Matthew Healy and Deacon Seals

USING GENETIC PROGRAMMING TO GENERATE SCHEDULING HEURISTICS FOR REAL-TIME TASKS

Abstract – Real-time job scheduling is an active field of research that has been largely ignored by the field of computational intelligence. We suggest that genetic programming stochastic optimization search is applied to this domain in order to automate the highly-manual process of scheduling algorithm selection and design. We show that genetic programming is capable of producing human-competitive search algorithm design and is capable of automating this task.

1 INTRODUCTION

Job scheduling is a well-known NP-Hard open problem that is still the subject of active research. Despite the prevalence of research on this topic, many publications do not address real-time scheduling. Additionally, real-time scheduling algorithm selection is still a highly manual process that costs expensive engineering man hours. Advances in artificial intelligence (AI) and more broad computational intelligence (CI) offer tantalizing solutions to these problems, but existing publications focus primarily on more generalized non-real-time versions of the job scheduling problem. Additionally, many existing CI publications involving real-time task scheduling come with significant computational overhead. We propose the use of genetic programming to create general-purpose scheduling algorithms that aim to minimize computational overhead while maintaining high performance in real-time task scheduling. Genetic programming is proposed as a solution to this problem due to its ability to produce human-competitive results in design tasks [1].

2 BACKGROUND

While much research has been done on the subject of making computationally intelligent systems more real-time, the problem of using computational intelligence to advance real time systems has been much less explored[2-4]. Several efforts have been undertaken to make the internal decision making of computational intelligences faster; many of these efforts have been labeled as “real-time” due to their performance, but include no actual task scheduling or hard deadlines. Efforts to directly embed artificially intelligent decision making *within* real-time systems creates a large overhead, since online searches are rarely performant enough to meet the stringent deadlines of most real-time systems[5]. In the case of RESCU, one of the first real-time expert systems, continual

balances had to be made between ‘inferencing short cuts’ and the risk of losing vital hypotheses within the inference engine, due to performance restrictions alone[6]. Efforts that do actually involve task scheduling use neural networks which create the entire task schedule rather than evolving a heuristic a priori that can be used in real time[7].

Genetic programming (GP) is a subfield of computational intelligence that aims to perform a stochastic optimizing search through means of biologically-inspired mechanisms. In particular, GP often targets design challenges such as circuit design [9]. In these tasks and others, GP has proven to be capable of producing human-competitive designs [1]. GP utilizes fundamental search methods from its parent field, evolutionary algorithms. Evolutionary algorithms operate by representing potential problems solutions as genetic information that is then assessed for fitness and propagated in a survival-of-the-fittest-style cycle of parent selection, child generation, and survival selection. This process will be explained in greater detail in the methodology section. One particularly important part of genetic programming (and evolutionary computing) is the way in which solutions are represented as genetic information. This problem is addressed in a contribution by Dr. John Koza through the proposal of Koza-style GP parse trees [10] which allow for the representation of numerical expressions as trees made of internal mathematical operator nodes and value-returning leaf nodes called sensor nodes. This representation enables the generation of complex domain-specific heuristics through biologically-inspired GP methods. The detailed mechanisms of this approach will be discussed in the methodology section.

GP-generated heuristics are normally applied to scenarios in which the fitness or performance of a heuristic can be calculated on a set of representative or generalized problems. This enables the application of generalized solutions to similar problems and reduces the need for a priori information. Existing publications have attempted to solve scheduling problems through the use of GP-generated scheduling heuristics, but often don’t consider real-time scheduling characteristics. Among these are considerations of periodicity-based task priority and computational overhead at runtime. Additionally, most publications only approach the problem of job scheduling with the goal of minimizing runtime and do not consider the job timeout and consequent critical failures that can occur in real-time systems [11-20]. Publications such as [11,12] apply genetic algorithms (not genetic programming) to simultaneously evolve schedules directly and rules to represent those schedules. These publications focus on decreasing execution time provided a set of job dependencies and resources, but do not directly apply to real-time scheduling tasks and do not consider the creation of generalized solutions for jobs not considered in the fitness evaluation.

Other publications focus specifically on multiprocessor scheduling with the goal of minimizing compute time across the entire system [13,14]. Genetic algorithms are still employed for direct schedule evolution instead of scheduling heuristic generation, but the consideration of message passing and other multiprocessor overhead are considered

[13,14] and these publications are slightly more relevant to real-time system applications where computing resources should be considered in detail. Dispatching rules are another popular topic of scheduling research [15-17] that relate closely to our approach as they resemble the scheduling algorithms our GP approach will generate. These publications fail to consider targeting multiple job scheduling scenarios simultaneously and thus do not generate general-purpose scheduling heuristics for tasks represented by multiple scenarios. The GP work presented also doesn't target real-time specific scheduling characteristics [15-17]. Publications that use GP to address application-specific problems like scheduling in security-focused applications exist [18], but are altogether not applicable to real-time scheduling in a direct manner.

A related field to GP is hyper-heuristics, which specifically focuses on using heuristics as building blocks in an evolutionary search to generate complex general-purpose heuristics capable of scheduling multiple problems through the use of application-specific identification knowledge utilized via the evolution process. Publications that utilize hyper-heuristics for job scheduling exist[19-20], but target the flexible job scheduling problem instead of real-time job scheduling. The flexible job scheduling problem focuses on minimizing execution time provided a set of jobs with an execution order across multiple machines with no preemption and is the subject of many scheduling publications [8,19,20].

Our contribution to this field of research is the application of genetic-programming-generated scheduling algorithms to sets of real-time tasks. This enables engineers in the field of real-time systems to automate the process of scheduling algorithm selection and design by simply generating a set of tasks that is representative of scenarios the real-time system may encounter when deployed. By enabling the a priori automated development of computational-overhead-aware heuristics, novel scheduling algorithms may be generated with minimal engineering overhead.

3 METHODOLOGY / METRICS

Our approach involves the generation of heuristics to schedule multiple sets of tasks that may be representative of scheduling scenarios the algorithm may encounter when deployed and online. This enables the generation of general-purpose scheduling algorithms capable of reduced computational overhead due to the representative data provided by these sets of tasks. Scheduling heuristics are generated in a manner that considers computational overhead and penalizes heuristics that utilize dynamic scheduling or complex functions. Heuristics are created through the use of a stochastic optimization algorithm, GP.

A GP is a specific type of evolutionary algorithm that relies on specific representations and evolutionary characteristics to address normally manual design tasks. As such, it is important to discuss the mechanisms of an evolutionary algorithm and what mechanisms

are specific to GP. Evolutionary algorithms operate via the creation of a population of potential solutions to the target problems. This population is comprised of individuals with encoded solutions that are similar to genetic information in biology. Individuals in a population are evaluated with a fitness metric and assigned a fitness value that is used to determine the selection of parents for child generation. In GP, child generation can be performed by cloning an individual and mutating their encoded solution via a random modification or by selecting multiple individuals and combining their solutions to create a new solution. Once new children solutions are generated, they are evaluated and added to the main population which is then reduced in size so the more fit individuals have a higher probability of participating in the next generation of evolution. The next generation of evolution begins again with parent selection of the current population and continues through consequent generations until a termination criteria is reached. For the purposes of this experiment, the termination criteria is simply the number individuals that have been evaluated.

Our GP makes use of Koze-style GP parse trees [10] to represent solution encodings as functions that can be evaluated recursively from root to leaf. Each node is either an operator that returns the result of an operation performed on the values returned by its two children nodes or the node is a leaf node that returns literal values that correspond to sensor values from our scheduling scenarios. For the purpose of scheduling real-time tasks and reducing computational overhead, we have divided our sensor node types into task information and job information to easy represent whether or not a scheduling relies on static priority (task information only) or dynamic scheduling (utilizes job-specific information or current time). The sensor nodes used in our parse trees are as follows: randomly initialized constant value, current time, task offset, task period, task deadline, task execution time, task blocking time, start of task blocking time, task periodicity, job release time, job deadline, remaining job execution time, start of job blocking, and remaining job blocking time. Our operator nodes are as follows: addition, subtraction, multiplication, division (returns 0 if the denominator is 0), modulo (returns the numerator if the denominator is 0), minimum of two values, maximum of two values. This heuristic is evaluated to generate a priority value for each job when simulated then sorted in ascending order where the first job is selected for execution. By using this heuristic representation and sorting mechanism, many common sorting algorithms can be represented as simple trees. For example, earliest deadline first (EDF) can be represented as a tree containing only a single job deadline node.

Statistics measured to create the fitness of an individual include the average percentage of periodic jobs scheduled, the average percentage of sporadic jobs scheduled, and the response time of aperiodic jobs. Our fitness function sums these three statistics (weighting the scheduling of periodic tasks to be the most important term) and subtracts a term correlated with the complexity of the heuristic. The complexity of a heuristic is determined by the size of the tree which represents it, as well as whether or not the

heuristic requires non-static information (e.g. job deadline, current time, job release, etc). Be discouraging the development heuristics requiring non-static information, our generated heuristics are more likely to have a reduced overhead, reducing the execution time of a scheduling server task, and decreasing the amount of information needed at any one time. To simulate the scheduling of jobs, we step through each moment in time, use our evaluate function to assign priorities to jobs, sort based on priority, and run the highest priority job. In the event of a tie between equally high-priority jobs, the tie is given to the job which is already running, if such a job exists. For more information, see the evaluate function in Appendix C.

4 Results and Discussion

Our problem set includes jobs with blocking time, periodic tasks, sporadic tasks, and aperiodic tasks. We have a problem involving sporadic jobs with blocking times, two problems made up entirely of periodic jobs, a problem with periodic jobs, an aperiodic job, and a sporadic job, and a problem with periodic jobs and aperiodic jobs. A detailed breakout of tasks used for each problem in our problem set can be found in Appendix D.

4.1 Plots and figures

The results of our experiment are as follows in Figure 1 and were generated with 30 runs of 2000 individual evaluations. Due to the stochastic nature of GP searches, the results are averaged over these 30 runs and each data point represents the population at a particular generation. The values plotted in are the best population fitness, the average population fitness, and the fitness of EDF as a baseline.

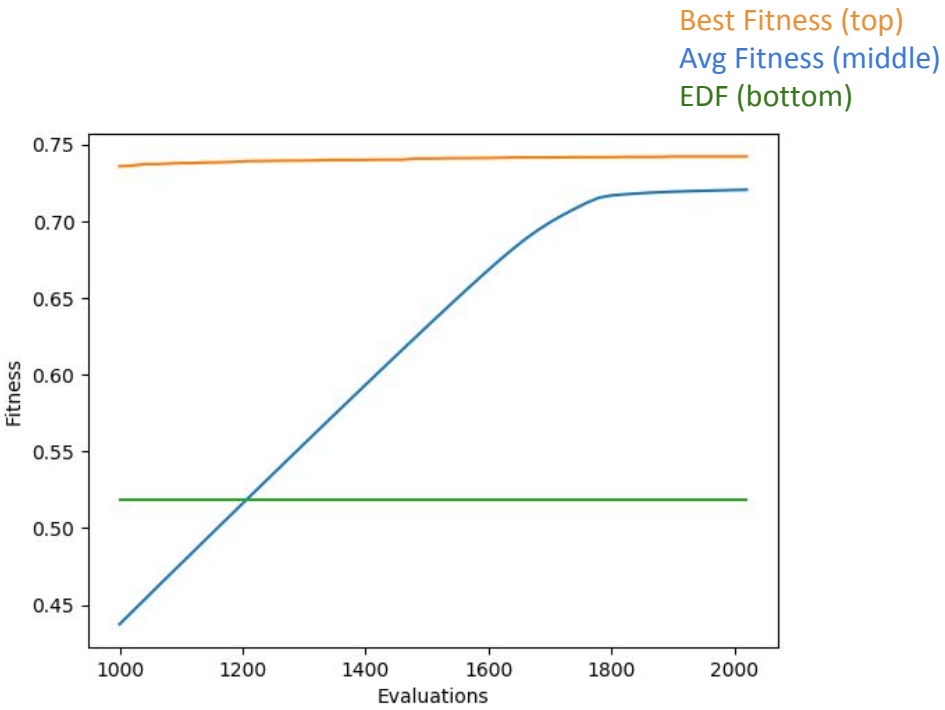

Figure 1. Fitness evolution over time

	Fitness
Average	0.7206843001073753
Best	0.758888185070311

Table 1. Fitness of best individual in population after 2000 evaluations averaged over 30 runs

4.2 Comment on the results

As can be observed in Figure 1, our GP approach was capable of generating heuristics that successfully schedules the sets of tasks with lower computational overhead than EDF. The global optimum result from all 30 runs was deadline monotonic as this was able to best schedule all of the tasks in each set of tasks while minimizing computational overhead due to static task-based priority. This result demonstrates the potential of our

GP to generate human-competitive design results and automate the process of scheduling algorithm selection.

5 Conclusions

Our GP scheduling solution demonstrated the ability to produce human-competitive design results and automating the selection and design of high-quality scheduling algorithm. The heuristics generated by our GP did, however, have the tendency to successfully schedule tasks with low-quality or even random scheduling heuristics due to the binary schedulability of some of the tasks scheduled. We were able to address this issue by adding aperiodic tasks and the consideration of response time to our fitness metric provided a gradient for our search algorithm to traverse and differentiate the performance of various heuristics in the population. Our results could also be more meaningful if more challenging task scheduling scenarios were attempted. Some scenarios for consideration are multi-core real-time scheduling and multi-resource preemptable resource blocking. These scenarios could be the subject of future research and were not attempted in this work due to development overhead and the need to define desired execution behavior in multi-core environments such as moving jobs between cores and data dependencies caused by message passing or lack thereof.

6 References

[1] J. R. Koza, “Human-competitive results produced by genetic programming,” *Genet. Program. Evolvable Mach.*, vol. 11, no. 3–4, pp. 251–284, 2010.

[2] M. Blej and M. Azizi, ”Task parameter impacts in fuzzy real time scheduling”, Journal of Universal Mathematics, 1(2):195-203, 2018. ISSN-26185660

[3] D. Xue and Z. Dong, ”An intelligent contraflow control method for real-time optimal traffic scheduling using artificial neural network, fuzzy pattern recognition, and optimization”, IEEE Transactions on Control Systems Technology, 8(1):183-191, 2000.

[4] V. Botti and C. Carrascosa and V. Julian and J. Soler and Informticos Computacin,”The ARTIS agent architecture: modelling agents in hard real-time environments”, In Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW99):63-76, 1999.

[5] D. J. Musliner, J. A. Hendler, A. K. Agrawala, E. H. Durfee, J. K. Strosnider and C. J. Paul, "The challenges of real-time AI," in *Computer*, vol. 28, no. 1, pp. 58-66, Jan. 1995

[6] R. Shaw, "RESCU - on-line real-time artificial intelligence," in *Computer-Aided Engineering Journal*, vol. 4, no. 1, pp. 29-30, February 1987.

[7] T. P. Liang and H. Moskowitz and Y. Yih, ”Integrating neural networks and semi-markov processes for automated knowledge acquisition: an application to real-time scheduling”. Decision Sciences, 23: 1297-1314, 1992. doi:10.1111/j.1540-5915.1992.tb00450.x

[8] C. J. Tan and S. c. Neoh and C. P. Lim and S. Hanoun and W. P. Wong and C. K. Loo and L. Zhang and S. Nahavandi, ”Application of an evolutionary algorithm based ensemble model to job-shop scheduling”, Journal of Intelligent Manufacturing, 30(2):879-890. ISSN 0956-5515

[9] J. R. Koza, F. H. Bennett, D. Andre, M. A. Keane and F. Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming," in *IEEE Transactions on Evolutionary*

Computation, vol. 1, no. 2, pp. 109-128, July 1997.

[10] J. R. Koza and J. R., *Genetic programming : on the programming of computers by means of natural selection*. MIT Press, 1992.

[11] L. Ozdamar, "A genetic algorithm approach to a general category project scheduling problem," in *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 29, no. 1, pp. 44-59, Feb. 1999.

[12] F. Pezzella, G. Morganti, and G. Ciaschetti, "A genetic algorithm for the Flexible Job-shop Scheduling Problem," *Comput. Oper. Res.*, vol. 35, no. 10, pp. 3202–3212, Oct. 2008.

[13] R. C. Correa, A. Ferreira and P. Rebreyend, "Scheduling multiprocessor tasks with genetic algorithms," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 8, pp. 825-837, Aug. 1999.

[14] A. S. Wu, H. Yu, S. Jin, K. -. Lin and G. Schiavone, "An incremental genetic algorithm approach to multiprocessor scheduling," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 9, pp. 824-834, Sept. 2004.

[15] S. Nguyen, M. Zhang and K. C. Tan, "Surrogate-Assisted Genetic Programming With Simplified Models for Automated Design of Dispatching Rules," in *IEEE Transactions on Cybernetics*, vol. 47, no. 9, pp. 2951-2965, Sept. 2017.

[16] S. Nguyen, M. Zhang, M. Johnston and K. C. Tan, "A Computational Study of Representations in Genetic Programming to Evolve Dispatching Rules for the Job Shop Scheduling Problem," in *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 5, pp. 621-639, Oct. 2013.

[17] S. Nguyen, M. Zhang, M. Johnston and K. C. Tan, "Automatic Design of Scheduling Policies for Dynamic Multi-objective Job Shop Scheduling via Cooperative Coevolution Genetic Programming," in *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 2, pp. 193-208, April 2014.

[18] S. Song, Kai Hwang and Yu-Kwong Kwok, "Risk-resilient heuristics and genetic algorithms for security-assured grid job scheduling," in *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 703-719, June 2006.

[19] J. Grobler and A. P. Engelbrecht, "Hyper-heuristics for the flexible job shop scheduling problem with additional constraints," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9713 LNCS. pp. 3–10, 2016.

[20] J. Grobler, "A Multi-objective Hyper-Heuristic for the Flexible Job Shop Scheduling Problem with Additional Constraints," in *Proceedings - 2016 3rd International Conference on Soft Computing and Machine Intelligence, ISCMI 2016*, 2017, pp. 58–62.

7 APPENDIX A

```python
# GP.py
from random import sample, randrange

from copy import deepcopy
from statistics import meanfrom math import floor, ceil
import numpy as np
import matplotlib.pyplot as plt
from Tree import Individual, Node
from Task import Problem

K_CONST = 10
MAX_EVALUATIONS = 2000
MIN_DELTA = 0.001
RUNS = 30


class GP:
   def __init__(self, population_size=1000, children_size=20, mutation=0.05, parsimony = 0.5):
       # dummy population representation
       self.population = []
       for _ in range(floor(population_size/2)):
           individual = Individual(parsimony)
           individual.grow(3)
           self.population.append(individual)
       for _ in range(ceil(population_size/2)):
           individual = Individual(parsimony)
           individual.full(3)
           self.population.append(individual)
       self.parents = []
       self.population_size = population_size
       self.children = []
       self.children_size = children_size
       self.mutation = mutation
       self.evaluations = 0
       self.parsimony = parsimony

   def parentSelection(self):
       # K-tournament
       self.parents = []
       for _ in range(self.children_size):
           self.parents.append(max(sample(self.population, K_CONST)))

   def childGeneration(self):
       self.children = []
       for i in range(self.children_size):
           if randrange(0, 1) < self.mutation:
               random_tree = Individual(self.parsimony)
               random_tree.root.grow(3)
               parent_copy = deepcopy(self.parents[i])
               parent_copy.stats = []
```

```
                parent_copy.root.choose_node(True, random_tree.root)
                self.children.append(parent_copy)
            else:
                parent_copy = deepcopy(self.parents[i]).recombine(self.parents[i+1%self.children_size])
                parent_copy.stats = []
                self.children.append(parent_copy)

    def reintroduction(self):
        self.population += self.children
        self.children = []

    def survivalSelection(self):
        # K-tournament
        new_pop = []
        for _ in range(self.population_size):
            chosen = max(sample(self.population, K_CONST))
            new_pop.append(chosen)
            self.population.remove(chosen)
        self.population = new_pop

    def evaluate(self, problems):
        for individual in self.children:
            individual.evaluate(problems)
            self.evaluations += 1

    def not_finished(self):
        return self.evaluations <= MAX_EVALUATIONS

    def run(self, problems):
        bests = []
        generations = ceil((MAX_EVALUATIONS - self.population_size)/self.children_size)+2
        data_best = [[] for i in range(generations)] # preallocate statistics arrays
        data_avg = [[] for i in range(generations)]
        for run in range(RUNS):
            self.__init__()
            # initial evaluation
            self.children = self.population
            self.evaluate(problems)
            self.population = self.children
            # end init
            fitness_data = [i.fitness for i in self.population]
            data_best[0].append(max(fitness_data))
            data_avg[0].append(mean(fitness_data))
            generation = 1
            while self.not_finished():
                self.parentSelection()
                self.childGeneration()
                self.evaluate(problems) # update fitness
                self.reintroduction()   # reintroduce children to population
                self.survivalSelection()
                fitness_data = [i.fitness for i in self.population]
                data_best[generation].append(max(fitness_data))
                data_avg[generation].append(mean(fitness_data))
                generation += 1
            print('==== RUN {} ===='.format(run))
```

```
            current_best = max(self.population)
            print('best: {}\nheuristic: {}'.format(
                current_best.fitness, current_best.root.string()))
            print('stats: {}'.format(current_best.stats))
            bests.append(current_best)
        print('==== GLOBAL OPTIMUM ====')
        best = max(bests)
        print('best: {}\nheuristic: {}'.format(best.fitness, best.root.string()))
        print('stats: {}'.format(best.stats))
        data_avg  = [i for i in map(mean, data_avg)] # lord forgive me
        data_best = [i for i in map(mean, data_best)]
        x = np.arange(self.population_size, self.population_size+self.children_size*generations,
self.children_size)
        plt.plot(x, data_avg, x, data_best)
        plt.xlabel('Evaluations')
        plt.ylabel('Fitness')
        plt.show()

# Helper Functions

def pairwise(iterable):
    's -> (s0, s1), (s2, s3), (s4, s5), ...'
    a = iter(iterable)
    return zip(a, a)
```

8 APPENDIX B

```python
# Task.py
class Task:
    def __init__(self, period=0, release=0, deadline=0, exec_time=0, blk_st=0, blk_dur=0):
        self.period = period
        self.release = release
        self.deadline = deadline # relative
        self.exec_time = exec_time
        self.blocking_start = blk_st
        self.blocking_duration = blk_dur
        self.priority = None

    def __lt__(self, other):
        return self.priority < other.priority

class Job:
    def __init__(self, task: Task, time):
        self.task = task
        self.exec_time = task.exec_time # remaining execution time
        self.release = time
        self.deadline = self.release + task.deadline if task.deadline != 0 else 0
        self.blocking_start = task.blocking_start + self.release
        self.blocking_duration = task.blocking_duration
        self.priority = None
        self.has_run = False

    def __lt__(self, other):
        if self.priority == other.priority and self.has_run == True and other.has_run == False:
            return True
        else:
            self.priority > other.priority


class Problem:
    def __init__(self, tasks=[], hyper_period=0):
        self.tasks = tasks
        self.hyper_period = hyper_period
```

9 APPENDIX C

```
# Tree.py
from random import choice, randint, shuffle
from statistics import mean
from copy import deepcopy
from Operations import *  # import operation constants (bastardized Sum-Type)
from Task import Job, Problem


class Node:
   def __init__(self, left=None, right=None, val=0, op=CONST):
       self.left = left
       self.right = right
       self.val = val
       self.op = op

   def size(self):
       return (self.left.size() if self.left != None else 0) + 1 + (self.right.size() if self.right !=
None else 0)

   def uses_nonstatic(self):
       if self.left == None and self.right == None:
           return self.op in NONSTATIC
       elif self.left == None and self.right != None or self.right == None and self.left != None:
           print('!!!!!!!malformed tree!!!!!!!!!')
       else:
           return True if self.op in NONSTATIC else self.left.uses_nonstatic() or
self.right.uses_nonstatic()

   def grow(self, depth_limit):
       '''
       generate tree with max depth depth_limit
       '''
       if depth_limit == 0:
           self.op = choice(LEAVES)
       else:
           self.op = choice(OPSUM)
       if self.op in OPERATORS:
           self.left = Node()
           self.left.grow(depth_limit - 1)
           self.right = Node()
           self.right.grow(depth_limit - 1)
       elif self.op == CONST:
           self.val = randint(0, 255)

   def full(self, depth_limit):
       '''
       generate tree full to passed depth_limt
       '''
       if depth_limit == 0:
           self.op = choice(LEAVES)
       else:
           self.op = choice(OPERATORS)
```

```
            self.left = Node()
            self.left.full(depth_limit - 1)
            self.right = Node()
            self.right.full(depth_limit - 1)
        if self.op == CONST:
            self.val = randint(0, 255)

    def choose_node(self, graft=False, node=None):
        '''
        Copyright Kool Kids Klub
        '''

        def choose_r(tree_array, node, i):
            if node.left != None and node.op not in LEAVES:
                next_idx = 2 * i
                tree_array.append(next_idx)
                tree_array = choose_r(tree_array, node.left, next_idx)
            if node.right != None and node.op not in LEAVES:
                next_idx = (2 * i) + 1
                tree_array.append(next_idx)
                tree_array = choose_r(tree_array, node.right, next_idx)
            return tree_array

        tree_array = [1]
        tree_array = choose_r(tree_array, self, 1)
        random_node = 1 if tree_array == [] else choice(tree_array)
        parent_list = []  # Was parent_list = [random_node], but I changed this so the selected node is
never moved to
        while random_node != 1:  # generate lineage
            random_node = random_node // 2
            parent_list.append(random_node)  # += [random_node // 2]
        #print(parent_list)
        if parent_list != []: parent_list.pop()  # remove root, last element is parent
        #print(parent_list)
        parent_list.reverse()
        current_node = self
        for node_idx in parent_list:
            # follow tree back to chosen node
            current_node = current_node.left if node_idx % 2 == 0 else current_node.right
        if graft:
            #print('NODE:{}'.format(node))
            spam = False
            if random_node == 1:
                self = deepcopy(node)
            else:
                if current_node.op in LEAVES:
                    print('PANIC!!!!!')
                    print('OP:{}'.format(current_node.op))
                    spam = True
                if random_node % 2 == 0:  # graft on randomly selected node
                    if spam:
                        print('CURRENT:{}'.format(current_node))
                        print('LEFT:{}'.format(current_node.left))
                    current_node.left = deepcopy(node)  # changed to deepcopy
                else:
```

```
                if spam:
                    print('CURRENT:{}'.format(current_node))
                    print('RIGHT:{}'.format(current_node.right))
                current_node.right = deepcopy(node)  # changed to deepcopy
        return current_node

    def recombine(self, other):
        self.choose_node(True, other.choose_node)

    def evaluate(self, job, current_time):
        if self.op == CONST:
            return self.val
        elif self.op == BLK_ST:
            return job.task.blocking_start
        elif self.op == BLK_TOT:
            return job.task.blocking_duration
        elif self.op == RELEASE:
            return job.task.release
        elif self.op == PERIOD:
            return job.task.period if job.task.period != 0 else float('Inf')
        elif self.op == EXEC:
            return job.task.exec_time
        elif self.op == DEADLINE:
            return job.task.deadline if job.task.deadline != 0 else float('Inf')
        elif self.op == PLUS:
            return self.left.evaluate(job, current_time) + self.right.evaluate(job, current_time)
        elif self.op == MINUS:
            return self.left.evaluate(job, current_time) - self.right.evaluate(job, current_time)
        elif self.op == MOD:
            right = self.right.evaluate(job, current_time)
            left = self.left.evaluate(job, current_time)
            return left if right == 0 else left % right
        elif self.op == TIMES:
            return self.left.evaluate(job, current_time) * self.right.evaluate(job, current_time)
        elif self.op == DIVIDED_BY:
            right = self.right.evaluate(job, current_time)
            return 0 if right == 0 else self.left.evaluate(job, current_time) / right
        elif self.op == MAX:
            return max(self.left.evaluate(job, current_time), self.right.evaluate(job, current_time))
        elif self.op == MIN:
            return min(self.left.evaluate(job, current_time), self.right.evaluate(job, current_time))
        elif self.op == CURRENT_TIME:
            return current_time
        elif self.op == J_DEADLINE:
            return job.deadline if job.deadline != 0 else float('Inf')
        elif self.op == J_RELEASE:
            return job.release
        elif self.op == NOT_PERIODIC:
            return 0 if job.task.period != 0 else 1 if job.task.deadline != 0 else 2
        else:
            print('HELP')

    def string(self):
        if self.op == CONST:
            return repr(self.val)
```

```
        elif self.op == BLK_ST:
            return 'BLK_ST'
        elif self.op == BLK_TOT:
            return 'BLK_TOT'
        elif self.op == RELEASE:
            return 'TASK_RELEASE'
        elif self.op == PERIOD:
            return 'TASK_PERIOD'
        elif self.op == EXEC:
            return 'TASK_EXEC'
        elif self.op == DEADLINE:
            return 'TASK_DEADLINE'
        elif self.op == PLUS:
            return '(' + self.left.string() + ' + ' + self.right.string() + ')'
        elif self.op == MINUS:
            return '(' + self.left.string() + ' - ' + self.right.string() + ')'
        elif self.op == MOD:
            return '(' + self.left.string() + ' % ' + self.right.string() + ')'
        elif self.op == TIMES:
            return '(' + self.left.string() + ' * ' + self.right.string() + ')'
        elif self.op == DIVIDED_BY:
            return '(' + self.left.string() + ' / ' + self.right.string() + ')'
        elif self.op == MAX:
            return 'MAX(' + self.left.string() + ', ' + self.right.string() + ')'
        elif self.op == MIN:
            return 'MIN(' + self.left.string() + ', ' + self.right.string() + ')'
        elif self.op == CURRENT_TIME:
            return 'TIME'
        elif self.op == J_DEADLINE:
            return 'JOB_DEADLINE'
        elif self.op == J_RELEASE:
            return 'JOB_DEADLINE'
        elif self.op == NOT_PERIODIC:
            return 'PERIODICITY'
        else:
            print('HELP')

class Individual:
    def __init__(self, parsimony = 0.5):
        self.fitness = 0
        self.fitnesses = []
        self.stats = []
        self.root = Node()
        self.size = 0
        self.parsimony = parsimony

    def __lt__(self, other):
        return self.fitness < other.fitness

    def grow(self, depth):
        self.root.grow(depth)

    def full(self, depth):
        self.root.full(depth)
```

```
    def recombine(self, other):
        self.root.recombine(other.root)

    def tree_complexity(self):
        self._size = self.root.size()
        self._use_nonstatic = self.root.uses_nonstatic()
        return 1-1/(self.root.size() * self.parsimony * (2 if self.root.uses_nonstatic() else 1))


    def evaluate(self, problems):
        'this is where the  p a i n  begins'
        fitness_vals = []
        self.fitnesses = []
        for problem in problems:
            hyper_period = problem.hyper_period
            periodic = False
            sporadic = False
            for task in problem.tasks:
                if task.period != 0:
                    periodic = True
                elif task.deadline != 0:
                    sporadic = True
            total_periodic = 0 if periodic else 1
            total_sporadic = 0 if sporadic else 1
            missed_periodic_deadlines = 0
            missed_sporadic_deadlines = 0
            sum_response_time = 1
            job_queue = []
            just_popped = False
            for time in range(hyper_period+1):
                for task in problem.tasks:
                    if task.release == 0 and time == 0 or (task.period + task.release) == time or
(task.period != 0 and (time - task.release) % task.period == 0):
                        job_queue.append(Job(task, time)) # release job
                        if task.period != 0:
                            total_periodic += 1
                        elif task.deadline != 0:
                            total_sporadic += 1
                if just_popped:
                    for job in job_queue:
                        job.priority = self.root.evaluate(job, time)
                    shuffle(job_queue)
                    job_queue.sort()
                if len(job_queue) > 0:
                    if time > job_queue[0].blocking_start and job_queue[0].blocking_duration > 0:
                        job_queue[0].blocking_duration -= 1
                    else:
                        if not just_popped:
                            for job in job_queue:
                                job.priority = self.root.evaluate(job, time)
                            shuffle(job_queue)
                            job_queue.sort()
                        else:
                            just_popped = False
                    job_queue[0].exec_time -= 1
```

```
                    job_queue[0].has_run = True
                    if job_queue[0].exec_time <= 0:
                        if job_queue[0].deadline == 0: # aperiodic job
                            sum_response_time += time - job_queue[0].release
                        if len(job_queue) != 1:
                            job_queue[0] = job_queue[1]
                            job_queue[1] = job_queue[-1]
                            job_queue.pop()
                            just_popped = True
                        else:
                            job_queue.pop()
                            just_popped = True
                    for job in job_queue:
                        if job.deadline != 0 and job.deadline < time and job.exec_time > 0:
                            if job.task.period != 0:
                                missed_periodic_deadlines += 1
                            else:
                                missed_sporadic_deadlines += 1
                            job_queue.remove(job)
                fitness_vals.append((1-missed_periodic_deadlines/total_periodic)**2 +
(1-missed_sporadic_deadlines/total_sporadic) + (1/sum_response_time))
            self.fitnesses.append(((mean(fitness_vals) - self.tree_complexity()) / 2) / 2.5)
            self.stats.append([1-(missed_periodic_deadlines/total_periodic),
1-(missed_sporadic_deadlines/total_sporadic), 1/sum_response_time if sum_response_time != 1 else
None])
        self.fitness = mean(self.fitnesses)
```

10 APPENDIX D

```
# main.py
from GP import GP
from Task import Problem, Task

def main():
   gp = GP()
   problems = []
   task_list = [Task(period=0, release=6, deadline=14, exec_time=5, blk_st=2, blk_dur=2),
                Task(period=0, release=2, deadline=17, exec_time=7, blk_st=2, blk_dur=4),
                Task(period=0, release=0, deadline=18, exec_time=6, blk_st=1, blk_dur=4)]
   problems.append(Problem(task_list, 20))

   task_list = [Task(period=2, deadline=2, exec_time=1),
                Task(period=3, deadline=3, exec_time=2),
                Task(period=7, deadline=7, exec_time=3)]
   problems.append(Problem(task_list, 42))

   task_list = [Task(period=2.5, exec_time=2, deadline=3),
                Task(period=4,   exec_time=1, deadline=4),
                Task(period=5,   exec_time=2, deadline=5)]
   problems.append(Problem(task_list, 20))

   task_list = [Task(period=3, exec_time=1, deadline=3),
                Task(period=4, exec_time=1, deadline=4),
                Task(period=5, exec_time=1, deadline=5),
                Task(exec_time=12),
                Task(exec_time=10)]
   problems.append(Problem(task_list, 60))

   task_list = [Task(period=5, exec_time=6, deadline=5),
                Task(period=6, exec_time=5, deadline=5),
                Task(period=9, exec_time=4, deadline=5),
                Task(exec_time=24),
                Task(exec_time=7, deadline=12)]
   problems.append(Problem(task_list, 90))
   gp.run(problems)

if __name__ == '__main__':
   main()
```