju dovoljno dobrog rešenja i najboljeg rešenja, što jeste ono što ova metoda pokušava da ispravi. Genetski algoritam vrši globalnu pretragu rešenja dok simulirano kaljenje uzima to rešenje i unapredjuje ga.
- Sekvencijalna primena simuliranog kaljenja, potom genetskog algoritma. Ideja je slična prethodnoj, stim što se simulirano kaljenje koristi da bi se našla dovoljno dobra jedinka, čijom se mutacijom (nasumičnim permutacijama dva elementa iz rasporeda zadataka) generiše početna populacija za genetski algoritam. Oba algoritma postižu slične rezultate.
- Paralelno izvršavanje oba algoritma. Oba algoritma se pokreću paralelno, ali periodično obaveštavaju jedan drugog o do tada najboljem pronadjenom rešenju. Ovo rešenje, iako je malo sporije od prethodnih, pronalazi iste rasporede zadataka.

```python
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 4.2: Sekvencijalna primena genetskog algoritma, potom simuliranog kaljenja

```python
#treba proslediti najbolju individuu za generisanje populacije
def genetic_algorithm(best_individual, population_size, num_generations,
                      tournament_size, elitism_size, mutation_prob, tasks, edges, delays
                      ):
    def mutate_initial_individual(individual):
        new_ind = individual.copy()
        idx1, idx2 = random.sample(range(len(individual)), 2)
        new_ind[idx1], new_ind[idx2] = new_ind[idx2], new_ind[idx1]
        return new_ind

    population = [best_individual] + [Individual(mutate_initial_individual(best_individual.schedule), edges, delays)
                                      for _ in range(population_size - 1)]
    new_population = []

    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)

            child_schedule = crossover(parent1, parent2)

            if random.random() < mutation_prob:
                mutate(child_schedule)

            new_population.append(Individual(child_schedule, edges, delays))

        population = new_population.copy()

    best_individual = max(population)
    return best_individual
```

Figure 4.3: Modifikacija genetskog algoritma za rad SAGA algoritma

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

    #select a random individual
    individual = create_individual(tasks, edges, delays)
    after_SA = simulatedAnnealing(individual, edges, delays)

    #
    best_individual = genetic_algorithm(    after_SA,
                                            population_size=40,
                                            num_generations=40,
                                            tournament_size=9,
                                            elitism_size=7,
                                            mutation_prob=0.5,
                                            tasks=tasks,
                                            edges=edges,
                                            delays=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 4.4: Algoritam SAGA

```python
def genetic_algorithm(population_size, num_generations,
                      tournament_size, elitism_size, mutation_prob, tasks, edges, delays,
                      exchange_queue):
    population = create_initial_population(population_size, tasks, edges, delays)
    new_population = []

    for i in range(num_generations):
        population = sorted(population, reverse=True)
        new_population[:elitism_size] = population[:elitism_size]


        #calc the best individual
        best_individual = max(population)

        #exchange results with SA
        if i % 5 == 0:
            if not exchange_queue.empty():
                sa_solution = exchange_queue.get()
                if sa_solution.fitness > best_individual.fitness:
                    best_individual = copy.deepcopy(sa_solution)
            exchange_queue.put(best_individual)

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

            child_schedule = crossover(parent1, parent2)

            if random.random() < mutation_prob:
                mutate(child_schedule)

            new_population.append(Individual(child_schedule, edges, delays))

        population = new_population.copy()

    return best_individual
```

Figure 4.5: Modifikacija genetskog algoritma za rad algoritma paralelnog izvršavanja

```python
def simulated_annealing(individual, edges, delays, exchange_queue, iters=5000, initial_temp=1.0, alpha=0.99):
    best_individual = copy.deepcopy(individual)
    current_temp = initial_temp

    for i in range(1, iters + 1):
        original_schedule = copy.deepcopy(individual.schedule)
        original_fitness = individual.fitness

        # Perform a more substantial change (multi-step if necessary)
        idx1, idx2 = individual.invert()
        new_fitness = individual.calc_fitness(edges, delays)

        if new_fitness > individual.fitness:
            individual.fitness = new_fitness
        else:
            # Simulated annealing acceptance criteria
            delta_f = new_fitness - individual.fitness
            p = min(1.0, np.exp(delta_f / current_temp))
            q = random.uniform(0, 1)
            if p > q:
                individual.fitness = new_fitness
            else:
                # Revert to original schedule if the change isn't accepted
                individual.schedule = copy.deepcopy(original_schedule)
                individual.fitness = original_fitness

        # Update the best individual found
        if individual.fitness > best_individual.fitness:
            best_individual = copy.deepcopy(individual)

        # razmena sa GA\
        if random.random() < 0.1:
            if not exchange_queue.empty():
                ga_solution = exchange_queue.get()
                # ga_fitness = fitness_func(ga_solution)
                if ga_solution.fitness > best_individual.fitness:
                    best_individual = copy.deepcopy(ga_solution)
            exchange_queue.put(best_individual)

        # Gradually cool down
        current_temp *= alpha

    return best_individual
```

Figure 4.6: Modifikacija simuliranog kaljenja za rad algoritma paralelnog izvršavanja

```python
def run_parallel_ga_sa(tasks, edges, delays):
    exchange_queue = Queue()
    start_time = time.time()

    initial_solution = create_individual(tasks, edges, delays)

    # Set up the GA thread
    ga_thread = threading.Thread(target=genetic_algorithm, args=(
        40,
        40,
        9,
        7,
        0.5,
        tasks,
        edges,
        delays,
        exchange_queue
    ))

    # Set up the SA thread
    sa_thread = threading.Thread(target=simulated_annealing, args=(
        initial_solution,
        edges,
        delays,
        exchange_queue
    ))

    # Start both threads
    ga_thread.start()
    sa_thread.start()

    # Wait for both threads to complete
    ga_thread.join()
    sa_thread.join()

    # Get the final solutions from GA and SA
    if not exchange_queue.empty():
        best_individual = exchange_queue.get()

    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 4.7: Algoritam paralelnog izvršavanja oba algoritma

Rezultati i zaključak

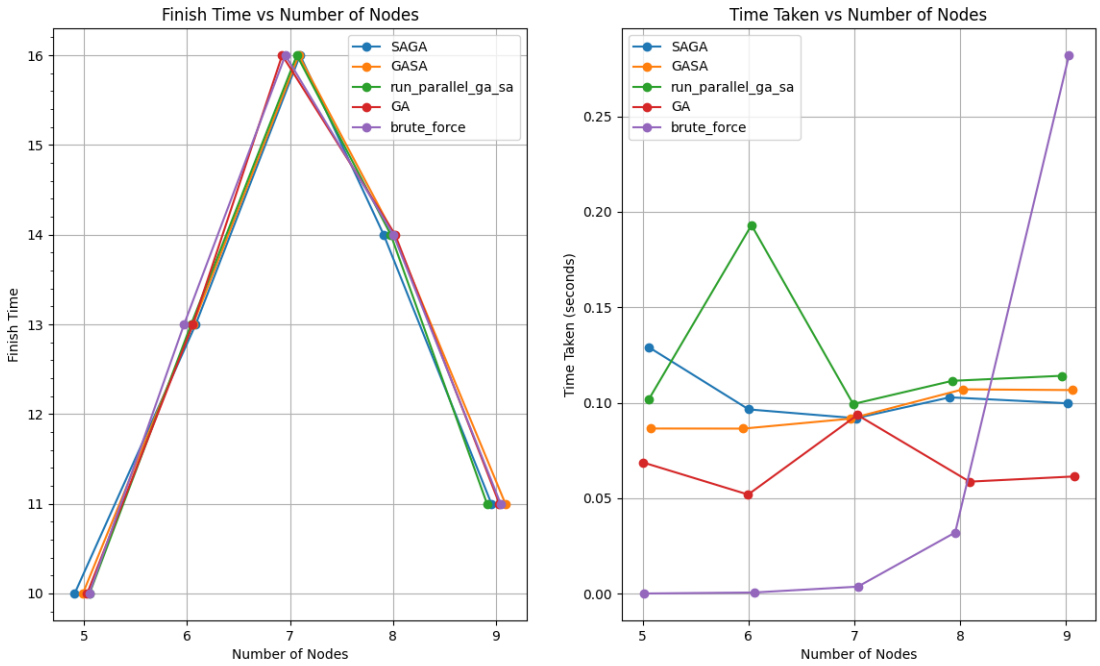

Figure 5.1: Rezultati algoritama primenjenih nad malim podacima

Za male grafove, svi algoritmi pronalaze optimalno rešenje. Iako najbrže rešava problem, vreme izvršavanja algoritma grube sile raste srazmerno faktorijelu broja čvorova. Ostali algoritmi imaju uniformno trajanje izvršavanja, koje je nezavisno od broja čvorova grafa. Na osnovu ovih grafovskih prikaza, možemo zaključiti da je metoda sekvencijalne primene genetskog algoritma i simuliranog kaljenja, dala najbolje rezultate.

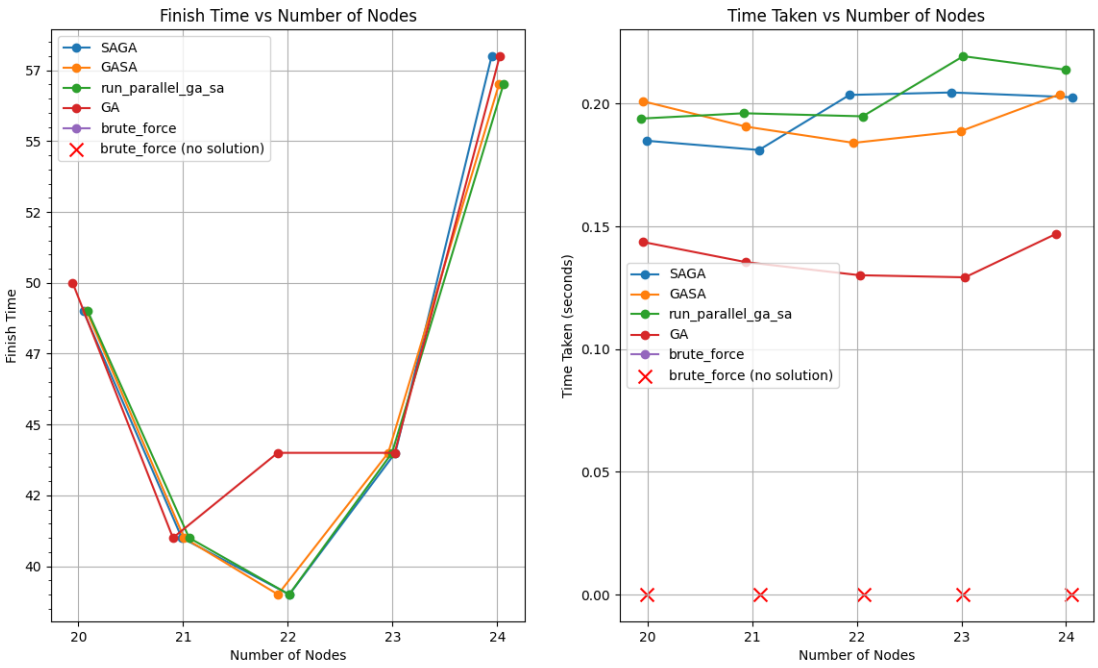

Figure 5.2: Rezultati algoritama primenjenih nad velikim podacima