Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Artificial Intelligence

ENCS3340

Project #1

Genetic Algorithm Project

Prepared by:

- Ali Shaikh Qasem ID: 1212171
- Abdalrahman Juber ID: 1211769

Instructor: Dr. Ismail Khater.

Section: 2.

Date: 18/5/2024

Birzeit University

2023-2024

Problem formulation:

The project has divided into many parts such that chromosome representation, cross-over, mutation, and the used objective function.

Chromosome representation: the order of jobs for each product was used as chromosome representation for all individuals for instance if we have 3 jobs one of chromosomes will be (J1, J2, J3) and another one will be (J2, J3, J1) and so on this representation will be for other chromosomes.

Initial population: In order to determine the initial population, we randomly choose a group of people (chromosomes) and then remove any duplicates.

Cross-over: After getting best fitness of two parents we do for them the cross-over operation which will chooses random position for copying the two parents to two children before the cross-over position then finding jobs are not in that child from other parent.

For instance if we had :C1 = ['Job_3', 'Job_1', 'Job_2'] and C2=['Job_1', 'Job_2', 'Job_3'] such that the cross-over operator will be in position 2 : C3 = ['Job_3', 'Job_1', 'Job_2'] , C4 = ['Job_1', 'Job_2', 'Job_3'] First we take all the jobs before the crossover without changing them, then we check for a job is not inside that child from the other parent and so on.

As we noticed above, we have similar values for parents and child so we need to do mutation so that we don`t get stuck.

Mutation: For not getting stuck inside the genetic algorithm by having similar chromosomes every time we do that loop, we need to mutation to made a random change. for this project the mutation will find two random positions for each gen then replacing them.

So, from previous results if we swap the jobs for 2 children we got:

C3 = ['Job_3', 'Job_2', 'Job_1']

C4 = ['Job_2', 'Job_1', 'Job_3']

C3 = ['Job_3', 'Job_1', 'Job_2']

And,

C4 = ['Job_2', 'Job_3', 'Job_1']

Also, we need a specific number of iteration so we can get the best solution and not getting stuck in the loop.

Objective function

The objective function (also known as fitness function) is used in genetic algorithm to determine how good is a solution. In our problem, we designed a fitness function that determines the makespan time, that is the time where the last operation of the last job is finished. Thus, a solution with less fitness value will be better.

The process of determining the makespan time of each solution is pretty simple, we first initialize a list contains machines end-times and jobs end-times, then we iterate through list of jobs and their sequences based on their order in the chromosome in a column wise direction and we keep updating the end-times for the corresponding machines and jobs, the makespan value is the maximum value in the jobs end-times list (see the code in Appendix).

Example: assume the following jobs and their sequences:

Job_1: M1[10] -> M2[5] -> M4[12]

Job_2: M2[7] -> M3[15] -> M1[8]

Job_3: M1[3] -> M3[5] -> M4[12]

The schedule is as shown:

- M1: J1:10, J3:16, J2:30
- M2: J2:7, J1:15
- M3: J2:22, J3:27
- M4: J1:27, J3:39

The mekaspan time here is 39 time-units.

Test Cases

Test case 1:

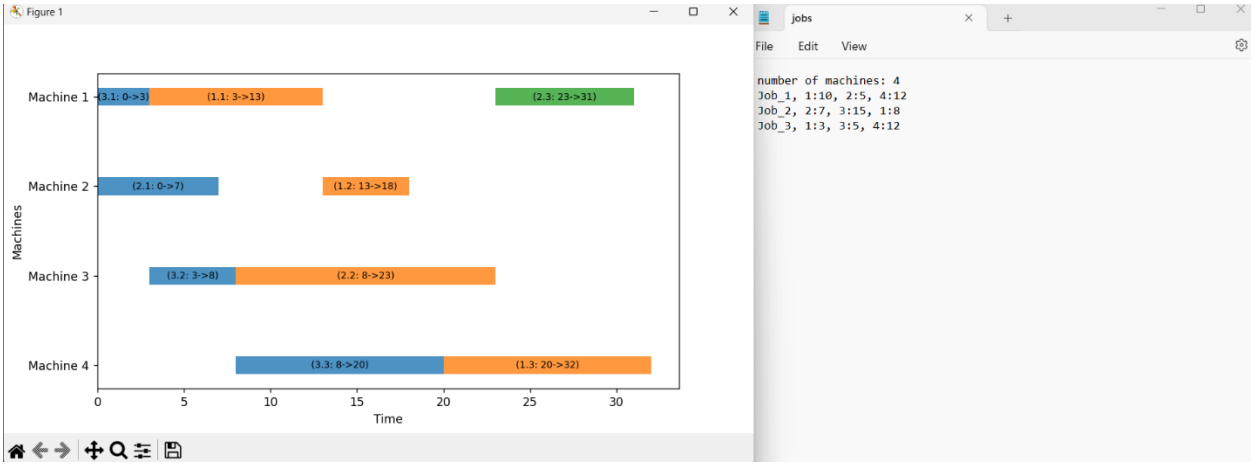

Figure 1: Test case 1

In the test case shown above, the makespan time of the solution is 32 time-units.

Test case 2:

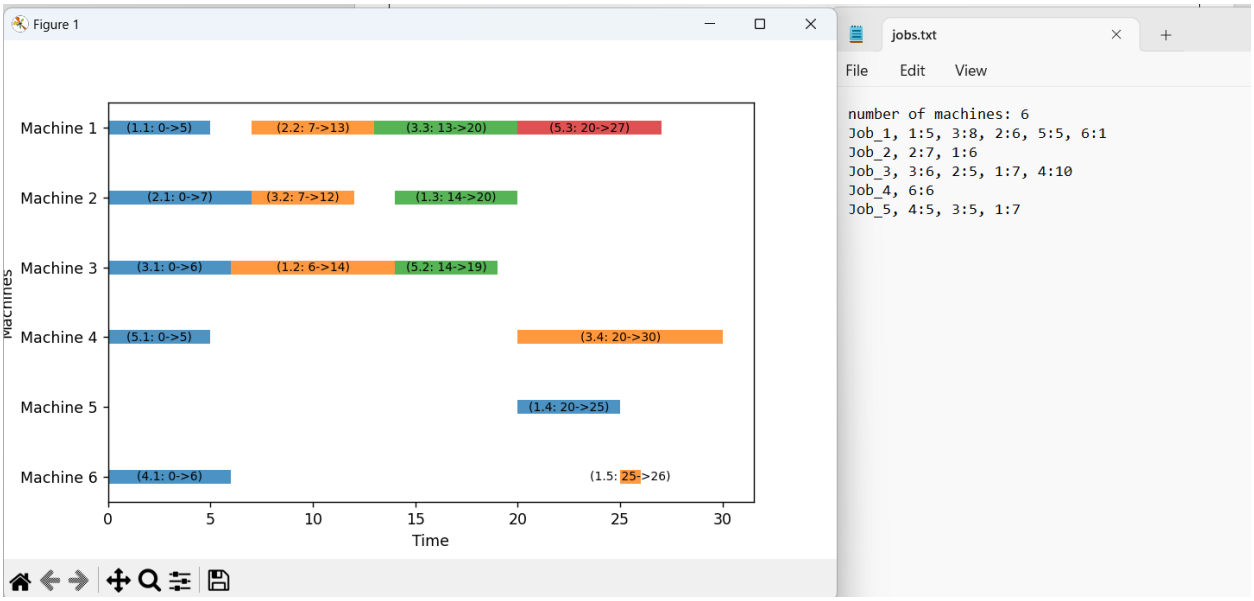

Figure 2: Test case 2

In the test case shown above, the makespan time of the solution is 30 time-units.

Test case 3:

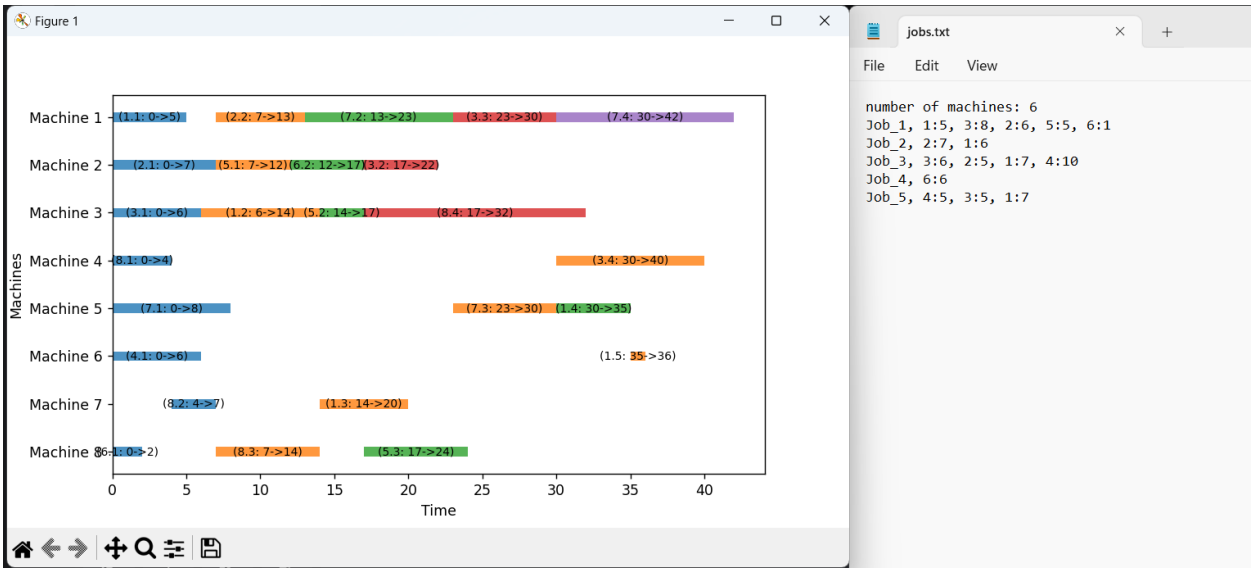

Figure 3: Test case 3

In the test case shown above, the makespan time of the solution is 42 time-units.

Appendix

Fitness function code:

```
def fitness_function(chromosome, jobs, num_of_machines):
    # Initialize completion time matrix for machines
    machine_completion_times = [0] * num_of_machines
    # Initialize completion time for each job
    job_completion_times = [0] * len(jobs)

    # determine the max num of operations between all jobs
    num_of_operations = []
    for job in jobs:
        num_of_operations.append(len(job))
    max_operations = max(num_of_operations)

    # iterate through operations in all jobs
    for operation_index in range(max_operations):
        for job_index in chromosome:
            # get the current job
            job = jobs[job_index - 1]
            if operation_index < len(job):
                # getting the required operation of the current job
                (job_id, machine, duration) = job[operation_index]
                # finding the start and end time
                start_time = max(machine_completion_times[machine - 1],
job_completion_times[job_id - 1])
                end_time = start_time + duration
                # modify machine and job completion times
                machine_completion_times[machine - 1] = end_time
                job_completion_times[job_id - 1] = end_time

    # The fitness value is the maximum completion time of all jobs
    fitness_value = max(job_completion_times)
    return fitness_value
```