Creation of Simple, Deadline, and Priority Scheduling Algorithms using Genetic Programming

Thomas P. Adams
3710 Rawlins Ave, Suite 940, Dallas, Texas 75219
tadams@stanford.edu / 214-965-0777

Abstract: Genetic programming is used to evolve three different schedulers, each intended to optimize the selection/processing of a job from queues with increasingly complex job data. In the first, a queue of simple jobs with known run times must be processed in a way that minimizes the average wait time of all jobs. In the second, a queue of jobs with known run times and deadlines must be processed in a way to minimize the sum of job delays. In the third, a queue of jobs with known run times, deadlines, and assigned relative priorities must be processed in a way to minimize the sum of job delays relative to the priority of each job. GP is shown to match or outperform the basic scheduling techniques of FIFO and Shortest Job First. For the simple queue case, GP evolves the provably optimal selection mechanism.

1. Introduction

Scheduling involves the allocation of resources in order to perform a specific task or *job*. Scheduling algorithms are designed to allocate the resources among the jobs to be processed. The primary intention of scheduling algorithms vary among environments: when the resources themselves are limited or expensive to operate, algorithms are intended to maximize the utilization of those resources; when an expense or penalty is incurred by delayed jobs, algorithms are commonly intended to minimize the average wait time of jobs. The complexity of the scheduling problem domain increases with the number of resources, job dependencies, and constraints.

Because of the generalized nature of scheduling in terms of resources and tasks, scheduling algorithms are applied in a variety of areas. For example, in computer science, scheduling algorithms are used to determine the next process which the CPU should run; in manufacturing environments, scheduling is used to determine when and which products should be produced; in hospitals, scheduling is used to manage critical resources such as operating rooms, MRI, and other limited-availability equipment. Even among these few examples, it can be seen that sub-optimal scheduling wastes time, resources, and money while jeopardizing profits or even patient health.

This paper explores the ability of Genetic Programming (GP) to evolve programs capable of performing scheduling algorithms under three different models, each of which is briefly described below.

Simple Scheduling
The simplest scheduling model involves a queue of jobs and a single resource capable of processing one job at a time, from beginning to completion. Each job has a known (or estimated) time necessary for completion. In this case, it is the responsibility of the scheduling algorithm to select the next job that is to be processed from the queue. The selected job is then removed from the queue and processed exclusively by the resource. The scheduling algorithm is then presented with the queue of remaining jobs, and the cycle repeats until there are no more jobs to be selected. A job's *wait time* is the time spent in the queue before being selected for processing. After all the jobs in a queue have been processed, the *average wait time* (AWT) can be calculated as the sum of all wait times divided by the number of jobs processed.[1]

Deadline Scheduling
A deadline scheduling model is built from the simple scheduling model above, with additional information that assigns a *deadline* to each job. A deadline is expressed as a number of time units after a job's arrival into the queue after which a job is considered delayed. A job's *delay time* is the amount of time between its selection time and

[1] The simple model assumes that all jobs to be processed exist in the queue before the algorithm begins. In other models, a more complicated simulation will assign arrival times to jobs. The jobs which arrive into the queue while other jobs are being processed will then be available to the scheduling algorithm during the next selection cycle. In terms of analyzing the simple scheduling model as presented, such a simulation is superfluous since (1) the optimal selection mechanism (discussed later) is Shortest Job First and would still need to be discovered, (2) such a mechanism would perform identically whether or not new jobs are added between each selection cycle, and (3) would not be recognized under the simulation used since no operators to provide for the memory of prior jobs processed since they are removed from the queue at selection time.

deadline. Two basic, alternative intentions of a scheduling algorithm in this case are (1) to minimize the average delay time of jobs, or (2) to minimize the number of jobs that experience any delay.

Deadline Scheduling with priority
A deadline scheduling with priority model functions as the deadline scheduling model above, with additional information that assigns a *priority* to each job. The priority value itself is meaningful only in the value it has relative to other priorities. In scheduling with priorities, the primary intention of a scheduling algorithm can be either the minimization of the average delay of jobs or the minimization of the number of jobs that experience delay as above; however, simulation using either intention must specify how jobs with different priorities influence the values to be minimized. The approach used in this paper is to weight the actual job delay using a priority-relative factor.

The preparation, simulation, and results of the three scheduling models are discussed below, each in its own section. All runs were performed on one of two single-processor 866MHz machines with 256 or 512MB of RAM, on the Windows 2000 operating system. Java-based, publicly available **ECJ 8** by Sean Luke, available at www.cs.umd.edu/projects/plus/ec/ecj , was used with IBM's Visual Age for Java.

2. Simple Scheduling

A program capable of simple scheduling as defined above has as its input a queue of jobs, each with a run time. The output of the program's invocation is a single job, that is, the job to be processed next. The algorithm is repeatedly invoked until the queue is empty, at which time processing statistics are used to assign a fitness to the program. The simulation of the simple scheduling environment is shown in Figure 1.

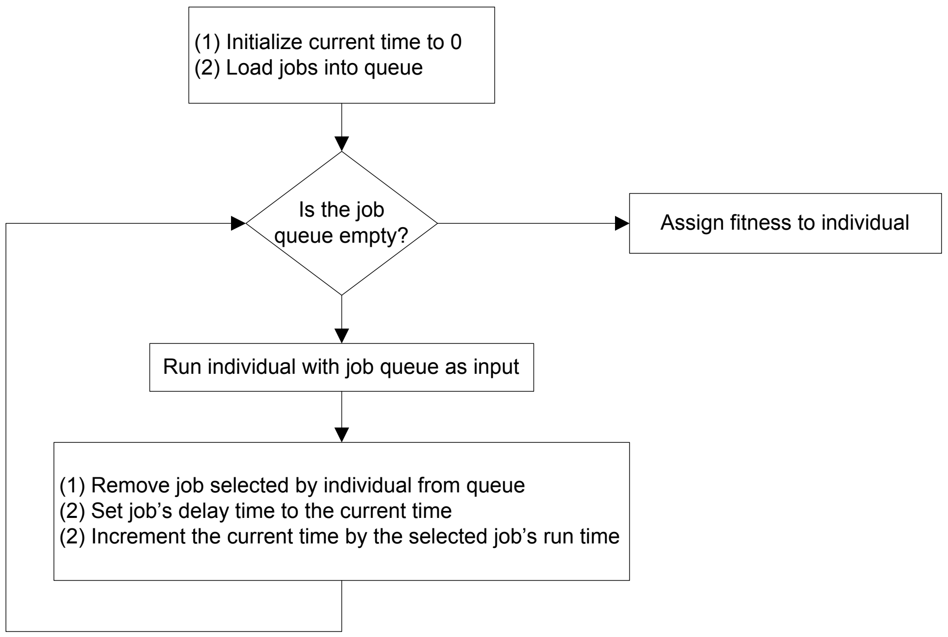
(1) Initialize current time to 0
(2) Load jobs into queue
Is the job queue empty?
Assign fitness to individual
Run individual with job queue as input
(1) Remove job selected by individual from queue
(2) Set job's delay time to the current time
(2) Increment the current time by the selected job's run time

Figure 1. Simulation environment for evaluating simple scheduler individuals

Simple Scheduling Preparatory Steps
<u>Program Architecture</u>
From the beginning of this project, it was obvious that any evolved program capable of making an optimal selection from a job queue would need to examine every job in the queue before determining which job should be selected. After early unsuccessful attempts (discussed in detail below) to endow individuals with the ability to traverse the job queue "on their own" by making iteration functions available to the individuals themselves, it became obvious that such an approach monumentally increases the time, if ever, to evolve functional individuals. As a result, the structure of the individuals themselves was fixed such that each contained independently evolved branches: an iteration branch and a selection branch.

When the individual is presented with a job queue, the iteration branch is invoked exactly once for each job currently in the queue. Then the selection branch is invoked once, the result of which is the job to be selected. The concept of this *restricted iteration* architecture (Koza 1994), is that the individual is "shown" each job one at a time, and may "remember" relevant information by storing jobs into any of its memory locations (discussed below). After all iterations are complete, the selection branch uses the memory locations presumably set by the iteration branch to select the job to be processed.

Terminal Set
Before discussing the terminal and function set, it is necessary to understand the data structure that is evaluated at each node.

The purpose of the selection branch is to select one of the jobs from the queue; thus, the output of every node (since any may be at the top of the selection branch) must be interpretable directly or indirectly as referring to one of the jobs from the queue. Many early attempts to interpret any integral value whatsoever produced by the selection branch (i.e., by applying the mod operator with the queue's current size) were woefully inadequate. On advisement, a data structure that itself always identifies a job while the value portion of the data structure -- in this case the runtime of the job -- provides the function arguments used by the program nodes, was used. Thus, every terminal is actually a job-runtime data structure, and every function set examines only the runtime data portion of the structure. Only the simulator itself, after selection has been made, examines the job-identifying portion of the data structure. In this way, the output of any individual, however unfit, always yields a selectable job.

The terminal set consists of five memory locations and one node representing the current job (i.e., the job which the iteration branch is being "shown"). Each of the memory locations are initialized to the first job in the queue. Such initialization effectively translates a null (uninitialized) memory location arbitrarily into the first job in the queue. Such arbitrary assignments are liable to be recognized and exploited by GP, but without detrimental effects in this case. Similar to protected division, the initialization provides for meaningful processing while not biasing individuals to an optimal solution. If anything, individuals are biased to a first-in/first-out (FIFO) selection mechanism which would never produce the optimal results from the data used.

Thus, terminal set for the simple scheduler is

$$T_s = \{M0, M1, M2, M3, M4, CUR\}$$

where M*x* is the memory location *x*, and CUR denotes a data structure storing the identity and runtime of the current job.

The entire terminal set is used in the iteration branch. Only the memory location terminals are used in the selection branch.

Function Set
Two types of functions are necessary to provide an individual with the capability to compare jobs. First, memory read and write functions are necessary to enable remembering previously seen jobs from the queue, and comparative functions are necessary to compare the runtimes of remembered jobs.

Thus, the function set is

$$F_s = \{SETM0, SETM1, SETM2, SETM3, SETM4, IFLT, IFGT\}$$

SETM*x* takes one argument and sets memory location *x* to that value.

IFLT/GT takes four arguments: left, right, if, else. If the runtime of the job returned by the left branch is GT/LT the job returned by the right branch, the result of the "if" subtree is evaluated and returned; otherwise, the "else" branch is evaluated and returned.

The entire function set is used in the iteration branch. Only the IFLT and IFGT functions are used in the selection branch.

Originally, two additional operators, Larger/Smaller (>, <) each taking two arguments and returning the job with the larger/smaller runtime were used. While their inclusion admittedly provided individuals with capability redundancies, they proved to be neither necessary nor material; however, it was important to establish early positive results, even if heavy-handedly so. These operators *were* included in the other scheduling models to follow.

Fitness Measure
The value of the fitness measure of each individual was the sum of wait times after processing a queue of 32 jobs, with runtimes randomly distributed between 1 and 1000. The GP software used in this project ranks 0 as the ideal fitness and positive infinity as the worst.

It should be noted that a fitness case of a single queue of 32 jobs, from an individual's point of view, is effectively 32 different fitness cases. The individual is presented with a queue of 32 jobs and makes a selection. The memory locations are then initialized and the individual is presented with a job queue of 31 jobs, then 30, 29, etc. until the job queue is empty. While each subsequent trial is related from the point of view of the simulator (since it tabulates the results of all 32 invocations), there is no memory from one selection to the next on the part of the individual. The same logic is applied to each of the 32 differently sized queues, and the fitness is a measure of performance against all 32 trials.

Given a job queue of 32 jobs, there are 32! or 2.6E+35 different ways to order the processing of jobs.

Additionally, the best individual is validated against a formerly unseen queue of 32 jobs. As discussed in the results section, the nature of the optimal solution for a simple scheduling algorithm makes the identification of an ideal individual unambiguous.

A hit was recorded when the job selected by the individual was the job known to the simulator to be the ideal selection.

Control parameters
The genetic operators of crossover (90%) and reproduction (10%) were used on a population of 1024 individuals.

Termination
The run was terminated when either an individual reached 32 hits or 50 generations were processed. An individual scoring 32 hits indicates that it always selects the optimal job for processing.

Table 1. Tableau for the Simple Scheduling problem

Objective:	To process a queue such that the average wait time of jobs is minimized.
Terminal set, iteration branch:	M0, M1, M2, M3, M4, CUR
Terminal set, selection branch:	M0, M1, M2, M3, M4
Function set, iteration branch:	SETM0, SETM1, SETM2, SETM3, SETM4, IFLT, IFGT
Function set, selection branch:	IFLT, IFGT
Fitness cases:	A queue of 32 jobs
Fitness:	The sum of wait times of all jobs
Parameters:	M=1024, G=50
Termination criteria:	Ideal individual found or 50 generations processed.
Result designation:	The best individual of run.

Simple Scheduling Results
It is known that selecting the job with the minimum runtime -- the Shortest Job First (SJF) algorithm -- is the optimal selection technique under the circumstances represented by this simple scheduling problem (Silberschatz 1998). To help understand the program architecture and the terminal and function set, an example of a specifically

designed ideal individual is presented in Figure 2. Here, M1 of the iteration branch is used to store the job with the lowest runtime yet seen. The selection branch simply selects the remembered job.

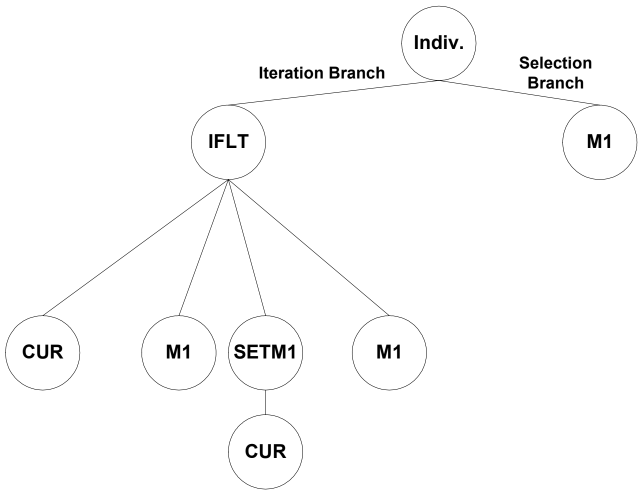

Figure 2. Specifically designed ideal individual for Simple Scheduling

In the run of GP, an ideal individual, validated against a previously unseen job queue, was found after only 2 generations. The individual is:

```
Tree 0:
 M0
Tree 1:
  (IFGT (SETM4 (SETM3 (SETM3 (SETM1 M2))))
      (IFGT (IFGT (IFGT (IFLT M1 M3 M3 M4) (IFLT
          M1 M3 M2 CUR) (IFGT M1 M0 M0 M2) (IFLT M2
          M3 M4 M1)) (SETM0 (SETM4 M1)) (SETM2 (SETM1
          M0)) (SETM4 (IFGT M0 M3 M3 M0))) (SETM4 (SETM3
           (SETM0 CUR))) (SETM0 (SETM0 (SETM0 CUR)))
           (SETM3 (SETM0 (SETM2 M1)))) (IFGT (SETM2
      (IFLT (SETM2 M4) (SETM3 M0) (IFGT M4 M4 M3
          M2) (IFGT M3 CUR M0 M1))) (SETM3 (SETM3 (SETM2
      M0))) (SETM1 (SETM4 (SETM2 CUR))) (SETM1
      (SETM1 (SETM4 M2)))) (IFLT (IFLT (SETM2 (SETM4
      M0)) (SETM2 (SETM1 M0)) (SETM3 (SETM1 M0))
      (SETM2 (SETM3 M1))) (SETM3 (SETM1 (SETM4
      M3))) (SETM1 (SETM2 (SETM4 M2))) (SETM0 (SETM3
      (SETM3 M1)))))
```

In this case, the individual stores the job with the minimum runtime on each iteration into location M0. The selection branch simply returns the job stored in location M0.

When the same run was made against job queues of varying lengths, and using more than one job queue, the results were comparable. The explanation for this seems to be that since the retention of the shortest job by the iteration branch is a technique discovered fairly quickly, its application immediately yields optimal results on cases of any size. Although only mildly interesting since an ideal individual was found so quickly, the results validated the architecture and data structure approach that would be applied to the more complicated models.

3. Deadline Scheduling

The deadline scheduling model adds an additional attribute to each job, its *deadline*. Jobs which are not completed in the simulated environment by the deadline are assigned a delay time; jobs completed on time or before their deadline are assigned a delay time of zero. An individual's fitness is the sum of all delays.

Under this set of conditions, individuals must now perform a calculation in order to select the appropriate job. Rather than merely store the shortest job, an individual must now consider the current time, the time at which a job if selected will be completed, and the delay that a job will experience. It must then store the relevant information in memory locations for use by subsequent iterations.

To endow individuals with this capability, the data structure was modified and additional functions were added. The data structure now becomes:

{ job identifier, runtime, deadline, value }

The *value* portion of the data structure is the argument that each function reads in order to operate on and sets before returning; it is initialized to the job's runtime.

Terminal and Function Set

The terminal set for this problem is the same used in simple scheduling, that is:

$$T_s = \{M0, M1, M2, M3, M4, CUR\}$$

The function set is:

$$F_s = \{SETM0, SETM1, SETM2, SETM3, SETM4, IFLT, IFGT, >, <, +, -, RT, REQ, TIME\}$$

Where previously mentioned functions have the same definition, and + and – take their natural meanings with 2 arguments. Both functions arbitrarily set the value portion of the left branch and return its associated structure.

RT, REQ, and TIME each takes a single argument, sets the *value* portion of the data structure to the runtime, deadline, or simulator's current time respectively, and returns the structure.

As an example, Figure 3 shows how an individual could calculate the estimated completion of the current job, and store that estimate into memory location M2 in the iteration branch. In this example, the selection branch blindly selects the job stored in M0, thereby ignoring the potentially useful calculation performed by the iteration branch.

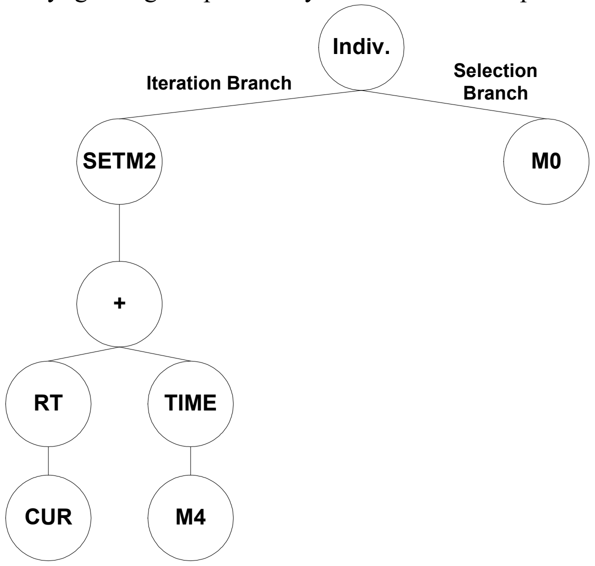

Figure 3. Individual capable of projecting the current job's completion time for Deadline Scheduling

Fitness Measure
The value of the fitness measure is the sum of all delays after processing 5 queues of 32 jobs each. Each queue is populated with jobs having a randomly assigned runtime between 1 and 1000. The deadline is randomly assigned to a value between 5 time units greater than its runtime and the sum of all runtimes for the queue (i.e., the time it takes to run all jobs regardless of their ordering).

Table 2 summarizes the run parameters.

Table 2. Tableau for the Deadline Scheduling problem

Objective:	To process a queue such that the sum of all delay times is minimized.
Terminal set, iteration branch:	M0, M1, M2, M3, M4, CUR
Terminal set, selection branch:	M0, M1, M2, M3, M4
Function set, iteration branch:	SETM0, SETM1, SETM2, SETM3, SETM4, IFLT, IFGT, >, <, +, -, RT, REQ, TIME
Function set, selection branch:	IFLT, IFGT, >, <, +, -, RT, REQ, TIME
Fitness cases:	5 queues of 32 jobs each
Fitness:	The sum of all delay times; a hit is scored for each job completed prior to its deadline
Parameters:	M=2048, G=50
Termination criteria:	50 generations processed
Result designation:	The best individual of run.

Deadline Scheduling Results
The best individual of the run was a 12-point selection branch, 153-point iteration branch individual which scored 123 hits. Although the maximum number of hits is 160, it must be recognized that the random generation of deadlines does not guarantee that there exists a schedule such that all jobs meet their deadline (in fact that would be unlikely); instead, it ensures that an optimal selection of jobs with only a few delayed jobs likely exists.

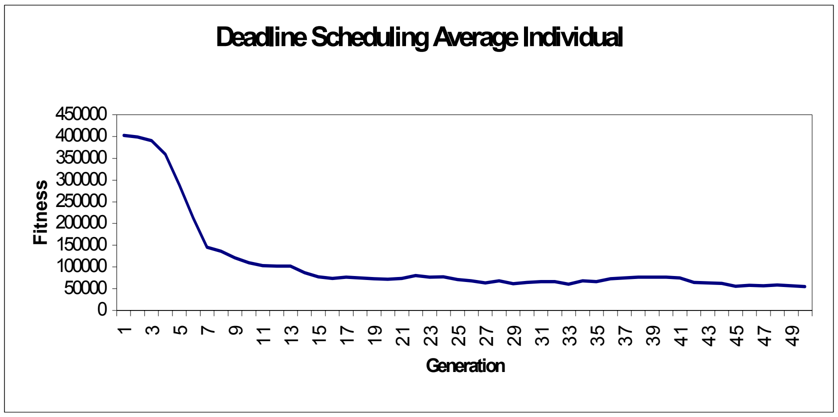

Figure 4: Average Individual for Deadline Scheduling

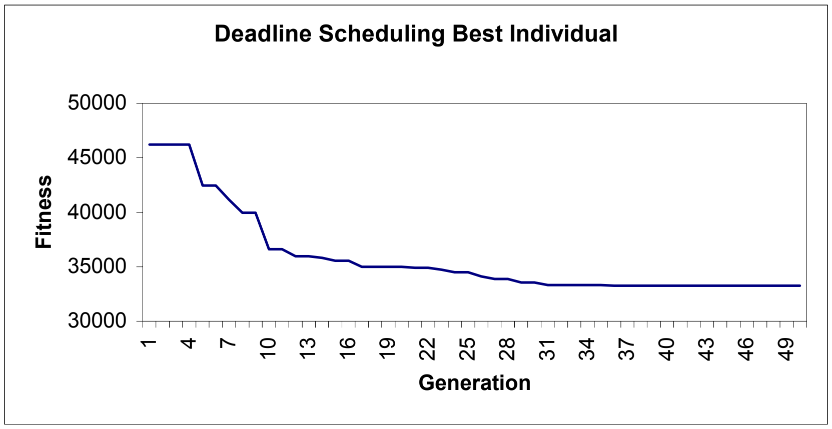

Figure 5: Best Individual for Deadline Scheduling

The best individual was achieved at Generation 36. Analysis of the selection data reveals that the best individual achieves its performance by having evolved the common sense notion that jobs with earlier deadlines should be scheduled first. Additional runs were performed in which the randomly selected deadline was some fraction of the total run times, thereby making it inevitable that a larger percentage jobs would miss their deadline. Similar results were observed under these conditions, namely, that jobs with earlier deadlines were selected first.

4. Deadline Scheduling with Priority

The deadline scheduling with priority model adds an additional attribute of *priority* to each job. Under this model, jobs continue to have deadlines, but jobs with higher priorities incur a higher delay penalty than those with lower priorities. A job which is not completed by the deadline is assigned a calculated or *weighted delay* equal to its actual delay multiplied by (actual_delay * priority / max_priority). The precise formula, which means that the calculated delay of a job with the maximum priority will be the square of its actual delay, was settled upon after several trial runs of various weighting ratios. It is somewhat arbitrary and merely reflects one way to indicate to evolving individuals that failing to schedule a high priority job will be penalized. If the penalty is too low, the data does not allow GP to give it due consideration (within 100 generations) and merely causes rediscovery of the deadline scheduling technique. If the penalty is too high (for example, cubic), no recognizable pattern or real improvement emerged within 25 generations and the run was terminated without further investigation.

A priority variable was added to the data structure and the function PRI was added to access this value, and multiplication and division functions were added to make the consideration of weighted delays possible. Two additional memory locations were added. Finally, an additional restricted iteration branch was added. Table 3 summarizes the run parameters.

Table 3. Tableau for the Deadline Scheduling with Priority problem

Objective:	To process a queue such that the sum of all weighted delay times is minimized.
Terminal set, iteration branch 1 and 2:	M0, M1, M2, M3, M4, M5, M6, CUR
Terminal set, selection branch:	M0, M1, M2, M3, M4, M5, M6
Function set, iteration branch 1 and 2:	SETM0, SETM1, SETM2, SETM3, SETM4, IFLT, IFGT, >, <, +, -, *, /, RT, REQ, PRI, TIME
Function set, selection branch:	IFLT, IFGT, >, <, +, -, *, /, RT, REQ, PRI, TIME
Fitness cases:	10 queues of 32 jobs each
Fitness:	The sum of all weighted delay times; a hit is scored for each job completed prior to its deadline
Parameters:	M=4096, G=50
Termination criteria:	50 generations processed
Result designation:	The best individual of run.

Deadline Scheduling with Priority Results

The best individual of the run contained 62-points in the selection branch, 51-points in the first iteration branch and 18-points in the second iteration branch. It achieved 99 hits out of 320 (as a theoretical maximum not actually achievable by any algorithm). Table 4 shows the best individual performance against one of the ten job queues.

Table 4. Best-of-run performance against one job queue

Time	Runtime	Deadline	Priority	Wt Delay	Time	Runtime	Deadline	Priority	Wt Delay
0	2	453	18	0	766	33	114	0	685
2	44	623	10	0	799	92	809	0	82
46	70	214	10	0	891	17	42	0	866
116	25	735	7	0	908	83	791	0	200
141	7	559	14	0	991	5	237	0	759
148	41	316	19	0	996	51	284	0	763
189	45	642	8	0	1047	92	454	0	685
234	88	561	8	0	1139	35	597	0	577
322	8	370	18	0	1174	30	132	0	1072
330	48	578	17	0	1204	61	526	0	739
378	64	547	14	0	1265	37	397	0	905
442	99	460	8	2624	1302	74	626	0	750
541	89	491	7	6762	1376	7	683	0	700
630	18	600	5	576	1383	32	521	0	894
648	58	455	5	1570	1415	4	509	0	910
706	60	240	5	69169	1419	31	524	0	926

Here the best individual selects higher priority jobs first but not universally. The individual has learned that higher priority jobs are more important but that deadlines should still be considered. Although the scheduling does not appear to be optimal – for example, consider the high penalty paid at time 706 for neglecting the job with a runtime of 240 – it does reveal a rudimentary tradeoff strategy. The absolute values that priorities assume are not inherently meaningful to GP as discovered by at least two other runs in which the range of priorities were altered slightly from the one described here. GP is eventually able to recognize the general principle that higher priority jobs are more important. It is not clear that the ideal individual has discovered the exact penalization formula used to produce the weighted delay, although the functions necessary to derive it are available.

Problems Encountered

Early runs demonstrated the inadequacy of two approaches: (1) unrestricted iteration and (2) direct terminal values without data structures.

The initial approach to the queue iteration problem was attempted with the definition of a single-argument ITER function node, which would invoke its sub-tree once for every job currently in the queue. Since ITER was treated as any other function, it would occur several times in any given individual, and would frequently be nested two or more times. This nesting made the evaluation of individuals prohibitive and several runs had to be terminated. In an attempt to shorten evaluation time, the function itself was altered to only allow some maximum number of nested iterations over the queue. However, this proved unhelpful as well, since the evaluation time was still prohibitive, and it was advised that the introduction of arbitrary constants of this sort may inadvertently acquire significance and influence the course of evolution.

Early approaches also used values such as the current queue position QPOS, the runtime of the current job RT, and constants zero and one as terminals. The function set operated on single integral values only, and memory locations stored only these integral values. In sharp contrast to the data structure approach which yielded an ideal individual in only 2 generations, the purely numeric approach was unsuccessful over many different runs of different population sizes and was eventually abandoned. It appears that the single iteration branch had difficulty sorting out the incommensurate terms (a job's position in the queue added to its runtime is meaningless) to the extent that operations against runtimes (one terminal) should be performed only in order to preserve a job's queue position (another terminal) for the selection branch. Because of this, the queue position itself was subject to operators such as addition, which meant that the result of the selection branch was rarely even a value that indicated a position in the queue. In order to produce a meaningful selection, the queue position was calculated using the value actually produced by the selection branch mod the job queue size. This produced, even after many generations, individuals that simply made what appeared to be a random selection from the queue.

Conclusions

These experiments demonstrate that GP with restricted iteration and memory is able to produce individuals capable of considering an unspecified number of entities each with multiple attributes. Further, it is able to make considerations and calculations using those attributes in order identify a single entity for selection. By utilizing memory, GP is able to infer attribute relationships among the entities being considered. For example, in deadline scheduling, the ideal individual is able to compute that an entity's runtime is related to its deadline in consideration of the current time, which value changes at each selection opportunity. Further, GP is able to discover that priorities, although the range of values is itself arbitrary, represent relative importance even though their exact mathematical nature is determined by the simulator's delay-weighting penalty formula and may not have been explicitly discovered.

Future Work

Given that GP successfully operates on a group of entities with multiple attributes, further studies could explore how GP behaves given more complicated scheduling models and job descriptions. There are several directions in which to add complexity to the scheduling models to more closely simulate real world dilemmas: (1) with explicitly stated relationships between the entities, say with the notion of precedence wherein some jobs require the completion of others prior to selection (perhaps via pointers to other jobs), how quickly would GP discover them? (2) How well would GP identify generalized constraints placed on any of the attributes? (3) Given the availability of more than one resource, how soon would GP maximize their parallel use? In general, the introduction of more complex job descriptions and resource constraints would provide interesting fields for further study.

References

Koza, John R. 1994. *Genetic Programming II Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press.

Silberschatz, Abraham. 1998. *Operating System Concepts*. Addison-Wesley.