Partitioning Sets of Schedules with Genetic Algorithms

Joshua Essex

CS 499
Submitted: Tuesday May 17, 2011

Statement of Topic

This project is an attempt to devise a method for evolving optimal groupings of individuals based upon sameness in terms of pre-defined schedules, or availability. Such an algorithm would take in as input a list of individuals (and each individuals' respective schedule) and the size of desired groups and return as output a grouping of these individuals such that the groups are configured for a maximal matching of availability.

Topic Area

The problem is one of discrete optimization: there exists a set of binary sequences which must be grouped based upon sameness. In this optimization problem, there are a finite amount of unique partitions of the total set and the optimal solution is the one which has a maximal fitness, where fitness represents the aforementioned sameness in availability. Each individual is represented by their schedule, which in turn can be modeled by a string of 0s and 1s, corresponding to unavailable and available for a given time span, respectively. This modeling of individuals as strings of bits lends itself well to a Genetic Algorithm.

Genetic Algorithms are a subset of Evolutionary Algorithms which can be used to search for an optimal solution by utilizing processes and operations found in natural evolution. GAs require two primary components: a fitness function to determine the relative worth of a candidate solution and a genetic representation of candidate solutions. This is part of what makes the project interesting: determining the representation of the problem space and the design of the different evolutionary operations.

It is also interesting in that it has a solid real world application: the idea was inspired by a task in which a class of students had to be grouped, by hand, based on their schedules. Such optimization is remarkably tedious and considerably challenging for a human. Automating a traditionally human-performed task represents an advancement. And of course, any methodology by which optimization can

be accurately and quickly performed is an important contribution to applied mathematics.

Prior Research

On one hand, this objective is reminiscent of scheduling problems. After all, the problem involves optimizing a grouping of schedules, which in many ways can be thought of as sequences of tasks. In such a view, this problem can be viewed as one of scheduling wherein each grouping of individuals is a machine, each individual is a sequence of tasks and each period of availability within a given individual's schedule is an actual task. Genetic Algorithms, in addition to a variety of other brands of Evolutionary Algorithms, have been applied to scheduling problems historically [1]. GAs have been utilized with a multitude of crossover operations and representations, including direct representation via encoding of the schedule itself and indirect representation via encoding of merely the order of placement within said schedule. A number of routes over this problem have been traced; however, these have mostly applied to traditional scheduling problems, such as Job Shop Scheduling, as opposed to the problem of this project which does not fit so snuggly within the standard scheduling mold.

In another way, this problem can be thought of simply as a partitioning problem: the schedules are merely a medium through which partitioning of a set is performed. When thought of in this sense, the dilemma of this project becomes quite similar to the Equal Piles Problem of partitioning a set of numbers into subsets such that the sum of the numbers in a given subset are as equal as possible to that of other subsets. Genetic Algorithms have been applied to this problem with a good deal of success [2]. Though the idea of bit-string representation of individuals does not apply as well to the Equal Piles Problem, integer representation has been used and thus it still resides within the domain of discrete optimization. This project can thus be thought of in an Equal Piles sense by simply switching from optimizing differences in sums to optimizing differences in sequences. The idea of partitioning a set into optimal subsets remains as the dominant feature of the landscape.

Approach

The structure of the problem is the one of the most critical factors to consider when designing an approach to finding its solution. The input data essentially consists of a set of individuals whom are available at certain times and unavailable at others. Because the goal is to group these individuals such that their availabilities match up optimally, it is remarkably convenient to think of each individual as a binary string, where each bit corresponds to a discrete, evenly-spaced chunk of time. An entire individual schedule can be represented as a simple string of 0s and 1s where each 1 represents a chunk of time where the individual is available and each 0 represents time where the individual in unavailable.

For instance, for an individual who is available from 3:00pm to 4:00pm and again from 5:00pm to 6:00pm, assuming the discrete time chunks are 30 minutes periods, the total period of time from 3:00pm to 6:00pm can be written as 110011. Another individual with an availability string of 110001 during the same time period would be relatively fit to the first individual, whereas an individual with an availability string of 001100 would be perfectly *unfit* to said individual.

With this in mind, the structure of the project can be thought of as evolution of a partitioning of binary strings (individuals) such that the strings in each group will be maximally similar (the individuals will be optimally arranged). The input for the algorithm can be a total of n strings and an integer k which represents the size of the groups to divide individuals into. The individuals are randomly partitioned into n/k groups and the evolution begins. This structuring is particularly adept to a Genetic Algorithm: the encoding of genotypes as binary strings and of phenotypes as a partitioning of binary strings is very well suited to the structure of GAs.

However, it must be stated that the phenotypes are not groups of individuals: remember that the desire is a *set* of these groups. Thus, the phenotypes (the candidate solutions which make up the population) are groups of partitions. For instance, consider an input of six individuals (A, B, C, D, E, F)

and a desired group size of two (leading to three groups of two). In this problem, the genotypes are the binary strings which represent the individuals A through F and two example phenotypes would be the partitions of {(A,C), (D,F), (B,E)} and {(D,E), (A,B), (C,F)}. The goal is to find the phenotype (the group of groups) which is optimally configured. To do this, we will apply a genetic algorithm which consists of a population of many candidate solutions and perform evolutionary techniques, such as selection, recombination, mutation and fitness evaluation, to converge toward an evolved optimum.

The nature of this problem, however, is quite diverse and multifaceted. There are many potential variations and considerations to be made about how exactly one wishes to proceed. For instance, the original inspiration for the project consisted of students filling out time sheets for an entire week. Seven days of 48 blocks of 30 minutes a piece leads to 336-bit strings for each individual, whereas limiting the scope of the problem to one day from midnight to midnight leads to only 48-bit strings.

Additionally, determination of fitness has numerous choices. As an example, say that there are only two groups of individuals in a phenotype. Say that one candidate solution has a group with extremely high local fitness (the individual subset is very well configured) and another group with equally low local fitness (the individual subset is very poorly configured). Say that another candidate solution has two groups with equally average local fitness, and furthermore that both candidate solutions have equal total fitness values. Which candidate solution is *better*? The one with the more balanced solution, the one with the more radical solution, or are they objectively equal?

Some initial considerations being made in this project include the following:

- Only one day is used for each individual, and the chunks of time are equal to 30 minutes, meaning each individual is modeled by a 48-bit string of 0s and 1s.
- Fitness will be evaluated without any subjective input, which is to say that two separately configured candidate solutions with equal fitness values are simply equal; nothing more or less.
- The algorithm will only accept and work with input values which divide evenly; that is to say, the subset size must divide evenly into the total amount of individuals so that each subset is of equal size.

Genetic Algorithm

The process begins by first reading in a list of schedules (the input for the algorithm) and initializing a population of random partition candidates. The algorithm will then operate by altering its population at each new generation, and checking the fitness of its best candidate at the end of each generation. If this fitness is better than the best recorded fitness so far, we take note of it. Once we have reached an optimal solution or have reached the maximum amount of generations, we report our best recorded partition candidate.

Updating of a population in a given generation consists of separating an elite portion of the population from the rest, breeding the rest of the population through crossover to produce a population of children, recombining the elite with the children to produce a new population of equal size to the previous generation, sorting the population by fitness, performing mutation based on placement within the population and finally sorting the population by fitness again. At this stage, the population has been passed through a single generation and is ready for the next generation.

The crossover operation will operate thusly: choose two partition parents through a deterministic tournament selection, establish a pointer to the best individual subset in each parent partition, choose the best subset of the two subsets currently pointed to in each parent to place in the child and advance that parent's pointer to the next best subset. Continue to choose subsets to place into the child in this manner until the correct amount of subsets are in the child. Because each parent has only one of each individual schedule across all of its subsets, a child may have up to two of a given schedule across its own subsets (if it inherited it once from parent X and once from parent Y). For any duplicate schedule, we choose the lesser (in terms of fitness) of the two subsets which feature the duplicate and replace the duplicate with a random choice of one of the schedules which must have been left out. This is the same crossover as seen in the GA for the Equal Piles Problem highlighted in [2], with the difference that the choice of the originally left out schedule is random as opposed to heuristic in nature. This process is designed to help breed the best subsets into a child with a bit of genetic

variety to combat population stagnation.

The selection operation for crossover is a deterministic tournament selection. We select a number of individuals at random and then select the best of these individuals with a probability of 1, which is to say we automatically take the most fit of the randomly chosen individuals. We perform this process twice for each crossover, once to select parent X and once to select parent Y. This selection process was chosen for its ease and convenience. Though a weighted roulette wheel selection process may ensure that more fit parents are selected on average, a large enough number of candidates for a given tournament helps to make this more even. Additionally, this problem is one of minimization (lower fitness levels are superior) and the typical roulette wheel relies on the problem being one of maximization, due to the way that probabilities are calculated. For these reasons, the algorithm currently uses a tournament selection for the crossover operation.

The mutation operation is to simply choose two random subsets, choose a random schedule in each subset and swap the two; it is the equivalent of randomly swapping two people between groups. Choosing whether to mutate and how many times to mutate is more interesting. To begin a round of mutation on a population, it must first be sorted so that may perform a weighted randomized selection of mutation. For the top 10% of candidate partitions, we perform one mutation operation with a chance of 10%. For the next 30% of candidate partitions, we perform four mutation operations, each with a chance of 50%. For the final 60% of candidate partitions, we perform ten mutation operations, each with a chance of 50%. Thus, for the top 10% of the most fit individuals in the population, there is only a 10% that each will be mutated. For the next 30% we expect two mutations steps to take place and for the final 60% we expect five mutation steps to take place, on average. This methodology for mutation is aggressive in that the less fit of the individuals are mutated strongly in a desperate sort of effort.

Perhaps the most critical factor of the algorithm, however, is determining how fit a candidate solution is: the fitness function. To determine the fitness of a partition, the local fitness of each subset is first calculated. A subset's local fitness is a simple measure of similarity between all schedule strings:

we check the amount of 1s (or 0s, it is ultimately irrelevant) in a given index for each respective string in the set and sum values based on this count for each index. For instance, say a subset of size four has three schedules with a 1 at a given index and one schedule with a 0 at the same index. The total of three 1s is assigned a value of one (four minus three) and added to the sum of our values. If there had been a total of only one 1, the assigned value would still have been one. Essentially, if the amount of ones is greater than half of the subset size, we add that amount to our sum, and if the amount of ones is less than or equal to half the subset size, we add the subset size minus that amount to our sum. This sum is a subset's local fitness score. The local fitness for each subset is gathered and an Euclidean norm is then calculated as the partition's fitness: fitness is equal to the square root of the sum of the squares of each subset fitness score. This is the same fitness function as the genetic algorithm highlighted in [2].

Result Evaluation

Since the goal for the algorithm is to determine an optimal grouping of schedules, the most straight-forward approach to evaluation of the algorithm is to provide test data sets for which an optimal solution is known. The output corresponding to the sample input sets can be compared against the known optimal solutions to determine if the algorithm is functioning properly. The algorithm can then be compared against alternate algorithms and approaches. To do so, however, a baseline algorithm with specified parameterization must be determined as the point of comparison.

A systematic approach to this process was developed wherein four different parameters, population size, elitism ratio, selection tournament size and mutation level, were configured to different values for testing purposes. Each of the four parameters was given two different candidate values, leading to a total of $2^4 = 16$ different parameter configurations. Each of these candidate configurations was tested to determine the overall best parameter configuration for the algorithm.

To provide test data from which to determine which configuration was “best”, ten test data sets were created for which the optimal solution was known. These tests are described in the Appendix of

this report. Each baseline candidate was run ten times for each of the ten tests for a total of 100 individual test runs per candidate. For each individual run, the output was examined for optimality, fitness error and the average amount of individuals to reach optimality. The fitness error is the difference between the fitness value of the best candidate produced by the algorithm and that of the optimal solution; a test run which produces the optimal solution thus has a fitness error of exactly 0. The average amount of individuals to reach optimality is equal to the amount of generations that the algorithm requires to reach an optimal solution multiplied by the population size. In the event that a test run does not produce an optimal solution, the amount of generations to reach optimality is equal to the maximum amount of generations that the algorithm is allowed to run, which is set to 250.

After testing each baseline candidate, the sixteen different candidates were examined to determine how well the algorithm performs in general. The candidate which performed the best throughout testing was then selected as the baseline algorithm to be compared against alternative algorithms.

Results

The ten different tests which were run can be described in terms of varying schedule size, subset size, subset total and schedule similarity. The tests were designed to provide a wide variety of landscapes in order to more thoroughly evaluate the algorithm. The sixteen baseline candidates were created from the possible combinations of parameter configurations:

- population size = { 50, 150 }
- elitism = { 0.00, 0.07 }
- tournament size = { 4, 8 }
- mutation rate = { 'low', 'high' }, where:
  - '*low*' means the top 10% of individuals have 1 mutation step at a mutation rate of 0.1, the next 30% have 4 mutation steps at a mutation rate of 0.5 and the final 60% have 10 mutation steps at a mutation rate of 0.5
  - '*high*' means the top 10% of individuals have 2 mutation steps at a mutation rate of 0.25, the next 30% have 6 mutation steps at a mutation rate of 0.5 and the final 60% have 12 mutation steps at a mutation rate of 0.75

Each of the sixteen baseline candidates were evaluated over ten runs on each of the ten tests for a total of 100 test runs per candidate. The candidates are listed on the far left as a set of adjustable parameters {parameter size, elitism, tournament size, mutation rates}. For instance, the baseline candidate with a population size of 50, elitism of 0.07, a tournament size of 8 and 'high' mutation rates is listed as {50, 0.07, 8, 'high'}. The results of these tests are shown below:

Baseline Candidate ({population size, elitism, tournament size, mutation rates})	Optimality Ratio	Average Fitness Error	Average Generations to Optimality	Average Individuals to Optimality
{50, 0.00, 4, 'low'}	84/100	1.8692	82.49	4124.5
{50, 0.00, 4, 'high'}	61/100	5.1167	118.15	5925.5
{50, 0.00, 8, 'low'}	83/100	1.6609	96.28	4814
{50, 0.00, 8, 'high'}	74/100	1.4968	102.21	5110.5
{50, 0.07 4, 'low'}	77/100	2.7132	103.58	5179
{50, 0.07, 4, 'high'}	58/100	6.6595	133.75	6687.5
{50, 0.07, 8, 'low'}	83/100	1.9434	92.32	4616
{50, 0.07, 8, 'high'}	63/100	5.2278	118.47	5923.5
{150, 0.00, 4, 'low'}	86/100	2.2599	76.77	11515.5
{150, 0.00, 4, 'high'}	64/100	5.4374	113.77	17065.5
{150, 0.00, 8, 'low'}	91/100	0.9757	72.4	10860
{150, 0.00, 8, 'high'}	71/100	1.9352	78.53	11779.5
{150, 0.07, 4, 'low'}	69/100	3.3850	106.11	15916.5
{150, 0.07, 4, 'high'}	62/100	5.3601	115.85	17377.5
{150, 0.07, 8, 'low'}	79/100	2.3903	95.2	14280
{150, 0.07, 8, 'high'}	67/100	4.5434	103.18	15477

The results are generally positive for most candidates, with optimality higher than 70% for nine of the sixteen configurations and higher than 80% for five of the sixteen. With regards to the individual tests, all sixteen candidates were able to find an optimal solution on each of the runs to five of the ten test sets. The other five tests which were not solved with 100% optimality tended to be those with the largest subset counts or the highest level of similarity between schedules. Tests with comparatively smaller subset counts or more diverse schedules were easier landscapes for the algorithm to converge upon optimality (a sample of individual test data can be seen in the Appendix of this report). However,

the five candidates with optimality rates higher than 80% were able to reach optimality on these more challenging tests with relatively high ratios. The very best baseline candidates were able to find optimal partitions on the most challenging landscapes with high success.

There are some trends to be observed in the resulting data which can help to determine a baseline. For instance, one can see that all candidates with mutation rates of 'high' performed considerably worse than their 'low' mutation counterparts: higher fitness error, lower optimality and higher average individuals to optimality. These candidates should be avoided.

Additionally, those candidates with a population size of 150 showed considerably higher amounts of average individuals to optimality than those with a population size of 50: the larger population size lead to overall higher cost for the algorithm. This makes sense as a higher population directly leads to higher individual counts; for the candidates with a population size of 150 to have lower individual counts than their counterparts with population sizes of 50, the higher population candidates would have to have average generation counts at least 3x lower than the lower population candidates. Though these generation counts were indeed lower in some cases, they were also equal or greater in some cases and the costs ended up being considerably higher.

Because the candidates with higher population sizes had much higher costs, these candidates would also need to have considerably better performance (that is, higher optimality and lower fitness error) to be considered as a baseline. In general, this was not the case as there was little overall difference in performance between the lower population subset and the higher population subset.

It is also observed that candidates with a tournament size of eight had, in almost every case, better performance and lower cost than their counterparts with a tournament size of four. The only case where this did not hold was in the comparison between {50, 0.0, 4, 'low'} and {50, 0.0, 8, 'low'}. However, {50, 0.0, 4, 'low'} had comparable performance to {50, 0.0, 8, 'low'}, with an *extremely* low overall cost, which leads one to believe that it was a possible anomaly.

And finally, it can also be observed that the candidates with no elitism outperformed those with

0.07 elitism. In every comparison between two otherwise equivalent candidates, the one without elitism had higher optimality. And in almost every such comparison, the candidate without elitism had lower average error and lower average cost than that with elitism.

So it is seen that the parameter configurations which had superior overall performance are those with a population size of 50, elitism of 0.0, a tournament size of eight and 'low' mutation rates. Thus, the candidate {50, 0.0, 8, 'low'} can be considered the overall best configuration. This candidate did have the second lowest average error, the fourth highest optimality and the second lowest cost. Its cost was only surpassed by {50, 0.0, 4, 'low'} and its fitness error was only surpassed by {150, 0.0, 8, 'low'}. This places {50, 0.0, 8, 'low'} in a “sweet spot” of sorts. For these reasons, it is selected as the baseline algorithm.

The baseline algorithm was then compared against three alternative algorithms: random search, greedy algorithm and (1+5)-Evolution Strategy. The random search was global: at each generation, it picked a completely random partition to jump to and kept the more fit partition between the current partition and new, random partition. The greedy algorithm operated by selecting a random schedule to start a new subset, iteratively selecting the available schedule which would provide the best fit within the subset until the subset was up to the proper size, then constructing new subsets in this manner until the correct amount of subsets had been filled, thus completing the partitioning. The (1+5)-ES was performed by starting with a random partition and mutating five children at each generation: the best of these children was chosen as the parent for the next generation. This continued for the maximum amount of 250 generations, or until the optimal solution had been reached.

These three algorithms were tested with the same data sets as the GA baseline candidates. The results of these tests are shown below, along with the selected GA baseline, for comparison:

Algorithm	Optimality Ratio	Average Fitness Error	Average Generations to Optimality	Average Individuals to Optimality
Genetic Algorithm	83/100	1.6609	96.28	4814
Random Search	15/100	28.1690	218.79	218.79

Greedy Algorithm	39/100	17.5149	--	--
(1+5)-Evolution Strategy	7/100	48.5303	242.94	1214.7

Perhaps the most interesting result to be seen here is how staggeringly poor the Evolution Strategy performed. It consistently failed to find the optimal solution and often did so in spectacular fashion, as shown by its enormous average fitness error. It did even more poorly than random search, which found an optimal solution only 15 times out of 100, with a high average fitness error value. Though these two algorithms did have much lower costs than the genetic algorithm, it should be remembered that these values are held artificially low by the limit of 250 maximum generations per run. If this had been elevated, the fact that the algorithms showed a relative inability to converge upon optimality would result in quickly rising costs and little gain in performance.

The greedy algorithm performed modestly well, at least in comparison to the random search and the Evolution Strategy. It found the optimal solution on 39% of the test runs and had a fitness error considerably lower than the other two alternative competitors. A big factor in the greedy algorithm's favor is its considerably low cost: because it is not a generationally iterative algorithm and it simply constructs a partition solution schedule by schedule, its cost for each run is only one individual. It is considerably more light weight than the genetic algorithm. However, its performance still pales in comparison to the GA as the GA was able to converge upon optimality on over twice as many test runs. The average error was also much, much lower, which is a product of not only its higher optimality ratio but its ability to consistently find solutions which were close to optimal. The greedy algorithm did not have this ability.

However, the greedy algorithm did perform exceedingly well on test landscapes which featured schedules that were highly similar, a landscape which the genetic algorithm struggled with. Because of its procedure whereby it selects schedules one at time, it was able to quickly piece together subsets of extremely similar schedules. In short, the fact that the greedy algorithm operates on a more local,

subset-wide scale than the genetic algorithm, which operates on a more global, partition-wide scale, gave the edge with these types of tests to the greedy algorithm. It should be noted, however, that these landscapes on which the greedy algorithm outperformed the genetic algorithm are more “academic” in nature: they are artificially constructed and unlikely to be similar to those found in the “real world” where individuals' respective schedules vary widely. This, in some sense, negates the applicability of the greedy algorithm to this problem.

Conclusion

The fact remains that the genetic algorithm outperformed its competitors by a considerable margin. It found optimal solutions at a much higher rate and with significantly lower fitness error, though this did come at a higher cost at runtime. The algorithm was also able to converge on optimal or near-optimal solutions in the amount of time that it would take a human to even read an individual's schedule! It is simply another example of the ability of computation algorithms to solve challenging problems with great efficiency and precision. And it is an example of the ability of Genetic Algorithms to solve hard problems with ill-defined goals that more traditional algorithms struggle mightily to handle.

However, there is still more work to be done. This algorithm assumes that all input will have equally divided subsets: allowing for subsets of different sizes substantially changes how the problem should be represented and the evolutionary operators should be designed. There are other considerations that could be made, as well. For instance, features such as placing weights on different time slots, taking into account the closeness of available time slots or adding new constraints, such as effectively “tying” multiple individuals together or “separating” individuals, could provide new directions for research.

Additionally, the amount of parameter configurations which were tweaked for testing purposes

could be expanded significantly. Providing two values for each parameter limits the ranges from other potentially reasonable or desirable values. Opening up other parameters to configuration would be beneficial, as well. Using larger data sets in tests would be helpful, as well, as the size of the current tests is limited to 24 individuals or fewer. It is difficult to scale tests in this way as the tests must be created so that the optimal solution is known; as tests become arbitrarily large, this becomes arbitrarily more difficult. Perhaps a methodology for problem generation could be developed to help in this regard. In any event, more testing and more experimentation with the design of the GAs components would shed a great deal more light on this unique problem.

References

1. E. Hard, P. Ross, and D. Corne. “Evolutionary Scheduling: A Review”. *Genetic Programming and Evolvable Machines* 6 (1995): 191-220. Web.

2. Green, William A. “Partitioning Sets with Genetic Algorithms.” *American Association for Artificial Intelligence*. 2000. Web.

Appendix

I. Test Descriptions

Test	Schedule Count	Subset Count	Subset Size	Schedule Similarity
1	Small	Small	Medium	High
2	Medium	Medium	Small	Low
3	Large	Large	Small	High
4	Large	Medium	Medium	Medium
5	Medium	Large	Small	High
6	Medium	Small	Large	Medium
7	Medium	Low	Medium	Low
8	Large	Large	Medium	Low
9	Large	Medium	Large	Medium
10	Large	Small	Large	High

II. Baseline Genetic Algorithm Results by Test

Test	Optimality Ratio	Average Fitness Error	Average Generations to Optimality	Average Individuals to Optimality
1	10/10	0.0	1.0	50
2	10/10	0.0	13.8	690
3	5/10	12.4687	206.9	10345
4	8/10	0.4005	168.6	8430
5	10/10	0.0	33.8	1690
6	10/10	0.0	3.3	165
7	10/10	0.0	16.0	800
8	6/10	1.8602	169.8	8490
9	8/10	0.6545	182.4	9120
10	6/10	1.2247	167.2	8360

III. Project Information

This paper, a PowerPoint presentation and the source code for the project can all be found at https://github.com/jessex/CS499-Final-Project.