Generative Perturbative Hyper-Heuristic for Curriculum-based Course Timetabling Problem

Isheanesu Joseph Dzingirai

September 2023

Contents

1 Introduction

Efficiently organizing courses within an academic institution is a complex task that involves multiple constraints and objectives. The Curriculum-based Course Timetabling problem (CB-CTT) seeks to allocate courses to time slots and rooms while satisfying various institutional constraints. Traditional methods for solving the CB-CTT often rely on manual intervention and heuristics, which can be time-consuming and sub-optimal.

This report presents an approach to tackling the CB-CTT problem using a generative-perturbative hyper-heuristic (GPHH). Hyper-heuristics offer a meta-level approach by generating and selecting heuristics or heuristic combinations to solve optimization problems. In this study, we introduce the concept of a GPHH specifically designed for the CB-CTT.

Generative Perturbative Hyper-Heuristic (GPHH) is a sophisticated optimisation methodology that uses the power of generative algorithms and perturbation strategies to automatically generate high-performing heuristics for complex optimisation situations. GPHH, in contrast to standard heuristics, uses a generative component to generate new heuristics and a perturbation component to fine-tune and optimise them over time. This approach enables GPHH to adapt to varied issue situations and dynamically evolve heuristics capable of solving various sorts of optimisation challenges. GPHH is especially useful in sectors where manual heuristic design is difficult since it automates the heuristic generating process and may produce cutting-edge solutions for a wide range of optimisation challenges.

To evaluate the effectiveness of the GPHH, comprehensive experiments were conducted using benchmark datasets from the International Timetabling Competition 2007 and compared against competition's best results as well as the results from Professor Pillay's 2016 paper. The results showcase the capability of the GPHH to consistently generate high-quality solutions across different problem instances. Additionally, it demonstrates robustness in handling various constraints of the CB-CTT.

2 Experimental Setup

2.1 System

The experiments were performed on a Dell Latitude 5510 laptop with the following machine specifications:

• RAM :	31.6 GB
• Disk Space:	417 GB
• CPU:	Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, 2304 MHz, 4 Core(s)

2.2 Programming Language and Libraries

The generation perturbative hyper heuristic was implemented using Python. No external libraries except for the built-in data structures from Python were used.

2.3 Parameter Values

• Maximum Generations:	100
• Population Size:	500
• Tournament Size:	4
• Crossover Probability:	0.7
• Mutation Probability:	0.3
• Upper Bound Codons:	16
• Lower Bound Codons:	8
• Upper Bound Codons:	16

3 Data Description

The data used to test the GPHH was obtained from the Optimization Hub. The data was used in the International Timetabling Competition 2007.

The structure of the data was:

1. The first 7 lines gave basic information about the problem instance. The basic information is:
    - name of problem instance
    - number of courses
    - number of rooms
    - number of days in the timetable
    - number of periods per day
    - number of unavailability constraints
2. Course - The following information is provided about the course:
    - id for unique identification
    - id of teacher who will be lecturing the course
    - number of lectures to be scheduled
    - number of minimum working days between scheduled lectures of the course
    - number of students enrolled in course
3. Room - The following information is provided about the room:
    - id for unique identification
    - room capacity
4. Curriculum - The following information is provided about the curriculum:
    - id for unique identification
    - number of courses in the curriculum (x)
    - x course ids for the courses in the curriculum
5. Unavailability Constraints - The following information is provided about the unavailability constraints:
    - course id that is constrained
    - the day that course is restricted
    - the period on the day that the course is constrained

3.1 Problem Instances

The problem instances used are:

- comp04.ctt.txt

– Name:	Ing0405-3
– Courses:	79
– Rooms:	18
– Days:	5
– Periods per day:	5
– Curricula:	57
– Constraints:	396

- comp05.ctt.txt

– Name:	Let0405-1
– Courses:	54
– Rooms:	9
– Days:	6
– Periods per day:	6
– Curricula:	139
– Constraints:	771

- comp08.ctt.txt

– Name:	Ing0607-3
– Courses:	86
– Rooms:	18
– Days:	5
– Periods per day:	5
– Curricula:	61
– Constraints:	478

- comp09.ctt.txt

– Name:	Ing0304-3

– Courses:	76
– Rooms:	18
– Days:	5
– Periods per day:	5
– Curricula:	75
– Constraints:	405

- comp12.ctt.txt

– Name:	Let0506-2
– Courses:	88
– Rooms:	11
– Days:	6
– Periods per day:	6
– Curricula:	150
– Constraints:	1368

- comp13.ctt.txt

– Name:	Ing0506-3
– Courses:	82
– Rooms:	19
– Days:	5
– Periods per day:	5
– Curricula:	66
– Constraints:	468

- comp15.ctt.txt

– Name:	Ing0203-1
– Courses:	72
– Rooms:	16
– Days:	5
– Periods per day:	5

  - Curricula: 68
  - Constraints: 382

- comp16.ctt.txt
  - Name: Ing0607-1
  - Courses: 108
  - Rooms: 20
  - Days: 5
  - Periods per day: 5
  - Curricula: 71
  - Constraints: 518

- comp18.ctt.txt
  - Name: Let0304-1
  - Courses: 47
  - Rooms: 9
  - Days: 6
  - Periods per day: 6
  - Curricula: 70
  - Constraints: 594

- ToyProblem.ctt.txt
  - Name: ToyExample
  - Courses: 4
  - Rooms: 2
  - Days: 5
  - Periods per day: 4
  - Curricula: 2
  - Constraints: 8

These instances are selected to represent a diverse range of scenarios and challenges commonly encountered in the curriculum-based course timetabling domain.

4 Methodology

4.1 Approach for Hyper-Heuristic

The approach selected for the hyper heuristic is the Grammatical Evolution based Generative Perturbative Hyper Heuristic (GE-GPHH).

4.2 Representation

The Grammatical Evolution (GE) uses variable length strings known as chromosomes to represent solutions or programs. These chromosomes consist of a linear sequence of genes (codons) that correspond to elements of a grammar. Each gene or codon (8-bit binary string) in the chromosome represents a production rule or terminal symbol in the grammar. They hold instructions on how to choose a grammar's production rules. The mapping of variable length linear chromosomes or genomes to executable programs in Grammatical Evolution makes use of a context-free Backus-Naur Form grammar. Each mapped chromosome represents a perturbative heuristic.

Reasons for choosing chromosomes and codons to represent the solutions in Grammatical Evolution are:

- Flexibility - Can handle complex solution spaces with a variety of structures by employing a grammar-based method. The representation of alternative solution structures is made possible by the language, which offers a set of rules that specify the valid codon compositions.
- Syntactic Correctness - The generated solutions are always syntactically valid because to the use of grammars. The evolution process is guided by the grammar rules, which minimizes the search space and prevents the development of useless solutions.
- Genetic Operators - Operators like mutation and crossover can be used to readily modify chromosomes. Individual codons can change as a result of mutations, creating new variations in the solutions. Crossover makes it possible to explore the search space by allowing the exchange of genetic material between chromosomes.

4.2.1 Grammar

The grammar is represented by the four-tuple $< \text{N, T, P, S} >$, where N represents a set of non-terminals, T a set of terminals, P, set of production rules that map the elements of N to T and S (the start symbol that is contained in the set N)

N = { < start >, < accept > , < heuristic >, < action >, < swap >, < swap_lectures() >, < swap_slots() >, < single_move() >, < n >, < comp > }

T = { single_move(), swap_slots(), swap_lectures(), room, course, slot, 1, 2, 3,4 , 5, 6, 7, 8, 9, 10, ILTA, AI, AEI }

S = < start >

The production rules P are represented by the structure on page 10.

4.2.2 Mapping Process

The process involves converting each gene (codon / binary string) in the chromosome to decimal and using them to select the production rules from the grammar to apply.

The equation is:

$$Rule = \text{(codon decimal value) \% (number of production rules)} \qquad (1)$$

By mapping across the codon sequence iteratively, a derivation tree (phenotype) evolves. The process of derivation is carried out from left to right, beginning with the leftmost non-terminal.
The operation proceeds by loping to the beginning of the codon sequence, a process known as wrapping, if the iteration process reaches the end of the series of codons before the derivation tree is evolved. By applying the generated phenotype to the problem, the fitness it is assessed.

```
⟨start⟩ ::= ⟨accept⟩ ⟨action⟩ ⟨heuristic⟩                (0)

⟨accept⟩ ::= ILTA                                       (0)
  |  AI                                                 (1)
  |  AEI                                                (2)

⟨heuristic⟩ ::= ⟨action⟩ ⟨heuristic⟩                     (0)
  |  ⟨action⟩                                           (1)

⟨action⟩ ::= ⟨swap⟩                                     (0)
  |  ⟨move⟩                                             (1)
  |  ⟨single_move⟩                                      (2)
  |  ⟨swap_lectures⟩                                    (3)
  |  ⟨swap_slots⟩                                       (4)
  |  ⟨action⟩                                           (5)

⟨swap⟩ ::= swap( ⟨n⟩ ⟨comp⟩ ⟨comp⟩ )                     (0)

⟨move⟩ ::= move( ⟨n⟩ ⟨comp⟩ ⟨comp⟩ )                     (0)

⟨single_move⟩ ::= single_move()                         (0)

⟨swap_lectures⟩ ::= swap_lectures()                     (0)

⟨swap_slots⟩ ::= swap_slots()                           (0)

⟨comp⟩ ::= room                                         (0)
  |  slot                                               (1)
  |  course                                             (2)

⟨n⟩ ::= 1                                               (0)
  |  2                                                  (1)
  |  3                                                  (2)
  |  4                                                  (3)
  |  5                                                  (4)
  |  6                                                  (5)
  |  7                                                  (6)
  |  8                                                  (7)
  |  9                                                  (8)
  |  10                                                 (9)
```

4.3 Initial Population Generation

A number of chromosomes are generated. The chromosomes are represented as variable length strings. The population size and the limits for the length of each chromosome are by the user-defined. Each chromosome's number of codons is decided at random within the predetermined range. For instance, if the range is [8 - 20], a chromosome with a length of 13 codons might be generated at random. The generated chromosomes are then mapped to a BNF grammar to produce a phenotype (derivation tree to be applied to the problem).

4.4 Objective/ Fitness Function

The goal of the GE-GPHH is to find a feasible timetable that minimizes the combined cost of hard constraint violations and soft constraint violations. The definition of the constraints are:

- Hard Constraints - must not be broken under any circumstances. A feasible timetable is one that satisfies all of the hard constraints.
- Soft Constraints - desirable, but may be broken if all of them cannot be met. They differ in both nature and importance from one institution to the next. The quality of timetables is frequently judged by how much the soft constraints in the timetables are violated.

Overall, the GE-GPHH finds a feasible timetable with no hard constraints and the fewest soft constraints.

4.4.1 Hard Constraints

The hard constraints for CB-CTT are:

- Lecture Allocations - The timetable must include the required number of lectures and each lecture must not be scheduled more than once in a period
- Conflicts - Lectures for courses in a curriculum must be arranged at distinct periods
- Room Occupancy - Each room must only be scheduled once in a period
- Teacher Availabilities - Each teacher must not be scheduled more than once in a period

- Lecture Availabilities - Each lecture must not be scheduled on the specific periods on specific days.

4.4.2 Soft Constraints

The soft constraints for CB-CTT are:

- Room Capacity - The number of students in a course assigned to a room for a lecture must not exceed the room's capacity
- Minimum Working Days - The lectures for a course must be spread out across the minimum number of working days indicated
- Curriculum Compactness - On each day, lectures for curriculum courses should be scheduled next to each other
- Room Stability - All lectures for a course should be held at the same room

4.5 Selection Method

The selection method used was tournament selection. In tournament selection, a fixed number of individuals are randomly selected from the population and the individual with the best fitness from the individuals selected is chosen as a parent for the next generation.

The reason tournament selection was chosen over fitness proportionate selection is for diversity. It allows all the individuals, regardless of the fitness to participate. This helps preserve a diversified population an makes it possible to explore a larger portion of the search space and avoid an early convergence to less-than ideal solutions

4.6 Genetic Operators

4.6.1 Crossover

Single-point crossover is a genetic operator that is used to recombine genetic material between two parent individuals. In single-point crossover, a random point (position) along the chromosomes of two parent individuals is selected from the population. The exchange of genetic material between the parents before to the chosen point results in the creation of two offspring.

The reasons single-point crossover was chosen are:

- Exploration - it made it possible for individuals to exchange genetic information, which makes it possible to explore various parts of the search space. It diversified the population and made it easier to recombine the individuals.
- Exploitation - encourages sharing of traits or solutions by merging the genetic material for the two parent individuals. It increases the probability of convergence towards an ideal solution.

The rate at which the single-point crossover operator was applied was high in order to prevent premature convergence of the grammatical evolution by exploring different areas of the program space.

4.6.2 Mutation

Mutation is a genetic operator used to introduce random changes into an individual within a population. It's purpose is to introduce diversity and allow the population to explore new areas in the search space that may potentially contain better solutions.

The two parents selected through the tournament selection process were then perhaps put to crossover and the mutation operator.

The following actions led to the mutation of each of the two trees:

- Selected a random gene or codon in the chromosome
- Generated a new gene or codon
- Replace the selected random gene or codon with the newly generated codon

This is similar to the bit mutation in a genetic algorithm.

The use of this operator was necessary to guarantee that the offspring were sufficiently different from both their parents and other population members. In order to keep the grammatical evolution algorithm from becoming too chaotic while still allowing for enough variety to successfully explore the program space, the mutation operator's rate was kept modest (less than 0.4)

4.7 Population Control Model

The population control model utilized was a steady-state control model (which assesses if the two children are fitter than certain solutions in the parent pool after applying crossover and mutation operators to two selected solutions). If they are, they replace the weakest programs in the pool). It was utilized because it lowered the amount of variety loss and improved the likelihood of retaining the best individuals in the population, resulting in faster convergence towards the optimal solution. It is also more memory efficient because it just keeps a certain number of solutions in the parent pool.

4.8 Termination Criteria

The following were used as termination criteria for the grammatical evolution algorithm:

- Max Number of Generations: The algorithm terminated after a fixed number of generations were reached. This was to ensure that the algorithm terminates when there is no clear convergence towards the best solution.
- Convergence of Fitness: The algorithm terminates when the best fitness achieved is better than threshold. This was to ensure that the algorithm converged to a satisfactory solution.

4.9 Construction of Initial Timetable

4.9.1 Construction Heuristic

The initial timetable (solution) was constructed using the following construction heuristics:

- Saturation Degree (SD) - Priority is given to the lectures with the fewest number of feasible times in the timetable at the current stage of construction
- Largest Enrolment (LE) - Priority is given to the lectures with the most students

The saturation degree is a dynamic heuristic (its value is adjusted during the timetable construction process.) whereas largest enrolment is a static heuristic (computed before the timetable construction process and remain the same throughout the process). When selecting a course to schedule, if there are courses with the same saturation degree, the course with the greatest number of students enrolled is chosen.

4.9.2 Lecture Placement

The strategy used to select a period to schedule a lecture on the timetable is the Minimum Penalty Period (MPP). The reason why it was chosen instead of First Period (which schedules a lecture in the first feasible period in the timetable) and Random Period (which schedules a lecture in a randomly chosen period from all the feasible periods in the timetable) is because:

- Efficiency - starting with feasible solutions can help to speed up the optimization process. Because solutions generated by the Minimum Penalty Period strategy are more likely to be close to the optimal solution space, following optimization phases can concentrate on refining and increasing the quality of these solutions rather than rectifying serious violations (hard constraints).
- Because it actively seeks to reduce penalties, MPP tends to be more stable in terms of solution quality. Random First and First Period may provide very variable beginning solutions, making it impossible to predict the behavior of the optimization process.

With the construction heuristics and the lecture placement choice, the initial timetable was construction using the following algorithm:

Algorithm 1 Algorithm for Generating Initial Timetable

```
Input: The timetable object
Output: None
1: while $\exists$ course with lectures left to schedule do
2:     calculate the saturation degree for each course left to be scheduled
3:     select the course with the lowest saturation degree
4:     place the selected course in the most feasible slot
5:     update the number of lectures left to be scheduled for the course
6: end while
```

4.10 Low-Level Heuristics

A low-level heuristic (llh) is a domain-specific operation or modification that can be applied to a solution to make it more optimal or to explore new parts of the solution space. It is used in conjunction with a hyper-heuristic.

The low-level heuristics used with GE-GPHH on CB-CTT are:

- Single Move - moves a single lecture selected a new feasible slot
- Swap Slots - swaps two slots selected at random in the timetable
- Swap Lectures - swaps two lectures selected at random in the timetable
- Swap - Swaps n components specified in the first ¡comp¿ for those supplied in the second ¡comp¿. For example, n courses in a specific slot can be swapped with courses in another slot.
- Move - n components provided in the first ¡comp¿ are moved to a new position in the second ¡comp¿. For example moving n courses to a new slot.

4.11 Move Acceptance

Move acceptance is used to determine whether to accept the perturbative heuristic's solution. The move acceptances employed in the GE-GPHH for CB-CTT are:

- Iterated limited threshold accepting (ILTA) - ILTA typically accepts improving and equal movements, as well as moves that produce poorer timetables, if the current iteration exceeds the stated iteration limit, the hard constraints for the produced timetable are zero, and the soft constraints of the current timetable are greater than the previous timetable of the produced solution by an amount specified by the threshold.
- Accept Improving (AI) - only accepts the application of the heuristic to the timetable if there is an improvement in quality of the resulting timetable.
- Accept Equal and Improving (AEI) - accepts moves producing timetable of the same or better quality

4.12 Algorithm

With the various GE-GPHH components defined above. This is how they work together:

Algorithm 2 Grammatical Evolution based Generation Perturbative Hyper Heuristic

```
Input: problem instance
Output: best chromosome representing the best heuristic
 1: Create an initial timetable (with the given problem instance) using a
    constructive heuristic
 2: Store the initial timetable as a current timetable
 3: Generate an initial population of variable length chromosomes (binary
    strings)
 4: Map all the chromosomes to their corresponding parse trees representing
    the perturbative heuristic (phenotype) using a BNF grammar
 5: Apply each perturbative heuristic in the population to a provided initial
    solution to determine its fitness
 6: while termination criterion not met do
 7:   Select the fitter perturbative heuristics for regeneration using the
      tournament selection method
 8:   Create new perturbative heuristics for the next generation by applying
      genetic operators to the earlier selected heuristics
 9:   Apply each new perturbative heuristic to a provided initial solution to
      determine its fitness
10:   Replace all the perturbative heuristics in the old population with the new
      perturbative heuristics
11:   Update the best solution
12: end while
13: return perturbative heuristic with the best fitness as the best solution
```

5 Results

5.1 comp04.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	41	185.0015
16223899	0	41	190.0011
22878896	0	41	209.2994
68625033	0	41	218.9474
101006424	0	41	137.1928
113139835	0	41	136.7042
125838671	0	41	121.42024
213232696	0	41	178.7096
311489368	0	41	186.0552
376423770	0	41	149.3832

Table 1: Constraint costs for problem instance comp04 for different runs with different seeds

5.2 comp05.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	190	128.8038
16223899	0	174	167.5845
22878896	0	173	97.2978
68625033	0	180	80.8659
101006424	0	173	74.9499
113139835	0	197	107.2147
125838671	0	170	61.7395
213232696	0	190	80.9476
311489368	0	172	62.3998
376423770	0	184	130.6333

Table 2: Constraint costs for problem instance comp02 for different runs with different seeds

5.3 comp08.ctt.txt

Seed	**Hard Constraints Cost**	**Soft Constraints Cost**	**Execution Time (s)**
5778856	0	31	461.2277
16223899	0	35	457.9260
22878896	0	33	459.8936
68625033	0	34	459.0906
101006424	0	33	458.9888
113139835	0	32	454.7071
125838671	0	34	452.2025
213232696	0	40	453.7278
311489368	0	33	457.3348
376423770	0	32	458.0043

Table 3: Constraint costs for problem instance comp08 for different runs with different seeds

5.4 comp09.ctt.txt

Seed	**Hard Constraints Cost**	**Soft Constraints Cost**	**Execution Time (s)**
5778856	0	46	102.7325
16223899	0	46	104.7320
22878896	0	46	105.7126
68625033	0	46	104.0310
101006424	0	46	103.7356
113139835	0	46	103.1468
125838671	0	46	102.8265
213232696	0	46	105.4772
311489368	0	46	101.2258
376423770	0	46	104.7852

Table 4: Constraint costs for problem instance comp09 for different runs with different seeds

5.5 comp12.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	45	150.9197
16223899	0	45	151.9048
22878896	0	45	150.5183
68625033	0	45	149.9296
101006424	0	45	149.5211
113139835	0	45	150.4493
125838671	0	45	149.9698
213232696	0	45	148.1111
311489368	0	45	151.0289
376423770	0	45	150.3698

Table 5: Constraint costs for problem instance comp12 for different runs with different seeds

5.6 comp13.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	49	114.6081
16223899	0	49	115.1732
22878896	0	49	115.0429
68625033	0	49	114.8807
101006424	0	49	113.9933
113139835	0	49	116.6482
125838671	0	49	119.6684
213232696	0	49	115.8449
311489368	0	49	115.1226
376423770	0	49	113.4993

Table 6: Constraint costs for problem instance comp13 for different runs with different seeds

5.7 comp15.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	118	79.4299
16223899	0	118	80.0201
22878896	0	118	79.3051
68625033	0	118	79.3282
101006424	0	118	79.7536
113139835	0	118	78.6902
125838671	0	118	78.9898
213232696	0	118	79.0318
311489368	0	118	78.5857
376423770	0	118	79.4873

Table 7: Constraint costs for problem instance comp15 for different runs with different seeds

5.8 comp16.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	82	199.8778
16223899	0	82	200.6794
22878896	0	82	200.5794
68625033	0	82	205.5090
101006424	0	82	231.7716
113139835	0	82	511.2931
125838671	0	82	459.6515
213232696	0	82	417.4849
311489368	0	82	485.0332
376423770	0	82	526.9747

Table 8: Constraint costs for problem instance comp16 for different runs with different seeds

5.9 comp18.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	39	66.8615
16223899	0	39	63.2838
22878896	0	39	70.9083
68625033	0	39	74.6128
101006424	0	39	72.1200
113139835	0	39	72.7200
125838671	0	39	71.6221
213232696	0	39	71.9996
311489368	0	39	69.5393
376423770	0	39	70.4894

Table 9: Constraint costs for problem instance comp18 for different runs with different seeds

5.10 comp21.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	59	400.1551
16223899	0	59	398.4619
22878896	0	59	382.7470
68625033	0	58	349.7590
101006424	0	57	341.9945
113139835	0	59	309.1746
125838671	0	57	332.4882
213232696	0	58	317.1779
311489368	0	57	458.8211
376423770	0	59	332.2962

Table 10: Constraint costs for different runs with different seeds

5.11 ToyProblem.ctt.txt

Seed	Hard Constraints Cost	Soft Constraints Cost	Execution Time (s)
5778856	0	0	0.0577
16223899	0	0	0.1049
22878896	0	0	0.0743
68625033	0	0	0.0552
101006424	0	0	0.0378
113139835	0	0	0.0353
125838671	0	0	0.0358
213232696	0	0	0.0359
311489368	0	0	0.0386
376423770	0	0	0.0400

Table 11: Constraint costs for problem instance TonyProblem for different runs with different seeds

5.12 Summary of Problem Instances

Since the all the timetables generated for the problem instances are feasible, all data related to the hard constraints has been omitted because the cost is always zero.

NB: SCC stands for Soft Constraints Cost

Problem	Best SCC	Avg. SCC	Std. SCC	Avg. Execution Time (s)
comp04	41	41	0	171.2714
comp05	170	180.3	8.9560	99.2436
comp08	32	33.7	2.3685	457.3103
comp09	46	46	0	103.8405
comp12	45	45	0	150.2722
comp13	49	49	0	115.4481
comp15	118	118	0	79.26217
comp16	82	82	0	343.8854
comp18	39	39	0	70.4156
comp21	57	58.2	0.8717	362.3075
ToyProblem	0	0	0	0.0516

Table 12: Statistics for the different problem instances

The table above shows the best, average and standard deviation of the soft constraints cost obtained for the different problem instances.

6 Discussion of Results

6.1 Comparison with Optimization Hub ITC 2007 Results

The GE-GPHH received a lower cost for the majority (80 percent) of the problem instances on which it was run. In some cases, the difference was significant, although in others, the optimization hub problem instances cost were 10 percent more than the cost calculated using the hyper-heuristic.

6.2 Comparison with Prof. Pillay 2016 Research Paper

The GE-GPHH achieved a lower cost for all of the problem instances on which it was run that the costs stated in the research paper. The costs stated in the research paper were about 80 percent more than the cost calculated using the hyper-heuristic.

6.3 Time Comparison

On average, it did not take long to find a very excellent achievable timetable in most cases. Finding the most viable / ideal timetable took about 3 minutes. This shows that quality timetables were generated in a short period of time.

6.4 Standard Deviation

Based on the results, the average standard deviation is low (signifies that the data points are relatively close to the mean or central value.) for each problem instance and over all the problem instances. This indicates that the performance of the hyper-heuristic is relatively consistent across different runs.

7 Conclusion

Finally, the incorporation of grammatical evolution generation perturbative hyper-heuristic is a potential and novel way to addressing complicated optimization problems. This hybrid methodology addresses a wide range of real-world difficulties by using the ability of grammatical evolution to generate solution representations and combining it with perturbative hyper-heuristic techniques for fine-tuning and optimization. The approach displays adaptability and scalability through the automatic development of heuristics based on grammatical evolution, making it a helpful tool in solving difficult optimization issues across many fields. The interaction of these two strategies has the potential to significantly improve problem-solving abilities and advance the field of optimization.

8 Extra Notes

- Please refer to the README.md for instructions on how to run the hyper-heuristic
- The results of the hyper-heuristic were compared to the results from here: https://opthub.uniud.it/problem/timetabling/edutt/ctt/cb-ctt-ud2