IT 11 076

Examensarbete 15 hp
Oktober 2011

Backbone Solver for Water Retaining Magic Squares via Constraint Based Local Search

Nicholas Baltzer

Institutionen för informationsteknologi
Department of Information Technology

Abstract

Backbone Solver for Water Retaining Magic Squares via Constraint Based Local Search

Nicholas Baltzer

Constraint programming [3] is a powerful tool for solving complex or computationally non-trivial problems. Providing a backbone model for these mathematically formulated problems, in this case the Magic Square with water retention capabilities, makes this tool more available to those who otherwise do not have the prerequisite experience, thus widening the field of use beyond current institutions. A Magic Square is a matrix where each cell has a value ranging from one to the number of cells. These values may only appear once in the matrix, and they must be placed such that the summation of each row, column, and major diagonal is the same. The water retention capacity [4] of a Magic Square is the volume that can be contained in this matrix if each index value is treated as a height. The model has been constructed using the Comet 2.1.0 software platform, based on a Local Search algorithm, since the problem search space increases too rapidly for a systematic search solution via Gecode [1]. Furthermore, two example constraints have been included beyond the scope of the basic solver.

Handledare: Pierre Flener
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
IT 11 076
Tryckt av: Reprocentralen ITC

Contents

1 Constraint Programming in Local Search **6**
1.1 The n-Queens problem 6

2 The Magic Square Water Retention Problem **8**
2.1 Symmetry and dependency 10

3 The problem **12**

4 The process **12**
4.1 The algorithm 12
4.2 The constraints 12
4.3 The Tabu search 14
4.4 The water retention calculations 16

5 Results **19**

6 Discussion **26**

7 Acknowledgments **27**

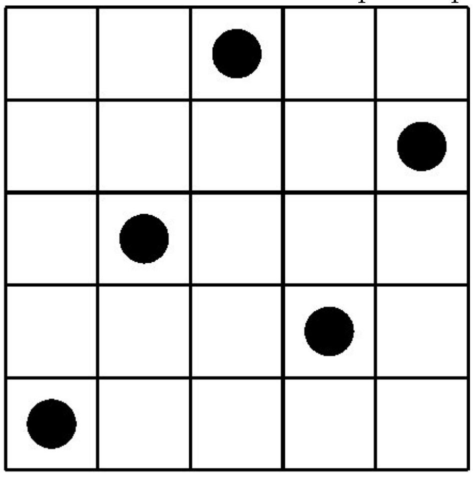
List of Figures

1 A solution to the five queens problem 6
2 A Magic Square with summations displayed 8
3 A 3x3 square randomly generated. 10
4 A 6x6 square randomly generated. 23
5 The same 6x6 square solved with Tabu length 6, retaining 78 units. . . . 23
6 A 7x7 square randomly generated. 24
7 The same 7x7 square solved with a diamond pattern constraint and Tabu length 7, retaining 165 units. 24
8 A 9x9 square randomly generated. 25
9 The same 9x9 square solved with a diamond pattern constraint, a low corner values constraint and Tabu length 9, retaining 849 units. 25

List of Tables

1 Iteration performance 21
2 Running time performance in milliseconds 21

Figure 1: A solution to the five queens problem



1 Constraint Programming in Local Search

Constraint programming [3] is a tool-set of algorithms that specify relations between variables. The basis of the paradigm is to implement relations between variables and specify the criteria for a solution rather than detail the process to reach a solution.

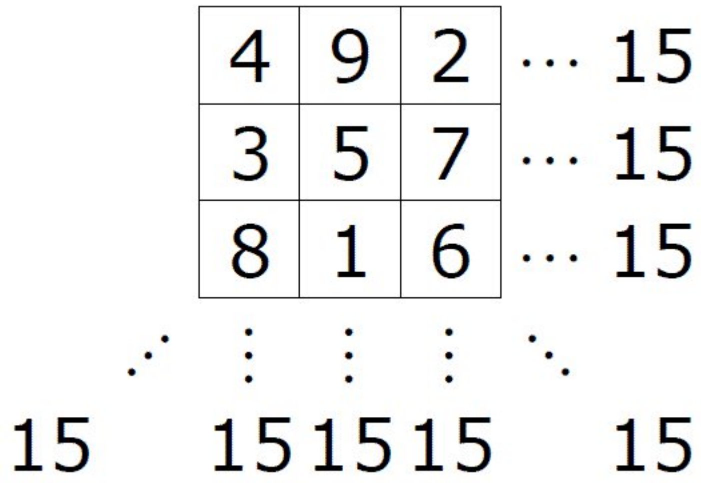
Local Search is an approach for providing a solution non-deterministically, that is, it will compute the best solution from a subset of the solution space. The concept is employed to find the best solutions when the solution space is too large to realistically sort through in a deterministic fashion. For this reason, the constraint solver Comet 2.1.0 [2] was used as it has the requisite tools to perform Local Search operations.

1.1 The n-Queens problem

Consider the following problem known as n Queens [5], originally introduced in 1850 by Carl Gauss. The goal is to place n queens on a Chess board of size $n \times n$ without having any queen threaten another. Thus, no queen may be placed on the same row, column, or diagonal. Using Constraint Programming deterministically, a solution is found by placing a single queen on the board, then removing from all other queens the possibility of placement on a threatening square. As each queen is placed on the board, the threatening squares are removed as possibilities from the remaining queens. The process is a two step sequence, first removing placements from the queens, and then placing a queen on the board. Should no square remain viable while one or more queens still need to be placed, the algorithm will backtrack to its most recent non-failed branching choice and pick an alternate path. If this algorithm is run multiple times, the same solution will always be presented.

Now, consider the problem from a Local Search perspective. Instead of placing the queens one by one on an initially empty board, all queens are placed randomly on the board at the start. Each queen will then receive a ”violation” value, indicating how many constraints it is breaking (in this case, how many other queens it threatens). To solve the problem, repeatedly move a queen onto a free square on the board such that the sum of these violations decreases. Once the sum of violations reaches zero, a solution has been found. By randomly placing queens on the board, the algorithm will not necessarily find the same solution if run multiple times in a row. Thus, consider the additional optimality constraint that queens should be placed as close to the bottom left corner as possible. With the random starting point some of these solutions will have queens closer to the bottom left corner than others, resulting in a different optimality value. The Local Search approach cannot guarantee that the solution provided is the optimal one, where as the deterministic approach can. However, if the board is of great size, the deterministic algorithm is unlikely to ever finish its calculations within human patience.

Figure 2: A Magic Square with summations displayed



2 The Magic Square Water Retention Problem

A Magic Square is a square of arbitrary size $n \times n$, though n is always greater than two. Each index of the Magic Square matrix holds a number in the range $(1..n^2)$ where n is the length of the matrix side. The "magic" component holds if the summations of each row, each column, and both main diagonals are the same.

While calculating a Magic Square is non-deterministic polynomial (NP)-hard; the sum each row, column, and diagonal should have is not. This value is given by the formula $n^2 \cdot (n^2 + 1) \div 2n$.
The water retention of a Magic Square is the volume contained by the non-peripheral indices. Consider the value of each index to be the height of a block on a three dimensional Magic Square. Water can only flow in straight directions, so one lower block will retain water if bordered by four blocks of greater height. Water will flow off the matrix should there be a path leading from a given block onto any block on the periphery of the matrix. The water retained on any given index is the minimum of the total height (height of the index + water retained on the index) of the four adjacent indices. If the Magic Square matrix is represented by m and the water retention matrix is represented by w, the formula for water retention is

$w_{i,j} = m_{k,l} + w_{k,l} - m_{i,j}$
where $m_{k,l} \in A(m_{i,j}) : m_{k,l} + w_{k,l} \leq m_{x,y} + w_{x,y}$
$\forall x \in (i - 1..i + 1), \forall y \in (j - 1..j + 1)$
and $A(m_{i,j})$ is the set of all indices adjacent to $m_{i,j}$.

Implemented via Comet 2.1.0, the constraint code is described below.

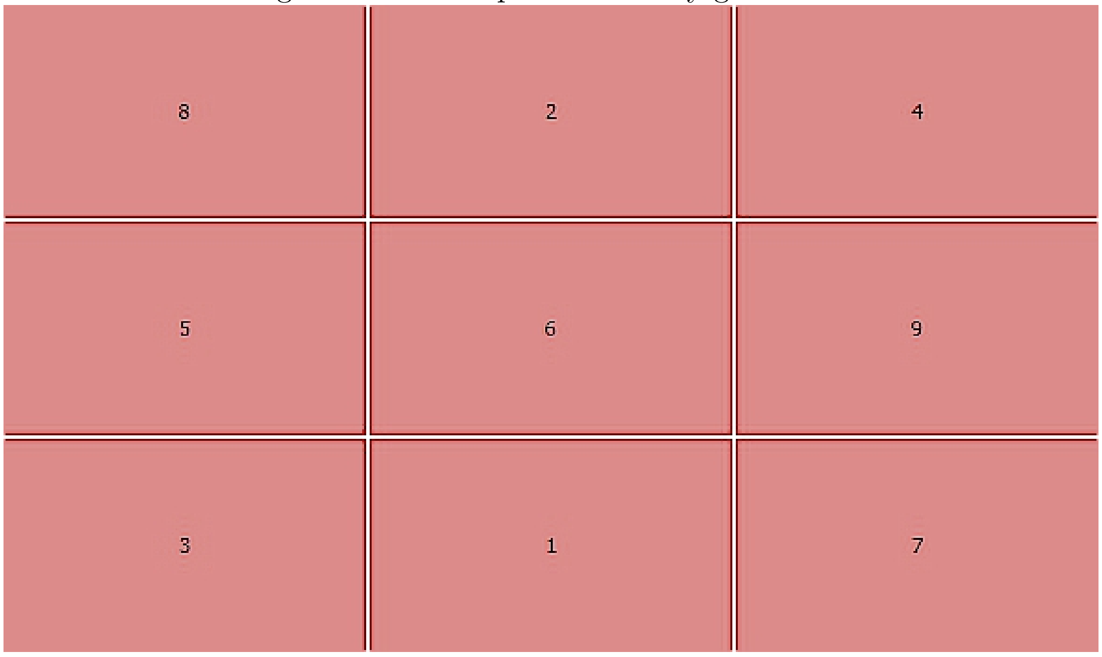
```
WATER RETENTION CONSTRAINT()
1  for each $i$ in $rInnerRange$
```

```
    **do for  each** $j$ **in** $rInnerRange$
        **do** $w[i,j] = \textsc{max}(m[i,j], \textsc{min}($
            $\textsc{min}(\textsc{max}(w[i+1,j], m[i+1,j]), \textsc{max}(w[i-1,j], m[i-1,j])),$
            $\textsc{min}(\textsc{max}(w[i,j+1], m[i,j+1]), \textsc{max}(w[i,j-1], m[i,j-1]))));$
```

Likewise, the total water retention is given by the sum of water on all the indices.

The problem is thus one of optimisation. The Magic Square constraints greatly reduce the possibilities of value placement on the indices, and introduce symmetries in the matrix. Every changed value in the matrix will affect the values in the same row, the same column, and possibly one or more diagonals. Furthermore, each change in a value might affect all non-periphery water retention values.

Figure 3: A 3x3 square randomly generated.



8	2	4
5	6	9
3	1	7

2.1 Symmetry and dependency

Consider Fig. 3, a 3×3 Magic Square randomly generated. By using the formula $n^2 \cdot (n^2 + 1) \div 2n$, we know the sum of each row, column, and main diagonal should be $3^2 \cdot (3^2 + 1) \div 2 \cdot 3 = 15$.
Currently, the first row has a sum of 14, which means that any index of that row can be preferentially swapped with another index if its value is one more. Thus, 8 can be swapped with 9, 2 can be swapped with 3, and 4 can be swapped with 5. All of these swaps will reduce the number of violations for the first row. Looking at the first column, the sum is 16 with values 8, 5, and 3. The preferred swaps will thus reduce the sum by 1, and preferred values for the indices will be 1 less than current values, namely 7, 4, and 2. Finally, looking at the right diagonal, the sum is 13, with preferred swaps being at least 1 point greater than the current values. The sums are listed below.

Row_1: 14, Row_2: 20, Row_3: 12
$Column_1$: 16, $Column_2$: 9, $Column_3$: 20
$Diagonal_{left}$: 21, $Diagonal_{right}$: 13

Swapping the values 5 and 4 is a good move, since it reduces violations in row 1, column 1, and the right diagonal. However, doing this will also change the preferred values in row 1, column 1, and the right diagonal. Furthermore, it will change the preferred values in column 3. Row 1 and column 1 no longer need to change anything at all, so there are no preferable swaps for them. Any swap that includes these two will increase the number of violations, making them less likely to be selected for further swaps (especially in larger matrices). Since column 3 has a larger sum than before the move, the number of preferred swaps has increased. For instance, the bottom right index of value 7 can now be swapped with its adjacent index of value 1 for a greater reduction in violations. The sums after the move are listed below.

Row_1: 15, Row_2: 19, Row_3: 12
$Column_1$: 15, $Column_2$: 9, $Column_3$: 21
$Diagonal_{left}$: 21, $Diagonal_{right}$: 14

Thus, swapping the values 5 and 4 in the 3×3 matrix has indirectly changed all preferred values, and the likelihood of any index being selected for swapping.

3 The problem

Forming a Magic Square is computationally intensive. Each index is affected by row and column constraints, and some are also affected by diagonal constraints. Moving one index value will affect a minimum of one row and one column, which affects the possible zero-violation values of each index on this row and column, which in turn affect the zero-violation values on each row, column and diagonal. Thus, swapping two values will change all possible solutions for all other indices.

4 The process

The program consists of a visual element, a set of constraints, a meta-heuristic algorithm based on Tabu Search [6], and a post-process water computation.

4.1 The algorithm

The Magic Square algorithm implemented in this project is described in pseudo-code below.

```
Magic Square Algorithm()
1   Post the required constraints for Magic Squares.
2   Generate a random matrix.
3   while conflicts are not resolved
4       do Get the most conflicting index that is not Tabu locked.
5          Get the unlocked index which when switched with the previous
6              will reduce the conflicts the most.
7          Swap the two index values.
8          Tabu lock both indices selected in this iteration.
```

4.2 The constraints

Apart from the search algorithm, constraints to enforce the Magic Square properties are needed. Line one of the pseudo code posts the following constraints.
For each row and for each column the total value must match the given formula such that $T = n^2 \cdot (n^2 + 1) \div 2n$. Indices are identified by their coordinate enumerators $i, j, k, l \in E$ and E is the set of all index enumerators.

$\sum_{i=1}^{n} x_{ij} = T$ for each row j, and

$\sum_{j=1}^{n} x_{ij} = T$ for each column i. The major diagonals are given by

$\sum_{i=1}^{n} x_{ii} = T$ and $\sum_{i=1}^{n} x_{i(n-i+1)} = T$

Last, all indices must hold unique values, phrased as $x_{ij} \neq x_{kl} : i \neq k \vee j \neq l$

In the program, these constraints are written using the Comet inbuilt functions. n represents the specified size of the matrix.

```
CLASS WRS()
  Solver<LS> $m$;
  RandomPermutation $distr$;
  ConstraintSystem<LS> $S$;

  int $n$;
  int $nLineSum$;
  int $nWaterAmount$;
  int $nTotalWater$;
  int $nTabuLength$;
  int $nwTabuLength$;
  int $it$;
  int $nRut$;
  int $nReset$;
  int $nOldViolation$;

  range $Size$;
  range $rSquareRange$;
  range $rInnerRange$;

  int[,] $w$;
  int[,] $nLakeTrace$;
  int[,] $nTabu$;
  int[,] $nwTabu$;

  var{int}[,] $m$;

  string $s$;

  CometVisualizer $vWindow$;

  VisualTextTable $ttTable$;

  WRS(int $nSize$, int $nTLength$);
  int LowerBound(int $i$, int $j$, int $x$, int $y$);
  void PostConstraints();
  void CreateUI();
  void Search();
  void ComputeWater();
  void UpdateUI();
  int GetTotalWater();
```

```
void PostPatternConstraints();
```

Many of these variables are initialised during the model object initialisation. The relevant parts of this function will therefore be shown together with the code where it is used (as seen below). For clarity, variable type is displayed in these code snippets.

```
WRS::WRS(.)
..
range $Size = 1..n$;
int $T = n^2 \cdot (n^2 + 1) \div 2n$;
RandomPermutation $distr$ = new RandomPermutation($1..n \cdot n$);
int{var}[,] $m$ =new var{int}[$i$ in $Size, j$ in $Size$]($solver, 1..n \cdot n$) := $distr$.get();
...
```

```
void WRS::PostConstraints()
for each $i$ in $Size$
    do $S$.post(sum($j$ in $Size$) $m[i, j] == T$);
       $S$.post(sum($j$ in $Size$) $m[j, i] == T$);

$S$.post(sum($i$ in $Size$) $m[i, i] == T$);
$S$.post(sum($i$ in $Size$) $m[i, n - i + 1] == T$);
$S$.post(alldifferent(all($i$ in $Size, j$ in $Size$) $m[i, j]$));
```

4.3 The Tabu search

The Tabu search aspect will find the index that violates the most constraints, assuming that it is not Tabu locked. Once this index has been found, a second search will start, looking for an unlocked index to swap with the first. The criterion for the second index is to reduce the number of violations more than any other index, such that

> $swap(m_{i,j}, m_{k,l}) \geq swap(m_{i,j}, m_{x,y}) : \forall x, y \in E$ where E is the set of all index enumerators and $swap(m_{i,j}, m_{k,l})$ will swap the values of the two indices $m_{i,j}$ and $m_{k,l}$ with each other. Again, m represents the Magic Square matrix.

Once swapped, both indices selected are Tabu locked with the current iteration number and cannot be selected again until the lock has expired. The lock is a simple index table (called *nTabu*) where each index corresponds to its match in the Magic Square matrix table m. When an index becomes locked, the index in the lock table is set to the current value of the iteration counter (called *it*) plus the value of the Tabu lock length (called *nTabuLength*). That is, if index a becomes locked on the tenth iteration and the Tabu lock length is 20, the index value for a in the lock table will be set to 30. After that, index a cannot be selected again until the thirtieth iteration.

Apart from the direct search functions, there are other elements included to allow for

greater flexibility.
Before initiating the Tabu search, a randomisation is performed on the Magic Square matrix so that it may be called independently of location in the program and still return a valid result.
Once the while loop has started, a check is made on each iteration to see if the search is proceeding towards a solution with fewer constraint violations. If no progress has been made in a certain amount of iterations (default value at 50 iterations), the Magic Square matrix will be re-permuted and the search will start again from this new location in solution space.

```
WRS::WRS()
...
range $Size = 1..n$;
int[n,n] $nTabu$;
int $nOldViolation = 0$;
int $nRut = 0$;
int $nReset = 50$;
int $nTabuLength = nTLength$;
int $it = 0$;
RandomPermutation $distr$ = new RandomPermutation($1..n \cdot n$);
...
```

```
void WRS::Search()
for each $p$ in $Size$
    do for each $q$ in $Size$
        do $[p, q] := distr.\text{GET}()$;
            $nTabu[p, q] = 0$;

while VIOLATIONS($m$) $\neq 0$
    do if $nOldViolation == S.\text{VIOLATIONS}()$
        then $nRut = nRut + 1$;

        else $nRut = 0$;

    $nOldViolation = S.\text{VIOLATIONS}()$;
    SelectMin($i, j$ in $Size$ : VIOLATIONS($m[i, j]) \leq 0$, and $nTabu[i, j] \leq it$,
    $k, l$ in $Size$ : $nTabu[k, l] \leq it$ and $(k \neq i$ or $\neq j)$)
    (getSwapDelta($m[i, j], m[k, l]$)){
        $[i, j] :=: m[k, l]$
        $nTabu[i, j] = it + nTabuLength$;
        $nTabu[k, l] = it + nTabuLength$;
    }
    $it++$;
    if $nRut \geq nReset$
        then RandomPermutation $distr(1..n \cdot n)$;
            for $p$ in $Size, q$ in $Size$
            do $m[p,q] := distr$.GET();
                $nTabu[p,q] = 0;

        $nRut = 0$;
```

4.4 The water retention calculations

When a solution satisfying the Magic Square constraints has been found by the Tabu search, the water levels of the matrix are calculated in three distinct steps using the water retention algorithm implemented in this project. First, the matrix w holding the water values is equalled at all periphery indices to the Magic Square matrix m.

```
CLASS WRS()
...
range $Size = 1..n$;
int{var}[,] $m$ =new var{int}[$i$ in $Size, j$ in $Size$]($solver, 1..n \cdot n$) := $distr$.GET();
int[,] $w$ =new int[$i$ in $Size, j$ in $Size$] = 0;
...
```

```
VOID WRS::COMPUTEWATER()
for each $i$ in $Size$
    do $w[i,1] = m[i,1]$;
        $w[1,i] = m[1,i]$;
        $w[n,i] = m[n,i]$;
        $w[i,n] = m[i,n]$;

...
```

Second, the starting points of the water are calculated. These are the indices where all adjacent indices are of greater height such that

$w_{ij} < w_{xy}\ \forall x \in (i-1..i+1), \forall y \in (j-1..j+1)$ where $w_{xy} \in A(w_{ij})$
and $A(w_{ij})$ is the set of all indices adjacent to w_{ij}.

As a result, only the water retention matrix w is needed for subsequent calculations, as it represents the total height of previous examples $(w+m)$. The implementation used in the program is listed below.

```
CLASS WRS()
...
range $rInnerRange = 2..n-1$;
int{var}[,] $m$ =new var{int}[$i$ in $Size, j$ in $Size$]($solver, 1..n \cdot n$) := $distr$.GET();
int[,] $w$ =new int[$i$ in $Size, j$ in $Size$] = 0;
```

```
...
```

```
VOID WRS::COMPUTEWATER()
...
for each $i$ in $rInnerRange$
    do for each $j$ in $rInnerRange$
            do $w[i,j] = \text{MAX}(m[i,j], \text{MIN}($
                $\text{MIN}(\text{MAX}(w[i+1,j], m[i+1,j]), \text{MAX}(w[i-1,j], m[i-1,j])),$
                $\text{MIN}(\text{MAX}(w[i,j+1], m[i,j+1]), \text{MAX}(w[i,j-1], m[i,j-1]))));$

...
```

Third, the overflow is calculated. The starting points of water will have their water value equal the Magic Square value of the lowest adjacent index ($\exists x, y : w_{ij} = m_{xy}$ and $m_{xy} \in A(m_{ij})$. From these overflow points, an exploration is made in each direction to find the lowest boundary greater than the starting point value w_{ij}.
Once a new boundary has been found, a trace map is formed that details the "lake" formed by flooding the starting point until its next boundary. This map can then be used to calculate the water levels of all indices in this "lake". Once the "lake" has been flooded by one level, the third step is repeated until no water level can be increased. Since an index can be the lowest boundary to one or more "lakes" and a higher boundary to other "lakes" at the same time, it is important to calculate "lakes" separately and not let one overflow into another unless they have the same lowest boundary point.

The third part of the water retention calculations, called LowerBound, is managed in a recursive loop called on the index adjacent to the lowest boundary of the "lake".
The LowerBound function works in the following manner:
First, a check is made to see if the current index lies on the periphery.
If it does, the index cannot be part of the lake, though it can be part of the boundary. As such, its value is returned as a candidate for the lowest boundary point.
If the index does not lie on the periphery, its height is compared to that of the starting point, and should it be higher, the lake map will be updated to include the current index as a boundary point while its value is returned as a candidate for the lowest boundary point.
If the lake map already has a value for the current index (a 1 indicating it is part of the new lake or a 2 indicating it is part of the new boundary) then an impossible value is returned.
Should none of these conditions be met, the current index will be part of the new lake, setting the lake map index to 1 and returning the minimum return value of four new LowerBound calls, one in each direction from the current index.

```
CLASS WRS()
...
range $Size = 1..n$;
int[,] $w$ =new int[$i$ in $Size$, $j$ in $Size$] = 0;
```

```
int[,] $nLakeTrace$ =new int[$i$ in $Size, j$ in $Size$] = 0;
...

int WRS::LowerBound(int $i$,int $j$,int $x$,int $y$)
if $i == 1$ or $j == 1$ or $i == n$ or $j == n$
   then return $w[i, j]$;

if $w[i, j] > w[x, y]$
   then $nLakeTrace[i, j] = 2$;
         return $w[i, j]$;

if $nLakeTrace[i, j] > 0$
   then return $n \cdot n + 1$;

$nLakeTrace[i, j] = 1$;
return min(min(LowerBound($i, j - 1, x, y$), LowerBound($i, j + 1, x, y$)),
  min(LowerBound($i - 1, j, x, y$), LowerBound($i + 1, j, x, y$)));
```

5 Results

Initial attempts to create a solver for the Magic Square water retention problem were created in a C++ environment using Gecode[1] as the Constraint Programming[3] library. While solutions were provided with this algorithm, the considerable expansion of the solution space was too great as the size of the problem increased, and thus the deterministic solver was abandoned in lieu of a Local Search paradigm. The new algorithm was implemented in Comet 2.1.0 [2].

The Local Search algorithm proved difficult to implement, due to the different approach of Comet 2.1.0 in regards to constraints. Furthermore, a problem was found in the platform software, causing a failure to update solutions unless a visual element was included. After these issues were successfully sidestepped, a working algorithm was attained. This algorithm, in order to comply with the Comet language, takes the form of a class, with all functions and variables global to that class.

During the implementation phase, several modifications were made to the original design. The water retention matrix was changed from a decision variable, included in the Tabu search optimisation, to a post-process calculation matrix since no direct benefit could be found in including it pre-solution. The choice of making Tabu moves based on a union of violations and water retention did not produce results of greater optimality than making the moves based purely on violations. This change reduces the complexity of the search procedure while retaining the same flexibility. As a result, several constraints and invariants could also be removed, speeding up the verification process in between index swaps. The post-process calculation was eventually formulated as a recursive conditional function, spreading out in each direction of the computing point in the form of a ripple.

To assist in extending the program, a separate constraint-posting function was added. The intention is for extensions to avoid damaging the base functions of the program, even though adaptation and modification of the base program is recommended for more experienced users.

The Magic Square solutions with water retention values are presented with the lake boundaries clearly delineated. There are two example optimisation procedures included in the function named PostPatternConstraints, though these are turned off by default. One is a set of constraints to enforce that indices in the corner of the matrix are of lower value, and one is a set of constraints to enforce a diamond shape of higher than average values in the matrix.

```
WRS::PostPatternConstraints()
1  int $nCornerMaxValue = 6$;
2  $S$.post($m[1,1] \leq nCornerMaxValue$);
3  $S$.post($m[1,n] \leq nCornerMaxValue$);
4  $S$.post($m[n,n] \leq nCornerMaxValue$);
5  $S$.post($m[n,1] \leq nCornerMaxValue$);
```

```

range $rCeiling = 1..(\text{int})ceil(n+1) \div 2$;
**for each** $i$ **in** $rCeiling$
    **do** $S.\textsc{post}(m[i, (n \div 2) + i] > (n \cdot n) - 3 \cdot n)$;
        $S.\textsc{post}(m[(n \div 2) + i, i] > (n \cdot n) - 3 \cdot n)$;
        $S.\textsc{post}(m[(n \div 2) - i + 2, i] > (n \cdot n) - 3 \cdot n)$;
        $S.\textsc{post}(m[n + 1 - i, (n \div 2) + i] > (n \cdot n) - 3 \cdot n)$;
```

Outside of constraints, the main procedure includes a lower bound condition for the acceptable water volume in the final solution. The program thus satisfies the requirements for a backbone (sometimes called a barebone) solver.

Table 1: Iteration performance

n	water record (best known)	min	avg	max	stdDev
7	162 (418)	193	3875	7718	4419.36
8	331 (794)	40	2112	4184	2403.18
9	535 (1408)	391	3286	5662	3692.74
10	933 (2267)	166	2829	4486	3042.32
11	1407 (3492)	321	2319	4646	2625.77
12	1847 (5185)	132	3174	5817	3590.21
13	2611 (7442)	376	3181	6390	3576.61
14	3785 (10397)	416	3811	8252	4468.11
15	5040 (14154)	112	3893	7264	4688.71
28	73207 (219822)	444	4081	7504	4440.67

Table 2: Running time performance in milliseconds

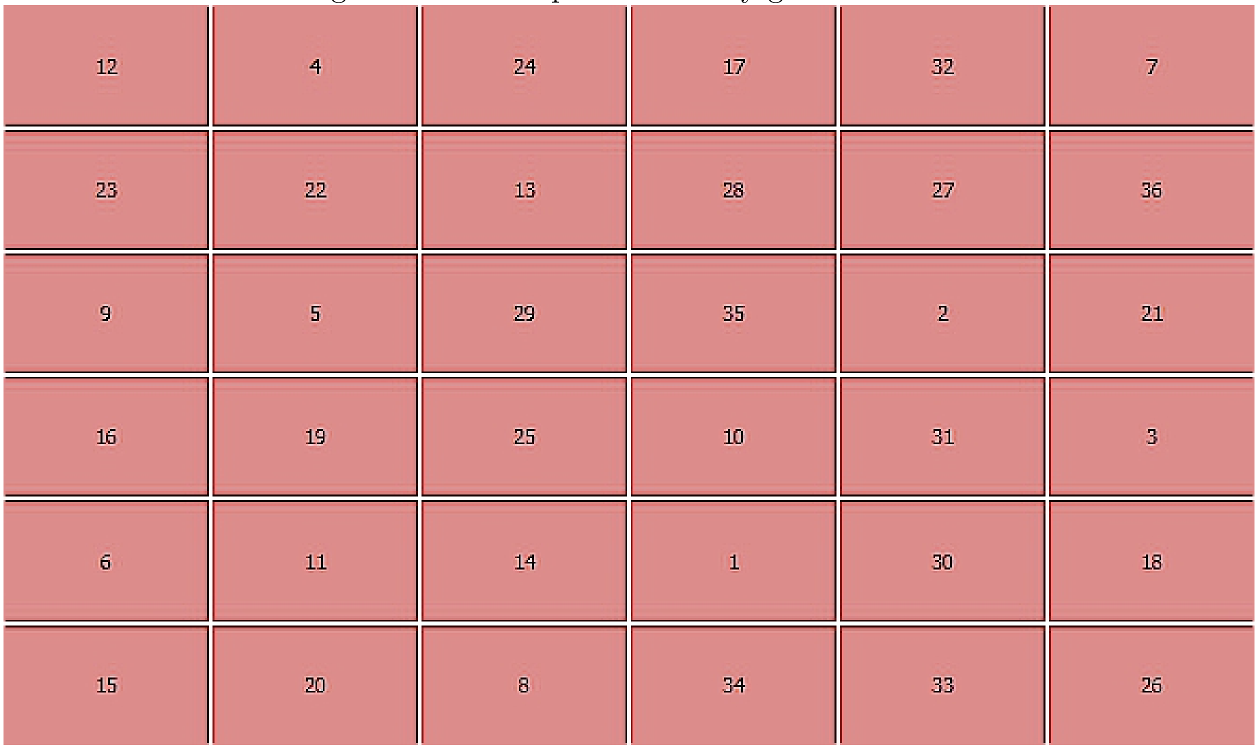
n	water record (best known)	min	avg	max	stdDev
7	167 (418)	31	126.68	358	150.61
8	349 (794)	31	195.92	484	229.01
9	506 (1408)	94	237.12	670	268.49
10	944 (2267)	125	395.04	827	432.43
11	1487 (3492)	187	514.80	951	549.06
12	1952 (5185)	343	846.80	3120	1034.18
13	2734 (7442)	452	1002.08	4274	1244.47
14	3749 (10397)	686	1332.92	2901	1415.31
15	4928 (14154)	1077	1837.68	3682	1961.14
28	71034 (219822)	23946	36473.56	75286	38050.26

Table 1 shows gathered data samples of the programs performance. This data has been gathered over 25 runs for each n represented in the table with no pattern constraints imposed. Table 1 lists the performance in terms of iterations and Table 2 lists the performance in terms of running time. The highest water capacity reached during these runs are given in the second column with the current best known value for that n given in parenthesis. Of note is that iterations are fairly constant over n, while the average running time is strictly increasing over n, growing by a factor of 287.9 when n is multiplied by four.

The results were gathered on a hardware platform with an Intel Core i7 920 processor, 6GB of memory, a Radeon HD4870 graphics card, and two Western Digital Caviar hard-drives in a RAID1 configuration. The Comet program (version 2.1.0) was run in a Windows 7 SP1 environment.

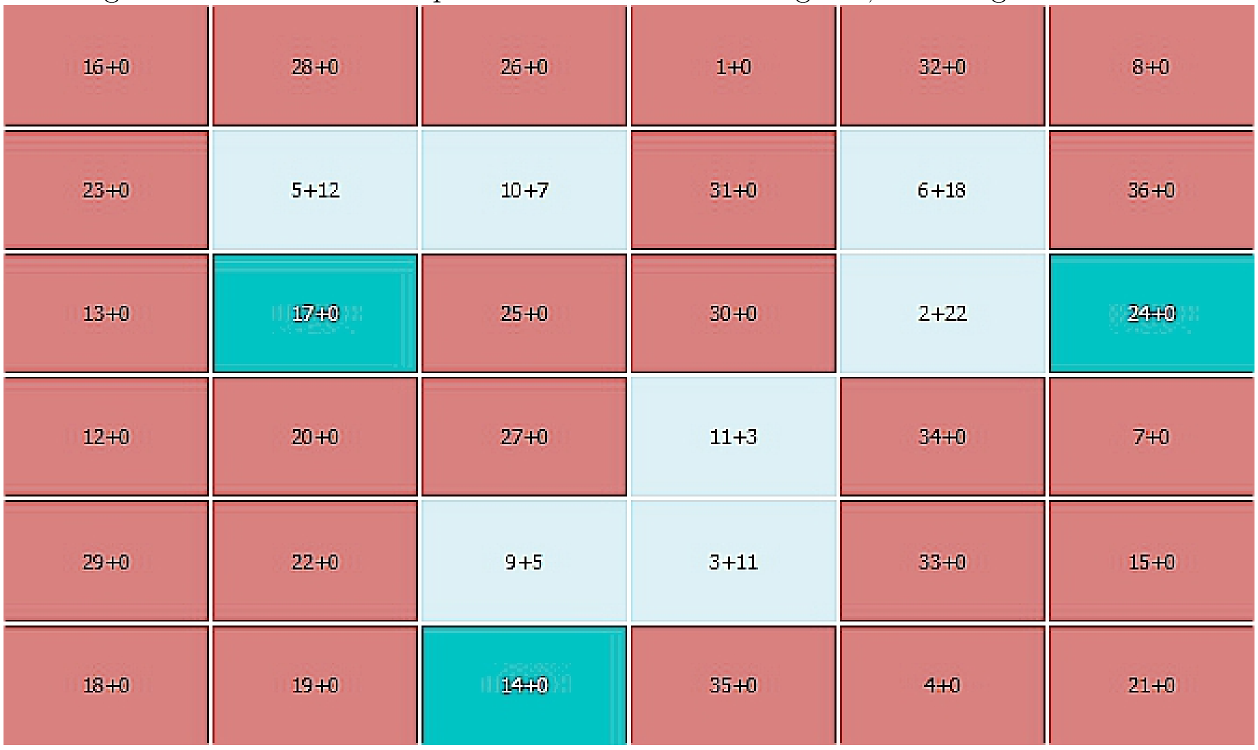
A set of solutions is included below, showing generated matrices before the search, as well as the end result. Some of these solutions make use of the included pattern constraints. Brown indicates an index with no water, blue indicates an index with water, and teal indicates a boundary point to the adjacent lake. Fig. 5 shows a solution without any pattern constraints, while Fig. 7 and 9 show solutions using a diamond pattern constraint.

Figure 4: A 6x6 square randomly generated.



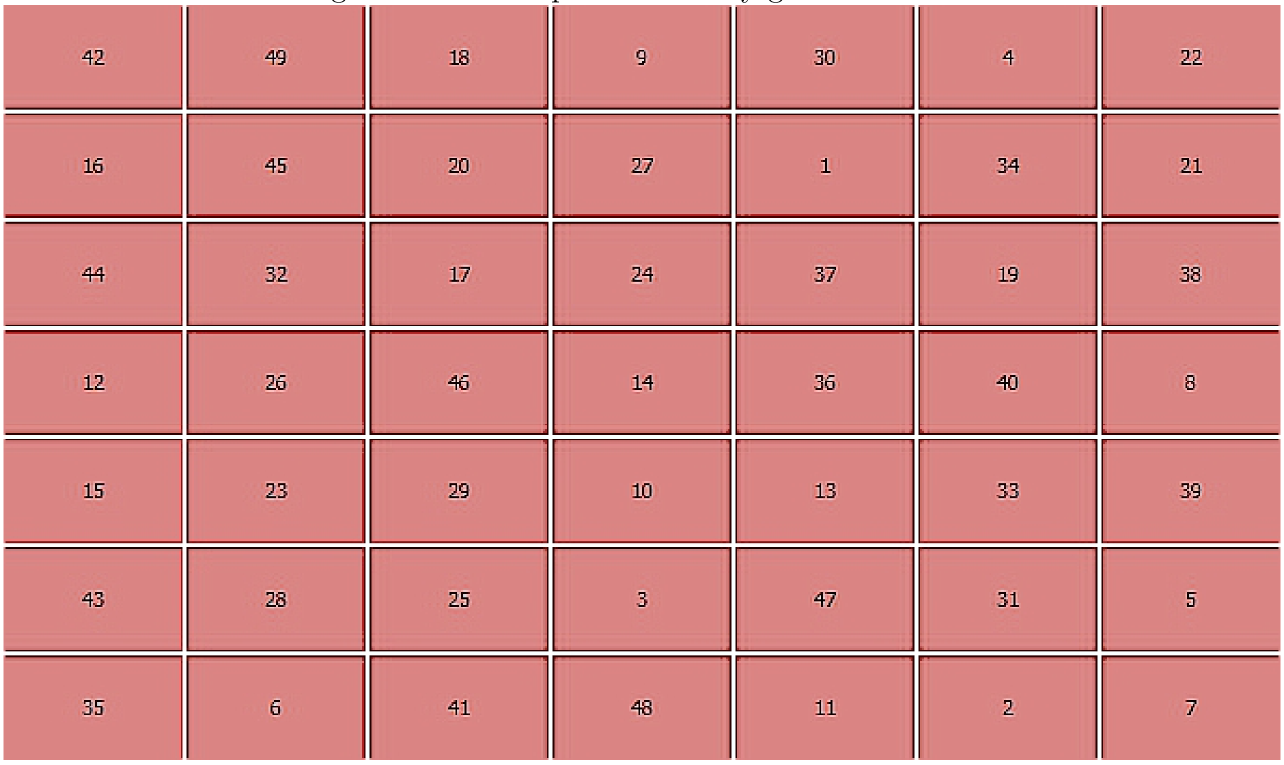
12	4	24	17	32	7
23	22	13	28	27	36
9	5	29	35	2	21
16	19	25	10	31	3
6	11	14	1	30	18
15	20	8	34	33	26

Figure 5: The same 6x6 square solved with Tabu length 6, retaining 78 units.



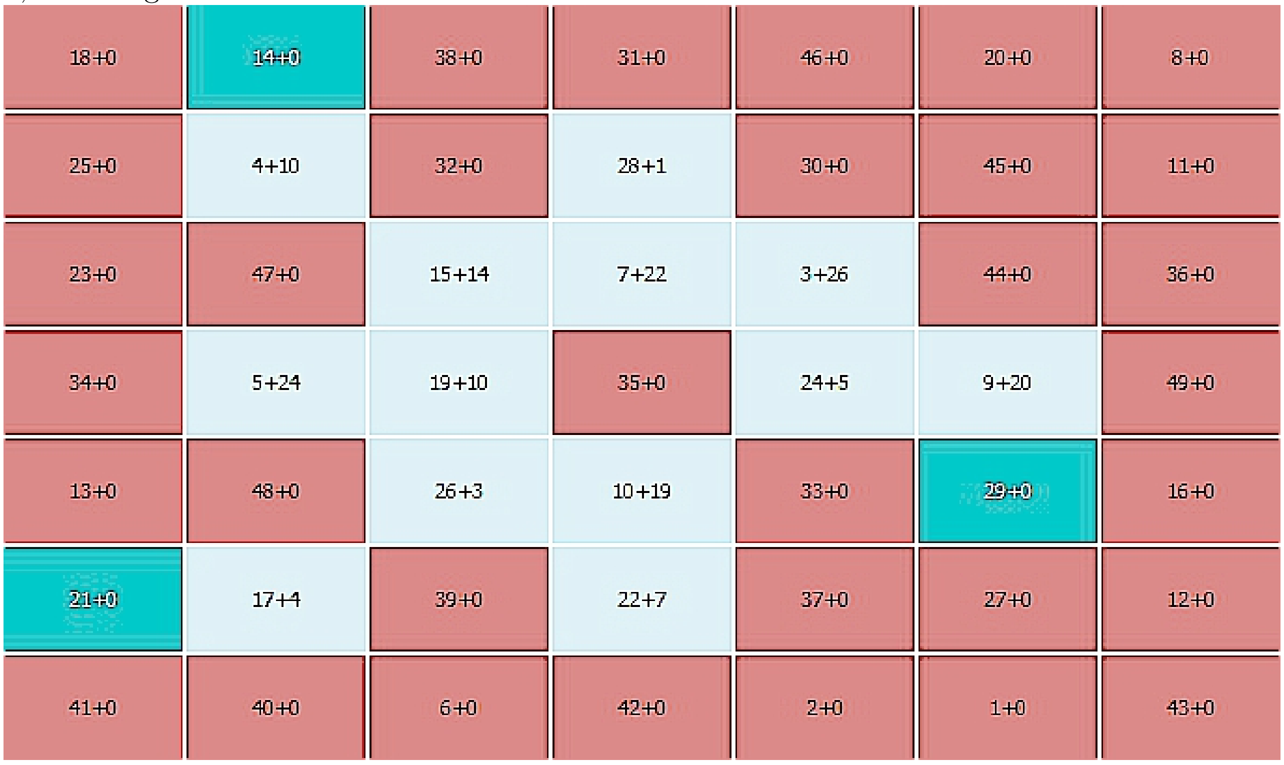
16+0	28+0	26+0	1+0	32+0	8+0
23+0	5+12	10+7	31+0	6+18	36+0
13+0	17+0	25+0	30+0	2+22	24+0
12+0	20+0	27+0	11+3	34+0	7+0
29+0	22+0	9+5	3+11	33+0	15+0
18+0	19+0	14+0	35+0	4+0	21+0

Figure 6: A 7x7 square randomly generated.



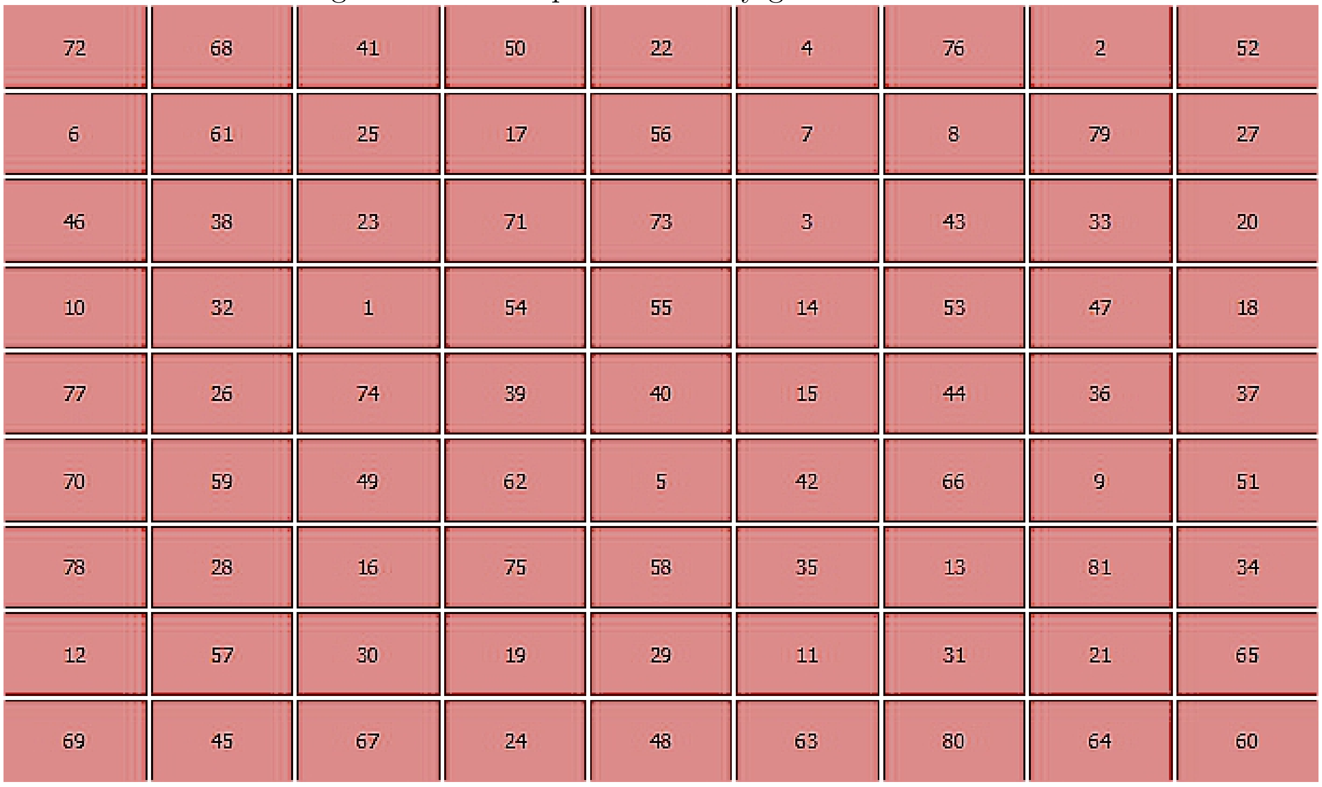
42	49	18	9	30	4	22
16	45	20	27	1	34	21
44	32	17	24	37	19	38
12	26	46	14	36	40	8
15	23	29	10	13	33	39
43	28	25	3	47	31	5
35	6	41	48	11	2	7

Figure 7: The same 7x7 square solved with a diamond pattern constraint and Tabu length 7, retaining 165 units.



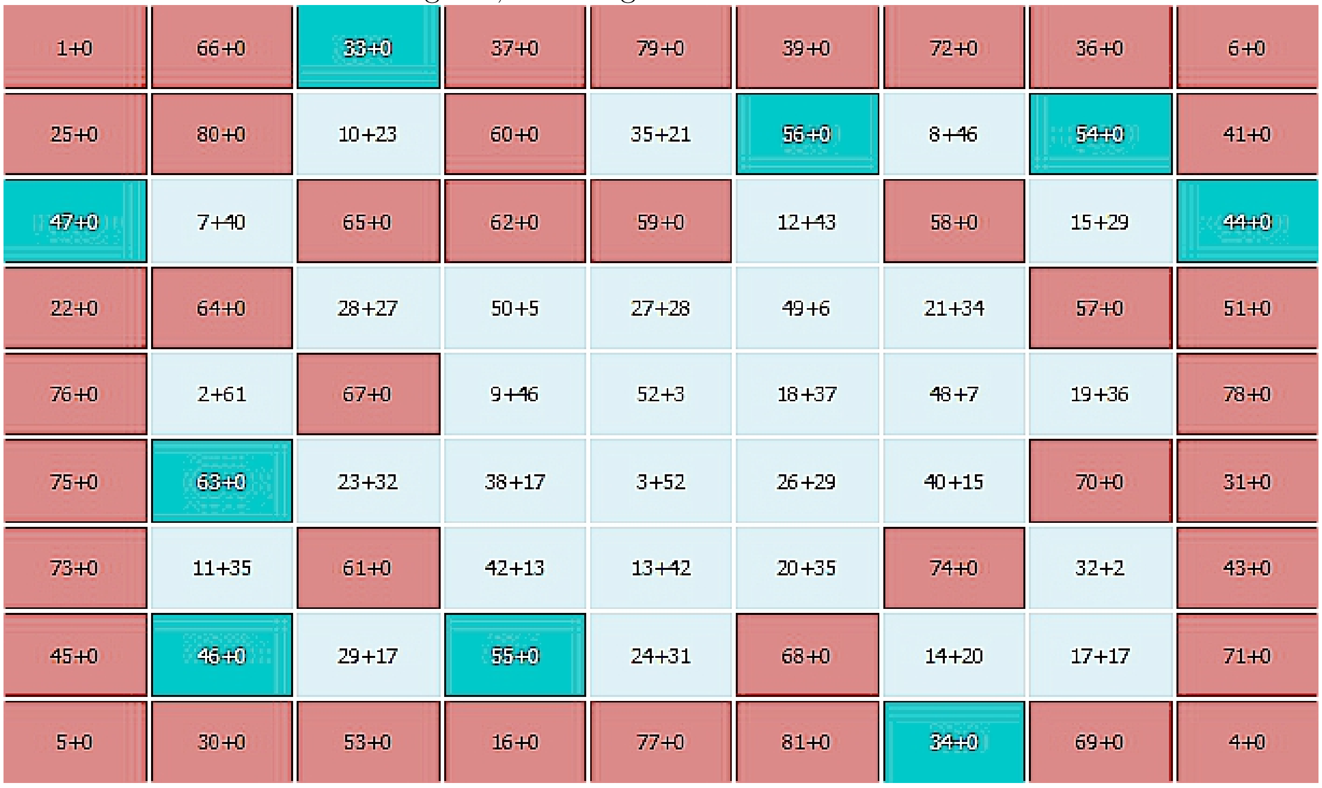
18+0	14+0	38+0	31+0	46+0	20+0	8+0
25+0	4+10	32+0	28+1	30+0	45+0	11+0
23+0	47+0	15+14	7+22	3+26	44+0	36+0
34+0	5+24	19+10	35+0	24+5	9+20	49+0
13+0	48+0	26+3	10+19	33+0	29+0	16+0
21+0	17+4	39+0	22+7	37+0	27+0	12+0
41+0	40+0	6+0	42+0	2+0	1+0	43+0

Figure 8: A 9x9 square randomly generated.



72	68	41	50	22	4	76	2	52
6	61	25	17	56	7	8	79	27
46	38	23	71	73	3	43	33	20
10	32	1	54	55	14	53	47	18
77	26	74	39	40	15	44	36	37
70	59	49	62	5	42	66	9	51
78	28	16	75	58	35	13	81	34
12	57	30	19	29	11	31	21	65
69	45	67	24	48	63	80	64	60

Figure 9: The same 9x9 square solved with a diamond pattern constraint, a low corner values constraint and Tabu length 9, retaining 849 units.



1+0	66+0	33+0	37+0	79+0	39+0	72+0	36+0	6+0
25+0	80+0	10+23	60+0	35+21	56+0	8+46	54+0	41+0
47+0	7+40	65+0	62+0	59+0	12+43	58+0	15+29	44+0
22+0	64+0	28+27	50+5	27+28	49+6	21+34	57+0	51+0
76+0	2+61	67+0	9+46	52+3	18+37	48+7	19+36	78+0
75+0	63+0	23+32	38+17	3+52	26+29	40+15	70+0	31+0
73+0	11+35	61+0	42+13	13+42	20+35	74+0	32+2	43+0
45+0	46+0	29+17	55+0	24+31	68+0	14+20	17+17	71+0
5+0	30+0	53+0	16+0	77+0	81+0	34+0	69+0	4+0

6 Discussion

It is possible that extending this program via Comet version 2.1.0 will prove complicated since during the implementation a platform bug was encountered. In particular, it forced the local search process to a halt unless the visual representation of the current iteration matrix was updated. Thus, for more extensive work, a rewrite of the program should be made to a more stable platform, whether that be an updated version of Comet or a different platform altogether. Until such a necessity however, the Comet API provides a set of pre-designed heuristics and other algorithms that will assist in creating smaller extensions without experience in the field. Furthermore, the class-based design of the program is well suited for extension, and a specific function within the class has been created for the explicit purpose of housing any new constraints implemented, called PostPatternConstraints.

Given the difficulty of finding global optima with Local Search procedures, constraints that further reduce the search space without compromising viable solutions need to be found. An early on correlation between water retention and Magic Square layout would narrow the search space in a weighted manner. Another improvement would be clear "always fail" patterns in the Magic Square itself that could be discarded from the search space via constraints.

The example pattern constraints included increase solution water capacity at the cost of speed. Solutions are found less frequently, but maintain an overall higher water retention (see Fig. 7 and Fig. 9 for examples of diamond pattern constraints). This area seems the most likely for extension in order to find higher water retention solutions in the lower end of the Magic Square size.

7 Acknowledgments

I would like to offer my sincere thanks to Mr. Farshid Hassani Bijarbooneh for his assistance with Comet and Gecode in both practical and theoretical aspects, and to Pierre Flener for providing me with his insights and experience in dealing with non-trivial problems. Last, my appreciation to Mr. Craig Knecht for his enthusiasm and helpful emails on the subject.

References

[1] C. Schulte et al. *Gecode: A generic constraint development environment, 2006.* Available from http://www.gecode.org/.

[2] Pascal Van Hentenryck, Dynadec. *Comet Studio.* Available from http://dynadec.com/support/downloads/.

[3] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

[4] C. Knecht. *Knecht Magic Squares Site, 2007.* Available from http://www.knechtmagicsquare.paulscomputing.com/.

[5] C. Letavec, J. Ruggiero. *The n-Queens Problem.* INFORMS Transactions on Education 2:3 (101-103) Available from http://archive.ite.journal.informs.org/Vol2No3/LetavecRuggiero/LetavecRuggiero.pdf

[6] F. Glover, M. Laguna, R. Mart. *Tabu Search.* Springer; 1 edition, 1998