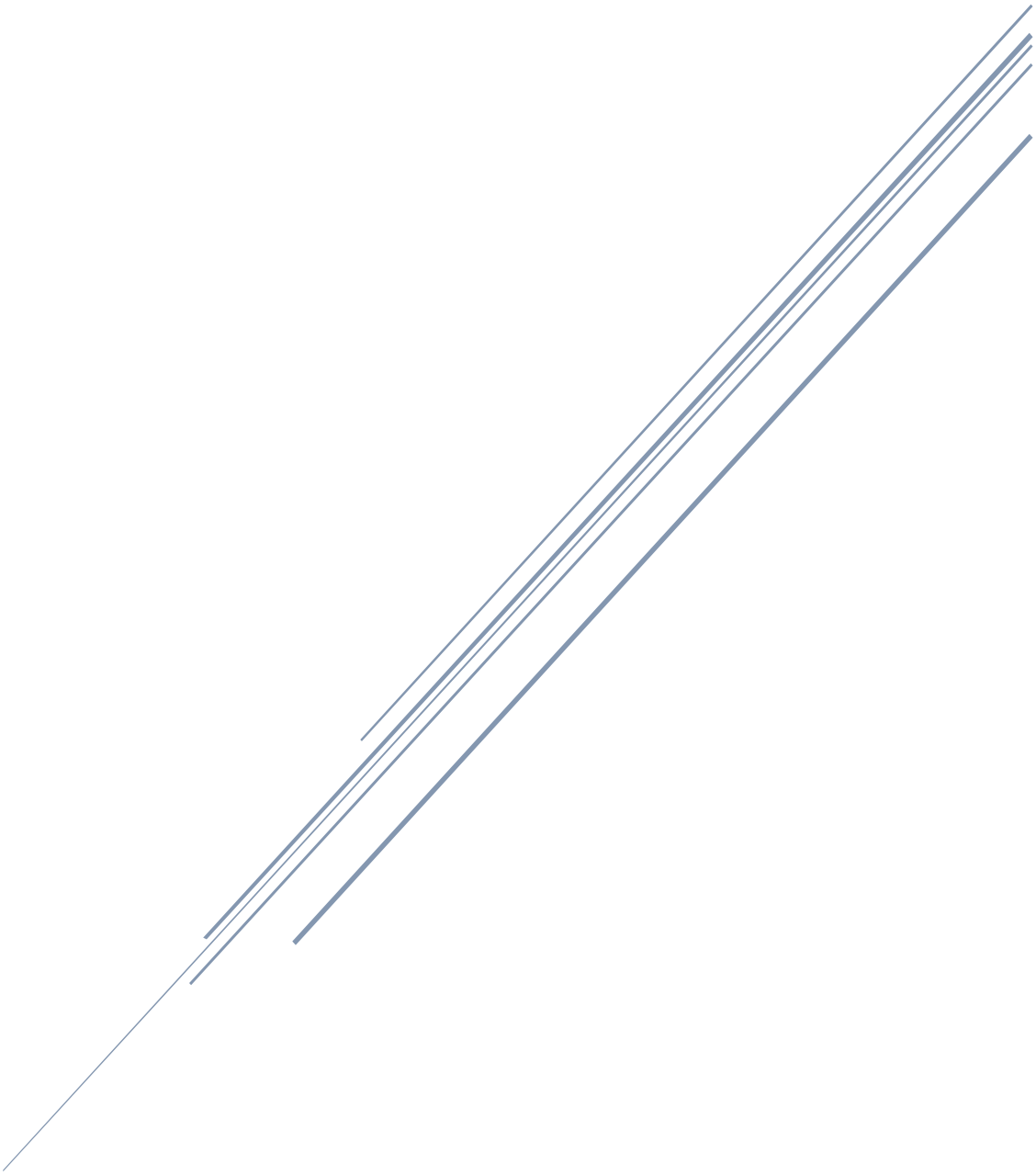
HOMEWORK3 – GENETIC ALGORITHM ON FLOW SHOP SCHEDULING PROBLEM

IE 517 - Spring17



Burak Suyunu
2012400156

Important Note: This report is a summary of my detailed work. So, please check the Jupyter Notebook that I have created in https://github.com/suyunu/GA-FSS repository. You can directly view the code from here: http://nbviewer.jupyter.org/github/suyunu/GA-FSS/blob/master/ga-fss.ipynb. You can find the detailed process of the code with heavy commenting.

Introduction

In this project, we tried to solve Flow Shop Scheduling Problem (FSSP) with Genetic Algorithm (GA). Before I start doing anything on the problem, I made a literature survey and found these 2 papers:

- Murata, Tadahiko, Hisao Ishibuchi, and Hideo Tanaka. "Genetic algorithms for flowshop scheduling problems." Computers & Industrial Engineering 30.4 (1996): 1061-1071
- Reeves, Colin R. "A genetic algorithm for flowshop sequencing." Computers & operations research 22.1 (1995): 5-13.

These 2 papers have done lots of good optimization tests for the parameters and obtained good results. So, I took pieces' form both papers:

- Murata et al.
    - General Structure
    - Crossover
    - Mutation
- Reeves
    - Selection

Flow Shop Scheduling

There are n machines and m jobs. Each job contains exactly n operations. The i-th operation of the job must be executed on the i-th machine. No machine can perform more than one operation simultaneously. For each operation of each job, execution time is specified.

Operations within one job must be performed in the specified order. The first operation gets executed on the first machine, then (as the first operation is finished) the second operation on the second machine, and so until the n-th operation. Jobs can be executed in any order, however. Problem definition implies that this job order is exactly the same for each machine. The problem is to determine the optimal such arrangement, i.e. the one with the shortest possible total job execution makespan.

Solution Representation

I used a simple permutation representation.

The string "ABCDEF" represents a job sequence where "Job A" is processed first, then "Job B" is processed, and so on. If the string "ABCADE" is generated by genetic operators such as crossover and mutation, this string is not a feasible solution of the flow shop scheduling problem because the job "A" appears twice in the string and the job "F" does not appear. Therefore, the string used in the flow shop scheduling problem should be the permutation of given jobs.

Genetic Algorithm

In this part I will explain the genetic operators such as crossover and mutation as well as the selection mechanism for the flow shop scheduling problem.

Pseudocode

1. **(Initialization)** Randomly generate an initial population P_1 of N_{pop} strings (i,e., N_{pop} solutions).
2. **(Selection)** Select N_{pop} pairs of strings from a current population according to the selection probability.
3. **(Crossover)** Apply crossover operator to each of the selected pairs in Step 2 to generate N_{pop} solutions with the crossover probability P_c. If the crossover operator is not applied to the selected pair, one of the selected solutions remains as a new string.
4. **(Mutation)** Apply mutation operator to each of the generated N_{pop} strings with the mutation probability P_m (we assign the mutation probability not to each bit but to each string).
5. **(Elitist Update)** Randomly remove one string from the current population and add the best string in the previous population to the current one.
6. **(Termination)** If a prespecified stopping condition is satisfied, stop this algorithm. Otherwise, return to Step 2

Initialization

I randomly generated N_{pop} number of solutions for the initial population. Surprisingly after lots of test, $N_{pop} = 3$ gave the best results.

Selection

The usual method (for maximization) is to measure relative fitness as the ratio of the value of a given chromosome to the population mean. However, in minimization problems, we need to modify this so

that low-valued chromosomes are the "good" ones. One approach is to define fitness as $(v_{max} - v)$, but of course we are unlikely to know what v_{max} is. We could use the largest value found so far as a surrogate for v_{max}, but in practice this method was not very good at distinguishing between chromosomes. In essence, if the population consists of many chromosomes whose fitness values are relatively close, the resolution between "good" and "bad" ones is not of a high order. This approach was therefore abandoned, in favor of a simple ranking mechanism. Sorting the chromosomes is a simple task, and updating the list is also straightforward. The selection of parents was then made in accordance with the probability distribution

$$p(k) = \frac{2k}{M(M+1)}$$

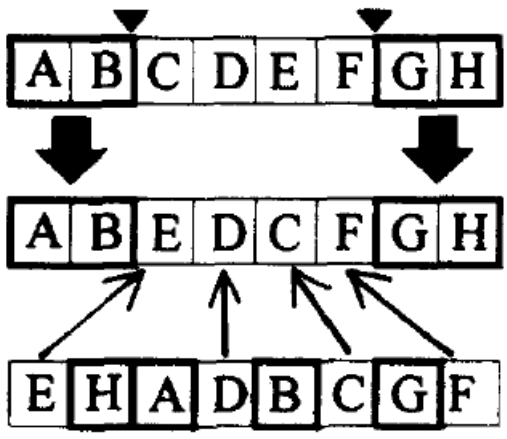
where k is the k^{th} chromosome in ascending order of fitness (i.e. descending order of makespan). This implies that the median value has a chance of $\frac{1}{M}$ of being selected, while the M^{th} (the fittest) has a chance of $\frac{2}{M+1}$, roughly twice that of the median.

We choose N_{pop} pairs of strings from the current population according to this selection probability. So, at the end we will have N_{pop} number of children reproduced from those selected pairs.

Crossover

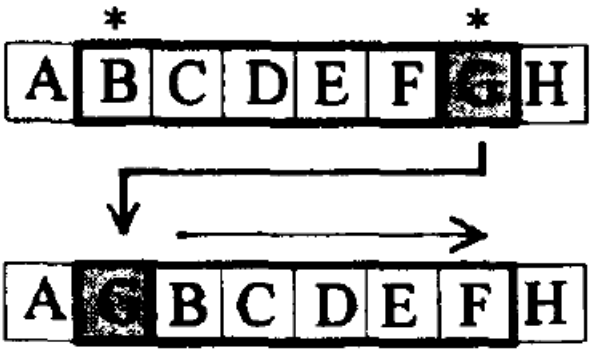
According to Murata et al. Two-point crossover gives overall better results than other crossover techniques. Also, having a crossover probability $P_c = 1$ gives the best results. So, we applied crossover to every parent.

In Two-point crossover, two points are randomly selected for dividing one parent. The jobs outside the selected two points are always inherited from one parent to the child, and the other jobs are placed in the order of their appearance in the other parent.



Mutation

Again, according to Murata et al. Shift change gives overall better results than other mutation techniques. Also, having a mutation probability $P_m = 1$ gives the best results. So, we applied mutation to every child. In shift change mutation, a job at one position is removed and put at another position. Then all other jobs shifted accordingly. The two positions are randomly selected.



Elitist Update Strategy

We applied all the operations to N_{pop} pairs of strings (parents). So we ended up with N_{pop} number of children. As a last step we randomly remove one string from the current population and add the best string in the previous population to the current one. Then we continue our processes with this newly generated population.

Termination

Total number of evaluations/generations used as a stopping condition. The total number of generations was specifed as to be inversely proportional to the population size N_{pop}. For example, when the total number of evaluations was 10000, the total number of generations was specified as $\frac{10000}{N_{pop}}$. I choose the number of generation to stop the program as 10000

Summary

- Population size: $N_{pop} = 3$
- Crossover probability: $P_c = 1$
- Crossover: Two-point crossover
- Mutation probability: $P_m = 1$
- Mutation: Shift change mutation
- Stopping Condition: Generation $= 10000$

Results

	Instance: FSSP car1			
Solution (sequence of jobs processed by the machines)	Obj. val. of the best chromosome	Avg. obj. val. of all chromosomes	% Gap of the best chromosome	CPU Time (s)
[7, 6, 4, 2, 5, 1, 0, 3]	4534	5042.67	0.00	21.75

	Instance: FSSP reC05			
Solution (sequence of jobs processed by the machines)	Obj. val. of the best chromosome	Avg. obj. val. of all chromosomes	% Gap of the best chromosome	CPU Time (s)
[11, 2, 4, 6, 12, 9, 10, 8, 1, 7, 5, 0, 3]	920	1003.67	0.00	43.08

	Instance: FSSP reC09			
Solution (sequence of jobs processed by the machines)	Obj. val. of the best chromosome	Avg. obj. val. of all chromosomes	% Gap of the best chromosome	CPU Time (s)
[3, 15, 11, 13, 14, 8, 17, 7, 0, 12, 9, 19, 5, 16, 18, 6, 4, 10, 2, 1]	1302	1339.67	0.00	62.60

Code

Importing required libraries

```
import numpy as np
import math
import time
import random
import itertools
import queue
```

Reading data

```
# Dataset number. 1, 2 or 3
dataset = "2"

if dataset == "1":
   optimalObjective = 4534
elif dataset == "2":
   optimalObjective = 920
else:
   optimalObjective = 1302

filename = "data" + dataset + ".txt"
f = open(filename, 'r')
l = f.readline().split()

# number of jobs
n = int(l[0])

# number of machines
m = int(l[1])

# ith job's processing time at jth machine
cost = []

for i in range(n):
   temp = []
   for j in range(m):
      temp.append(0)
   cost.append(temp)

for i in range(n):
   line = f.readline().split()
   for j in range(int(len(line)/2)):
      cost[i][j] = int(line[2*j+1])

f.close()
```

Genetic Algorithm Operators and Helper Functions

```
def initialization(Npop):
    pop = []
    for i in range(Npop):
        p = list(np.random.permutation(n))
        while p in pop:
            p = list(np.random.permutation(n))
        pop.append(p)

    return pop

def calculateObj(sol):
    qTime = queue.PriorityQueue()

    qMachines = []
    for i in range(m):
        qMachines.append(queue.Queue())

    for i in range(n):
        qMachines[0].put(sol[i])

    busyMachines = []
    for i in range(m):
        busyMachines.append(False)

    time = 0

    job = qMachines[0].get()
    qTime.put((time+cost[job][0], 0, job))
    busyMachines[0] = True

    while True:
        time, mach, job = qTime.get()
        if job == sol[n-1] and mach == m-1:
            break
        busyMachines[mach] = False
        if not qMachines[mach].empty():
                j = qMachines[mach].get()
                qTime.put((time+cost[j][mach], mach, j))
                busyMachines[mach] = True
        if mach < m-1:
            if busyMachines[mach+1] == False:
                qTime.put((time+cost[job][mach+1], mach+1, job))
                busyMachines[mach+1] = True
            else:
                qMachines[mach+1].put(job)

    return time


def selection(pop):
```

```
    popObj = []
    for i in range(len(pop)):
        popObj.append([calculateObj(pop[i]), i])

    popObj.sort()

    distr = []
    distrInd = []

    for i in range(len(pop)):
        distrInd.append(popObj[i][1])
        prob = (2*(i+1)) / (len(pop) * (len(pop)+1))
        distr.append(prob)

    parents = []
    for i in range(len(pop)):
        parents.append(list(np.random.choice(distrInd, 2, p=distr)))

    return parents

def crossover(parents):
    pos = list(np.random.permutation(np.arange(n-1)+1)[:2])

    if pos[0] > pos[1]:
        t = pos[0]
        pos[0] = pos[1]
        pos[1] = t

    child = list(parents[0])

    for i in range(pos[0], pos[1]):
        child[i] = -1

    p = -1
    for i in range(pos[0], pos[1]):
        while True:
            p = p + 1
            if parents[1][p] not in child:
                child[i] = parents[1][p]
                break

    return child


def mutation(sol):
    pos = list(np.random.permutation(np.arange(n))[:2])

    if pos[0] > pos[1]:
        t = pos[0]
        pos[0] = pos[1]
        pos[1] = t
```

```
    remJob = sol[pos[1]]

    for i in range(pos[1], pos[0], -1):
        sol[i] = sol[i-1]

    sol[pos[0]] = remJob

    return sol

def elitistUpdate(oldPop, newPop):
    bestSolInd = 0
    bestSol = calculateObj(oldPop[0])

    for i in range(1, len(oldPop)):
        tempObj = calculateObj(oldPop[i])
        if tempObj < bestSol:
            bestSol = tempObj
            bestSolInd = i

    rndInd = random.randint(0,len(newPop)-1)

    newPop[rndInd] = oldPop[bestSolInd]

    return newPop

# Returns best solution's index number, best solution's objective value and average objective value of
the given population.
def findBestSolution(pop):
    bestObj = calculateObj(pop[0])
    avgObj = bestObj
    bestInd = 0
    for i in range(1, len(pop)):
        tObj = calculateObj(pop[i])
        avgObj = avgObj + tObj
        if tObj < bestObj:
            bestObj = tObj
            bestInd = i

    return bestInd, bestObj, avgObj/len(pop)
```

Main Solver and Results

```
# Number of population
Npop = 3
# Probability of crossover
Pc = 1.0
# Probability of mutation
Pm = 1.0
# Stopping number for generation
stopGeneration = 10000
```

```
# Start Timer
t1 = time.clock()

# Creating the initial population
population = initialization(Npop)

# Run the algorithm for 'stopGeneration' times generation
for i in range(stopGeneration):
   # Selecting parents
   parents = selection(population)
   childs = []

   # Apply crossover
   for p in parents:
      r = random.random()
      if r < Pc:
         childs.append(crossover([population[p[0]], population[p[1]]]))
      else:
         if r < 0.5:
            childs.append(population[p[0]])
         else:
            childs.append(population[p[1]])

   # Apply mutation
   for c in childs:
      r = random.random()
      if r < Pm:
         c = mutation(c)

   # Update the population
   population = elitistUpdate(population, childs)

   #print(population)
   #print(findBestSolution(population))

# Stop Timer
t2 = time.clock()

# Results Time

bestSol, bestObj, avgObj = findBestSolution(population)

print("Population:")
print(population)
print()

print("Solution:")
print(population[bestSol])
print()

print("Objective Value:")
print(bestObj)
```

```
print()

print("Average Objective Value of Population:")
print("%.2f" %avgObj)
print()

print("%Gap:")
G = 100 * (bestObj-optimalObjective) / optimalObjective
print("%.2f" %G)
print()

print("CPU Time (s)")
timePassed = (t2-t1)
print("%.2f" %timePassed)
```