TA Assignment Problem

AI – Project Report 3

Assumptions:

The following assumptions/relaxations were taken into consideration for the implementation.

- A TA can be assigned to a course, if at least one of the skill matches with the requirement of the course.
- It is required that during the recitation the TA also has to be present. This will enforce more conflicts and helps us to test the efficiency of the CSP algorithm
- The time is always considered in the AM and PM format. All skills are presented in the single line and each table is separated by a new line. The input given should strictly follow the format

Implementation:

Domain: Set of Courses

Variables: Set of TA's that can be assigned to each course

Each course is represented by an object of the Course class :

```
class Course:

    def __init__(self, courseName):
        self.courseName = courseName
        self.classStartTimes = []
        self.recitationStartTimes = []
        self.classDuration = 0
        self.recitationDuration = 0
        self.numberOfStudents = 0
        self.TAtoAttendClass = 0
        self.skills = []
        self.recitationRequired = 0
        self.isAssignedTA = 0
        self.assignedTAs = []
        self.possibleTAs = []
        self.totalAssignedTAs = 0
        self.neededTAs = 0
        self.banList=[]
```

The attributes were so chosen that we can customize it to a variety of TA assignment problems.

Each TA is represented by an object of TA class:

```
class TA:
    def __init__(self, TAName):
        self.TAName = TAName
        self.TATimings = []
        self.TASkills = []
        self.assignedCourses = []
        self.isAssigned = 0
```

To check the conflicts among TA schedule and the class/recitation schedule, we converted the time to represent unique strings.

How to find if a TA is free for a specific class?

- **Motivation :**
- Comparing in strings may be tough as opposed to integers
- Taking inspiration from the Java, System.getTimeMillis, where everything is relative to Jan 1 1970 UTC, here all time is relative to Monday

Implementation:

Every day is composed of some minutes. We have 24 hours, 1440 minutes in a day

	Time : P	Q = x:y AM	R = x:y PM
Mon	0	X * 60 + y	(X + 12) *60 + y
Tue	1440		
Wed	1440 * 1		
Th	1440 * 2		
Fri	1440 * 3		
Sat	1440 * 4		

- Time at any exact point : P+Q+R. (the exception of when the hour is 12 is handled separately)
- This integer is unique given Monday to Sun. As the number of minutes elapsed from Monday is unique.
- Now we can make numerical comparisons to check if the time conflicts.

- so For ex : Tue 2:30 PM is converted to :
  1440 + 14*60 + 30

The time conflicts can now be easily checked through a simple integer comparison.

Algorithm Implementation:

- All assignments of TA's is done in half units
- When there is no assignment possible for a course, we quit saying that the assignment is not possible for that course:

PseudoCodes:

Pseudocode for backtracking -

1. For each course –
   Check if its last course.
   If yes, and if TA requirement for it is met, then exit
2. Check if TA requirement of course is met
   if yes, go to next course
   go to step 1
3. Else, check which TA is next in course's list of valid TAs
   if that TA has not been fully assigned if that assignment is not in ban list
   assign that TA to that course
   reduce TA requirement for that course
   increment assignment list
   go to step 2
   else
   if next valid TA exists for that course
   go to step 3
   else
   call go back function

Go Back function pseudocode –

1. if moves list is empty
   return "no complete allocation possible"
   exit
   else
   pop last assignement from assignment list
   insert assignment into ban list
   set current course as popped course
   increase TA requirement of current course

Forward Checking:

Most Restrained Value : We start the forward checking with the one, which has the least number of possible TA's to be assigned.

Pseudocode for backtracking with forward-checking -

1. For each course –
   Check if its last course.
   If yes, and if TA requirement for it is met, then exit
2. Check if TA requirement of course is met
   if yes, go to next course
   go to step 1
3. Else, check which TA is next in course's list of valid TAs
   if that TA has not been fully assigned if that assignment is not in ban list
   if forward checking returns "okay"
   assign that TA to that course
   reduce TA requirement for that course
   increment assignment list
   go to step 2
   else
   go to step 3
   else
   if next valid TA exists for that course
   go to step 3
   else
   call go back function

Forward Checking Function Pseudocode –

1. For each course –
   Check if course for which assignment is being made
   .
   If yes, skip that course
2. Check if TA requirement of course is met
   if yes, go to next course
   go to step 1
3. Else, check which TA is next in course's list of valid TAs
   if that TA has not been fully assigned and if that assignment is not in ban list and if that TA is different from the TA currently being checked for
   go to next course and step 2
   else
   go to step 3
4. If for a course no, valid TA can be identified, return "not possible" and exit
5. Else, if all courses checked and have other options, return "okay" and exit

Forward Checking with constraint propagation:

The constraint Propagation was enforced with two methods :

- **Node Consistency**
- **Arc Consistency**

Node consistency: The domain of each course of possible TA's was restricted to the ones whose times were non conflicting and had at least one of the skill possessed by the course.

Arc consistency: For every TA removal, we checked the other domains of courses with possible TA's recursively until the entire set was consistent.

```
#perform node consistency by restricting the domain of TA's that an be assigned to a course
arcConsistency():
   isConsistent = true;
 # after the removal of ta in other domains check whose domains have a length of 1
 toCheckTAs = getAllTAs who has only one possible TA for a course
 # Check the possible removal of this TA now will not end up in a empty domain

   for ta in toCheckTAs:
      if this TA is the only TA in possibleTAs for more than one course
      isConsistent = false;

   return isConsistent

runFCwithCP()
{
   is the AssignmentComplete?
 return true

 pick a course in ( unassigned courses)

 if course.possibleTA's.len == 0:
    return false

 for every TA present in possible TA's of the course:
    course.assignedTAs.add(ta)
            course.mark = assigned
            course.requiredTAs = course.requiredTAs - 0.5

      remove this TA from the possible list of other courses
      #for every removal, apply arc consistency
            if !ArcConsistent()
               return false
```

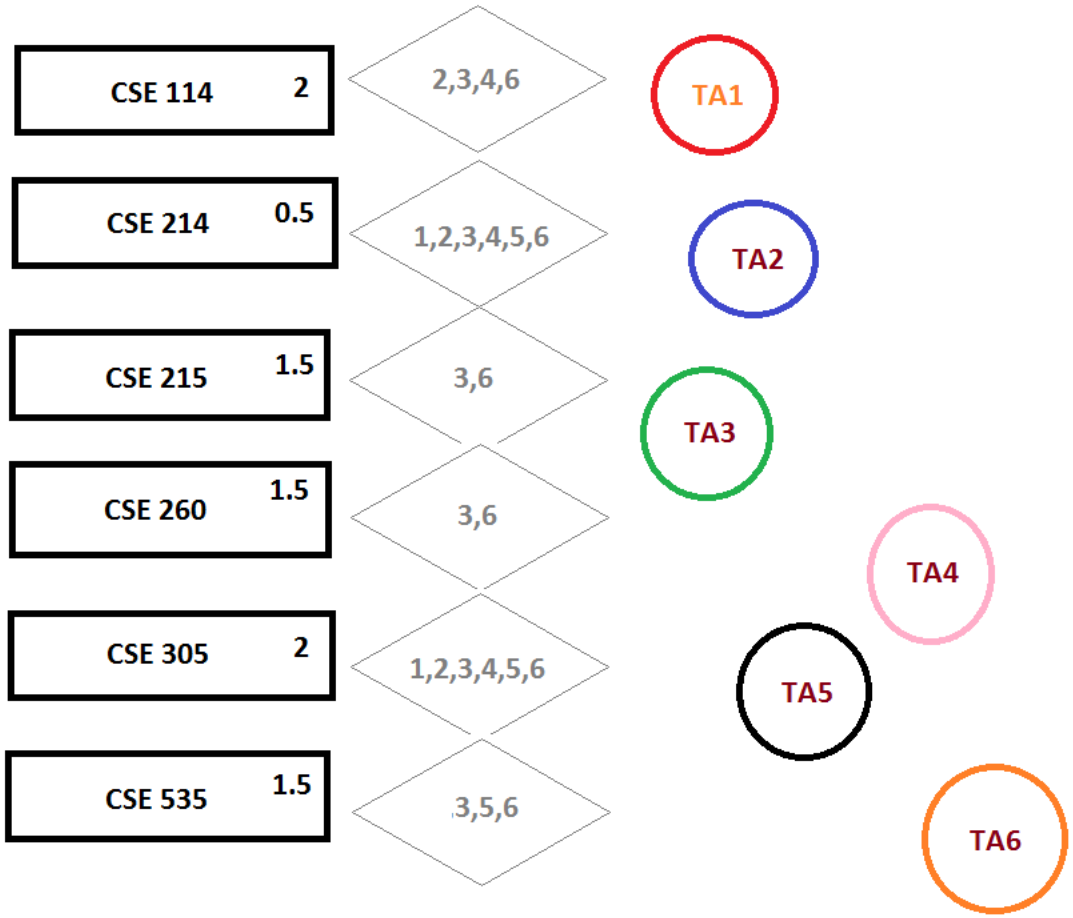
```
if (runFCwithCP()) # Make a recursive call with the new assignment
    return true

    # the TA assignment failed, so undo the move
    course.assignedTAs - {ta}
    course.mark = unassigned
    course.requiredTAs = course.requiredTAs + 0.5
```

Now assign back remaining free TA's who are in the possible TA list to the courses.

Sample Output Analysis for all methods:

Consider the following system of courses and TA's. The number of TA's required are indicated by each course on the top right of the rectangle. The possible TA's are in the rhombus.



The total number of TA's required as per the requirement is 8 and we have only 6. The algorithm runs in such a way that if it is possible to satisfy all the courses(timings and skills) with the given TA's it assigns the maximum number of courses.

CSE114 : TA2, 1,TA3, 0.5,TA4, 0.5

CSE214 :TA1, 0.5

CSE215 :TA3, 0.5,TA6, 0.5, needed : 0.5

CSE260 :TA6, 0.5, needed : 1.0

CSE305 :TA4, 0.5,TA5, 0.5, needed : 1.0

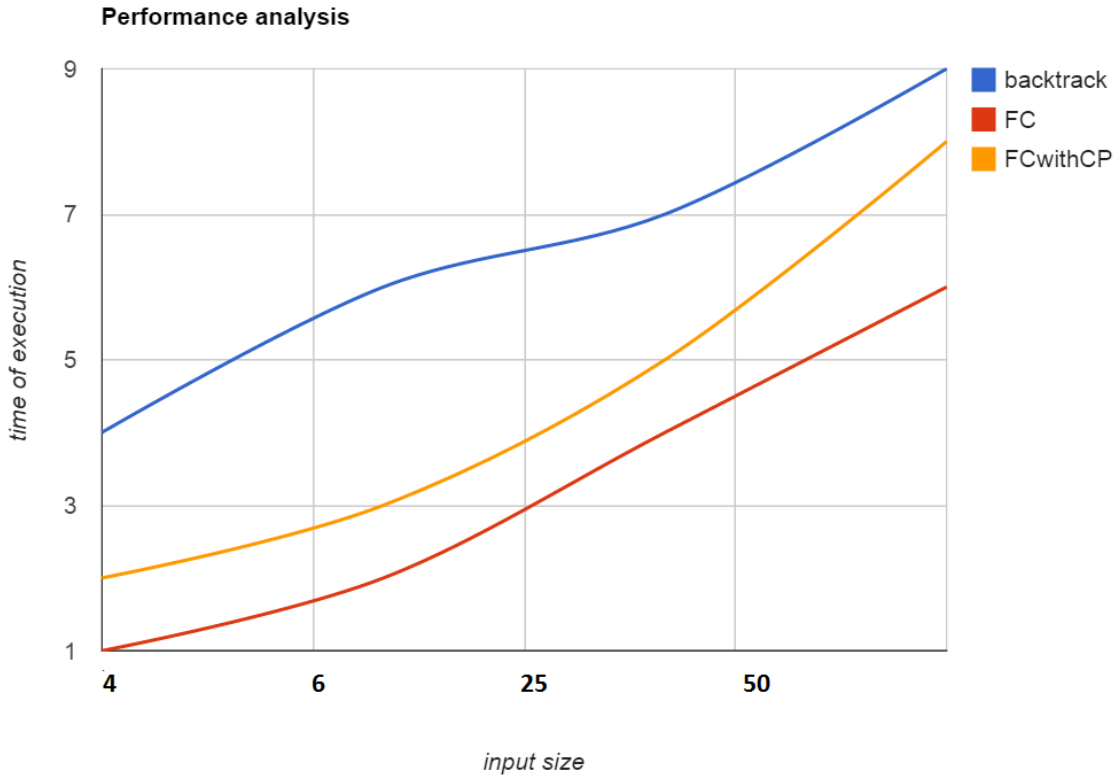
CSE535 :TA5, 0.5, needed : 1.0

Observations –

1. Backtracking is an exhaustive strategy and is similar to a brute-force assignment technique
2. Forward checking reduced the time for assignment by backtracking. This difference was more when the input file was large. This could be because the benefits of forward checking increased compared to the cost of checking.
3. Constraint propagation is a technique whose performance largely depends on its implementation and choice of constraint. For our implementation, we chose to go ahead with Node Constraint Propagation and arc consistency. It performs good for small inputs, but if the number of inputs increases, forward checking gives good performance

Performance Analysis:

- The backtracking performs the worst.
- The FC performs the best
- FC with constraints performs average due to the number of constraints



Conclusions -

1. Forward Checking allows for backtracking to be more efficient as it saves the time between identifying a false move.
2. Forward checking can be further improved by increasing the depth until which it is done before assignment is made. But, that would be a trade-off with computation.
3. Constraint Propagation sometimes proves costly, if the constraints have to be checked among large set of inputs.
4. Unlike Backtracking, Constraint Propagation is not brute-force exhaustion but is a form of optimization.