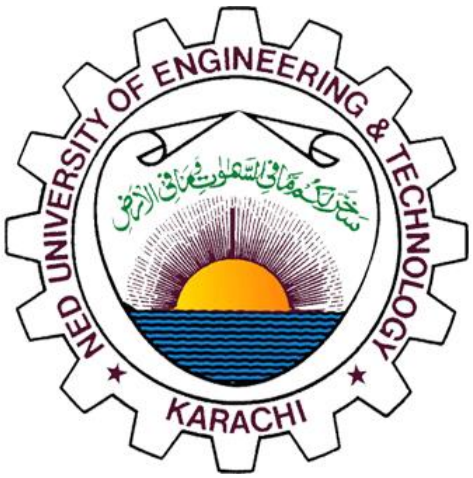
NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER AND INFORMATION SYSTEM ENGINEERING



OPEN ENDED LAB

CS-323 ARTIFICIAL INTELLIGENCE

TE CS BATCH 2022

FALL SEMESTER 2024

TITLE: "JOB SCHEDULING USING GENETIC ALGORITHM"

TABLE OF CONTENTS

INTRODUCTION:

The genetic algorithm (GA) is an optimization technique inspired by natural selection and evolution. It is widely used to solve complex optimization problems, especially where traditional methods might fail due to the vast solution space. In this project, we have applied a genetic algorithm to solve the **Job Scheduling Problem**, a classic optimization problem where the objective is to minimize the **makespan** (the total time required to complete all jobs) by scheduling jobs across multiple machines.
We will also find the best scheduling order of jobs for the machines which will require the minimum time to complete all the jobs.

PROBLEM DESCRIPTION:

The **Job Scheduling Problem** involves scheduling a set of jobs, each with a specific processing time, on a fixed number of machines. The objective is to minimize the makespan, i.e., the time taken by the last machine to finish its assigned jobs. The scheduling needs to balance the load across the machines so that the maximum time taken by any machine is as small as possible.

In this implementation, we have:

- **12 jobs** with varying processing times.
- **3 machines** where jobs can be scheduled.
- The goal is to determine the best sequence of jobs for each machine that minimizes the makespan.

WORKING OF ALGORITHM:

The genetic algorithm operates on a **population** of possible solutions, evolving over a number of generations. The genetic algorithm will iteratively improve the solutions(schedules) by mimicking evolutionary processes such as selection, crossover, and mutation. In each generation:

- **Selection**: Individuals (schedules) are selected from the population based on their fitness (the makespan).
- **Crossover**: Pairs of individuals (parents) are combined to create offspring (new schedules).
- **Mutation**: Random modifications are applied to offspring to introduce diversity.
- **Replacement**: The new population is formed by replacing the old population with the new generation.

1. Initialization

The algorithm begins by creating a population of random schedules. Each schedule is represented as a list of job indices, where the order of jobs indicates their processing sequence. The population size is set to 20, meaning there are 20 initial random schedules.

2. Fitness Function

The fitness function evaluates the quality of each schedule. It is calculated by determining the makespan of the schedule, where the makespan is the total time taken by the last machine to complete its assigned jobs. The fitness of a schedule is defined as the reciprocal of the makespan, i.e. schedules with smaller makespans are assigned higher fitness values.

3. Selection

Selection is performed using a **roulette wheel** mechanism. Two parents are selected from the population by randomly choosing a set of three candidates and picking the one with the best fitness (one with the smallest makespan).

4. Crossover

Crossover involves combining two parent schedules to produce a child schedule. This is done by randomly selecting a segment (a sublist of jobs) from one parent and filling in the remaining jobs from the second parent. This ensures that the child schedule inherits the structure of both parents while maintaining diversity. The probability of crossover rate is set to 0.7.

5. Mutation

Mutation introduces small random changes to a schedule. In this implementation, mutation involves swapping two randomly selected jobs in the schedule. This operation is applied with a probability of 0.1 (mutation rate).

6. Termination

The algorithm runs for a fixed number of generations (50 in our case) which is the termination condition. The best schedule from the final generation is selected as the solution, which corresponds to the schedule with the minimum makespan.

CODE IMPLEMENTATION:

```python
import random
jobs = 12
processing_times = [3, 2, 7, 5, 1, 6, 4, 8, 2, 1, 4, 5]
machines = 3
population_size = 20
no_of_generations = 50
crossover_rate = 0.7
mutation_rate = 0.1

def calculate_makespan(schedule):
    machine_times = [0] * machines
    for job in schedule:
        next_machine = machine_times.index(min(machine_times))
        machine_times[next_machine] += processing_times[job]
    return max(machine_times)

def initialize_population():
    population = []
    for i in range(population_size):
        individual = random.sample(range(jobs), jobs)
        population.append(individual)
    return population


def fitness(schedule):
    return 1 / calculate_makespan(schedule)

def select_parents(population):
    parents = []
    for i in range(2):
        candidates = random.sample(population, 3)
        parents.append(max(candidates, key=fitness))
    return parents

def crossover(parent1, parent2):
    if random.random() > crossover_rate:
        return parent1[:]
    start, end = sorted(random.sample(range(jobs), 2))
    child = [None] * jobs
    child[start:end] = parent1[start:end]
    pointer = end
    for job in parent2:
        if job not in child:
            if pointer >= jobs:
                pointer = 0
            child[pointer] = job
            pointer += 1
    return child

def mutate(schedule):
    if random.random() < mutation_rate:
        i, j = random.sample(range(jobs), 2)
        schedule[i], schedule[j] = schedule[j], schedule[i]
```

```
54
55  def genetic_algorithm():
56      population = initialize_population()
57      for generation in range(no_of_generations):
58          new_population = []
59          for i in range(population_size):
60              parent1, parent2 = select_parents(population)
61              child = crossover(parent1, parent2)
62              mutate(child)
63              new_population.append(child)
64          population = new_population
65          best_schedule = min(population, key=calculate_makespan)
66          print(f"Generation {generation+1}: Best Makespan = {calculate_makespan(best_schedule)}")
67      return best_schedule
68
69  best_solution = genetic_algorithm()
70  print("Best Schedule:", best_solution)
71  print("Minimum Makespan:", calculate_makespan(best_solution))
72
```

EXPLANATION OF CODE :

• **calculate_makespan(schedule)**: This function computes the makespan for a given schedule. It allocates jobs to machines in the order specified by the schedule and calculates the maximum time taken by any machine.

• **initialize_population( )**: This function generates a population of random schedules. It creates a list of schedules where each schedule is a random permutation of job indices.

• **fitness(schedule)**: The fitness function computes the reciprocal of the makespan, ensuring that schedules with smaller makespans are fitter.

• **select_parents(population)**: This function selects two parents from the population using tournament selection. It randomly selects three candidates and chooses the one with the best fitness.

• **crossover(parent1, parent2)**: This function performs a one-point crossover between two parent schedules to generate a child schedule. The crossover ensures that the child inherits jobs from both parents while maintaining job order consistency.

• **mutate(schedule)**: This function performs mutation by randomly swapping two jobs in the schedule with a probability determined by the mutation rate.

• **genetic_algorithm()**: This is the main function that runs the genetic algorithm for a given number of generations. It iteratively selects parents, performs crossover and mutation, and evolves the population towards an optimal schedule.

EXAMPLE:

```
jobs = 12
processing_times = [3, 2, 7, 5, 1, 6, 4, 8, 2, 1, 4, 5]
machines = 3
population_size = 20
no_of_generations = 50
crossover_rate = 0.7
mutation_rate = 0.1
```

By setting these values we get:

OUTPUT:

```
Generation 1: Best Makespan = 17
Generation 2: Best Makespan = 17
Generation 3: Best Makespan = 17
Generation 4: Best Makespan = 17
Generation 5: Best Makespan = 17
Generation 6: Best Makespan = 17
Generation 7: Best Makespan = 17
Generation 8: Best Makespan = 16
Generation 9: Best Makespan = 16
Generation 10: Best Makespan = 16
Generation 11: Best Makespan = 16
Generation 12: Best Makespan = 16
Generation 13: Best Makespan = 16
Generation 14: Best Makespan = 16
Generation 15: Best Makespan = 16
Generation 16: Best Makespan = 16
Generation 17: Best Makespan = 16
Generation 18: Best Makespan = 16
Generation 19: Best Makespan = 16
Generation 20: Best Makespan = 16
Generation 21: Best Makespan = 16
Generation 22: Best Makespan = 16
Generation 23: Best Makespan = 16
Generation 24: Best Makespan = 16
Generation 25: Best Makespan = 16
Generation 26: Best Makespan = 16
Generation 27: Best Makespan = 16
Generation 28: Best Makespan = 16
Generation 29: Best Makespan = 16
Generation 30: Best Makespan = 16
```

```
Generation 31: Best Makespan = 16
Generation 32: Best Makespan = 16
Generation 33: Best Makespan = 16
Generation 34: Best Makespan = 16
Generation 35: Best Makespan = 16
Generation 36: Best Makespan = 16
Generation 37: Best Makespan = 16
Generation 38: Best Makespan = 16
Generation 39: Best Makespan = 16
Generation 40: Best Makespan = 16
Generation 41: Best Makespan = 16
Generation 42: Best Makespan = 16
Generation 43: Best Makespan = 16
Generation 44: Best Makespan = 16
Generation 45: Best Makespan = 16
Generation 46: Best Makespan = 16
Generation 47: Best Makespan = 16
Generation 48: Best Makespan = 16
Generation 49: Best Makespan = 16
Generation 50: Best Makespan = 16
Best Schedule: [11, 4, 0, 5, 2, 7, 3, 10, 6, 9, 1, 8]
Minimum Makespan: 16
```

NOVEL SIMULATION AND APPLICATION:

1) Job Scheduling in manufacturing areas:

This genetic algorithm approach can be extended to solve more complex job scheduling problems in manufacturing environments. For example, when scheduling jobs across multiple machines in a factory, each machine may have different capabilities and constraints, such as varying processing speeds or maintenance schedules. The genetic algorithm can be adapted to consider these factors and provide an optimized job schedule that minimizes downtime and improves overall throughput.

2) Job Scheduling for Cloud Computing

In a cloud computing environment, jobs are dynamically allocated to machines based on resource availability. The genetic algorithm can be used to optimize the allocation of virtual machines to incoming jobs, minimizing the time taken to process all tasks and balancing the load across machines in real-time.

CONCLUSION:

The genetic algorithm is an effective method for solving optimization problems like job scheduling. By simulating natural evolutionary processes, it can efficiently explore large solution spaces and find near-optimal solutions. In this project, we applied GA to solve the job scheduling problem with multiple machines and varying job processing times. The results show that the GA can successfully minimize the makespan, demonstrating its potential for use in real-world optimization problems.

DEPARTMENT OF COMPUTER & INFORMATION SYSTEMS ENGINEERING

Course Code:

CS-323

Course Title: Artificial Intelligence

Open Ended Lab

TE Batch 2022,

Fall Semester

2024 Grading

Rubric

Group Members:

Student No.	Name	Roll No.
S1	**Muhammad Muneeb Zafar**	**CS-22129**
S2	**Ali Mehdi**	**CS-22140**
S3	**Muhammad Sarim Khan**	**CS-22142**

CRITERIA AND SCALES				**Marks Obtained**		
				S 1	**S 2**	**S 3**
Criterion 1: Has the student appropriately simulated the working of the genetic algorithm?						
0	1	2	-			
The explanation is too basic.	The algorithm is explained well with an example.	The explanation is much more comprehensive.				
Criterion 2: How well is the student's understanding of the genetic algorithm?						
0	1	2	3			
The student has no understanding.	The student has a basic understanding.	The student has a good understanding.	The student has an excellent understanding.			
Criterion 3: How good is the programming implementation?						
0	1	2	3			
The project could not be implemented.	The project has been implemented partially.	The project has been implemented completely but can be improved.	The project has been implemented completely and impressively.			
Criterion 4: How good is the selected application?						
0	1	2	-			
The chosen application is too simple.	The application is fit to be chosen.	The application is different and impressive.				
Criterion 5: How well-written is the report?						
0	1	2	-			
The submitted report is unfit to be graded.	The report is partially acceptable.	The report is complete and concise.				
			Total Marks:			