UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

MAI611 AI Fundamentals (Fall Semester 2023)

Second Assignment: Solving the Bingo Constraints

Assignment Deadline: 28th November 2023

1. Introduction

For the second MAI611 assignment you would need to solve by depth-first search a constraint satisfaction problem that deals with the construction of templates for Bingo cards.

2. Requirements

(i) The implementation involves the definition in Java of object Bingo_State as an extension of the abstract object Search_State. You are given the objects Search_State, Search_Node, Search, and Bingo_Search, the latter as an extension of Search. You are also given the driving class, Run_Bingo_Search. The implementation requirements will be discussed further in the sequel.

(ii) You may work in pairs.

(iii) During the tutorial session of November 28th, 2023, you will need to demonstrate your program. At the end of the demonstration, you will need to submit, in written form, a report explaining how your design of object Bingo_State addresses the representation problem for the given constraint satisfaction problem; the report should include your code for the Bingo_State object, with sufficient explanatory comments, as an appendix.

3. Bingo Cards

A Bingo player holds a cluster of six, 3 x 9, cards. Thus the cluster of cards can be considered as an 18 x 9 block, where 72 positions are blank and the remaining 90 positions hold the 90 numbers as illustrated in Figure 1 where the blanks are shown in red; hence each number from 1 to 90 appears once.

2		21	36		52	64		
6		24				69	70	82
	12			49	59		78	89
4				44		63	71	80
		23		48		66	79	86
8	13	26	37		58			
	10	22	35	43	51			
	14	27	39	45			72	
1					55	62	73	90
7					50	61	74	85
	17		31		57	65	76	
	18	23	32	46				88
5			30		53		75	83
9	11		38	42	54			
	15	29		47		68		87
3				40		60	77	81
	16	20	33	41				84
	19	25	34		56	67		

Figure 1: Example Bingo Cards

4. Constraints

The constraints, referring to an 18 x 9 block arrangement as shown in Figure 1 are the following:

1. Each number from 1 to 90 appears once as already mentioned.
2. Each row contains five numbers.
3. The first column contains the 9 numbers from the domain {1, 2, .., 9}.
4. The last column contains the 11 numbers from the domain {80, 81, …, 90}.
5. Each column from the second to the last but one, contains the 10 numbers from its corresponding domain, i.e., the domain {10, 11, …, 19} for the second column, the domain {20, 21, …, 29} for the third column and so on.
6. For each of the six, 3 x 9, sub-blocks as shown in Figure 2:
   - Each 3-element column contains either one or two numbers, i.e. no such column is entirely blank.
   - When there are two numbers these should be in ascending order, irrespective of whether there is a blank between them.
   - Overall, there should be six such columns with 2 numbers and three with 1 number, giving the overall fifteen numbers contained in each sub-block.
7. The 6 consecutive triplets of the first column (see Figure 3) of the block should comprise 3 triples with 2 numbers and 3 triples with 1 number. Likewise the internal columns should comprise 4 triples of 2 numbers and 2 with 1 number, while the last column should comprise 5 triplets with 2 numbers and 1 with 1 number.

Since the assignment concerns the construction of bingo card templates, constraint no. 1 is ignored and constraints 3-5 are simplified as follows:

3. The first column contains 9 numbers.
4. The last column contains 11 numbers.
5. Each column from the second to the last but one, contains 10 numbers.

1	2		21	36		52	64		
	6		24				69	70	82
		12			49	59		78	89
2	4				44		63	71	80
			23		48		66	79	86
	8	13	26	37		58			
3		10	22	35	43	51			
		14	27	39	45			72	
	1					55	62	73	90
4	7					50	61	74	85
		17		31		57	65	76	
		18	23	32	46				88
5	5			30		53		75	83
	9	11		38	42	54			
		15	29		47		68		87
6	3				40		60	77	81
		16	20	33	41				84
		19	25	34		56	67		

Figure 2: Illustrating the six, 3 x 9, sub-blocks

2		21	36		52	64		
6		24				69	70	82
	12			49	59		78	89
4				44		63	71	80
		23		48		66	79	86
8	13	26	37		58			
	10	22	35	43	51			
	14	27	39	45			72	
1					55	62	73	90
7					50	61	74	85
	17		31		57	65	76	
	18	23	32	46				88
5			30		53		75	83
9	11		38	42	54			
	15	29		47		68		87
3				40		60	77	81
	16	20	33	41				84
	19	25	34		56	67		

Figure 3: Each column viewed as six consecutive triplets

6. Problem Decomposition

In a game of Bingo, each player has a different arrangement of cards. The crucial step in generating “unique” cards is in constructing an 18 x 9 template showing the blank positions and the positions to be filled with the numbers. It is the construction of such a template that constitutes a constraint satisfaction problem, while the filling of the template with the actual numbers (although it can also be solved as a CSP) is easily done by decomposing it into nine sub-problems corresponding to the assignment of values to the 9, 10, or 11 variables of each column, that respectively have disjoint value domains; each variable can be randomly assigned a value from its domain that is not already assigned to another variable of the given column, while the ascending order constraint can also be addressed as a separate sub-problem.

Thus the problem of constructing “unique” bingo cards can be decomposed into two main sub-problems that are sequentially applied, namely the construction of a template, and the filling of the template. The outputs of these two sub-problems for the Bingo cards of Figure 1 are illustrated below:

```
1  0  1  1  0  1  1  0  0
1  0  1  0  0  0  1  1  1
0  1  0  0  1  1  0  1  1
1  0  0  0  1  0  1  1  1
0  0  1  0  1  0  1  1  1
1  1  1  1  0  1  0  0  0
0  1  1  1  1  1  0  0  0
0  1  1  1  1  0  0  1  0
1  0  0  0  0  1  1  1  1
1  0  0  0  0  1  1  1  1
0  1  0  1  0  1  1  1  0
0  1  1  1  1  0  0  0  1
1  0  0  1  0  1  0  1  1
1  1  0  1  1  1  0  0  0
0  1  1  0  1  0  1  0  1
1  0  0  0  1  0  1  1  1
0  1  1  1  1  0  0  0  1
0  1  1  1  0  1  1  0  0
```

```
2  0 21 36  0 52 64  0  0
6  0 24  0  0  0 69 70 82
0 12  0  0 49 59  0 78 89
4  0  0  0 44  0 63 71 80
0  0 23  0 48  0 66 79 86
8 13 26 37  0 58  0  0  0
0 10 22 35 43 51  0  0  0
0 14 27 39 45  0  0 72  0
1  0  0  0  0 55 62 73 90
7  0  0  0  0 50 61 74 85
0 17  0 31  0 57 65 76  0
0 18 28 32 46  0  0  0 88
5  0  0 30  0 53  0 75 83
9 11  0 38 42 54  0  0  0
0 15 29  0 47  0 68  0 87
3  0  0  0 40  0 60 77 81
0 16 20 33 41  0  0  0 84
0 19 25 34  0 56 67  0  0
```

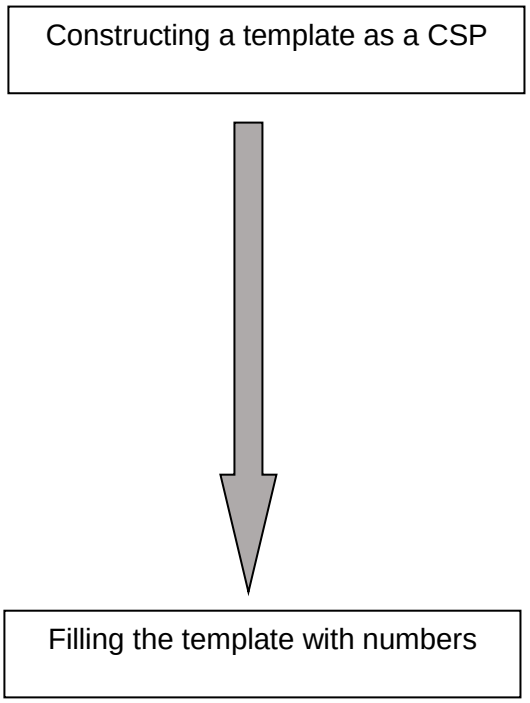

Figure 4: Problem decomposition

The assignment concerns the first sub-problem which is the construction of a template, where some randomness has to be incorporated, as explained below, so that a different template can be constructed every time the program is run.

7. Constructing Templates for Bingo cards as a CSP

For the given CSP there are 162 (18 x 9) binary variables; a value of 0 denotes a blank space and a value of 1 denotes a space that is to be filled by a number. Hence the size of the search space is 2^{162}.

The CSP problem in question is to be solved through depth-first search with backtracking. The unbound variables can be ordered row-wise or column-wise, or the next unbound variable can be randomly selected; experiment with all these orderings. The two possible values, 0 and 1, for the selected unbound variable are to be randomly ordered when generating the successor states. Look ahead (see below) ought to be used when vetting the successor states to reduce the possibility of backtracking, and thus not to end up having to revoke large parts of the search tree.

8. Java Implementation

The Java implementation is illustrated in Figure 5. The generic object classes, Search_State, Search_Node and Search have been discussed in Unit 2 of the course (please note that in that discussion Search_State was referred to as Problem_State). You are given these generic objects, together with the specific object Bingo_Search that extends Search, and the driving class Run_Bingo_Search. Hence what you are required to do for this assignment is to define concrete object Bingo_State as an extension of object Search_State.

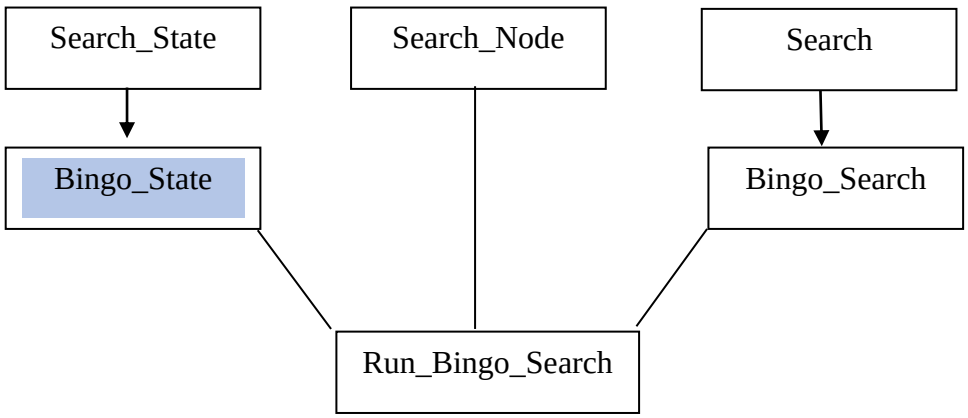

Figure 5: Classes involved for solving the CSP

Please note that the basic constructor for the Bingo_State object has no arguments. This constructor is called for the construction of the initial template configuration whereby all 162 variables are unbound; hence this constructor gives the denotation for an unbound variable, e.g. through the non-domain value -1. A copy constructor would also need to be defined.

Please also note that in order to allow for the possibility of storing the constructed goal template into a file, the constructor for the Bingo_Search object accepts as input a name for this file. The name of an actual file is given as a command line argument to the driving class Run_Bingo_Search. This argument needs to be given but it can

be ignored whereby no such saving is done. If the constructed template is to be saved, this should be done in method goalP of the Bingo_State object.

9. Global and local constraints and looking ahead

Two of the public methods of object Bingo_State that you need to implement are the goalP method and the get_Successors method, that respectively deal with global and local constraints.

Method goalP checks whether *this* is a goal state. To be a goal state, all the variables in its (encapsulated) template must be assigned a value (0 or 1), i.e. no variable is unbound, and in addition the following global constraints must be satisfied:

- The variables on each row must add up to 5
- The variables on each column must add up to 9 for the first column, 11 for the last column and 10 for the other columns.
- The (six) triplets of the first column comprise three whose variables add up to 2 and three whose variables add up to 1; likewise, the triplets of the last column comprise five whose variables add up to 2 and one whose variables add up to 1, while the triplets of each of the inside columns comprise four whose variables add up to 2 and two whose variables add up to 1.

Method get_Successors computes the successors of *this* and returns them in an ArrayList. The method selects an unbound variable (row-wise, column-wise or randomly) and forms its two, at most, successor states, respectively assigning to the selected unbound variable one of its domain values, 0 or 1, whereby the value ordering is done randomly. Hence, potentially there are at most two successor states, one in which the selected variable is assigned the value of 1 and the other the value of 0, and these two successor states are randomly ordered. In each successor state, the selected variable is designated by its row and column indices in the state's template. If a successor state is to be kept, it must be locally consistent, meaning that the value assignment for the selected variable does not violate the relevant constraints regarding the given row and the given column, taking into consideration unbound variables. For example, if the variable assignment results in the given row having three of its variables assigned a value of 1, five assigned a value of 0 and one variable being unbound, a local violation is detected since even if the unbound variable is subsequently assigned the value of 1, the global row constraint for a sum of 5 cannot be satisfied. Hence checking local constraints for consistency embodies looking ahead in an attempt to predict whether the given variable assignment is doomed to future failure.

In addition to the looking ahead involved in the local consistency checking, i.e., with respect to the implicated row and column, further, broader scope (involving all rows and columns), looking ahead can be applied, especially if the variable selection is done column-wise. This looking ahead entails the projection of a successor state into a state resulting from necessarily imposed future variable assignments. For example, if a row already includes 5 variables assigned to 1, all its unbound variables will necessarily be assigned to 0. Likewise if the number of variables assigned to 1 plus the number of unbound variables equals 5 in a given row, all the unbound variables will necessarily be assigned to 1. Following these row-based projections, all columns

can then be tested for local consistency and if any violation is detected, the given successor state is dropped since it will necessarily lead to backtracking.

10. Solvability of CSP

The given CSP is solvable and in fact there are numerous feasible solutions. Thus, the specified constraints (involving the 162 variables) are mutually consistent. A CSP is unsolvable if none of the possible variable assignments (2^{162} in this problem) satisfies the set of constraints and thus all assignments are considered unfeasible (in such a case all values for at least one variable are refuted). Deriving a feasible solution may vary substantially depending on the order of selecting the variables in conjunction with the applied looking ahead. In your report discuss this aspect by comparing the derivation of feasible solutions when the variables are selected row-wise, column-wise or randomly.

11. Using a language other than Java

You are allowed to use a language other than Java for this assignment, e.g. Python, if you so wish. In this case you would need to implement your solution from scratch including the implementation of a search tree and of a depth-first search method. Using Java simplifies your task, since by making use of the given code that implements a search tree and various search methods, all you have to do is to define a class (Bingo_State) as an extension of a given abstract class (Search_State).