An Evolutionary Algorithm for Assigning Students to Courses

Christine Shannon and Drew McKinney
Centre College, 600 W. Walnut, Danville, KY 40422
christine.shannon@centre.edu

Abstract

In this paper we describe an evolutionary algorithm for assigning students to courses in a situation where each student specifies a set of courses in order of preference, each course has a limited enrollment, and the object is to maximize the overall student satisfaction by assigning each student to a course as high on his or her preference list as possible. Results of using the algorithm on historical data are compared to the success of a human in making the assignments. This work was done as part of a summer undergraduate research project while the second author was still a student. We also report preliminary results for using this problem as the basis for an assignment in a course in Artificial Intelligence.

Introduction

Like many liberal arts colleges, Centre College offers a short, intensive term between its more traditional long terms. During this time, first year students must enroll in specially designed First Year Seminars. The class size is limited to fifteen and courses are offered across a broad spectrum of disciplines on a wide variety of topics. During the registration process students are required to list four courses that they would like to take in their order of preference. The registrar attempts to satisfy these requests as equitably as possible with the hope of granting as many students as possible their first or second choice. Clearly, the problem does not always admit a solution. In the extreme case, all students could request exactly the same four courses with the result that only sixty of them could be accommodated under the stated rules. Fortunately, nothing close to this has occurred but in a year where nearly all classes had to be completely full, it was necessary to raise the limit on a few courses to insure that each student received one of the listed choices. Normally, the assignments are made by the registrar using a variety of heuristics which eventually produce good results. While not initially interested in an alternate solution, the registrar was willing to share historical data so that we could empirically compare the results of the assignments he made by hand with those produced by various evolutionary algorithms arising from our study. The recent growth in the size of the first year class has increased his interest in these results.

Problem Description

The problem at hand can be viewed as an instance of the Generalized Assignment Problem in which each agent in one set is matched to a single task in another set. Furthermore, each task has a limited capacity for agents and the goal is to maximize some objective function. Formally, the problem can be stated as an integer programming problem.
Maximize

$$S = \sum_{i=1}^{n} \sum_{j=1}^{m} s_{ij} d_{ij} \qquad (0.1)$$

given the constraints

$$\sum_{i=1}^{n} d_{ij} \le 1 \quad \text{for } j = 1 \ldots m \qquad (0.2)$$

and

$$\sum_{j=1}^{m} d_{ij} \le 15 \quad \text{for } i = 1 \ldots n \qquad (0.3)$$

where m is the number of students, n is the number of courses, s_{ij} is a measure of the preference of student j for course i and

$$d_{ij} = \begin{cases} 1 & \text{if student } j \text{ is assigned to class } i \\ 0 & \text{otherwise} \end{cases} \qquad (0.4)$$

The requirement (0.2) insures that no student will be assigned to more than one course. Similarly (0.3) addresses the need for class sizes to be capped at 15.

There are many ways in which to define the preferences s_{ij}. For example s_{ij} might equal 5 if class i is the first choice of student j and a correspondingly lower weight if it is a lower choice. Alternate values assigned to the weights will reflect a desire to reach certain goals. For example, one could significantly raise the weights for first and second choices and significantly penalize (with a negative weight) any placement which was not one of the student's original choices. In any event, attempting to maximize S will encourage assignments which match these preferences.

Previous Work

The Generalized Assignment Problem is a well studied domain and there are numerous ways to construct solutions to this problem including many which employ genetic algorithms. We mention only a few of many works dealing with this and related problems. Because this was an undergraduate research project, the existence of these solutions did not detract from the opportunity to develop, test and assess a potentially novel approach to the problem.

One interesting and related application of the Generalized Assignment Problem is found in a paper on the *Sailor Assignment Problem* (Garrett et al. 2005). Because the United States Navy requires that each sailor in the Navy change jobs every two years, there is a need to find a mapping between the sailors, all having their own particular skills and desires, and the available jobs with their particular requirements. This must be accomplished in a way that somehow maximizes the satisfaction of the sailors and their potential commanders while conforming to budgets and so on. In this case the constraints in (0.3) would have the right hand sides equal to one since each job can be taken by at most one sailor.

(Feltl and Raidl 2004) has background information and bibliography on the Generalized Assignment Problem. In this paper the authors describe several exact and heuristic approaches to this problem including a hybrid genetic algorithm from Chu and Beasley.

Exact methods such as integer programming are frequently based on some version of branch and bound. Often designers employ some sort of heuristic to guide the order in which they explore the solution space during the branching process. This sort of approach is not practical for the problem at hand because, for example, if there were 300 students and 21 courses, there would be 6300 variables and 321 constraints in addition to the requirement that the solutions be binary.

We turned our attention to developing a genetic algorithm not only because an exact solution seemed infeasible but also because the genetic approach offered the opportunity for experimentation and creativity in the context of an undergraduate research project and the opportunity to apply a classroom topic to an actual problem.

An Evolutionary Algorithm

Evolutionary algorithms work with a population of potential solutions, often called chromosomes. At each step of the process, members of the population are evaluated by a fitness function. Those chromosomes with the highest value of the fitness function are most likely to survive to the next generation and/or produce additional offspring by some methods which mimic the processes of natural selection and mutation in the natural world. This process continues for many generations until at some point the chromosome with the maximum fitness in the current population is selected as an approximation to the true maximum.

There are many techniques which have been used for the selection and mutation operators. This paper motivates and describes one such set of variation operators and the results of the experiments that compared the values of (0.1) for the registrar's assignment of students to that for various versions of an evolutionary algorithm.

Representation

Typically, a chromosome is represented by a string of zeros and ones. The crossover operation which produces the offspring is frequently accomplished by splicing together a copy of an initial segment of one string with a terminal segment of another. If, as is frequently the case, the length of the chromosomes is fixed at some value t then the crossover can occur by concatenating the first k bits from one chromosome with the last $t - k$ bits from the other for some k with $0 < k < t$ Mutations occur by randomly flipping a bit. The particular algorithm must determine how a solution can be mapped to such a string of bits, the protocol under which the operations occur, and an assortment of probabilities, counts, and frequencies.

Given that there are n different classes and m students, initially it seemed appropriate to represent a *chromosome* which is a potential solution to the problem as a vector of m* k bits where $k = ceiling\ (\log_2 (n + 1))$. The binary representation of the class to which the solution assigns student i would then be located in bits $k*(i-1)...k*i-1$. While this would make it easy to execute the usual crossover and mutation operations, it is obvious that these operations could regularly result in an infeasible solution. Even if we split the chromosomes at a multiple of k, there is no reason to believe the resulting chromosome would observe the constraint in (0.3). Eventually, we decided on an alternative. A chromosome would be represented by a permutation of the digits *1..m* and this permutation would represent the order in which the students would select a class. Each student would of course select the class highest on his or her priority list which still had not reached capacity. If it should happen that all four courses which the student had selected were already filled when it was that student's turn to select a course, then that particular student would remain unassigned.

The Operators

Unfortunately, the typical *crossover* operator which takes two chromosomes from the population and creates a new chromosome for the next generation by splicing together parts of each will still not work. Given the selected representation, such an operation would almost certainly create a sequence which is not a permutation of *1..m* and hence is not an admissible chromosome. Bearing in mind that the purpose of generating new chromosomes by crossover is to produce a better candidate solution from two pretty good ones, we opted to forsake the two parent analogy. Instead, the offspring come from a single parent by the process of randomly selecting two integers, *a* and *b* between *1* and *m* and randomly permuting the contents of the chromosome between index *a* and index *b.* This guarantees that the resulting chromosome will be a permutation of the numbers *1* to *m* and has the effect of changing the order in which a subset of the students make their selections. Just as in the case of the two parent operation, this operation does not guarantee that the offspring will have better fitness than the parent. It might be better to call this operation *reproduce.*

The *mutation* operator randomly selects two indices *i* and *j* between *1* and *m* and interchanges the contents of the chromosome at those two points. Again, this operator preserves the requirement that each chromosome must be a permutation of the numbers *1* to *m.* It has the effect of selecting two students and switching the order in which they select their classes. If originally the first student was the *ith* one to make a selection and the second was the *jth* one to do so, now the second student will be the *ith* one to make a choice and the first will be making a choice in the *jth* position.

Finally, there is an *improve* function. This is an effort to locally repair a chromosome which results in unassigned students. When student *i* is unassigned, the algorithm looks up the first choice c_1 of that student, randomly selects a student *j* who was assigned to class c_1 and then swaps the positions in the chromosome of students *i* and *j*. Since course c_1 must have been available at the time student *j* made a choice, this guarantees that student *i* will be assigned and at worst it may mean that student *j* is unassigned. More likely, student *j* will simply get a lower priority choice which would still improve the value of the fitness function described below for the new chromosome.

The Fitness Function

The fitness function must be selected in such a way as to generate a solution which meets the objectives of the problem. In this case, it was desirable to assign students to their highest choices while attempting to assign all students to some class and no student to a class which did not appear on his or her priority list. As pointed out earlier, it is possible that such a solution does not exist and the algorithm may terminate with some students unassigned. In this case human intervention would be required to perhaps increase the limit on a few classes to make certain that all students could be accommodated in one of their desired courses. This is much more likely to occur when there is very little slack in the total number of available course slots when compared to the number of students who must be assigned or when there are a small number of courses which are ranked highly by a large portion of students.

In view of these considerations, one fitness function that was used awarded four points for each assignment that gave a student his or her first choice, three points for a second choice, two points for a third and one point for a fourth choice. Ten points were subtracted for every unassigned student. This particular function did not differentiate greatly among any of the choices listed by a student, but placed a great penalty on unassigned students.

An alternate fitness function which placed greater value on students getting one of their first two choices could award 8 points for every first choice, 4 for a second, 2 for a third choice, 1 for a fourth choice and -10 for no placement. This should produce a much larger number of students receiving their first or second choice. This function is used in many of the experiments reported below since it matches the registrar's desire to give students their first or second choice.

Obviously, one can experiment with a wide collection of potential fitness functions and observe the results. Sample outcomes are given below.

Generating a Solution

A variety of experiments were performed with different population sizes. However, in each case, for a population of size *p*, the initial population was created by randomly generating *2*p* chromosomes, sorting by their fitness values, and then selecting the *p* chromosomes with the highest fitness. This starts the process with a population possessing above average fitness.

To get the subsequent population, *p* offspring are created from the current population by randomly selecting chromosomes (with replacement) and with a probability proportional to their fitness using roulette wheel selection (Goldberg 1989) which is described below. Chromosomes with a higher fitness are more likely to be chosen but even those with a very low fitness have a positive, though possibly very small, probability of being selected.

The reproduce operation is then applied to each selected chromosome. In order to insure that no good solutions are lost between generations, the current population is merged with its offspring, yielding $2*p$ chromosomes. Mutation is applied to a fraction of the population, followed by the *improve* function and then the p chromosomes with the highest fitness are selected for the next generation.

After a fixed number of generations the largest value of the fitness function for members of the population is determined. If there are multiple chromosomes in the population with that fitness, one with a minimum number of unassigned elements is selected at random.

Implementation

The algorithm was implemented in Python and the following remarks will point out a few of the ways that we employed language features to increase the efficiency of the operations. The population of chromosomes was maintained as a dictionary indexed by the fitness. If D is the dictionary and f is a fitness value, then $D[f]$ is the list of all chromosomes with fitness f. Since Python offers very good support for dictionary operations, this organization facilitates the process of producing the next generation by selecting chromosomes for creating offspring by their fitness and then keeping those in the union of the current population and their offspring with the greatest fitness.

Specifically, given the set of fitness values f_i for $i = 1..k$ and their frequencies $n_i = length(D[f_i\ ])$ construct the intervals: $[b_0, b_1], [b_1, b_2] \ldots [b_{k-1}, b_k]$ where $b_0 = 0$, $b_1 = n_1 f_1$ and for $t > 1$, $b_t = b_{t-1} + n_t f_t$. These intervals are proportional in length both to the fitness value and the frequency with which it appears. Select a random number r in the interval $[0, b_k]$, determine the subinterval $[b_{t-1}, b_t]$ containing r and then randomly select a chromosome from $D[f_t]$. This procedure will select a chromosome randomly but with a probability proportional to its fitness.

To select the new population from the union of the previous generation and its offspring, the new chromosomes are added to the dictionary for the previous generation and the fitness values of the dictionary are sorted in decreasing order. The next generation is created by the following process:

1. Let f^* be the greatest fitness in the dictionary
2. If adding all the elements in $D[f^*]$ to the next generation would not exceed the desired population size add all of them to the next generation. Otherwise, randomly select the desired number of chromosomes from $D[f^*]$ to add to the next generation and exit
3. While the number of elements in the next generation is less than the desired number let f^* be the next largest fitness value in the dictionary and go to step 2.

Maintaining lists of all students assigned to each course makes the *improve* operation efficient. When student j has a chance to make a selection and all four choices on his or her priority list are full, j is added to an *unassigned* list. The *improve* function then processes this list as described above. Similarly, when the specified number of generations has been computed selecting the final candidate chromosome merely requires identifying the largest key in the dictionary and processing the list of chromosomes with that fitness to select one with the smallest number of unassigned students. This is usually a very short list. Thus the ease and efficiency with which Python allows the programmer to make dictionaries and lists facilitates a speedier solution at the usual cost of extra space.

Experimentation and Results

This is actually an excellent problem for student experimentation. Even when all these choices are made, there are still a great number of parameters which must be decided such as the size of the population, the frequency of mutation, and of course the stopping condition which in this case is simply a fixed number of generations. Because we had six years of historical results we could also compare the results produced by the algorithm with those of the assignments the registrar made by hand.

For all the results shown below, the population size is set at 200, and one percent of the population experiences a mutation. The populations are processed through 1000 generations. We begin with a fitness function in which the number of first choices assigned is multiplied by 8, second choices by six, third choices by two and fourth choices by one. Ten points are subtracted for each student who is not assigned to a course. This is consistent with the desire to give as many first and second choices as possible.

The first table gives the results for three runs of the data for 2002. There were 291 students and 22 courses which could hold 330 students. There was quite a bit of extra room and hence it was easier to satisfy the requests. The column labeled “Registrar” gives the number of first place, second place, etc assignments that were made by the registrar when this was done by hand. The last three columns give the results for three runs of the evolutionary algorithm.

	Registrar	Run 1	Run 2	Run 3
First choice	209	220	221	213
Second choice	71	67	66	69
Third choice	2	3	4	3
Fourth choice	9	1	0	0
Unassigned	0	0	0	0

Table 1: Data for 2002

The algorithm performed very well in these cases. Because everything is done at random each run gives different results and there can frequently be very different results for individual students. However, the fact that we iterate through 1000 generations means that the fitness of the solutions will be fairly close. Similar results can be seen for the following year when there are 286 students and 23 courses.

	Registrar	Run 1	Run 2	Run 3
First choice	182	188	190	190
Second choice	86	81	79	79
Third choice	18	14	14	14
Fourth choice	0	3	3	3
Unassigned	0	0	0	0

Table 2: Data for 2003

Again, the algorithm performs very well when compared to the assignments made by the registrar. Note that the last two runs of the algorithm given identical distributions as far as the number of first place choices, second place choices, etc. However, if you examine the particular assignment of student to course there are many differences because of the random selection process.

Where we would expect the evolutionary algorithm to have more difficulty is the situation where the number of available places and the number of students is very close. Of course this also poses problems for hand calculation. The data for the year 2004 gives us an opportunity to test this. In fact, there were only 18 courses offered that year and there were 274 students – four more than could be accommodated with only 15 per course. Furthermore, there were several courses which were very highly ranked by a large fraction of the students. In order to satisfy all the requests of the students, the registrar had to increase the limit on six courses to 16 and on one particularly attractive class to 17.

The algorithm was run with the usual limits but then human intervention was used to individually place the students who were not assigned into one of their choices by raising the limits. Table 3 shows the data for the registrar's placement and the results of two runs of the algorithm. The data for the algorithm indicate the original results as well as the improved results after human intervention placed the unassigned students as highly as possible without violating the limits as extended by the registrar.

	Reg	Run 1		Run 2	
		Orig	Imp	Orig	Imp
First choice	148	156	158	158	161
Second choice	86	73	77	71	73
Third choice	37	24	25	25	26
Fourth choice	3	14	14	16	16
Unassigned	0	7	0	6	0

Table 3: Data for 2004

Again, the results are quite satisfactory except that perhaps there are more students that end up with their fourth choice when using the evolutionary algorithm. Observe, however, that the totals of third and fourth choice placements are pretty close. This could be influenced by changing the weights in the fitness function as described elsewhere.

As a final experiment, let us change the weights in the fitness function so they no longer put so much emphasis on getting a first or second choice. This time we will give weights of four, three, two, and one for first, second, third and fourth choices respectively and subtract ten for every unassigned student to minimize the human intervention.

For this we used the data from 2005 where there were 294 students and 20 courses. There was not much extra space but there was a little. Under these circumstances, the algorithm could terminate with unassigned students but this only occurred in one of the three runs. Run 1 was done with the usual weights that we have been using and runs 2 and 3 with the new weights

	Reg	Run 1		Run 2	Run 3
		Orig	Imp		
First choice	159	187	188	186	191
Second choice	115	78	78	70	65
Third choice	20	17	17	25	24
Fourth choice	0	11	11	13	14
Unassigned	0	1	0	0	0

Table 4: Data for 2005

This change in weights caused an approximately 30 per cent increase in the number of third and fourth choices and illustrates that one can achieve a variety of goals by manipulating the weights appropriately. In this particular year the registrar seemed to want to avoid fourth place choices even though it meant fewer first place choices. One could come closer to that goal by adjusting the fitness function to place much greater distance between the weights on the first three choices and that on the fourth.

An AI Assignment

In the past when the first author discussed genetic algorithms in an introductory Artificial Intelligence class, it was often in the context of a textbook example such as maximizing the value of a function. The success of this project provided an alternative in a problem domain with which all the students were very familiar since they had once selected their own list of preferences for a First Year Seminar. Of course, in a two week assignment, students could not be expected to achieve the same kind of results as in a summer project. Nor could they be expected to read research articles or consider as many alternatives. On the other hand, they could be expected to define a reasonable encoding, fitness function, and operations. To facilitate things, they were provided with some utility functions and the dictionary framework for their use. Almost all the teams had some initial difficulty in selecting a crossover operation with a reasonable chance of producing improved offspring. Some began with random chromosomes without considering how few of them would be feasible. Having students submit a preliminary description at the halfway point was very beneficial.

In the end, four of the five teams submitted projects on time. Two teams had a crossover operation involving two chromosomes and two had used a single chromosome to create a single offspring. The programs were tested with a new set of data which the students had not seen in the development phase. Not surprisingly, there were some difficulties. When the registrar had visited the class in the role of a client, he indicated that in addition to satisfying student requests, he was also interested in balancing class sizes between 11 and 15. One team wrote a fitness function which was heavily biased toward enforcing this balance to the detriment of satisfying student course preferences. Another weighed this so lightly that while over 88 per cent of the students obtained one of their top two choices, classes were somewhat unbalanced. The other two teams opted for enforcing at least the upper bounds and produced assignments with about 75% of students in their top two choices, 95% in one of their four choices and the rest unassigned, leaving the registrar to make the final placements. This particular issue needs additional discussion and experimentation.

Conclusions

The evolutionary algorithm which we developed in our summer research compared very favorably with the assignments of students to courses as done by the registrar. Furthermore, it was easy and fast to use, generally producing an assignment of all or nearly all students to courses in less than eight minutes. When there were a handful of unassigned students it was not very difficult to assign them by hand by raising the limits on a few courses. An additional benefit was that all placements were done at random, freeing the registrar from any complaints about unfairness or partiality.

Given the time constraints, the corresponding assignment in the Artificial Intelligence class was reasonably successful in that four of the five teams produced working results and had the opportunity to encounter a number of issues which they would not have confronted otherwise. While there may be better ways to solve this problem, the first author found it a good vehicle to engage the students and several continued to work on it after it was graded and discussed. Clearly, the fitness function will get greater attention next time. This was a good problem for both the undergraduate research project and the class assignment.

Acknowledgements

Although the design and development of the algorithm was collaborative, almost all of the code used in this project was written by the second author while he was an undergraduate at Centre College. He graduated in 2009. In addition he developed a good user interface and manual and provided the kind of documentation in the code which would facilitate its use by a non-computer scientist. He was supported by enabling funds of the Margaret V. Haggin endowed professorship held by the first author.

References

Feltl, H. and Raidl, G. 2004. An Improved Hybrid Genetic Algorithm for the Generalized Assignment Problem. In *Proceedings of the 2004 ACM symposium on Applied Computing* (Nicosia, Cyprus, March 14-17, 2004). SAC ’04. ACM, New York, NY, 990-995.

Garrett, D., Vannucci, J., Silva, R., Dasgupta, D., and Simien, J. 2005. Genetic Algorithms for the Sailor Assignment Problem. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation.* (Washington, D.C., June 25-29, 2005), GECCO ’05. ACM, New York, NY, 1921-1928.

Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning.* Los Altos. CA: Morgan Kaufmann.

Kellerer, H., Pferschy, U. and Pisinger, D. 2004. *Knapsack Problems.* Berlin: Springer.

Michalewicz, Z. and Fogel, D. 2002. *How to Solve It: Modern Heuristics.* Berlin: Springer.