Artificial Intelligence	**Programming Assignment 3**
Comp440 Fall 2007	**9 October 2007**

This program is due October 25 in class.

For this problem, you will use constraint satisfaction search to play the Minesweeper game.

Minesweeper

In this game, you start with a grid of blank squares, some of which conceal mines. In each move you uncover one square. If it contains a mine, it explodes and you lose the game. Otherwise a number is revealed, telling you how many mines there are in the eight adjacent squares. This information can help you decide which square to uncover in the next move. When you infer from the revealed numbers that a square contains a mine, you can mark it with a flag. The total number of mines M is known in advance, and if you uncover all but M squares in the grid without exploding a mine, you win the game. Flagging the mines is thus optional. Whenever a square is uncovered that has no adjacent mines, it is displayed as blank, and all of its neighbors are automatically uncovered for you.

CSP formulation of Minesweeper

Minesweeper can be formulated as a CSP, where the variables correspond to covered squares, whose values are in $\{1, 0\}$. 0 denotes a safe square, and 1 denotes a square with a mine in it. Each uncovered square represents a constraint on the possible values of any neighboring squares that are still covered.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	1					
2	0	0	0	0	2					
3	1	1	1	0	2					
4			1	0	1					
5			3	2	2					
6										
7										
8										
9										
10										

As an example, consider the 10x10 board shown above with a total of 20 mines. We identify each square by its (row,column) coordinates. The top left hand corner is (1,1). In this example, the squares (1,1), (1,2), (1,3), (1,4), (2,1), (2,2), (2,3), (2,4), (3,4) and (4,4) have already been played and have no mine in their respective surrounding squares. The squares (1,5), (3,1), (3,2), (3,3), (4,3) and (4,5) have been played too, and are surrounded by one mine each. The squares (2,5), (3,5), (5,4) and (5,5) each have two mines in their neighborhood, while the square (5,3) has three mines around it. At any point in the game, the set of squares that are still covered, not flagged as mines, and adjacent to an uncovered square are the *fringe*. The simplest CSP formulation has boolean variables x_{ij}, denoting fringe squares at row i and column j. In our example, there are thirteen fringe variables: x_{16}, x_{26}, x_{36}, x_{46}, x_{56}, x_{66}, x_{65}, x_{64}, x_{63}, x_{62}, x_{52}, x_{42}, and x_{41}. In the figure we show the number of mines revealed in each square that has been uncovered. For each uncovered square adjacent to a fringe square, we have the constraint that the number revealed in that square is the sum of the mines in the adjacent uncovered squares. This yields the following 11 constraints, one for each uncovered position adjacent to a fringe square.

Name	Square	Constraint
C_1	(1,5)	$(x_{16}, x_{26}) \in \{(0,1),(1,0)\}$
C_2	(2,5)	$(x_{16}, x_{26}, x_{36}) \in \{(0,1,1),(1,0,1),(1,1,0)\}$
C_3	(3,5)	$(x_{26}, x_{36}, x_{46}) \in \{(0,1,1),(1,0,1),(1,1,0)\}$
C_4	(4,5)	$(x_{36}, x_{46}, x_{56}) \in \{(0,0,1),(0,1,0),(1,0,0)\}$
C_5	(5,5)	$(x_{46}, x_{56}, x_{64}, x_{65}, x_{66}) \in \{(1,1,0,0,0),(1,0,1,0,0),(1,0,0,1,0),(1,0,0,0,1),(0,1,1,0,0),$ $(0,1,0,1,0),(0,1,0,0,1),(0,0,1,1,0),(0,0,1,0,1),(0,0,0,1,1)\}$
C_6	(5,3)	$(x_{42}, x_{52}, x_{62}, x_{63}, x_{64}) \in \{(1,1,1,0,0),(1,1,0,1,0),(1,1,0,0,1),(1,0,1,1,0),(1,0,1,0,1),$ $(1,0,0,1,1),(0,1,0,1,1),(0,1,1,1,0),(0,1,1,0,1),(0,0,1,1,1)\}$
C_7	(5,4)	$(x_{63}, x_{64}, x_{65}) \in \{(0,1,1),(1,0,1),(1,1,0)\}$
C_8	(4,3)	$(x_{42}, x_{52}) \in \{(0,1),(1,0)\}$
C_9	(3,3)	$x_{42} \in \{1\}$
C_{10}	(3,2)	$(x_{41}, x_{42}) \in \{(0,1),(1,0)\}$
C_{11}	(3,1)	$(x_{41}, x_{42}) \in \{(0,1),(1,0)\}$

As you can see from the structure of the constraints, they are all *m_of_n* constraints. That is, they all state that m out of the n variables are ones, the rest are zeros. You can use this fact in designing your constraint representation and in devising algorithms for checking constraints.

The constraint on the total number of mines is easy to express algebraically as

$$\sum_{i,j \in \{1 \ldots 20\}} x_{ij} = 20, x_{ij} \in \{0,1\}$$

where 20 is the total number of mines in some instance of the problem.

Search Algorithm

For the first part of the problem, we will consider the following approach to playing the game. Possible assignments of the fringe squares to values in $\{0,1\}$ constitutes the search space. In our example, it is a space of size 2^{13} because there are 13 fringe variables.

A key step in setting up the CSP for a grid state is to assert all the constraints above. Of course, finding a single solution to the CSP containing the fringe variables is not enough. Before we uncover a square, we want to be certain that it is clear of mines in ALL possible solutions to the CSP. Otherwise, we run the risk of being blown up. In our example, we have $x_{42} \in \{1\}$, which means that x_{42} has a mine in all solutions to the CSP. Thus, unlike most CSP problems, which look for a single solution and stop, we examine all possible assignments of values to the fringe variables to find all the ones that satisfy the constraints. This set of possible solutions is then examined for squares that are mines in every solution, or clear in every solution. Such squares can then be flagged and uncovered, respectively. In some situations, there will be no square that is clear in every solution. Then, there is no fringe square that can be safely uncovered. In this case, your program has to guess, by uncovering a square that might be clear.

You will implement and test a series of increasingly efficient searches; extra credit extensions will let you try to improve the efficiency even more, or to improve the performance of guessing. The initial, very inefficient, algorithm is:

```
Uncover the upper-left square at position (1,1).
Repeat
   For the current grid state, assert all constraints on the fringe variables.
   Perform a depth-first search of the space of all assignments to fringe variables.
      For each complete assignment, test whether all constraints are satisfied.
      If the assignment satisfies all constraints,
         add it to the set of possible solutions.
   Flag known mines and uncover squares known to be clear.
   If no squares are known to be clear, uncover a guess and check whether game is lost.
Until game is won or lost
```

The Task

1. (10 points) **Asserting constraints**. Given a Minesweeper board, design and build data structures that contain all constraints on the fringe variables as well as the sum of mines constraint. You should test your code using a random 4x4 board with 4 mines, with the square (1,1) uncovered.

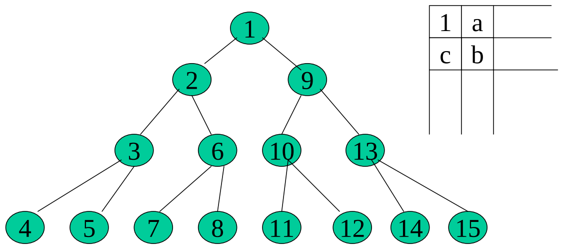
2. (10 points) **Depth first-search**. Build a depth-first searcher that examines every assignment of value in $\{0,1\}$ to the fringe square variables. This depth-first search should simply enumerate all the possible assignments of values for a given number of fringe variables. You need to decide on a fixed ordering for the fringe variables. Test your code on a random 4x4 board with 4 mines with square (1,1) uncovered.

   As an example, consider the portion of the 4x4 board shown in Figure 1, with fringe variables a, b, c. The tree shown in this figure depicts the order in which a depth-first searcher explores all assignments of values to the fringe variables. The fixed order chosen is a, b, c.

3. (10 points) **Testing constraints I**. Implement a function `MeetsConstraint` that tests whether an instantiated fringe satisfies a given constraint. The inputs to this function are an instantiated fringe and a constraint; the output is a boolean which says whether the fringe satisfies the constraint. For example, given your representation of the fully instantiated fringe ($x_{16} = 1, x_{26} = 0, x_{36} = 0, x_{46} = 0, x_{56} = 0, x_{66} = 0, x_{65} = 0, x_{64} = 1, x_{63} = 0, x_{62} = 1, x_{52} = 1, x_{42} = 1, x_{41} = 0$) and your representation of the constraint C_1, this function must return true, since the pair $x_{16} = 1, x_{26} = 0$ is consistent with C_1.

4. (5 points) **Testing constraints II**. Extend this function to build `MeetsConstraints` which checks that an instantiated fringe satisfies *all* constraints in a specified set of constraints. This simply requires iterating over a set of constraints, calling `MeetsConstraints` on each one, and returning the boolean AND of them all.

5. (10 points) **Testing constraints III**. To make the next parts easier to implement, extend `MeetsConstraint` and `MeetsCOnstraints` to work with partially instantiated fringes. A



Node Name	Assignments
1	$\{\}$
2	$\{a = 0\}$
3	$\{a = 0, b = 0\}$
4	$\{a = 0, b = 0, c = 0\}$
5	$\{a = 0, b = 0, c = 1\}$
6	$\{a = 0, b = 1\}$
7	$\{a = 0, b = 1, c = 0\}$
8	$\{a = 0, b = 1, c = 1\}$
9	$\{a = 1\}$
10	$\{a = 1, b = 0\}$
11	$\{a = 1, b = 0, c = 0\}$
12	$\{a = 1, b = 0, c = 1\}$
13	$\{a = 1, b = 1\}$
14	$\{a = 1, b = 1, c = 0\}$
15	$\{a = 1, b = 1, c = 1\}$

Figure 1: A board with the (1,1) square revealed. There are three fringe variables, a, b and c. The tree shows the order in which the assignments are explored. The fixed variable ordering is a,b,c. The table shows the assignments associated with the numbered nodes.

partially instantiated fringe is one in which the values of a strict subset of the fringe square variables are known. Test your code on a random 4x4 board with 4 mines and square (1,1) uncovered.

6. (20 points) **Consistency checking and backtracking**. The next step is to implement consistency checking to avoid useless search along paths that are already inconsistent. Modify your depth-first search code to call `MeetsConstraints` on partial assignments and not generate any successors for a node which violates any of the known constraints. Run your new consistency checker on the given boards and report on the number of nodes expanded by depth-first search for a random 4x4 grid with 4 mines.

7. (20 points) **Forward checking**. Now you can look a little forward and eliminate search paths where some variables have no legal values.

   In our running example, C_1 has two variables associated with it: x_{16} and x_{26}. Implement the function `GetVarDomain` that takes a variable and a constraint and finds all values of that variable that are consistent with that constraint. For C_1 and x_{16}, this function returns the set $\{0, 1\}$. Extend this function to implement `GetVarDomains` which takes a variable and a set of constraints and returns the set of values for that variable consistent with all constraints. For C_9 and C_{10} and variable x_{41}, your function should return the set $\{0\}$. This is an instance of constraint propagation, where we have inferred that $x_{41} = 0$ from C_9 and C_{10}. Extend the `MeetsConstraints` function to check the domains of all uninstantiated fringe variables, and to return false if any of them become empty.

   Modify your DFS searcher to do forward checking at each step. Report on the number of nodes expanded in the quest for all solutions to the set of constraints, for a random 4x4 board with 4 mines.

8. (10 points) **Dynamic variable ordering**. Modify your DFS searcher so that instead of picking the first uninstantiated fringe variable on its list (based on your pre-selected fixed ordering), it uses the minimum remaining values (MRV) heuristic to pick the next fringe variable to instantiate. Report on the number of nodes expanded while searching for all solutions to the set of constraints, for a random 4x4 board with 4 mines.

9. (20 points) **Performing actions to get new information**. Sometimes you just have to gather more information to be able to locate the mines. Once a fringe square is guaranteed to be clear, you should uncover it and see if it gives you new constraints. Add a new function `Propagate` that, given a partially-assigned fringe, checks the domains of all unassigned fringe squares using `GetVarDomains`, and instantiates any variable whose domain is reduced to a single value. `Propagate` should iterate until no new singleton variable is found, in which case it returns with a modified fringe and constraints. `Propagate` should be invoked right after the constraints are asserted and before DFS is started. If any squares are found to be mines, flag them. If any of the squares are found to be clear, uncover them and go back to asserting new constraints, if any. If no mines or clear squares are found, continue as usual with the search. How many nodes are expanded by DFS with this method on a random 4x4 board with 4 mines?

10. (40 points) **Game Play**: Test your player with all of these embellishments on 10 20 by 20 random boards at mine densities from 5 to 25 percent. Report how many of them it played successfully and how many nodes were expanded during search. For the same boards, report what percentage of the games were won by the provided Random Player.

11. (20 points) **Improved guessing: extra credit**. Try to improve the performance of guessing when no square can be safely uncovered. One obvious approach is to pick a fringe square with the highest fraction of 0 values out of all the possible solutions found. Another approach is to pick a non-fringe square when all the fringe squares appear too risky to uncover. Would it matter which non-fringe square is chosen? Explain why or why not. Run your improved guesser on a number of random boards, and see how many it is able to solve compared to a naive guessing policy.

Code and Javadocs

You can download the code from `http://www.owlnet.rice.edu/~comp440/code/pa3.tgz`. The javadocs can be found at `http://www.owlnet.rice.edu/~comp440/pa3-javadoc/`. You must wrap your solver into the SmartPlayer class for this assignment.

Hints and Tips

1. **Read the full assignment first** - This assignment is laid out as iterative improvements to a Mine Sweeper solver. Do not implement it that way though. Understand the entire assignment (and get any questions answered) before you start writing one line of code. As an example, one of the last steps is variable re-ordering. It will be easier for you to make such re-ordering possible from the beginning in terms of the data structures you use.

2. **Spend time comparing algorithm improvements** - A really neat trick is to fix the seed of the random generator in the mine generation component. This will cause it to generate the

exact same boards each time. This will allow you to compare and contrast various versions of your solver. While you have fixed ordering, it will even allow you to trace the exact depth-first-search trees.

3. **Implement a configurable SmartPlayer** - Setup your SmartPlayer to be configurable as to which algorithmic improvements it implements. For example, you could pass a bit vector to turn on or off dynamic reordering, forward checking, etc.. This will make comparing improvements easier.

4. **Understand forward checking** - The class notes and book talk about performing forward checking only after a move has been made. In general, this is when you would normally apply forward checking. However, you can also apply forward checking to a set of constraints at anytime, as long as you remember that forward checking is *not an iterative process*. For example, you could put all the variables on a worklist and process them one at a time. If a variable X has a domain of 1, then you can propogate this information to X's neighbors. You may not put any variable back on the worklist, however! Using this technique is how you would apply forward checking to the homework. Do you see why forward checking may not catch all cases in which the constraints are violated? Does the order you process the nodes in make a difference? What if your initial worklist only contains nodes that start with a domain of 1, instead of all the nodes?

   On a side note: arc consistency is a similar process except that variables are allowed to be put back on the worklist if their domain changes. How does this let arc consistency catch more cases than forward checking? Hint: arc consistency is an iterative process, becuase nodes can be processed more than once; the algorithm only stops when a fixed point is reached. What cases does arc consistency miss? See your book for more information about k-consistency, a more general form of arc consistency. What is the advantages and disadvantages between using a more powerful consistency checker vs. just doing the search