Ryan Nathaniel B. Resoles
CMSC 170 ST-1L

Final Project Report

Abstract

Genetic Algorithm, a search heuristic inspired by Charles Darwin is a concept that is currently being applied to computer science problems to come up with optimal (or near optimal) solutions. The process of natural selection, which involves selecting the fittest individuals for reproduction from a population (Mallawaarachchi, 2017) works similarly in genetic algorithm. Solutions in genetic algorithm are created into a pool of chromosomes; and only those with better solutions are retained in the pool. For the Aircraft Landing Scheduling Problem (ALSP), genetic algorithm has been used to come up with the near optimal solution.

Introduction

The machine used to run the genetic algorithm trials runs on an Intel Core i5-8250U CPU @1.60GHz, 4 core(s) and 8 logical processors with a 16GiB RAM using Windows operating system, 256gb SSD and 1TB HDD of disk capacity.

Python was used to implement the program. The trials aim to get the near optimal solution to the ALSP by coming up with the scheduling of aircrafts that inflict the least amount of penalty.

Objectives

The ALSP used a heuristic approach in the form of genetic algorithm. The schedules are treated as solutions to the problem, each having several attributes that contribute to it being classified as an optimal solution or not. The specific objectives of the project are listed below.

1. Create N aircrafts from the input file with the following attributes:
    - $\mathbf{A_i}$ = arrival time
    - $\mathbf{E_i}$ = earliest possible time
    - $\mathbf{T_i}$ = target time
    - $\mathbf{L_i}$ = latest possible time
    - $\mathbf{EP_i}$ = early schedule penalty
    - $\mathbf{LP_i}$ = late schedule penalty

2. Create M schedules of the N aircrafts by shuffling the aircrafts in a random ordering.

3. Perform mutation on the current pool of solutions. Invalid solutions may be transformed into a valid solution after mutation.

4. Remove invalid solutions from the pool of solutions.

5. Perform selection on the M number of schedules using a roulette wheel implementation. Better solutions are more likely to get selected.

6. Use order crossover within the mating population. Order crossover selects only a portion of one parent's chromosome and copies the genes of the other parent that is not yet present in the current chromosome.

7. Recompute fitness values and chances to be picked for roulette wheel selection.

8. Test the limit of the machine.

9. Repeat from step 3 until the iteration reaches MAX_ITERATION.

Methodology

The aforementioned specifications were used to run the Genetic Algorithm implementation of the ALSP. The following functions and classes were used to come up with the near-optimal solution:

- The class Plane takes in the parameters *plane number*, *arrival time*, *earliest time*, *target time*, *latest time*, *early penalty*, and *late penalty*. The following attributes are important to note the ordering of the planes and the computation of the total penalty of a schedule.

- The class Schedule takes in a list of the Planes ordered randomly and the separation value of each plane. Additionally, it has the *fitness* attribute which determines how good the scheduling is, the *chance* attribute to determine its chance of getting picked in the roulette wheel implementation, and the *valid* attribute that determines whether the schedule is valid or not. The class also has the following methods:
  - `calculate_fitness()` : calculates the fitness value of the solution.
  - `mutate()` : mutates the current schedule by swapping the ordering of two random aircrafts.

- The `compute_chance()` function computes the chance of each schedule being selected.

- The `selection()` function selects *n* number of pairs that will undergo reproduction and stores it in a list called *mating_population*.

- The `order_crossover()` function performs order crossover between the two schedules. The image below illustrates how order crossover functions, as explained in the objectives section of this paper.

parent 1	2	1	3	5	8	6	4	9	7	0
parent 2	0	1	2	3	4	5	6	7	8	9
offspring	0	1	2	5	8	6	4	3	7	9

Figure 1. Order Crossover between two schedules, resulting in one offspring.

- The `addOffsprings()` adds the offsprings generated via order crossover in the pool of solutions. The pool of solutions has a limited size, therefore replacing less fit solutions with fitter solutions in the long run.

- The `readFile()` function reads the input file and generates an adjacency matrix of each aircraft with their corresponding separation time. It also generates the values needed for the *Plane* class.

- The `main()` function is the driver function used to create objects from the classes and generate all functions listed above.

Results and Discussion

The MAX_ITERATION count was set to 100, 200, 300, 400, and 500, respectively. This generated different results for the input files.

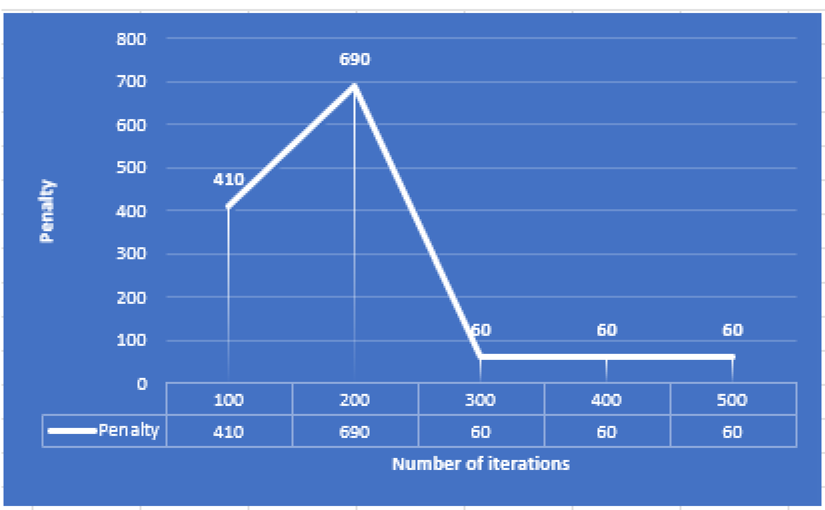


Chart 1. Total penalty per iteration from ALSP01.txt.

As illustrated in the chart above, the near optimal solution of only 60 penalty was reached at approximately 300 iterations. The same penalty was computed at 400 and 500 iterations. The aircraft scheduling outputted was = [2,3,4,6,5,7,8,0,1,9]. A downward trend can be observed from the chart.

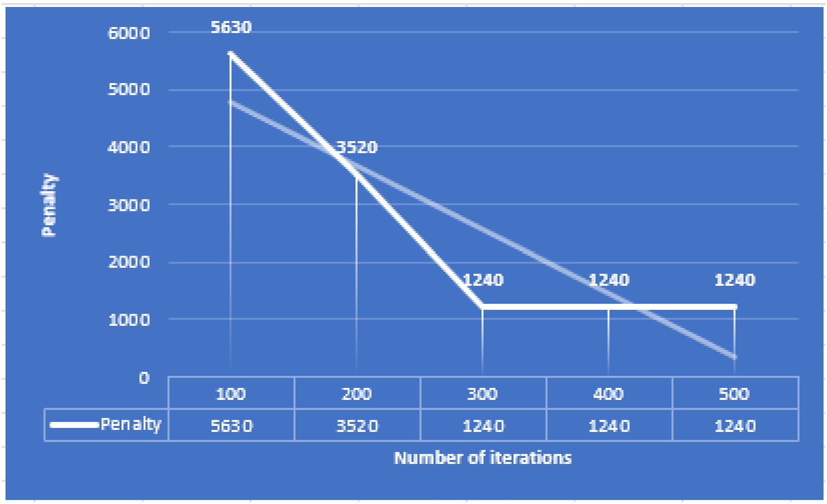


Chart 2. Total penalty per iteration from ALSP02.txt.

As illustrated in the chart above, the near optimal solution of 1240 penalty was reached at approximately 300 iterations. The same penalty was computed at 400 and 500 iterations. A downward trend can be observed from the chart. The aircraft scheduling outputted was = [2, 3, 4, 7, 6, 5, 8, 9, 12, 13, 0, 1, 10, 11, 14]. The same trend was observed from ALSP01.txt file.

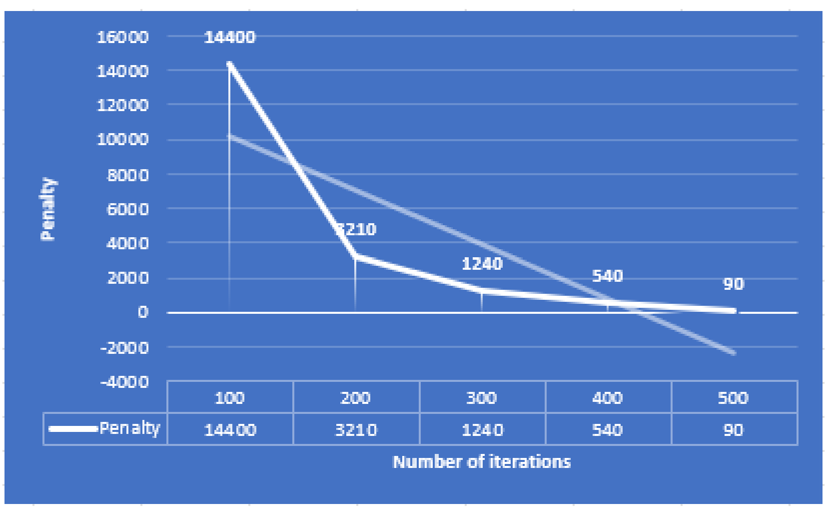


Chart 3. Total penalty per iteration from ALSP03.txt.

As illustrated in the chart above, the near optimal solution of only 90 penalty was reached at approximately 500 iterations. A downward trend can be observed from the chart. The aircraft scheduling outputted was = [2, 3, 4, 7, 6, 5, 8, 9, 12, 13, 0, 1, 10, 11, 14]. The same trend was observed from the previous input files.

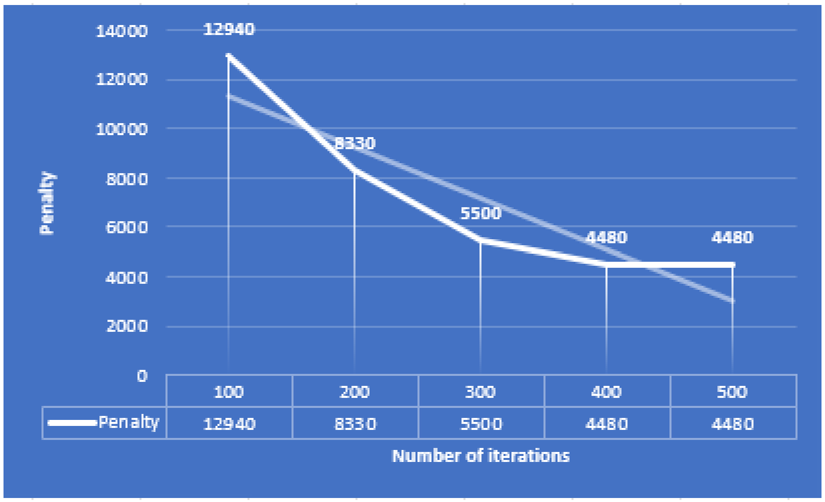


Chart 4. Total penalty per iteration from ALSP04.txt.

As illustrated in the chart above, the near optimal solution of 4480 penalty was reached at approximately 400 iterations. The same penalty was obtained at 500 iterations. A downward trend can be observed from the chart. The aircraft scheduling outputted was = [0, 1, 4, 8, 7, 6, 12, 5, 15, 11, 16, 18, 14, 17, 2, 10, 9, 13, 3, 19]. The same trend was observed from the previous input files.

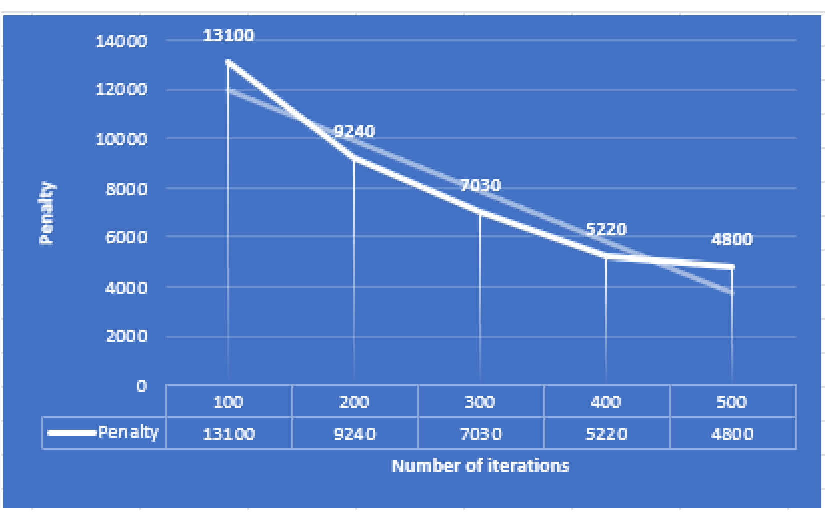
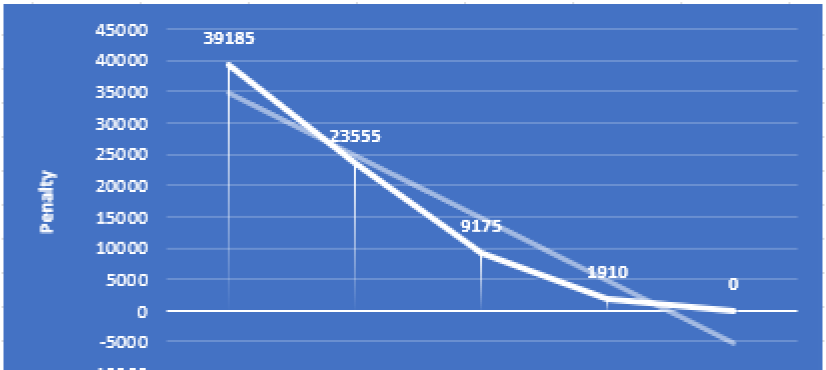


Chart 5. Total penalty per iteration from ALSP05.txt.

As illustrated in the chart above, the near optimal solution of 4800 penalty was reached at approximately 500 iterations. A downward trend can be observed from the chart. The aircraft scheduling outputted was = [2, 3, 4, 6, 8, 7, 5, 9, 13, 12, 17, 18, 16, 19, 0, 14, 10, 15, 11, 1]. The same trend was observed from the previous input files.



	100	200	300	400	500
Penalty	39185	23555	9175	1910	0

Chart 6. Total penalty per iteration from ALSP08.txt.

As illustrated in the chart above, the optimal solution of 0 penalty (no penalty) was reached at approximately 500 iterations. A downward trend can be observed from the chart. The aircraft scheduling outputted was =[0, 9, 5, 35, 7, 3, 11, 14, 8, 18, 10, 2, 23, 31, 43, 47, 19, 1, 6, 22, 42, 13, 34, 4, 28, 24, 46, 41, 33, 25, 16, 17, 32, 29, 49, 12, 21, 45, 15, 30, 40, 39, 27, 26, 44, 37, 48, 20, 38, 36]. The same trend was observed from the previous input files.

The files ALSP06.txt, ALSP07.txt, ALSP09-13.txt generated no results. This is possibly because of the method used in implementing the Genetic Algorithm solution for the ALSP. The method used is as follows:

1. The randomly shuffled solution starts at the first aircraft's target landing time. The succeeding aircrafts tries to go for their target landing time as well to minimize penalties incurred.
2. A generated schedule may be invalid if the current time exceeds the current aircraft's latest landing time possible. This deems the generated schedule invalid.
3. There is a time limit for trying to generate a schedule. For the input files mentioned above, the algorithm was unable to generate a population within the time limit (30 seconds). Further optimization and improvements of the current source code (see appendices) may yield better results in the future.

Conclusion

In conclusion, using Genetic Algorithm to solve the ALSP problem proved to be a valid approach. Due to the heuristic nature of the solution, we get fluctuating results each time we run the Python script. A heuristic approach is easier to implement as it considers randomness as an element of the algorithm, but going for a more deterministic approach could be better.

At ALSP08.txt file, we were able to get the optimal solution with 0 penalty. However, running the script may also result in worse solutions. In some cases, the population even fails to produce better solutions. As evident in the results above, the algorithm even failed to generate an initial population at some input files.

We can therefore conclude that while the Genetic Algorithm is a possible solution to the ALSP problem, it is not sufficient and not even close to being the best solution. Other algorithms such as the Ant Colony Optimization (ACO) or a hybrid of Genetic Algorithm and ACO may yield better solutions.

References

Mallawaarachchi, V. (2017, July 8). Introduction to genetic algorithms - including example code. Medium. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

APPENDIX A

Source code for the Aircraft Landing Scheduling Problem using Genetic Algorithm

```
# Ryan Nathaniel B. Resoles
# CMSC 170 ST-1L Final Project
# Aircraft Landing Scheduling Problem (ALSP) using Genetic Algorithm

import random
import numpy as np
import math
import sys
import time

INITIAL_POP = 10
MAXPOP = 1000
PAIR_COUNT = 4

# plane object
class Plane:
    def __init__(self, no, arrival, earliest, target, latest, ep, lp):
        self.plane_no = no
        self.arrival = arrival
        self.earliest = earliest
        self.target = target
        self.latest = latest
        self.early_penalty = ep
        self.late_penalty = lp

class Schedule:
    def __init__(self, sched, sep):
        self.sched = sched
        self.separation = sep
        self.placeholder = self.calculate_fitness()
        self.fitness = self.placeholder[0]
        self.chance = 0
        self.finish = self.placeholder[1]
        self.valid = self.placeholder[2]

    # calculates fitness based on penalties (computes total penalty)
    # penalty is inversely proportional to fitness
    def calculate_fitness(self):
        penalty = 0
        time = self.sched[0].target
        valid = 1
```

```
        for i in range(1, len(self.sched)):
            time += self.separation[i]
            temp_penalty = 0

            # invalid scheduling if current time > latest time
            if self.sched[i].latest < time:
                valid = 0
                break

            # late schedule
            if self.sched[i].target < time and self.sched[i].latest >= time:
                penalty += ((time - self.sched[i].target)*self.sched[i].late_penalty)
                continue

            # early schedule
            if self.sched[i].target < time:
                if self.sched[i].earliest < time:
                    penalty += ((self.sched[i].target -
self.sched[i].earliest)*self.sched[i].early_penalty)
                else:
                    penalty += ((self.sched[i].target - time)*self.sched[i].early_penalty)
                time = self.sched[i].target


        return [penalty, time, valid]

    def mutate(self, newSched):
        self.sched = newSched
        self.placeholder = self.calculate_fitness()
        self.fitness = self.placeholder[0]
        self.finish = self.placeholder[1]
        self.valid = self.placeholder[2]

def readFile(filename):
    fileReader = open(filename, "r")
    adj_matrix = []
    plane_vals = []

    nplanes = 0
    ctr = 0
    ctr2= 0

    flag = 0
    line_count1 = 0
```

```
    line_count2 = 0

    adder = []

    for i in fileReader:
        if i=='\n': continue
        i = i[1:].split(" ")
        if i[-1] == "\n": i=i[:-1]
        else: i[-1] = i[-1][:-1]

        if ctr2==0:
            nplanes = int(i[0])
            ctr2+=1
            continue

        if flag==0:
            line = [float(x) for x in i]
            line_count1 += len(line)
            plane_vals.append(line)
            ctr+=1

        if flag==1:
            line = [int(x) for x in i]
            line_count2 += len(line)
            adder += line
            ctr+=1

        if line_count1==6:
            flag=1
            line_count1=0

        if line_count2==nplanes:
            flag=0
            line_count2=0
            adj_matrix.append(adder)
            adder = []

    fileReader.close()

    return [nplanes, plane_vals, adj_matrix]

# computes chance of getting selected for roulette wheel selection
def compute_chance(sched_list):
    chance = sum([x.fitness for x in sched_list])
    total_chance = 0
```

```
    for i in sched_list:
        total_chance += i.fitness/chance
        i.chance = total_chance

    return chance

def mutation(schedule):
    x = random.randint(0,len(schedule.sched)-2)
    y = random.randint(0,len(schedule.sched)-2)

    if x>y:
        x, y = y, x

    new_path = schedule.sched[:x] + [schedule.sched[y]] + schedule.sched[x+1:y] + [schedule.sched[x]] +
schedule.sched[y+1:]
    schedule.mutate(new_path)

def selection(sched_list):
    pairs = []

    for i in range(PAIR_COUNT):
        temp = np.random.uniform(0,1,2)

        ind1, ind2 = 0, 0
        for j in range(1,len(sched_list)):
            if temp[0] > sched_list[j].chance:
                ind1 = j
            if temp[1] > sched_list[j].chance:
                ind2 = j
        pairs.append((ind1,ind2))

    return pairs

# crossover using order crossover, swaps every bit
# except bits appearing in selected portion
def order_crossover(mating_population, sched_list, plane_count):
    offsprings_holder = []
    os = []

    temp = random.sample(range(1, plane_count), 2)
    if temp[0] > temp[1]:
        t = temp[0]
        temp[0] = temp[1]
        temp[1] = t
```

```
    for i in mating_population:
        os = ([0]*temp[0]) + [*sched_list[i[0]].sched[temp[0]:temp[1]]] + ([0]*(plane_count-temp[1]))

        for j in range(len(os)):
            for k in sched_list[i[1]].sched:
                if os[j]==0 and k not in os:
                    os[j] = k
                    break

        offsprings_holder.append(os)

    return offsprings_holder

def get_separation(sched, adj_matrix, plane_count):
    sep_indiv = []
    prev = sched[0].plane_no
    for i in range(0,plane_count):
        curr = sched[i].plane_no
        sep_indiv.append(adj_matrix[prev][curr])
        prev = curr

    return sep_indiv

def addOffsprings(offsprings, sched_list, adj_matrix, plane_count):
    for i in offsprings:
        if len(sched_list)<MAXPOP: sched_list.append(Schedule(i, get_separation(i, adj_matrix,
plane_count)))
        else: sched_list[sched_list.index(max(sched_list, key=lambda x:x.fitness))] = Schedule(i,
get_separation(i, adj_matrix, plane_count))



###########################################################
######################## MAIN CODE ########################
###########################################################
if __name__=="__main__":
    plane_list = []
    sched_list = []
    content_holder = readFile('ALSP01.txt')
    plane_count = int(content_holder[0])
    plane_vals = content_holder[1]
    adj_matrix = content_holder[2]

    for i in range(plane_count):
```

```
        plane_list.append(Plane((i), plane_vals[i][0], plane_vals[i][1], plane_vals[i][2],
plane_vals[i][3], plane_vals[i][4], plane_vals[i][5]))

    while True:
        num_iter = int(input("Number of iterations (should be above or equal to 3): "))
        if num_iter > 2: break

    total_fitness=start=generation = 0
    duration = 30
    for i in range(num_iter-1):
        # increments pair count every 5 generations
        if i%5==0: PAIR_COUNT+=1

        # creates initial population
        if start==0:
            start_time = time.time()

            while len(sched_list)==0:
                elapsed_time = time.time() - start_time
                if elapsed_time >= duration:
                    print("Max time reached. Valid solution not found.")
                    sys.exit(1)
                for i in range(INITIAL_POP):
                    order = random.sample(range(0,plane_count), plane_count)
                    #print(order)
                    sched_indiv = []
                    sep_indiv = []
                    prev = order[0]
                    for j in range(0,plane_count):
                        curr = next(x for x in plane_list if x.plane_no==order[j])
                        sched_indiv.append(curr)
                        sep_indiv.append(adj_matrix[prev][curr.plane_no])
                        prev = curr.plane_no

                    sched_list.append(Schedule(sched_indiv, sep_indiv))

                    # mutation starts [TO BE IMPLEMENTED]
                    for i in sched_list:
                        mut_rate = random.randint(1,100)
                        if mut_rate==1:
                            mutation(i)

                sched_list = [x for x in sched_list if x.valid==1]
                compute_chance(sched_list)
```

```
            #print([x.valid for x in sched_list])


        total_chance = compute_chance(sched_list)

        if start==0:
            ind = min(sched_list, key=lambda x:x.fitness)
            print(f'GEN#{generation+1}: path:{[x.plane_no for x in ind.sched]} | fitness:{ind.fitness} |
validity:{ind.valid}')
            generation+=1
            start+=1

        # selection/crossover begins
        mating_population = selection(sched_list)
        offsprings = order_crossover(mating_population, sched_list, plane_count)
        addOffsprings(offsprings, sched_list, adj_matrix, plane_count)
        sched_list = [x for x in sched_list if x.valid==1]
        compute_chance(sched_list)
        generation += 1

        ind = min(sched_list, key=lambda x:x.fitness)
        print(f'GEN#{generation} | Path:{[x.plane_no for x in ind.sched]} | Penalty:{ind.fitness} | Time
Taken:{ind.finish} | Validity:{ind.valid}')
```