IT 12 018
Examensarbete 15 hp
Juni 2012

Water Retention on Magic Squares with Constraint-Based Local Search

Johan Öfverstedt

Abstract

Water Retention on Magic Squares with Constraint-Based Local Search

Johan Öfverstedt

Water Retention on Magic Squares is the hard combinatorial optimisation problem of searching for magic squares with a maximum amount of water retained by their structure when viewed as a heightmap. Constraint-Based Local Search, which is a non-exhaustive search technique for problems with constraints, has been applied to this problem. Several objective functions to guide the search have been developed and compared with respect to execution time and quality of solutions found.

Handledare: Pierre Flener
Ämnesgranskare: Justin Pearson
Examinator: Olle Gällmo
IT 12 018
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

The problem of Water Retention on Magic Squares is a very hard combinatorial optimisation problem. A magic square is defined as a square with distinct elements from $\{1..n^2\}$ where n is the order of the square and all the rows, columns and main diagonals sum to the same constant value. Water Retention is the problem of pouring water over a surface and calculating how much water is retained in the holes of the surface while the rest spills over the edges. For this thesis project I have tried to solve the problem of finding magic squares, maximising for retained water, with a technology called Constraint-Based Local Search, a local search technique for problems involving constraints. Every constraint has a violation function which gives its degree of dissatisfaction. The violation of the whole system of constraints can then be used to guide the local search towards a satisfied state. To achieve good performance with Constraint-Based Local Search one can use the general meta heuristic Tabu Search which temporarily bans changed elements from being changed again for a certain amount of time. I have developed 3 objective functions for searching for highly retaining magic squares which all have different strengths and weaknesses. One objective function making use of the actual water retention executes very slowly but achieves highly retaining squares. Another is an approximation using a diamond-shaped pattern, which fits observed good characteristics of highly retaining squares. It's able to quickly find squares with good but not great retention values.

2 Water Retention on Magic Squares

The problem of maximising the amount of water retained by a magic square is quite a new problem. The earliest source I've found regarding the problem was written in the year 2007 by Knecht [4]. In the spring of 2010 Al Zimmerman arranged a competition which yielded a lot of interest and led to the finding of most of the maximum known retaining magic squares. The results from that competition will serve as the benchmark against which I will measure my own results. In the following sub-chapters I will present the two main concepts of the problem: magic squares and water retention.

2.1 Magic Squares

The magic square is a mathematical concept which has fascinated mathematicians and enthusiasts for ages. Definition 1 gives the mathematical definition.

Order	Number of Magic Squares
1	1
2	0
3	1
4	880
5	275305224

Table 1: Number of Different Magic Squares by Order.

Definition 1. *Let $n \in N$. Let $m \in N$. Let A be a square matrix of size n by n such that $A[i,j] \in \{1..n^2\}$ where $i, j \in \{1..n\}$. Then A is a magic square if it satisfies the following constraints:*

$$A[i,j] \neq A[k,l] : \forall i,j,k,l \text{ such that } i \neq k \text{ or } j \neq l. \qquad \textit{Distinct constraint}$$

$$\sum_{i=1}^{n} A[x,i] = m : \forall x \in \{1..n\} \qquad \textit{Row constraints}$$

$$\sum_{i=1}^{n} A[i,x] = m : \forall x \in \{1..n\} \qquad \textit{Column constraints}$$

$$\sum_{i=1}^{n} A[i,i] = m \qquad \textit{Right diagonal constraint}$$

$$\sum_{i=1}^{n} A[i,n-i+1] = m \qquad \textit{Left diagonal constraint}$$

English description: Every cell in the matrix takes on a value in the range $\{1..n^2\}$ distinct from every other cell. Every row and every column and the two diagonals sum to the same constant m.

The number of different magic squares, counting all magic squares reachable by rotation and reflection as the same, has been calculated precisely up to order 5 as given by Table 1 [6]. As the number appears to explode exponentially any kind of exhaustive search over the space of magic squares of a certain order becomes infeasible as the order increases.

Next we'll look at the other part of the problem: Water Retention. We discuss its definition and an algorithm for calculating it for any square heightmap.

2.2 Water Retention

Let H be a 2-dimensional matrix with elements $h_{ij} \geq 0$. Viewing H as a heightmap, then the water retention can be defined as the volume of water that is retained when water is poured over this surface, spilling out from the edges, flooding between non-diagonally adjacent cells in the matrix and gathering in the places surrounded by elements of greater values. Given no other restrictions than the size of the matrix the maximum amount of water is achieved by placing a big wall, the higher the better, along the edges and place all zeros in the middle. A water retention algorithm [4] has been designed by Gareth

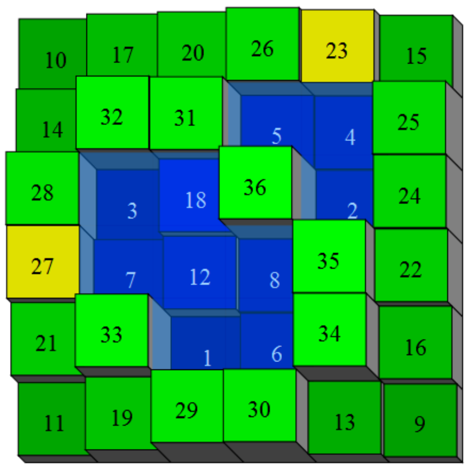

Figure 1: Magic Square of Order 6 Retaining 192 Units of Water.

McCaughan, which Algorithm 1 is based on, which calculates the water retention volume in $O(n^4)$ time with array-implementation for the priority queue. This could be reduced to $O(n^2 \cdot \log n)$ with a min heap data structure but experiments showed it not to pay off for my Comet implementation. The algorithm is based on the concept of relaxation used by some algorithms for computing Shortest Paths.

Figure 1 displays a magic square of order 6 with its height map and retained water displayed in a 3-dimensional perspective view. The green colour means dry land. Blue colour means water has been retained on top of that cell. Yellow colour marks the sensitive cells, which if lowered would lower the water level of the whole adjacent lake.

The problem of maximising water retention on magic squares is a hard one. In the next chapter we'll develop the main ideas and a basic algorithm of the Constraint-Based Local Search-method. Then in later chapters we'll see how this method can be tuned to produce good solutions to the problem.

2.3 Prior Work

Aside from the invention of the WATER-RETENTION-algorithm and the work during Zimmerman's competition, the amount of prior work on this problem does not appear to be particularly large. The good methods discovered during the competition have not been published in any scientific setting that I'm aware of. van der Plancke [7] had a similar approach to mine as he used Tabu Search with swapping pairs of elements. He made use of some interesting approximations, one of them was a way to establish an upper bound on the water retention. I would have explored it as a part of this project had the project time not come to an end shortly after finding the description. Many of the methods used in the competition made use of human-aided techniques which let the user set certain patterns to reduce the complexity of the problem or trying to improve on previous good

Algorithm 1 Water Retention Algorithm

```
WATER-RETENTION($H, W, n, h_{max}$)
// **Signature:** $Matrix[1..n, 1..n], Matrix[1..n, 1..n], int, int \rightarrow int$
// **Pre:** $H$ is a heightmap where $H[*, *] \in \{0..h_{max}\}$. $n > 0$. $h_{max} > 0$
// **Post:** $W$ contains the calculated water levels. Returns the water retention.
Let $Q$ be a minimum priority queue of elements with position $i$ and $j$
The priority of $Q$'s elements is given by $W[i, j]$

// Initialisation
for $i$ from 1 to $n$
    for $j$ from 1 to $n$
        if $i = 1$ or $i = n$ or $j = 1$ or $j = n$ // Edge element
            $W[i, j] := H[i, k]$
            ENQUEUE($Q, i, j$) // Add edges to queue
        else
            $W[i, j] := h_{max}$

// Calculate water levels
while $Q \neq \emptyset$
    $(i, j) :=$ DEQUEUE($Q$) // Extract min
    DRAIN($H, W, n, Q, i - 1, j, W[i, j]$) // Drain adjacent elements
    DRAIN($H, W, n, Q, i + 1, j, W[i, j]$)
    DRAIN($H, W, n, Q, i, j - 1, W[i, j]$)
    DRAIN($H, W, n, Q, i, j + 1, W[i, j]$)

// Calculate retention from water level matrix
Let $r$ be an integer
$r := 0$
for $i$ from 1 to $n$
    for $j$ from 1 to $n$
        $r := r + W[i, j] - H[i, j]$
return $r$
```

Algorithm 2 Water Retention Algorithm - DRAIN Sub Procedure

```
DRAIN($H, W, n, Q, i, j, p$)
if $i > 1$ and $i < n$ and $j > 1$ and $j < n$
    Let $x$ be an integer.
    $x :=$ MAX($H[i, j], p$)
    if $x < W[i, j]$ // If there is water to drain...
        $W[i, j] := x$
        ENQUEUE($Q, i, j$)
```

specimens. These results reside in a Yahoo-group which requires membership to be able to gain access.

3 Local Search

3.1 Constraint-Based Local Search

Constraint-Based Local Search [8] is a form of non-exhaustive perturbative search which can be used to solve constraint satisfaction problems as well as optimisation problems with constraints. In general let C be a system of n constraints $C = \{c_1, .., c_i, .., c_n\}$ over m variables $X = \{x_1, .., x_i, .., x_m\}$, possibly with different domains, where the constraints of C may constrain some or all the variables of X. Let S be the set of possible assignments for X given the domains of the variables. Consider state $s \in S$ which may or may not be a state satisfying all the constraints of C. Every constraint c_α comes equipped with a violation function $\mathit{Violation}(c_\alpha, s)$ which gives some metric of how far it is from being satisfied in state s, or possibly a binary value signalling whether it's satisfied or not. Equation 1 shows how the total violation of a state is computed, summing the violation functions for all the constraints.

$$StateViolation(s) = \sum_{i=1}^{n} Violation(c_i, s) \tag{1}$$

Example 1. *The $Sum(X) = m$ constraint which sums a set of variables X and contrains the sum to be m has a natural violation function, which may or may not be good for a given problem : $Violation(Sum(X) = m, s) = |Sum(X) - m|$.*

Two important concepts at the heart of Local Search are operators and neighbourhoods. Let O be some operator which transforms state s into a set of new states $N_O(s)$ respecting the domains of the variables in s. Let $N_O(s)$ be called the neighbourhood of state s w.r.t. operator O. Examples of operators are changing the value of a variable, swapping two elements of a matrix or swapping rows and columns of a matrix.

When an operator (or possibly a set of operators) has been selected, the general search procedure is given by Algorithm 3.

To achieve good performance with this algorithm fast methods for calculating how the $StateViolation$ function changes for the potential moves of N_O is instrumental as it must be calculated every iteration for every potential move allowed by the operator. Let's define the $\Delta_O(c, s_1, s_2)$ which gives the change of violation value for constraint c when moving from state s_1 to s_2 w.r.t. operator O.

Algorithm 3 Constraint-Based Local Search Algorithm

```
CBLS($s, iterations$)
1  // **Signature:** Initial state $s$ and maximum number of iterations $\rightarrow$ new state $s'$.
2  // **Pre:** $s$ is a valid state w.r.t. the domains of all the variables within.
3  // **Post:** $s'$ is a state satisfying the constraints or a timeout error was signalled.
4  Let $t$ be an integer
5  $t := 0$
6  **while** ($\mathit{StateViolation(s)} > 0$ and $t < \mathit{iterations}$ )
7        $\mathit{SelectMin}(n \in N_O(s) : \mathit{StateViolation}(n) + \alpha(t) \cdot \mathit{Cost}(n, t))$
8
9        $s := n$ // Make the selected $n$ the new current state
10       $t := t + 1$
11 $s' := s$ // Copy state $s$ and assign to $s'$
12 **if** $\mathit{StateViolation(s)} > 0$
13       TIMEOUT // Signal failure
```

Example 2. *Let's continue with the constraint from the previous example and consider the operator that swaps the value of two variables x_1 and x_2. Let's assume that we have saved the previous sum calculation which is stored in an auxiliary variable s and the set of variables participating in the sum constraint is X. Instead of swapping the values and then recalculating the sum for every potential move we can just check what the difference will be assuming we have a fast way of knowing if a variable participates in the sum. There are four cases to consider.*

$$\Delta_{Swap}(\mathit{Sum}(X) = m, x_1, x_2) = \begin{cases} 0 & \text{if } x_1 \in X \text{ and } x_2 \in X \\ 0 & \text{if } x_1 \notin X \text{ and } x_2 \notin X \\ |(s - x_1 + x_2) - m| & \text{if } x_1 \in X \text{ and } x_2 \notin X \\ |(s - x_2 + x_1) - m| & \text{if } x_1 \notin X \text{ and } x_2 \in X \end{cases}$$

Constraint-Based Local Search can be applied to problems with no other objective function than $StateViolation(s)$ to guide the search, as well as to problems for which the search should be based on some combination of $StateViolation(s)$ and some other function $Cost(s, t)$. In that case the combination can be done in many ways. One reasonable way is to simply do a linear combination of the two like in Equation 2.

$$\mathit{Objective}(s, t) = \mathit{StateViolation}(s) + \alpha(t) \cdot \mathit{Cost}(s, t) \tag{2}$$

Finding the optimal weight $\alpha(t)$ can then be a very difficult task and as we shall see in future chapters when I let $\alpha(t)$ be a decaying function we find quite good results for the Water Retention on Magic Squares-problem.

One problem with this basic algorithm is that it runs a high risk of getting trapped in a local minimum. In the next sub-chapter we'll look at how Tabu Search, which is a concept in very common use, can be used to avoid this pitfall.

3.2 Tabu Search

Tabu Search [3] is a general meta heuristic which can be applied to most local search methods including Constraint-Based Local Search. The big idea of Tabu Search is to ban recently changed elements or moves from being considered by the search algorithm to overcome situations where the search would get stuck in a loop, going back and forth between two or more states because the move is the locally best choice. This allows the search to get far enough away from the loop so it takes a different route through the search space. The main difficulty of using Tabu Search is to balance the number of iterations to block an element against the need to revisit it because it actually is the better route. Let's consider a very small example of an optimisation problem where we are to maximise the value.

Example 3. *Let A, B, C, D be 4 states with objective function Objective(A) $= 5$, Objective(B) $= 7$, Objective(C) $= 1$, Objective(D) $= 10$ and let A be the starting state. Valid moves are one step to the left in the alphabet or one step to the right. We clearly want to get to state D which has the maximum objective value of the 4 states. The first move takes us to state B since it's the only valid move. The next move we look at the two possible moves and see that if we go to A we get an objective value of 5 while if we go to C we only get an objective value of 1, hence we go to A. Clearly we will never reach D. If we introduce a Tabu Length of 1 where we ban the previous element, we will go straight to D in 3 moves because we are not permitted to turn back.*

An important note here is that had the example been constructed in a slightly different way, with the tabu search we might have started out in the wrong direction with no ability to turn back. For more advanced problems, a Tabu List may need to be bigger to overcome the sizes of the local looping structures in the search space. This is an engineering aspect to Tabu Search as it needs to be tuned right to work at its best.

Algorithm 4 gives an example of using tabu search in conjunction with CBLS. This time the CBLS works only with swap moves to make the machinery of the Tabu Search implementation more clear. Note that CBLS-TABU is a template algorithm where α and $Cost$ and the constraints handled by $StateViolation$ may be filled with arbitrary functions and constraints respectively.

4 Model and Algorithm

In chapters 2 and 3 we have seen all the basic methods which I will use to construct the algorithms to solve the Water Retention on Magic Squares-problem. We'll begin by defining the model and then we'll proceed by looking at the most naive algorithm which will be a baseline for the other methods. Then we'll try to use more intelligent methods to maximise the water retention, resulting in one algorithm which uses the real water retention value as the cost-function and another which uses an approximation pattern.

Algorithm 4 Constraint-Based Local Search Algorithm with Tabu Search

```
CBLS-TABU($s, iterations, tabulength$)
// **Signature:** Initial state $s$, iterations and tabu length $\rightarrow$ new state $s'$.
// **Pre:** $s$ is a valid state w.r.t. the domains of all the variables within.
// **Post:** $s'$ is a state satisfying the constraints or a timeout error was signalled.
Let $t$ be an integer
$t := 0$
Let $TABU[1..|s|]$ be an array
Initialize $TABU[*] = 0$
**while** ($StateViolation(s) > 0$ and $t < iterations$ )
    $SelectMin(n \in N_{Swap}(s),$ where elements $x_1$ and $x_2$ being swapped ,
    $TABU[index(x_1)] \leq t$ and $TABU[index(x_2)] \leq t$
    $: StateViolation(n) + \alpha(t) \cdot Cost(n, t))$

    $s := n$ // Make the selected $n$ the new current state
    $t := t + 1$
    $TABU[index(x_1)] := t + tabulength$ // Make swapped elements tabu
    $TABU[index(x_2)] := t + tabulength$
$s' := s$ // Copy state $s$ and assign to $s'$
**if** $StateViolation(s) > 0$
    TIMEOUT // Signal failure
```

4.1 The Model

The model [1] I've selected for the problem is the one where we consider the space of all permutations of a square which satisfy the DISTINCT-constraint. The invariant of satisfying DISTINCT is preserved at all times by starting out with a randomly generated permutation where every number in the range $\{1..n^2\}$ occurs in the square exactly once and then we perform a local search with the swap-operator that swaps the value of two elements of the square each iteration preserving the invariant. We are then left with a system of SUM-constraints (See Definition 1) to satisfy, namely that every row, every column and the two main diagonals must sum to the same constant m. The m in this case is called the magic constant [2] and is given as a function of the order of the square n in Equation 3.

$$m(n) = \frac{n(n^2 + 1)}{2} \tag{3}$$

4.2 The Main Algorithm

Algorithm 5 is the common algorithm we'll use for the 3 different approaches of the next 3 sub-chapters. We'll make use of the CBLS-TABU template algorithm and plug in the 3 different objective functions. Note that this algorithm stops searching as soon as it finds a magic square. It would be a simple matter to allow it to continue until it times

Algorithm 5 Magic Square with Water Retention Search Algorithm

```
MS-WR-SEARCH($size, iterations, tabulength$)
 1  // **Signature:** Problem size, iterations and tabu length $\rightarrow$ ($int[1..size, 1..size], int$).
 2  // **Pre:** $size > 0, iterations > 0, tabulength > 0$.
 3  // **Post:** CBLS-TABU signals TIMEOUT or a magic square and retention returned.
 4  Let m be an integer
 5  $m := size(size^2 + 1)/2$
 6  Let $s$ be a square $matrix[1..size, 1..size]$ of integers
 7  Initialize $s[*, *]$ as random permutation with distinct elements $\{1..size^2\}$
 8
 9  // Post constraints to global constraint system used by CBLS-TABU
10  for $i$ from 1 to $size$
11        POST(SUM($s[i, *]$) $= m$) // Rows
12        POST(SUM($s[*, i]$) $= m$) // Columns
13  POST(SUM($s[1, 1], s[i, i], .., s[size, size]$) $= m$) // Right Diagonal
14  POST(SUM($s[1, size], s[i, size - i + 1], .., s[size, 1]$) $= m$) // Left Diagonal
15
16  // Run search and return result on success
17  CBLS-TABU($s, iterations, tabulength$)
18  return (s, WATER-RETENTION($s$))
```

out and keep track of the best magic square found. This could allow the algorithm to improve on a good solution instead of starting fresh with another random permutation.

4.3 The Naive Objective

For the most naive approach, let's define the α and $Cost(s)$ of CBLS-TABU as follows:

$$\alpha(t) = 0, \quad Cost(s, t) = 0 \tag{4}$$

$$Objective(s, t) = StateViolation(s) \tag{5}$$

The constant function 0 will serve as weight function and cost function. This objective function reduces the problem to a pure Constraint Satisfaction Problem which has no consideration for water retention during search and simply checks whatever retention we get afterwards. Next we'll examine a more elaborate method.

4.4 The Retention Objective

This objective function is the best I've found during the project if only the water retention values found is concerned. It's also the slowest with a wide margin because we have to calculate the water retention-Δ for every possible swap-pair. $O(n^4)$ swap-pairs and a water WATER-RETENTION-algorithm which is $O(n^4)$ yields an $O(n^8)$-algorithm which becomes problematic for larger problem sizes.

We'll begin by defining the cost function with a slight abuse of notation (We implicitly assume here that the state s is in the form of a square matrix so it can map to the parameter of WATER-RETENTION to avoid serious clutter in the definitions. We also at this point assume that there is some water level matrix W_{aux} with the same dimensions as s to pass into the procedure.) :

$$Cost(s,t) = -\text{WATER-RETENTION}(s, W_{aux}, n, n^2) \tag{6}$$

The fact that we introduce the minus-sign in front is because CBLS-TABU is minimising the $StateViolation(s)$ and since we want to maximise retention we must make it negative.

The weight-factor α is defined as a function as is described in Equation 7.

$$\alpha(t) = (0.99^t) \cdot w_{initial} \tag{7}$$

The caller of this algorithm provides an initial weight for the water retention which then decays iteration-by-iteration. This makes the algorithm first go after high retention values and gradually shift focus to optimising for constraint satisfaction. This is what made the water retention as a cost-function work because if the weight is too low, it will not find a good retention value. If the weight is high the constraints are not normally able to be satisfied so we can't find a valid solution. Let's write down the complete Objective-function for this method.

$$Objective(s,t) = StateViolation(s) + (0.99^t) \cdot w_{initial} \cdot Cost(s,t) \tag{8}$$

One problem with using the water retention as the objective function is that we may be in close vicinity of the maximum retention magic square without suspecting it. Experiments show every good magic square has a lot of low retention squares in its neighbourhood. If we are 2 swaps away and the in-between swap has a low retention this method won't find it.

Next we will see an approximation objective function which shares some ideas with the retention objective function.

4.5 The Approximation Objective

The retention method from the previous sub-chapter works very well but its $O(n^4)$ objective function time complexity is prohibitive when the order increases as the time statistics in Table 7 shows. For larger problem sizes an approximation method is the only viable method.

This approximation makes use of the Manhattan distance metric together with the value of the element to optimise for high values in a diamond shape near the edges and small values in the middle to create a large lake. Looking at highly retaining magic squares, they all share this structure.

$$Distance(n,i,j) = \begin{cases} \left| \left|i - \frac{n+1}{2}\right| + \left|j - \frac{n+1}{2}\right| - \frac{n+1}{2} \right| & \text{if n even} \\ \\ \left| \left|i - \frac{n+2}{2}\right| + \left|j - \frac{n+2}{2}\right| - \frac{n+2}{2} \right| & \text{if n odd} \end{cases} \tag{9}$$

The cost function uses the negative of the distance multiplied by the complement of each element as that places small values far from the diamond patterned walls and high values close.

$$cost(s,t) = \sum_{\forall(i,j)} -Distance(n,i,j) \cdot (n^2 - s[i,j]) \tag{10}$$

Just as for the retention objective function of the previous subsection we have a decaying weight function to handle the balancing issues which would cause trouble in the search.

$$\alpha(t) = (0.99^t) \cdot w_{initial} \tag{11}$$

Hence the final objective function is:

$$Objective(s,t) = StateViolation(s) + (0.99^t) \cdot w_{initial} \cdot Cost(s,t) \tag{12}$$

The distance function could use another distance metric than the one used here. Max-distance and Euclidean-distance are two other viable metrics that I've tried with good results for some problem sizes.

One missing piece of the puzzle is that for the highest retaining squares there are always ponds outside of the main wall in the corners adding a lot of water to the final retention. This is not captured by the pattern this method optimises for. Experiments where I compared the value of this metric for the best known squares of different orders with squares found with this method showed that the best known squares had a lower metric value than the inferior squares. This means that more work on this approximation method is required to make it perform really well.

4.6 Other Models

Another model I considered during the research was to stay within the space of valid magic squares and use operators which preserve the magic constraint satisfaction. The simplest such operator is to take the complement of every element value $e := n^2 - e + 1$. This neighbourhood is of size 1 so we don't reach much of the possible magic squares using it. Another operator which is more complex is to swap two rows and two columns on the opposite sides of the middle. This also preserves the satisfaction of the constraints. The neighbourhood of that operator is small as well for the small orders but quickly grows in size when the order increases. I made experiments with these operators and it didn't show promising results, as it appears to break the structures giving a high retention. Hence I didn't pursue the research on this type of model further.

5 Results and Analysis

In this chapter I will present some of the data I have accumulated during the project. The squares listed in Table 2, which are the highest retaining squares I've found during this project, are illustrated in Appendix A.

Order	My Best	Best Known	Gap	Normalised Gap
5	69	-	-	-
6	192	192	0	0
7	413	418	5	0.012
8	750	797	47	0.059
9	1323	1408	85	0.060
10	2071	2267	196	0.086
11	3134	3492	358	0.103

Table 2: My Best Squares from the Entire Project Against the Best Known.

5.1 Experiments

The experiments were conducted with Comet Language 2.1.1 on a MacBook with an Intel Core 2 Duo 2.26 GHz processor and 4GB 1067 MHz DDR3 RAM memory with 3MB of L2-cache. The two cores were mostly useful for allowing Comet to work undisturbed as the academic trial license I've been working under is restricted to run on one core. For each algorithm I've made 25 runs for each order in the interval [5..11] where I started from random permutations. For each run, I kept a record of the retention achieved, the run-time in seconds and the number of iterations needed to find the first magic square.

5.2 Data and Analysis

In the following sections I will display my experimental data to compare the different methods along with some analysis of what the numbers may signify.

Order	Max	Min	Avg	StdDev
5	38	2	15.56	8.32
6	110	12	53.68	27.11
7	164	42	108.80	27.59
8	312	110	192.88	45.27
9	573	242	392.68	79.46
10	1083	284	601.48	178.86
11	1350	598	914.80	146.25

Table 3: Naive Method - Water Retention Statistics (units of water).

Order	Max	Min	Avg	StdDev
5	69	34	56.32	10.42
6	178	134	162.04	11.57
7	396	319	364.68	20.09
8	741	621	682.24	33.74
9	1284	1098	1198.36	51.88
10	2071	1705	1925.88	82.56
11	3134	2577	2905.56	158.73

Table 4: Retention Method - Water Retention Statistics (units of water).

Order	Max	Min	Avg	StdDev
5	60	39	53.88	4.31
6	164	110	148.48	10.63
7	278	143	196.32	37.22
8	461	251	345.16	49.13
9	780	496	679.52	70.42
10	1402	742	1038.32	201.77
11	1610	825	1297.44	206.87

Table 5: Approximation Method - Water Retention Statistics (units of water).

Order	Max(s)	Min(s)	Avg(s)	StdDev(s)
5	0.14	0.003	0.03	0.03
6	0.10	0.01	0.03	0.02
7	0.12	0.01	0.04	0.02
8	0.25	0.03	0.08	0.05
9	0.21	0.05	0.10	0.05
10	0.49	0.07	0.16	0.09
11	0.83	0.12	0.29	0.19

Table 6: Naive Method - Time Statistics (seconds).

Order	Max(s)	Min(s)	Avg(s)	StdDev(s)
5	7.28	0.82	3.26	1.59
6	25.90	2.92	11.29	4.91
7	63.60	15.24	37.31	12.66
8	421.10	34.66	92.04	75.42
9	506.39	97.78	261.15	106.67
10	707.47	179.49	432.21	150.22
11	1563.64	478.58	900.83	325.92

Table 7: Retention Method - Time Statistics (seconds).

Order	Max(s)	Min(s)	Avg(s)	StdDev(s)
5	2.74	0.01	0.47	0.58
6	1.27	0.08	0.47	0.38
7	2.51	0.17	0.67	0.60
8	4.35	0.46	1.40	1.00
9	2.79	0.51	1.07	0.56
10	6.03	1.22	2.46	1.02
11	5.99	1.60	2.67	1.07

Table 8: Approximation Method - Time Statistics (seconds).

Order	Max(#)	Min(#)	Avg(#)	StdDev(#)
5	2037	26	342.72	482.03
6	632	26	179.20	151.38
7	471	26	135.84	100.55
8	602	26	175.96	121.69
9	333	42	147.36	87.95
10	569	41	155.04	116.52
11	776	45	215.60	184.50

Table 9: Naive Method - Iteration Count Statistics (iterations).

Order	Max(#)	Min(#)	Avg(#)	StdDev(#)
5	1354	81	486.68	354.33
6	1013	61	278.08	189.67
7	952	110	310.04	176.04
8	1328	117	263.84	234.03
9	777	135	289.84	159.12
10	775	128	270.44	143.58
11	697	126	258.40	121.89

Table 10: Retention Method - Iteration Count Statistics (iterations).

Order	Max(#)	Min(#)	Avg(#)	StdDev(#)
5	8566	35	1498.24	1806.67
6	2028	122	742.68	602.26
7	2000	137	536.04	477.22
8	2043	219	662.52	470.95
9	818	150	316.12	162.82
10	1119	226	455.68	189.70
11	760	201	336.60	136.34

Table 11: Approximation Method - Iteration Count Statistics (iterations).

5.2.1 Naive Method Data Analysis

The naive method has very low max, min and avg retention. On the other hand it is really fast compared to the retention method. It's also faster than the approximation method but the difference in speed between them is much less. Clearly not the best method to use in any situation.

5.2.2 Retention Method Data Analysis

The slowest method but also with a wide margin the one that finds the highest retention values of the algorithms. It starts to break down with problem sizes 12, 13, 14 .. as the time to perform every iteration in the search algorithm explodes, the algorithm being of complexity $O(n^8)$. By using a faster implementation than I have, for example a well tuned C++ program, particularly for the WATER-RETENTION procedure we could perhaps use it on a couple of larger sizes before hitting the wall. A crude experiment I performed showed the Comet implementation of the WATER-RETENTION procedure to be about 38 times slower than the C++ implementation by White [5]. The competition ran all the way up to $n = 28$ and this algorithm stands almost no chance of ever finding a solution there with reasonable hardware in reasonable time.

5.2.3 Approximation Method Data Analysis

The data shows that this approximation method is a substantial improvement over the naive method even though it falls short compared to the retention method. It's slightly slower than the naive method since it has more calculation, though constant time, to make for every potential swap. The iteration count is higher than both the naive and the retention algorithm on average and this might mean that the structure fitting the pattern is further away from being magic and thus more moves are required to achieve satisfaction of the magic constraints.

6 Conclusion

In this thesis project I have explored the Water Retention on Magic Squares-problem which is a combinatorial problem with many intrinsic difficulties when it comes to generating highly retaining solutions. I used Constraint-Based Local Search to solve the problem, a local search technology where every constraint has a violation function which gives its degree of dissatisfaction. These violation values can be used to guide the search. To avoid getting stuck around local optima I used Tabu Search on top of CBLS, a meta heuristic where the recently changed elements are banned from changing before some time has passed. I constructed a naive method which only used the magic constraints, a retention method which uses the actual water retention in the objective function and an approximation method using a diamond pattern which can be detected in all good solutions. The method which uses the actual water retention is the one which gives the highest retaining solutions while the approximation gives water retention values somewhere in between the naive method and the retention method while executing much faster than the retention method. Hence the approximation method can find solutions for magic squares of higher orders than the retention method hardware being the equal.

7 Future Work

For someone taking on this problem I would recommend reading van der Plancke's descriptions of his approximation method and try other approximation patterns which captures the ponds in the corners which the approximation method I used does not capture. It might be a good idea to find a way to shrink the neighbourhood without seriously damaging the algorithm's capability of finding a solution since the $O(n^4)$ neighbourhood grows very fast as the order increase. Another interesting topic to pursue might be to study how additional constraints might complicate the problem, and how the methods I've presented in this thesis work when the problem is further constrained.

Acknowledgements

I would like to thank my wonderful supervisors Pierre Flener and Jean-Noël Monette for their persistant help during my work on this hard problem. Thank you Craig Knecht for sending me a lot of useful information about the state of the research on this topic and for your website which has been an excellent resource. Thanks mom for bringing delicious dinners while I was busy writing this thesis and otherwise supporting this endevour.

References

[1] Dynadec. Comet documentation and code 2.1.1, 2009. The propreitary Comet-language contains a sample model for generating magic squares.

[2] L. Euler. On magic squares (de quadratis magicis). *Commentationes arithmeticae*, 2:593–602, December 1849. Available as index E795, in www.eulerarchive.org.

[3] F. Glover and M. Laguna. Tabu search. In *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–150. John Wiley & Sons, 1993.

[4] C. Knecht. Magic square: Topographical model, 2007. Can be found under Topographical model-link at: www.knechtmagicsquare.paulscomputing.com. Existence and contents verified 2 June 2012.

[5] H. S. White. Magic squares. Can be found at: http://users.eastlink.ca/∼sharrywhite. Existence and contents verified 8 June 2012.

[6] N. J. A. Sloane. Number of different magic squares of order n that can be formed from the numbers $1, ..., n^2$. Can be found at url: http://oeis.org/A006052.

[7] F. van der Plancke, 2010. Existence and contents verified 25 May 2012. tech.groups.yahoo.com/group/AlZimmermannsProgrammingContests/message/4706.

[8] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* The MIT Press, 2005.

A My Best Squares

Here is the hall of fame. The highest retaining magic squares found during this project are illustrated here. They were all found with the retention method.

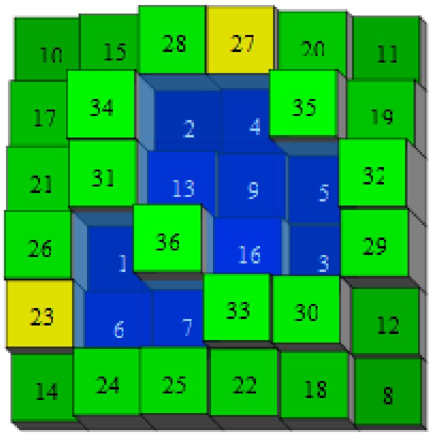

Figure 2: Magic Square of Order 6 Retaining 192 Units of Water.

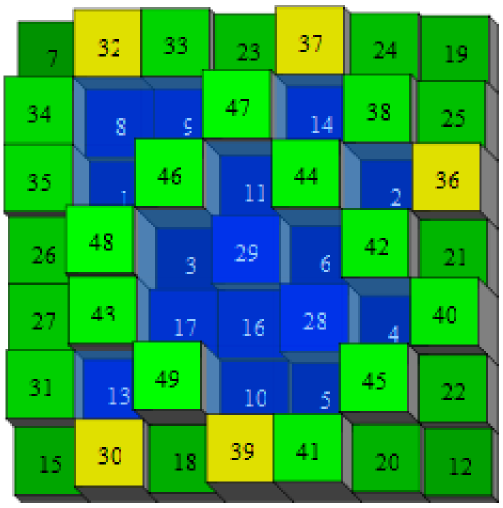

Figure 3: Magic Square of Order 7 Retaining 413 Units of Water.

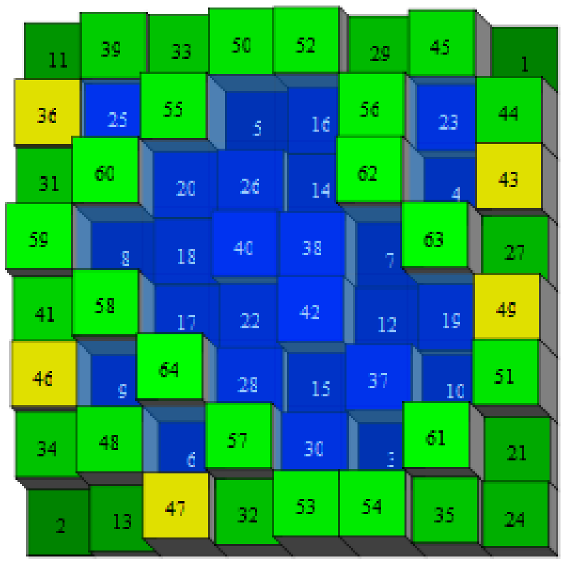
Figure 4: Magic Square of Order 8 Retaining 750 Units of Water.

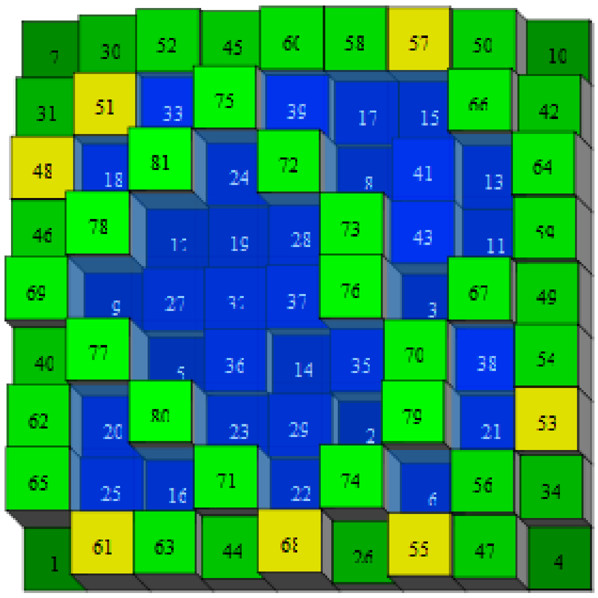
Figure 5: Magic Square of Order 9 Retaining 1323 Units of Water.

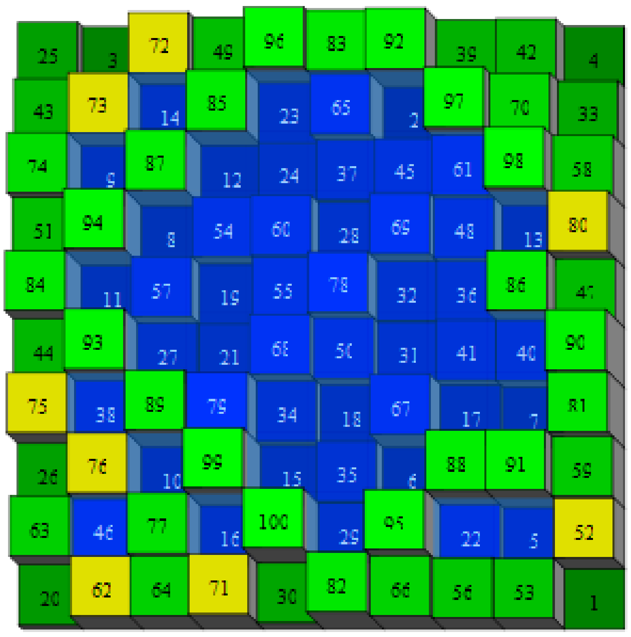

Figure 6: Magic Square of Order 10 Retaining 2071 Units of Water.

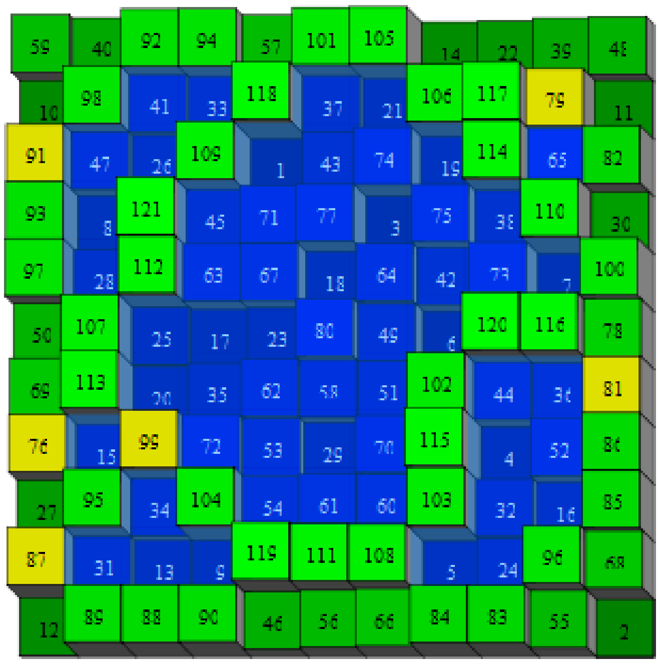

Figure 7: Magic Square of Order 11 Retaining 3134 Units of Water.