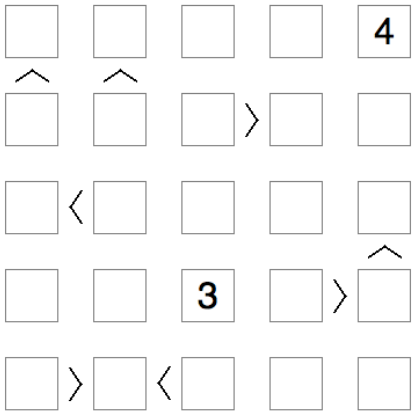
159.271 Computational Thinking

Assignment 1: Futoshiki Solver



This assignment is **now worth 15%** of your final mark. It will be marked out of a total of 25 marks. The due date for this assignment is now **Monday May 4th**. You are expected to work on this assignment *individually* and all work that you hand in is expected to be your *own* work.

In this assignment your task is to write a Futoshiki solver. You will develop an animated application that uses recursive backtracking as the basis of the implementation. A Futoshiki puzzle is similar to a Sudoku puzzle. The input to the problem is a square board having a given fixed size (5x5 for example), with some of the board cells containing digits between 1 and the board's size, and with some inequality constraints between adjacent pairs of cells on the board provided. When complete, every cell on the board must be filled with a digit between 1 and the board's size. For each row and column, each digit must appear exactly once. In addition, the choice of digit for each cell must respect any inequality constraints between adjacent pairs of cells. See http://www.futoshiki.org/.

Go to Stream and download the file

`Futoshiki.zip`

The unzipped folder contains a template program that implements the code needed to read Futoshiki puzzles from files and to display them, and to run the solver in animation mode. Set up a new project and add the folder **`src`** to it. The main game module is **`FutoshikiApp.py`**. If you run this, a screen will appear and clicking on one of the **`'t'`**, **`'e'`** or **`'h'`** keys will display a randomly chosen puzzle (trivial, easy or hard) from one of the three folders of puzzles included. Note that the trivial puzzles do not have any inequality constraints.

You need to complete the module **`Solver.py`**, so that a call to the **`solve`** function will do a bit more than just displaying the puzzle. You need to turn this function into a recursive backtracking solver, which, for animation purposes, displays a snapshot of the current state of the puzzle at the start of each recursive call.

The program uses two classes to represent a Futoshiki – **`Cell`** and **`Snapshot`**. **`Cell`** represents a single cell on a Futoshiki board, and has methods to get and set the position of the cell and the value of the cell. For the purposes of this assignment, in any complete solution, the value must be between 1..5. If the value is 0 this means that the cell is still empty. **`Snapshot`** describes a state of the Futoshiki at a certain point in time – some cells have values (other than zero), some may not. The **`Snapshot`** class has methods that allow to clone a snapshot (this is required to produce the next level of snapshots in the recursion tree), to query the cells in various ways, and to get and set cell values. It also has a method **`getConstraints`** that returns a list of the inequality constraints for the board. Each item in the returned list has the format **`((i,j),(k,l))`** indicating that the value in cell **`(i,j)`** must be smaller than the value in cell **`(k,l)`**.

`Futoshiki_IO.py` contains the code used to read Futoshiki puzzles from puzzle files and to display them. The puzzle files consist of 5 lines containing 5 digits each, encoding the initial values for the board cells, followed by some number of lines containing 4 digits each, encoding the inequality constraints for the board. Board cells are indexed from 0 to 4, so the line **`0 2 0 3`**, for example, encodes the inequality constraint **`cell(0,2) < cell(0,3)`**.

In general, to develop a recursive backtracking algorithm you need to consider:

1. What question to ask? Thinking only about the top level of recursion, you need to formulate one small question about the solution that is being searched for. Remember that, in order for the final algorithm to be efficient, it is important that the number of possible answers is small.
2. How to construct a subinstance? How to express the problem as a smaller instance of the same search problem?

For your Futoshiki solver, an input instance for the algorithm will be a **`snapshot`** specifying both the current state of the Futoshiki board and the provided inequality constraints. A solution is a valid way of filling the remaining empty cells. The simple question to ask is "What number can go in the next empty cell?"

The brute force strategy is simply to find the next empty cell in the snapshot, working left to right, top to bottom, and then to try all possible numbers in that cell, with a recursive call to the **`solve`** function for each possibility. Initial pruning of the recursion tree should include two strategies - 1. we don't continue on any branch that has already produced an inconsistent solution and 2. we stop once a complete solution has been found. Python code examples that implement this basic strategy for two straightforward problems have been given in the lecture slides.

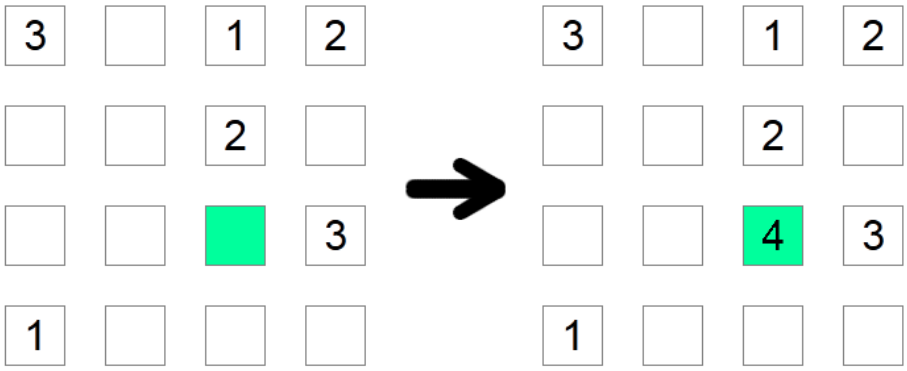
given in the lecture slides.

Remember that the tree of recursive calls will be traversed in depth-first order. A branch of the tree will be recursively explored until either we establish that the branch cannot lead to a complete solution - in which case we backtrack up the tree and try the next possible branch, or we find a complete solution. The brute force approach can lead to a very large tree of recursive calls. Suppose only a couple of the input board cells are already filled, and perhaps four inequality constraints are provided (this can be enough to provide a unique solution), in this case we might need to search through 5^{23} different combinations!

You are going to need a smarter approach, so think about the following ideas and develop pruning rules based on these.

1. **`Column and row exclusions`**

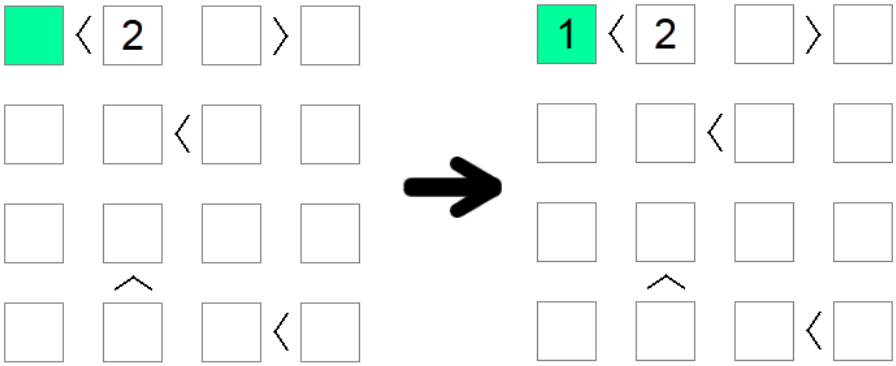
   If a cell's column and row already contain all possible digits, except one, then that cell must contain the missing digit. In the example below, the green cell must be 4. This cell is a 'singleton'. To find singletons, for each cell you need to keep a list of which digits are allowed within it, and then check the relevant row and column to remove those digits that have already been used. Of course, if you end up in a situation where there are no possible digits that can go in a cell then you've done something wrong, so your algorithm should stop and backtrack. You will need to add attributes and methods to the **`Cell`** class to implement this strategy. If you can't find any singletons, then it is a good idea to choose a 'doubleton' as the next cell to try, that is, one that has only two possible digit choices left.



2. **`Forced min and max values`**

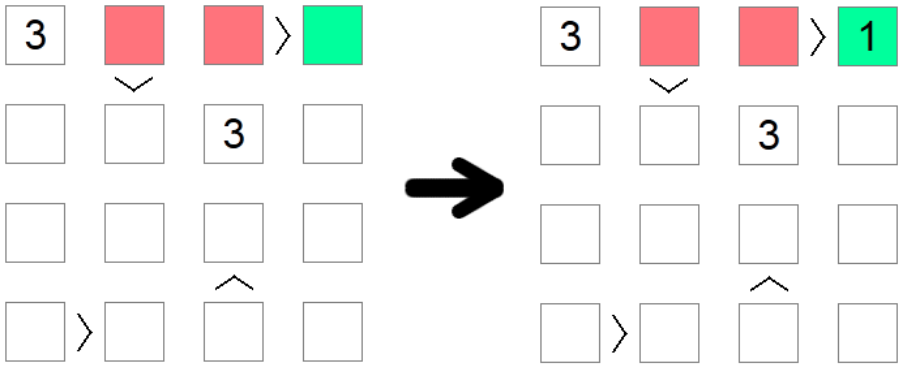
   Cells that are less than 2 must implicitly have the value 1 as it is the only admissible value on the board which respects that condition. Similarly, cells that are greater than

the board size minus 1 must be equal to the board size. In the example below, the only possible value for the green cell (less than 2) is 1.



3. **Exclusion of min and max values**

Cells that are greater than other cells cannot be 1, the lowest value allowed on the board, as there is no value smaller than 1. Similarly, cells that are less than than other cells cannot contain the max allowed value, as there would be nothing greater to be filled on the other side of the inequality. In the example below, 1 cannot be filled in the red squares as they are all greater than other board squares, so the only possible placement for 1 on the first row of the board is the green square.

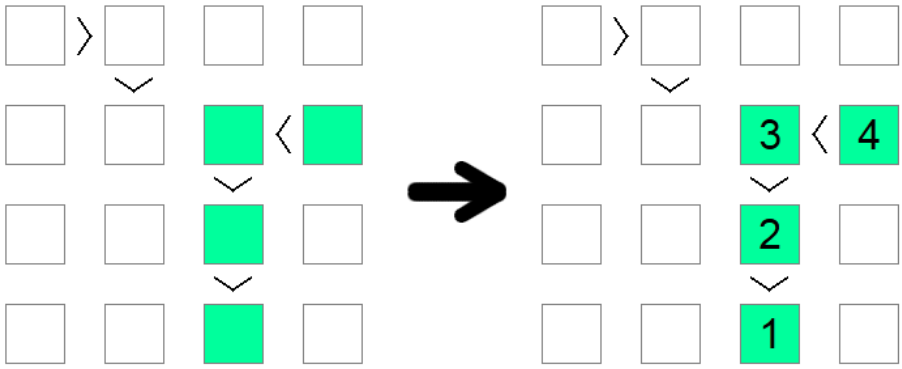


To find forced cell values, or excluded cell values, you need to inspect the inequality constraints and remove digits from your 'possibles' lists for certain cells, according to what you find.

4. (Harder to implement) **Chains of inequalities**

If you notice a chain of inequalities, be it either < (all ascending) or > (all descending), equal in size with the board's size, then that chain must be a sequence from 1 up to the length of the board. The length of the chain guarantees that this sequence is the only

possible solution that satisfies the monotone condition imposed by the inequality chain.



5. There are other intelligent things that can be done. You can 'cross-reference' between the 'possibles' list for a cell, and the 'missing' values in the relevant row and column. If there is only one value that is common to all sets, then this value must go in the cell.

Marking scheme

1. 10 marks for a correct program that uses a **`recursive backtracking`** strategy with basic pruning.
2. 5 marks for adding the facility to choose 'singletons' ahead of other cells, using an approach that inspects cell values. With this approach, your solution should be able to solve the trivial puzzles.
3. 5 marks for finding 'forced' and 'excluded' cell values, using an approach that inspects inequality constraints. With this approach, your solution should be able to solve the easy puzzles.
4. 5 marks for more sophisticated pruning strategies, either those described here or others, that allow for fast backtracking over hard puzzles.

We will test your program using puzzles similar to those provided, for trivial, easy and hard categories.

Submission

Submit your project in Stream as a single zipped file containing your completed code, contained in the folder **`Futoshiki_YourID`**.

Plagiarism

There are many Futoshiki solvers available on the internet (and many more Sudoku solvers). I am sure that many of these are recursive backtracking solutions implemented in Python.

For this reason, and also so that the concepts taught in lectures are reinforced, you must use the template provided as the basis of your implementation. You can improve the display or the animation however you like, you can add more classes or improve those provided (this part you actually need to do), you can make the solver as sophisticated as you like. **What you cannot do is download a solution from the internet and submit it as your own work.** If you try this, **you will get no marks**. Make an attempt to read the course notes, to follow the lecture slides and to apply the concepts discussed, in completing this assignment.

As stated at the beginning of this document, you are expected to work on this assignment *individually* and all work that you hand in is expected to be your *own* work. You are allowed to discuss any aspect of this assignment with others, in person, or on the Stream forum. **What you are not allowed to do is to submit someone else's solution as your own work.** If it appears that this is the case, **you will get no marks**.

Late submission:

Late assignments will be penalised 10% for each weekday past the deadline, for up to five (5) days; after this no marks will be gained. In special circumstances an extension may be obtained from the paper co-ordinator, and these penalties will not apply. Workload will not be considered a special circumstance – you must budget your time.