

**The Student-Course Assignment Problem:  
Simulated Annealing for Bipartite Graph  
Optimization and Comparative Analysis**

by

**Ryan Kulyassa**

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Bachelor in Arts

With Specialized Honors in Mathematics

May 2025

## Abstract

The Student-Course Assignment Problem (SCAP) models the challenge of assigning students to courses based on ranked preferences while satisfying real-world constraints such as course capacities and enrollment considerations. This problem arises frequently in academic institutions, where administrators must balance student demand with limited resources and ensure equitable access to course offerings. A traditional optimization approach like the Hungarian Algorithm (HA) guarantees optimal matchings under rigid conditions and assumes a square cost matrix with one-to-one assignments. While mathematically elegant, HA struggles to accommodate real-world considerations such as variable course capacities, enrollment distribution, and other complex constraints. It also suffers from poor scalability in large datasets due to its cubic time complexity. In this thesis, we formalize SCAP as a bipartite graph optimization problem and propose a heuristic solution using Simulated Annealing (SA). SA offers greater flexibility in handling complex, real-world constraints and provides fine-grained control over the optimization process. We implement and test SA against an HA-based implementation on real-world student preference datasets, comparing their accuracy, runtime, and ability to manage enrollment variance—our chosen real-world constraint. Our results show that SA achieves near-optimal solutions with better scalability and adaptability, making it a strong candidate for practical deployment in academic scheduling systems.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Student-Course Assignment Problem . . . . .	2
<b>2</b>	<b>Formalization</b>	<b>6</b>
2.1	Graphs . . . . .	6
2.2	Bipartite Graphs . . . . .	7
2.3	Weighted Graphs . . . . .	8
2.4	Adjacency Matrix . . . . .	9
2.4.1	Biadjacency Matrix . . . . .	11
2.5	Formalizing SCAP . . . . .	11
2.5.1	Weight Matrix . . . . .	11
2.5.2	Raw Preference Function . . . . .	13
<b>3</b>	<b>Background</b>	<b>14</b>
3.1	Optimization . . . . .	14
3.2	Time Complexity and Big O . . . . .	14
3.3	Historical Approaches . . . . .	16
3.3.1	Brute Force (Exhaustive Search) . . . . .	16
3.3.2	Hungarian Algorithm . . . . .	17
3.4	Simulated Annealing . . . . .	18
3.5	Relevant Literature . . . . .	19
<b>4</b>	<b>Implementation</b>	<b>22</b>

4.1	Overview . . . . .	22
4.2	Constraints . . . . .	22
4.2.1	Courses . . . . .	22
4.2.2	Spread . . . . .	22
4.3	Workflow . . . . .	23
4.3.1	Initialization . . . . .	23
4.3.2	Objective Function/Score Evaluation . . . . .	23
4.3.3	Neighborhood Exploration . . . . .	24
4.3.4	Acceptance Criterion . . . . .	24
4.3.5	Stopping Criterion . . . . .	25
4.4	Data Sourcing . . . . .	26
4.4.1	Data Anomalies . . . . .	26
<b>5</b>	<b>Results</b>	<b>29</b>
5.1	Objectives . . . . .	29
5.2	Computational Environment . . . . .	29
5.3	Parameter Settings . . . . .	29
5.3.1	Initial Acceptance Probability . . . . .	29
5.3.2	Preference Map . . . . .	30
5.3.3	Minimum Iterations and Stopping Criterion . . . . .	30
5.3.4	Course Capacities . . . . .	31
5.3.5	Penalty Weight/Variance . . . . .	31
5.4	Accuracy . . . . .	32

5.5	Plots . . . . .	34
5.6	Large Data . . . . .	37
5.7	Variance . . . . .	37
5.8	Special Cases . . . . .	38
<b>6</b>	<b>Discussion</b>	<b>40</b>
6.1	Accuracy . . . . .	40
6.2	Runtime . . . . .	41
6.3	Variance . . . . .	42
6.4	Selection of Preference Maps . . . . .	43
<b>7</b>	<b>Conclusion</b>	<b>44</b>
7.1	Further Research . . . . .	44

## List of Tables

1	Comparison of common time complexities. . . . .	15
2	Dataset sizes . . . . .	26
3	Preference map . . . . .	30
4	Comparison of SA and HA accuracy across all three datasets .	33
5	Results for scaled versions of the 2014 dataset with default parameters . . . . .	37
6	Results for various penalty weights on the 2022 dataset with default parameters . . . . .	38
7	Comparison of results for special cases . . . . .	39

## List of Figures

1	Simple graph . . . . .	6
2	Bipartite graph . . . . .	7
3	Not a bipartite graph . . . . .	8
4	Graph from Figure 1 with varying edge weights . . . . .	9
5	Adjacency matrix $M$ for Figure 1 . . . . .	10
6	Graph of the fictional university with varying edge weights . .	12
7	Weight matrix for student-course preferences, based on Figure 6	13
8	Classification of assignment problems [7] . . . . .	20
9	SA progress for a single trial on the 2018 dataset . . . . .	34
10	Close-up of iterations since improvement counter . . . . .	35
11	Close-up of scores early in runtime, iterations 0 to 160,000 . .	36
12	Close-up of scores later in runtime, iterations 550,000 to 950,000	36

# 1 Introduction

Assignment problems are a fundamental class of optimization problems that arise in many real-world contexts, from workforce scheduling and resource allocation to transportation logistics and academic course registration. At their core, these problems involve assigning one set of entities to another in a way that optimizes a given objective—such as minimizing cost, maximizing efficiency, or ensuring fairness—while adhering to various constraints. In some cases, the objective is straightforward, like pairing workers with jobs based on skill compatibility, while in others, the challenge lies in balancing competing priorities, distributing limited resources, and other more nuanced scenarios. One well-known example of a classical assignment problem is where costs are assigned to agent-task pairings, and the goal is to find an optimal matching that minimizes total cost. However, real-world applications often introduce complexities beyond this standard paradigm, requiring more flexible and dynamic approaches. The **student-course assignment problem** is one such case, where student preferences, course capacities, institutional policies, among many other real-world factors create a non-trivial optimization landscape. (We will refer to the student-course assignment problem as SCAP)

In this thesis, we introduce and formalize SCAP and propose a heuristic solution using a simulated annealing algorithm. As discussed in later chapters, historical approaches to SCAP have several limitations with regard to



real-world constraints that arise, and simulated annealing is shown to navigate these challenges due to its flexible and adaptive nature. Finally, we explore an implementation for this algorithm and assess its efficiency and accuracy compared to traditional algorithms.






## 1.1 The Student-Course Assignment Problem

Imagine that you are in charge of a college registration process, where students have listed their preferences for the specific courses they want to take. Their decisions are based on a variety of factors: Some may prioritize graduation requirements, while others may choose based on interest, time slots, or the professor teaching the course. Additionally, each course has a capacity for how many students it can accommodate. Your task is to assign students to courses, based on their preferences listed, to maximize the overall ‘satisfaction’ of the student body.

Ideally, each student would be assigned their 1st choice. However, real-world constraints often make this infeasible and instead introduce difficult questions: Who gets their 1st choice when there is competition for a class? Who has to settle for a second or third option? How or will you attempt to prevent overcrowding or under-enrolling? Furthermore, the challenge quickly scales in complexity when considering larger numbers of students and courses.

To illustrate this problem, let’s consider a simple example involving a students from a fictional university. Suppose there are five students—Ana, Bob, Cat, Dan, and Eva—and four courses—English, History, Math, and






Science—with capacities of two, three, two, and one seats, respectively. Their listed course preferences are as follows:

	 Ana	 Bob	 Cat	 Dan	 Eva
First	Math	Math	English	Math	Science
Second	English	Science	Math	Science	Math
Third	Science	English	Science	History	History






To which courses will we assign the students? A simple approach is to go down the list and assign each student to their first preference. We can make three assignments before reaching a problem: Ana to Math, Bob to Math, Cat to English. However, Dan's first preference is Math, but the course is already filled as it has a capacity of two after we assigned Ana and Bob. So Dan will be assigned to his second preference. Note that this decision was somewhat arbitrary—it so happened that Dan appeared later in the list, and so the course he wanted was unfortunately already filled by the time we had a chance to assign him. Even more troubling is Eva's assignment—Science and Math are already filled—and so we have to assign Eva to history. Our assignment now looks like:

Ultimately, we were able to satisfy three of the students with their first preference, one student with their second, and a last student with their third.

But is there a way we could have made Dan and Eva happier? The answer

	 Ana	 Bob	 Cat	 Dan	 Eva
First	Math	Math	English	Math	Science
Second	English	Science	Math	Science	Math
Third	Science	English	Science	History	History

is yes. Consider that instead of assigning Dan to his second preference, we assigned Ana to her second, English. We do this because we recognize the trend that these students are STEM-oriented, and Ana is the only one besides Cat who listed a non-STEM course as their second preference. With this, our new assignment might look like:

	 Ana	 Bob	 Cat	 Dan	 Eva
First	Math	Math	English	Math	Science
Second	English	Science	Math	Science	Math
Third	Science	English	Science	History	History

By taking a more “holistic” approach, we were able to achieve a more optimal solution. However, it’s important to note that while this solution is optimal in the context of student preference, it is not optimal with regard to other factors, such as course variety. In this case, History had a capacity for three students, yet we did not assign anyone. Perhaps in another instance

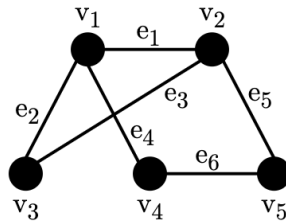
we want to ensure no course is left empty. This added context is important when considering an algorithmic approach to the problem, especially when analyzing which constraints are accounted for and which are not.

Although somewhat trivial (real courses have greater capacities and many more students involved), this example serves to highlight the core of the student-course assignment problem and some of the complex implications and nuances that inevitably arise in real-world contexts.

## 2 Formalization

### 2.1 Graphs

A standard tool in mathematics used to represent distinct nodes and connections between is a **graph**. The following formalization is based on notation outlined in [4]. A graph consists of a set  $V$  of vertices and a set  $E$  of 2-element subsets of  $V$  called edges. The field of **Graph Theory** explores these objects, their properties, and their applications. Pictured below in Figure 1 is a simple example of a graph. It is important to note that the positional arrangement of vertices in a graph is arbitrary, while the more defining feature is the set of edges present within the graph.



$$\begin{aligned}
 V &= \{v_1, \dots, v_5\} \\
 E &= \{e_1, \dots, e_6\} \\
 e_1 &= (v_1, v_2), e_2 = (v_1, v_3), \dots
 \end{aligned}$$

Figure 1: Simple graph

## 2.2 Bipartite Graphs

To model SCAP, we make use of a **bipartite graph**, a special type of graph whose vertices  $V$  can be partitioned into two disjoint sets  $V_1$  and  $V_2$  such that every edge in  $E$  joins a vertex of  $V_1$  with a vertex of  $V_2$ . In other words, no edges exist between pairs of vertices in the same partite set (partition). Figures 2 and 3 below are two examples of graphs. Figure 2 is a bipartite graph, as we can partition the vertices into two disjoint subsets, indicated by the dashed ellipses. Figure 3 however, is not a bipartite graph, due to the “problematic” edges indicated in red that connect vertices within the same set. Therefore, no matter how we rotated or reconfigured the vertices for the graph in Figure 3, there will never be a way to produce two disjoint sets satisfying the bipartite definition.

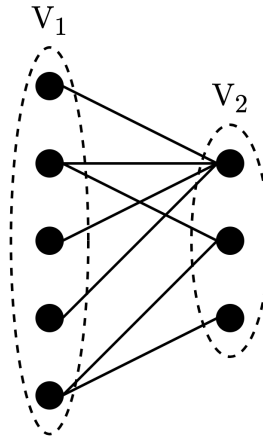


Figure 2: Bipartite graph

Nevertheless, Figure 3 is still a graph. Then why do we require a bipartite

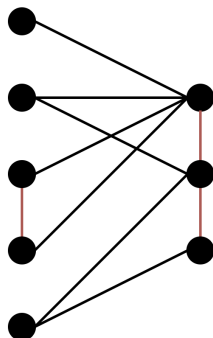


Figure 3: Not a bipartite graph

graph to represent SCAP? If we think about the first set of vertices  $V_1$  as students and the complementary set  $V_2$  as courses, it then follows that the edges represent the assignment of students to courses. For example, consider a student  $s \in V_1$  and course  $c \in V_2$ , then the edge  $e = (s, c) \in E$  translates as “assign student  $s$  to course  $c$ ”. If an edge existed between two students, i.e.  $e = (s_1, s_2)$ , this would mean “assign student  $s_1$  to course  $s_2$ ”, which doesn’t make sense in our context. Clearly, SCAP requires we partition our set of vertices into distinct student-course groups.

## 2.3 Weighted Graphs

We now have an understanding of how the graph in Figure 1 can represent one possible assignment of students to courses. We will from now on refer to such an assignment as a **matching**. As we saw in Chapter 1, a set of students and courses can have many different possible matchings, and some will be “better” or “worse” than others. Fortunately, Graph Theory provides a

higher-level object that allows us to model the concept of student preferences that we introduced. A **weighted graph** is a graph  $G = (V, E, w)$ , where  $(V, E)$  defines the vertex-edge structure as before, and  $w : E \rightarrow \mathbb{R}$  is a weight function that maps edges to real-number values. In traditional assignment problems these can correspond to costs or capacities, but in our case they will represent student preferences. Figure 4 depicts the same graph from Figure 1 with the addition of varying edge weights.

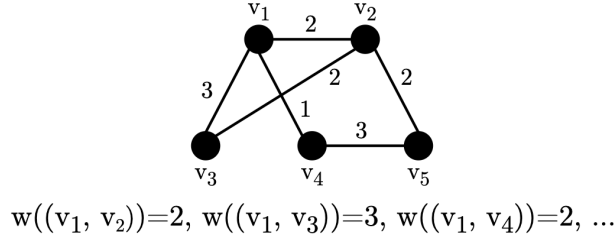


Figure 4: Graph from Figure 1 with varying edge weights

## 2.4 Adjacency Matrix

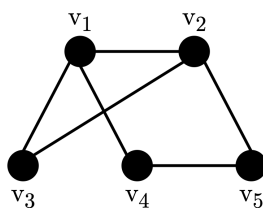
Next we introduce the use of matrices, mathematical objects representing two dimensions of data. Matrices can be applied to Graph Theory if we think of each dimension of a matrix as the set of all vertices. We then have a  $|V| \times |V|$  matrix  $M$ , with binary entries (0 or 1) corresponding to the presence of an edge between any two vertices  $v_i$  and  $v_j$ . We call such a matrix the



**adjacency matrix** of a graph. Formally, we have

$$m_{ij} = \begin{cases} 1, & \text{if } v_i v_j \in E(G) \\ 0, & \text{otherwise} \end{cases}$$

Figure 5 below shows the adjacency matrix  $M$  for the graph in Figure 1. The graph is reproduced for convenience.



$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 5: Adjacency matrix  $M$  for Figure 1

To illustrate the connection between this matrix and the graph, consider that the matrix entry in the second row and third column is 1, i.e.  $m_{2,3} = 1$ . Then, we expect to find an edge between vertices  $v_2$  and  $v_3$ . In the graph,  $e_1$  is the edge between  $v_1$  and  $v_1$ . Note also that  $m_{2,4} = 0$ . The graph agrees that there is no edge between vertices  $v_2$  and  $v_4$ .

### 2.4.1 Biadjacency Matrix

A **biadjacency matrix** is a special type of adjacency matrix for a bipartite graph, where rows correspond to vertices in  $V_1$  and columns correspond to vertices in  $V_2$ . Since  $|V_1| \neq |V_2|$  in general, the biadjacency matrix is not necessarily square (equal number of rows and columns).

## 2.5 Formalizing SCAP

Putting all these formalized tools to work, we can now mathematically model the example scenario of SCAP outlined in our fictional university. The graph in Figure 6 includes all possible assignments between students and courses (a **complete** bipartite graph), including preferences indicated by **colored edges** (green: 1st choice, yellow: 2nd choice, red: 3rd choice). The solid-line edges correspond to one possible matching, while the dashed lines correspond to all other possible matchings. Course capacities are listed in parantheses. Finally, we can rename the vertex subsets  $V_1$  and  $V_2$  to  $C$  and  $S$ , for students and courses.

### 2.5.1 Weight Matrix

It is helpful to introduce a **weight matrix**  $W$ , depicted in Figure 7, an  $|S| \times |C|$  matrix that assigns numerical values to student preferences. Specifically, each entry  $w_{s,c}$  corresponds to student  $s$ 's preference for course  $c$ , where higher preferences generally correspond to a higher entry. For example, one

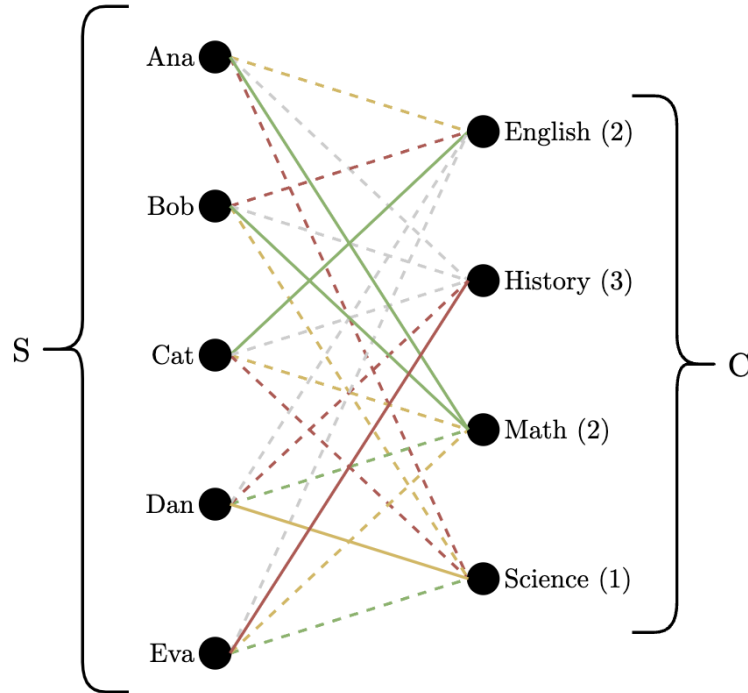


Figure 6: Graph of the fictional university with varying edge weights

way we can assign weights may look like:

Preference	Weight Entry
1st choice	3
2nd choice	2
3rd choice	1
No preference	0

Note that the weights we assign to these preferences is arbitrary, and doesn't necessarily have to be linear.

$$\begin{bmatrix} 2 & 0 & 3 & 1 \\ 1 & 0 & 3 & 2 \\ 3 & 0 & 2 & 1 \\ 0 & 1 & 3 & 2 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

Figure 7: Weight matrix for student-course preferences, based on Figure 6

### 2.5.2 Raw Preference Function

The overall “value” of a matching is quantified by finding the total weight of assigned student-course pairs. This is computed as the sum of the products between the student-course assignment matrix  $M$  (a binary matrix) and the preference weight matrix  $W$ . Formally, for a set of students  $S$  and courses  $C$ , we define the **raw preference function**  $f$  as

$$f(M) = \sum_{s \in S} \sum_{c \in C} m_{s,c} \cdot w_{s,c}$$

where  $m_{s,c}$  represents the assignment of student  $s$  to course  $c$ , and  $w_{s,c}$  denotes the corresponding preference weight. Our goal is to find the matching  $M$  such that  $f$  is largest, which we determine as the solution to the optimization problem.

## 3 Background

### 3.1 Optimization

Optimization is the process of finding the best possible solution to a given problem by maximizing or minimizing an objective function under a set of constraints. In practice, the problem requires an optimization model, allowing the formulation of the objective function and constraints in a mathematical framework that can be systematically analyzed and solved. Readers who have taken a calculus course may be familiar with a common optimization problem involving identifying the maximum value of a function  $f$ , by finding where its derivative is zero,  $\frac{d}{dx}f(x) = 0$ . In general, locating critical points of a mathematical model is a common approach to identifying potential optimal solutions. With regard to SCAP, our optimization model is based on the notation described in Chapter 1, with the objective of allocating students to courses in a way that maximizes overall satisfaction while respecting the real-world constraints we have imposed.

### 3.2 Time Complexity and Big O

When solving optimization problems, it is crucial to consider computational complexity in our approach, which measures how the time or space required to solve a problem scales with input size. A key tool for analyzing time complexity is **Big O notation**, which provides an upper bound on an algorithm's growth rate as the input size increases. Specifically, Big O notation

Complexity	Name	$n = 10$	$n = 100$	$n = 1,000$
$O(1)$	Constant	1	1	1
$O(\log_2 n)$	Logarithmic	$\sim 3.3$	$\sim 6.6$	$\sim 10$
$O(n)$	Linear	10	100	1,000
$O(n \log n)$	Log-linear	$\sim 33$	$\sim 660$	$\sim 10,000$
$O(n^2)$	Quadratic	100	10,000	1,000,000
$O(n^3)$	Cubic	1,000	1,000,000	1,000,000,000
$O(2^n)$	Exponential	1,024	$\sim 1.27 \cdot 10^{30}$	$\sim 1.07 \cdot 10^{301}$
$O(n!)$	Factorial	3,638,800	$\sim 9.33 \cdot 10^{157}$	$\sim 4.02 \cdot 10^{2,567}$

Table 1: Comparison of common time complexities.

describes the worst-case scenario in terms of the number of operations an algorithm needs to perform to find the optimal solution.

Big O notation expresses complexity as a function of input size  $n$ , ignoring constant factors and lower-order terms. For example, if an algorithm performs at most  $5n^2 + 3n + 7$  operations, we simplify its complexity to  $O(n^2)$  because the  $n^2$  term dominates as  $n$  grows large.

Big O notation is especially useful when evaluating an algorithm's feasibility for large inputs. Below is a table summarizing typical Big O complexities along with the number of operations required for varying values of input size  $n$ :

Some algorithms do not always perform the same number of operations for every input; instead, their runtime can vary based on factors like input order or structure. This is why Big O notation represents the worst-case complexity. However, algorithms may also have best-case and average-case complexities. For example, a linear search takes  $O(n)$  in the worst case but

only  $O(1)$  if the target element is found immediately.

### 3.3 Historical Approaches

Here we look at various historical approaches to problems like SCAP, with focus on their respective computational complexities.

Also detailed are the trade-offs associated with each—it is often the case that a more efficient algorithm requires a more stringent model, or makes assumptions that conflict with imposed constraints.

#### 3.3.1 Brute Force (Exhaustive Search)

A brute force approach to any optimization problem involves generating the entire set of possible solutions and evaluating each individual solution under the relevant model to find the most optimal solution. This guarantees an optimal solution but often comes at a prohibitive computational cost. [15]

To illustrate this trade-off for SCAP, imagine a situation where you have 15 students and 4 courses, and for simplicity also assume that the courses have no capacity constraints. Then, each student has 4 different possible assignments, and so the total number of ways we can assign students is  $4^{15}$ . In general, for  $m$  students and  $n$  courses, the **solution space**, or set of all possible solutions, consists of  $n^m$  total matchings. Under this model, the steps for a brute-force algorithm would look like:

1. Generate a matching  $M$ .

2. Evaluate  $\mathbf{F}(M)$
3. Repeat steps 1-2 until for the entire solution space.
4. Output  $M$  for which  $\mathbf{F}(M)$  was maximal/minimal.

Overall, given the time complexity of  $O(n^m)$ , this approach would require  $4^{15}$  possible solutions be checked, for a total of roughly over 1 billion iterations. For modern computers this small case may be manageable, but this time complexity quickly becomes a bottleneck as we increase  $n$  and  $m$ , not to mention the added complexity that comes with introducing additional constraints to the evaluation of  $\mathbf{F}$ .

Further reading on brute force algorithms

### 3.3.2 Hungarian Algorithm

The **Hungarian algorithm (HA)** or **Kuhn–Munkres assignment algorithm** exploits the adjacency matrix of a weighted bipartite graph to determine the optimal matching. The initial version of the algorithm developed by Harold Kuhn [11] in 1955 had a time complexity of  $O(n^4)$ , while a revision by James Munkres [14] in 1957 improved this to  $O(n^3)$ .

HA requires an  $n \times n$  square matrix to model the assignment problem, and then follows a series of matrix operations to ultimately find a minimal or maximal cost assignment. While HA does guarantee an optimal solution, the need to model the problem as a square adjacency matrix is significantly limiting when attempting to impose real-world constraints. For example, if



we want to handle varied course capacities, fairness in distribution, or soft constraints such as prioritization of certain students. Finally, the polynomial time complexity means that HA will quickly reach a bottleneck when met with large input.

### 3.4 Simulated Annealing

The pitfalls with HA described above served as motivation to explore alternative heuristic methods such as **simulated annealing (SA)** to solve SCAP. SA was formalized in [10] (1983) as a heuristic approach to optimization problems, and the name refers to the annealing process in metallurgy, where a material is first heated and then cooled to achieve a desired state.

As a **heuristic** technique, it does not guarantee an optimal solution but can achieve near-optimal results, which may be sufficient in many practical contexts. With its capacity to handle complex constraints and handle larger input more flexibly, we identified it as a viable approach to SCAP and an alternative to the traditional HA for general assignment problems.

At a high-level, SA functions by first generating a random initial matching, and then iteratively performing random alterations, or “moves” to the matching. Moves that improve the matching are accepted, and otherwise are only accepted based on a probability conditioned on a decreasing temperature. In other words, while the initial temperature is high, less-optimal matchings are more likely to be accepted, thus enabling a more holistic exploration of the solution space. As the temperature cools, improvements are

prioritized, pushing us towards an extrema.

Described below is a formalization based on notation defined in [7] and [19].

1. Let  $s = s_0$
2. For  $k = 0$  through  $k_{\max}$ :
 

$T \leftarrow \text{temperature}(k)$   
 Pick a random neighbor,  $s_{\text{new}} \leftarrow \text{neighbor}(s)$   
 If  $P(E(s), E(s_{\text{new}}), T) \geq \text{random}(0, 1)$ :  
 $s \leftarrow s_{\text{new}}$
3. Output: the final state  $s$

To handle additional constraints, we introduce a penalty function  $p$  that is applied directly to the preference score  $f$ . The flexible nature of the penalty function means it can be modified and scaled to encourage any intended behavior. In chapter 4, we discuss how we implemented this functionality with our desired constraints.

### 3.5 Relevant Literature

A classification of common assignment problems is covered in [7], and reproduced in Figure 8. Under this framework, SCAP is considered a general assignment problem (AP), as it involves a one-to-one, or linear matching of

static costs (in our case, student preferences are fixed and no do not evolve over time). Further, SCAP is not an instance of a quadratic assignment problem (QAP), as student-course assignments are not dependent on each other. In theory, SCAP could be extended as a QAP, however, if for example certain students are friends and want to be in the same class together.

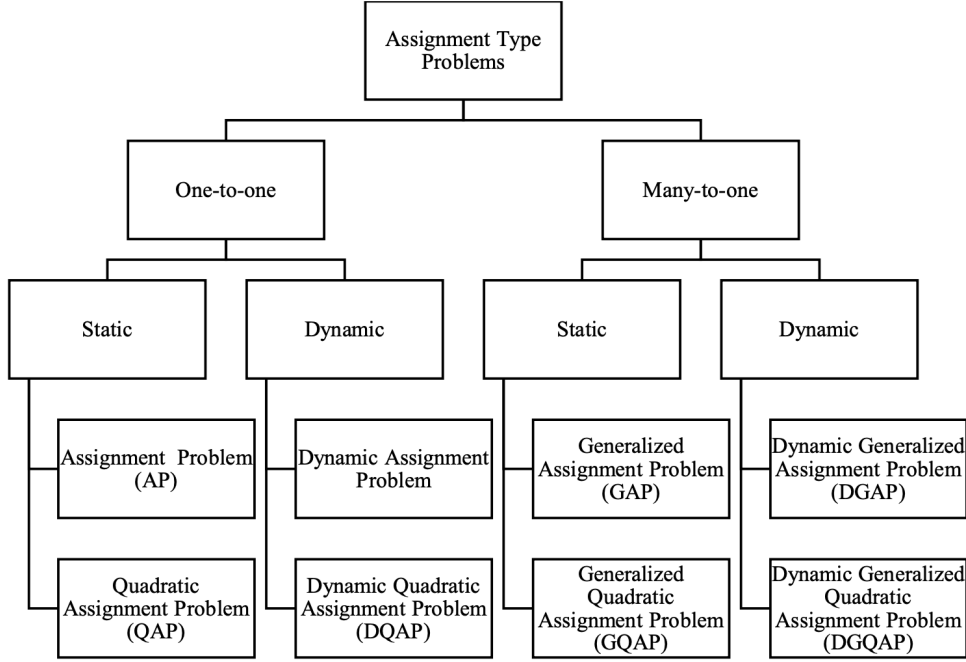


Figure 8: Classification of assignment problems [7]

SA, in general, has been applied to many pre-existing optimization problems. However, SA for graph optimization is a less popular use-case, as most applications to optimization problems involve continuous rather than discrete systems (Graph Theory falls into the latter). Several instances in literature do highlight the use of SA for assignment problems, including [18], [13], [12],

[8], [6], [17], and [16]. [5] explored a niche use-case of SA for a university enrollment-management software package, utilizing a penalty function with fine-tuned constraints. Their results suggest the implementation performed only marginally better than a greedy approach, but noted that being subject to multiple complex constraints inevitably produced many unresolvable enrollment conflicts.

## 4 Implementation

### 4.1 Overview

In this section, we describe the implementation of our student-course assignment algorithm, which applies SA to optimize course allocations based on student preferences while handling various practical constraints. The algorithm starts with a random assignment and iteratively refines the solution by probabilistically accepting beneficial moves while avoiding suboptimal local minima.

### 4.2 Constraints

As discussed previously, SCAP is constrained by many real-world limitations, which we incorporate into our algorithm.

#### 4.2.1 Courses

Each course has a predefined capacity, meaning that at no point can more students be assigned to a course than its limit. This is enforced in the random initialization stage and throughout the drop-add move, where any proposed assignment that would exceed a course’s capacity is rejected.

#### 4.2.2 Spread

To ensure a fair distribution of students across courses, we introduce a spread metric, calculated as the variance in the number of students assigned to

each course. This metric serves as the basis of a penalty function  $p$  when evaluating the score for a matching—solutions with high variance (uneven course loads) are penalized, making balanced assignments more favorable. This aligns with the mathematical formulation where we define a fairness criterion to avoid overcrowding and under-enrollment in courses. This aligns with broader principles in the study of fairness optimization. [2]

### 4.3 Workflow

#### 4.3.1 Initialization

The algorithm begins by generating an initial **random matching**  $M_0$  of students to courses, respecting course capacity constraints.

#### 4.3.2 Objective Function/Score Evaluation

Our raw preference function  $f$  and penalty function  $p$  are calculated for the current matching  $M_i$ . The ultimate value of the matching is given by the **objective function  $\mathbf{F}$** :

$$\mathbf{F}(M_i) = f(M_i) - p(M_i)$$

For simplicity, we will use the term **score** to refer to the value of  $\mathbf{F}$  for a particular matching  $M$ .

### 4.3.3 Neighborhood Exploration

At each iteration, we perform one of two moves:

1. **Drop-add:** Select a student  $s \in S$  at random assigned to course  $c_i \in C$ .  
Reassign them to a new course  $c_j \in C$ , with respect to course capacities.

$$(s, c_i) \rightarrow (s, c_j) : i \neq j$$

2. **Swap:** Select two unique students  $s_i, s_j \in S$  assigned to courses  $c_k, c_l \in C$  respectively. Assign student  $s_i$  to course  $c_l$ , and student  $s_j$  to course  $c_k$ .

$$(s_i, c_k), (s_j, c_l) \rightarrow (s_i, c_l), (s_j, c_k) : i \neq j, k \neq l$$

We select one of these two possible moves at random, and after performing it, we obtain a new candidate matching  $M_{i+1}$ .

### 4.3.4 Acceptance Criterion

We calculate  $\mathbf{F}(M_{i+1})$  and compare it to  $\mathbf{F}(M_i)$ .

If  $\mathbf{F}(M_{i+1}) > \mathbf{F}(M_i)$ , the move has improved the overall score of the matching, and we accept it as the matching for the next iteration by setting:

$$M_i = M_{i+1}$$

However if  $\mathbf{F}(M_{i+1}) \leq \mathbf{F}(M_i)$ , the move has not improved the overall

score, and so we accept it following our probabilistic approach:

1. Calculate  $\Delta \mathbf{F}$  for our current iteration  $i$ :

$$\Delta \mathbf{F} = \mathbf{F}(M_{i+1}) - \mathbf{F}(M_i)$$

2. Find the acceptance probability given temperature  $T_i$ :

$$P(\text{accept move} \mid T_i) = \exp\left(\frac{\Delta \mathbf{F}}{T_i}\right)$$

3. Determine whether to accept  $P(\text{accept move} \mid T_i)$  using a pseudo-random number generator.

4. Scale the temperature based on our cooling rate  $k$ :

$$T_{i+1} = T_i \cdot k; 0 < k < 1$$

#### 4.3.5 Stopping Criterion

Our program terminates after a fixed number of consecutive iterations without an improvement to the score. Other commonly used stopping criterion include execution time, maximum number of iterations performed, or maximum number of consecutive iterations without an improvement. [7]



Year	Students	Courses
2014	194	20
2018	279	23
2022	322	22

Table 2: Dataset sizes

## 4.4 Data Sourcing

We primarily tested on three real-world datasets, composed of anonymized entries from incoming freshmen at Drew University in the years 2014, 2018, and 2022. Students were asked to list their top 5 preferences of first-year seminar classes. The size of the datasets are listed below.

The datasets were provided in .xlsx format, and custom utility scripts in Python were created to convert the data into a structured .csv format that our SA implementation could interpret.

### 4.4.1 Data Anomalies

We feel it is important to mention several fringe cases that were discovered in the raw data, as it reflects the challenges of working with real-world datasets. These include formatting errors, typos, or other scenarios where the data doesn't conform to an expected schema or logic. Below we list some examples of these instances in the data we worked with and how they were ultimately handled.

- Missing preferences: some students were missing an  $n$ th choice. For example, a student may have provided their 1st, 2nd, 4th and 5th

choice but not a 3rd choice. A more common case was that the student only had a 1st choice but no other preferences given. One explanation for this is that the student’s assignment was already known, and their entry here was manually inserted or just a formality. Either way, this doesn’t actually affect the functionality of either algorithm, but it does introduce an unrealistic scenario as we do assume each student provided five total preferences.

- Duplicate courses: some students were listed as having the same course for multiple of their choices. In such a case, we treat the higher choice with priority. For example, if a student listed course  $c$  as both their 2nd and 4th choice, then we would treat course  $c$  as their 2nd choice and they would not have a 4th choice.
- Duplicate preferences: some students had assigned two different courses the same preference, i.e. having multiple 3rd choices. Thus, the student had actually provided preferences for more than five courses. Despite this being an unrealistic scenario, there’s really no “fair” way to handle it, so we just accept the data as it is since this won’t affect the functionality of either algorithm.
- One student in the 2022 dataset had only listed their 5th choice and no other preferences given. Because of this, a perfect score wasn’t actually possible, as this student could only be assigned to that choice, their 5th choice, by default. This is why the maximum score for HA in Table 7

is 100 points short of the theoretical maximum.

## 5 Results

### 5.1 Objectives

We performed three sets of trials, each assessing a different metric of the problem:

- Overall accuracy of SA vs. HA in finding an optimal or close-to-optimal solution.
- Runtime and efficiency for arbitrarily large datasets.
- Ability for SA to balance class satisfaction with course enrollment variance.
- Special cases involving predetermined course capacities, and SA and HA handle each.

### 5.2 Computational Environment

The programs were written in Python3 and run on a Apple M2 CPU with 8 GB of RAM on MacOS operating system.

### 5.3 Parameter Settings

#### 5.3.1 Initial Acceptance Probability

We set the initial acceptance probability,  $P_i$  to be 0.7. We made this choice based on [7], that found that a  $P_i$  of 0.7 consistently produced more optimal

matchings for their SA implementation than other choices such as 0.6, 0.9, and 0.99.

### 5.3.2 Preference Map

We use the following preference map:

Listed Preference	Weight
1st	100
2nd	30
3rd	10
4th	5
5th	0
Not listed	-1000

Table 3: Preference map

Recall the preference map is our way of defining how valuable a student's 1st choice is compared to their 2nd, etc. There is, of course, no “correct” way to assign these values, but the assignments have the potential to greatly affect the results of the matchings. One conscious choice we made was to make 1st choices worth 100, which allows the maximum theoretical score to be simply calculated by multiplying the number of students:  $100 \cdot |S|$ . More discussion on the preference map can be found in section 6.4.

### 5.3.3 Minimum Iterations and Stopping Criterion

A minimum iterations parameter is passed to the algorithm, denoting the minimum number of iterations to perform. After this threshold, the program

terminates after a fixed number of iterations without any improvements.

By default, the SA implementation uses a minimum iterations of 100,000 and stopping iterations of 10,000. However for our accuracy comparison, we choose both parameters to be 1,000 times the size of the dataset (specifically, the number of students). This ensures SA has enough time to find a near-optimal solution and can sufficiently explore the problem’s landscape.

#### 5.3.4 Course Capacities

To choose the maximum course capacities for SA, we take a multiple of the average class size, rounded up. That is,

$$\lceil a \cdot \frac{|S|}{|C|} \rceil, a > 0$$

For our trials, we use  $a$  of 1.5 and 2. This approach ensures that our course capacities are sufficiently large for the given dataset, and the extra space allows for some ”play” in the algorithm to test varying course fulfillment.

#### 5.3.5 Penalty Weight/Variance

In order to better compare SA and HA, we do not include any form of penalty calculation for our trials except for the trials in section 5.7, where we focus explicitly on the variance penalty and it’s implications, discussed in section 6.3.

## 5.4 Accuracy

In order to assess SA’s accuracy, we ran 10 trials for each dataset, half with  $a = 1.5$  and the other half with  $a = 2$ . Then, to run a subsequent trial of HA, we used the course enrollment sizes that SA produced for that respective trial. The maximum theoretical scores are also provided, which is the (often impossible) case that every student received their 1st choice. Finally, the average preferences provide an unweighted measure of score—0 meaning students got their 1st choice on average, 1 meaning they got their 2nd, and so on. The “worst” value here would be a 5, meaning students got no of their preferred courses, on average.

Dataset	$a$	SA Score	HA Score	Max Theoretical Score	SA Average Preference	HA Average Preference
2014	1.5	14480	14565	19400	0.474	0.454
		14485	14555		0.490	0.464
		14525	14565		0.479	0.454
		14505	14540		0.485	0.464
		14455	14535		0.485	0.469
	2	16630	16650		0.211	0.206
		16540	16640		0.222	0.216
		16630	16650		0.211	0.206
		16630	16650		0.211	0.206
		16650	16650		0.206	0.206
2018	1.5	26480	26500	27900	0.075	0.072
		26500	26500		0.072	0.072
		26500	26500		0.072	0.072
		26500	26500		0.072	0.072
		26500	26500		0.072	0.072
	2	27690	27690		0.011	0.011
		27690	27690		0.011	0.011
		27690	27690		0.011	0.011
		27690	27690		0.011	0.011
		27690	27690		0.011	0.011
2022	1.5	26750	26810	32200	0.261	0.252
		26805	26850		0.255	0.245
		26805	26830		0.255	0.248
		26725	26850		0.267	0.245
		26750	26790		0.261	0.255
	2	29090	29090		0.146	0.146
		29090	29090		0.146	0.146
		29090	29090		0.146	0.146
		29090	29090		0.146	0.146
		29090	29090		0.146	0.146

Table 4: Comparison of SA and HA accuracy across all three datasets



## 5.5 Plots

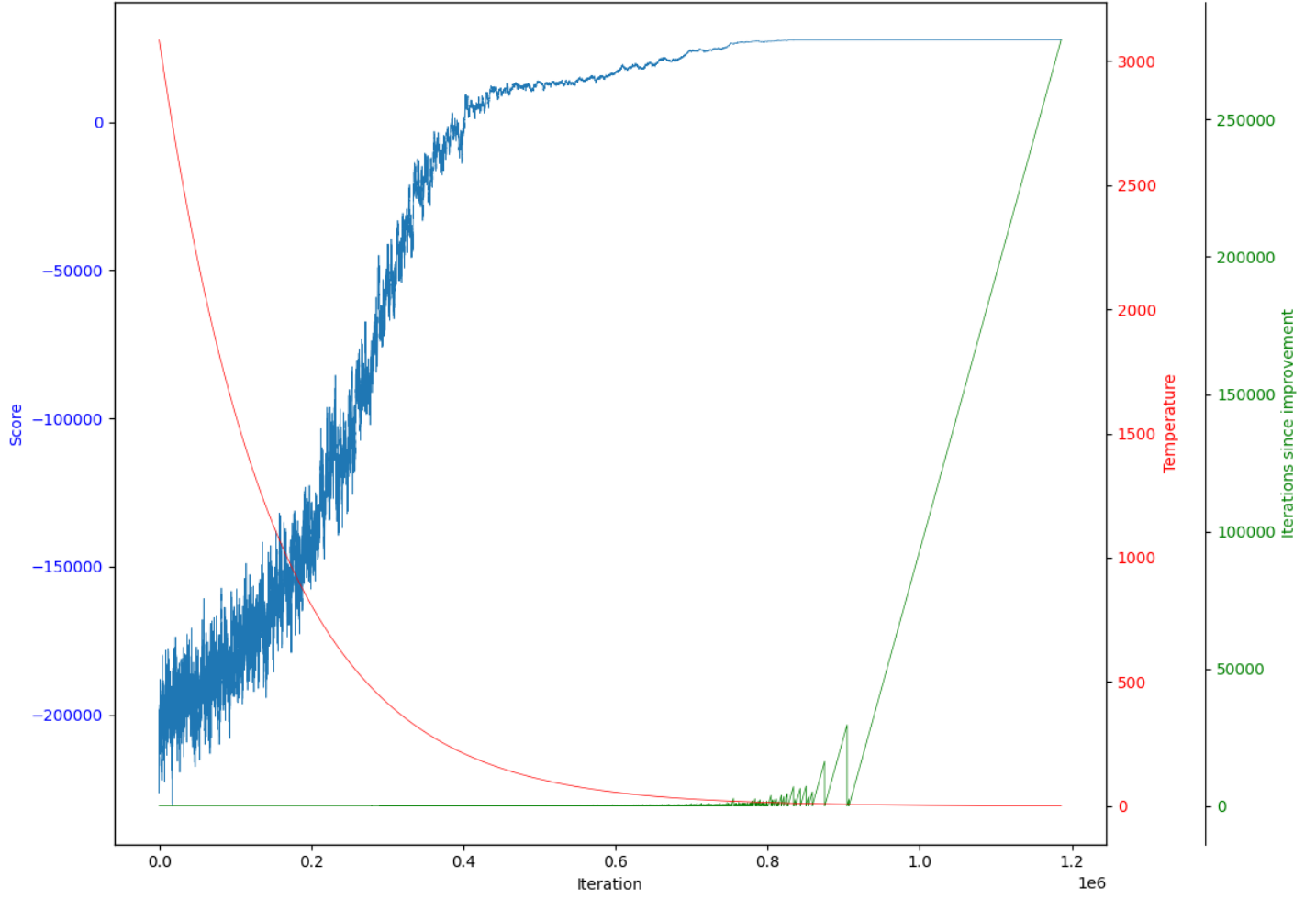


Figure 9: SA progress for a single trial on the 2018 dataset

The plot in Figure 9 shows the evolution of a single trial of SA over 1,186,085 iterations on the 2018 dataset. Each aspect of the plot reflects the behavior of three core variables in our optimization process:

- The score (blue line) fluctuates heavily to explore the landscape while the temperature (red line) is high, but ultimately approaches a near-optimal solution as the temperature cools. It also becomes more conservative when accepting worse solutions.
- Recall our acceptance logic: at iteration 0, we accept 70% of “bad” moves, but when we hit the `min_iterations` parameter (for the 2018 dataset, `min_iterations = 279000`) we only accept 10%, after which our acceptance probability approaches 0%.
- The number of iterations since improvement (green line) increases linearly after we reach `min_iterations`, as it is simply a counter of iterations. It only resets to zero at each improvement, resulting in a saw-tooth pattern (see Figure 10).

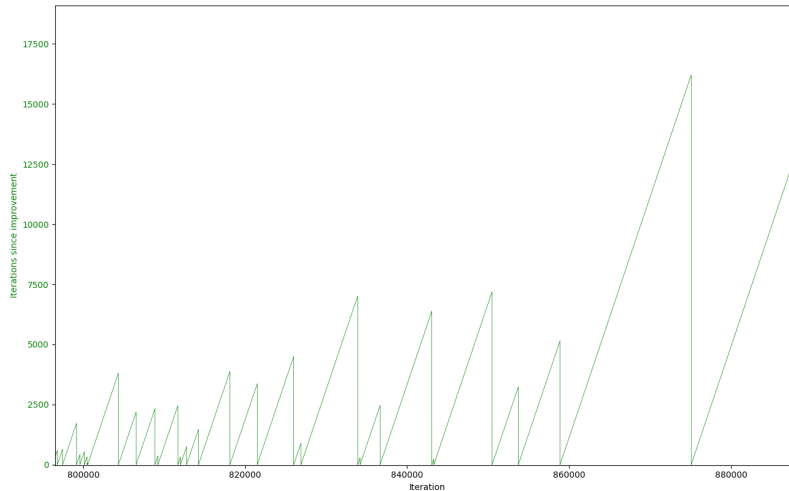


Figure 10: Close-up of iterations since improvement counter

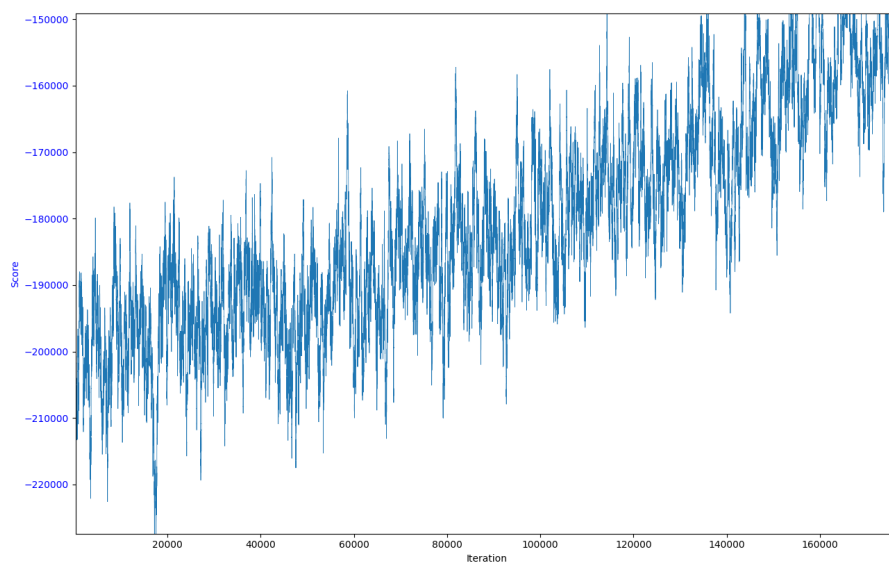


Figure 11: Close-up of scores early in runtime, iterations 0 to 160,000

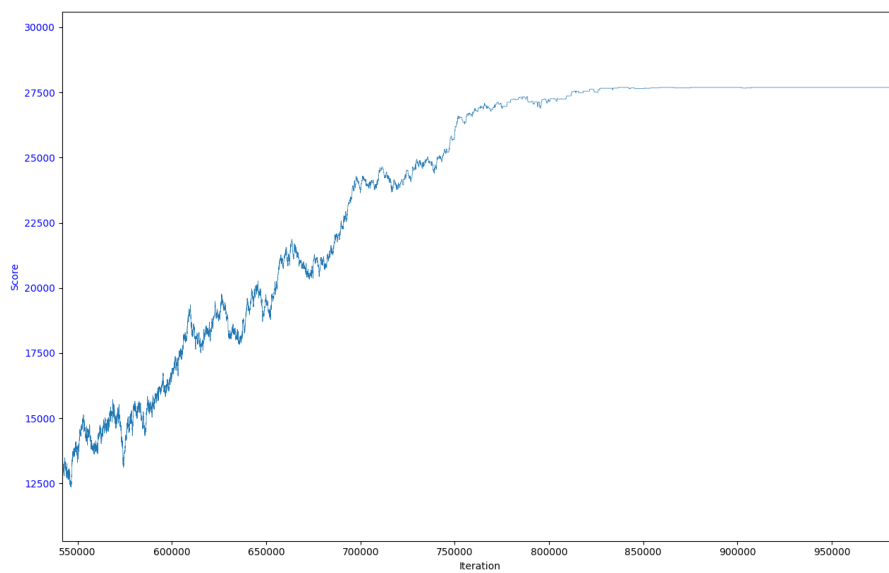


Figure 12: Close-up of scores later in runtime, iterations 550,000 to 950,000

## 5.6 Large Data

In the interest of comparing runtime and efficiency of the algorithms, we manually scaled the data by duplicating the existing rows and columns, enabling us to run tests on larger datasets. Literature often notes SA for scaling more gracefully than exact algorithms in larger spaces. [1] The course capacities that were used for these tests were based on the weighted proportions method, described in section 5.8b.

Scale factor	SA Score	HA Score	SA Time	HA Time
1	16360	16480	12.0s	0.229s
2	32390	32960	24.6s	1.84s
4	63780	65920	49.4s	14.8s
8	127515	131840	160s	122s
16	252185	263680	762s	958s

Table 5: Results for scaled versions of the 2014 dataset with default parameters

## 5.7 Variance

While our other trials do not include any affect of the course enrollment variance penalty, here we examine what implications adjusting the variance penalty weight has on the resulting assignment. Since the calculation of variance is dependent on the scale of our preference weights, we pick penalty weights that are within a reasonable window.

Penalty weight	Variance	Score	Average Preference
0	78.9	29000	0.152
1	72.8	29000	0.152
10	72.8	29010	0.167
50	43.6	28410	0.199
100	12.4	26235	0.360
250	3.60	24880	0.478
500	0.686	23875	0.584
750	0.595	23560	0.630
1000	0.322	23200	0.599
10000	0.231	22285	0.677

Table 6: Results for various penalty weights on the 2022 dataset with default parameters

## 5.8 Special Cases

The data lends itself to consideration of some particular special cases. Specifically, cases where we can predefine the course capacities based on a measure of the popularity of each course. We explore two methods of doing so:

- (a) Set the capacity for each course to the respective number of students that listed the course as their 1st choice.
- (b) Set the capacity for each course to the sum of all listed preferences for that specific course divided by the total sum of all preferences across all courses. For our case, the sum of all preferences across all courses will be  $|S| \cdot (5 + 4 + 3 + 2 + 1) = |S| \cdot 15$ , as we assume each student listed preferences 1 through 5.

Method	Dataset	SA Score	HA Score	Max Theoretical Score	SA Average Preference	HA Average Preference
a	2014	19400	19400	19400	0	0
	2018	27305	27900	27900	0.039	0
	2022	31410	32100*	32200	0.062	0
b	2014	15360	16480	19400	0.438	0.237
	2018	24415	25800	27900	0.229	0.108
	2022	28105	29160	32200	0.208	0.143

Table 7: Comparison of results for special cases

\*See section 4.4.1 on data anomalies.

## 6 Discussion

### 6.1 Accuracy

Across all three datasets, SA scored 12% on average below the maximum theoretical score, while HA scored 11.88% below. Between both algorithms, SA scored on average 1.98% below HA across all trials. These disparities are less explicit in certain datasets, specifically in 2018, where in only one trial did SA perform worse than HA.

Recall that by the nature of both algorithms, we expect HA to score the same as SA or greater, since it guarantees an optimal solution. This fact, along with the results, indicate that SA achieved near-optimal or true optimal solutions for most of the trials on the 2018 data and 2022 for  $a = 2$ . One explanation for this is due to the composition of the data itself. While 2018 and 2022 presumably presented more favorable optimization landscapes, 2014 may have involved more "lose-lose" scenarios. For example, courses that were in heavy demand by a majority of students, and other courses that had little demand overall.

Another insight is that larger values of  $a$  resulted in higher scores. This is intuitive, as with greater course capacities, more students' higher preferences should be accommodated. A large enough  $a$  will guarantee the maximum theoretical score, specifically when all the course capacities,  $\lceil a \cdot \frac{|S|}{|C|} \rceil$ , are greater or equal to the greatest number of 1st choices given to any one course. Our special case in section 5.8 tested exactly this, and resulted in optimal

scores for HA and near-optimal scores for SA. In fact, the greedy algorithm would also succeed under these conditions.

Normally, our SA implementation would not be able to handle a case where the number of students equals the number of seats, as trying to perform a drop/add when all the courses are technically already full would result in an error. To circumvent this, we ran these tests using only swap operations. Ultimately, SA performed quite well under these circumstances. While it did reach the maximum possible solution for the 2014 data, it fell slightly short in 2018 and 2022. In theory, there are two reasons SA did not achieve the optimal solution here. First is that it got "stuck" in another extrema and was not able to navigate out. This is more likely the case in a more complex optimization landscape with multiple regions of high scoring potential. As indicated in our initial trials, 2014 was more limited with regard to scoring potential, so perhaps there was less chance of SA getting "lost" or "distracted" with less competing regions. Finally, it could just be that SA didn't have enough time to run. Provably, SA will reach a locally optimal solution given it can perform enough iterations to get there.

## 6.2 Runtime

Our results on larger datasets highlight a core advantage of SA when the number of students and courses becomes computationally unfeasible for HA. Recall the time complexity for HA is  $O(n^3)$ , and our data agrees: each doubling of the data increased the runtime of HA by a factor of  $2^3 = 8$ . Meanwhile, the



runtime of SA showed linear growth until 8x data size. We suspect this may be due to memory limitations, which would create a bottleneck that varies from system-to-system depending on physical memory constraints. The inherent value of a heuristic is also reflected here—SA was able to reach up to 75-90% of it’s ultimate score within as little as 50-60% of it’s total runtime. This enables more control over the extent of computational resources which should be dedicated to a problem, when scores past a “good-enough” threshold may be a case of diminishing returns.

### 6.3 Variance

Our results in Table 6 suggest a trade-off between overall score and enrollment variance. As we increase the penalty weight, both the score and average preference of the matching become less favorable, while the variance itself approaches zero. Due to the number of courses not evenly dividing the number of students, the minimum possible variance will ultimately be non-zero and positive, with course enrollments within a small range ( $\pm 1$ ) of half the average course size, rounded to the nearest integer:  $\lfloor \frac{S}{C} \rfloor \pm 1$ . For example, for the 2022 dataset that was tested on, high values of the penalty weight resulted in all courses being fulfilled with sizes of 14 and 15:  $\lfloor \frac{322}{22} \rfloor \pm 1 = \lfloor \sim 14.6 \rfloor \pm 1 = \{14, 15\}$ . Interestingly, the results show a steep inflection for penalty weights between 10 and 250, indicating this range may produce more balanced results for this specific dataset and our algorithmic parameters.

## 6.4 Selection of Preference Maps

The question of how much worth should be allotted to some ordinal set of values, as in the case of ranked preferences, is an interesting field of study itself [3], but beyond the scope of this research. As previously mentioned in section 5.3.2, there is no “correct” or “optimal” preference map. It ultimately is a philosophical choice at the discretion of the implementer. Our decisions were a result of considering that the difference between 5th, 4th, and 3rd choice isn’t very significant, but whether a student is given their 1st choice vs. their 2nd or 3rd should carry more weight. At last, a course that wasn’t listed by the student should be weighted devastatingly low in comparison.

## 7 Conclusion

This thesis explored the student-course assignment problem (SCAP) through the lens of bipartite graph optimization, comparing the classical methods of the Hungarian Algorithm (HA) with the heuristic approach of Simulated Annealing (SA). While HA guarantees optimal matchings under tightly defined conditions, it lacks flexibility in the presence of many real-world constraints and can be computationally infeasible for large sizes of data that often arise in practical contexts. SA offers a compelling alternative, through its ability to adapt naturally to complex optimization problems. Our implementation and results show that SA produces near-optimal matchings while respecting custom constraints, such as the variance of course enrollment.

### 7.1 Further Research

The nature of the problem introduces many avenues of expansion. First, with the flexibility of SA, there are no limits with regard to constraints one may want to impose, and therefore further study could look at aspects including, but not limited to:

- Graduation/curriculum requirements
- Scheduling/time conflicts
- Distance, travel time and cost
- Diversity initiatives such as race, gender, personal background, etc.

Next, fine-tuning of SA parameters can be explored further. While existing research sometimes mentioned settings that performed well, variables such as the preference map, stopping criterion, and acceptance probability logic introduce more nuanced aspects of the problem that may have a greater impact on overall accuracy and performance. For example, literature suggests variable or adaptive cooling techniques may improve SA’s exploration of the solution space. [9]

Finally, future work could explore hybrid approaches, such as using SA to generate candidate matchings and refining them with HA, steepest descent, or other exact methods.

Ultimately, SA proves to be a powerful, adaptable framework for SCAP and similar assignment problems. Its value lies not just in finding “good enough” solutions, but in offering decision-makers the flexibility, practicality, and transparency needed to handle complex, real-world constraints.

## References

- [1] Aarts, E. H., & Korst, J. H. (1989). *Simulated Annealing and Boltzmann Machines*. John Wiley & Sons.
- [2] Bertsimas, D., Farias, V. F., & Trichakis, N. (2011). "The Price of Fairness." *Operations Research*.
- [3] Bogomolnaia, A., & Moulin, H. (2001). "A New Solution to the Random Assignment Problem." *Journal of Economic Theory*.
- [4] Chartrand, G., & Zhang, P. (2012). *A First Course in Graph Theory*. Dover Publications.
- [5] Ciebiera, K., & Mucha, M. (2014). "Student-class assignment optimization using simulated annealing." *EUNIS 2014 Congress*. Retrieved from [https://eunis.org/download/2014/papers/eunis2014\\_submission\\_49.pdf](https://eunis.org/download/2014/papers/eunis2014_submission_49.pdf)
- [6] Connolly, D. T. (1996). "An improved annealing scheme for the QAP." *European Journal of Operational Research*, 89(2), 429–438. [https://doi.org/10.1016/S0360-8352\(96\)00265-3](https://doi.org/10.1016/S0360-8352(96)00265-3)
- [7] Dhungel, Y. (2022). *Simulated Annealing Heuristics for the Dynamic Generalized Quadratic Assignment Problem*. West Virginia University Research Repository. <https://doi.org/10.33915/etd.11162>

- [8] Gallego, J. C., & Briceño, W. (2020). "School assignment using simulated annealing to minimize distance." In *Proceedings of the 2020 IEEE Colombian Conference on Applications in Computational Intelligence (ColCACI)* (pp. 1–5). IEEE. <https://doi.org/10.1109/ColCACI49338.2020.9281242>
- [9] Karabin, M., & Stuart, S. J. (2020). *Simulated annealing with adaptive cooling rates*. The Journal of Chemical Physics, 153(11), 114103. <https://doi.org/10.1063/5.0018320>
- [10] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). "Optimization by Simulated Annealing." *Science*, 220(4598), 671–680.
- [11] Kuhn, H. W. (1955). "The Hungarian Method for the Assignment Problem." *Naval Research Logistics Quarterly*, 2(1–2), 83–97. <https://doi.org/10.1002/nav.3800020109>
- [12] Marinov, M., & Marinova, V. (1998). "Optimization of the allocation of classrooms to classes using simulated annealing." In *Proceedings of the 20th International Conference on Information Technology Interfaces* (pp. 575–580). IEEE. <https://doi.org/10.1109/ITI.1998.726655>
- [13] Matsumura, S. (2018). "A simulated annealing approach to the student-project assignment problem." *American Journal of Physics*, 86(9), 701–705. <https://doi.org/10.1119/1.5042918>

- [14] Munkres, J. (1957). "Algorithms for the assignment and transportation problems." *Journal of the Society for Industrial and Applied Mathematics*, 5(1), 32–38. <https://doi.org/10.1137/0105003>
- [15] Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications.
- [16] Rao, R. V., & Patel, V. K. (2004). "Applying simulated annealing to the multidimensional assignment problem." In S. Y. Chen (Ed.), *Advances in Metaheuristics for Hard Optimization* (pp. 25–41). World Scientific. [https://doi.org/10.1142/9789812796592\\_0003](https://doi.org/10.1142/9789812796592_0003)
- [17] Sahu, A., & Tapadar, R. (2007). "Solving the assignment problem using genetic algorithm and simulated annealing." *IAENG International Journal of Applied Mathematics*, 36(1). Retrieved from [https://www.iaeng.org/IJAM/issues\\_v36/issue\\_1/IJAM\\_36\\_1\\_7.pdf](https://www.iaeng.org/IJAM/issues_v36/issue_1/IJAM_36_1_7.pdf)
- [18] Shojaei Ghandeshtani, K., Seyedkashi, S. M. H., Mollai, N., & Neshati, M. M. (2010). "New simulated annealing algorithm for quadratic assignment problem." In *Proceedings of the Fourth International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP 2010)*. Retrieved from [https://personales.upv.es/thinkmind/dl/conferences/advcomp/advcomp\\_2010/advcomp\\_2010\\_5\\_10\\_20111.pdf](https://personales.upv.es/thinkmind/dl/conferences/advcomp/advcomp_2010/advcomp_2010_5_10_20111.pdf)

- [19] Teferra, D. (2019). "Simulated annealing: Theory and applications." ResearchGate. Retrieved from [https://www.researchgate.net/publication/338294434\\_Simulated\\_Annealing\\_Theory\\_and\\_Applications](https://www.researchgate.net/publication/338294434_Simulated_Annealing_Theory_and_Applications)