CS 561: Foundations of Artificial Intelligence Fall 2012

Programming Assignment #1 due October 1, 2012

Constraint Satisfaction Problems

Overview

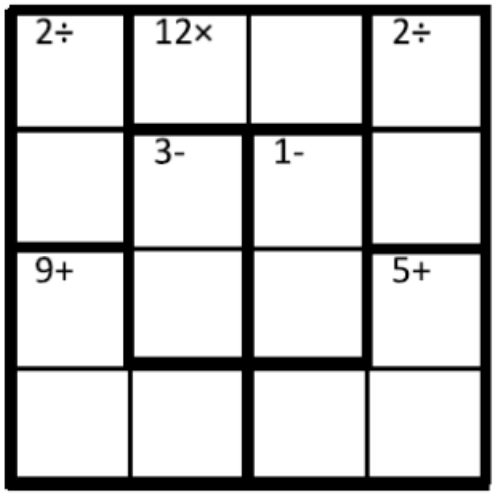
In this assignment, you are asked to implement a KenKen puzzle solver. This includes (1) formulating a representation for KenKen puzzles as a Constraint Satisfaction Problem (CSP) and (2) coding a backtracking algorithm to solve arbitrary KenKen puzzles. You should implement the variant of the backtracking that returns all possible solutions for a given puzzle. You are then asked to use your implementation to solve several puzzles and answer questions about them.

Several have been supplied that you must solve computationally and then answer a few questions regarding their solutions. Below, you will find a brief description of KenKen, a description of the input format and several questions that you must answer as part of your assignment.

KenKen

KenKen is a variety of logic puzzle that has many similarities to the highly popular Sudoku puzzles, but also incorporates arithmetic aspects. As with Sudoku, KenKen is played on an n by n grid, each square of which must be filled in with integer values between 1 and n. Also like Sudoku, none of the rows or columns can contain any repeated digits.

KenKen is unique in that the board is also divided into heavily outlined areas called cages, each of which encompasses several connected squares of the grid. Associated with each cage is a basic arithmetic operation (+, −, × and ÷) which, when applied to the numbers in that cage, produces a value which exactly matches the provided "target" result. More information about KenKen can be found at www.kenken.com. Below is an example of a blank KenKen puzzle (n = 4) and its solution.



1	3	4	2
2	1	3	4
3	4	2	1
4	2	1	3

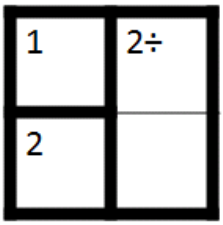
Note that special attention must be paid to cages that have subtraction or division as their operator, since these operations are not commutative. Such cages must encompass exactly 2 squares, and the magnitude of the values within the cage must be considered when deciding how to carry out the operation. For subtraction, the smaller number must always be subtracted from the larger number to produce a positive result. For division, the smaller number must always be the divisor, to produce a non-fractional result.

Additionally, in the case of cages that encompass a single square, no operator is relevant. The correct value for that cage/square is fixed to be the number listed in the cage.

Problem 1 (10 points):

a) (5 points) Briefly explain, describe and illustrate your method of representing KenKen puzzles, including the board and constraints. Which aspects are explicitly represented and which, if any, are implicitly represented?
b) (5 points) Briefly explain why a naïve search approach such as those represented by the tree-search algorithms in Chapter 3 would be, at best, inefficient for solving a KenKen puzzle (what property of CSPs is being ignored?). Why might a backtracking algorithm provide a better solution?

Problem 2 (15 points):

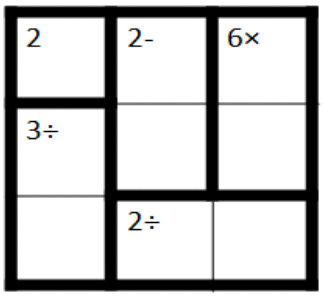


a) (5 points) How many solutions are there to this puzzle?

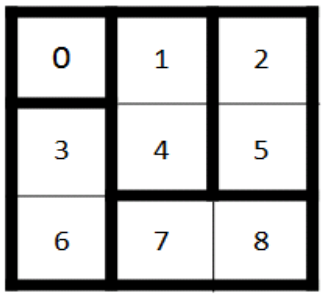
b) (5 points) Show a complete trace puzzle boards that were considered by the backtracking procedure, from the root node.

c) (5 points) Point out the steps where a branch of the search tree was abandoned because it was found to contain no possible solutions (i.e., the locations where backtracking occurred).

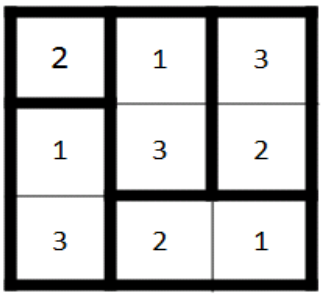
Instruction for Problem 3 and 4:
Implement a generic KenKen solver. Your KenKen solver will take an input file which can represent any arbitrary KenKen puzzle and solve a puzzle, then write all possible solutions on an output file. The input formats are the following:
Example 1:



Puzzle 1



Square Index



Solution 1

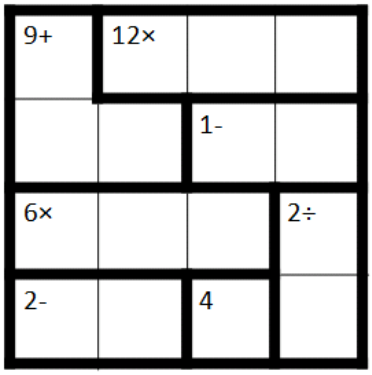
The representation of the puzzle 1 is
2 . 0
2 - 1 4
6 * 2 5
3 / 3 6
2 / 7 8

Thus, the input format is:
Cage_target_1 Cage_operator_1 square_index_1 square _index_2 … square _index_n
Cage_target _2 Cage_operator_2 square l_index_1 square _index_2 … square _index_n
Cage_target _3 Cage_operator_3 square _index_1 square _index_2 … square _index_n
…
Cage_result_M Cage_operator_M square _index_1 square _index_2 … square _index_n

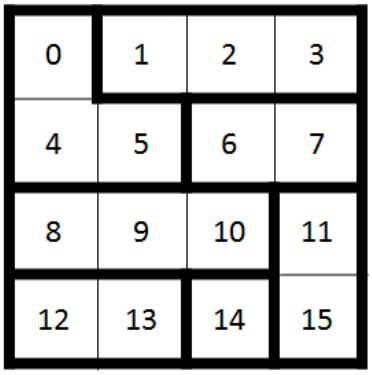
Each line represents a cage. Each column is separated by space. Each line can have different number of columns depending on number of squares in a cage. The first column always represents a “target result”. The second column always represents “operator”. There are five types of operator {+, -, *, /, .} The “+”, “-“ , “*” , “/” represent +, - , ×, ÷ respectively. The “.” represents a “no operator” which refers to the cage that contains a single square.
From the third column, they represent the square index contained in a cage.

For example, “2 . 0” has 3 columns which represents a cage that encompasses a single square. This cage has only one square whose index is 0. “2 - 1 4” has 4 columns represents a cage contains two squares 1 and 4. Its target result is 2 by using “-“operator. “6 * 2 5” has 4 columns represents a cage contains two squares 2 and 5. Its target result is 6 by using “×“ operator.

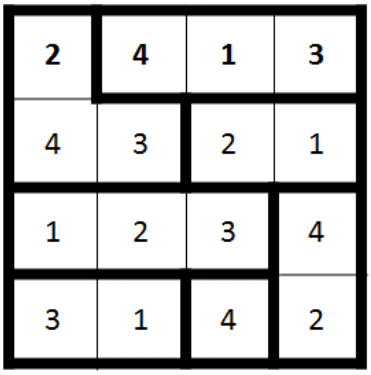
Example 2:



Puzzle 2



Square Index



Solution 2

The representation of the puzzle 2 is
9 + 0 4 5
12 * 1 2 3
1 - 6 7
6 * 8 9 10
2 / 11 15
2 - 12 13
4 . 14

Note that the size of board can be determined by the input. For example, the first puzzle has index from 0-8. Thus, your program should determine the size as 3 × 3. The puzzle2 has index from 0-15. Thus, your program should determine the size as 4 × 4. The index will be ordered by row then column. You can determine row and column by

row = floor(index/size)
column = mod(index, size)

For example, if size = 4 and index = 10, the row is floor(10/4) = 2 and the column is mod(10, 4) = 2.

The first line “9 + 0 4 5” has 5 columns represents a cage contains three squares 0, 4 and 5. Its target result is 9 by using “+“operator. The second line “12 * 1 2 3” has 5 columns represents a cage contains three squares 1, 2 and 3. Its target result is 12 by using “×“operator.

Deliverables
Your program will take an input file location as the first argument and take an out file location as the second argument. The unseen input files are used to test correctness of your program. You must name your main file as **KenKenPuzzleSolver.** Implementation language choices are **C++ or Java**. You must write all possible solutions on an output file. We will check your solutions from output files.

Your program is tested by

KenKenPuzzleSolver <input_file_location> <output_file_location>

For example,

In case of C/C++,
KenKenPuzzleSolver input_example1.txt output_example1.txt

In case of Java,
java KenKenPuzzleSolver input_example1.txt output_example1.txt

The example input and output files are provided on DEN: input_example1.txt, input_example2.txt, output_example1.txt and output_example2.txt

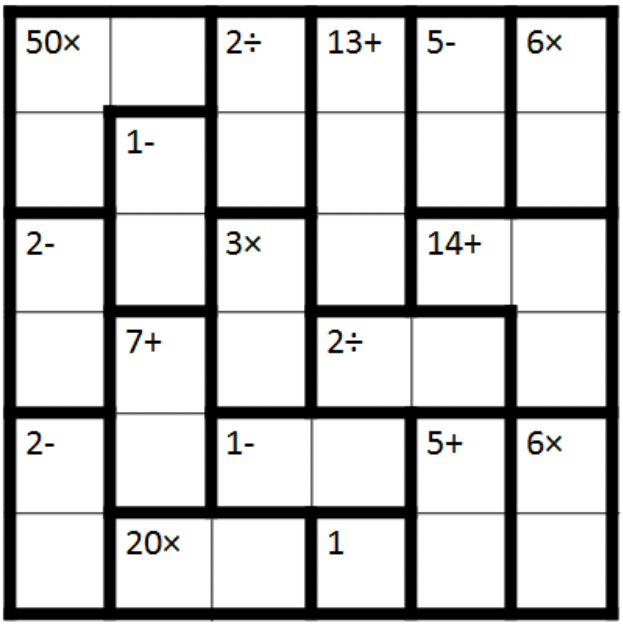
You are required to hand in all codes that you wrote to complete this assignment along with executable file named **KenKenPuzzleSolver**. In case of Java, you must submit all .class files and your main class must be named **KenKenPuzzleSolver**. In addition, you must hand in thorough answers to the problems 1, 2, 3 and 5.

The deadline for this assignment is **October 1 at 11:59pm Los Angeles time**.
Please turn in all materials as a .zip file via the DEN system, with the title format **[firstname]_[lastname]_Program1.zip (e.g., marc_spraragen_Program1.zip)**

Total points = 100

Problem 3 (30 points):

a) (25 points) Use your algorithm to solve the following KenKen puzzle:



The input file is provided as **input_q3.txt.** We will grade your solutions from your output file.

b) (5 points) How many solutions are there to this puzzle?

Problem 4 (35 points):

a) (35 points) Use your algorithm to solve the unseen KenKen puzzle:
Your program will be tested by unseen input files.

Problem 5 (10 points):

The book describes two domain-independent heuristics for CSPs. One is called the "most constrained variable" (or MRV = Minimum Remaining Values) heuristic and the other is the "least constraining value" heuristic. Explain whether these heuristics can be useful for KenKen puzzles and why or why not.

Extra Credit Question 1 (5 points):

Can you think of any domain-specific heuristics that might further limit the number of puzzles considered by backtracking?

Extra Credit Question 2 (15 points):

Implement a CSP heuristic from the book in your algorithm, for instance one suggested in Problem 5 or Extra Credit Question 1, and compare the performance on several puzzles vs. non-heuristic search.