AI Method CourseWork Report

Shujun JIANG

July 5, 2023

1 Introduction

A magic square is an arrangement of numbers in a square grid, where the sum of each row, column, and diagonal is the same. Magic squares have been studied for centuries, and there are many different algorithms for solving them. In this report, I will introduce the algorithm I designed to solve the magic square problem using an improved variable neighborhood search (VNS) with multiple matrix transformation heuristics and some simulated annealing (SA) ideas.

2 Components of the Algorithm

This algorithm is based on the VNS algorithm framework and is mainly divided into two parts: local search and shake. This section will introduce the composition and workings of local search and shake. Additionally, this section will cover solution encoding, fitness functions, neighborhoods, and considerations about intensification and diversification mechanisms.

2.1 Local Search Method

In the VNS algorithm, local search is a method used to find the local optimal solution at the current stage. This algorithm's local search uses a method similar to Random Permutation(RP)[1] with Best Descent(BD). The original RP generates a random sequence of heuristic calls, allowing more efficient heuristics to be called more often and less efficient heuristics to be called less often. In this algorithm, RP is modified to generate a value within a function. If the value falls within a certain range, the corresponding heuristic is called for local search.

These heuristics are used to exchange some elements of the matrix. There are five exchange methods in this algorithm, which are written in the code:

1. swap_elements:
   It randomly swaps two elements of the matrix to form a new matrix.

2. swap_LLH0[2][1]:
   The function checks whether the rows, columns, and diagonals of a matrix meet the magic square constant. If not, it selects an element from the non-conforming row, column, or diagonal that may cause non-conformance (for example: if the sum of a row exceeds the magic square constant, it will randomly select an element greater than M/n from this row) and swaps it with an element from another row, column, or diagonal that meets the magic constant and causes non-conformance in the matrix.

[1]LLH0,LLH3,LLH5,LLH8 refer to the heuristics written in the two papers cited in the report, and the labels are the same as those in "Multi-stage hyper-heuristics for optimisation problems". It also corresponds to LLH1, LLH4, LLH6 and LLH9 in "Constructing constrained-version of magic squares using selection hyper-heuristics". The algorithms in this report refer to these methods in how to select the row and column diagonals, but how exactly elements are selected for swapping on the row and column diagonals is not the same as they are. For example, if the sum of rows is greater than the magic square constant, select a random element greater than M/n in the row to exchange for other elements less than M/n in the row that do not satisfy the magic square constant. The purpose line of the heuristic is strengthened in this algorithm to make it more efficient in local search.

3. swap_LLH3:
   This function is similar to the LLH0 function. The only difference is that the process of checking whether the rows, columns, and diagonals of a matrix meet the magic square constant and swapping elements to adjust the matrix is repeated until the selected row, column or diagonal satisfies the magic square rule or until no further improvement is observed. This means that the function will continue to make adjustments until the desired result is achieved or until it is determined that no further progress can be made.

4. swap_LLH5:
   This function selects two elements from the matrix randomly and swaps them. The selected elements must not be located on a row, column or diagonal that already satisfies the magic square constant. This means that the function will only make adjustments to rows, columns or diagonals that do not yet meet the desired condition. By swapping these two elements, the function aims to bring the matrix closer to satisfying the magic constant rule for all rows, columns and diagonals.

5. swap_LLH8:
   This function selects the row, column or diagonal with the largest sum and another row, column or diagonal with the lowest sum. Then, for each pair of corresponding elements in these two rows, columns or diagonals, the function swaps them with a probability of 0.5. This means that there is a 50% chance that each pair of elements will be swapped. By doing this, the function aims to balance the sums of the selected rows, columns or diagonals and bring them closer to satisfying the magic square constant.

The RP method in the algorithm calculates a value and calls the corresponding matrix swap method based on whether the value falls within a certain range. Each call to the RP method randomly generates a value to simulate the call probability, and a random sequence of calls is simulated by continuously calling the RP method.

Algorithm 1: RP algorithm

Input: magic square $magic\_square$, order n, target value M, objective function value obj
Generate a random number $rand\_num$ between 0 and 99;
if $rand\_num < 45$ **then**
  $swap\_elements(magic\_square, n)$;
else if $rand\_num < 80$ **then**
  $swap\_LLH0(magic\_square, n, M)$;
else if $rand\_num < 90$ **then**
  $swap\_LLH3(magic\_square, n, M)$;
else if $rand\_num < 97$ **then**
  $swap\_LLH8(magic\_square, n, M)$;
else
  $swap\_LLH5(magic\_square, n, M)$;

The call probability of LLH0, LLH3, LLH5 and LLH8 is designed according to the following figure 1[1][2]. Then, according to the design when creating the algorithm, 45% of the probability is given to swap_elements, and the remaining probability is proportionally distributed to the four exchange methods, and then adjusted according to the test results.

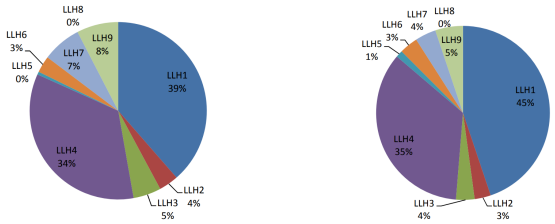

Figure 1: Heuristic call probability in RP function

[2]In the article "Constructing constrained-version of magic squares using selection hyper-heuristics", the heuristic count starts from 1, so the 1 in the figure corresponds to the 0 written in the report.

2.2 Shake Function

When the magic difference of the matrix cannot be improved for a long time, the algorithm will consider that the current matrix is trapped in a local optimum solution and a shake operation is needed. The conditions for shake to occur are different for matrices of different orders. For higher-order matrices, more times of not being able to find a better solution may be required before shake occurs.

For shake, the algorithm needs it to escape from a local optimum, but does not want it to escape "too far". The algorithm gets out of the state of being unable to get any further by randomly swapping two elements and simply making it the current matrix. However, this operation may also cause the local search done before shake to be of no help to the next local search, and to restart the descent from the point far away from the standard magic square. So for the shake operation, if it can cause the matrix to be closer to the magic square, continue from this closer to the magic square, and if it will cause the matrix to deviate from the magic square, the algorithm will also make it deviate from the magic square as small as possible. The generated shake will run n^2 times, and the best shake result will be selected as the operation object of the next local search.

Algorithm 2: shake_magic algorithm

```
Data: magic square magic_square, order n, target value M
Result: optimal magic square best_shake
min_difference = INT_MAX;
best_shake = 2D array of size n × n;
shake_candidate = 2D array of size n × n;
magic_square_cost = objective_function(magic_square, n, M);
for i = 0 to n × n − 1 do
    shake_candidate = magic_square;
    swap_elements(shake_candidate, n);
    shake_candidate_cost = objective_function(shake_candidate, n, M);
    difference = shake_candidate_cost − magic_square_cost;
    if difference < 0 then
        return shake_candidate;
    difference = abs(difference);
    if difference < min_difference then
        min_difference = difference;
        best_shake = shake_candidate;
return best_shake;
```

2.3 Solution Encoding

The solution encoding is magic_square, a two-dimensional integer vector that represents a matrix of order n. Each element magic_square[i][j] is the element in the ith row and JTH column of the matrix, and M is the value of the magic square constant when the matrix becomes a magic square. Thanks to the encapsulation of C++ for Vector library and some algorithms, the algorithm can efficiently and simply complete some matrix operations, such as swapping elements only need to use swap function, no longer need to double loop traversal copy and create a temp for temporary cache.

2.4 Fitness Function

The fitness function is the objective_function, which calculates the sum of the absolute values of the differences between the sum of the elements in each row, column and diagonal of the matrix and a given value M. The smaller this value, the closer the matrix is to a magic square. After performing a local search and calculating a new cost, it is compared with the previous cost value. If the new cost is smaller, then this new matrix will be used as the basis for the next operation. If the cost increases, the idea of Simulated Annealing (SA) is used, where there is a certain probability that this new solution will be accepted for further operation to avoid falling into a local optimum.

2.5 Neighbourhoods

In this algorithm, the neighborhood of the current matrix is defined as all matrices that can be reached by applying one of five matrix change functions. During local search, the RP method selects a neighboring matrix to transform into. The fitness function evaluates the new matrix's fitness, and based on its value and temperature, the algorithm decides whether to accept it as the current solution.

If the new matrix has a better fitness value than the current solution, it is more likely to be accepted. This process is repeated until a satisfactory solution is found or a stopping criterion is met.

2.6 Intensification and Diversification Mechanisms

The intensification and diversification mechanisms are reflected in the matrix swapping methods. The swap_elements function achieves matrix diversification by randomly swapping two elements in the magic square. On the other hand, LLH0, LLH3, LLH5, and LLH8 focus more on using specific swapping rules to achieve intensification, attempting to improve the objective function value. In addition, the algorithm uses the idea of simulated annealing to avoid getting stuck in local optima. When a new solution is worse than the current best solution, there is still a certain probability of accepting the new solution, thus achieving diversification. In the shake part, the algorithm deliberately selects a matrix that can escape from local optima and is closest to the target magic square matrix to achieve an intensification effect.

Algorithm 3: variable_neighborhood_search algorithm

Input: magic square $magic\_square$, order n, target value M
Output: output square $output\_square$
Initialize $current\_solution$ and $best\_solution$ to $magic\_square$;
Initialize pre_cost, $current\_cost$, $output\_cost$ and $best\_cost$ to $objective\_function(current\_solution, n, M)$;
Initialize SA parameters: $temperature$ and $SA\_TS = 500$, $SA\_TF = 0$, and $SA\_BETA = 0.001$;
while $time\ elapsed \leq MAX\_TIME$ *and* $temperature \geq SA\_TF$ **do**
    Set $current\_solution$ to $best\_solution$;
    $RP(current\_solution, n, M, best_cost)$;
    Set $current\_cost$ to $objective_function(current\_solution, n, M)$;
    if $current\_cost < best\_cost$ **then**
        Set $best\_solution$ to $current\_solution$;
        Set $best\_cost$ to $current\_cost$;
        if $best\_cost = 0$ **then**
            Set $output\_cost$ to $best\_cost$;
            Set $output\_square$ to $best\_solution$;
            Return;
    else
        Update temperature: $temperature = temperature/(1 + SA_BETA * temperature)$;
        Calculate delta: $delta = current_cost - best_cost$;
        Calculate probability: $probability = exp(-delta/temperature)$;
        if $random\ number \leq probability$ **then**
            Set $best\_solution$ to $current\_solution$;
            Set $best\_cost$ to $current\_cost$;
    if $pre\_cost = best\_cost$ **then**
        Increment $no\_improve$ by 1;
    else
        Set $no\_improve$ to 0;
    Set pre_cost to $best\_cost$;
    if $no\_improve \geq 10000$ *and* $n \leq 10$ **then**
        Set $best\_solution$ to $shake\_magic(output\_square, n, M)$;
        Set $best\_cost$ to $objective\_function(best\_solution, n, M)$;
        if $best\_cost < output\_cost$ **then**
            Set $output\_cost$ to $best\_cost$;
            Set $output\_square$ to $best\_solution$;
        Set $no\_improve$ to 0;
    else if $no\_improve > 800000$ *and* $n \leq 20$ *and* n ¿ *10* **then**
        Set $best\_solution$ to $shake\_magic(output\_square, n, M)$;
        Set $best\_cost$ to $objective\_function(best\_solution, n, M)$;
        if $best\_cost < output\_cost$ **then**
            Set $output\_cost$ to $best\_cost$;
            Set $output\_square$ to $best\_solution$;
        Set $no\_improve$ to 0;
    else if $no\_improve > 3000000$ *and* n ¿ *20* **then**
        Set $best\_solution$ to $shake\_magic(output\_square, n, M)$;
        Set $best\_cost$ to $objective\_function(best\_solution, n, M)$;
        if $best\_cost < output\_cost$ **then**
            Set $output\_cost$ to $best\_cost$;
            Set $output\_square$ to $best\_solution$;
        Set $no\_improve$ to 0;
Set $output\_square$ to $best\_solution$;

3 Statistical Results

Order	Best Result	Worst Result	Average Result	Zero Occurs
5	0	0	0	10
6	0	0	0	10
10	0	0	0	10
12	0	0	0	10
15	0	0	0	10
20	0	0	0.2	9
25	0	18	5.7	5
30	0	9	3.7	5

Table 1: Statistics of the results of 10 runs

4 Discussion and Reflection

This algorithm uses a variable neighborhood search algorithm to solve the tesseract problem. It has the advantage of being able to find a feasible solution quickly and the code structure is clear and easy to understand. It finds the optimal solution by continuously swapping elements in the tesseract, using several different swapping strategies and judging whether to accept a new solution based on the value of the objective function. One of the algorithms also uses the idea of annealing algorithm to prevent it from falling into a local optimal solution. This method is able to find a better solution in a short time.

The variable neighborhood search algorithm is a heuristic algorithm that finds the optimal solution by continuously changing the neighborhood structure. This algorithm can effectively avoid falling into a local optimum and thus find a better solution in a short period of time. In this algorithm, the authors use several different swapping strategies to change the neighborhood structure, including swapping elements randomly, swapping rows and columns, swapping diagonals with several heuristics, and so on. These strategies can effectively expand the search space and improve the efficiency of the algorithm.

However, any algorithm has its limitations. This code has a high time complexity and may not be very applicable for large-scale problems. In addition, it has limited scalability, and although more excellent heuristics can be added to expand and optimize the algorithm, depending on how good the newly added heuristics are, the matching parameters and other parameters in RP need to be adjusted to be able to handle more complex work. However, this does not affect its excellent performance in solving the Rubik's Cube problem.

In conclusion, this algorithm is a good example showing how to use a variable neighborhood search algorithm to solve the tesseract problem. It is able to find a feasible solution quickly and has high practical value. Of course, the algorithm can continue to be improved to make it better able to cope with more complex problems.

5 Summary

This report presents an algorithm designed to solve the tesseract problem using an improved variable neighborhood search (VNS) with multiple matrix transformation heuristics and some simulated annealing (SA) ideas. One of the key components of the algorithm is the randomized permutation RP method, which generates a random sequence of heuristic calls to efficiently swap elements in the tesseract. The algorithm is of high practical value as it is able to find a feasible solution quickly. However, it has a high time complexity and may not be well suited for large-scale problems. In addition, it has limited scalability. Overall, this algorithm is a good example of how to use a variable neighborhood search algorithm to solve a tesseract problem.

References

[1] Ahmed Kheiri and Ender Özcan. Constructing constrained-version of magic squares using selection hyper-heuristics. *The Computer Journal*, 57(3):469–479, 2014.

[2] Ahmed Kheiri. *Multi-stage hyper-heuristics for optimisation problems*. PhD thesis, University of Nottingham, 2014.