Minimum Multiprocessor Scheduling

Računarska inteligencija
Matematički fakultet

Nenad Dobrosāvljević
mi20088@alas.matf.bg.ac.rs

Sažetak

U ovom dokumentu ću prikazati kako sam uspešno rešio izazov raspodela skupa zadataka na više procesora kao i koje sam strategije primenio kako bih optimizovao proces pronalaženja rešenja. Projekat možete pogledati ovde.

Sadržaj

1 Definicija problema

Problem Minimum Multiprocessor Scheduling predstavlja jedan od klasičnih problema u oblasti optimizacije i raspodele zadataka u računarstvu. Cilj ovog problema je raspodela skupa zadataka na više procesora kako bi se minimizovalo vreme završetka (makespan). Ovaj problem je NP-težak, što znači da optimalno rešenje može biti teško pronaći u razumnom vremenu za velike instance problema.

U ovom radu ću opisati i uporediti nekoliko pristupa za rešavanje ovog problema:

- Brute-force metod
- Local Search algoritam
- Simulated Annealing (SA)
- Genetski algoritam (GA)
- Optimizacija rojem čestica

2 Reprezentacija podataka u kodu

Ulazni podaci:

- Skup zadataka (T)
- Broj procesora (m)
- $f(t)$ je funkcija dodele koja kaže na koji procesor je zadatak t dodeljen
- $l(t,i)$ je vreme izvršavanja zadatka t na procesoru i.
- Vreme izvršavanja svakog zadatka na određenom procesoru (execution times)

Izlaz:

- Funkcija dodele koja zadatke dodeljuje procesorima

Cilj:

- Minimizovati vreme završetka:

$$\max_{i\in\{1,\dots,m\}} \sum_{t\in T, f(t)=i} l(t,i)$$

```
schedules = product(range(m), repeat=T)
optimal_schedule = None
min_completion_time = float('inf')
```

```
(0, 0, 0, 0)   (0, 1, 1, 0)
(0, 0, 0, 1)   (0, 1, 1, 1)
(0, 0, 1, 0)   (1, 0, 0, 0)
(0, 0, 1, 1)   (1, 0, 0, 1)
(0, 1, 0, 0)   (1, 0, 1, 0)
(0, 1, 0, 1)   (1, 0, 1, 1)
```

```
def calculate_completion_time(schedule, execution_times, m):
    processor_times = [0] * m
    for task, processor in enumerate(schedule):
        processor_times[processor] += execution_times[task][processor]
    return max(processor_times)
```

3 Primena grube sile

Brute force pristup u problemu minimalnog raspoređivanja zadataka na više procesora podrazumeva iscrpnu pretragu svih mogućih načina dodele zadataka dostupnim procesorima, kako bi se pronašao raspored koji daje minimalno ukupno vreme završetka (makespan). Ovaj pristup je teoretski moguć za manji broj zadataka i procesora, ali postaje neefikasan za veće ulaze zbog eksponencijalnog rasta broja mogućih rasporeda.

Za problem sa **T** zadataka i **m** procesora, broj mogućih rasporeda iznosi $\boldsymbol{m^T}$, jer se svaki zadatak može dodeliti bilo kom procesoru. Brute force algoritam proverava sve moguće kombinacije i računa ukupno vreme izvršavanja (maksimalno opterećenje među procesorima) za svaki raspored, kako bi pronašao onaj sa najmanjim makespan-om

```python
from itertools import product

T = 4  # number of tasks
m = 2  # number of processors

execution_times = [
    [3, 2],
    [2, 1],
    [4, 3],
    [1, 2],
]

def calculate_completion_time(schedule, execution_times, m):
    processor_times = [0] * m
    for task, processor in enumerate(schedule):
        processor_times[processor] += execution_times[task][processor]
    return max(processor_times)

schedules = product(range(m), repeat=T)
optimal_schedule = None
min_completion_time = float('inf')

for schedule in schedules:
    completion_time = calculate_completion_time(schedule, execution_times, m)
    if completion_time < min_completion_time:
        min_completion_time = completion_time
        optimal_schedule = schedule

optimal_schedule, min_completion_time
```

Slika 4: Kod funkcije brute force

Funkcija product iz modula itertools generiše sve moguće kombinacije dodele zadataka procesorima, pri čemu se svaki zadatak može izvršavati na bilo kojem od dostupnih procesora.
Iako ova metoda garantuje pronalazak optimalnog rasporeda, zbog eksponencijalnog rasta prostora pretrage vreme izvršavanja naglo raste sa povećanjem broja zadataka i procesora, što brute force čini nepraktičnim za veće instance problema.

4 Optimizacije

4.1 Lokalna pretraga

Lokalna pretraga (*Local Search*) je heuristička metoda optimizacije koja polazi od jednog početnog rešenja i pokušava da ga poboljša postupnim malim promenama. U kontekstu problema **minimalnog raspoređivanja zadataka na više procesora**, lokalna pretraga funkcioniše tako što u svakoj iteraciji nasumično menja dodelu jednog zadatka nekom drugom procesoru i proverava da li novo rešenje daje manje ukupno vreme izvršavanja (*makespan*). Ako je novo rešenje bolje, ono se prihvata kao trenutno rešenje, dok se lošija rešenja odbacuju.

Osnovni koraci algoritma su:

1. **Inicijalizacija:** Generiše se početni raspored zadataka na procesore nasumično.
2. **Lokalne promene:** U svakoj iteraciji bira se jedan zadatak i menja mu se procesor na kome će se izvršavati.
3. **Evaluacija:** Ako novo rešenje smanjuje ukupno vreme izvršavanja, prihvata se kao bolje rešenje.
4. **Zaustavljanje:** Algoritam se zaustavlja nakon određenog broja iteracija ili ako se ne pronalaze bolja rešenja.

Prednost lokalne pretrage je u njenoj jednostavnosti i brzom poboljšanju početnih rešenja, ali ima i ograničenja – često može zapeti u lokalnom minimumu, tj. pronaći rešenje koje je bolje od trenutnog, ali ne i globalno optimalno. Zbog toga se lokalna pretraga često kombinuje sa drugim metaheuristikama (npr. simulirano kaljenje ili genetski algoritmi) kako bi se poboljšala raznolikost pretrage.

Ovaj algoritam je koristan kod problema raspoređivanja jer brzo pronalazi zadovoljavajuća rešenja, naročito kada broj mogućih kombinacija m^T raste eksponencijalno i iscrpna pretraga postaje nepraktična.

```python
def local_search(execution_times, m, max_iterations):
    n = len(execution_times)
    schedule = [random.randint(0, m - 1) for _ in range(n)]
    current_makespan = calculate_completion_time(schedule, execution_times, m)

    for _ in range(max_iterations):
        new_schedule = schedule[:]
        task = random.randint(0, n - 1)
        new_schedule[task] = random.randint(0, m - 1)

        new_makespan = calculate_completion_time(new_schedule, execution_times, m)

        if new_makespan < current_makespan:
            schedule = new_schedule[:]
            current_makespan = new_makespan

    return schedule, current_makespan

max_iterations = 1000
best_schedule, best_makespan = local_search(execution_times, m, max_iterations)

print(f"Optimalni raspored sa lokalnom pretragom: {best_schedule}")
print(f"Minimalno vreme završetka: {best_makespan}")
```

4.2 Simulirano kaljenje

Simulirano kaljenje je metoda optimizacije inspirisana procesom kaljenja metala. Ideja je simulirati proces zagrevanja metala do visoke temperature, a zatim postepeno hlađenje, što dovodi do promene strukture metala i prelaska u stanje sa minimalnom energijom. Ovaj princip se prenosi na optimizaciju kako bi se pronašlo globalno optimalno rešenje.

U kontekstu problema **minimalnog raspoređivanja zadataka na više procesora**, simulirano kaljenje funkcioniše tako što se početno rešenje (raspored zadataka na procesore) postepeno menja tokom iteracija. U svakoj iteraciji algoritam nasumično menja dodelu jednog zadatka i izračunava novo ukupno vreme završetka (makespan).

Ako je novo rešenje bolje (kraće ukupno vreme završetka), algoritam ga prihvata. Ako je lošije, može ga prihvatiti sa određenom verovatnoćom koja je veća na početku (kada je "temperatura" visoka) i postepeno opada tokom vremena. Na taj način algoritam izbegava prerano zaglavljivanje u lokalnom optimumu i istražuje veći deo prostora mogućih rasporeda.

```python
def simulated_annealing(execution_times, m, max_iterations):
    n = len(execution_times)
    schedule = [random.randint(0, m - 1) for _ in range(n)]
    current_makespan = calculate_completion_time(schedule, execution_times, m)
    best_schedule = schedule[:]
    best_makespan = current_makespan

    for i in range(1,max_iterations):
        new_schedule = schedule[:]
        task = random.randint(0, n - 1)
        new_schedule[task] = random.randint(0, m - 1)

        new_makespan = calculate_completion_time(new_schedule, execution_times, m)

        if new_makespan < current_makespan:
            schedule = new_schedule
            current_makespan = new_makespan

            if new_makespan < best_makespan:
                best_schedule = new_schedule[:]
                best_makespan = new_makespan

        else:
            p = 1.0 / i ** 0.5
            q = random.uniform(0, 1)
            if p > q:
                schedule = new_schedule
                current_makespan = new_makespan

    return best_schedule, best_makespan
```

Slika 7: Kod funkcije *simulated annealing*

Ovaj pristup omogućava nalaženje boljeg rasporeda nego jednostavna heuristika, ali ne garantuje apsolutno optimalno rešenje. Algoritam se zaustavlja nakon određenog broja iteracija.
Ovim pristupom algoritam može pronaći raspored koji značajno smanjuje ukupno vreme izvršavanja zadataka, naročito u slučajevima kada postoji veliki broj zadataka i procesora, gde brute force metoda postaje previše spora.

4.3 Genetski algoritam

Genetski algoritam je metaheuristička metoda optimizacije inspirisana prirodnom evolucijom. Počinje sa populacijom nasumično generisanih rasporeda zadataka po procesorima (jedinki). Svaki raspored ima svoju prilagođenost (fitness) koja odražava koliko je raspored dobar – u našem slučaju, što je ukupno vreme završetka (makespan) manje, to je rešenje bolje.

Tokom evolucije populacije, algoritam koristi nekoliko osnovnih operatora:

- **Elitizam:** Najbolje jedinke (rasporedi sa najmanjim makespan-om) direktno se prenose u sledeću generaciju, čuvajući najbolja rešenja tokom evolucije.
- **Selekcija:** Iz postojeće populacije biraju se jedinke koje imaju veće šanse da učestvuju u stvaranju novih rešenja. Bolja rešenja imaju veću verovatnoću izbora.
- **Ukrštanje (Crossover):** Dva roditeljska rasporeda kombinuju se tako da deo rasporeda dolazi od jednog roditelja, a drugi deo od drugog, stvarajući potomke sa potencijalno boljim karakteristikama od oba roditelja.
- **Mutacija:** Sa određenom verovatnoćom menja se procesor na kojem je neki zadatak zakazan, čime se održava diverzitet populacije i izbegava prerano zaglavljivanje u lokalnom optimumu.

Ovaj proces se ponavlja kroz unapred definisan broj generacija. Na kraju algoritam vraća raspored zadataka sa najmanjim pronađenim makespan-om.

```python
class Individual:

    def __init__(self, T, m, execution_times):
        self.code = [random.randint(0, m - 1) for _ in range(T)]
        self.execution_times = execution_times
        self.m = m
        self.calc_fitness()

    def calc_fitness(self):
        processor_times = [0] * self.m
        for task, processor in enumerate(self.code):
            processor_times[processor] += self.execution_times[task][processor]
        makespan = max(processor_times)
        self.fitness = -makespan
```

```python
def selection(population, k):
    k = min(len(population), k)
    participants = random.sample(population, k)
    return max(participants, key=lambda x: x.fitness)

def crossover(parent1, parent2, child1, child2):
    breakpoint = random.randrange(1, len(parent1.code))
    child1.code[:breakpoint] = parent1.code[:breakpoint]
    child1.code[breakpoint:] = parent2.code[breakpoint:]

    child2.code[:breakpoint] = parent2.code[:breakpoint]
    child2.code[breakpoint:] = parent1.code[breakpoint:]

def mutation(child, p):
    for i in range(len(child.code)):
        if random.random() < p:
            child.code[i] = random.randint(0, child.m - 1)
```

```python
def ga(T, m, execution_times, population_size, num_generations, tournament_size, mutation_prob, elitism_size):
    assert population_size >= 2
    population = [Individual(T, m, execution_times) for _ in range(population_size)]
    new_population = [Individual(T, m, execution_times) for _ in range(population_size)]

    if elitism_size % 2 != population_size % 2:
        elitism_size += 1

    best_fitnesses = []
    for it in range(num_generations):
        population.sort(key=lambda x: x.fitness, reverse=True)
        best_fitnesses.append(-population[0].fitness)  # Append makespan (positive value)
        new_population[:elitism_size] = deepcopy(population[:elitism_size])

        for i in range(elitism_size, population_size, 2):
            parent1 = selection(population, tournament_size)
            tmp, parent1.fitness = parent1.fitness, float('-inf')
            parent2 = selection(population, tournament_size)
            parent1.fitness = tmp

            crossover(parent1, parent2, new_population[i], new_population[i + 1])

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

            new_population[i].calc_fitness()
            new_population[i + 1].calc_fitness()

        population = deepcopy(new_population)

    best_individual = max(population, key=lambda x: x.fitness)
    print(f'Best makespan: {-best_individual.fitness}, Best schedule: {best_individual.code}')
    plt.plot(best_fitnesses)
    plt.xlabel("Generation")
    plt.ylabel("Best Makespan")
    plt.title("Genetic Algorithm - Task Scheduling")
    plt.show()
```

4.4 Optimizacija rojem čestica

Optimizacija rojem čestica (eng. *Particle Swarm Optimization*) je metaheuristički algoritam inspirisan ponašanjem jata ptica ili roja insekata u prirodi. Ideja je da čestice (kandidati rešenja) "lete" kroz prostor mogućih rešenja, prateći svoje najbolje pronađeno rešenje (lokalno najbolje) i najbolje rešenje koje je do sada pronašao ceo roj (globalno najbolje).
U kontekstu problema **minimalnog raspoređivanja zadataka na više procesora**, svaka čestica predstavlja jedan raspored zadataka na procesore. Algoritam funkcioniše na sledeći način:

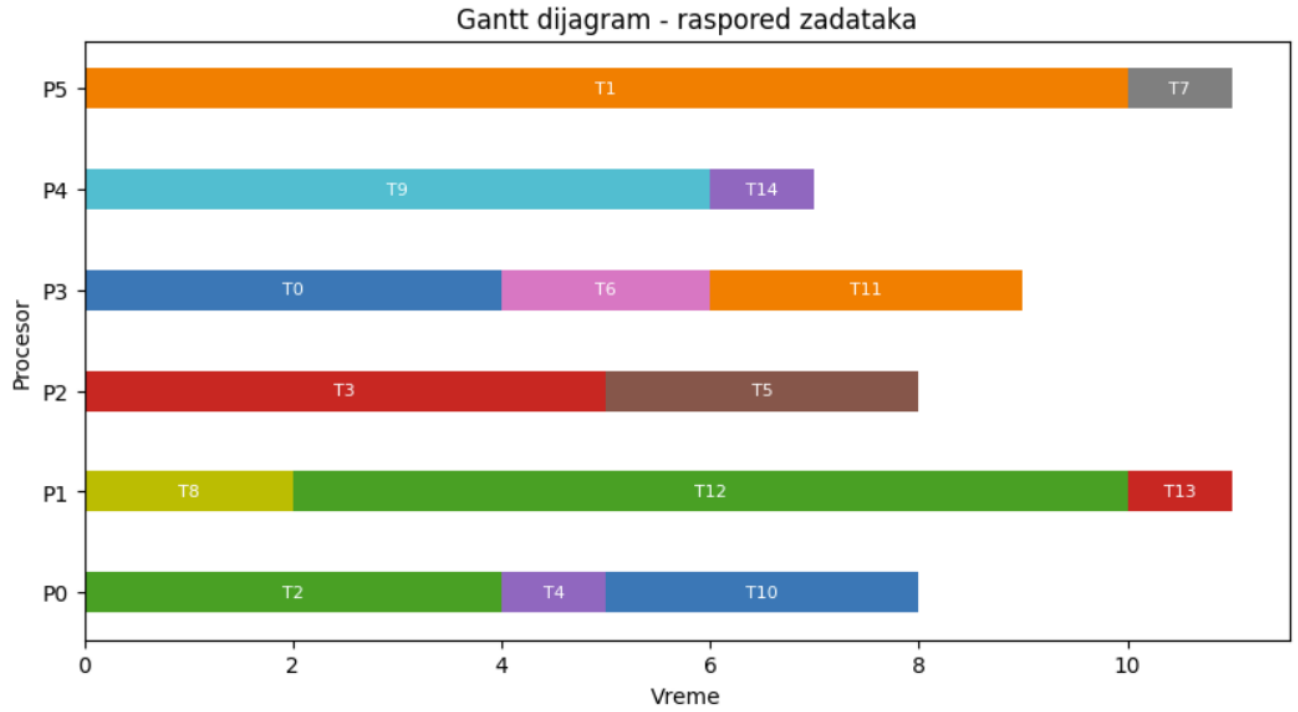
- **Inicijalizacija:** Svaka čestica se inicijalno postavlja na nasumičan raspored i dobija početnu brzinu kretanja kroz prostor pretrage.
- **Lično najbolje rešenje:** Svaka čestica pamti svoj najbolji raspored koji je do sada pronašla.
- **Globalno najbolje rešenje:** Roj pamti najbolje rešenje koje je pronašla bilo koja čestica u roju.
- **Ažuriranje brzine i pozicije:** Čestice menjaju svoj položaj u prostoru rešenja na osnovu sopstvenog najboljeg rešenja (kognitivna komponenta) i globalnog najboljeg rešenja (socijalna komponenta). Brzina se ažurira kako bi čestice konvergirale ka boljem rasporedu zadataka.
- **Zaustavljanje:** Nakon unapred definisanog broja iteracija, algoritam vraća raspored koji daje najmanje ukupno vreme završetka (makespan).

PSO algoritam je posebno koristan kod problema gde postoji veliki broj mogućih rasporeda, jer umesto iscrpne pretrage koristi kolektivno ponašanje čestica za brzo približavanje optimalnom rešenju.

- Position = [1,0,1,0]
- Velociy -> Koliko ce se pomaknuti cestica

```
class Particle:
    swarm_best_position = None
    swarm_best_value = float('inf')

    def __init__(self, T, m, execution_times, c_inertia, c_social, c_cognitive):
        self.T = T
        self.m = m
        self.execution_times = execution_times
        self.c_inertia = c_inertia
        self.c_social = c_social
        self.c_cognitive = c_cognitive

        self.position = np.random.randint(0, m, size=T)
        self.velocity = np.random.uniform(-1, 1, size=T)
        self.value = self.calculate_completion_time()

        self.personal_best_position = self.position.copy()
        self.personal_best_value = self.value

        if Particle.swarm_best_position is None or self.value < Particle.swarm_best_value:
            Particle.swarm_best_position = self.position.copy()
            Particle.swarm_best_value = self.value

    def calculate_completion_time(self):
        processor_times = [0] * self.m
        for task, processor in enumerate(self.position):
            processor_times[processor] += self.execution_times[task][processor]
        return max(processor_times)

    def update_velocity(self):
        r_s = np.random.random(len(self.position))
        r_c = np.random.random(len(self.position))
        social_velocity = self.c_social * r_s * (Particle.swarm_best_position - self.position)
        cognitive_velocity = self.c_cognitive * r_c * (self.personal_best_position - self.position)
        self.velocity = self.c_inertia * self.velocity + social_velocity + cognitive_velocity

    def move(self):
        self.update_velocity()
        self.position = np.clip(self.position + self.velocity, 0, self.m - 1).astype(int)
        self.value = self.calculate_completion_time()

        if self.value < self.personal_best_value:
            self.personal_best_position = self.position.copy()
            self.personal_best_value = self.value
            if self.value < Particle.swarm_best_value:
                Particle.swarm_best_position = self.position.copy()
                Particle.swarm_best_value = self.value
```

```
Testiram za T = 15, m = 6...
```



Nakon izvršavanja PSO algoritma, dobijamo raspored zadataka na procesore koji daje najmanje ukupno vreme izvršavanja (makespan). Vrednosti prikazane u testovima pokazuju kako algoritam pronalazi rešenja za različit broj zadataka (**T**) i procesora (**m**).

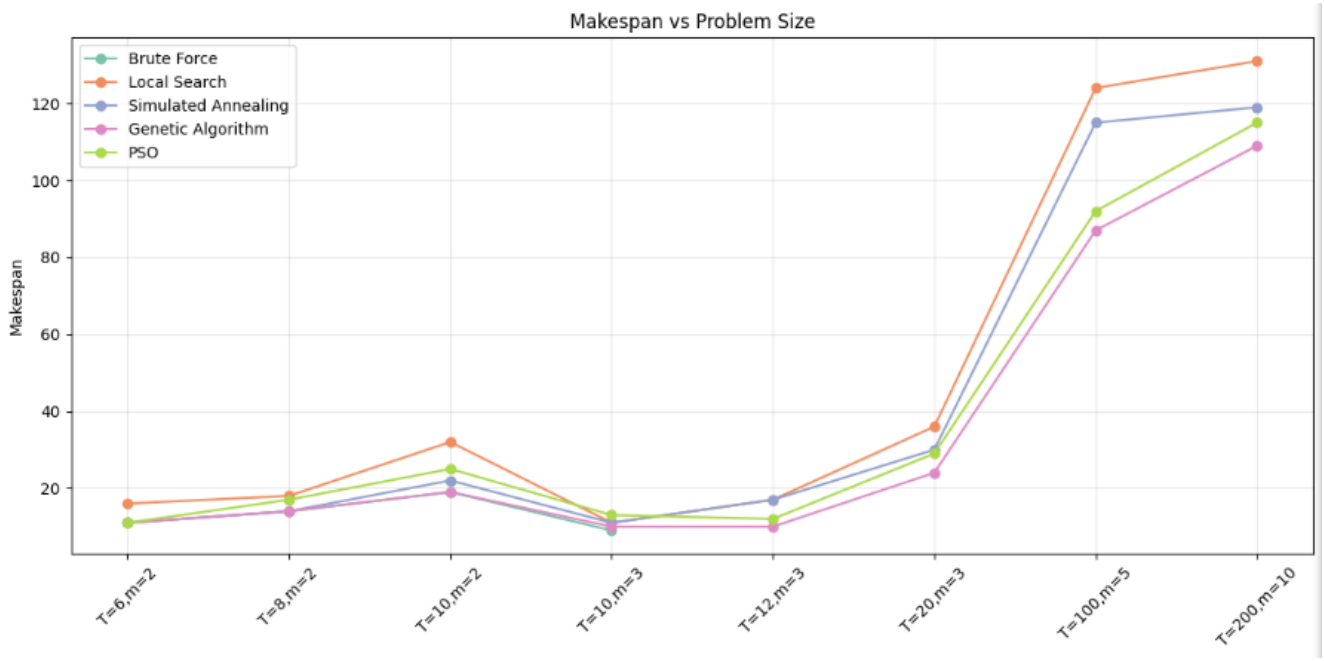
Algoritam pokazuje sledeće karakteristike:

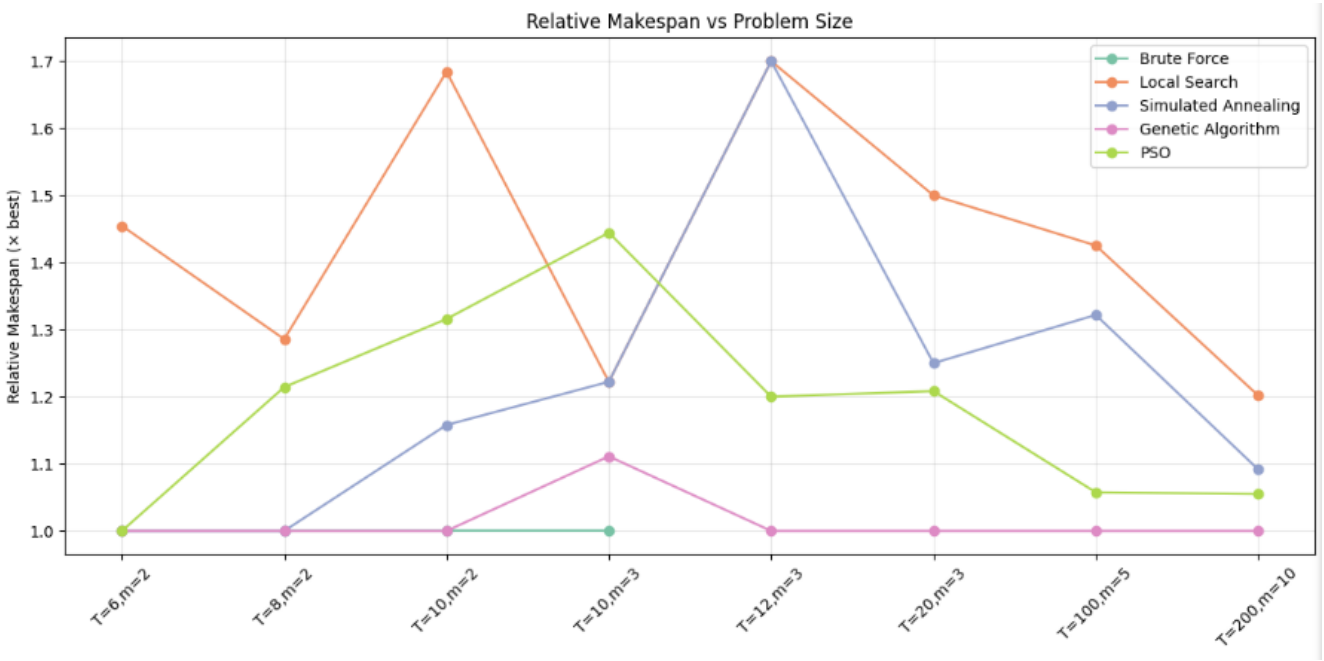
- **Brza konvergencija:** PSO već nakon određenog broja iteracija pronalazi rešenja bliska optimalnim, značajno smanjujući vreme pretrage u poređenju sa brute force metodom.
- **Adaptivno ponašanje:** Zahvaljujući socijalnoj i kognitivnoj komponenti, čestice dele informacije o dobrim rešenjima i adaptivno se kreću ka boljim rasporedima.
- **Primena na veće probleme:** Za veći broj zadataka brute force pristup postaje neupotrebljiv zbog eksponencijalnog rasta broja mogućih rasporeda (m^T), dok PSO uspeva da pronalazi kvalitetna rešenja u razumnom vremenskom roku.
- **Nedostatak globalne garancije:** Pošto je PSO heuristički algoritam, ne garantuje uvek pronalaženje apsolutno optimalnog rešenja, ali u praksi često daje dovoljno dobra rešenja vrlo brzo.

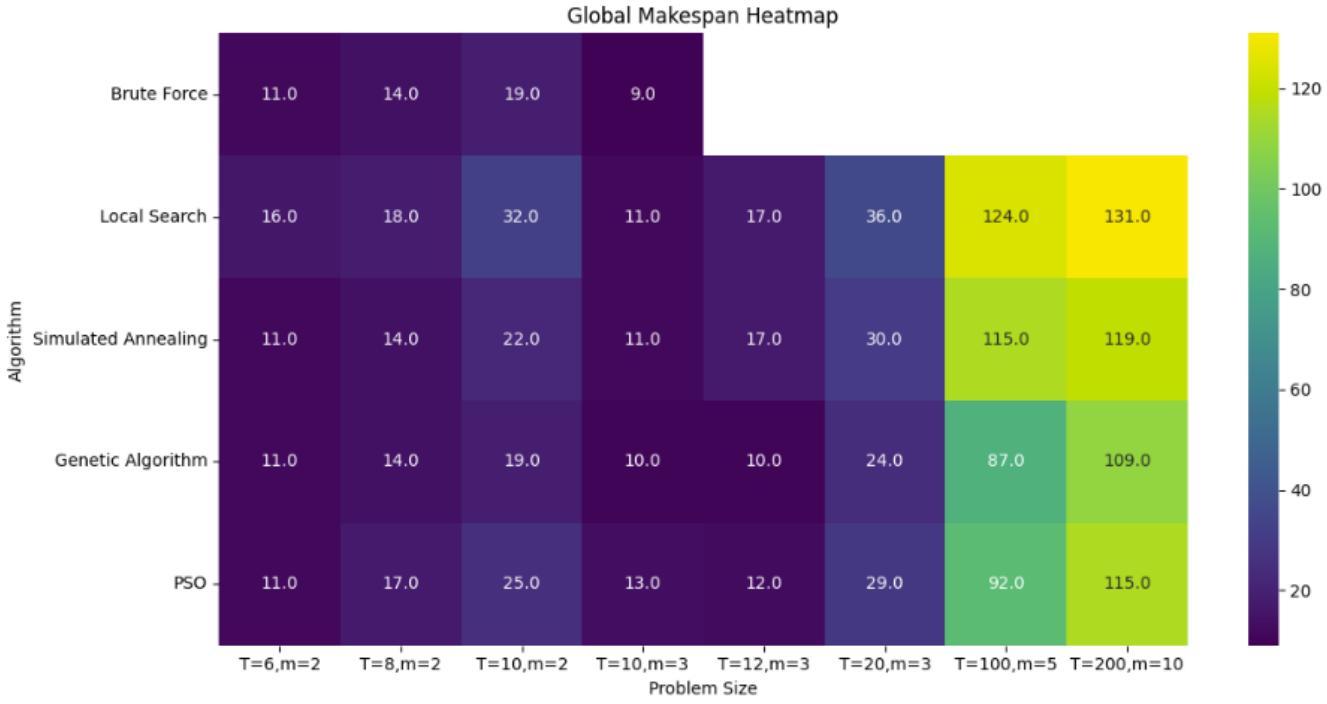
Ovaj algoritam je pogodan za probleme planiranja i raspoređivanja, gde je broj mogućih rešenja veoma velik, a cilj je pronaći dobar raspored u kratkom vremenskom intervalu, što ga čini efikasnijim od determinističkih metoda poput brute force-a.

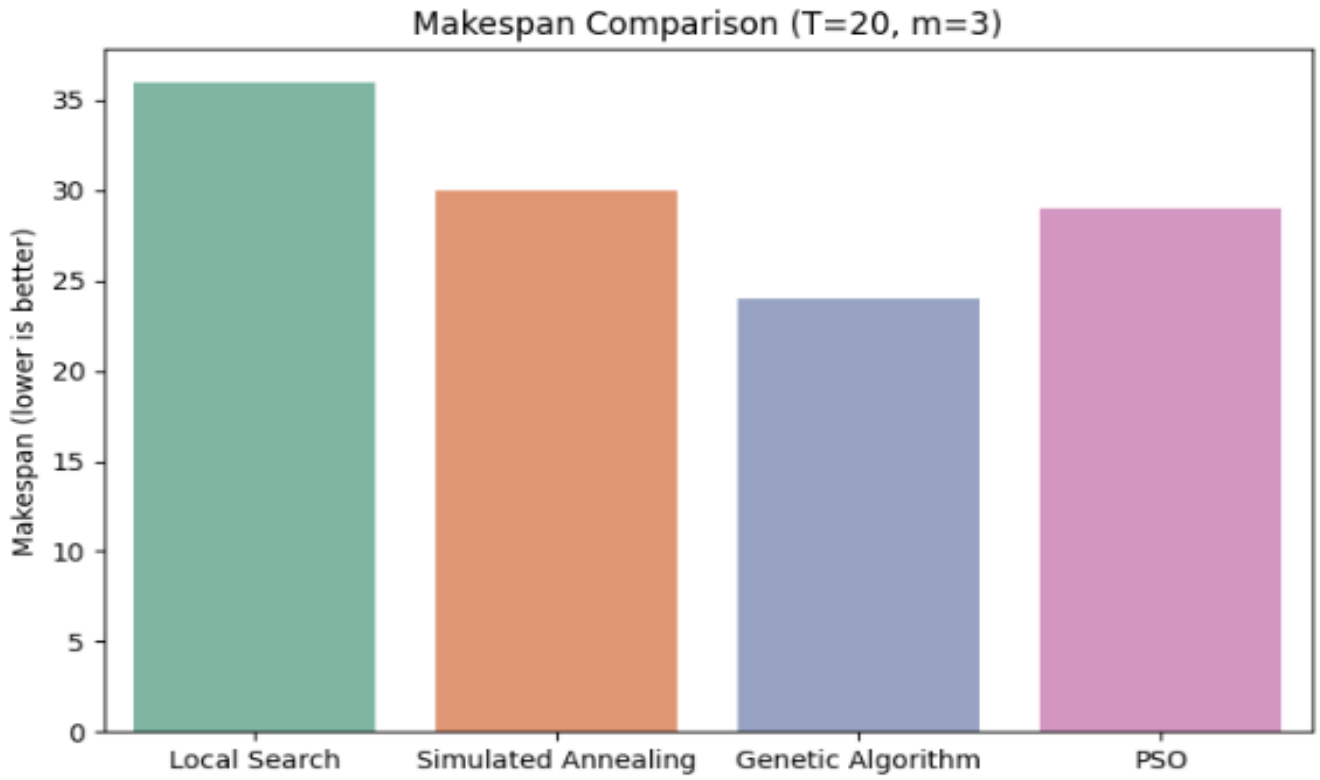
5 Poredjenje

5.1 Makespan algoritama po veličini problema

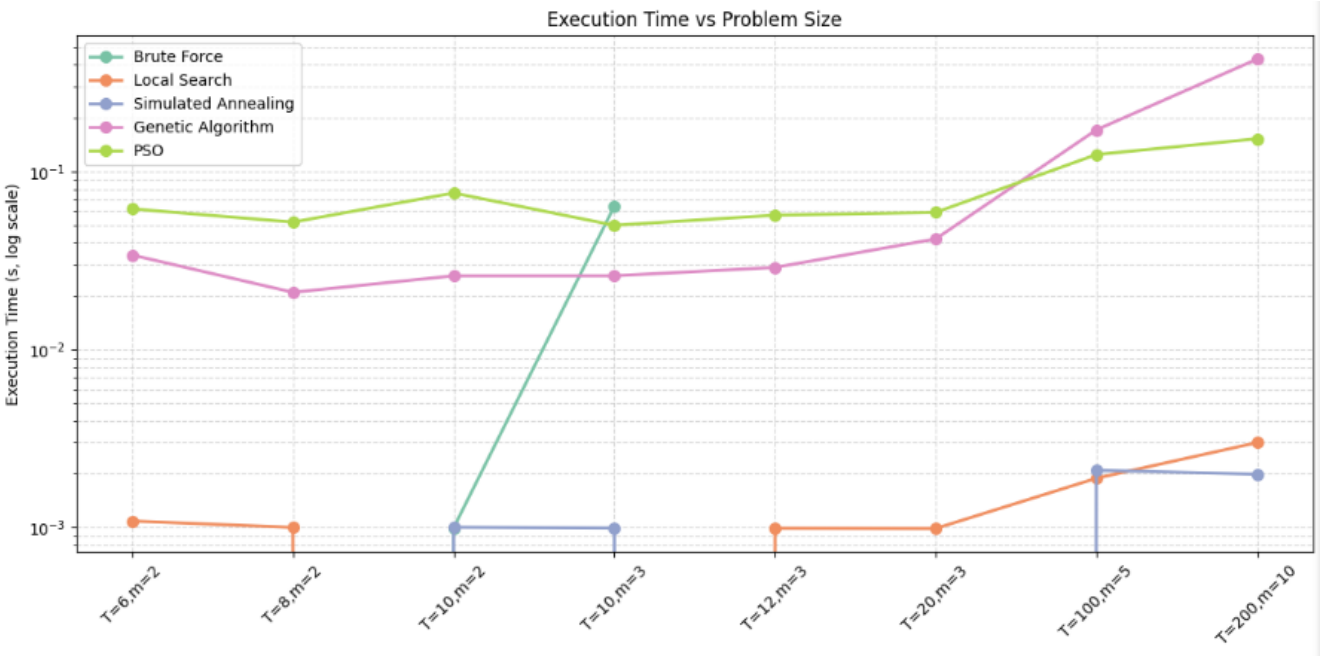


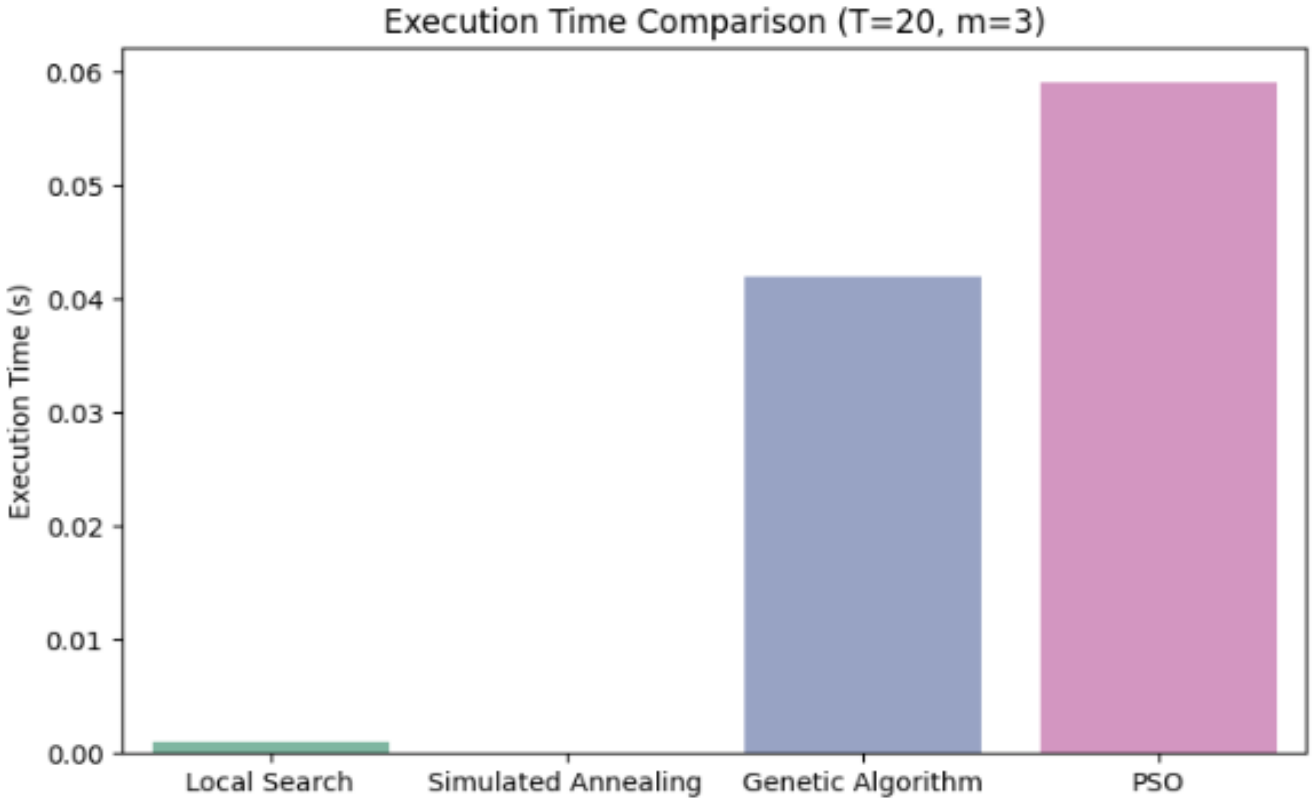



Global Makespan Heatmap
Brute Force
Local Search
Simulated Annealing
Genetic Algorithm
PSO
Algorithm
11.0 14.0 19.0 9.0
16.0 18.0 32.0 11.0 17.0 36.0 124.0 131.0
11.0 14.0 22.0 11.0 17.0 30.0 115.0 119.0
11.0 14.0 19.0 10.0 10.0 24.0 87.0 109.0
11.0 17.0 25.0 13.0 12.0 29.0 92.0 115.0
T=6,m=2
T=8,m=2
T=10,m=2
T=10,m=3
T=12,m=3
T=20,m=3
T=100,m=5
T=200,m=10
Problem Size
120
100
80
60
40
20

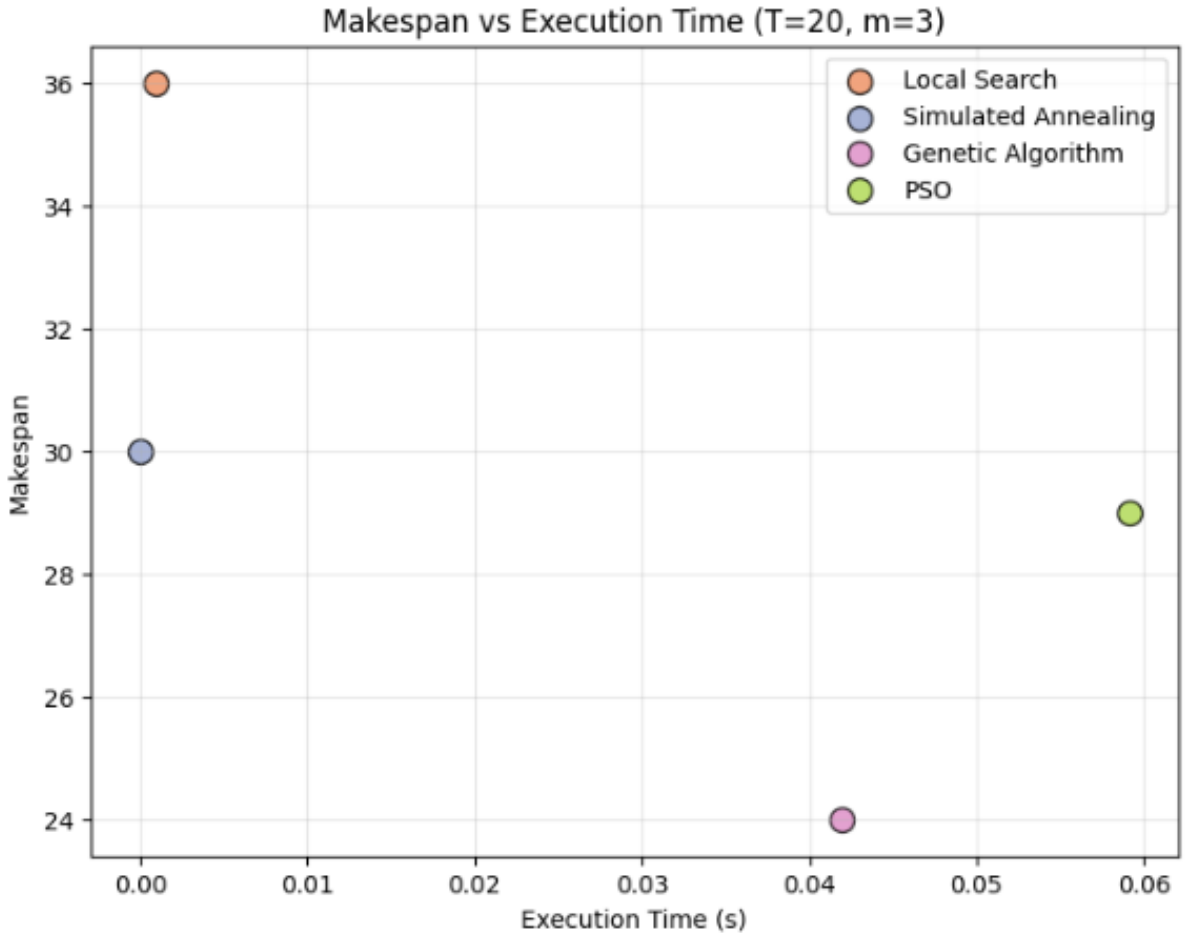

Makespan Comparison (T=20, m=3)
Makespan (lower is better)
0
5
10
15
20
25
30
35
Local Search
Simulated Annealing
Genetic Algorithm
PSO

5.2 Vreme algoritama po veličini problema

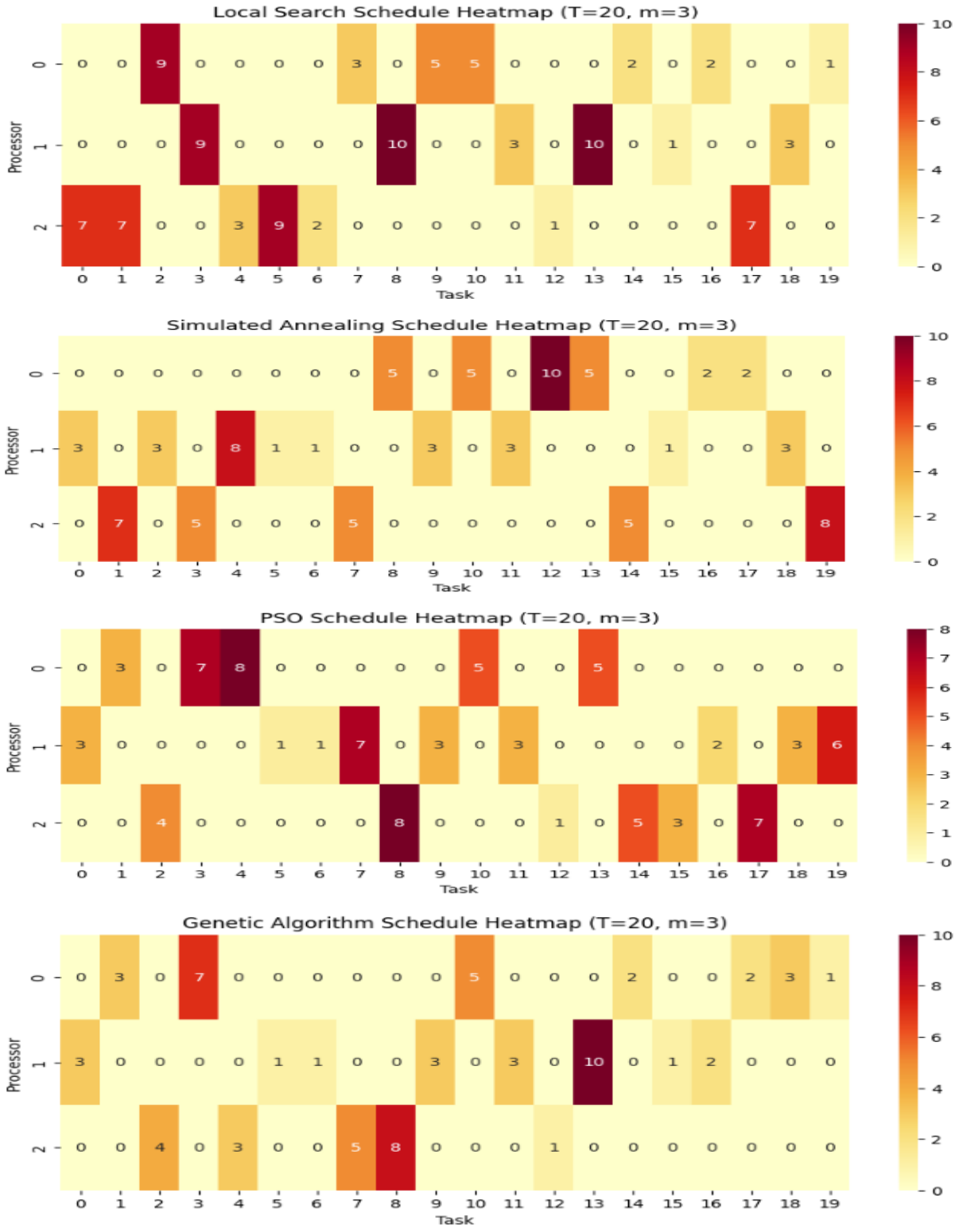




5.3 Makespan naspram vremena



5.4 Heatmapa procesora na zadacima



6 Praktična primena

Problem minimum multiprocessor scheduling nalazi svoju primenu u različitim oblastima gde je neophodno optimalno rasporediti zadatke na ograničen broj procesora ili resursa kako bi se minimizovalo ukupno vreme izvršavanja.

Jedna od tipičnih primena jeste u računarskim klasterima i cloud okruženjima, gde se veliki broj zadataka mora rasporediti na više procesora ili servera. Cilj je da se obezbedi ravnomerno opterećenje procesora i da se smanji vreme završetka svih zadataka, što direktno utiče na efikasnost celog sistema.

Sličan problem se javlja i u proizvodnim sistemima, gde različite operacije (zadaci) treba da budu izvršene na određenom broju mašina (procesora). Efikasno raspoređivanje zadataka na mašine omogućava bolje korišćenje resursa i kraće vreme izrade proizvoda.

U telekomunikacionim mrežama, multiprocessor scheduling se koristi za dodelu komunikacionih zahteva više kanala ili linkova. Zadatak je da se zahtevi rasporede tako da nijedan kanal nije preopterećen, a da se ukupno vreme prenosa podataka minimizuje.

Takođe, značajna primena postoji u real-time sistemima, gde zadaci moraju biti izvršeni u strogo definisanim vremenskim ograničenjima. U ovim slučajevima, algoritmi raspoređivanja moraju garantovati da će svi kritični zadaci biti izvršeni na vreme, čime se obezbeđuje pouzdan rad sistema (npr. u avio-industriji, medicinskoj opremi ili robotici).

Zahvaljujući ovim primenama, problem minimalnog raspoređivanja na više procesora ima ključnu ulogu u projektovanju sistema visokih performansi, jer omogućava balans između efikasnog korišćenja resursa i smanjenja vremena izvršavanja zadataka.

7 Zaključak

Izbor odgovarajućeg algoritma za minimum multiprocessor scheduling zahteva pažljivo balansiranje između preciznosti rešenja, vremena izvršavanja i zahteva za resursima. Svaki od algoritama koji se primenjuje na ovaj problem ima svoje prednosti i mane, pa je važno prilagoditi izbor konkretnoj instanci problema kako bi se postiglo optimalno rešenje. U nekim slučajevima, kombinacija više algoritama ili optimizacija parametara može predstavljati efikasnu strategiju za rešavanje složenih zadataka raspoređivanja.

Prvo, veličina problema (broj zadataka T i broj procesora m) je od suštinskog značaja. Kao što je pokazano, Brute Force pristup, iako precizan, postaje nepraktičan za veći broj zadataka, jer broj mogućih rasporeda raste eksponencijalno. Sa druge strane, heuristički algoritmi kao što su Genetski algoritmi ili Particle Swarm Optimization (PSO) mogu biti mnogo efikasniji za veće probleme, jer obezbeđuju stabilne performanse i dobru skalabilnost.

Preciznost rešenja je takođe važan faktor. Ako je cilj pronaći tačno optimalno rešenje, Brute Force može biti izbor, ali uz cenu velikog vremena izvršavanja. Ukoliko je prihvatljiva određena aproksimacija, algoritmi kao što su Simulirano kaljenje ili lokalna pretraga mogu ponuditi dovoljno dobra rešenja uz znatno kraće vreme izvršavanja. Genetski algoritmi i PSO se u praksi često izdvajaju kao brži i robusniji, čineći ih pogodnim u situacijama gde je brzina presudna.

Na kraju, pažljiv izbor algoritma za minimum multiprocessor scheduling zavisiće od konkretnih zahteva sistema. Ako je ključna tačnost, algoritmi egzaktne prirode (Brute Force ili Integer Programming) imaju prednost. Kada su u pitanju veliki sistemi gde su brzina i efikasnost prioritet, heuristike i metaheuristike poput PSO, Genetskih algoritama, lokalne pretrage ili simuliranog kaljenja nude dobar balans između tačnosti i vremena izvršavanja. Takođe, kombinovanje različitih metoda i fino podešavanje parametara može biti korisna strategija za rešavanje složenih instanci ovog problema.

Literatura

[1] MINIMUM MULTIPROCESSOR SCHEDULING,
online at: https://www.csc.kth.se/~viggo/wwwcompendium/node181.html