PROGRAMMING ASSIGNMENT 1
Student: Hien To
Student ID: 7969-0916-46

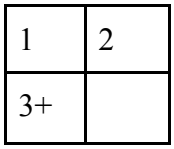
Problem 1:

Problem formulation:

- **Variable:** X11,..., Xnn
- **Domain**: D{1,..N}
- **Cage**: {Xij}
- **Constraints**
  - Row constraints X11 <> X21, … X11 <> Xn1
  - Column constraint: X11 <> X12, … X11 <> X19
  - Cage constraints: arithmetic operation is true
- **Goal**: assign a value to every variable such that all constraints are satisfied
- **Initial state**: empty assignment
- **Successor function**: unassigned variable = value without conflicting with current assigned variables.

a) I used BACKTRACKING SEARCH algorithm for solving this constraint satisfaction problem (CSP) (Pseodo code on page 215).

Input: a text file:

Board	**File**	**Explain**
	2 1,=;0,0 2,=;1,0 3,+;0,1;1,1	The first line is the size of each dimension The following next rows are the cages. For example, the last cage 3 is the sum, + is the operator, and [0,1] & [1,1] are the cells in the cage

Data Structures:

Variable	**Description**

enum Operator {
equal, plus, minus, multiply, divide
};
	Define 5 operators

class Couple • X • Y	Define a position of a cell
Class Cage • Operator opt; • int number; • Vector<Couple> varList;	Define a cage, including an operator, a number and a list of cell

Vector <Cage> cagesList	Define a list of cage which can be read from the input file
int assignment[][];	One instance of the results
static boolean isAssigned[][];	To keep track assigned variables
Vector<int[][]> resultSets;	To store all results
Hashtable<Couple, HashSet<Integer>> variableCandidates;	To store all possible candidates of a cell at a particular point

Note: I also need another variable to map from a cell to a cage: Hashtable<Couple, Cage> hashCages;

Constraints:
Explicitly represented:

- All the cell values must be between 1 and N
- For each constraint (row, column, cage) I have a function to check constraints: whether a new value for an UNSIGNED-VARIABLE conflicts with other cells in a row, column and cage in corresponding or not?

Implicitly represented:

- If we have 36 cells, the board is 6x6 square.

b)
The naïve search approach tries all possible search space (brute force), so the cost and time-complexity is worse and this approach is definitely not optimal.

The naïve approach ignores crucial property common to all CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. Therefore, we only need only to consider a single variable at each node in the search tree.

Moreover, the naive approach does not use the constraints of the problems to prune the search space. Whereas back tracking algorithm for solving this CSP provides a better solution because it can utilizes the constraints on variables. We are also able to add some heuristics to the algorithm to make it better. Specifically, I added some heuristics to my algorithms such as:

- Minimum remaining value: choose the cell with the minimum number of candidates values choose the Cage with the least unassigned cells.

Problem 2:

a)
The result of my algorithm for this problem is in the file **src/kenken/test1.out**

My algorithm to solve this Kenken problem only takes two steps:

1. First, I have a function named fixSingleSquareCage() to fill all single square cage initially.

2. Second, the algorithm run the backTracking() function. The problem terminates after one forward checking, finally I have the solution below.

b) There is one solution

1	2
2	1

c) Complete trace puzzle boards

d) On the image above, there are four steps where a branch of the search tree was abandoned and backtracking occurred because it was found to contain no possible solutions.

Problem 3:

a) The result of my algorithm for this problem is in the file **src/kenken/test2.out** at the code project. Please look at the file readme.txt in the project to run it.

b) There are 8 solutions to this puzzle as below.

2 5 4 1 6 3 5 2 1 4 3 6 4 1 6 3 5 2 1 4 3 6 2 5 3 6 2 5 1 4 6 3 5 2 4 1	2 5 4 1 6 3 5 2 1 4 3 6 4 1 6 3 5 2 1 4 3 6 2 5 3 6 5 2 1 4 6 3 2 5 4 1
2 5 4 1 6 3 5 2 1 4 3 6 4 1 3 6 5 2 1 4 6 3 2 5 3 6 5 2 1 4 6 3 2 5 4 1	2 5 4 1 6 3 5 2 1 4 3 6 4 1 3 6 5 2 1 4 6 3 2 5 3 6 2 5 1 4 6 3 5 2 4 1
2 5 4 1 6 3 5 2 1 4 3 6 4 1 3 6 2 5 1 4 6 3 5 2 3 6 2 5 1 4 6 3 5 2 4 1	2 5 4 1 6 3 5 2 1 4 3 6 4 1 3 6 2 5 1 4 6 3 5 2 3 6 5 2 1 4 6 3 2 5 4 1

2 5 4 1 6 3 5 2 1 4 3 6 4 1 6 3 2 5 1 4 3 6 5 2 3 6 5 2 1 4	2 5 4 1 6 3 5 2 1 4 3 6 4 1 6 3 2 5 1 4 3 6 5 2 3 6 2 5 1 4

6	3	2	5	4	1	6	3	5	2	4	1

c) The size of the complete search space (using standard depth-limited search for example):

If we use a standard depth-search, a state would be a partial assignment, and an action would be adding var = value to the assignment. If we have n variables of domain size d, the branching factor at the top level is nd because any of d values can be assigned to any of n variables. At the next level branching factor is (n-1)d, and so for n levels. Finally, we generate a tree with n!.d^n leaves.

Levels	**Branching factor**	**Number of boards**
Root node	nd	nd
level 1	n(n-1)d^2	nd + n(n-1)d^2
...		
level n-1	n!d^n	**nd + n(n-1)d^2 + .. + n!d^n**

- n is the depth of the tree = 36
- d is the domain size = 6

So the size of complete search space is: **nd + n(n-1)d^2 + .. + n!d^n**
= 36*6 + 35*36*6^2 + … + 36!*6^36

By adding an counting number to the code and increase the number by one on the function isAssignmentCompleted(), I found that the number of boards are considered by my algorithm is **173** (result from test2.out)

Problem 4:

a) The result of my algorithm for this problem is in the file **src/kenken/test3.out**
b) There is one solution as below

5	4	6	1	3	2
2	6	1	5	4	3
1	5	3	6	2	4
4	3	5	2	6	1
6	2	4	3	1	5
3	1	2	4	5	6

c) The size of complete search space is: **nd + n(n-1)d^2 + .. + n!d^n** (the same like problem 3.c)

The number of boards are considered by my algorithm is **264** (result from test3.out).

Problem 5:

The Minimum Remaining Values (MRV) heuristic picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. So, this heuristic avoids pointless searches. Because Kenken problem to get an answer normally we have to prune a lot (for the solution space is much less than the search space). So MRV may be useful for Kenken puzzles.

It depends. The Least Constraining Value (LCV) heuristic can be useful if we only need to find one solution for Kenken problem. Otherwise, if we need to find all possible solutions, the heuristic may not be efficient. Because the heuristic tries to leave the maximum flexibility for subsequent variable assignments, we are likely to find the first solution early. But if we need to find all solutions to a problem, then the ordering does not matter.

Extra credit Question 1:
I come up with a domain-specific heuristic that might further limit the number of puzzles considered by backtracking:

1. **Completed cage:** if the cage has only one unassigned cell, then we can calculate the last cell. So the cage is completed.

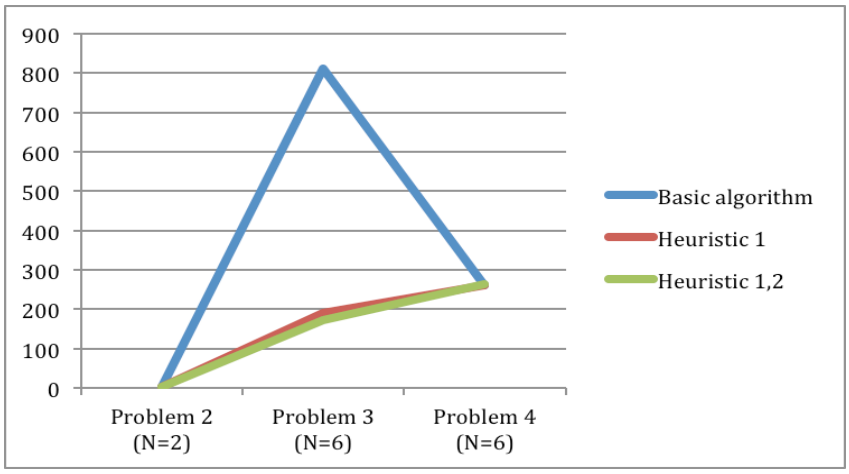
Extra credit Question 2:
I used two a domain-independent heuristic and one domain-specific heuristic in the above question.

1. **Minimum remaining cages:** choose a cage which has minimum number of unassigned cell.
2. **Minimum remaining cells:** choose a cell which has minimum remaining value
3. **Completed cage**

The number of boards is considered (to find all solutions) is as below table:

Data set	**Basic algorithm**	**Heuristic 1**	**Heuristic 1,2**
Problem 2 (N=2)	3	3	3
Problem 3 (N=6)	**811**	**192**	173
Problem 4 (N=6)	260	262	264

Number of boards considered



So when I applied my heuristics to the problem, the performance is better. Especially the heuristic 1 improves the algorithm significantly for many-solution problems such as problem 3.