CS 254 Project Report

Made by- Aryan Verma (180001008)
Srijan Saini (180001056)

Student Project Allocation Problem: Three Sided Matching System

Under the guidance of

Dr. Kapil Ahuja

Department of Computer Science and Engineering

Indian Institute of Technology, Indore

Spring 2020

Contents:

Introduction:

In universities, many students tend to take projects under different professors along with their degree programmes. Usually, there is a wide range of projects available to be offered, and similarly, there are a lot of students eager to take those. Each lecturer usually has more than one project under him/her, although it is not expected that all of them will be taken up.

Each student selects some projects that he/she finds interesting and prepares a list of them from most to least preferred. Similarly, each lecturer has some preferences of students whom he/she is willing to supervise in their projects. They also prepare a list of their most to least preferred students. The reasons for this may vary but most influential of them are generally students' academic performance, their research aptitude and their hard work and dedication. This assignment of students to projects keeping in mind the preference lists and capacity constraints is called the *Student-Project Allocation (SPA) Problem.*

SPA is an example of a three-sided matching system with added constraints. Three-sided matching system is an extension of the classical two-sided matching system used in mathematics, economics and computer science, the most common real-life use case of which is the stable marriage problem proposed by David Gale and Lloyd Shapley in 1962. Other examples include The National Resident Matching Problem (NRMP) in the US, allocating pupils to secondary schools in Singapore, Hospitals/Residents Problem (HR), assigning users to servers in a large distributed internet service, stable roommates problem.

The Stable Marriage problem involves sets of men and women where each person ranks members of the opposite gender in order of their preference to marry them. After a matching, if there are two people of opposite sex who would both rather have each other than their current partners, then that pair is called a blocking pair. If no such pair exists, the arrangement is called *stable matching*.

Historical evidence indicates that participants of two-sided matching problems should not be allowed to construct a matching on their own by directly approaching each other and making ad hoc arrangements. So, an automated allocation system should be constructed by means of a centralized matching scheme which can provide stability.

Stable three sided matchings do not always exist as proved by A. Alkan. But some restricted preference orders may result in stable matchings in some cases. Ours is one of such cases where stable matchings exist in two different scenarios under some constraints despite being a three-sided system. The two scenarios being the student-oriented case and lecturer-oriented case.

Objective:

- The scope of the project includes two optimal algorithms for allocating students to projects subject to the preference and capacity constraints. We aim to analyse and design the two algorithms for finding a stable matching, given an instance of SPA.
    - The student-oriented algorithm for SPA (SPA-Student Oriented) aims to provide the best possible matching for each student based on their preferences of the projects.
    - The lecturer-oriented algorithm for SPA (SPA-Lecturer Oriented) aims to provide the best possible matching for each lecturer based on their preferences of the students.
- We also aim to implement both of them using vectors and determine their time complexities.
- Further, we try to improvise our implementation using Hash Tables/Maps, and
- Analyse the design and time complexity of the optimized algorithms.

Algorithms Analysis:

Important Terms:

- ***S*** - set of students $\{s_1, s_2, \dots, s_n\}$
- **P** - set of projects $\{p_1, p_2, \dots, p_m\}$
- **L** - set of lecturers $\{l_1, l_2, \dots, l_q\}$
- $\mathbf{A}_i$ - set of projects that s_i finds acceptable
- $\mathbf{P}_k$ - set of projects offered by l_k
- $\mathbf{B}_k$ - set of students who find acceptable a project offered by l_k
- $\mathbf{L}_k$ - preference list of l_k ranking B_k in strict order.
- $\mathbf{L}_k^j$ - preference list L_k for a project $p_u \in P_k$
- $\mathbf{d}_k$ - maximum number of students for a lecturer l_k
- $\mathbf{c}_j$ - maximum number of students to be assigned to project p_j

1. SPA-Student Oriented

Pseudo Code:

SPA-student (I) {
    assign each student to be free;
    assign each project and lecturer to be totally unsubscribed;
    while (some student s_i is free **and** s_i has a non-empty list) {
        p_j = first project on s_i's list ;
        l_k = lecturer who offers p_j ;
        /* s_i applies to p_j */
        provisionally assign s_i to p_j ; /* and to l_k */
        if (p_j is over-subscribed) {
            s_r = worst student assigned to p_j ; /* according to L_k^j */
            break provisional assignment between s_r and p_j ;
        }
        else if (l_k is over-subscribed) {
            s_r = worst student assigned to l_k ;
            p_t = project assigned to s_r ;
            break provisional assignment between s_r and p_t ;
        }
        if (p_j is full) {
            s_r = worst student assigned to p_j ; /* according to L_k^j */
            for (each successor s_t of s_r on L_k^j)
                delete (s_t , p_j) ;
        }
        if (l_k is full) {
            s_r = worst student assigned to l_k ;
            for (each successor s_t of s_r on L_k)
                for (each project $p_u \in P_k \cap A_t$)
                    delete (s_t , p_u) ;
        }
    }
    return { (s_i , p_j) $\in S \times P$: s_i is provisionally assigned to p_j } ;
}

Description of the Algorithm:

1. The algorithm starts with assigning each student and lecturer to be **free**.
2. We iterate through the list of students and check if any student is **unassigned** and still has a project in their preference list.
3. For a student that satisfies this, we assign him the **first** project in his list. And, also assign the student to the lecturer associated with that project.
4. After that, we check for two conditions:
    a. If the project assigned to him has **more** students than its capacity, we select the **worst** student assigned to that project according to the preference list of that project's lecturer and **delete** the assignment of that student to the given project. Which means, we delete that project from that student's preference list.
    b. Otherwise, if the lecturer for that project has **more** students than his capacity, we select the **worst** student assigned to him according to his preference list, **delete** the assignment of that student to the lecturer and also delete that project from the student's preference list.
5. Following this, we check for the project if the total number of students assigned to it is **equal** to its capacity. If they are equal, we consider the **worst** student assigned to that project according to its lecturer's preference list. For **all students after that student** in the preference list of the lecturer for that particular project (L_k^j), we **delete** that project from their preference lists and also their names from L_k^j. <u>This results in all future assignments of students to that project to be better than at least one student already assigned to it.</u>
6. Then, we check if the total number of students assigned to the lecturer is **equal** to his capacity. If they are equal, we consider the **worst** student assigned to

him according to his preference list. For **all the students after that student** in his preference list, we **delete** all the projects associated with that lecturer from the students' preference lists. We also **delete** those students from the lecturer's preference list. This results in all future assignments of students to that lecturer to be better than at least one student already assigned to him.

7. This completes **one iteration** of the loop.
8. The algorithm **ends** with assigning all possible matchings of students to projects.

2. SPA-Lecturer Oriented

Pseudo Code:

SPA-Lecturer(l){

assign each student, project and lecturer to be free;

while(some lecturer l_k is under-subscribed **and**
there is some (student,project) pair (s_i, p_j) where s_i is not provisionally assigned to p_j **and** $p_j \in P_k$ is under subscribed **and** $s_i \in L_k^j$)
{
s_i = first such student on l_k's list;
p_j = first such project on s_i's list;

if(s_i is provisionally assigned to some project p)
{
break the provisional assignment between s_i and p;
/* l_k offers p_j to s_i */
provisionally assign s_i to p_j; /* and to l_k */
for each successor p to p_j on s_i's list
delete(s_i, p);
}
}

return $\{(s_i, p_j) \in S \times P : s_i$ provisionally assigned to $p_j\}$;

}

Description of the Algorithm:

1.Initially, all Lecturers and projects are **unsubscribed** and all the students are **unallocated**.

2.As the Algorithm is **Lecturer-Oriented** , so we'll start our iteration from the lecturers set so that each lecturer will get the best possible set of students.

3. As soon as we get the pair of Student and Project (s_i,p_i) compatible with Lecturer we'll add the Pair into the answer list.

1. The above Allocation must be **stable** and shouldn't arise any conflict , so to prevent any such kind of situation we'll check some constraints.
2. For instance
    1. If a student (s_i) is free yet and the respective project is in the list of the student and the **project capacity(c)** and the **Lecturer capacity(d)** also allow this allocation, then we can add that pair in our list.
    2. If a student (s_i) is already assigned to some other lecturer (l_k) but he/she finds a better project (p_l to p_i) then the allocated one in the list, it will be **modified (or deleted first and then added in answer list)** by the current lecturer and project name (if capacities allow it).

4. After the modification of the already existing pair, we have to look after the lecturer list for any other matching that **can exist** with the project(p_j).

5. To get any matching optimally, we'll start iterating through the students (s_i) in the list of the lecturer (l_k) and if any matching exists that we can add it to the list provided, all of the above steps for a valid matching must satisfy .

6. At last, we'll return the answer list.

Algorithms Design and Implementation:

Both our algorithms are **greedy** in design. In greedy algorithms, the solution is built in small parts with a decision taken at each step to optimize an underlying criterion and obtain the best possible answer of each small part which constructs the overall optimal answer.

In our algorithms, the students and lecturers have their preference lists ready from most preferred to least preferred projects for each student, and most preferred to least preferred students for each lecturer. We iterate through these lists in that order only which results in an overall optimal solution at each step.

Also, when the project or lecturer capacity is reached, we delete all items in the corresponding list which are less preferred than the worst assigned one in that list. This ensures that any further assignment resulting in increase in the count of the list making it greater than its capacity is possible only when it is replacing some item less preferred than this. This results in a more optimal answer each time a replacement of this sort takes place.

Hence, we can say that our algorithms are greedy in design.

Complexity Analysis:

1. SPA-Student Oriented

Let ***u*** be the upper bound of the number of projects that each student can include in his preference list.

s = total number of students

p = total number of projects

l = total number of lecturers

u number of iterations that take place for the outermost loop. For each iteration of this loop, the students' list is iterated.

Let O (f_1) be the time complexity (T.C.) of the if-else condition (if p_j is over-subscribed … else if l_k is over-subscribed). O (f_1) is (O ($f_{1.1}$) + O ($f_{1.2}$)) where O ($f_{1.1}$) is the T.C. of 'if' condition and O ($f_{1.2}$) is the T.C. of 'else if' condition.

Let O (f_2) be the T.C. of the condition 'if p_j is full' and O (f_2) be the T.C. of the condition 'if l_k is full'.

So, overall T.C. = O (***u.s.*** (O (f_1) + O (f_2) + O (f_3)))

= O (***u.s.*** (O ($f_{1.1}$) + O ($f_{1.2}$) + O (f_2) + O (f_3)))

= O (***u.s.*** (O (***s***) + O (***s***) + O (***s.p***) + O (***s***^{2}***.p***)))

= O (***u.s.*** (O (***s***^{2}***.p***)))

= O (***u.s***^{3}***.p***)

So, the **time complexity** of SPA-Student Oriented Algorithm is O (***u.s***^{3}***.p***).

The additional space used in this algorithm is to construct the L_k^j list. It is a 2-dimensional array consisting of a list of projects. And for each project, the students who find it acceptable are listed. So, this array takes O (***s.p***) space.

Hence, the **space complexity** of SPA-Student Oriented Algorithm is O (***s.p***).

2. SPA-Lecturer Oriented

Some notations :

l = Total Number of Lecturers

s = Total Number of students

a = Average number of projects that students have

b = Average Number of Projects Lecturers can offer

m = max number of matchings possible

Let **O(f_1)** be the T.C of the first three loops that will run to iterate over the lecturer list of students and along with that the list of projects students have.

Let **O(f_2)** be the T.C of finding the first common project in the lecturer's project list and the respective student list

Let **O(f_3)** be the T.C of project allocation to free student

Let **O(f_4)** be the T.C of the project allocation to already assigned student i.e deletion

Let **O(f_5)** be the T.C of the checking of a new pair that might exist due to deletion or modification.

Overall Time Complexity:

$=(O(f_1).O(f_2).(O(f_3)+O(f_4).O(f_5)))$

$=O((l.s.sp).(b).(O(1) + m.s.s.a))$

$=\mathbf{O(l.s^3.a^2.b.m)}$

So, the T.C of the SPA-Lecturer Oriented Algorithm is $\mathbf{O(l.s^3.a^2.b.m)}$

The additional space used in this algorithm is to construct the 'visited' vector and the 'M' vector to store the answer. 'visited' vector has the size of the number of students **O (*s*)** and stores a bool value. 'M' vector also has the size of the number of students and stores three integer values. So, it occupies 3.s space. Hence, overall **space complexity** is O (3.*s*) or **O (*s*)**.

Optimisation using Hash Tables/Maps:

1. SPA-Student Oriented

As we saw in complexity analysis, the term which is a major contributor to the overall time complexity is O (f_3). The 'if l_k is full' condition which is responsible for this term involves delete operation. The complexity of this operation incorporates the T.C. of a search operation which takes O (***s***) time. But this complexity can be reduced to O (1) time using hashing.

2. SPA-Lecturer Oriented

As we know the searching ,insertion and deletion reduces down to O(1) when we use Hash Maps and this method is very much useful in this algorithm also. The comparison time of projects is reduced down to O(1) and the searching time of assigned projects in the answer list is also reduced down to O(1) .

Implementation of Optimised Algorithms:

1. SPA-Student Oriented (Optimised)

The delete operation involved iterating through the L_k^j list. So, we made two Hash Tables for this list. One to check if a student is present in a project's list, and another to determine the position of a student in a project's list.

```
224       // Optimised version - using hashing table to reduce time taken by delete operation
225       vector<vector<int> > hash_stud_pref(p_no+5, vector<int> (s_no+5,0));  // hashing tab
226       vector<vector<int> > hash_stud_pref_check(p_no+5, vector<int> (s_no+5,0));  //hashin
227       // for(int i=0;i<p_no;i++){
228       //  for(int j=0;j<s_no;j++){
229       //      hash_stud_pref[i][j][1]=0;
230       //  }
231       // }
232       for(int i=0;i<p_no;i++){
233           for(int j=0;j<s_no;j++){
234               hash_stud_pref_check[i][student_preference[i][j]]=1;  // if that student is
235               hash_stud_pref[i][student_preference[i][j]]=j;  // storing student_preferenc
236           }
237       }
```

The delete operation which had 3 nested loops looked like this earlier -

```
//Deleting (st,pu) pairs for each successor st of sr in lecturer list such that pu belongs to all projects under lk which st finds acceptable
for(int j=ind_sr+1; j < lecturer[lk].size(); j++){
    for(int k=0; k < student[lecturer[lk][j]].size(); k++){
        if(lecturer_project[student[lecturer[lk][j]][k]] == lk){
            for(int l=0; l < student_preference[student[lecturer[lk][j]][k]].size(); l++){
                if(student_preference[student[lecturer[lk][j]][k]][l] == lecturer[lk][j]){
                    student_preference[student[lecturer[lk][j]][k]].erase(student_preference[student[lecturer[lk][j]][k]].begin() + l);
                    break;
                }
            }
            student[lecturer[lk][j]].erase(student[lecturer[lk][j]].begin() + k);
            k--;
        }
    }
    lecturer[lk].erase(lecturer[lk].begin() + ind_sr + 1);
}
```

After using Hash Tables, the number of loops reduce to 2 -

```
//Deleting (st,pu) pairs for each successor st of sr in lecturer list such that pu belongs to all projects under lk which st finds acceptable
for(int j=ind_sr+1; j < lecturer[lk].size(); j++){
    for(int k=0; k < student[lecturer[lk][j]].size(); k++){
        if(lecturer_project[student[lecturer[lk][j]][k]] == lk){
            if(hash_stud_pref_check[student[lecturer[lk][j]][k]][lecturer[lk][j]] == 1){
                hash_stud_pref_check[student[lecturer[lk][j]][k]][lecturer[lk][j]] == 0;
                int l = hash_stud_pref[student[lecturer[lk][j]][k]][lecturer[lk][j]];
                student_preference[student[lecturer[lk][j]][k]].erase(student_preference[student[lecturer[lk][j]][k]].begin() + l);
            }
            student[lecturer[lk][j]].erase(student[lecturer[lk][j]].begin() + k);
            k--;
        }
    }
    lecturer[lk].erase(lecturer[lk].begin() + ind_sr + 1);
}
```

2. SPA-Lecturer Oriented (Optimised)

```
for(int i=0;i<L.size();i++)          //For Lecturer
{
    for(int j=0;j<L[i].lp.size();j++)     // loop for students list
    {
        if(L[i].d>0)
        {
            flag=0;

            for(int k=0;k<L[i].lp[j].sp.size();k++)      //loop for particular student project preference list
            {
                for(int l=0;l<L[i].lps.size();l++)        // set of lecturer project
                {
```

As we can see we were using four loops for our desired result without any optimization.

After Optimization:

The same work can be done in three loops as shown in the code below.

```
for(int i=0;i<L.size();i++)          //For Lecturer
{
    for(int j=0;j<L[i].lp.size();j++)     // loop for students list
    {
        if(L[i].d>0)
        {
            for(int k=0;k<L[i].lp[j].sp.size();k++)      //loop for particular student project prefe
            {
                if(L[i].lps.find(L[i].lp[j].sp[k])!=L[i].lps.end() && P[L[i].lp[j].sp[k]-1].c>0)
                {
```

In addition to this, the searching of the assigned project in the answer list takes O(1) time now which was done in O(m) in the worst case scenario earlier.

Design and Complexity Analysis of Optimised Algorithms:

1. SPA-Student Oriented (Optimised)

The algorithm design still remains fundamentally greedy as the approach to solve the problem is not changed, just a slight modification is being done in the delete operation.

Time Complexity -

overall T.C. = O (***u.s.*** (O (f_1) + O (f_2) + O (f_3)))

= O (***u.s.*** (O ($f_{1.1}$) + O ($f_{1.2}$) + O (f_2) + O (f_3)))

= O (***u.s.*** (O (***s***) + O (***s***) + O (***s.p***) + O (***s.p***)))

= O (***u.s.*** (O (***s.p***)))

= O (***u.s***^{2}***.p***)

Which is an **improvement** to the previous case when T.C. was O (***u.s***^{3}***.p***).

Space Complexity -

As the Hash Tables have the same dimensions as L_k^j list (***s.p***), the space complexity remains the same, i.e. O (***s.p***).

2. SPA-Lecturer Oriented (Optimised)

The algorithm design still remains fundamentally greedy as the approach to the problem is not changed, just a slight modification is being done. After performing the technique of Hashing in the algorithm, our T.C reduces by a significant amount.

Time Complexity -

Overall Time Complexity = $(O(f_1).O(f_2).(O(f_3)+O(f_4).O(f_5)))$

$= O((l.s.sp).O(1).(O(1) + O(1).s.s.sp))$

$= \mathbf{O(l.s^3.a^2)}$

So, the T.C of the optimised version of SPA-Lecturer Oriented Algorithm is **$O(l.s^3.a^2)$**, which is a clear improvement over the initial method's T.C., i.e., **$O(l.s^3.a^2.b.m)$** .

As the hash map has the same dimension as the 'M' vector present in unoptimised algorithm, the space complexity of the algorithm does not change. Hence, the **space complexity** of the algorithm is **O (*s*)** .

Results and Conclusions:

1. SPA-Student Oriented

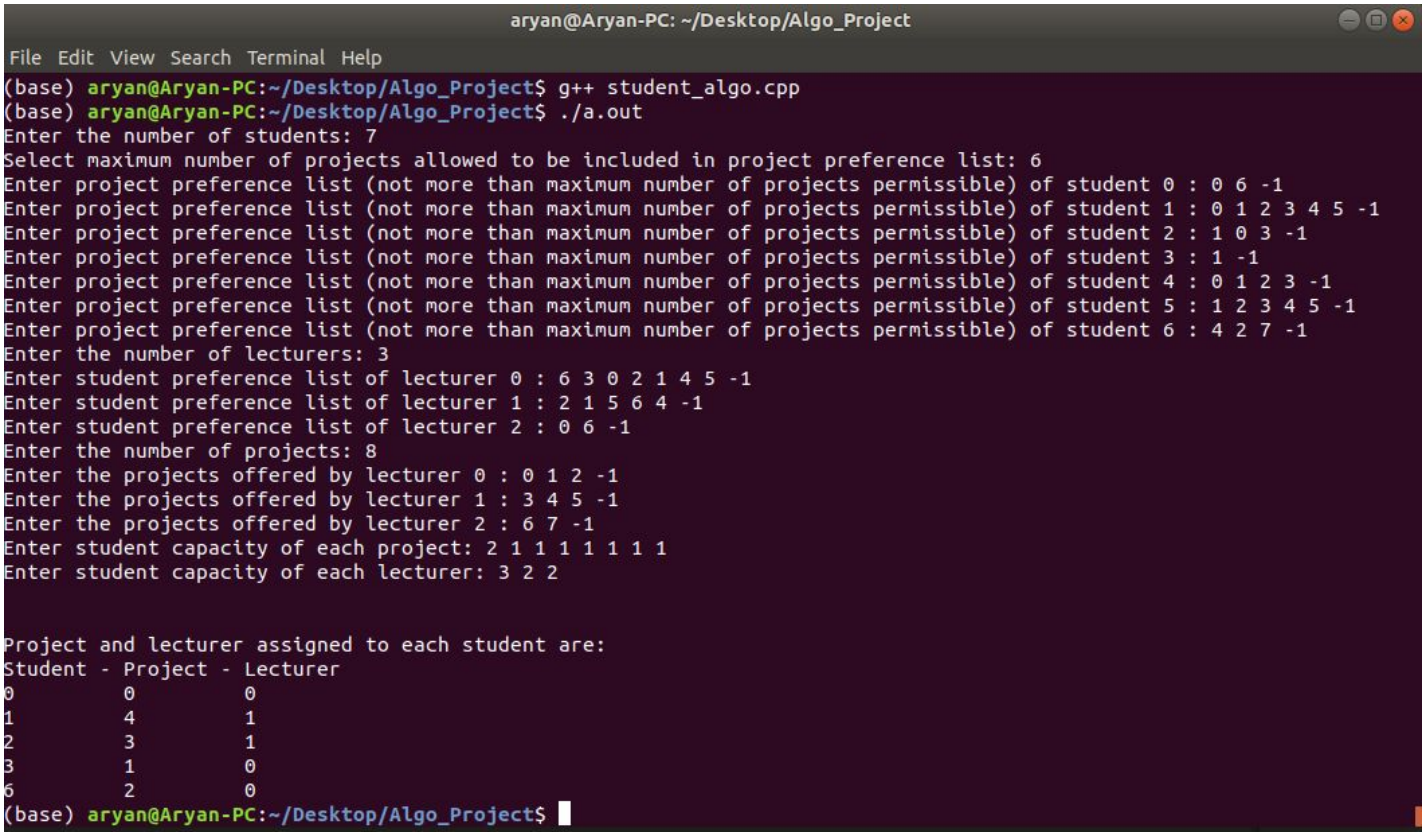
The Algorithm was tested for the following test case -

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

The output (as mentioned in the research paper) should be -

SNo	PNo	LNo
1	1	1
2	5	2
3	4	2
4	2	1
7	3	1

The output we got is -



Which is exactly equal to the given result. Note here that our implementation of all lists is zero-indexed which means Student 0 in our implementation implies Student 1 actually and subsequently for all lecturers, projects and other students. So,

Result in Paper				Our Result		
SNo -	PNo -	LNo		SNo -	PNo -	LNo
1	1	1		0	0	0
2	5	2	==	1	4	1
3	4	2		2	3	1
4	2	1		3	1	0
7	3	1		6	2	0

2. SPA-Lecturer Oriented

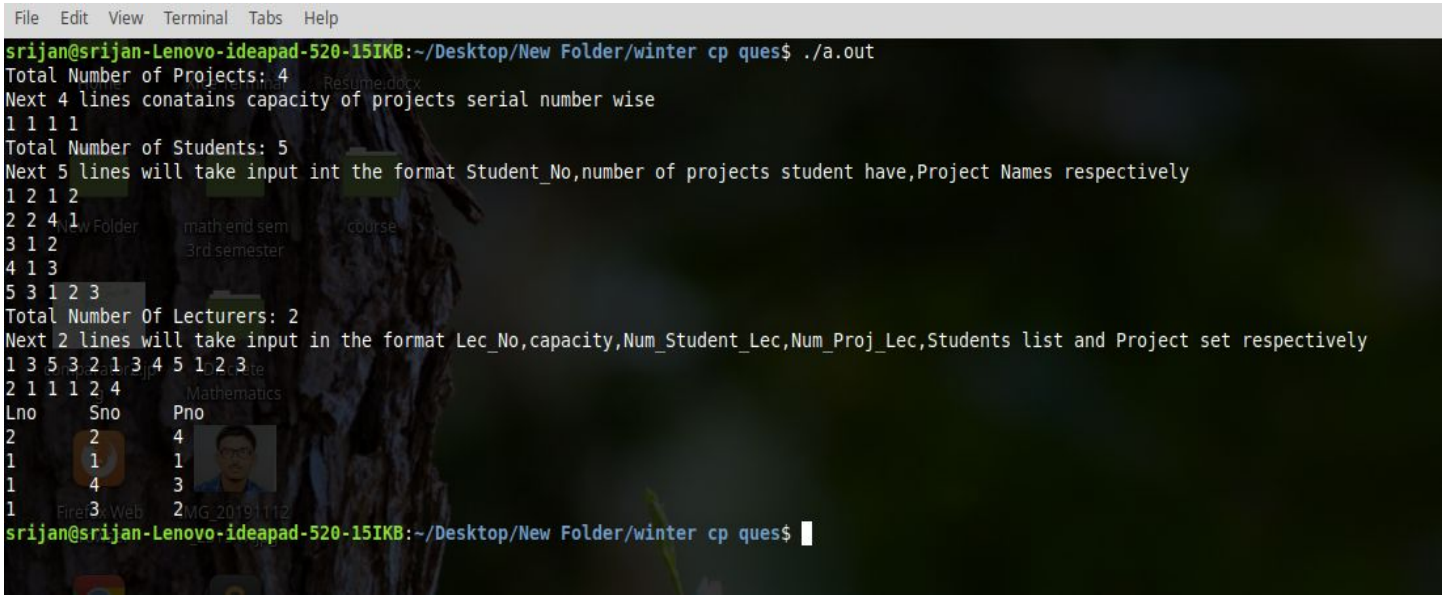
The algorithm was tested for the following test case -

Student preferences	Lecturer preferences	
S1 : p1 p2	L1 : S2 S1 S3 S4 S5	L1 offers p1,p2,p3
S2 : p4 p1	L2 : S2	L2 offers p4
S3 : p2		
S4 : p3	Project capacities: ci =1(1<=i<=4)	
S5 : p1 p2 p3	Lecturer capacities: d1=3, d2=1	

ANSWER:

LNo	SNo	PNo
1	1	1
2	2	4
1	4	3
1	3	2

The output we got is -

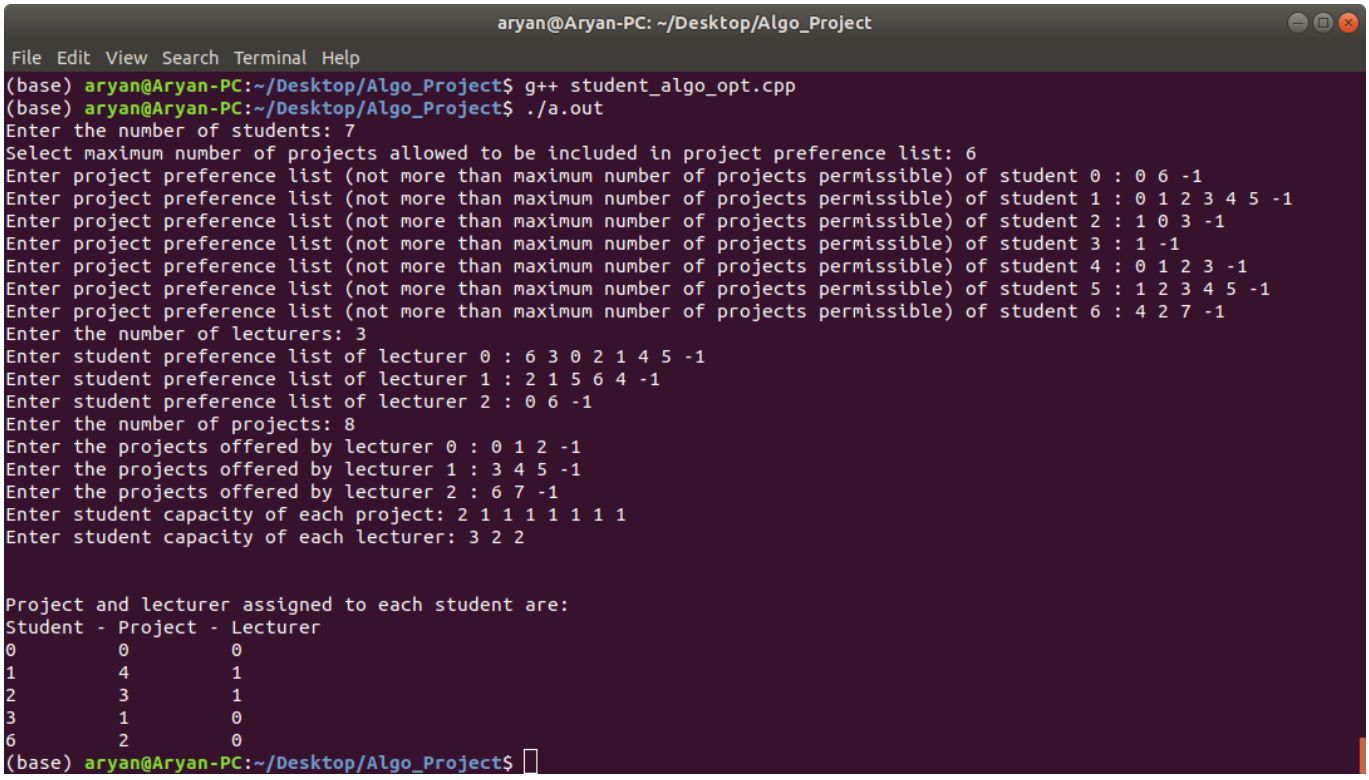


Which is exactly equal to the given result.

3. SPA-Student Oriented (Optimised)

For input, we took the same test case as for the SPA-Student Oriented one.

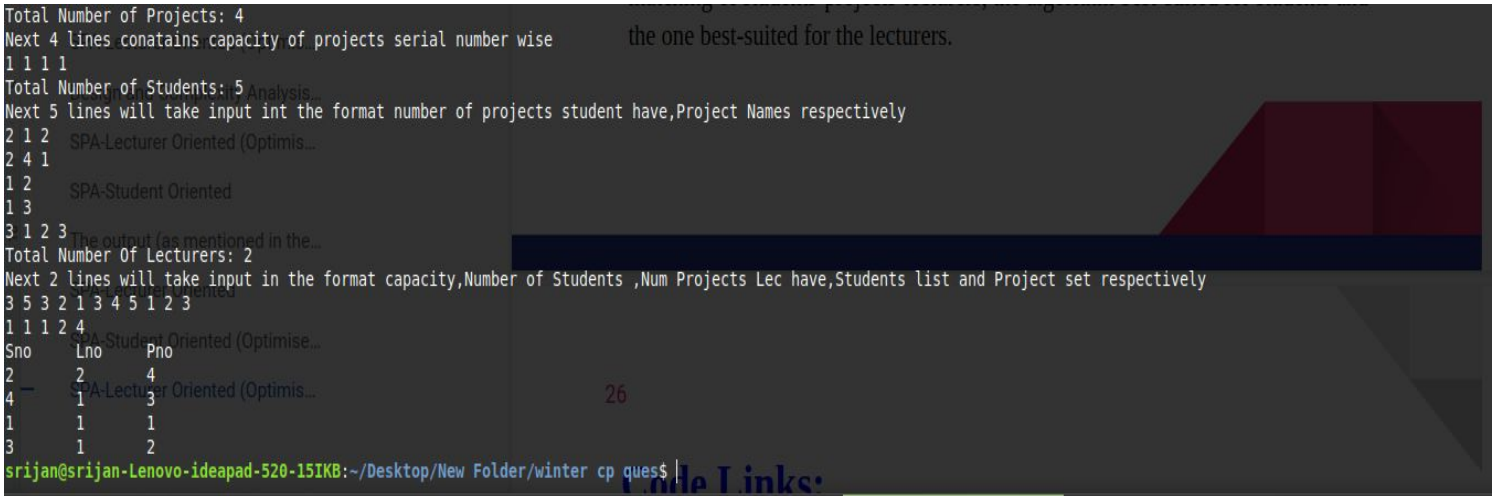
The output received is -



Which is the same as the above one.

4. SPA-Lecturer Oriented (Optimised)

For the same input as initial we got the exact same answer-



Hence, we conclude that we are able to implement two algorithms for stable matching of students-projects-lecturers, the algorithm best-suited for students and the one best-suited for the lecturers.

Code Links:

Here is the link to our GitHub Repository -

https://github.com/Aryan-Verma/Student-Project-Allocation-Problem

The SPA-Student Oriented Algorithm -

SPA-Student Oriented

The SPA-Lecturer Oriented Algorithm -

SPA-Lecturer Oriented

The SPA-Student Oriented (Optimised) Algorithm -

SPA- Student Oriented (Optimised)

The SPA-Lecturer Oriented (Optimised) Algorithm -

SPA-Lecturer Oriented(optimised)

References:

- D.J. Abraham, R.W. Irving, D.F. Manlove
  The Student-Project Allocation problem Proceedings of ISAAC 2003: the 14th Annual International Symposium on Algorithms and Computation
  Lecture Notes in Computer Science, vol. 2906, Springer-Verlag (2003), pp. 474-484
  Here is the link to this paper -
  Two algorithms for the Student-Project Allocation problem

- Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.