EECS 592: Introduction to Artificial Intelligence, Fall 2018

Homework 1 – KenKen

Assigned: Sept 5, 2018

Due: Sept 24, 2017 at 11:59 PM

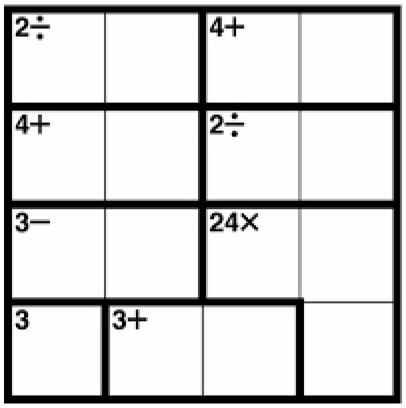
Submit your work to Canvas (https://umich.instructure.com). In the page for this course there is a tab for Assignments, under Homework 1 there will be a place to upload your files and answers. Please upload a single zip file containing all your files and answers. Below we give the details of what should be included.

In this homework, you will create a series of programs in Java that solve KenKen puzzles. https://tirl.org/software/kenken/ is an online solver to give you inspiration.

KenKen

KenKen is similar to Sudoku and involves filling numbers 1-N into cells an NxN grid. The figure below is a 4x4 puzzle, and cells are filled with the numbers 1-4. We use the following coordinate system: (x, y), where x is the column starting from left to right, and y is the row starting from top, both starting at 1. So (1, 1) is the top-left cell, which has 2÷, and cell (1, 2) has 4+.

In KenKen, each row and column must be filled with all of the numbers 1-N, in this case 1-4, so that no number is repeated in a row or a column. In addition, there are *groups* that define arithmetic constraints on a subset of the cells, so that when the operation in the group is applied to the numbers in the cells, the target number in the top corner is computed. In the upper right corner below, there is a group for cells (3, 1) and (4, 1) that requires that the number sum (+) to the target value 4, which means the cells must be either 1 | 3 or 3 | 1. For the grouping in the upper left corner, one of the two numbers must be divided by the other number to produce 2. In this case, the numbers can be 1, 2, or 4, and the order doesn't matter, so a correct solution could be 1 | 2, 2 | 1, 2 | 4, or 4| 2. When there is just a number and no operation, such as in cell (1, 4), the cell must have that value, which in this case is 3. A number can be repeated in a group if it is not in the same row or column.



In this problem, you will write a Java program that solves 4x4 and 5x5 KenKen puzzles using five different approaches.

The goal of this project is for you to learn about the power of using more and more knowledge to constrain the search for solutions, and potentially the cost of using that knowledge. You will start with a program that knows next to nothing about KenKen, and must search many states to find the solution. For the other approaches, you will incorporate more and more knowledge, significantly shrinking the search as your program uses that knowledge to avoid generating states that cannot possibly solve the problem. In the limit, it is possible to create programs that involve little to no search; however, even your final programs will probably require significant search for some of the problems.

We will give your program a series of 4x4 and 5x5 problems to solve. We define the format of a KenKen problem below. You program will need to read in that format, solve the puzzle using the approaches defined below, and for each approach report the following statistics:

- The solution
- Number of states generated
- The time (in microseconds) required to generate a solution.

We give a precise definition of the format of the input and output below.

You will submit your programs as a single Java source file, using no other libraries except java.util and java.io.

We will run a program on the output of your program to confirm that your solutions are correct (and do not copy other programs).

NOTE: It is a violation of the Honor code to use any part of anyone else's program for this assignment. This includes students in the class and programs from any other source, such as the internet. We will perform tests to ensure that every program is unique. If you find yourself unable to complete to programs in time, instead of copying someone else's program, contact Professor Laird (laird@umich.edu).

Approaches

Here are the approaches you must implement. Your program will apply all of these approaches to each problem. You can implement more approaches for extra credit. All the problems we give you will have a unique solution. Each one of these approaches is cumulative, so for example, approach 4 should incorporate all the knowledge to prune the search from 1-3, and 5 should include all the knowledge from all other approaches.

1. Brute Force Counting: Starting with all 1's, you program will incrementally "count" until it finds a solution. The counting is base N (4 or 5), so that for a 4x4 puzzle, your program will start with the sixteen digit number that is all 1's and count up until it finds a solution. So the total space is over 4 billion. This will get you going, with you having to write the code to import the board description, test all of the row, column, and group constraints on a complete board, and output the statistics. You can

organize the testing of the constraints anyway you want (testing the groups before you test all of the rows and columns or vice versa), but your program should not do anything clever in generating the states and should have the potential of generating all possible states.

2. A bit smarter counting: In this approach, create an ordering over the cells (don't be clever here). We recommend (1,1) (1, 2), (1,3), (1,4), (2,1) … (4,4) so that you can compare your results to example results we will publish next week. Fill in the cells incrementally in that order, and test the *partial* boards using the constraints that apply to the filled-in cells. If a board fails, increment the count of the last number (this is *backtracking).* If the last number was 4, go to the previous number and increment that (also *backtracking).* For example using the ordering above, your program will first generate a "1" for (1,1). It will then generate another "1" for (1, 2). It will detect that that violates the constraints that numbers in the same column must be different, and then generate "2" for (1, 2). Your program will apply row and column constraints, as well as group constraints when all of the cells in a group are filled.
3. Smarter counting (1): columns and row constraints: Instead of generating each number for a cell, your program will maintain a list of *open* numbers for each cell. Whenever a number is added to the same row or column of that cell, that number will be removed from the list of open numbers of that cell. When counting, as in 2 above, only those numbers in the open list will be used. So after filling in "1" for (1, 1), "1" will be removed from all cells in the first column and row. Thus, when filling in (1, 2), "2" will be used as a first guess, and then in filing (1, 3), "3" will be used.
   Complex Backtracking: If the current cell's open list is empty, or if the only remaining number fails a group constraint, or if 4 was tried and failed, then the search must *backtrack*, and the previous cell must be returned to and incremented (or backtracked). The tricky part about this approach is that backtracking requires "*undoing"* any removals from open lists that were done because of the values that are being backtracked from. The easiest way to do this is to do a recursive function call whenever setting the value of a cell through counting, where the board is copied down (arguments are passed by value). When there is backtracking, the most recent board is popped off the stack, and any of the changes to the open list are essentially undone. If this is confusing, come to the discussion class on this assignment.
4. Smarter counting (2): set unique values, backtrack on empty open lists. If during the search above, an ungenerated cell somewhere on the board has an open list become *empty*, backtrack. If during the search above, an ungenerated cell anywhere on the board has its open list set to a *single* value, choose that value for that cell, and then apply the column and row constraints (and then apply this if appropriate, and then apply column and row constraints, …). Note: for groups with a single valued in the problem definition (such as (1, 4) in the example board above), the row and column constraints should be applied at the beginning of the problem. Your program should not do any of these things in approaches 1-3.
5. Smarter counting (3): forward application of group constraints. If when a number is set for a cell (either through counting or approach 4 above), and if all but one of the cells in a constraint are filled, use the constraint to compute the number(s) that are still legal for the unfilled cell. Remove any numbers from the open list of the unfilled cell that do not equal those computed numbers. For example: If there is a group of two

cells that must add up to 5, and one cell is set to 2, the other cell should be set to 3. If there is a group that is “-1”, and one cell is set to 3, then 1 can be removed from the other cell. If this operation eliminates all numbers from the open list, then backtrack, if it goes down to one number apply approach 4. For multiplication and addition groups, having all but one cell filled determines a unique value for the empty cell, but for other groups (most subtractions and divisions), there can be multiple valid values for the unfilled cell.

Additional Approaches

For extra credit, you can develop an additional approach that further constrains the search so that a solution is found faster by generating further states and taking less time. Provide a description of your approach and include it in the analyses below.

Analysis document

The main point of this homework is to study how the different approaches affect the performance of the search algorithm. Your submission must include a PDF document with some analyses of your results. For each of the approaches you implement (should be at least the five above), include a table that includes:

1. The name and number of the approach.
2. For each puzzle size (4x4 and 5x5) create a table with summary data (min, max, mean, median) for each of these statistics.
    a. The number of states searched.
    b. The time it took in microseconds.
    c. The number of states/second.

Create a graph whose x axis is the different approaches (at least 1-5 above), and then overlapping plots (with different scales) of the median values of: the number of states searched, the time required, and the states/second. You might need to rescale some of these values to make everything fit.

Write a short paragraph of analysis on your takeaways from these data.

Input Format

The code should read a file named ‘input.txt’ which contains the problems one after another in a format with the first line containing N denoting the size of NxN grid. It is followed lines that define each of the group constraints as

<Operation><Space><Value><Space><Space delimited coordinates in x,y format>

Example:
4
Subtract 1 1,1 2,1
Divide 2 3,1 4,1
Multiply 8 1,3 1,4 2,4
And so on…

Output File Format

The code should generate a file named ‘output.txt’ which contains the solutions in the following format:

```
<Puzzle Number>
<Approach Number>
<Solution>
<Number of states generated>
<Time in microseconds>
```

The solution should contain each row in a new line with tabs in between columns:
Example (The values for number of states and time are just for illustration):

```
Puzzle 1:
Approach 1:
Solution:
2	4	3	1
1	3	4	2
4	1	2	3
3	2	1	4
States generated: 154368
Time in microseconds: 2587953
States/microseconds: .05964869
And so on…
```

Deliverables

On Canvas you will submit a single compressed file called <your-uom-unique-name>.zip. In that file, you must include the following files:

1. `analysis.pdf` – This is your analysis document described above, as a PDF document.
2. `solutions.txt` – A text file that has solutions to the problems, using each approach. Note: all solutions for the same problem should be the same, independent of approach. If they aren’t, you have a problem.
3. `Kenken.java` – A single Java source file containing all your program code, and which does not refer to any external libraries other than java.util and java.io. When we run this file, it should be able to take in input from the file ‘input.txt’ present in the same folder as the file and generate ‘output.txt’. We will use a program to analyze these source files to detect plagiarism. We expect your code to be your own original work and not copied either from the web or from a friend.

Grading

If you do all the things described above, including getting reasonable data for strategies 1 through 5, and have a good analysis, you will get an A. To get an A+ you must implement an additional strategy that improves (decreases) the number of solutions/states visited and document it along with the others.

There will be a 10% penalty for every day your program is late, up to 4 days.