Shaikh Shawon Arefin Shimon (skshimon)

Question 1:
For the problem that you submitted, explain where you got it or how you constructed it. Why do you think it is a good test problem? Do you think it is/will be challenging? Why or why not? Provide any background on the type of problem if you have it.

Answer:

a) **How the problem was constructed:**

The problem instance was generated using a problem instance generator code that I wrote in python. In this code, I took the following parameter:

Number of courses (> 100)

Number of students (> 200)

Minimum number of courses that each student can take (>=2)

Maximum number of courses that each student can take.

For each student, a random number was selected between the minimum and the maximum number of courses the student can take. Let's say that the minimum number of courses = 2 and the maximum number of courses the student can take is 5. Then a student s_i will have between 2 to 5 courses.

For the next part of the discussion, we will assume that our

Minimum number of courses = 2 and

Maximum number of courses the student can take is 5

To ensure that there are no courses with 0 students, we employ the following strategy, assuming a student s_i can have x_i number of courses ($2 <= x_i <= 5$):

Initialize choice list C of courses with all available courses (from a master course list $\underline{C}$.

```
For i = 1 to number of students:
        (selecting courses for $s_i$)
        If C has at least $x_i$ number of courses then
                $x_i$ number of courses are removed randomly from Choice list C and put into the
list of exams taken by student $s_i$ . Updated choice list becomes $C = C - C_x$ . Next student picks his
courses from this updated choice list.

                i <- i + 1 (move to the next student)

        else
```

all courses n are removed from C and put into the list of exams taken by student s_i . Say this list of all courses is C_n .

Choice list is now empty. Re-initialize choice list C of courses with all available courses.

Remove all elements in C that are in C_n . The student can pick (x_i – n) number of courses from C\` = C - C_n . This ensures that in this list there are no courses that this student s_i has previously taken.

(x_i – n) courses are removed randomly from C\` and put into the list of exams taken by student $s_i$. C\` updates to C\`\` after removing (x_i – n) courses.

Elements in C_n are added to C\`\` and this list is updated to C\`\`\` . This ensures that the next student will have all available courses to choose from from the master course list $\underline{C}$, except for the ones that student s_i has chosen.

i <- i + 1 (move to the next student)

This strategy ensures that there are no courses with 0 students, and ensures that each student will have between 2 and 5 courses.

Determining Room Capacity:

Once we have generated courselist for each student, we determine the number and identity of students in each course, and from here determine the maximum number of students in a course, and minimum number of student in a course.
To ensure that the exam room can accommodate all the students of the course with the maximum number of students, we generate the exam room size with the following equation:

*Exam room size = 2 * (maximum number of students in a course + minimum number of students in a course).*

This ensures that the exam room size is big enough to accommodate the exam for the course with the largest student enrollment, and can house multiple course exam simultaneously if we want to.

Determining Maximum Number of Slots:

Maximum number of slots cannot exceed the number of exams (number of courses). So for generating problem instance with a confirmed valid solution, we used the following formula:

Maximum number of slots = number of courses.

Determining the problem should be solvable:

The generated .crs and .stu files were tested against assignment 2 code, and a feasible solution was generated. The generated files, with the feasible solution were tested against the verifier.

Against the generated .crs and .stu file, the solution that was produced was verified as a feasible solution.

b) Why is this a good test problem?

Having more than 100 exams, and 150 students with each student having more than 2 course tests creates a complex test input. The sample test input that was given to us had a minimal number of student and courses for which tests need to be scheduled. In real life environment, exam scheduling mostly involves a large number of students and a large number of courses – which make the course scheduling without violating the hard constraints a complex affair. The code I generated for constructing the sample problem can generate a large size test case – which more or less reflects a real life environment in exam scheduling. Also, the test problem that was generated has verified feasible solutions, which has been tested against the verifier for correctness.

When I was writing solution for both PA2 and PA3 assignments – I could not measure the performance of my written solution against the given test data – as it had a small number of courses and students. But when I ran my solutions against this generated problem, and other generated problems of varying size (generated using the generator code that was used to produce the solution), I was able to identify performance issues with my original solutions , and worked to tweak them.

In conclusion, I think this was a good test problem because of the following:

The submitted problem has a verified feasible solution

The submitted problem can give a good account of the performance of the solutions generated because of it's larger size.

It a good representation of a real life exam scheduling problem where there are a large number of courses and students.

c) Will this problem be challenging? Why or why not?

As this problem has a large number of exam and students, solving this problem in my opinion becomes challenging, and it can take a long time to generate a feasible solution – depending on the order of the algorithm designed for solving the problem. The problem in question can give a indication of the performance of the code – which cannot be easily identified with the trivial test data that was provided during A2. In my case, it helped me identify areas of improvement in generating a faster feasible solution, which I assume will be the case for other people using my generated problem.

Shaikh Shawon Arefin Shimon (skshimon)

Question 2: Describe your algorithm and its implementation, explaining why you designed it as you did, citing any relevant literature or pilot experiments you did to tune your implementation.

Answer:

For solving the timetabling problem, I employed a simple Genetic Algorithm with the following pseudo code:

```
1) Create initial population P with N solutions (Initialization)
2) Evaluate fitness of each member of initial population P
3) While (termination criteria not met) do:
{
    4) While all parents (or all but one, if N%2 != 0) from generation t = P has been selected:
    {
        a) Select 2 random parents from P, parent a and parent b
        b) Generate 2 member of immediate generation a` and b` by recombination
        c) For i in [a,b] do
        d) if i violates any hard constraint:
            e) Apply mutation strategy 1 to generate i`` from i
            else:
            f) Apply mutation strategy 2 to generate i`` from i
          // i`` = [a``, b``] is next generation (t+1) from current generation t [a,b]
        g) Evaluate fitness of a`` and b``
        h) Select best 2 out of a, b, a`` and b`` based on fitness value and replace a, b in P
to become P`. Suppose the selected pair from [a, b, a``, b``] is (x, y). So P` = P – {a, b} U {x, y}
where {x, y} in {a, b, a``, b``}
        i) Select another random pair c , d from a subset of P` which excludes slots
replaced in step h. New pair is always selected from confirmed members of generation t.
    }
    //at this stage, P` contains N` <= N members of generation t+1
    P = P`
}
```

Initialization:

Initialization was dependent on the selection of the objective function. Two different techniques of initialization were implemented: one for the first objective function (most packed schedule) and another for second objective function (spread schedule with minimal student cost).

Initialization for most packed schedule:

Here, I tried to generate an initial population with a packed schedule. I started with various random ordering of courses. With each ordering, I constructed an initial solution that has the most packed schedule possible for that ordering. This is the same heuristics that was used to build the initial solution set for PA2. No local SLS method was applied for generating this initial population.

Initialization for minimal student cost:

Here, I tried to generate initial solutions that are not densely packed. I started with various random ordering of courses. At each step of placing a course at a slot, I tried to make a choice between two options: a) If a course that is not already placed, can be placed on the current slot being considered without violating any hard constraints, and b) A course that is not already placed – to be placed in the next slot.

Choice a) cannot exist sometimes. In those cases, I went with choice b by default. In cases where both choices existed, I made a random choice between choice a) and b) – assigning more probability to making choice b). This was done so that the randomly generated schedule can be more spread.

This heuristic of constructing initial solution specifically targeting minimal student cost is similar to the heuristics applied for PA2 assignment. No local SLS method was applied for generating this initial population.

This heuristics is slightly modified from the initial construction generation of PA2. In PA2, the choice between option a) and b) was based on minimal student cost. There, the choice between same slot or next slot was taken based on whichever move had the smallest increase in student cost. But here that technique was replaced by a random choice with more probability assigned to choosing next slot. This decision was based on experimental data. In test cases with smaller number of exams (<50), the minimal student cost increase heuristics generates initial solution within an acceptable time – but with increased number of exams (>100), the performance of the algorithm decreases because of increased complexity associated with calculating student cost at each move. This had a heavy impact in generating initial population – impacting the overall performance of the solution. Hence, this heuristic was replaced with a biased random choice for constructing the initial solution.

Fitness Calculation:

Based on the selection of the objective function, two different calculations for calculating a solution's fitness were used.

For most packed schedule, the following calculation was done for determining a solution's fitness:

*Fitness = (Hard_Constraint_Satisfaction_Value) * ((number_of_occupied_slot_For_solution * 10000) + total_student_cost)*

If the solution violates any hard constraint, then Hard_Constraint_Satisfaction_Value = 0. Otherwise, Hard_Constraint_Satisfaction_Value = 1.

For most packed schedule, our primary criteria for selecting an optimal solution is the feasible solution with most packed schedule – meaning a solution that has least number of occupied slot. If there are multiple feasible solution that has the same number of occupied slot, the ranking of best solution then depends on the student cost – the least the better.

Keeping this heuristic in mind, we designed our fitness equation that puts more emphasis on the packing of the exams. If two packing occupy the same number of slots, slight variation is introduced to the packing value to determine the best solution for most packed schedule.

For least student cost, the following calculation was done for determining a solution's fitness:

Fitness = (Hard_Constraint_Satisfaction_Value)(total_student_cost)*

For least student cost, we are only concerned with minimizing the total student cost, provided that the solution does not violate any hard constraint. The fitness function for total student cost was designed keeping this in mind.

Termination criteria for Algorithm:

The population only consists of unique schedules (unique chromosomes). So given enough time to run, the population starts to converge. While running experiments with the current solution, it was observed that it takes a significant amount of time to converge for test data with large number of (>100) exams. As a starting point, I started with 100 unique chromosomes (schedules) as initial solution set. As it takes a long time (>2.5 minutes on average) if termination criteria is based on convergence, we made a greedy choice of generating 100 generations and then terminate, keeping the test environment in mind.

Selection of Parents:

For selecting parents, I employed *Tournament selection* where two chromosomes were randomly selected from the current generation and the best 2 among the parents and children were placed into the new population. The best 2 chosen were designated as next generation chromosomes so that they are not selected for recombination/mutation in the run for the current generation.

Representation of chromosome:

001	002	003	004	005	006	007	008	009	010	
4	1	5	3	2	6	7	1	4	6	parent 1
2	1	3	5	4	6	1	3	4	6	parent 2

To build up the chromosome, we employed a problem-domain specific heuristics. In a parent chromosome, each allele position corresponds to an exam identifier (course number), and a allele represents the actual timeslot number occupied by that exam. For example, parent 1 schedule has exam 007 at day 2 slot 2, so the actual timeslot number for 007 in parent 1 is (day-1)*5+slot_in_day = (2-1)*5+2 = 7. Encoding the schedule chromosomes this way ensures that chromosomes are of the same length, which helps later in recombination.

From the above example, it can be seen that if parent 1 and parent 2's total occupied timeslots vary (parent 1 takes total 7 time slots, and parent 2 takes total 6 time slots), then the symbol set for alleles in parent 1 and parent 2 can differ.

Recombination Operator and Mutation Strategy:

Here, a domain specific recombination operator was designed for generating immediate generation from generation t. This recombination operator is related to *reduced surrogate* method where the common alleles from both parents are transferred to the children, and crossover points are chosen randomly within differing allele positions.

This recombination operator has *Respect* and *Transmission* characteristics, but the chromosomes generated in the intermediate generation can violate hard constraints.

If the immediate generation violates hard constraint, we apply a mutation that fixes the hard constraints by rearranging hard constraint violating courses (changing their slots). This generates a mutated chromosome that closely resembles the immediate generation chromosome, only differing in a few allele positions.

If the immediate generation does not violate hard constraints, then we apply another mutation strategy where two exam slots are chosen randomly and swapped. For example, lets say the immediate generation chromosome has {001, 005} at slot 1 and {002,009} at slot 3. After mutation, slot 1 contains {002,009} and slot 3 contains {001, 005}.

Strings in new generation:

To ensure that the population converges towards optimal solution, we try to put intensive pressure on population for improvement. To do so, we always pick the best of offspring and parents based on the fitness function. As the fitness function for *least student cost* and *most packed schedule* are different in nature, it ensures that newer generations will have improved timetable focusing on the specific soft criteria that was targeted. The population contains only unique chromosomes to ensure convergence. In each run for a generation, parents are randomly selected for recombination to ensure that newer chromosomes that were generated later on have equal chance to recombine and mutate with an older surviving chromosome, or a relatively new chromosome.

Shaikh Shawon Arefin Shimon (skshimon)

Question 3:

Find and read a published research paper on applying an evolutionary algorithm to timetabling. Provide the citation, summarize the paper and describe what you learned from it.

Answer:

The paper I read for applying EA to timetabling was titled: *"Generating University Course Timetable using Genetic Algorithm and Local Search "*by Abdullah and Turabieh. In this paper, the authors establish a new algorithm based on Genetic Algorithm (GA) and Stocastic Local Search (SLS) to solve timetabling problem. The main theme of this paper is to apply GA to create new generation of solutions, and then apply local search on the new generation to make more improvement on the new generation of solutions. This application of SLS method on newly created generation makes the convergence of GA faster according to the experimental data provided in the paper.

Although SLS was involved in this paper – the main focus of this paper was on a genetic algorithm based approach for solving examination timetabling. This paper was chosen because of the hard constraint similarity between the assignment problem for PA2 and PA3, and the hard constraints considered for the problem described in the paper. On the course of searching a suitable paper that would help in implementing PA3, I found this paper to be the most use – hence the reason of discussing the paper.

This paper starts by introducing the examination timetable problem, with a set of hard and soft constraints. Different approaches of solving the discussed problem are touched on briefly in the introduction section. After describing the problem, the authors discuss the main concept of genetic algorithms, genotype representation and general idea on how to combine SLS with GA to increase performance of GA based solution. Here, the SLS is applied to improve soft constraint values on a newly created generation, and modified new generation fed to the GA for recombination and modification. This technique improves convergence in GA. After briefly giving a top-level overview on GA and SLS and how it can relate to the examination timetable problem – the authors go deeper, discussing specific details for their GA based solution.

The authors apply constructive heuristic for generating initial solutions that satisfy the hard constraints – without considering soft constraint violation. In their genotype representation, each gene in the chromosome represents a timeslot for corresponding event, and length of each chromosome equals to the number of events to be held. For parent selection, they apply a tournament selection process, and apply a single point crossover for recombination. Immediate generations generated after recombination is mutated by applying random changes. Random mutation point selection strategy is used. The authors apply repair function after crossover and mutation operation to repair infeasible timetables. They define a fitness function for evaluating each timetable which represents a better solution for lowering values. Fitness function is applied on all generated off-springs for current generation t to select the best offspring, on which local search algorithm is applied to improve the next generation. If the GA is not terminated, then the local search result is inserted back to the population for GA and run again.

Simulation results compare this proposed GA+SLS algorithm with other algorithms like RII, VNS+Tabu search, Graph based solution. The authors describe the result of the proposed algorithm within acceptable limits and claims to have a promising result – specifically achieving faster convergence with SLS based tweak.

Learning from the paper:

This paper gave a useful insight on how to represent chromosomes for timetabling problem when using Genetic algorithm based solution. But the most useful part of this paper was it's briefing on repair function to fix the resulting chromosomes from crossover and mutation operation. Ideas from this repair function algorithm was taken and customized to fit the solution for assignment PA3 where applicable. This paper also provides insight on constructing a fitness function equation applicable for our specific problem.

I tried to apply the initial population formation technique discussed here for assignment PA3, and used ideas of the repair function and fitness function to apply in my customized solution. Although SLS method was applied in this paper, it was applied for faster convergence and could be avoided – which I did as per the assignment requirement.

Reference:

Abdullah, S.; Turabieh, H., "Generating University Course Timetable Using Genetic Algorithms and Local Search," *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on* , vol.1, no., pp.254,260, 11-13 Nov. 2008
doi: 10.1109/ICCIT.2008.379
keywords: {computational complexity;genetic algorithms;genetic algorithms;sequential local search;timetabling;university course timetable;Artificial intelligence;Computational intelligence;Evolutionary computation;Genetic algorithms;Hybrid power systems;Information technology;Operations research;Scheduling;Search methods;Signal processing algorithms;Course Timetabling Problem;Genetic algorithm;Sequential Local Search},
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4682035&isnumber=4681984