Meta-LS Solver for Magic Square Competition

Author: Xiao-Feng Xie
Email: xfxie@cs.cmu.edu

November 20, 2011

This document describes Meta-LS Solver for the constrained Magic Square Problem.

1 Usage

The basic program runs with the commands:

$ java -cp MagicSquare.jar MSMain -n 100 -c

where -n gives the size of the problem, and -c indicate that the problem is a constrained one.

There are two further options (if -c is used):

(1) -p number1 number2: where (number1, number2) specify the placement of the upper lef-hand corner of the sub-matrix within the main matrix;

(2) -f filename: which is used for loading a sub-matrix. The default sub-matrix is:

1 2 3
4 5 6
7 8 9

The output is a text file called "results.txt".

2 Algorithm Description

The algorithm contains four iterative local search steps, i.e., optimizing row elements, optimizing column elements, optimizing forward diagonal, and optimizing backward diagonal.

The initial solution is generated at random. During the first two steps, The sub matrix is placed at (0, 0), since it can be re-located anywhere in the last two steps.

The first step optimizes the matrix based on the row values. Any conflicting rows are optimized row by row. If one row is optimized, then the row is no longer considered. Each row is exhaustively scanned, and each element is greedily replaced by the best one from conflicting rows. If no improvement can be achieved in a whole round, some elements are mutated in conflicting rows. The process is iteratively applied on those conflicting rows again, until all of them are optimized.

The second step further optimizes the matrix based on the column values, as well as keeping that all rows remain optimized. This can be simply achieved by only exchanging each element with another element in the same row. All columns in conflicting are scanned, and each column is no longer considered if it is optimized. If no improvement can be achieved, some elements in remaining columns are mutated in their rows. The process is iteratively applied on conflicting columns again, until all columns are optimized.

After the first two steps, a semi-magic square is obtained. In this work, I considered two kinds of operations that will not change the nature of a semi-magic square. The first kind of

operations are permutations of rows and columns. Given a semi-magic square, An auxiliary data structure of two permutations (of orders of rows and columns) can be used for reducing computing time, and the semi-magic square itself is changed only if necessarily. Note that each changes on the permutations will change both the values of the forward and backward diagonals. The second kind of operations are *two-swaps*. Here a smaller rectangle $\left(\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right)$ is placed on the square. One of its vertices A is on the diagonal, and all others (B, C, D) are not on diagonals (and all vehicles are not in the sub matrix. For the values in these vertices, if $A - B \equiv D - C$, then the two swaps $A \leftrightarrow B$ and $C \leftrightarrow D$ will not changes the nature of the semi-magic square and the elements on another diagonal, although it will change the value of the diagonal on A ($\Delta = B - A$). The similar result can be obtained if $A - C \equiv D - B$.

During the last two steps, the sub matrix are moved to the required place, i.e., the fixed place that assigned by "-p", or an arbitrarily assigned place if no place is assigned. In the latter case, I place it at $((n - SubSquareRowSize)/2, n - SubSquareColSize)$. This move can be simply achieved by changing the permutations.

The third step is applied for optimizing the forward diagonal by iterative local search. Greedy 2-opt permutations on rows and columns are exhaustive applied. Afterward, two-swaps on the forward diagonal are exhaustively applied. If the forward diagonal is still not optimized, mutations are performed on the two permutations. Then the process is iteratively applied until the forward diagonal is optimized.

The last step is applied for optimizing the backward diagonal. Here only greedy two-swaps are applied. If the backward diagonal is not optimized, some mutations (two-swaps with arbitrary delta values) are applied. Then the process is iteratively applied until the backward diagonal is optimized. To avoid the heavy tail effect of iterative greedy search, a multi-run setting of this step is considered, as suggested in the thoery of algorithm portfolios.