

The Simulated Annealing Group Assignment (SAGA) Application

Jaymie Strecker
College of Wooster
Wooster, Ohio 44691
jstrecker@wooster.edu

Dr. Jon Breitenbucher^{*}
College of Wooster
Wooster, Ohio 44691
jbreitenbuch@wooster.edu

Dr. Denise Byrnes[†]
College of Wooster
Wooster, OH 44691
dbyrnes@wooster.edu

ABSTRACT

The task of assigning students to Critical Inquiry Course (CIC) sections, taking into account the preferences and demographics of the students and the sizes of the sections, is a formidable task by hand and is not trivial to automate. The Simulated Annealing Group Assignment (SAGA) application employs simulated annealing, a stochastic approximation algorithm, to search for an optimal CIC section assignment within a reasonable amount of time. SAGA may be applied not only to the problem of assigning students to CIC sections, but also to a much more general range of group assignment problems.

1. INTRODUCTION

Each July, a certain boardroom at the College of Wooster became the setting for a repetitive task of overwhelming proportions: assigning first-year students to their required Critical Inquiry Course (CIC). Hundreds of papers, on which incoming first-year students had listed (in no particular order) the CIC sections in which they would prefer to be placed, were strewn across the table. A cabinet filled with files of demographic information for all of the first-year students was lugged into the room. A few college employees would spend the next two weeks regrouping the preference papers on the table into different CIC sections, consulting the demographic files, and regrouping again. Their quixotic mission is to place as many students as possible into CIC sections of the students' choice, while producing CIC sections that are both diverse (in gender, race, SAT/ACT scores, and other attributes) and are all close to the same size.

Assignment of CIC sections by hand not only monopolized the time of the College's staff, but also resulted in suboptimal section assignments. The College staff members were able to eliminate any blatant problems with diversity and

size of sections, such as a section composed entirely of one gender. However, any sort of in-depth analysis of the sections' size and diversity was too computationally intense to perform manually. To resolve this situation, SAGA was created.

In the pages that follow, the author will discuss the combinatorial optimization problem outlined above in terms of "students" and "sections," since CIC section assignment is the specific task that the Simulated Annealing Group Assignment (SAGA) application is designed to tackle. However, SAGA may be applied to a much broader array of problems.

Suppose that each student is represented by a vector of attributes, and let S be the set of all of these vectors. If c is the number of subsets of S (i.e. sections) required, then $C_i \subseteq S, 1 \leq i \leq c$, represent the sections. SAGA is able to solve group assignment problems in which the following conditions must be satisfied:

1. $\bigcup_{i=1}^c C_i = S$
2. $C_i \cap C_j = \emptyset, \forall i \neq j$
3. $\forall s \in S, s \in C_i$ only if $i \in p_s$, where p_s is the set of indices of all subsets of which s may be a member.
4. $\vec{w} \cdot \sum_{i=1}^c eval(C_i)$ is minimized, where \vec{w} is a vector of weights. $eval(C_i)$, the evaluation function, yields a vector of values for C_i based on the cardinality of C_i and the vectors of attributes that are members of C_i ; this function is explained in more detail later.

2. BODY

2.1 The Choice of Algorithm

The time-consuming process of assigning students to CIC sections cries out to be automated, but creating an algorithm to do so is not straightforward. With hundreds of students, and dozens of CIC sections to which they may be assigned, it would be computationally intractable to try every possible combination of students in sections. Just considering all CIC section assignments in which every section contains as close to the same number of students as possible, the number of different assignments is

$$\binom{s}{r} \binom{s-r}{k_1, k_2, \dots, k_c} \binom{c}{r} r!$$

^{*}Adviser

[†]Adviser

In the above expression, s is the number of students, c is the number of sections, r is the remainder of s/c , and $k_i = \frac{s-r}{c}$ for $1 \leq i \leq c$. This expression comes about by pulling out r of the s students, then choosing, from the remaining $s - r$ students, $\frac{s-r}{c}$ students to fill each of the c sections. Finally, to account for the r students that have not yet been assigned, r of the c sections are chosen to add an additional student to, and this can be ordered in $r!$ ways. If sections may be any size (including empty), the number of possible assignments increases to c^s .

It would also be extremely difficult and computationally intractable to use some other, more deliberate deterministic algorithm to produce the best CIC section assignment, since there are so many unrelated attributes to consider for each student. Moving a student from one section to another in order to improve those two sections' diversity in one attribute is likely to make some other attributes' diversities worse.

Fortunately, approximation algorithms offer an alternative approach to the methods described above. The Simulated Annealing Group Assignment (SAGA) application employs an approximation technique known as simulated annealing. Essentially, SAGA stochastically tries many different CIC section assignments (though not nearly so many as the brute force method) and, guided by a heuristic, gradually arrives at better and better assignments.

The problem of assigning students to CIC sections lends itself to a nondeterministic algorithm, since a large number of acceptable solutions exist. Also, an approximation algorithm well suits the problem, since a perfect CIC section assignment is not a requirement and runtime is a major concern. The simulated annealing method is chosen in part because it has been successfully applied to many difficult (including NP-complete) optimization problems [2]. Among these problems are the Traveling Salesperson problem, image reconstruction from noisy data, integrated circuit layout, and robotic path planning [2].

Genetic algorithms (also called evolutionary programming), another popular approximation technique, were considered as a possible approach to CIC section assignment but were ultimately rejected. In a genetic algorithm, several elements of the solution space are looked at simultaneously; these elements are analogous to individuals which make up a biological population. Like individuals in a biological population, the elements of the solution space undergo evolution, which occurs through reproduction and survival of the fittest. In order for a genetic algorithm to be applied to a problem, elements of the solution space must be encoded in such a way that two elements can reproduce by swapping some portion of themselves with their partner, just as biological reproduction involves the swapping of bits of DNA. The problem with encoding CIC section assignments in this way is that it is not obvious how two section assignments, by swapping groups of students or sections with each other, could produce a new, valid section assignment. The requirements that students be placed in sections of their preference and that each student be placed in exactly one section mean that arbitrary swaps of portions of section assignments are unlikely to result in valid assignments, making genetic algorithms an unreasonable approach to this problem [6].

2.2 About Simulated Annealing

Simulated annealing algorithms are one type of iterative improvement algorithm. Iterative improvement algorithms can be used to search a solution space methodically for an optimal solution. The idea of iterative improvement algorithms is to begin with a potential solution (in this case, a CIC section assignment) and to make adjustments to the solution over many iterations, moving the solution gradually toward a global optimum [6].

Russell and Norvig [6] describe the process of iterative improvement as "trying to find the top of Mount Everest in thick fog while suffering from amnesia." As this description suggests, an iterative improvement program has little or no knowledge about the solution space as a whole, and very limited knowledge about particular points in the solution space. The program's only sense of direction lies in determining whether a sampled solution is more or less optimal than the best solution found so far. With such a restricted view of the solution space, iterative improvement algorithms must be able to avoid becoming stuck in locally optimal solutions that are not globally optimal [6].

Simulated annealing imitates the physical process of annealing, which involves heating a solid until it melts, then allowing it to solidify into a crystalline structure. In annealing, the trick is to control the rate of cooling "in order not to get trapped in locally optimal lattice structures with crystal imperfections." The goal of annealing is to arrive at a globally optimal configuration of molecules, the solid's "ground state" [1].

Similarly, the goal of simulated annealing is to reach a globally optimal solution without getting stuck in local optima. Gallant [3] likens the process of simulated annealing to a strategy that one might utilize to position a marble in the deepest well inside a closed box containing wells of various depths. One might shake the box a number of times, with decreasing force each time. At some point, the energy of the shaking would have decreased enough to allow the marble to fall into the deepest well but not to be shaken out of it [3].

A more technical explanation of simulated annealing is as follows: The algorithm iterates over a temperature variable, which decreases at a rate defined by the cooling schedule. With each iteration, a potential new solution is created by making random changes to the current best solution [1].

The current and potential new solutions are scored by an evaluation function, which calculates how close each of the solutions is to a globally optimal solution. If the evaluation function yields a better result for the potential new solution than for the current best solution, then the current solution is replaced by the new one. If the result of the evaluation function for the potential new solution is worse, then the new solution may still replace the current one; the probability of this happening is proportional to the temperature [1]. In this respect, the temperature corresponds to the amount of shaking in the marble example [3].

Aarts and Korst [1] state that "under certain conditions the simulated annealing algorithm converges asymptotically to

the set of optimal solutions, i.e.

$$\lim_{k \rightarrow \infty} P\{X(k) \in S_{opt}\} = 1,$$

where k denotes the number of potential solutions tried. The simulated annealing algorithm is guaranteed to converge in finite time if the cooling schedule is sufficiently slow, following the conditions specified by Li [4]. Even the finite cooling schedule takes too much time to be feasible, so most actual simulated annealing applications sacrifice convergence for a reasonably fast cooling schedule [4].

2.3 The Simulated Annealing Algorithm

The pseudocode in Figure 1, drawn from Russell and Norvig [6], illustrates the simulated annealing algorithm. A slight difference between this pseudocode and the algorithm used in SAGA is that the pseudocode attempts to maximize the value of the evaluation function, while SAGA attempts to minimize it.

The algorithm begins by creating *current*, an element of the solution space, from the information given in *problem*, which in the case of SAGA is the set of students and sections. The algorithm then iterates through a loop, decrementing T , the temperature, at each pass. During each iteration, a new solution, *next*, is generated by making random changes to *current*. If the evaluation function score for *next* is better than the score for *current*, then *next* replaces *current*. If the evaluation score for *next* is worse, there is still some chance that *next* replaces *current*. The probability of this happening is given by $e^{\Delta E/T}$, which decreases as T decreases and as $|\Delta E|$, the difference between the evaluation function results of *next* and *current*, increases (since ΔE is negative). The algorithm terminates when $T = 0$.

2.4 The Cooling Schedule

For simulated annealing, any of a wide variety of cooling schedules may be used [5]. In SAGA, a linear cooling schedule is chosen for simplicity; it may be defined by

$$T_{i+1} = T_i - COOLING_RATE$$

The more iterations the simulated annealing algorithm runs, the closer on average it approaches an optimal solution (up to convergence). Figures 4 and 5 illustrate that the evaluation function result decreases on average after each iteration, and that the evaluation function result after the final iteration is typically lower the more total iterations have been performed, or in other words, the more slowly the simulated annealing has been “cooled.”

The constant `COOLING_RATE`, which controls the number of iterations, is selected by considering the tradeoff between evaluation function score and runtime. A value of `COOLING_RATE` that produces 40,000 iterations is chosen because the marginal benefit of slowing the cooling rate any further is relatively low. Figures 6 and 7 demonstrate this point, as the final evaluation function result barely improves, if at all, after 40,000 iterations.

2.5 Creating New Assignments

To generate a new CIC section assignment from the current best assignment, each of a number of students is moved ran-

domly from one section to another. The number of students moved is a small proportion of the total number of students, since smaller proportions of students moved both induce a more successful evaluation function result and decrease the runtime of the program. Figures 6 and 7 show that the runs of the program with the smallest percentages of students moved result in the best evaluation function results, regardless of the total number of iterations.

Although students are moved randomly, they may only be moved into a section that they indicated as one of their preferences. Students who failed to articulate any preferences are wildcards; they may be placed in any section. Ensuring that all students receive a CIC placement of their choice was not a requirement of this program. However, in sample data from a CIC section assignment that had been done by hand, student preference seemed to be the paramount concern, as nearly all of the students had been assigned to a section that they had requested.

2.6 The Evaluation Function

In SAGA, the evaluation function measures how distant each section is from the optimal CIC section. The population of students in the optimal CIC section is statistically identical to the student population as a whole, as measured by some selected statistical functions. The distance between a given section and the optimal section may be thought of as the number of students in the given section who are out of place — in other words, the number of students who would have to be replaced in order to reach the optimal section.

The pseudocode in Figure 3 illustrates the evaluation function algorithm used in SAGA to score each section. The evaluation function is composed of three parts, one for each different type of attribute: the textual attributes, the numerical attributes, and the section size attribute. The evaluation function score for a CIC section assignment is simply the sum of the evaluation function results for all of the sections.

For textual data (such as gender or race), the number of out-of-place students in a section is calculated for each attribute by comparing the proportion of students in each possible value of the attribute (such as female and male) to the proportion found in the student data as a whole. For numerical data (such as SAT or ACT scores), the statistical measure used to compare individual sections to the optimal section is the median. The number of out-of-place students for numerical data is the number of students who would have to be added (or removed) from the given section in order for the optimal section’s median to fall in the middle of the data for the given section. In addition to textual and numerical scores, the evaluation function calculates a size score for each section, which is simply the difference between the number of students in the section and the average section size.

When comparing a particular section to the optimal CIC section, it is likely that not all attributes will be equally important to the user of the program. *Ceteris paribus*, the user may favor a section with a higher diversity of academic interests over one with an optimal median GPA. For this reason, SAGA allows the user to weight each attribute according to its importance. In the evaluation function, each

attribute’s score is multiplied by the weight specified by the user, so that, for example, two out-of-place students in the GPA attribute may have the same effect on the evaluation function result as just one out-of-place student in the academic interest attribute.

For a simple example of how the evaluation function is used, consider the section shown in Table 1.

Table 1: A sample section.

Student ID	Country	SAT
0	US	1050
1	Other	1200
2	US	1300
3	Other	1400

Suppose that, in the optimal section, 60% of students reside in the US, the median SAT score is 1100, and the section size is 5 students. The textual score, which is just the score for country, is the difference between the number of US students in this section (2) and the number of US students in an optimal section of this size (2.4), plus the difference between the number of non-US students in this section (2) and the number of non-US students in an optimal section of this size (1.6); this is equal to $(2.4 - 2) + (2 - 1.6) = 1$. For the numerical score, which is just the score for SAT, the SAT value closest to the optimal section’s median SAT value is 1050, and its index in a sorted list of this section’s SAT values is 0. The index of the median SAT score in the same sorted list is between 1 and 2; SAGA’s evaluation function rounds this down to 1, so the numerical score is $1 - 0 = 1$. The score for size is simply the difference between the number of students in this section and in the optimal section, which is equal to $5 - 4 = 1$. If the weights for textual and numerical attributes are both 1, and the weight for section size is 2, then the total evaluation function score for this section is $1 * 1 + 1 * 1 + 2 * 1 = 4$.

2.7 Runtime Analysis

During the simulated annealing process, the three sections of code in which the program spends most of its time are in the generation of new section assignments, the evaluation function, and the storing or discarding of each new section assignment. In runtime analysis, it is storing or discarding section assignments that has the worst execution time of the three.

When SAGA decides, based on the most recent evaluation function result, to accept or reject a new section assignment, the assignment that is kept must be copied into the space that is holding the discarded assignment. Pseudocode for this process is given in Figure 2.

With a constant number of iterations during the simulated annealing process, the runtime of SAGA is equivalent to the runtime of the storing or discarding of new section assignments. This is given by

$$O(c * a_t * v + s * a_n),$$

where c is the number of sections, s is the number of students, a_t and a_n are the number of textual and numerical

attributes, respectively, and v is the maximum number of values that any textual attribute assumes.

2.8 Generality

Among the most noteworthy features of SAGA is its generality. SAGA may be used in a wide variety of group assignment situations, not just by the single institution and under the specific conditions for which it was created.

Because the College’s specifications for a CIC section assignment application do not require generality, early versions of SAGA are inflexible in the number and type of attributes that may be input into the program. The set of attributes that these versions use corresponds exactly to the database columns that the College has used in recent years to store student data. This set of hard-coded attributes presents an obvious problem: if the College decided to change the set of student attributes, then the source code for SAGA would have to be modified. Furthermore, these versions of SAGA could not be used by other institutions with similar sets of attributes, or by a wider range of users seeking solutions to group assignment problems. The inflexibility of the attributes in these versions needlessly limits the program’s applicability to a small subset of the group assignment problems that it has the potential to solve.

The current version of SAGA has the ability to read input files that contain an arbitrary number of attributes. At runtime, the user may specify the type of each attribute (textual, numerical, preferences, and other types). The type of each attribute determines how SAGA uses the attribute in the evaluation function and in generation of new section assignments. Textual and numerical attributes are scored in different ways by the evaluation function, while the preferences attribute determines which sections the students may be moved into during generation of a new section assignment.

2.9 Difficulties

The primary challenge that had to be overcome in creating SAGA was to quantify and objectify the concept of an optimal CIC section. When placing students manually it was impractical to adhere to any strict quantitative process because of the amount of data involved in assigning CIC sections. As a result, the algorithm for the evaluation function had to be invented from scratch by the group of programmers who originated SAGA.

One of the major concerns of this program is speed, since the simulated annealing algorithm iterates thousands of times. Efficient data representation helps to ameliorate the problem of speed. When the program is moving students into different sections, the students are represented as integer indices to an array rather than as (pointers to) objects. In addition, each section stores running totals of its students’ data so that these totals do not need to be recalculated each time the evaluation function is called.

When this program was first created, its intended use was for assigning students with an unchanging set of attributes to CIC sections. As SAGA was generalized, conflicts arose between making the program general enough to apply to a wide range of group assignment problems, yet keeping it

specific enough to continue its use in the particular college for which it was designed. For instance, since the College uses abbreviations for the students' countries of origin, pre-generalized versions of SAGA translate these abbreviations into regions of the world, which are used in the calculation of the evaluation function. However, it would not have made sense to include that feature in the generalized SAGA, since most potential users would probably not include that particular attribute with those particular abbreviations. In order to customize the program for the College, extra scripts will be provided to transform the student data into a practical form for inputting into SAGA.

3. CONCLUSIONS

SAGA was used successfully to assign the CIC sections at the College of Wooster for the Fall 2003 semester. Even since then, SAGA has been significantly improved; in particular, it has been made more generalized and more effective. The simulated annealing algorithm has proved to be an appropriate approach to the problem of assigning CIC sections. Areas that lay open for future research include the effectiveness of SAGA with different student datasets and with other group assignment problems.

4. ACKNOWLEDGMENTS

The other members of the team that created the original version of SAGA were Adam Anthony and Daniel Pawlowski. The author would like to thank Dr. Amon Seagull, for whose course the original version was created, and who offered assistance to the author at several stages of the project. The author would also like to thank those at the office of the Dean of Faculty at the College of Wooster for their willingness to take a risk by trying a new method of assigning CIC sections.

5. REFERENCES

- [1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, Chichester, 1989.
- [2] B. J. Buckham and C. Lambert. Simulated annealing applications. Retrieved September 2003 from the World Wide Web: http://www.me.uvic.ca/~zdong/courses/mech620/SA_App.PDF.
- [3] S. I. Gallant. *Neural Network Learning and Expert Systems*. The MIT Press, Cambridge, Massachusetts, 1993.
- [4] S. Z. Li. Markov random field modeling in computer vision. Retrieved September 2003 from the World Wide Web: <http://www.vision.ee.ethz.ch/~rpaget/Markov/book.html>, 1995.
- [5] B. T. Luke. Simulated annealing. Retrieved September 2003 from the World Wide Web: <http://fconyx.ncifcrf.gov/~lukeb/simann1.html>.
- [6] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Saddle River, New Jersey, 1995.

```

function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs: problem, a problem
         schedule, a mapping from time to “temperature”
static: current, a node
         next, a node
         T, a “temperature” controlling the probability of downward steps

current ← MAKE-NODE(INITIAL-STATE[problem])
for t ← 1 to ∞ do
    T ← schedule[t]
    if T=0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE[next] – VALUE[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability  $e^{\Delta E/T}$ 

```

Figure 1: The simulated annealing algorithm.

```

for each section section
    for each textual attribute attrib
        for each possible value val of attrib
            copy val
    for each numerical attribute attrib
        for each student student in section
            copy the value that student has for attrib
    for each student student in section
        copy student

```

Figure 2: The algorithm for storing or discarding section assignments.

```

inputs :  $section_{actual}$ , the section that is being scored
         $section_{optimal}$ , the optimal section
         $attributes$ , a vector of textual, numerical, and size attributes
         $weights$ , a vector of weights for the attributes

evaluationFunction( $section_{actual}$ ,  $section_{optimal}$ ,  $attributes$ ,  $weights$ )
/* textual attributes */
scoretextual ← 0
for each textual attribute  $attrib$ 
    scoreattrib ← 0
    for each possible value  $val$  of  $attrib$ 
         $num_{actual}$  ← number of students in  $section_{actual}$  with  $attrib = val$ 
         $num_{optimal}$  ← number of students in  $section_{optimal}$  *
            proportion of students in  $section_{optimal}$  with  $attrib = val$ 
        scoreattrib ← scoreattrib +  $|num_{actual} - num_{optimal}|$ 
    scoreattrib ← scoreattrib *  $weights_{attrib}$ 
    scoretextual ← scoretextual + scoreattrib
/* numerical attributes */
scorenumerical ← 0
for each numerical attribute  $attrib$ 
     $vals$  ← sorted list of the values of  $attrib$  for the students in  $section_{actual}$ 
     $median$  ← median of the values for  $attrib$  in  $section_{optimal}$ 
     $medianindex_{optimal}$  ← index in  $vals$  of the closest value to  $median$ 
     $medianindex_{actual}$  ← index in  $vals$  of the median of  $vals$ 
    scoreattrib ←  $|medianindex_{actual} - medianindex_{optimal}|$ 
    scoreattrib ← scoreattrib *  $weights_{attrib}$ 
    scorenumerical ← scorenumerical + scoreattrib
/* section size */
sizeactual ← number of students in  $section_{actual}$ 
sizeoptimal ← number of students in  $section_{optimal}$ 
scoresize ← scoresize +  $|size_{actual} - size_{optimal}|$ 
scoresize ← scoresize *  $weights_{size}$ 

return scoretextual + scorenumerical + scoresize

```

Figure 3: The evaluation function algorithm.

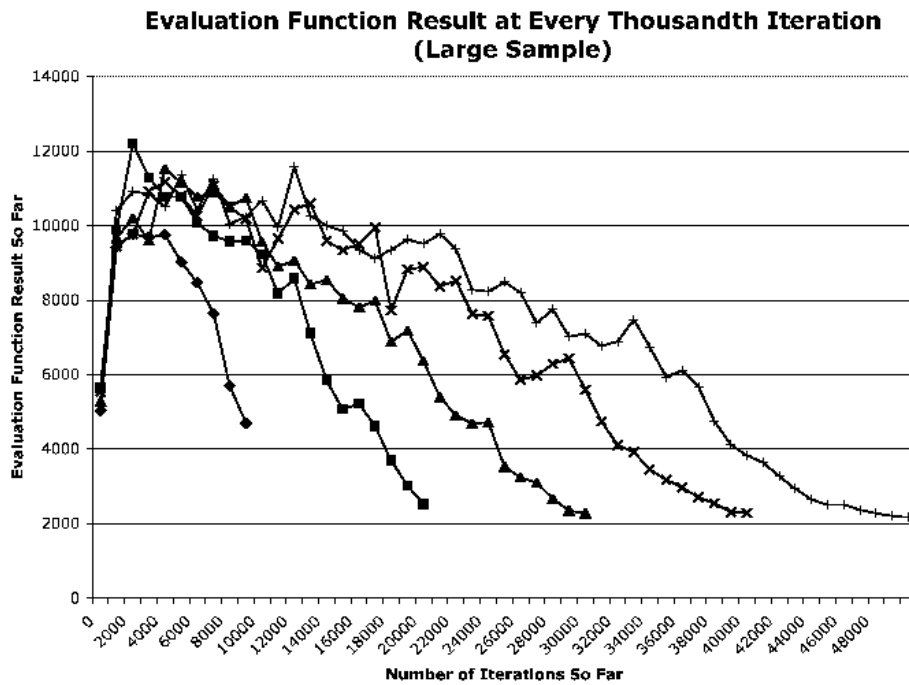


Figure 4: The evaluation function result at every thousandth iteration of simulated annealing is shown for test runs of 10,000, 20,000, 30,000, 40,000, and 50,000 total iterations. On average, the evaluation function result decreases after each iteration throughout the simulated annealing process. The final evaluation function result is lower the more total iterations have been run. A data sample with 528 students, 36 sections, and 17 attributes is used.

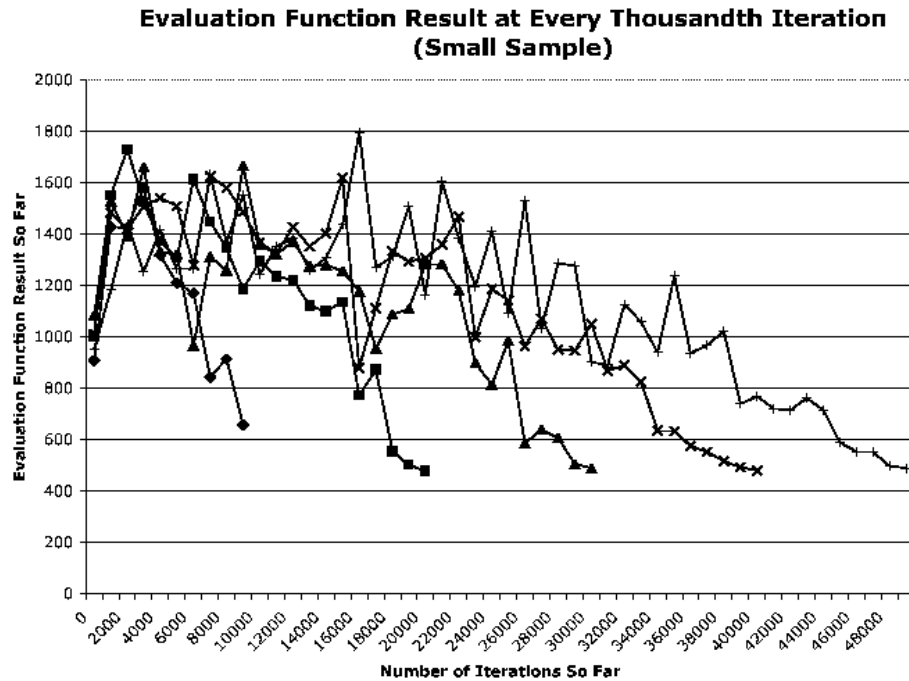


Figure 5: The evaluation function result at every thousandth iteration of simulated annealing is shown for test runs of 10,000, 20,000, 30,000, 40,000, and 50,000 total iterations. On average, the evaluation function result decreases after each iteration throughout the simulated annealing process. The final evaluation function result is lower the more total iterations have been run. A data sample with 100 students, 7 sections, and 8 attributes is used.

Final Evaluation Function Result (Large Sample)

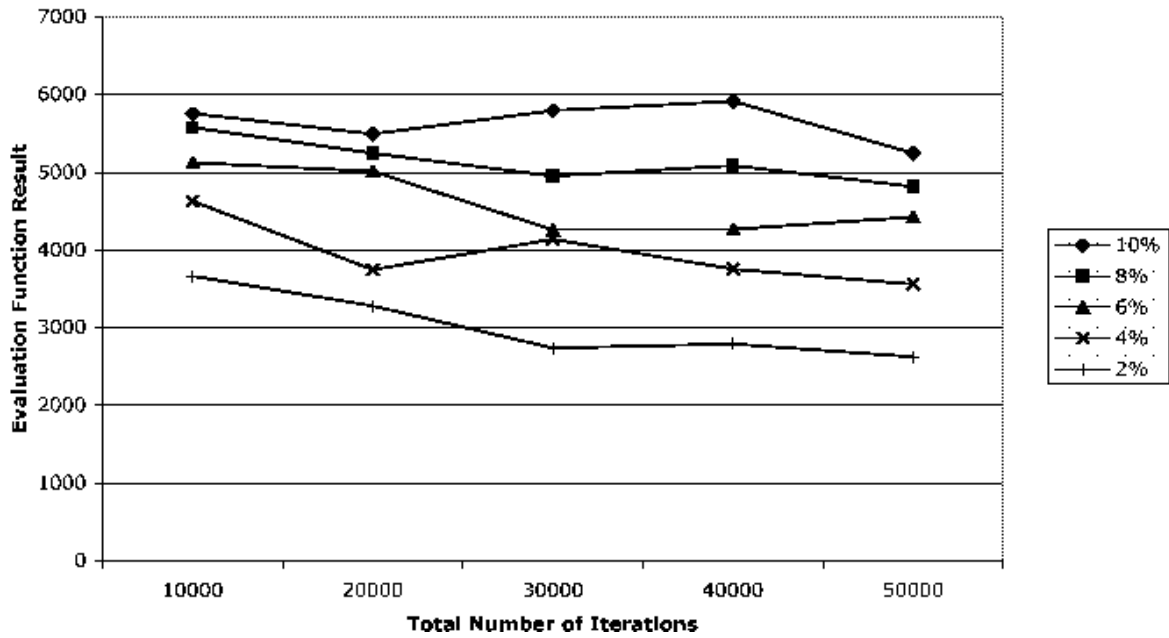


Figure 6: The final evaluation function result depends on both the total number of iterations of simulated annealing run and the percentage of students moved during each iteration when a new section assignment is created. With smaller percentages of students moved, the evaluation function result is lower. As the total number of iterations run increases, the final evaluation function result decreases. A data sample with 528 students, 36 sections, and 17 attributes is used.

Final Evaluation Function Result (Small Sample)

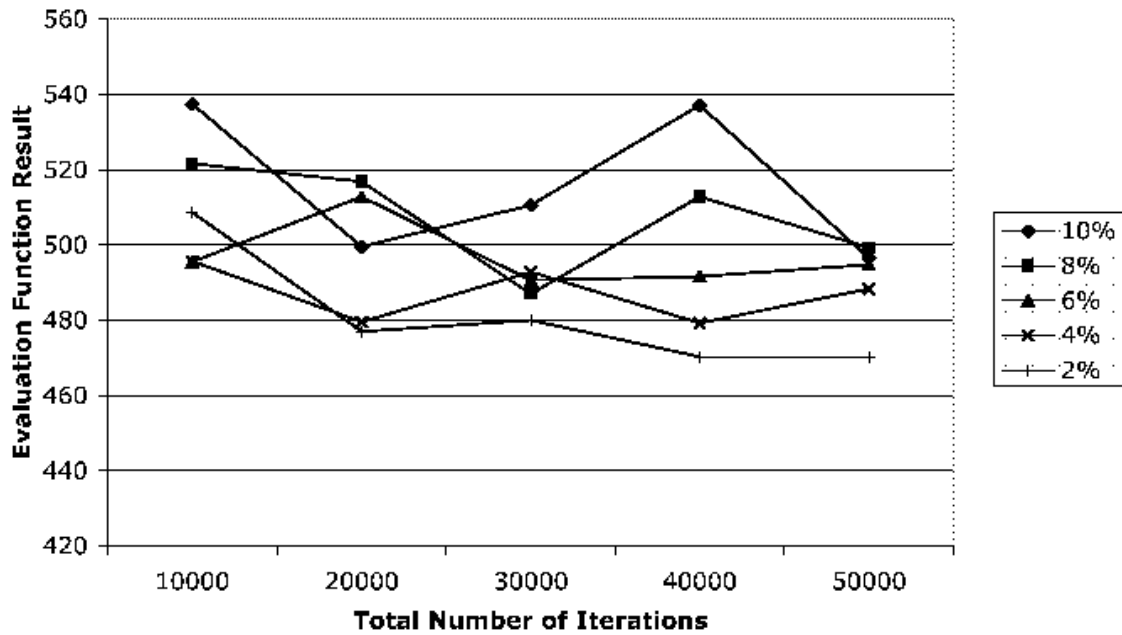


Figure 7: The final evaluation function result depends on both the total number of iterations of simulated annealing run and the percentage of students moved during each iteration when a new section assignment is created. With smaller percentages of students moved, the evaluation function result is lower. As the total number of iterations run increases, the final evaluation function result decreases. A data sample with 100 students, 7 sections, and 8 attributes is used.