CS362 (Spring 2017) Machine Intelligence

Assignment 3 – Constraint Satisfaction Problems (CSPs)

Due date: 22/03/2017

This assignment has a weight of 9% on your final grade

You are given a list of 16 students that you would like to have seated during lunch time. However, each of these students is conflicted with a given subset of students in this list. Students sit in a grid configuration. If a student sits next to a student (in any direction, i.e., horizontal, vertical and both diagonals) that they are conflicted with, they will fight. You wish to find a sitting arrangement that would lead to no fights among the students. The list of conflicted students is given below:

```
Alan-Dan,Jack,Jeff,Jill,Joe,Kim,Sue
Bill-Jack,Joe,John,Kim,Nick,Sue,Tom
Dan-Alan,Jeff,Mike,Nick,Sam,Tom,Will
Dave-Jeff,John,Mike,Nick,Sue,Tom,Will
Jack-Alan,Bill,Jill,John,Nick,Sam,Sue
Jeff-Alan,Dan,Dave,John,Sam,Tom,Will
Jill-Alan,Jack,Joe,John,Kim,Sam,Sue
Joe-Alan,Bill,Jill,Mike,Nick,Tom,Will
John-Bill,Dave,Jack,Jeff,Jill,Kim,Sam
Kim-Alan,Bill,Jill,John,Mike,Nick,Will
Mike-Dan,Dave,Joe,Kim,Sam,Sue,Tom
Nick-Bill,Dan,Dave,Jack,Joe,Kim,Sue
Sam-Dan,Jack,Jeff,Jill,John,Mike,Will
Sue-Alan,Bill,Dave,Jack,Jill,Mike,Nick
Tom-Bill,Dan,Dave,Jeff,Joe,Mike,Will
Will-Dan,Dave,Jeff,Joe,Kim,Sam,Tom
```

Note that you will find this list in the file *conflicts.txt*. The data given above can be interpreted as follows:

If Alan sits next to Dan, Jack, Jeff, Jill, Joe, Kim or Sue then there will be a conflict. There will also be a conflict if Bill sits next to Jack, Joe, John, Kim, etc. This rule applies to all the students in the data given above.

You will use various approaches to solving CSPs that you have learnt in class in order to tackle this problem. For the first part of this assignment, you may assume that the initial sitting arrangement is given as shown in Figure 3.1 below. This sitting arrangement is represented by the following 2D array,

```
init_arrangement = [["Alan","Bill","Jack","Jeff"],
                    ["Dan","Dave","Jill","Joe"],
                    ["John","Kim","Sam","Sue"],
                    ["Mike","Nick","Tom","Will"]]
```

Alan	Bill	Jack	Jeff
Dan	Dave	Jill	Joe
John	Kim	Sam	Sue
Mike	Nick	Tom	Will

Figure 3.1 – Initial student-sitting configuration

We have provided you with a couple of abstract classes that you should use in your implementation. For question 1, you will use the class **CSP_Solver** defined in the script *min_conflicts.py*.

1) MIN CONFLICTS (40 pts)

(a) Your first task is to find the number of conflicts of a given student in a given sitting configuration. The method *get_num_of_conflicts* takes 2 arguments. The first argument is a string representing a student, and the second is a 2D array representing a sitting arrangement. This method should return the number of conflicts that the student has in the given arrangement. For example from Figure 3.2 we see that Jill is conflicted with 5 students, Jack, Joe, Kim, Sam and Sue. Therefore, as an example if we call your method as *get_num_of_conflicts ("Jill", init_arrangement)* it should return 5. – **10 points**

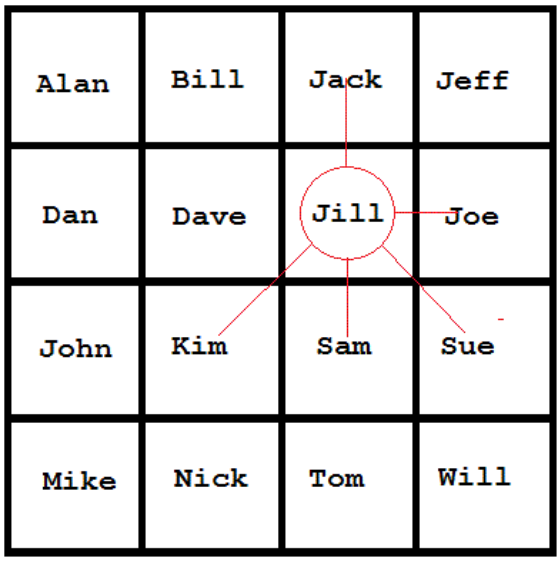

Figure 3.2 – Number of conflicts Jill has in the initial sitting arrangement (5 conflicts)

(b) The method *get_total_conflicts* takes as input a sitting arrangement and should return the total number of conflicts present in that arrangement. This should be done by summing up the total number of conflicts each student has in the given arrangement (note that there will be double counting). If you call this method with the initial assignment in Figure 3.1, it should return 26 (since there are 26 total conflicts in the initial sitting arrangement). – **5 points**

For this part of this assignment you will now solve the CSP using the min-conflicts algorithm. The way this algorithm works is as follows:

1. Start with the initial arrangement
2. If there aren't any conflicts then the problem is solved, otherwise move to step 3
3. Choose one of the conflicting seats at random
4. Find the seat that the conflicting seat can swap with (you can swap any seat with the conflicting seat) that will give the minimum number of overall conflicts. If there is a tie, find the seat with the minimum row index and if there is still a tie, find the seat with the minimum column index. Swap the seats.
5. Repeat steps 2 through 4 until the problem is solved

You already have the methods defined for step 2. For step 3, please use the method we defined for you *find_a_random_student* to select a seat at random and make sure that that student has a conflict (otherwise keep calling *find_a_random_student* until you find a conflicting student).

(c) For step 4 of the algorithm in part (b) you will use the method *get_best_arrangement* which takes as input a string representing a conflicted student e.g. "Kim", as well as a 2D array representing an arrangement. This method should implement step 4 of the above algorithm and return the new arrangement. – **15 points**

(d) The method *solve_csp* takes as input an arrangement and should solve the CSP using the algorithm we defined above. It should return the final sitting arrangement. – **10 points**

2) <u>Backtracking Search</u> (60 pts)

For this part of the assignment you will now use backtracking search to solve the CSP, so please make use of the script *backtracking.py*. The initial assignment is now given as follows:

```
init_arrangement = [["","","",""],
                    ["","","",""],
                    ["","","",""],
                    ["","","",""]]
```

This means that at the beginning, there are no assigned seats.

(a) The method *backtracking_search* takes as input an initial empty arrangement like the one given above, and assigns a student to every seat by using the backtracking algorithm. At every step you should select the next seat, the next seat being to the right of the previously

assigned seat. If the previously assigned seat was the rightmost seat, then you should go to the next row and first column to place the next student. The first student should be assigned to the leftmost top corner in the seating arrangement. For a given seat, you should try to assign the students in alphabetical order. For instance, for the top leftmost seat, you will first try assigning Alan, and if there is a conflict you will backtrack and try assigning Bill, then Dan and so forth. This method should return the final sitting arrangement. – **25 points**.

(b) For this section you will implement forward checking to improve your backtracking algorithm. To do this you will first create a domain dictionary like the one shown below:

{ (0, 0): ['Alan', 'Bill', 'Dan', 'Dave', 'Jack', 'Jeff', 'Jill', 'Joe', 'John', ……]

(0, 1): ['Alan', 'Bill', 'Dan', 'Dave', 'Jack', 'Jeff', 'Jill', 'Joe', 'John', ……]

(0, 2): ['Alan', 'Bill', 'Dan', 'Dave', 'Jack', 'Jeff', 'Jill', 'Joe', 'John', ……]

….. }

The keys in the dictionary represent the index of a seat in a given configuration and can therefore take the values (0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1)… (3, 3) for a 4x4 grid configuration. The values of the dictionary represent the domain (where the domain indicates all the possible students that a given seat can be assigned to). Initially every seat can be assigned any value, therefore the domain should consist of all the 16 students. Remember that with forward checking, whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y ’s domain any value that is inconsistent with the value chosen for X. The method *forward_checking* takes as input two arguments. The first argument is a tuple representing an assignment e.g. ((0, 0), “Alan”) which can be interpreted as assigning the seat at index (0, 0) to Alan. The second argument is a domain dictionary like the one shown above. Given any assignment and any domain dictionary, this method should perform forward checking and return an updated domain dictionary. For example, by assigning (0, 0) to Alan, the following keys in the domain dictionary should be updated:

{(0, 0): ['Alan']

(0, 1): ['Bill', 'Dave', 'John', 'Mike', 'Nick', ……]

(1, 0): ['Bill', 'Dave', 'John', 'Mike', 'Nick', ……]

(1, 1): ['Bill', 'Dave', 'John', 'Mike', 'Nick', ……]

….. }

The rest of the dictionary remains the same. – **15 points**

(c) For this section you should implement backtracking search with the MRV heuristic and forward checking, by making use of the method *backtracking_with_forward_checking*. This method should return the final arrangement. – **5 points**

(d) For this section you should implement backtracking search with the MRV heuristic and arc-consistency, by making use of the method *backtracking_with_ac3*. This method should return the final arrangement. – **15 points**

You may write additional classes, methods, functions of your own, but make sure that the names of the classes and methods that we have provided you remain unchanged. Prepare and upload a zip file which contains all your code and name this file as *<your first name>_<your last name>_ assignment2.zip*
Send email to both the instructor and the TA if you decide to return your assignment late