Birzeit University
Department of Computer Engineering
ENCS3340: Artificial Intellegence
Project 1: Task scheduling using genetic algorithms

Karim Marayta
1211610@student.birzeit.edu

May 20, 2024

1 Abstract

This paper discusses the use of genetic algorithms for solving scheduling problems. The particular type of problems discussed involves machines, parts and operations. Each machine can process only one part at a time. Each part can be processed on one machine at a time. Each operation specifies a machine and a duration. Each part requires a certain number of operations to be done in a certain order.

2 Problem formulation

2.1 Description

The discussion is about a typical task scheduling problem with multiple uniquely skilled workers (machines), multiple parts that require multiple operations to be done on a part in a specified order.

2.2 State space

The set of all possible starting times for every operation.

2.3 Chromosome Representation

Each operation from the input needs to be done on a particular machine. Filtering operations by this property results in a double list of tasks. A task is an object that contains data about the machine, part, prerequisite task...
Each chromosome is a double list with the following meaning list[machine][task]. Variations between chromosomes happen in the order of the inner list of tasks (I will refer to it as sublist or subchromosome). Each possible permutation of elements in the sublists of tasks represents a new unique solution.

2.4 Updated State Space

Each possible permutation of the elements in the sublists of tasks for each machine. Shuffling at least one of the task lists results in an entirely new solution.
This state space is smaller that the original (infinite) state space. This was achieved by removing bad solutions that involve aimlessly waiting.

2.5 Converting the solution from the updated state space to the original state space

It is possible to transfer a solution from the updated state space into the original state space by performing a greedy simulation that would perform the tasks in the order stated in the chromosome. Greedy here refers to the fact that the simulation algorithm will run the task as soon as it can if it doesn't violate any rules stated in the description (subsection 2.1). The simulation continues until all tasks are done or until time t reaches t_{max}, where $t_{max} = \Sigma_{task} duration$ for every $task$ in $tasks$. The simulation returns the total time t_{total}, the number of finished tasks D, the number of failed tasks F. The simulation also returns the starting times of each task to be converted later into output and a Gantt chart.

2.6 Initial State

The operations are filled as tasks in the order of input to get the first list. Then this list (chromosome) gets reshuffled and saved for N times to obtain a diverse initial population (G0).

2.7 Fitness function

The fitness function takes a chromosome c and performs the simulation as stated in subsection 2.5 and gets t_{total}, D and F. Then it computes and returns a weighted sum.

$$F(c) = w_1(t_{max} - t_{total}) + w_2 D - w_3 F \tag{1}$$

, where $w_i > 0$ and t_{max} was defined earlier.

2.8 Crossover Operator

The crossover operator is a binary operator (takes two chromosomes that are called parents) and return two children. The type used in my program is a single point crossover over a randomly selected pair of subchromosomes that correspond to the same machine. The point of crossover is also random. For example take the following two parent subchromosomes:

- 02134
- 43210

.Performing a crossover at a position $n = 2$ (start counting from 0) results in the following child subchromosomes:

- 02110
- 43234

. You may have noticed that the child subchromosomes contain duplicates. To fix that I introduce a repair function that will detected missing characters and the one of the positions of the duplicated characters. Then it will randomly place the missing characters in the positions of the duplicated characters (overwriting the duplicates). For example, the result may look like this:

- 02134
- 43201

2.9 Mutation Operator

Although the repair function already preforms a kind of mutation. A specified mutation operator will reduce the chance of hitting a local minimum even more. It's a unary operator that takes a chromosome, randomly selects a subchromosome and swaps position of two random characters. For example the subchromosome:

- 43210

Swap the characters at position $n = 1$ and $n = 3$ to get:

- 41230

2.10 Path

The idea of path is not applicable for genetic algorithms (as it's a local search algorithm). I will describe here the process of evolving and obtaining new (likely better) generations. Starting with an initial population G0. The algorithm sorts it in a descending order according to the fitness function. The best S% of the population is called Survivors. The rest of G0 will be overwritten by chromosomes generated from a crossover of two survivor chromosomes or mutations of the survivors. This is the way of getting G1. The process gets repeated many times to get even better solutions.

2.11 Goal State and Goal Test

There are at least two ways to stop a genetic algorithm:

- When Kth generation is reached, the algorithm outputs the best chromosome and terminates.
- When a good enough solution is reached ($t_{total} < t_{cutoff}$, the algorithm outputs the solution and terminates.

Since there is no t_{cutoff} provided, I went with the first way.

3 Test Cases

3.1 Input file format

Saved as input.txt and the directory is provided to the code as a shell argument(Example: python main.py .test_casestest1).

```
[Number of Machines] [Number of Parts]
[Number of Operations for Part0] #Repeat for each part
[Machine number] [Duration of operation]
#Repeat the last line for multiple operations
```

Example:

```
3 3
3
0 1
1 4
2 6
3
```

```
2 2
1 4
0 1
2
1 2
2 1
```

Note that the machine and part counters begin from 0.

3.2 Output text format

The output is saved output.txt in the same directory as input.txt. The format is:

```
[Machine number] [Part number] [Start time] [End time]
#The line is repeated for each operation
```

Example:

```
0 0 0 0
0 1 10 10
1 0 2 5
1 1 6 9
1 2 0 1
2 0 6 11
2 1 0 1
2 2 2 2
```

As you can see the way to identify an operation is the machine number and part number. The algorithm works without these limitations and can process multiple operations that have these parameters the same, but outputting them effectively would require introducing another unique key into the input. This is likely to make the output less readable and the Gannt chart less clear. For demonstration purposes, I left it as is.

3.3 Gannt Chart

The Gannt chart provides a much easier way to read the output. You can find it in the same directory. The vertical axis represents operations with their identifiers (P#M#), where # are numbers. P and M stand for part and machine respectively.

3.4 Best score vs generation plot

There is a figure in the same output directory called hill_climbing.txt. This figure allows the user to analyze whether it's worth to increase the number of generations by noticing if the best values increases in the last few generations or if the solution is the best it can find (the best score is constant). The last case could be a local minimum in certain cases and it may be worth changing the seed of random generation.

3.5 Test Case 1

The same input.txt and output.txt as in the examples above.

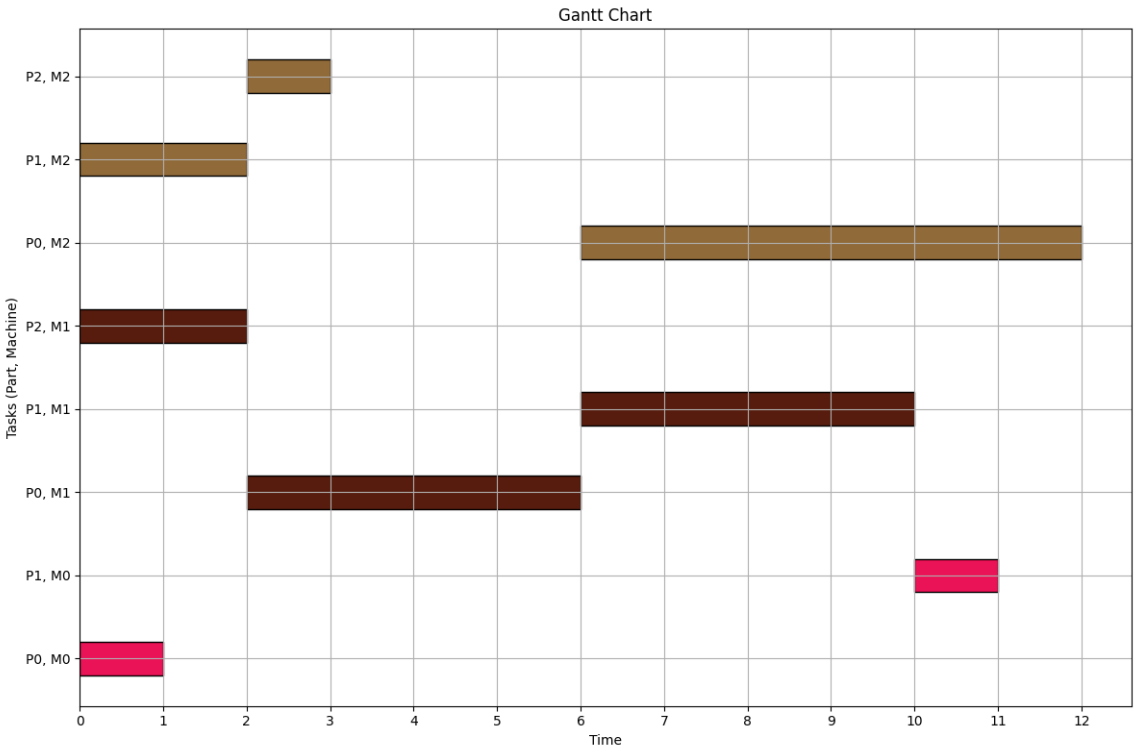

Figure 1: Gantt Chart for test case 1.

3.6 Other test cases

The report already took 6 pages. Each test case would need a separate page for it. I have 6 of them with their results in the zip archive. I don't want to make this report too long. So I invite you to check them if that is required for grading.

Note: test case 5 is testing the guarding conditions of the code and gets catched. There is no output other than a message to stdout for the user to read in such cases.