A Heuristic Algorithm for Student-Project Allocation Problem

Nguyen Thi Uyen[1,2], Giang L. Nguyen[2], Canh V. Pham[3], Tran Xuan Sang[4], and Hoang Huu Viet[1(✉)]

[1] School of Engineering and Technology, Vinh University, Vinh City, Vietnam
{uyennt,viethh}@vinhuni.edu.vn
[2] Institute of Information Technology, VAST, Hanoi, Vietnam
nlgiang@ioit.ac.vn
[3] ORLab, Faculty of Computer Science,, Phenikaa University, Hanoi, Vietnam
canh.phamvan@phenikaa-uni.edu.vn
[4] Cyber School, Vinh University, Vinh, Nghe An, Vietnam
sangtx@vinhuni.edu.vn

Abstract. The Student-Project Allocation problem with lecturer preferences over Students with Ties (SPA-ST) is to find a stable matching of students and projects to satisfy the constraints on student preferences over projects, lecturer preferences over students, and the maximum number of students given by each project and lecturer. This problem has attracted many researchers because of its wide applications in allocating students to projects at many universities worldwide. However, the main weakness of existing algorithms is their high computational cost. In this paper, we propose a heuristic algorithm to improve solution quality and execution time for solving the SPA-ST problem of large sizes. Experimental results on randomly generated datasets show that our algorithm outperforms the state-of-the-art algorithm regarding solution quality and execution time.

Keywords: Student-Project Allocation Problem · Heuristic algorithm · Blocking pairs · Stable matching · MAX-SPA-ST · Large sizes

1 Introduction

The problem of allocating students to projects based on their preferences, called SPA [2,7,18,19,21], is to find a stable matching of the students and the projects to satisfy the constraints on their preference lists. This problem has played an important role at many universities in the world [6,10,14,17]. However, it has a strict constraint on preference lists: students and lecturers must rank given projects in a certain order that cannot cover a reality case: two projects are ranked in the same order of preference in their lists. Abraham et al. [1] proposed a new variant of the SPA problem, called preferences over Students containing Ties (SPA-ST), with an adjustment that helps students have more options

T. N. Dinh and M. Li (Eds.): CSoNet 2022, LNCS 13831, pp. 280–291, 2023.
https://doi.org/10.1007/978-3-031-26303-3_25

when choosing projects from the lecturers: the list of preferences of lecturers and students can contain equality relations.

In the SPA-ST problem, there are three stability criteria of matching: *weakly stable*, *strongly stable*, or *super-stable* matching [19,20]. Irving et al. [22] proved that a super-stable matching is strongly stable, and a strongly stable matching is a weakly stable matching. If a super-stable matching is found, all weakly stable matchings have the same sizes. Besides, they also showed that weakly stable matching always exist and have different sizes [11]. Therefore, the problem of finding a maximum size weakly stable matching is known as the NP-hard problem [22]. In addition, strongly stable or super-stable matchings, whose goal is to find a stable matching with a maximum number of matched students, may not exist because the constraints are too tight [15,20,22].

Practically, the problem of finding a maximum weakly stable matching is the most suitable for real-life applications because it focuses on assigning as many students as possible to projects. This problem is known as the MAX-SPA-ST problem and it has attracted much attention from the research community because of its application in education optimization problems. Some universities applied the MAX-SPA-ST problem to optimize the assignment of projects for lecturers to satisfy the constraints of large-scale problems, such as the School of Computer Science, the University of Glasgow [14], Faculty of Science, University of Southern Denmark [6], Department of Computer Science, York University, and elsewhere [3–5,8]. Unfortunately, finding an efficient algorithm to solve the MAX-SPA-ST of large sizes is still a main challenge for the research community. The approximation algorithm is one of the popular methods for solving the SPA-ST problem [1,12,16,21]. Cooper et al. [7] first proposed a 3/2-approximation algorithm named APX for solving the MAX-SPA-ST based on Király's idea [13]. Latter, Manlove et al. [15] presented an Integer Programming (IP) model to find a strongly stable matching for SPA-ST. Recently, Olaosebikan et al. [19] provided a polynomial-time algorithm to find a super-stable matching. However, they proved that it might not exist for the SPA-ST problem.

In this paper, we call a *weakly stable* matching a *stable* matching. Accordingly, we propose a new heuristic algorithm for solving the MAX-SPA-ST problem. The main difference between our algorithm from the others is that we design *two heuristic functions* to estimate reasonable solutions: the first is used to choose the best project to match with students, while the second is to discard the worst student from the matching when the project or lecturer is *over-subscribed*. By combining two functions, our algorithm can find a suitable solution within a reasonable time. Experimental results show that our proposed algorithm performs better in terms of the average execution time and percentage of perfect matchings compared to the most recent APX algorithm [7] for solving the MAX-SPA-ST problem of large sizes.

The rest of this paper is organized as follows. Section 2 presents preliminaries of the SPA-ST problem, Sect. 3 describes our proposed algorithm, Sect. 4 discusses our experimental results, and Sect. 5 concludes our work.

2 Preliminaries

An SPA-ST instance consists of a set $\mathcal{S} = \{s_1, s_2, \cdots, s_n\}$ of students, a set $\mathcal{P} = \{p_1, p_2, \cdots, p_q\}$ projects, and a set $\mathcal{L} = \{l_1, l_2, \cdots, l_m\}$ of lecturers. Each student s_i ranks a set of acceptable projects in order of preference containing ties. Each lecturer $l_k \in \mathcal{L}$ offers a set of projects and ranks a set of students in a preference list containing ties. Each lecturer has a capacity $d_k \in \mathbb{Z}^+$ indicating the maximum number of students assigned to l_k. Each project $p_j \in \mathcal{P}$ is offered by one lecturer and has a capacity $c_j \in \mathbb{Z}^+$ indicating the maximum number of students that can be assigned to it. For example, an SPA-ST instance given in Table 1 consists of a set of $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ of students, a set $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$ of projects, and a set $\mathcal{L} = \{l_1, l_2, l_3\}$ of lecturers.

Table 1. An example of SPA-ST instance

Students' preferences	Lecturers' preferences	
s_1: $(p_1\ p_7)$	l_1: $(s_7\ s_4)\ s_1\ s_3\ (s_2\ s_5)\ s_6$	l_1 offers p_1, p_2, p_3
s_2: $p_1\ p_3\ p_5$	l_2: $s_3\ s_2\ s_7\ s_5$	l_2 offers p_4, p_5, p_6
s_3: $(p_2\ p_1)\ p_4$	l_3: $(s_1\ s_7)\ s_6$	l_3 offers p_7, p_8
s_4: p_2		
s_5: $p_1\ p_4$		
s_6: $p_2\ p_8$	Project capacities	$c_1 = 2$, $c_j = 1$, $(2 \leq j \leq 8)$
s_7: $(p_5\ p_3)\ p_8$	Lecturer capacities	$d_1 = 3$, $d_2 = 2$, $d_3 = 2$

For any pair $(s_i, p_j) \in \mathcal{S} \times \mathcal{P}$, where p_j is offered by l_k, we refer (s_i, p_j) as an *acceptable pair* if s_i and p_j both find each other acceptable, i.e., p_j is ranked by a student s_i and a lecturer l_k ranks s_i. Each student $s_i \in \mathcal{S}$ has a set $A_i \subseteq \mathcal{P}$ of acceptable projects that they rank in the order of preference. We let $R_{s_i}(p_j)$ and $R_{l_k}(s_i)$ be the rank of p_i in s_i's ranks list and the rank of s_i in l_k's ranks list, respectively.

A matching $\mathcal{T}$ of an SPA-ST instance is a set of acceptable pairs (s_i, p_j) or $(s_i, \varnothing)$ such that $|\mathcal{T}(s_i)| \leq 1$ for all $s_i \in \mathcal{S}$, $|\mathcal{T}(p_j)| \leq c_j$ for all $p_j \in \mathcal{P}$, and $|\mathcal{T}(l_k)| \leq d_k$ for all $l_k \in \mathcal{L}$, meaning that each s_i belongs to at most one pair. A project p_j is *under-subscribed*, *full*, or *over-subscribed* if $|\mathcal{T}(p_j)| < c_j$, $|\mathcal{T}(p_j)| = c_j$, or $|\mathcal{T}(p_j)| > c_j$, respectively. Similarly, a lecturer l_k is *under-subscribed*, *full*, or *over-subscribed* if $|\mathcal{T}(l_k)| < d_k$, $|\mathcal{T}(l_k)| = d_k$, or $|\mathcal{T}(l_k)| > d_k$, respectively. If an acceptable project and lecturer are *under-subscribed*, this project is a *potential* project. If $(s_i, p_j) \in \mathcal{T}$, then s_i is matched to p_j, denoted by $\mathcal{T}(s_i) = p_j$. If $\mathcal{T}(s_i) = \varnothing$, then s_i is *unassigned* in $\mathcal{T}$, and we let the set of *unsigned* students be $\mathcal{U}$.

Given a matching $\mathcal{T}$, a pair $(s_i, p_j) \in \mathcal{S} \times \mathcal{P}$ is a blocking pair in $\mathcal{T}$ if it meets the conditions of (1), (2), and (3) as follows:

1. *s_i and p_j find accept each other*;
2. *s_i prefers p_j to $\mathcal{T}(s_i)$ or $\mathcal{T}(s_i) = \varnothing$*;

3. *either* (*a*), (*b*) or (*c*) *holds as follows*:
   (a) $|\mathcal{T}(p_j)| < c_j$ and $|\mathcal{T}(l_k)| < d_k$;
   (b) $|\mathcal{T}(p_j)| < c_j$, $|\mathcal{T}(l_k)| = d_k$, and *(i) either* $s_i \in \mathcal{T}(l_k)$ *or (ii)* l_k *prefers* s_i *to the worst student in* $\mathcal{T}(l_k)$;
   (c) $|\mathcal{T}(p_j)| = c_j$ *and* l_k *prefers* s_i *to the worst student in* $\mathcal{T}(p_j)$.

A matching $\mathcal{T}$ is *stable* if it admits no blocking pair. Otherwise, it is *unstable*. The size of a stable matching $\mathcal{T}$, denoted by $|\mathcal{T}|$, is the number of students assigned in $\mathcal{T}$. If $|\mathcal{T}| = n$, then $\mathcal{T}$ is a *perfect* matching. Otherwise, $\mathcal{T}$ is a *non-perfect* matching. $|\mathcal{U}|$ is the number of unassigned students in a *non-perfect* matching $\mathcal{T}$. If the size of a weakly stable matching is equal to n, denoted by $|\mathcal{T}| = n$, then $\mathcal{T}$ is a *perfect* matching. Otherwise, $\mathcal{T}$ is *non-perfect*.

3 Proposed Algorithm

In this section, we introduce our proposed algorithm. The core of our algorithm is two *heuristic functions* that can improve execution time and solution quality for the MAX-SPA-ST problem. The first one helps students to choose a suitable project for matching, while the second one determines which students will be removed from the current matching when the lecturer or project is *over-subscribed*.

3.1 Heuristic Functions

This section presents two heuristic functions to guide students and lecturers to select appropriate projects and students, respectively, so that our algorithm can reach a perfect matching with a better solution quality and shorter execution time. To do so, we design two heuristic functions as follows:

(1) The heuristic function $h(p_j)$: For each student $s_i \in \mathcal{S}$, we define the first heuristic function $h(p_j)$ for every project p_j in s_i's rank list, where p_j is offered by l_k, to choose the best project in terms of the minimum value of $h(p_j)$ in Eq. 1 as follows:

$$h(p_j) = R_{s_i}(p_j) - \min(d_k - |\mathcal{T}(l_k)|, 1)/2 - (c_j - |\mathcal{T}(p_j)|)/(2 \times c_j + 1). \quad (1)$$

If (s_i, p_j) is an *acceptable* pair, then $1 \leq R_{s_i}(p_j) \leq q$. If l_k is *full*, i.e., $d_k - |\mathcal{T}(l_k)| = 0$, then $min(d_k - |\mathcal{T}(l_k)|, 1)/2 = 0$. If l_k is *under-subscribed*, i.e., $|\mathcal{T}(l_k)| < d_k$, then $min(d_k - |\mathcal{T}(l_k)|, 1)/2 = 0.5$ for all $l_k \in \mathcal{L}$. Besides, we have $0 \leq c_j - |\mathcal{T}(p_j)| \leq c_j$, then $0 \leq (c_j - |\mathcal{T}(p_j)|)/(2 \times c_j + 1) < 0.5$ for all $p_j \in \mathcal{P}$, meaning that $0 < h(p_j) \leq q$.

(2) The heuristic function $g(s_t)$: For each student $s_i \in \mathcal{S}$, when s_i offers to p_j, if p_j is *full* or l_k is *full*, then we choose the worst student in terms of the maximum heuristic value to remove her/his from the current matching. To do this, we define a heuristic function $g(s_t)$ in Eq. 2 as follows:

$$g(s_t) = R_{l_k}(s_t) + t(s_t) + r(s_t)/(q + 1). \quad (2)$$

For each student $s_t \in \mathcal{T}(l_k)$, we have $1 \leq R_{l_k}(s_t) \leq n$. Moreover, $t(s_t) = sum(min(d_z - |\mathcal{T}(l_z)|, 1) \times min(c_u - |\mathcal{T}(p_u)|, 1) \times n)$, where p_u is the same ties with $\mathcal{T}(s_t)$ in s_t's rank list and p_u is offered by l_z. If p_u is *under-subscribed*, i.e., $|\mathcal{T}(p_u)| < c_u$, then $1 \leq c_u - |\mathcal{T}(p_u)| \leq c_u$, thus we have $min(c_u - |\mathcal{T}(p_u)|, 1) = 1$. Otherwise, if p_u is *full*, i.e., $|\mathcal{T}(p_u)| = c_u$, then $c_u - |\mathcal{T}(p_u)| = 0$, thus we have $min(c_u - |\mathcal{T}(p_u)|, 1) = 0$. Similarly, we have $min(d_z - |\mathcal{T}(l_z)|, 1) = 1$ or 0, if l_z is *under-subscribed* or *full*, respectively. If p_u is a *potential* project, then we have $min(d_z - |\mathcal{T}(l_z)|, 1) \times min(c_u - |\mathcal{T}(p_u)|, 1) \times n = n$, otherwise, $min(d_z - |\mathcal{T}(l_z)|, 1) \times min(c_u - |\mathcal{T}(p_u)|, 1) \times n = 0$. Thus, we have $0 \leq t(s_t) \leq (q-1) \times n$. Let $r(s_t)$ be the number projects ranked by s_t, we have $1 \leq r(s_t) \leq q$, thus $0 < r(s_t)/(q+1) < 1$. This means that we have $1 < g(s_t) < (q \times n + 1)$ for all $s_t \in \mathcal{T}(l_k)$.

3.2 Our Algorithm

This section presents our proposed algorithm, called HAG (Algorithm 1). At the beginning, HAG assigns a matching $\mathcal{T} = \varnothing$ and a count variable $v(s_i) = 0$ for all students $s_i \in \mathcal{S}$. At each iteration, if there exists an *unassigned* $s_i \in \mathcal{S}$ such that s_i's rank list is non-empty, then HAG runs as follows. First, it calculates the heuristic function $h(p_j)$ for each project $p_j \in \mathcal{P}$ in s_i's rank list. Then, s_i proposes a project p_j corresponding to the minimum value of $h(p_j)$. Let l_k be a lecturer who offers p_j, we consider the following cases:

1. If both p_j and l_k are *under-subscribed*, HAG adds (s_i, p_j) into $\mathcal{T}$.
2. If p_j is *full*, HAG calculates $g(s_t)$ for all $s_t \in \mathcal{T}(p_j)$ and chooses a student s_t with maximal value of $g(s_t)$ by the function $Choose\_Student(\mathcal{T}(p_j), l_k)$. If $g(s_t) > n+1$ or $R_{l_k}(s_t) > R_{l_k}(s_i)$, then HAG removes (s_t, p_j), adds (s_i, p_j) into $\mathcal{T}$ and deletes p_j in s_t's rank list if $g(s_t) < n+1$. Otherwise, HAG deletes p_j in s_i's rank list. Note that if $g(s_t) > n+1$, then s_t contains a *potential* project with the same ties as p_j in s_t's rank list.
3. If l_k is *full*, HAG calculates $g(s_w)$ for all $s_w \in \mathcal{T}(l_k)$ and chooses a student s_w with maximal value of $g(s_w)$ by the function $Choose\_Student(\mathcal{T}(l_k), l_k)$. If $g(s_w) > n+1$ or $R_{l_k}(s_w) > R_{l_k}(s_i)$, then HAG removes (s_w, p_u), where $p_u = \mathcal{T}(s_w)$, and adds (s_i, p_j) into $\mathcal{T}$. If $g(s_w) < n+1$, then HAG deletes p_u in s_w's rank list. Otherwise, it deletes p_j in s_i's rank list. When a project p_u is removed, it can form blocking pairs with students in $\mathcal{T}(l_k)$, then HAG calls the function $Repair(p_u, l_k)$ to break blocking pairs satisfying the condition of *(3bi)*.

The above process repeats until a stable matching $\mathcal{T}$ is found. If $|\mathcal{T}| = n$, then HAG returns a perfect matching. Otherwise, HAG calls the function $Escape(p_u, l_k)$ to assign *unassigned* students for the current stable matching. Our HAG stops when a *perfect* matching is found or all *unassigned* students cannot find any projects to assign them. In the latter case, the algorithm returns a stable matching of a maximum size found so far.

Algorithm 1: HAG algorithm for MAX-SPA-ST

Input: An SPA-ST instance I
Output: A stable matching $\mathcal{T}$

function HAG(I)
  $\mathcal{T} := \varnothing$;
  $v(s_i) := 0$, $\forall s_i \in \mathcal{S}$;
  while *true* **do**
    $s_i :=$ an *unassigned* student that s_i's rank list is non-empty;
    if *there exists no student* s_i **then**
      if $|\mathcal{T}| = n$ **then** break;
      else
        $\mathcal{T}' :=$ `Escape`($\mathcal{T}$);
        if $\mathcal{T}' = \mathcal{T}$ **then** break;
        $\mathcal{T} := \mathcal{T}'$;
        continue;
      end
    end
    for *each* $p_j \in A_i$ **do**
      $l_k :=$ a lecturer who offers p_j;
      $h(p_j) := R_{s_i}(p_j)$-$min(d_k$-$|\mathcal{T}(l_k)|, 1)$/2-$(c_j$-$|\mathcal{T}(p_j)|)$ /$(2 \times c_j + 1)$;
    end
    $p_j :=$ argmin$(h(p_j) > 0)$, $\forall p_j \in \mathcal{P}$;
    $l_k :=$ a lecturer who offers p_j;
    if p_j *and* l_k *are under-subscribed* **then**
      $\mathcal{T} := \mathcal{T} \cup \{(s_i, p_j)\}$;
    else if p_j *is full* **then**
      $[s_t, g(s_t)] :=$ `Choose_Student`($\mathcal{T}(p_j), l_k$);
      if $g(s_t) > n + 1$ *or* $R_{l_k}(s_i) < R_{l_k}(s_t)$ **then**
        $\mathcal{T} := \mathcal{T} \setminus \{(s_t, p_j)\} \cup \{(s_i, p_j)\}$;
        if $g(s_t) < n + 1$ **then**
          $R_{s_t}(p_j) := 0$;
        end
      else
        $R_{s_i}(p_j) := 0$;
      end
    else
      $[s_w, g(s_w)] :=$ `Choose_Student`($\mathcal{T}(l_k), l_k$);
      if $g(s_w) > n + 1$ *or* $R_{l_k}(s_i) < R_{l_k}(s_w)$ **then**
        $\mathcal{T} := \mathcal{T} \setminus \{(s_w, p_u)\} \cup \{(s_i, p_j)\}$, where $p_u = \mathcal{T}(s_w)$;
        `Repair`(p_u, l_k);
        if $g(s_w) < n + 1$ **then**
          $R_{s_w}(p_u) := 0$;
        end
      else
        $R_{s_i}(p_j) := 0$;
      end
    end
  end
  return $\mathcal{T}$;
end function

The function $Choose\_Student(\mathcal{T}(p_j), l_k)$ is used to choose a student with the maximum value of $g(s_t)$. Let X be the set of students such that $X = \mathcal{T}(p_j)$, where p_j is *full*, or $X = \mathcal{T}(l_k)$ where l_k is *full*. For each $s_t \in X$, the algorithm calculates the heuristic value of $g(s_t)$ and returns a student s_t with the maximum value of $g(s_t)$.

The function $Repair(p_u, l_k)$ is used to break blocking pairs satisfying the condition of *(3bi)*. When a project p_u is removed, for each $s_k \in \mathcal{T}(l_k)$, if $R_{s_k}(p_u) < R_{s_k}(p_z)$, where $p_z = \mathcal{T}(s_k)$, then the algorithm removes (s_k, p_z) and adds (s_k, p_u) into $\mathcal{T}$. This process repeats for each project which is removed until it cannot form blocking pairs.

If HAG reaches a *stable* matching but it is *non-perfect* matching, it gets stuck at a local minimum. At each iteration, for each student $s_i \in \mathcal{S}$, HAG proposes a project $p_j \in \mathcal{P}$ which is the minimum rank in s_i's rank list and adds (s_i, p_j) into matching $\mathcal{T}$. Since ties were given in l_k's rank list (p_j is offered by l_k), there exists a different student s_u who prefers the most project p_j or p_z (p_z is offered by l_k) where $R_{l_k}(s_u) = R_{l_k}(s_i)$. When s_u applies to p_j (p_j is *full*) or p_z (l_k is *full*), we keep (s_i, p_j) and reject (s_u, p_z) or (s_u, p_j), then HAG can result in a *non-perfect* matching. If we add (s_u, p_z) or (s_u, p_j) and remove (s_i, p_j), HAG can result in a *perfect* matching. This means that we can add (s_u, p_z) or (s_u, p_j) and remove (s_i, p_j), then s_i applies to other projects. Moreover, we can improve stable matchings' size by using the function $Escape(\mathcal{T})$. For each *unassigned* student $s_u \in \mathcal{S}$, the algorithm finds a project p_z such that there exists a student s_i where p_j is assigned to s_i and $v(s_u) \geq v(s_i)$. Then, the algorithm replaces (s_i, p_j) by (s_u, p_z) in $\mathcal{T}$ and increases the value of $v(s_u)$. When p_j is removed, it can form blocking pairs with other students in $\mathcal{T}(l_k)$, thus we call the function $Repair(p_j, l_k)$ to break blocking pairs. It should be noted that the condition $v(s_u) \geq v(s_i)$ means that the number of replacements of s_u is higher than that of s_i, i.e., s_u is prioritized to assign to $p_z \in l_k$. Accordingly, matching $\mathcal{T}$ is *unstable* and s_i is an *unassigned* student, then s_i proposes other projects in s_i's rank list.

4 Performance Evaluation

In this section, we present several experiments to evaluate the efficiency of our HAG algorithm. We compared the average execution time and solution quality found by HAG with those found by the approximation algorithm, named APX [7] for solving MAX-SPA-ST problem of large sizes. We implemented these algorithms by Matlab R2019a software on a system with Xeon-R Gold 6130 CPU 2.1 GHz computer with 16 GB RAM.

Datasets: To perform experiments, we adapted a random SMTI problem generator [9] to generate SPA-ST instances with seven parameters (n, m, q, p_1, p_2, C, D), where n is the number of students, m is the number of lecturers, q is the number of projects, p_1 is the probability of incompleteness, p_2 is the probability of ties, C is the total capacity of projects, denoted by $C = \sum_{j=1}^{q} c_j$ where c_j is the capacity of project p_j offered by lecturer l_k, D is the total capacity of lecturers, denoted by $D = \sum_{k=1}^{m} d_k$ where d_k is the capacity of each lecturer l_k.

4.1 Evaluate the Variation of Ties and Incompleteness

In this experiment, we evaluate the influence of the probability of p_1 and p_2 on the solution quality and execution time of HAG in comparison with APX. We generated SPA-ST instances by letting parameters $(n, m, q, p_1, p_2, C, D)$, in which $n \in \{100, 200\}$, $m = 0.05n$, $q = 0.1n$, $p_1 \in [0.1, 0.8]$ and $p_2 \in [0.0, 1.0]$ with step of 0.1. The total capacities of projects and lecturers are $C = 1.2n$, and $D = 1.1n$, respectively. In addition, c_j is distributed for each project $p_j \in \mathcal{P}$ such that $0.6C/q \leq c_j \leq 1.4C/q$ and d_k is set of each lecturer $l_k \in \mathcal{L}$ such that $d_k = D/m$.

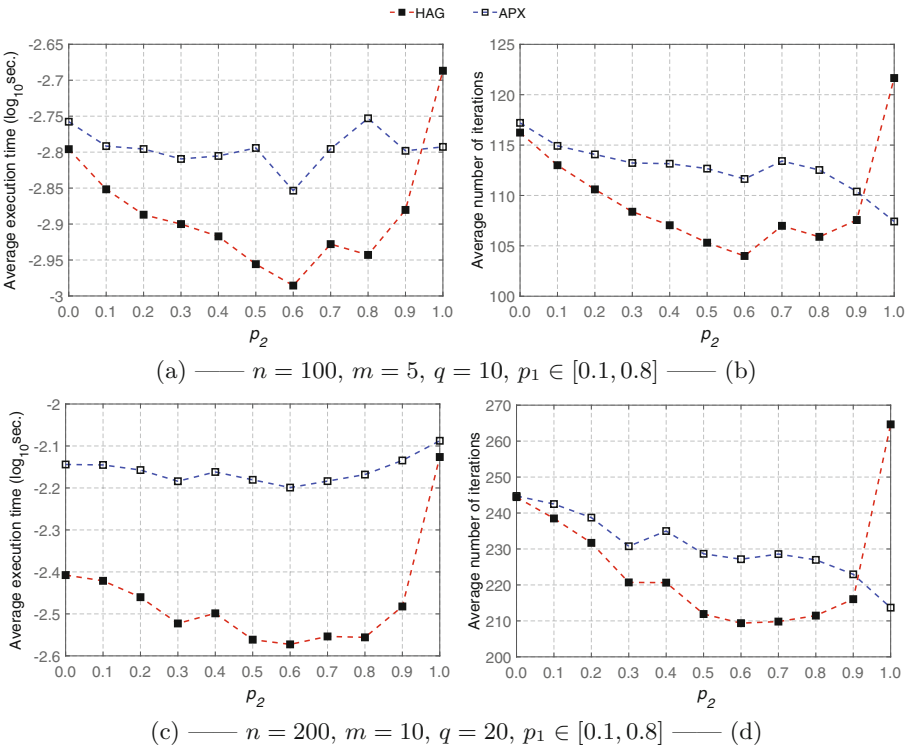

Fig. 1. Average execution time and average number of iterations of HAG vs. APX

We first compare the average execution time and the average number of iterations of HAG with APX for finding perfect matchings for $n = 100$ and $n = 200$. Then, we average results based on values of p_1 which is shown in Fig. 1. Accordingly, we see that Figs. 1(a) and 1(c) show that HAG runs faster than APX for $n = 100$ and $n = 200$ with every value of p_2. When p_2 increases from 0.0 to 0.9 for $n = 100$, HAG takes from $10^{-2.98}(s)$ to $10^{-2.68}(s)$, while APX takes from $10^{-2.85}(s)$ to $10^{-2.75}(s)$, but when $p_2 = 1.0$, the average execution

time of HAG increases. When $n = 200$, the average execution time of HAG increases from about $10^{-2.58}(s)$ to $10^{-2.28}(s)$, while APX takes from $10^{-2.23}(s)$ to $10^{-2.16}(s)$. Figures 1(b) and 1(d) give the average number of iterations used by HAG and APX for $n = 100$ and $n = 200$ with every p_2. HAG needs fewer iterations than APX for p_2 from 0.0 to 0.9 for $n = 100$ and $n = 200$. This explains that HAG runs much faster than APX as shown in Figs. 1(a) and 1(c).

Next, we compare the percentage of perfect matchings and average number of unassigned students found by HAG and APX for every value of p_1 and p_2. Our experimental results show that when p_1 varies from 0.1 to 0.5 with step 0.1, both HAG and APX result in perfect matchings approximately, therefore we only show the results in Fig. 2 for values of p_1 from 0.6 to 0.8 and every value of p_2.

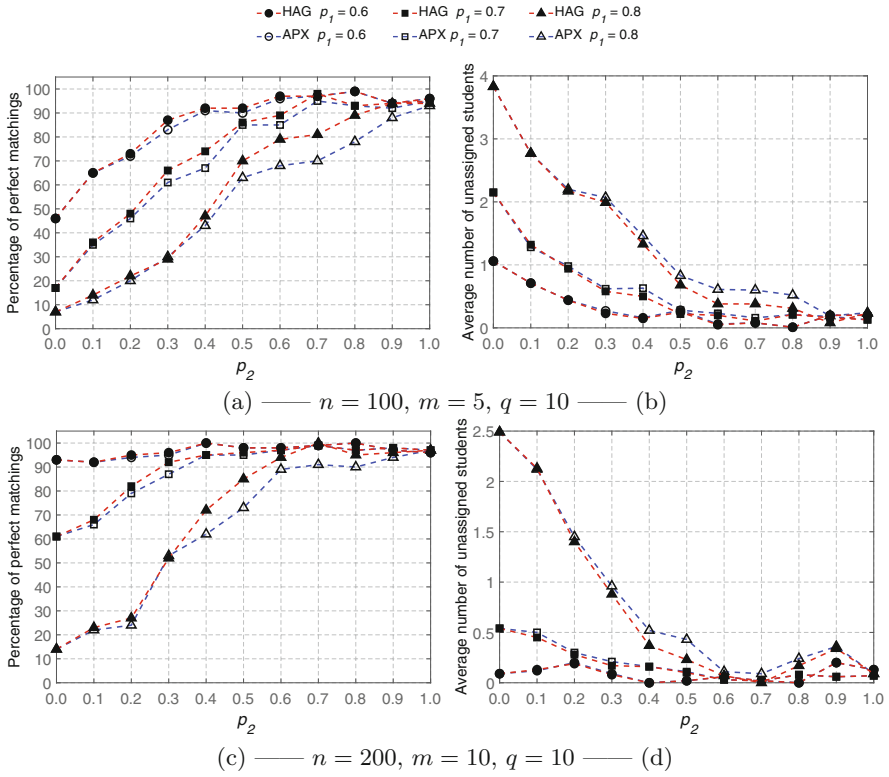

Fig. 2. Percentage of perfect matchings and number of unassigned students of HAG vs. APX

Figure 2 shows that when p_1 increases to 0.8, it is difficult for finding perfect matchings, but HAG finds better the percentage of perfect matchings than that found by APX. When p_2 varies from 0.1 to 1.0, HAG finds a much higher percentage of perfect matchings than APX as shown in Figs. 2(a) and 2(c).

Figures 2(b) and 2(d) show the average number of unassigned students found by HAG and APX. When p_2 varies from 0.1 to 1.0, HAG obtains a much smaller number of unassigned students than APX in non-perfect matchings. This means that HAG finds better maximum stable matchings than APX when p_1 increases in this experiment.

4.2 Evaluate the Variation of the Problem of Large Sizes

In this experiment, we set $n \in [1000, 10000]$ with step 1000, $m \in [0.02n, 0.04n]$ step 0.01, $q = 0.1n$, $p_1 = 0.9$, and $p_2 = 0.5$. The total capacity of projects and lecturers is set in two cases as follows:

Case 1: $C = 1.4n$ and $0.7C \leq D \leq 0.9C$, i.e., the capacity c_j of each project p_j is bound by $0.6C/q \leq c_j \leq 1.4C/q$ and the capacity d_k of each lecturer l_k is bound by $0.7\sum_{k=1}^{|P_k|} c_j \leq d_k \leq 0.9\sum_{k=1}^{|P_k|} c_j$, where P_k is a set of projects p_j offered by lecturer l_k.

Case 2: $C = 1.5n$, and $D = 1.2n$, i.e., we distribute C to the capacity c_j of each project p_j such that $0.6C/q \leq c_j \leq 1.4C/q$. Next, we set the capacity of each lecturer l_k to be $d_k = D/m$.

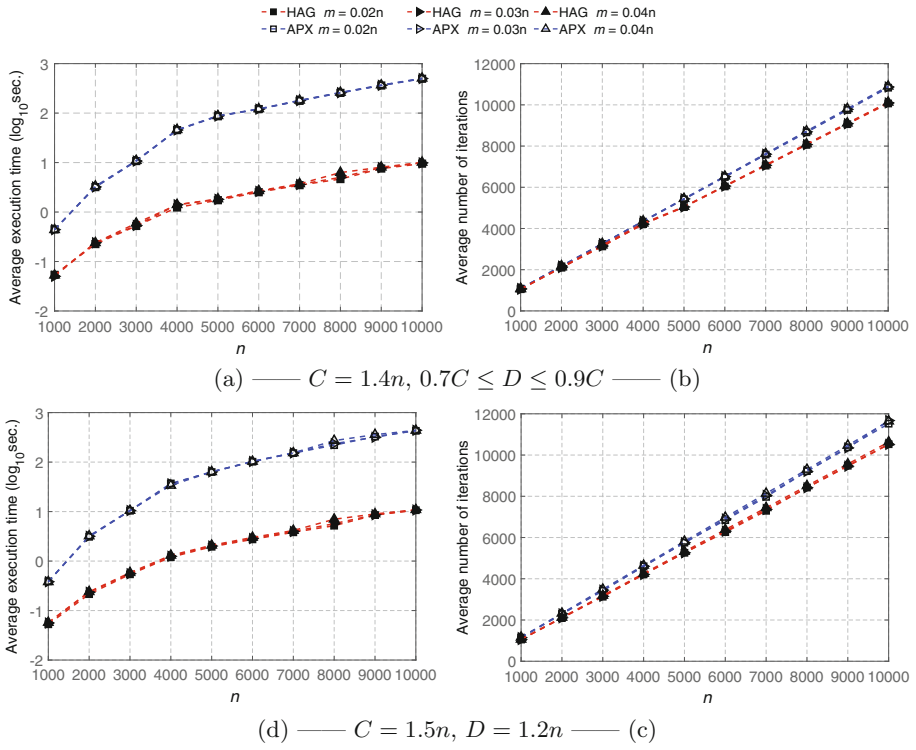

Fig. 3. Average execution time and average number of iterations HAG vs. APX

When $p_1 = 0.9$, 90% of projects or students do not rank in students' rank lists or lecturers' rank lists, respectively. However, because the number of projects and students is larger, both HAG and APX can find approximately 100% of perfect matchings. Figure 3 shows that HAG runs faster from 10 to 80 times and needs fewer iterations than APX in two cases of capacities.

5 Conclusions

This paper proposed an efficient heuristic algorithm HAG to solve the large size MAX-SPA-ST problem. Our algorithm starts from an empty matching and finds a solution for the problem based on two proposed *heuristic functions* to improve the performance of the searching process. If the algorithm reaches a *non-perfect* matching, we propose a heuristics strategy to increase the size of the stable matching by suggesting *unassigned* students to projects in the same ties with its partner in the current matching. Our experimental results show that our algorithm overcomes the APX algorithm [7] in terms of execution time and solution quality for the MAX-SPA-ST of large sizes. In the future, we will extend this proposed approach to find *strongly stable* or *super-stable* matchings for the SPA-ST problem [20].

References

1. Abraham, D.J., Irving, R.W., Manlove, D.F.: The student-project allocation problem. In: Ibaraki, T., Katoh, N., Ono, H. (eds.) ISAAC 2003. LNCS, vol. 2906, pp. 474–484. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-24587-2_49
2. Abraham, D.J., Irving, R.W., Manlove, D.F.: Two algorithms for the student-project allocation problem. J. Discrete Algorithms **5**(1), 73–90 (2007)
3. Aderanti, F.A., Amosa, R., Oluwatobiloba, A.: Development of student project allocation system using matching algorithm. In: International Conference Science Engineering Environmental Technology, vol. 1 (2016)
4. Binong, J.: Solving student project allocation with preference through weights. In: Bhattacharjee, D., Kole, D.K., Dey, N., Basu, S., Plewczynski, D. (eds.) Proceedings of International Conference on Frontiers in Computing and Systems. AISC, vol. 1255, pp. 423–430. Springer, Singapore (2021). https://doi.org/10.1007/978-981-15-7834-2_40
5. Calvo-Serrano, R., Guillén-Gosálbez, G., Kohn, S., Masters, A.: Mathematical programming approach for optimally allocating students' projects to academics in large cohorts. Educ. Chem. Eng. **20**, 11–21 (2017)
6. Chiarandini, M., Fagerberg, R., Gualandi, S.: Handling preferences in student-project allocation. Ann. Oper. Res. **275**(1), 39–78 (2019). https://doi.org/10.1007/s10479-017-2710-1
7. Cooper, F., Manlove, D.F.: A 3/2-approximation algorithm for the student-project allocation problem. In: 17th International Symposium on Experimental Algorithms, SEA 2018, 27–29 June 2018, L'Aquila, Italy, vol. 103, pp. 8:1–8:13 (2018). https://doi.org/10.4230/LIPIcs.SEA.2018.8

8. Gani, M.A., Hamid, R.A., et al.: Optimum allocation of graduation projects: Survey and proposed solution. J. Al-Qadisiyah Comput. Sci. Math. **13**(1), 58 (2021)
9. Gent, I.P., Prosser, P.: An empirical study of the stable marriage problem with ties and incomplete lists. In: Proceedings of the 15th European Conference on Artificial Intelligence, pp. 141–145. Lyon, France (2002)
10. Harper, P.R., de Senna, V., Vieira, I.T., Shahani, A.K.: A genetic algorithm for the project assignment problem. Comput. Oper. Res. **32**(5), 1255–1265 (2005)
11. Irving, R.W., Manlove, D.F., Scott, S.: The hospitals/residents problem with ties. In: SWAT 2000. LNCS, vol. 1851, pp. 259–271. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44985-X_24
12. Iwama, K., Miyazaki, S., Yanagisawa, H.: Improved approximation bounds for the student-project allocation problem with preferences over projects. J. Discrete Algorithms **13**, 59–66 (2012)
13. Király, Z.: Linear time local approximation algorithm for maximum stable marriage. Algorithms **6**(1), 471–484 (2013)
14. Kwanashie, A., Irving, R.W., Manlove, D.F., Sng, C.T.S.: Profile-based optimal matchings in the student/project allocation problem. In: Kratochvíl, J., Miller, M., Froncek, D. (eds.) IWOCA 2014. LNCS, vol. 8986, pp. 213–225. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19315-1_19
15. Manlove, D., Milne, D., Olaosebikan, S.: An integer programming approach to the student-project allocation problem with preferences over projects. In: Proceedings of 5th International Symposium on Combinatorial Optimization, pp. 213–225. Morocco (2018)
16. Manlove, D., Milne, D., Olaosebikan, S.: An integer programming approach to the student-project allocation problem with preferences over projects. In: Lee, J., Rinaldi, G., Mahjoub, A.R. (eds.) ISCO 2018. LNCS, vol. 10856, pp. 313–325. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96151-4_27
17. Manlove, D.F.: Algorithmics of Matching Under Preferences, Series on Theoretical Computer Science, vol. 2 (2013). https://doi.org/10.1142/8591
18. Manlove, D.F., O'Malley, G.: Student-project allocation with preferences over projects. J. Discrete Algorithms **6**(4), 553–560 (2008)
19. Olaosebikan, S., Manlove, D.: An algorithm for strong stability in the student-project allocation problem with ties. In: Changat, M., Das, S. (eds.) CALDAM 2020. LNCS, vol. 12016, pp. 384–399. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39219-2_31
20. Olaosebikan, S., Manlove, D.: Super-stability in the student-project allocation problem with ties. J. Comb. Optim. **43**, 1–37 (2020). https://doi.org/10.1007/s10878-020-00632-x
21. Paunović, V., Tomić, S., Bosnić, I., Žagar, M.: Fuzzy approach to student-project allocation (spa) problem. IEEE Access **7**, 136046–136061 (2019)
22. Irving, R.W., Manlove, D.F.: Finding large stable matchings. J. Exp. Algorithmics **14**(1), 1–2 (2009)