CS 440/ECE448: Introduction to Artificial Intelligence

Assignment: Machine Problem 1 **Due date:** February 17, 2011

The goal of this assignment is to demonstrate two techniques for solving Sudoku: recursive depth-first search and AC-3 (a constraint-satisfaction technique which will be covered in Lecture 5), and to explore their relative merits and trade-offs. Both techniques are general and not restricted to Sudoku, but, as a puzzle, Sudoku provides a natural domain for demonstrating them.

General instructions

You need to implement two search algorithms. In both cases, you will be graded according to 1. whether you implemented the algorithms correctly, 2. whether you have provided sufficient comments in your code, and 3. your high-level explanations of the algorithms.

1. Programming Your task is to implement depth-first search and AC-3 in `Sudoku.java`. You need to download this file from

`http://www.cs.illinois.edu/class/sp11/cs440/Sudoku.java`

Instead of using a queue, both algorithms utilize a recursive method which must return `true` if a solution exists and `false` otherwise.

When writing your code, you must store the final solution in the `int[][]vals` data-structure.

Your code needs to go into the places that we have indicated in the file. You are not allowed to change any other methods. All necessary libraries have already been imported. If you choose, you may *only* import and use other standard Java libraries. All code must be able to be compiled and run on the departmental machines. Any general questions about the assignment or the Java programming language should be addressed to the newsgroup (`class.sp11.cs440`).

Compiling and running the code To compile the code, open a terminal in the directory of `Sudoku.java` and execute:

```
$ javac Sudoku.java[1]
```

To run the code, you need to tell Sudoku which board it should attempt to solve. This is the first argument: *easy, medium, noSolution, hardNoSolution, random*. The first four are hard-coded boards, which you can use for testing and debugging. They will also be used during grading. Please

[1]If the command javac returns a "command not found" error, you need to ensure you have installed the Java Developers Kit (JDK) which can be found at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

test your code on these boards but do not necessarily expect your code to terminate in all cases. Specifically you should give serious thought to how the two algorithms behave in the context of the two boards which lack a solution.

The second argument (*DFS, AC-3*) specifies the search algorithm that will be used to solve the board. The code provides you the option of working either with or without a GUI (see Figure 1). If you don't provide a second argument, the GUI will start, and you have to set the search algorithm using the GUI (see figure 1).

To run the code with a GUI, execute the following command in the same terminal (choosing one of the boards given in curly braces):

```
$ java Sudoku {easy|medium|noSolution|random}
```

e.g. to run the easy board:

```
$ java Sudoku easy
```

To run the code without a GUI, execute the following command (again, choosing one of the boards given in curly braces, and additionally, one of the search algorithms):

```
$ java Sudoku {easy|medium|noSolution|random} {DFS|AC-3}
```

for example:

```
$ java Sudoku easy DFS
```

2. Code comments In your own words, you must clearly comment every part of the algorithms. You will be graded on whether we understand your source code without too much additional effort or not.

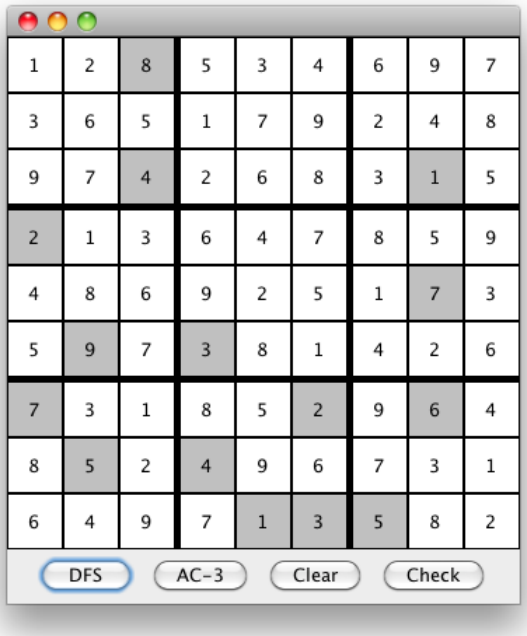
3. Explanations of the algorithms Additionally, there are designated comment sections in the java file where you must provide high level explanations of the algorithms and their respective strengths and weaknesses. These are above the recursive function definition for each algorithm.

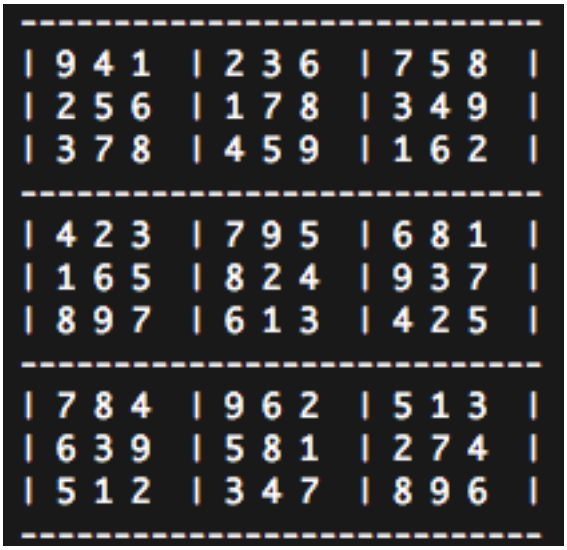
Specifically address why the algorithms find different problems "easy" or "hard" and explain the behaviors you notice when tackling boards that do not contain a solution. Proper documentation and discussion are required for full credit.

Hint: Recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem[2]

When implementing DFS with a queue, your data-structure grows as you continue to search. Every node in the search tree corresponds to a partial or completed Sudoku board. This means in practice you must make a copy of the board and save it to the queue. For many problems,





`$ java Sudoku easy` `$ java Sudoku easy DFS`

Figure 1: Run Sudoku board

Algorithm 1 Recursive DFS (cell)

for all valid values v for $cell$ **do**
  $cell \leftarrow v$
  if Recursive DFS($cell + 1$) **then**
    return true
  end if
end for
$cell \leftarrow \emptyset$
return false

this memory requirement makes implementation impractical. For this reason, we expect you to implement it recursively as in algorithm 1.

The Recursive DFS should follow conceptually from the definition of recursion cited above. An assignment to cell 0, the top left most cell, is correct if and only if that assignment allows for proper assignments of all other cells $[1, 80]$ (i.e. the remainder of the board can still be completed). We therefore start the search with an empty board, move to a board with a single 1 in the first cell $[1, ...]$, and then traverse (recursively) the left most branch of the search tree, assigning 1 to the next cell, and so forth. Because this board $[1, 1, ...]$ is invalid, the recursion returns false, leading the search down the next closest path $[1, 2, ...]$. By having the recursion trace out the search, we

[2] `http://en.wikipedia.org/wiki/Recursion_(computer_science)`

only require one board which is consistently edited to match every node in the search, making implementation tractable.

Task 1: Depth First Search (DFS) - 4.5pts

You must implement the function `RecursiveDFS(int cell)` which will be called initially on cell 0, the top left-most cell. Recursive DFS makes an assignment and then returns true if this assignment, plus the recursive call on the next cell creates a valid assignment.

Your `RecursiveDFS(int cell)` must return `true` if and only if it has discovered a valid assignment for cells `[cell, 80]` (i.e. all cells past and including the current one) and returns `false` otherwise.

To validate a move, you should call `valid(x,y,v)` which returns true if placing value v in cell (x, y) doesn't conflict with other values.

Task 2: Arc Consistency Algorithm #3 (AC-3) - 5.5 pts

You must implement the functions:

- `AC3_DFS` DFS with calls to `AC3` instead of `valid`
- `allDiff` explained in the book
- `AC3` pseudocode in book [3]
- `Revise` pseudocode in book

Here AC-3, covered in the book, wikipedia and Lecture 5, makes decisions (cell assignments) and then propagates the implications and restrictions imposed to all other cells on the board. Because we cannot guarantee there is only a single solution, you will perform the same Depth First Search but instead of checking if a move is valid, you will check if the current configuration still allows for Constraint Satisfaction to produce a valid board. This will be done by calling `AC3` rather than `valid`.

There are several data structures we believe will be useful for these functions and are therefore left in the code but you are not required to use them. These include:

- `globalDomains` provides the set of values for every cell
- `neighbors` the set of neighbors for every cell (as defined by the Arcs)
- `globalQueue` which is the initial set of arcs for AC-3
- `Arc` a simple tuple class for storing the indices of two cells

Again, the only **strict requirement** is that, once again, `AC3_DFS` return `true` if a valid board was discovered and `false` otherwise, with the final board stored in `int[][] vals`.

[3] Page 209: Available on Google Books: `http://books.google.com/books?id=8jZBksh-bUMC`

Submission

You will submit to compass the edited *single java file*. **Reminder: Please ensure you have tested your code thoroughly, have documented all code you have written and have provided the proper analysis of both algorithms. Please ensure your code compiles.**

You will submit your code on compass.

1. Login to `https://compass.illinois.edu`
2. Click Assignments
3. Choose MP1
4. Upload `Sudoku.java` only.

Extra fun ...

The following is not for extra-credit BUT if you find AC3 interesting but feel like there's something lacking, we recommend you read up on or implement (just for fun, on your own time, not for extra credit):

simple Backtracking search: in the textbook, or
`http://en.wikipedia.org/wiki/Backtracking`

expert Sinkhorn solves Sudoku `http://portal.acm.org/citation.cfm?id=1669471`