Chapter 3

Application of Scheduling Algorithms in College Admission Problem using Bipartite Graph

3.1 Introduction:

Suppose there are \( n \) students who are applying for colleges and there are \( k \) colleges that these students can apply for. Each student has a strict preference ordering over all colleges, and each college also has a strict preference ordering over all students. By strict preference, it is impossible for a college to accept all the students who apply for it, due to limited resources. In fact, a college only accepts a specific number of students (quotas) in each academic year. So every student cannot possibly get into their top choices. On the other hand, a student also accept offer of admission from only one college. Thus, it is not guaranteed that all students whom a college had made an offers. This problem is called college admissions problem. The college admissions problem has been widely studied by economists and game theorists. It is well-known to be closely related to the stable marriage problem. Instead of trying to find matching between students and colleges, the stable marriage problem tries to find matching between males and females in the community, such that each couple cannot be better off by pairing with other people in the community [14].

A college is considering a set of \( n \) applicants of which it can admit a quota of only \( q \). Having evaluated their qualifications, the admissions office must decide which ones to admit. The procedure of offering admission only to the \( q \) best-qualified applicants will not generally be satisfactory, for it cannot be assumed that all who are offered admission will accept.

Accordingly, in order for a college to receive \( q \) acceptances, it will generally have to offer to admit more than \( q \) applicants. The problem of determining how many and which ones to admit requires some rather involved guesswork. It may not be known (a) whether a given applicant has also applied elsewhere; if this is known it may not be known (b) how he ranks the colleges to which he has applied; even if this is known it will not be known (c) which of the other colleges will offer to admit him. A result of all this uncertainty is that colleges can expect only
that the entering class will come reasonably close in numbers to the desired quota, and be reasonably close to the attainable optimum in quality.

The usual admissions procedure presents problems for the applicants as well as the colleges. An applicant who is asked to list in his application all other colleges applied for in order of preference may feel, perhaps not without reason, that by telling a college it is, say, his third choice he will be hurting his chances of being admitted.

One elaboration is the introduction of the "waiting list," whereby an applicant can be informed that he is not admitted but may be admitted later if a vacancy occurs. This introduces new problems. Suppose an applicant is accepted by one college and placed on the waiting list of another that he prefers. Should he play safe by accepting the first or take a chance that the second will admit him later? Is it ethical to accept the first without informing the second and then withdraw his acceptance if the second later admits him?

A contend that the difficulties here described can be avoided. [5] Shall describe a procedure for assigning applicants to colleges which should be satisfactory to both groups, which removes all uncertainties and which, assuming there are enough applicants, assigns to each college precisely its quota [5].

The "college admissions problem" is the name given to a two-sided matching problem by Gale and Shapley (1962). Colleges have preferences over students and students have preference over colleges; each college $C$ can accept at most some number $q_C$ of students, and each student can enroll in at most one college. The problem is to analyze what kinds of assignments might arise from such a market, with the primary theoretical tool being the set of stable outcomes (which is closely related to, and a subset of, the core) of the resulting game, and, more recently, the dominant strategy and Nash equilibria of the corresponding strategic game. This and related models have recently been employed in both theoretical and empirical studies of labor markets [1].

The college admission problem is first studied by Gale and Shapely (1962) in a seminar paper in which they proposed the now well-known deferred-acceptance algorithm. Many variants and extensions of the original model with useful applications have been studied (Knuth (1976), Roth (1984), Gustfield and Irving
(1989), Roth and Sotomayor (1990), Roth (2002), Abdulkadiroglu and Sonmez (2003), Abdulkadiroglu (2005a, b) Roth et.al. (2007), Roth (2008)). It is only recently that the college admission problem with an entrance criterion has been studied (Perach, Polak and Rothblum (2007), Perach and Rothblum (2010). They design an algorithm that respects eligibility criteria and produces a weakly stable outcome. They also study incentive compatibility properties [2].

3.1.2 Quick sort:

One of the most popular sorting algorithms is quicksort. Quick sort executes in $O(n \log n)$ on average, and $O(n^2)$ in the worst-case. However, with proper precautions, worst-case behavior is very unlikely. Quicksort is a non-stable sort. It is not an in-place sort as stack space is required [15].

3.1.2.1 Theory:

The quicksort algorithm works by partitioning the array to be sorted, then recursively sorting each partition. In Partition (algorithm), one of the array elements is selected as a pivot value. Values smaller than the pivot value are placed to the left of the pivot, while larger values are placed to the right. Here in this chapter we use vice versa instead of increasing order we go by decreasing order.

3.1.2.2 Algorithm:

```c
int function Partition (Array A, int Lb, int Ub);
begin
select a pivot from A[Lb] ... A[Ub];
reorder A[Lb] ... A[Ub] such that:
all values to the left of the pivot are $\leq$ pivot
all values to the right of the pivot are $\geq$ pivot
return pivot position;
```

81
end;

procedure Quicksort (Array A, int Lb, int Ub);
begin
if Lb < Ub then
M = Partition (A, Lb, Ub);

Quicksort (A, Lb, M - 1);

Quicksort (A, M + 1, Ub);
end;

In Table 3.1(a), the pivot selected is 3. Indices are run starting at both ends of the array. One index starts on the left and selects an element that is larger than the pivot, while another index starts on the right and selects an element that is smaller than the pivot. In this case, numbers 4 and 1 are selected. These elements are then exchanged, as is shown in Table 3.1 (b). This process repeats until all elements to the left of the pivot are ≤ the pivot, and all items to the right of the pivot are ≥ the pivot. Quicksort recursively sorts the two sub-arrays, resulting in the array shown in Table 3.1 (c).

Table 3.1(a) | Lb | Ub |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Pivot

Table 3.1(b) | Lb | M | Lb |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1(c) | 1 | 2 | 3 | 4 | 5 |
As the process proceeds, it may be necessary to move the pivot so that correct ordering is maintained. In this manner, QuickSort succeeds in sorting the array. If we're lucky the pivot selected will be the median of all values, equally dividing the array. For a moment, let's assume that this is the case. Since the array is split in half at each step, and Partition must eventually examine all n elements, the run time is $O(n \log n)$.

To find a pivot value, Partition could simply select the first element ($A[Lb]$). All other values would be compared to the pivot value, and placed either to the left or right of the pivot as appropriate. However, there is one case that fails miserably. Suppose the array was originally in order. Partition would always select the lowest value as a pivot and split the array with one element in the left partition, and $Ub - Lb$ elements in the other. Each recursive call to quicksort would only diminish the size of the array to be sorted by one. Therefore $n$ recursive calls would be required to do the sort, resulting in a $O(n^2)$ run time. One solution to this problem is to randomly select an item as a pivot. This would make it extremely unlikely that worst-case behavior would occur.

3.1.2.3 Implementation

The source for the quicksort algorithm may be found in file qui.c. Typedef T and comparison operator compGT should be altered to reflect the data stored in the array. Several enhancements have been made to the basic quicksort algorithm:

- The center element is selected as a pivot in partition. If the list is partially ordered, this will be a good choice. Worst-case behavior occurs when the center element happens to be the largest or smallest element each time partition is invoked.
- For short arrays, insert Sort is called. Due to recursion and other overhead, quicksort is not an efficient algorithm to use on small arrays. Consequently, any array with fewer than 12 elements is sorted using an insertion sort. The optimal cutoff value is not critical and varies based on the quality of generated code.
- Tail recursion occurs when the last statement in a function is a call to the function itself. Tail recursion may be replaced by iteration, resulting in a better
utilization of stack space. This has been done with the second call to QuickSort in above algorithm.

- After an array is partitioned, the smallest partition is sorted first. This results in a better utilization of stack space, as short partitions are quickly sorted and dispensed with.

3.1.3 Scheduling Algorithms:

Scheduling theory is concerned with the optimal allocation of scarce resources to activities over time. The practice of this field dates to the first time two humans contended for a shared resource and developed a plan to share it without bloodshed. The theory of the design of algorithms for scheduling is younger, but still has a significant history the earliest.

More generally, scheduling problems involve jobs that must scheduled on machines subject to certain constrains to optimize some objective function. The goal is to specify a schedule that specifies when and which job is to be executed.

3.1.4 Basic Scheduling Algorithms:

3.1.4.1 FCFS - First-Come, First-Served [9]

- Non-preemptive
- Ready queue is a First In First Out queue
- Jobs arriving are placed at the end of queue
- Dispatcher selects first job in queue and this job runs to completion of CPU burst.
- Advantages: simple, low overhead
- Disadvantages: inappropriate for interactive systems, large fluctuations in average turnaround time are possible.
  - Example of FCFS
    - Workload (Batch system)
      - Job 1: 24 units, Job 2: 3 units, Job 3: 3 units
    - FCFS schedule:
3.1.4.2 SJF - Shortest Job First [9]

- Non-preemptive
- Ready queue treated as a priority queue based on smallest CPU time requirement
  - arriving jobs inserted at proper position in queue
  - dispatcher selects shortest job (1st in queue) and runs to completion
- Advantages: provably optimal w.r.t. average turnaround time
- Disadvantages: in general, cannot be implemented. Also, starvation possible!
- Can do it approximately: use exponential averaging to predict length of next CPU burst pick shortest predicted burst next!
- Example of SJF
  - Workload (Batch system)
    Job 1: 24 units, Job 2: 3 units, Job 3: 3 units
  - SJF schedule:
    | Job 2 | Job 3 | Job 1 |
    |-------|-------|-------|
    | 0     | 3     | 6     |
    | 24    | 27    | 30    |
  - Total waiting time: 6 + 0 + 3 = 9
  - Average waiting time: 3
  - Total turnaround time: 30 + 3 + 6 = 39
  - Average turnaround time: 39/3 = 13
  - SJF always gives minimum waiting time and turnaround time
- Exponential Averaging
  \[ T_{n+1} = \alpha T_n + (1 - \alpha) T_{n} \]
  - \( T_{n+1} \): predicted length of next CPU burst
3.1.4.3 RR - Round Robin [9]

- Preemptive version of First Come First Serve
- Treat ready queue as circular
  
  - arriving jobs are placed at end
  - dispatcher selects first job in queue and runs until completion of CPU burst, or until time quantum expires
  - if quantum expires, job is again placed at end.

![Diagram of RR](image)

**Example of RR**

- Workload (Batch system)
  
  Job 1: 24 units, Job 2: 3 units, Job 3: 3 units
  
  RR schedule with time quantum=3:
  
<table>
<thead>
<tr>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

  - Total waiting time: 6 + 3 + 6 = 15
  - Average waiting time: 5
  - Total turnaround time: 30 + 6 + 9 = 45
  - Average turnaround time: 15
  - RR gives intermediate wait and turnaround time (Compared to SJF and FCFS).
Properties of RR

- Advantages: simple, low overhead, works for interactive systems
- Disadvantages:
  - if quantum is too small, too much time wasted in context switching
  - if too large (i.e., longer than mean CPU burst), approaches FCFS
- Typical value: 20 – 40 msec
- Rule of thumb: Choose quantum so that large majority (80 – 90%) of jobs finish CPU burst in one quantum
- RR makes the assumption that all processes are equally important.

3.1.4.4 HPF - Highest Priority First [9]

- General class of algorithms => priority scheduling
- Each job assigned a priority which may change dynamically
- May be preemptive or non-preemptive
- Key Design Issue: how to compute priorities?

3.1.5 Bipartite graph:

In the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets $U$ and $V$ such that every edge connects a vertex in $U$ to one in $V$; that is, $U$ and $V$ are independent sets. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles [10].

Figure 3.2: Bipartite Graph
A graph \( G \) is a **bipartite graph** if there is \( X, Y \subseteq V(G) \) meeting the following conditions:

- \( V(G) = X \cup Y \),
- \( X \cap Y = \emptyset \),
- \( G[X] \) and \( G[Y] \) are both null graphs.

The word “bipartite” means “two parts”.

**Bipartite graphs** are often used to model some common real-world situations, where the vertices in one part \( X \) represents no. of students and the vertices in the other part \( Y \) represents no. of departments in which students apply for admissions in various departments by means of edges between \( X \) and \( Y \). These situations will be discussed in more detail in below proposed work.

Several important points are worth noting.

- A bipartite graph \( G \) with partition \( V(G) = X \cup Y \) implies that each edge of \( G \) has one end vertex in \( X \) and the other in \( Y \).
- Every null graph \( N_n \) is bipartite; any partition \( V(N_n) = X \cup Y \) demonstrates this fact.
- A bipartite graph cannot have any loops.
- A graph is bipartite if, and only if, each of its components is bipartite.
- For all \( m, n \in \mathbb{N} \), the complete bipartite graph \( K_{m,n} \) is a simple bipartite graph.

It is not hard to see that every cycle in a bipartite graph must have an even length.

### 3.2 Proposed work:

Given a set no. of students \( S = \{s_1, s_2, s_3, \ldots, s_n\} \) and a set no. of Department \( D = \{d_1, d_2, d_3, \ldots, d_k\} \), let \( S \times D \) denote the set of all possible pairs of the form \((s, d)\), where \( s \in S \) and \( d \in D \). Each students applies for the departments considering the applications that student can apply to three departments at a time by their choices. The goal is to find that every student should get admission, if not their first choice they can get through their second or third choices which ever they get a chance. The advantage of this is to try to get all students an admission and here students need not to apply new application for other departments because here we...
allowed them in one application can apply for three departments of their choices. Here
seats are not given according to student first choices but based on students scores.
According to students score a merit list is created and ranks are given and token
number are assigned according to students ranks. A counseling is conducted and
based on token number seats are allocated to every students. If any student is absent
when his token number is called in counseling then they get a chance in second round
counseling.

In College, we have many sections like Arts, Social Science, Science, Commerce, Law, Education, Management courses and many one year diploma courses. Each sections having many departments, here in this problem we take one section i.e. Science. In Science we have many departments like Computer Science, Mathematics, Statistics, Chemistry, Biotech, Microbiology, Genetics, Physics, Electronics, Zoology, Botany, and Geography. Here students apply in science section for admission.

Given a set no. of students $S = \{s_1, s_2, s_3, \ldots, s_n\}$ and a set no. of
Department $D = \{d_1, d_2, d_3 \ldots, d_k\}$, each student has a choice of selecting three
departments and each department has limited seats that are no. of students it can
accept in academic year. However, each student can accept only one course for
admission. The goal is to find that every student should get admission based on their
three choices. Based on the score, on purely merit system seat is allocated to the
students.

Consider the following set of students applied for admission according to their
scores as shown in Table 3.2. Student Applications for departments Computer

<table>
<thead>
<tr>
<th>Application number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student Names</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
</tr>
<tr>
<td>Score</td>
<td>430/500</td>
<td>425/500</td>
<td>426/500</td>
<td>427/500</td>
<td>420/500</td>
</tr>
</tbody>
</table>
The first algorithm to sort the list is quick sort. With this scheme we sort list of students considering their total marks and assign a rank and give token number. Here instead of increasing order, we go in decreasing order. Table 3.3 Sorting list

Table 3.3 (a)

<table>
<thead>
<tr>
<th>430</th>
<th>425</th>
<th>426</th>
<th>427</th>
<th>420</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pivot</td>
</tr>
</tbody>
</table>

Table 3.3 (b) Lb Ub

<table>
<thead>
<tr>
<th>430</th>
<th>425</th>
<th>426</th>
<th>427</th>
<th>420</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pivot</td>
</tr>
</tbody>
</table>

Table 3.3 (c)

<table>
<thead>
<tr>
<th>430</th>
<th>427</th>
<th>426</th>
<th>425</th>
<th>420</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In table. 3.3, the pivot selected is 426. Indices are run starting both ends of the array. One indices starts on the left and selects an element that is smaller than the pivot, while other indice starts on the right and selects an element that is larger than pivot. In this case number 430 and 420 is not selected because it is already larger and smaller than the pivot. So it selects 425 and 420, these elements are then exchanged as shown in table.3.3 (c). This process repeats until all elements to the left of the pivot are ≥ the pivot, and all items to the right of the pivot are ≤ the pivot. Quick Sort recursively sorts the two sub-arrays, resulting in the array as shown in Table.3.3. As the process, it may be necessary to move the pivot so that correct ordering is maintained. In this manner, Quick Sort succeeds in sorting the array. Here the list is sorted and the merit list is ready, we assign a token number from starting serially.

After assigning token number to all students, we use the simple first-come, first-served (FCFS) scheduling algorithm to allocate seats to the students by conducting counseling. With this scheme, in the counseling request 1st token number student and allocate seat of their 1st choice. The implementation of the First-come, First-serve is easily managed with first-in, first-out queue. When a student enters into the queue for seat, if their 1st choice seat available it is allocated or 2nd or 3rd choice at
the head of the queue. Once the seat allocation is done student is removed from the queue.

Consider the following set of students that arrive at time 0 along with time allocation to each student in minutes as shown in table 3.4. The main constraints for allocation of time to each student are verification of marks cards, income certificate, etc. Note that whatever constraints mentioned here is just author's opinion and it is not mandatory. If the student arrived in the order according to the token number and are served in First-come, First-serve order, we get the result shown in the Figure 3.3 and if any student absent when their term comes, they have to wait for 2nd round counseling.

Table 3.4: Counseling Schedule

<table>
<thead>
<tr>
<th>Application Number</th>
<th>1</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student Names</td>
<td>$s_1$</td>
<td>$s_4$</td>
<td>$s_3$</td>
<td>$s_2$</td>
<td>$s_5$</td>
</tr>
<tr>
<td>Counseling time taken</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

The waiting time for the 1st student $s_1$ is 0 minutes, 5 minutes for $s_4$, 12 minutes for $s_3$, 20 minutes for $s_2$ and 26 minutes for $s_5$. The average waiting time is 6 minutes. This algorithm is non-preemptive.

Figure 3.3: Gantt chart of First-come, First-serve algorithm.

The Round Robin algorithm is designed especially for time sharing system. Round Robin is similar to First-come, First-serve but preemptive is added to switch between students. A small unit of time called time quantum [4] is defined. In process of verification checking, there is all semester marks card verification, income verification, caste verification etc. In above algorithm we have to do these all steps within time of allocation of students. Wasting of time is possible while using above discussed algorithm in each step of verification. In this method we can avoid wasting of time between each step [11]. In this case, we give break to student between each
step namely time quantum so that simultaneously we can work with more than one student at a time. This leads to saving student time, counseling time and possibility of verifying more students. Here we can minimize average waiting time of students as shown in figure 3.4, in the following chart the time quantum is 4 units.

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_4$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 3.4: Gantt chart for Round Robin

Here Shortest Job First algorithm and Priority Scheduling algorithm is not applicable for this problem. Since the seats are allocated on purely merit bases to students, the priority is not given to any students like women quota, caste quota, sports quota, etc for allocation. Till student comes to counseling, how much time taken by each student in counseling is not known so cannot tell which student takes less time and which student takes more time so shortest job first algorithm is not applicable.

In summary, the above discussed algorithms having advantages as well as disadvantages but the best suitable algorithm is Round Robin algorithm.

3.2.1 **Graph theory** plays an important role in this problem. In this work, the admission problem is approached using the simpler class of bipartite graph. Given a set no. of students $S = \{s_1, s_2, s_3, \ldots, s_n\}$ and set no. of Dept $D = \{d_1, d_2, d_3, \ldots, d_k\}$, for ‘$S$’ students based on their score the available number of seats in ‘$D$’ departments to be allocated. This is done as follows, a bipartite graph (or bigraph whose vertices can be divided into two disjoint sets $U$ and $V$; that is, $U$ and $V$ are independent sets.) $G$ where the vertices are the numbers of students $\{s_1, s_2, s_3, \ldots, s_n\}$ and numbers of Departments $\{d_1, d_2, d_3, \ldots, d_k\}$ such that vertices are connected by edges, where edge represents choices of students to departments. For example consider the Table 3.2

In table 3.2 there are 5 students namely $S = \{s_1, s_2, s_3, s_4, s_5\}$ and 6 departments namely $D = \{\text{Computer Science- CS, Mathematics- M, Physics- P, Statistics- S, Chemistry- C, Botany- B}\}$. The bipartite graph is constructed as follows in figure 3.5.
All the students need not give 3 choices compulsory. They can give 1 or 2 or 3. From the graph we see that the total no. of vertices in set (S, D) gives the total no. of applications. The degree of each vertex in the set (S, D) specifies the student’s subject choice. The degree of each vertex in the set (D, S) gives the application received for that particular course.

### Figure 3.5: Bipartite graph with 5 students and 6 departments

<table>
<thead>
<tr>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
<th>$d_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$s_2$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$s_3$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$s_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$s_5$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.6 student choice matrix

### 3.2.2 Algorithm to find the students choices

```plaintext
for s ← 1 to n
    for d ← 1 to k
        if (SD [s][d] = 1)
            deg [s] ++;
```

93
The following example will illustrate the result of this section. Let the choices be given by:

- $S_1: CS, M, C$
- $S_2: P, M, C$
- $S_3: C, S, M$
- $S_4: CS, S, B$
- $S_5: P, C, B$

$CS : S_1, S_4$

$M : S_1, S_2, S_3$

$P : S_2, S_5$

$C : S_1, S_2, S_3, S_5$

$S : S_3, S_4$

$B : S_4, S_5$

After merit list the final outcome given by:

- $S_1: CS$
- $S_4: S$ (assume 1st choice are full so 2nd choice)
- $S_3: C$
- $S_2: P$
- $S_5: B$ (assume 1st choice and 2nd choice are full so 3rd choice)

After the seat allocation we get the graph as shown in figure 3.7. Here if the degree of any student vertex is zero then it means that particular person did not take admission in any department. The degree of department vertex shows the no. of
students admitted for that department. If degree of any department is zero, which means no one took admission for that particular department. By adding the degrees of each vertex in department sets we get the total no. of students admitted in the college.

\[
\begin{array}{cccccc}
 & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 \\
 s_1 & 1 & 0 & 0 & 0 & 0 & 0 \\
 s_2 & 0 & 0 & 1 & 0 & 0 & 0 \\
 s_3 & 0 & 0 & 0 & 1 & 0 & 0 \\
 s_4 & 0 & 0 & 0 & 0 & 1 & 0 \\
 s_5 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

Figure 3.8. Students admitted in department matrix

3.2.3 Algorithm to find the number of students in departments

for s ← 1 to n
    { for d ← 1 to k
        if adm [s][d]=1
            { deg [d] ++ //total no. of students admitted in dept
                deg ad [s] ++
                break
            }
        else
            deg ad [s]=0 // students has not taken admission in any department
    }

Here, finally seats are allocated on purely merit bases according to student choices over departments.
3.2.4: Program:

```c
#include<stdio.h>
#include<conio.h>
#include<string.h>

struct info
{
    char name[15];
    char dept[3][10];
    int perc;
};

struct info s[5];

void main()
{
    int i,j,k,temp,flag;
    char sub[][4]={"phy","chem","css"},name1[15],allo[3][10];

    clrscr();
    printf("Enter the information\n");
    for(i=0;i<5;i++)
    {
        printf("Enter the name\n");
        scanf("%s",s[i].name);
        for(j=0;j<3;j++)
        {
            printf("enter the %d sub ",j+1);
            scanf("%s",s[i].dept[j]);
        }
    }
    printf("\n");
}
```
printf("Enter the percentage\n");
scanf("%d", &s[i].perc);
}
for(i=0;i<5-1;i++)
{
   for(j=i+1;j<5;j++)
    {
       if(s[i].perc<s[j].perc)
       {
          temp=s[i].perc;
          s[i].perc=s[j].perc;
          s[j].perc=temp;
          strcpy(namel,s[i].name);
          strcpy(s[i].name,s[j].name);
          strcpy(s[j].name,namel);
          for (k=0;k<5;k++)
          {
             strcpy(namel,s[i].dept[k]);
             strcpy(s[i].dept[k],s[j].dept[k]);
             strcpy(s[j].dept[k],namel);
          }
       }
    }
}
for (i=0;i<5;i++)
{
}
Department of Computer Science, Karnataka University, Dharwad

k=0;
flag =1;
do
{
    for (j=0;j<5;j++)
    {
        if(!(strcmp(s[i].dept[k],sub[j])))
            {
                strcpy(allot[i],sub[j]);
                strcpy(sub[j],"\0");
                // printf("%s\t",s[i].dept[0]);
                flag=0;
                break;
            }
        }
k++;
    } while (flag);
}
printf("----------------------stud info-----------------
");
printf("name \tpercentage \tDept \forallot \n");
for (i=0;i<5;i++)
{
    printf(" %s\t",s[i].name);
    printf("%d\t",s[i].perc);
    for (j=0;j<5;j++)
\{
    printf("\t%s\n",s[i].dept[j]);
    printf("\t\t");  
\}
printf("\n");
\}
for(i=0;i<5;i++)
    printf("\t%s",allot[i]);
getch();
\}

OUTPUT
enter the name: s1
enter the 1 sub: cs
enter the 2 sub: m
enter the 3 sub: c
enter the percentage: 86
enter the name: s2
enter the 1 sub: p
enter the 2 sub: m
enter the 3 sub: c
enter the percentage: 85
enter the name: s3
enter the 1 sub: c
enter the 2 sub: s
enter the 3 sub: m
enter the percentage: 85
enter the name: s4
enter the 1 sub: cs
enter the 2 sub: s
enter the 3 sub: b
enter the percentage: 85
enter the name: s5
enter the 1 sub: p
enter the 2 sub: c
enter the 3 sub: b
enter the percentage: 84

<table>
<thead>
<tr>
<th>name</th>
<th>percentage</th>
<th>dept</th>
<th>allot</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>86</td>
<td>cs</td>
<td>cs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>85</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>85</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>85</td>
<td>cs</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>84</td>
<td>p</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

3.3 Conclusion:

Several Scheduling algorithm applications have been described in this paper and also graphical representation using bipartite graph described in simple way. We hope this chapter raises the awareness and adoption of Scheduling algorithm application amongst college admission problem.
References:


