UNIVERSITY OF SASKATCHEWAN

Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 3

Constraint Satisfaction Problems

Date Due: 29 October 2018, 11:59pm **Total Marks: 50**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- Each question indicates what to hand in.
- Do not submit folders, or zip files, even if you think it will help.
- **Assignments must be submitted to Moodle.**

Version History

- **23/10/2018**: Added requirement to submit execution instructions.
- **18/10/2018**: released to students

Overview

A Latin Square of order N is a $N \times N$ matrix containing the integers $1, \ldots, N$, such that each value appears exactly once in a row, and exactly once in a column. Another way to express this is that each row and column is a permutation of the numbers $1, \ldots, N$. For example, the following is a Latin Square of order 5:

```
1 2 5 3 4
3 5 1 4 2
5 4 3 2 1
2 1 4 5 3
4 3 2 1 5
```

These are *similar* to the popular Sudoku puzzles (after the puzzle is finished), but a little simpler, as they do not have "block constraints." As you may already have guessed, there are many Latin Squares of order N, and many different values of N we can use.

If we take a Latin Square and erase some of the values, we create a kind of a puzzle (again, similar to a Sudoku puzzle). For example, here's a partially filled in 5×5 matrix, with the _ symbol representing a blank cell:

```
_ 2 _ _ 4
3 _ 1 _ _
_ 4 _ _ 1
2 _ _ _ 3
_ _ 2 1 _
```

This matrix was created by deleting values from the Latin Square, above, so we know it can be completed. We can call this an incomplete Latin Square.

The **Latin Square Completion Problem** is to fill in the the blank cells of an incomplete Latin Square, using the values from $1, \ldots, N$, to create a complete Latin Square. This problem was often used by researchers as a source of solvable problem instances of various sizes (N) and difficulty levels (the number of blanks), to test and evaluate the performance of their constraint satisfaction algorithms.

Example problems

A number of example files, containing one or more incomplete Latin Squares, are available to you on the Moodle page. All the examples given in the examples file are solvable (and were constructed by a program to generate solvable instances). Some of them may have more than one completion. All we need is to find one of them. Some of the data files have just one example. One of the files has many examples. Each example has the same format, as follows.

- The first line in the file contains a single integer, indicating the number of examples in the file. The rest of the file consists of examples, as follows.
- The first line of an example is the order, i.e., the value for N.
- Exactly N rows with N values, separated by spaces. The value _ indicates a blank.
- There is always exactly one blank line after every example.

You'll be asked to apply your implementations to a file containing 56 examples, which are generally increasing in difficulty. Once you notice your program is not solving problems of a certain size, you can terminate the program to save some of your time.

Coding hints

The simplest way to deal with the data files is to write a single function that can read in the file, and produce a list of "squares", i.e., a 2-dimensional list (or array) of integers (or strings). Once you have one or more of these "squares" in a list, you have a lot of flexibility. You can implement a Problem class code ti initialize a Problem given any single square. This will prevent a lot of needless recoding as you move from files with 1 example to files with several examples.

The constraints in this problem are called "global" in the text, because they involve whole rows (and columns). For example, the constraint on a row is that each row is a permutation of the numbers $1, \ldots, N$. However, and this is important, all global constraints can be re-written using multiple binary constraints (i.e., constraints between 2 variables). In our case, a row is a sequence of cells, and each cell can be a variable. Furthermore, the permutation constraint can be rewritten as a so-called "all-different" constraint involving all the cells on the row, which in turn can be rewritten in terms of multiple "not equal to" constraints between all pairs of variables. The point is, don't over think the constraints, and don't over think the goal test.

A problem made up entirely of not-equals constraints is as straightforward as it gets (it's like searching a map with A* using straight-line distance as your heuristic — really well suited for the problem and a good teaching example, but not representative of the variety of problems to be solved). We can get a pretty good understanding of constraint solving using these simple constraints, but keep in mind that there are constraint problems involving other kinds of constraints.

Execution instructions

The markers may or may not wish to run your program, to verify your results. To help them, you should provide brief instructions on what to do to get your program running. Include:

- Programming language used (including version, e.g., Java 8 or Python 3)
- A simple example of compiling and/or running the code from a UNIX shell would be best.

Keep it brief, and name it with the question number as the following example: `a3q1_EXECUTION.txt`. If your assignment uses third party libraries, they have to be included in your submission.

Question 1 (15 points):

Purpose: To apply the techniques of AIMA Chapter 3 (Assignment 1) to this problem.

Degree of Difficulty: Easy.

To truly understand how bad the techniques of Chapter 3 (Assignment 1) are for this problem, we're going to implement it. It should not take too long, provided you understand the interfaces involved. The result will be a fairly easy program to write, but it will be very slow, except for very small problem instances.

You may use your own implementation of DFS from your search strategies for Assignment 1, or you can make use of the solutions provided (in Python only, sorry). In any case, your work should be focussed on the Problem classes, and none of what you do needs to affect the search strategies. DFS is not going to change!

You'll need to implement:

- **State**: Represent the problem naively, as a 2-dimensional list (or array) of integers (the blank in the file could be stored as a zero). You could add a list of blank locations by giving the row and column as integer pairs (e.g., $(2, 3)$). This would help speed up some of the other functions.
- **Problem** `is_goal(state)`: Checks if state array is a true Latin Square. Note that if there is a blank (a zero), it's definitely not a completed Latin Square; you don't have to keep checking if there are any blanks in your list of blanks.
- **Problem** `actions(state)`: A blank can be filled in with any number $1, \ldots, N$. You can specify which blank to fill in, or you can assume it will be the first blank in your list of blanks. But don't actually fill in the blank yet. That's the work of `result()`.
- **Problem** `result(state, action)`: Creates a new State object with the blank filled in, as described in `action`. Make sure you copy your lists/arrays, and change the copy.

Note: Don't try to add anything clever to the code to make it faster. Your program does not have to solve all the problems! Just implement the simplest version of `actions()` and `result()`. We'll be more clever in later questions. This is a straw-man implementation of an idea that seemed pretty good in Chapter 3, but is really not very good here. You need to see it to believe it. So be quick about getting the simplest implementation of the above specification as possible.

Questions to answer

Apply your implementation to the examples in the given data file `LatinSquares.txt`. Use a time-limit of 10 seconds for each problem.

Answer the following questions and submit them:

1. How many of the problems did your implementation solve within 10 seconds?
2. Consider the largest N that your implementation as able to solve. Do a rough, back-of-the-envelope calculation to predict how much time it would take to solve the *next largest* problem in the file, and the *very largest* problem in the file. Submit your rough calculations, and your predictions.

What to Hand In

- A file named `a3q1_EXECUTION.txt` containing brief instructions for compiling and/or running your code. See page 1.
- A file named `a3q1.LANG` containing your implementation of the State and Problem classes. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`.

- A file named `a3q1.txt` (other file formats like PDF, DOC, DOCX, RTF are also acceptable) with the responses to the above questions.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 6 marks: Your implementation of the State and Problem class captures the problem in a way that reflects Chapter 3. Your program does not have to solve all problems.
- 9 marks. Your answers to the questions are plausible and supported by evidence.

Question 2 (15 points):

Purpose: To implement the Constraint Satisfaction Problem concept of State as variables with domains.

Degree of Difficulty: Moderate. The transition from A3Q1 to A3Q2 will take the most time of all questions.

In this question, you'll replace the State from the previous question. It will involve making changes to your State and Problem classes, just to change the State representation from 2-dimensional lists/arrays, to a collection of CSP variables with domains. This is a preliminary step towards applying more advanced algorithms, but essentially, your implementation of a3q2 will explore the same set of combinations as the previous question. We won't get advanced performance in this question.

You should only need to change the Problem and State classes.

- **Variable**: A variable should be an object with a current value, and a list or set of domain values. If the variable is not assigned a value yet, the current value can be NULL or None. When the variable is assigned a value, the domain should be set to an empty list or set. A variable does not need to know its own identifier, but it could.
- **State**: In this question, your State should consist of a collection of Variables.
  - The *collection* could be a list, or a 2-dimensional list, or a dictionary indexed by variable identifiers.
  - Each variable needs an identifier, and for some problems a string would work fine. But here, it might be easiest to use a pair of numbers representing the row-column index of the variable's location in the square. The identifier is used to find the variable by name, without having to store multiple references (Using references as variable identifiers is possible but not advised. The programmer time spent debugging is not worth the speed increase you'd see. Leave that kind of optimization for your code that matters more. We're experimenting.)
  - Remember that variables start out unassigned, and are assigned as the search goes on. Your state could have a list of unassigned variable identifiers to help avoid searching for unassigned variables. Don't store your actual variables in two different lists, unless you enjoy headaches and frustration.
- **Problem** `is_goal(state)`: Checks if the State is a solution to the Latin Square completion problem. Note that if there is any unassigned variable, it's definitely not a completed Latin Square; you don't have to keep checking if there are any variables still unassigned. You could adapt your code for `A3Q1.is_goal(state)` to implement this function. You may need to combine the information contained in the given "square" with the values assigned in your state.
- **Problem** `actions(state)`: Here, choose an unassigned variable and create an action for each allowable domain value. But don't actually make the assignment yet. That's still the work of `result()`
- **Problem** `result(state, action)`: Creates a new State, with the variable assigned the given value, as provided by `action()`. Make sure you copy the variables appropriately, and change the copy.

Notes:

- Your Problem class constructor should take a "square" as an input argument, and from that square. You can have another method called `initial_state()` that uses the square to create a State object.
- For this question, the domain for every blank cell in the square should start out with all the values $1, \ldots, N$, i.e., don't remove values that appear on the same row and column. We'll fix that in the next question.
- In class we said it was possible to represent this problem 2 ways: Every cell (filled or blank) could be a variable; or only the blank cells are variables. **It's slightly more advantageous to make variables for only the blank cells. It complicates the goal test a bit.**
- The `is_goal(state)` function is also a bit hard to get right, but test it with examples, not by applying your search algorithms.

- The `action()` and `result()` functions should be straight-forward, once you have everything else settled.
- Test your `is_goal(state)`, `action()`, and `result()` functions before trying them out with search. A test script that you can re-run when you make changes to your code will save you lots of time!

Questions to answer

Apply your implementation to the examples in the given data file `LatinSquares.txt`. Use a time-limit of 10 seconds for each problem.

Answer the following questions and submit them:

1. How many of the problems did your implementation solve within 10 seconds? How does this compare to the previous implementation (A3Q1)?
2. Consider the largest N that your implementation as able to solve. Do a rough, back-of-the-envelope calculation to predict how much time it would take to solve the *next largest* problem in the file, and the *very largest* problem in the file. Submit your rough calculations, and your predictions.

What to Hand In

- A file named `a3q2_EXECUTION.txt` containing brief instructions for compiling and/or running your code. See page 3.
- A file named `a3q2.LANG` containing your implementation of the State and Problem classes. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`.
- A file named `a3q2.txt` (other file formats like PDF, DOC, DOCX, RTF are also acceptable) with the responses to the above questions.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 6 marks: Your implementation of the State and Problem class uses a collection of variables and domain values. Your program does not have to solve all problems.
- 9 marks. Your answers to the questions are plausible and supported by evidence.

Question 3 (5 points):

Purpose: To demonstrate the power of the very simplest of improvements to the previous implementation.

Degree of Difficulty: Easy.

In the previous question, the domains were deliberately left as the values $1, \ldots, N$. In this question, change your problem class code so that each variable's domain is restricted to value that do not appear on the cell's row and column. For example, consider the top left blank in the following square:

```
_ 2 _ _ 4
3 _ 1 _ _
_ 4 _ _ 1
2 _ _ _ 3
_ _ 2 1 _
```

Since the values 2, 4 appear in the row, and the values 2, 3 appear in the column, the cell cannot take those values, and so the domain for the top left cell should be limited to just $\{1, 5\}$.

Implement this restriction by creating variables whose domains are all feasible just looking at the row and column (probably in your `initial_state()` method). Don't change anything else.

Questions to answer

Apply your implementation to the examples in the given data file `LatinSquares.txt`. Use a time-limit of 10 seconds for each problem.

Answer the following questions and submit them:

1. How many of the problems did your implementation solve within 10 seconds? How does this compare to the previous implementation (A3Q2)?
2. Consider the largest N that your implementation as able to solve. Do a rough, back-of-the-envelope calculation to predict how much time it would take to solve the *next largest* problem in the file, and the *very largest* problem in the file. Submit your rough calculations, and your predictions.

What to Hand In

- A file named `a3q3_EXECUTION.txt` containing brief instructions for compiling and/or running your code. See page 3.
- A file named `a3q3.LANG` containing your implementation of the State and Problem classes. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`.
- A file named `a3q3.txt` (other file formats like PDF, DOC, DOCX, RTF are also acceptable) with the responses to the above questions.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 2 marks: Your implementation of the domain restrictions is correct. Your program does not have to solve all problems.
- 3 marks. Your answers to the questions are plausible and supported by evidence.

Question 4 (15 points):

Purpose: To implement the Constraint Satisfaction Problem concept of forward checking.

Degree of Difficulty: Easy.

Forward checking is a process of removing values from the domains of some unassigned variables, if those values can be determined to be inconsistent with the current partial assignment.

The way it will work in the LSCP is as follows: When a cell X is assigned the value v, we know that no other variable in X's row or column can also have the value v. So, if the value v is still a domain value in any unassigned variable on X's row or column, we can remove it in advance, so that it is never tried.

There are some complications.

1. The value v might not appear in the domain when you try to remove it. That's perfectly fine; it might have been removed for some other reason. This removal is more like a prevention. Only remove it if it is still there.
2. Don't try to remove a domain value from an assigned variable.
3. We're exploring states that represent different partial assignments. You really need to make sure that your states and domain values are copies, and not references to a common list. When you remove a domain value in one state, it can't affect other states by accident!
4. It could be that you'll remove the very last possible domain value for some unassigned variable. When that happens, there is no consistent assignment for that variable, and so the current partial assignment (the state) is inconsistent. The best thing to do is to recognize when a domain goes empty as you're removing the value. The easiest way to manage it is to have a Boolean flag in your state for consistent or inconsistent assignments. Set the flag as soon as possible.

Starting with A3Q3, implement the forward checking for LSCP:

- **State**: Include a boolean flag for to indicate if the state is known to be inconsistent.
- **Problem** `is_goal(state)`: With forward checking, every consistent partial assignment can be extended by one more variable (at least). That means when the assignment is complete (no unassigned variables), it must be consistent! This simplifies the goal test substantially!
- **Problem** `actions(state)`: Check if the current state is inconsistent. If it is not consistent, return an empty list of actions. This prevents inconsistent states from having children.
- **Problem** `result(state, action)`: The action is a variable X and a domain value v. Copy the state as usual, make the assignment, and then remove the value v from the domain of any unassigned variable in X's row and column. See the note above about removing values.

Questions to answer

Apply your implementation to the examples in the given data file `LatinSquares.txt`, using a time-limit of 10 seconds for each problem in this file. Then apply your implementation to the examples in the given data file `harder_examples.txt`, using a time-limit of 200 seconds for each problem in this file.

Answer the following questions and submit them:

1. How many of the problems did your implementation solve within the time limit? How does this compare to the previous implementation (A3Q3)?

What to Hand In

- A file named `a3q4_EXECUTION.txt` containing brief instructions for compiling and/or running your code. See page 3.
- A file named `a3q4.LANG` containing your implementation of the State and Problem classes. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`.
- A file named `a3q4.txt` (other file formats like PDF, DOC, DOCX, RTF are also acceptable) with the responses to the above questions.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 10 marks: Your implementation of the domain restrictions is correct. Your program does not have to solve all problems.
- 5 marks. Your answers to the questions are plausible and supported by evidence.

Tasks for the ambitious

It seems that forward checking is a big improvement, but it has its limits. It works really well on toy kinds of problems (Latin squares, N-Queens). However, most CSP solvers use a stronger form of processing called **arc consistency**, which has higher complexity than forward-checking, but usually results in much more significant domain reductions for most real-world problems. Implement a version of the AC-3 algorithm. You can use it in `initial_state()` to make sure that the initial domains for your problem start off possibly highly reduced even compared to A3Q3, and in `result()` as in A3Q4, to keep your domains even smaller than forward-checking. The bigger your problem, the more the extra processing pays off!