Electronic Preprint for Journal of Information Processing Vol.20 No.3

Regular Paper

Rogo, a TSP-based Paper Puzzle: Optimization Approaches

Shane Dye[1,a)] Nicola Ward Petty[1,b)]

Received: June 24, 2011, Accepted: March 2, 2012

Abstract: Rogo® is a new type of mathematical puzzle, invented in 2009. Rogo is a prize-collecting subset-selection TSP on a grid. Grid squares can be blank, forbidden, or show a reward value. The object is to accumulate the biggest score using a given number of steps in a loop around the grid. This paper introduces Rogo as a discrete optimisation problem. An IP formulation is given for the problem with two alternative sets of subtour elimination constraints. Enumeration-based algorithms are also proposed based on properties of solutions and Rogo instances. Some results of computational experiments are reported.

Keywords: combinatorial optimzation, puzzles, enumeration algorithms

1. Introduction

Rogo® is an entirely new type of puzzle, played on rectangular grids. Grid squares can be blank, forbidden (black), or show a reward value. The object is to accumulate the biggest reward score using a given number of steps in a loop around the grid. The loop must return to the starting square and may not revisit a square. The best possible score for a puzzle is given with it, providing a fast check for puzzle solvers that they have the solution. Rogo puzzles can include forbidden squares (black), which must be avoided in the loop. **Figure 1** shows a simple Rogo and its best loop, which accumulates a score of 8. The 'good' score indicates an intermediate, but sub-optimal, level of attainment. A well designed Rogo will only have one subset of rewards leading to the best score. The nature of the grid means that there may be alternative routes through the blank squares to collect the same set of rewards.

The focus of this paper is to propose algorithms to find the best (optimal) loop for a given Rogo grid and a given loop length. A version giving an upper bound on the loop length is also investigated. A typical Rogo puzzle for amusement has a 12, 16 or 20 length loop with grid sizes from 4×5 to 10×14.

The following notation is used. Define grid $G_{m\times n} = \{(i, j) : 1 \le i \le m,\ 1 \le j \le n\}$. Let $R \subseteq G_{m\times n}$ be a set of reward locations with each $(i, j) \in R$ having reward value r_{ij}. For convenience, $r_{ij} = 0$ for $(i, j) \notin R$. Let $F \subseteq G_{m\times n}$ be the set of forbidden squares. Two grid locations (i_1, j_1) and (i_2, j_2) are adjacent if $i_1 = i_2$ and $|j_1 - j_2| = 1$ or $j_1 = j_2$ and $|i_1 - i_2| = 1$.

Definition 1: A **loop**, ℓ, of length L is an ordered sequence of unique locations in $G_{m\times n}$, $[(i_1, j_1), (i_2, j_2), \ldots, (i_L, j_L)]$, such that (i_k, j_k) is adjacent to (i_{k+1}, j_{k+1}) for each $k = 1, \ldots, L-1$ and (i_L, j_L) is adjacent to (i_1, j_1).

An instance of Rogo can be mathematically defined as follows.

Instance: Non-empty grid size $m \times n$, (even) loop length $L > 0$, non-empty reward set $R \subseteq G_{m\times n}$, with positive reward values $r_{ij} > \forall (i, j) \in R$, and, a possibly empty, forbidden square set $F \subseteq G_{m\times n}$, with $R \cap F = \emptyset$.
Solution: A subset of reward locations $S \subseteq R$ of maximum score, $\sum_{(i,j)\in S} r_{ij}$, such that there exists a loop, ℓ, of length L such that $\ell \cap R = S$ and $\ell \cap F = \emptyset$.

Rogo was invented by the authors in 2009. In 2010 Rogo puzzles were published in a local newspaper, The (Christchurch) Press, and it was released as an iPhone app the same year. With the release of the app, Rogo was the subject of Michael Trick's Operations Research Blog [7], where he suggested that it was an interesting OR problem. Rogo was also the subject of another OR related blog [3], which presents a constraint programming formulation and some preliminary computational results.

As Rogo is new there is, so far, little about it in the literature. Petty et al. [6] described a pilot study to look at which aspects of Rogo puzzles might affect the degree of difficulty for human solvers. The results indicated that there was an element of universality in puzzle difficulty. As yet, no clear puzzle structures have been identified that can be used to gauge difficulty.

In this paper we investigate properties of Rogo and propose different formulations and optimisation algorithms.

2. Rogo Difficulty and its Relationship with Other Problems

Rogo is strongly related to the travelling salesperson problem (TSP) see, for instance, Ref. [4]. The main differences are that not all reward squares need to be visited, that loop paths cannot cross

[1] University of Canterbury, Christchurch, New Zealand
[a)] shane.dye@canterbury.ac.nz
[b)] nicola.petty@canterbury.ac.nz

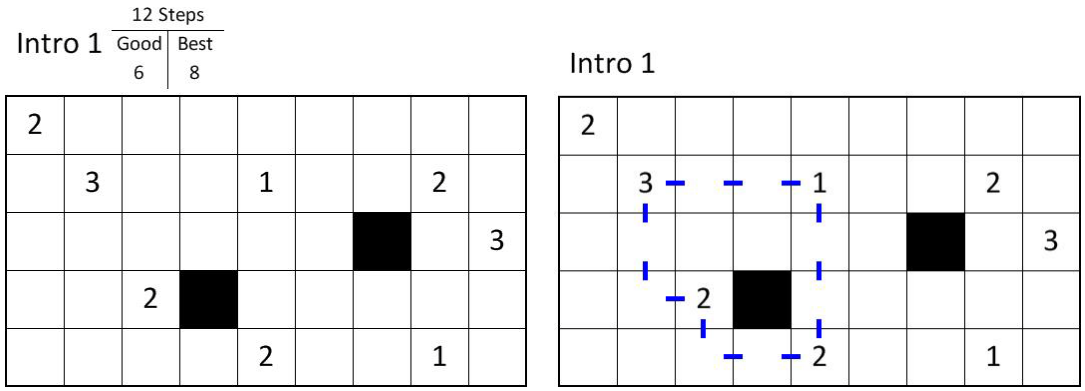

Fig. 1 A simple Rogo (left) and its 'best' (optimal) solution (right).

and that travel is on a grid. One question of interest is how these differences affect the difficulty of Rogo instances compared to TSP instances. Research into a variant of TSP has provided some insight. Laporte and Martello [5] looked at a subset-selection, prize-collecting TSP and found that solution times became significantly faster for enumerative-based algorithms as the maximum travel distance decreased with respect to the optimal TSP tour length.

Rogo instances can be characterised by the following parameters: grid size (m by n), the loop length L, and the number of reward squares, Q. In general Rogo is NP-hard, as shown in the appendix. Here, we focus on parameter restrictions which lead to classes of Rogo instances which will (or might) be solvable in time polynomial in the unrestricted parameters. The purpose of exploring such instance classes is to motivate possible solution approaches.

If the loop size is restricted to a given value L, there are a finite number of valid loops, $N(L)$. With L fixed, $N(L)$ is constant with respect to the other instance data and any Rogo instance can be solved in $O(nm)$ time by checking each loop using each grid-square, in turn, as the starting point. It is easy to show that $N(L)$ is bounded below and above by a function exponential in L. In particular, $N(L) \leq 3^L$ and a lower bound can be found by constructing a structured family of loops. The subclass of instances with L bounded to be $O(\ln(nm))$ can be solved in $O(n^2m^2)$ time using the upper bound on $N(L)$ and the previously suggested algorithm. Tight values of $N(L)$ for $L \leq 24$ are given in Section 4.

If the grid width and height are bounded by a fixed maximum value M, the loop length is bounded by M^2 and the above logic implies Rogo may be solved by a constant time algorithm.

When the maximum number of reward squares is bounded by a given constant Q, the number of ordered reward subsets is finite: $Q!$ or fewer. Assuming it is possible, in polynomial time, to test whether a given sequence of squares can be visited in order by a valid loop of length L, such instances could be solved in time polynomial in the characteristic parameters. We leave this question open.

Some of these cases provide ideas for algorithms for solving Rogo. We first look at IP formulations and then at enumeration based algorithms.

3. IP Formulations

Define two sets of binary variables: x_{ij} indicates whether the grid-square (i, j) is visited in the loop, and y_{dij} indicates whether edge (d, i, j) is used for the loop. Index d indicates whether an edge is horizontal, $d = h$, or vertical, $d = v$. Edge (h, i, j) corresponds to the edge between grid-squares (i, j) and $(i, j + 1)$ while edge (v, i, j) corresponds to the edge between grid-squares (i, j) and $(i + 1, j)$.

Two formulations are given with the same base formulation but different sets of valid-loop constraints. The base formulation is as follows.

$$\text{maximise} \quad \sum_{(i,j)\in R} r_{ij}x_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} = L \tag{2}$$

$$y_{vij} + y_{v,i-1,j} + y_{h,i,j} + y_{h,i,j-1} = 2x_{ij}$$
$$i = 1, \ldots, m,\ j = 1, \ldots, n. \tag{3}$$

$$+ \text{ Valid-loop constraints}$$

$$x_{ij} \in \{0, 1\}, y_{dij} \in \{0, 1\}$$
$$d \in \{v, h\}, i = 1, \ldots, m,\ j = 1, \ldots, n.$$

Objective function Eq. (1) and maximum loop length constraint Eq. (2) are obvious. Constraint Eq. (3) ensures exactly two edges are incident to a visited grid-square and no edges are incident to one not visited. For grid-squares on the sides of the grid, indicator variables for nonexistent edges do not appear in the constraint.

The first set of valid-loop constraints use edge-sets of loops with q squares for particular values of q. They require that not all edges of such a loop are used. This eliminates all subtours of length q. If subtours are present in a solution, at least one subtour must have length $L/2$ or less. Let $P(q, i, j)$ be the collection of edge sets of all loops using q squares starting at location (i, j). These valid-loop constraints are:

$$\sum_{(d,i,j)\in c} y_{dij} \leq q - 1$$
$$4 \leq q \leq L/2, 1 \leq i \leq m, 1 \leq j \leq n, c \in P(q, i, j) \tag{4}$$

These valid-loop constraints parallel the standard subtour elimination constraints for TSP [4]. This set of valid-loop constraints can be implemented using loop patterns separated from the start-

ing points. For 4, 6, 8 and 10 square loops there are 1, 2, 7 and 28 loop patterns, respectively (loops must use even numbers of squares). These numbers are small enough to make including all subtour elimination constraints potentially viable for loop lengths $L = 22$ or less. This set of valid-loop constraints is not valid when constraint Eq. (2) is relaxed to allow shorter loop lengths, since the optimal loop's length may be $L/2$ or less. **Figure 2** shows an example of such an instance.

The second set of valid-loop constraints uses collections, $C(i_1, j_1, i_2, j_2)$, which contain all cuts separating (i_1, j_1) and (i_2, j_2): these are minimal edge sets which separate the grid into exactly two connected components, one holding (i_1, j_1) and the other holding (i_2, j_2). These constraints are:

$$\sum_{(d,i,j)\in c} y_{dij} \geq 2x_{i_1 j_1} + 2x_{i_2 j_2} - 2$$
$$(i_1, j_1) \neq (i_2, j_2) \in R, c \in C(i_1, j_1, i_2, j_2) \quad (5)$$

They ensure that at least two arcs cross each cut separating (i_1, j_1) and (i_2, j_2) when both squares are visited. These constraints are a direct parallel of the standard connectivity constraints for the TSP [4]. This set of valid-loop constraints could be implemented by initially relaxing them all, adding violated constraints sequentially until a feasible solution is obtained. This set of valid-loop constraints remains valid when constraint Eq. (2) is relaxed to allow loop lengths of less than L.

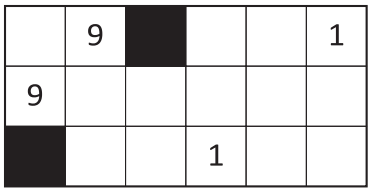

Fig. 2 Rogo with best solution for a loop of length four better than that with a loop of length eight.

4. A Loop Pattern Testing Algorithm

One potential solution method involves enumerating all possible loop shapes using L squares, then testing these loops for all possible starting points. By pre-processing generation of all valid loop patterns off-line, running time of the algorithm is $O(NmnL)$ where N is the number of loops to test. Algorithm speed can be improved by only checking non-forbidden squares as starting points and not allowing starting points which would cause part of the loop to fall outside of the grid boundary. This algorithm is labelled Pattern Testing.

An implementation of Pattern Testing is shown in **Fig. 3**. The set of loop patterns, P, includes all possible loop shapes for loops of length L. Each pattern is stored as a list of L offsets, corresponding to squares visited relative to a reference square. The offsets measure the number of columns to the right of the leftmost column (labelled 0) and the number of rows below the uppermost row (labelled 0). The width of a loop pattern, $w(\ell)$, is the maximum column offset and the height, $h(\ell)$, is the maximum row offset.

The algorithm can be adapted to the situation where loop length is bounded by L by including all loop patterns of length L or less in P.

Valid loops can be generated by enumeration, starting with the square in the top corner on the leftmost boundary of the loop and generating the loop to the right (this avoids much duplication). **Figure 4** shows this corner for a 16 square loop.

The remainder of the loop is sequentially extended by adding each of the three possibly valid edges in turn, repeating the process for each partially completed loop generated. Edges are skipped if the loop returns to a previously used square or it returns to the beginning too soon. The following properties can be used

```
Pattern Testing
 best := 0, bestloop := {}
 for each ℓ = {(i1,j1),...,(iL,jL)} ∈ P
    for i = 1 to m − h(ℓ)
       for j = 1 to n − w(ℓ)
          score := 0, valid := TRUE
          for k = 1 to L
             if (i + ik,j + jk) ∈ F then
                valid := FALSE
                exit for
             else if (i + ik,j + jk) ∈ R then
                score := score + r_{i+ik, j+jk}
             end if
          next k
          if valid = TRUE and score > best then
             best := score, bestloop := ℓ+(i,j)
          end if
       next j
    next i
 next ℓ
```

Fig. 3 Pattern Testing algorithm which tests loops from a given set of all possible loop patterns.

Table 1 Number of loop patterns for various loop lengths.

Loop length, L	12	14	16	18	20	22	24
Number of loops, N	124	588	2938	15 268	81 826	449 572	2 521 270

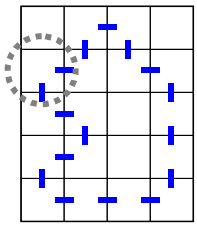

Fig. 4 The top corner on the leftmost edge of a 16 square loop.

to reduce running time. A valid loop must have as many leftwards edges as rightwards, and as many upwards edges as downwards. A square can only be visited if there are sufficient steps remaining to return to the starting square.

The performance of Pattern Testing will depend on N, the number of loops tested. Running a pre-processing algorithm based on the above gave values for N as shown in **Table 1**. Loop pattern storage requirements could be reduced by applying rotational and reflective transformations to loop patterns in the solution algorithm and removing duplicating patterns.

The running time should be relatively independent of the density of reward squares but decrease as the number of forbidden squares increases. The number of valid starting points is proportional to the number of non-forbidden squares. In addition, the average time to test loops would be reduced as the number of forbidden squares increases since loop testing can be abbreviated as soon as a forbidden square is encountered.

5. A Loop Enumeration Algorithm

An alternative to pre-processing the generation of loop patterns is to construct loops sequentially within the solution algorithm. During loop construction, the local pattern of reward and forbidden squares would be used to limit the shape of the loop. This reduces the total number of loops that need to be considered.

In the algorithm developed, loop construction is broken into two phases. The first phase generates potential reward sets and the order they are visited, ignoring forbidden squares. The second phase attempts to create a feasible loop visiting the set of rewards in the order given from the first phase. The loop must avoid forbidden squares, not revisit squares, and use the correct number of steps. Since any feasible loop which visits all of this set of rewards will suffice, the first such loop found is used.

The first phase is implemented by constructing visit-orders of reward squares. A visit-order is a finite sequence of reward squares in the order they will be visited by a loop.

Definition 2: A **visit-order**, $v = [(i_1, j_1), (i_2, j_2), \ldots, (i_K, j_K)]$, is an ordered array of unique reward squares, $(i_k, j_k) \in R$.

Visit-orders are enumerated by adding a new reward square to the end of an existing visit-order. The following notation is used: $v \cup (i, j) = [(i_1, j_1), (i_2, j_2), \ldots, (i_K, j_K), (i, j)]$ where $v = [(i_1, j_1), (i_2, j_2), \ldots, (i_K, j_K)]$.

Distances are measured as rectilinear distances, steps over the grid. These are used to eliminate visit-orders which would be impossible to visit using a loop of the correct length, even without the presence of forbidden squares.

Definition 3: Define the following distance measures:

(a) The **distance** *between* squares (i_1,j_1) and (i_2,j_2) is $d((i_1, j_1), (i_2, j_2)) = |i_2 - i_1| + |j_2 - j_1|$,

(b) The **path length** of visit-order, $v = [(i_1, j_1), (i_2, j_2), \ldots, (i_K, j_K)]$, is $d(v) = \sum_{k=1}^{K-1} d((i_k, j_k), (i_{k+1}, j_{k+1}))$,

(c) The **distance** *from* visit-order $v = [(i_1, j_1), (i_2, j_2), \ldots, (i_K, j_K)]$ *to* grid-square (i,j) is $d(v, (i, j)) = d((i_K, j_K), (i, j))$,

(d) The **distance** *to* visit-order $v = [(i_1, j_1), (i_2, j_2), \ldots, (i_K, j_K)]$ *from* grid-square (i,j) is $d((i, j), v) = d((i, j), (i_1, j_1))$,

We introduce the concept of a neighbour of a visit-order to be any reward square not in the visit-order which is close enough to it to allow the square to be visited within the loop length bound.

Definition 4: The neighbourhood of visit-order $v = [(i_1, j_1), (i_2, j_2), \ldots, (i_K, j_K)]$ selected from a subset of reward squares S is defined as: $T(v, S) = \{(i, j) \in S \setminus v : d(v) + d(v, (i, j)) + d((i, j), v) \leq L\}$.

The triangle inequality holds for the distance measures defined above, leading to the result that the neighbourhood contracts as reward squares are added to visit-orders.

Lemma 1: For visit-order v, subset of reward squares S and any reward square $(i, j) \in T(v, S)$ we have $T(v \cup (i, j), S) \subset T(v, S)$.

The algorithm in **Fig. 5** uses subroutine *GetLoop*(v) to attempt to construct a loop visiting all rewards squares of visit-order v in order. Further details of this subroutine are given below.

The algorithm implementation requires the reward squares to be provided as an ordered array of grid-square indices $R = [(i_1, j_1), (i_2, j_2), \ldots, (i_Q, j_Q)]$. These are sequentially used as the start of a loop. As the algorithm progresses, reward squares previously used as the start of a loop no longer need to be considered. This is because any loop containing that reward square must have been generated previously. At each iteration, a visit-order is selected and independently extended by each of its neighbours. Any newly created visit-order with total reward of more than the best loop found so far is tested to see whether it defines a valid loop – to become the current best loop. In the algorithm, the total reward of a visit-order is recorded as the lower bound (LB) for any pos-

```
Loop Construction
  best := 0, bestloop := []
  for k = 1 to Q
      v(1) := [(i_k, j_k)], LB(1) := r_{i_k j_k}, T(1) := ⊤(v(1), {(i_{k+1}, j_{k+1}), ..., (i_Q, j_Q)}), N := 1
      UB := r_{i_k j_k} + Σ_{(i,j)∈T(1)} r_ij
      if UB > best then
          while N > 0
              v_c := v(N), LB_c := LB(N), T_c := T(N), N := N − 1
              for each (i,j) ∈ T_c
                  v_t := v_c ∪ (i,j), LB_t := LB_c + r_ij;
                  if LB_t > best then
                      ℓ_t := GetLoop(v_t)
                      if ℓ_t ≠ [] then
                          best := LB_t, bestloop := ℓ_t
                      end if
                  end if
                  T_t := ⊤(v_t, T_c), UB := LB_t + Σ_{(i,j)∈T_t} r_ij
                  if T_t ≠ ∅ and UB > best then
                      N := N + 1, v(N) := v_t, LB(N) := LB_t, T(N) := T_t
                  end if
              next (i,j)
          end while
      end if
  next k
```

Fig. 5 Loop Construction algorithm which enumerates loops by generating potentially viable visit-orders.

sible valid loop containing it. All newly created visit-orders with non-empty neighbourhoods (and sufficient total reward over their neighbourhood) are stored for later extension. In the algorithm, the total reward over the visit-order and its neighbourhood provides an upper bound (UB) on any valid loop containing it. This algorithm, labelled Loop Construction, is shown in Fig. 5.

Subroutine *GetLoop*(v) attempts to construct a loop of length L on the grid visiting all reward squares of v in sequence and no others, returning to the first reward square, and obeying all loop requirements including avoiding forbidden squares. The subroutine returns the first loop found if one exists, and an empty array if not. The details of this subroutine are straightforward; loops are constructed by sequentially trying each possible next square in a usual enumeration framework.

The running time for Loop Construction would be expected to be exponential in the loop length since this would increase the number of reward squares in the neighbourhood of each possible starting square. A similar effect should be seen as the reward density increases. However, the average running time of GetLoop could be expected to decrease as more of the loop route is fixed.

Increasing the number of forbidden squares does not affect the number of visit-orders but should change the average running time of GetLoop. The size of the complete GetLoop enumeration tree will be reduced, but so will the chance of finding a valid loop quickly. This also reduces the number of visit-orders defining valid loops which could increase the amount of the Loop Construction enumeration tree explored.

6. Computational Results

Computational experiments using the IP formulation with the first set of valid-loop constraints took some minutes to solve instances of size 8×8 and smaller. Larger instances failed to solve in a reasonable time, if at all. These results are similar to those found for the selective TSP [5]. Solution times of over 10 seconds were found on 8×8 instances using the second set of valid-loop constraints with a naive implementation for adding violated constraints. Further computational experiments with IP-based solution methods were not pursued. The formulations and solution methods were not optimised for speed.

The two algorithms, Pattern Testing and Loop Construction, were compared on a set of randomly generated instances designed to compare performance issues. We briefly describe the design of the instances used before displaying and discussing results of computational experiments. The test instances are available from the authors. Loop lengths were required with equality in all tests. Further testing showed that the solution times differed little when the loop length constraint was relaxed to allow shorter loops.

Four aspects of Rogo instances were considered for computational testing. We were interested in understanding how the two algorithms performed as instance size, loop length, reward density and forbidden density increased. Three sets of instances were constructed; only square instances were used. All sets were con-

Table 2 Average solution times (seconds) for algorithms Pattern Testing (PT) and Loop Construction (LC) by grid width and loop length with reward density 0.22.

Grid width	Loop length 12		14		16		18		20		22	
	PT	LC	PT	LC	PT	LC	PT	LC	PT	LC	PT	LC
9	≤0.005	≤0.005	≤0.005	≤0.005	0.010	≤0.005	0.062	0.010	0.34	0.022	1.9	0.074
12	≤0.005	≤0.005	≤0.005	≤0.005	0.020	0.008	0.094	0.024	0.49	0.080	2.7	0.29
15	≤0.005	≤0.005	0.010	≤0.005	0.030	0.010	0.14	0.036	0.77	0.12	4.4	0.43
18	≤0.005	0.006	0.010	0.008	0.040	0.012	0.21	0.044	1.1	0.15	6.4	0.55
21	≤0.005	0.008	0.010	0.010	0.058	0.020	0.29	0.058	1.5	0.20	8.5	0.78
24	≤0.005	0.010	0.020	0.010	0.080	0.022	0.40	0.066	2.1	0.21	12	0.75
27	≤0.005	0.010	0.020	0.010	0.10	0.026	0.51	0.082	2.8	0.27	15	1.0
30	0.010	0.010	0.024	0.014	0.12	0.034	0.65	0.11	3.5	0.33	19	1.3
33	0.010	0.010	0.032	0.018	0.16	0.040	0.81	0.13	4.4	0.40	24	1.6

structed by combining randomly selected 3×3 grid blocks. All reward values were between one and nine, (uniformly) randomly selected.

The first set of instances was constructed to test the effect of instance size. They were constructed from large starting Rogos. The starting Rogos were made up of 121 of the 3×3 blocks to form 33×33 sized Rogos. The 3×3 blocks had at most one forbidden square, randomly selected, and one or two rewards squares, selected so as not to be directly adjacent horizontally or vertically. Smaller Rogos were constructed from the starting Rogos by alternately removing the rightmost and bottom layers of 3×3 blocks, then the leftmost and top. From each starting Rogo, nine Rogo instances were created with sizes from 9×9 to 33×33. The effect was to make the smaller instances subgrids of the larger ones. Five starting instances were created, to give 45 instances. Each instance was tested with six loop lengths from 12 to 22 steps.

Results for the two algorithms are shown in **Table 2**. The times are in seconds, given to two significant figures and averaged over five instances. The reward density of these instances is about 0.22. At this density all solution times are less than one minute with Loop Construction outperforming Pattern Testing by approximately an order of magnitude.

The average solution times for loop length 20 are illustrated in **Fig. 6** plotted by the instance grid area (total number of grid squares). From this we see that, as expected, the solution times of the two algorithms increase roughly in proportion to the grid area.

The second set of instances was constructed to test the effect of reward density. Instances for this set were all 21×21. Base instances were generated with one forbidden square and, at most, one reward square in each 3×3 block, both randomly placed. From each base instance, six more instances were generated in sequence with an additional reward randomly placed in a non-reward square for each 3×3 block. Five base instances were created, to give 35 instances in total. Each instance was tested with six loop lengths from 12 to 22 steps.

Results for the two algorithms are shown in **Table 3**. The times

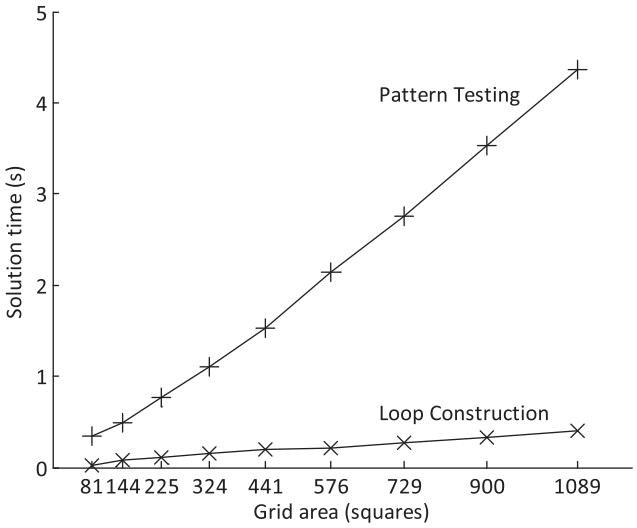

Fig. 6 Average solution time by grid area for rogos with loop length 20 steps and reward density 0.22.

are in seconds, given to two significant figures averaged over five instances. A maximum running time of 300 seconds was imposed. All instances use grids of size 21×21. Solution times for Pattern Testing are largely unaffected by reward density while times for Loop Construction appear to increase exponentially as density increases. This is illustrated in **Fig. 7**, which shows the average solution time by reward density for rogo instances of size 21×21 with loop length 18.

The computational results imply that the exponential increase visit-orders dominants the reduction in GetLoop running time as more parts of the loop are fixed. This might be explained by two aspects of the algorithm which reduce the overall affect of GetLoop. GetLoop returns as soon as a valid loop is found and is only called when the reward of the visit-order is better than the current incumbent loop. Further analysis is needed to better understand this relationship and could provide insights to improve algorithmic performance.

Figure 8 shows solution time as loop length increases, for rogo instances of size 21×21 with reward density approximately 0.56.

Table 3 Average solution times (seconds) for algorithms Pattern Testing (PT) and Loop Construction (LC) by reward density and loop length on 21×21 grids.

	Loop length											
Reward	12		14		16		18		20		22	
density	PT	LC	PT	LC	PT	LC	PT	LC	PT	LC	PT	LC
0.11	≤0.005	≤0.005	0.010	≤0.005	0.070	≤0.005	0.35	≤0.005	1.9	≤0.005	10	0.008
0.22	≤0.005	≤0.005	0.010	≤0.005	0.060	≤0.005	0.29	0.008	1.5	0.020	8.2	0.034
0.33	≤0.005	≤0.005	0.010	0.010	0.070	0.022	0.35	0.062	1.9	0.22	10	0.78
0.44	≤0.005	0.010	0.010	0.036	0.058	0.16	0.29	0.67	1.6	3.2	8.9	16
0.56	0.010	0.044	0.12	0.18	0.36	0.97	1.0	5.4	5.0	32	10	>300
0.67	0.010	0.084	0.020	0.45	0.070	2.7	0.53	18	3.0	140	13	>300
0.78	≤0.005	0.34	0.010	2.2	0.070	16	0.35	130	1.9	>300	10	>300

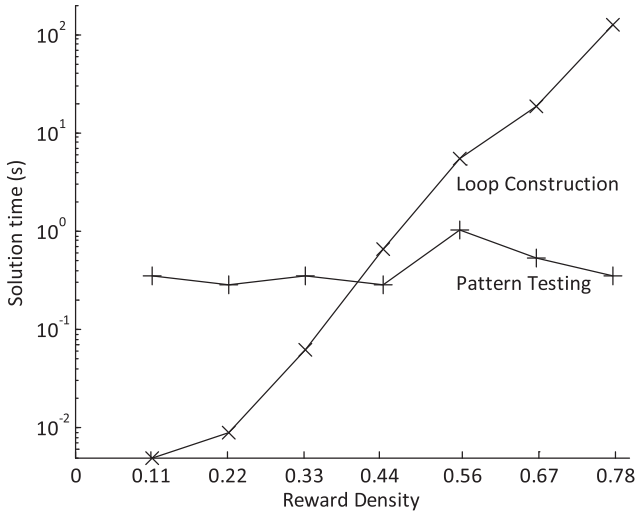

Fig. 7 Average solution time by reward density for 21×21 rogos with loop length 18 steps.

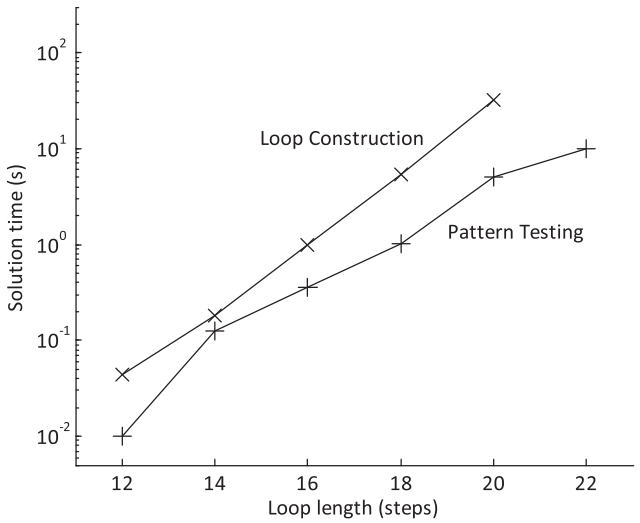

Fig. 8 Average solution time by loop length for 21×21 rogos with reward density 0.56. Missing values included instances which took longer than 300 seconds.

Solution times for both algorithms appear to increase exponentially with loop length.

The third set of instances was constructed to test the effect of forbidden density. Instances for this set were all 21×21. Base instances were generated with no forbidden squares and two reward squares in each 3×3 block, all randomly placed. From each base instance, four more instances were generated in sequence with an additional forbidden square added to each pair of 3×3 blocks. The final numbers of forbidden squares were 0, 24, 49, 73 and 98 respectively. Five base instances were created, to give 25 instances in total. Each instance was tested with a loop length 18 steps.

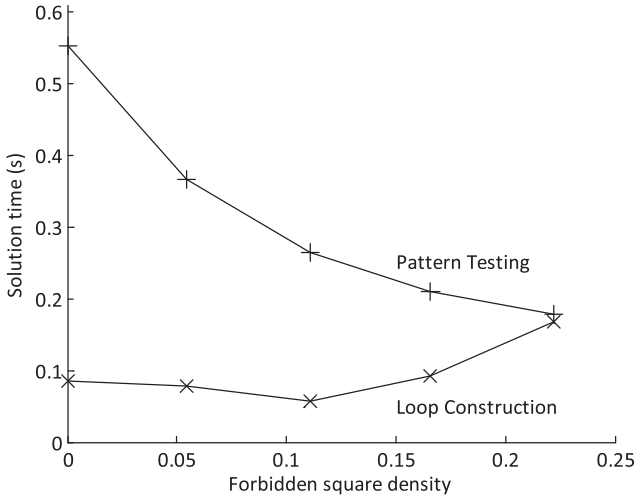

Fig. 9 Average solution time by forbidden square density for 21×21 rogos with loop length 18 and reward density 0.22.

Figure 9 shows solution time as forbidden density increases. The average solution time for Pattern Testing reduces as expected. For Loop Construction the average solution time drops at first then increases. The affect appears small. Further testing is needed to better understand the impact of forbidden squares on this algorithm.

From these results it would appear that typical Rogos designed for human amusement can be quickly solved by either of these two algorithms. Data requirements for Pattern Testing would suggest that Loop Construction is more suitable for testing puzzles created for amusement.

7. Discussion and Future work

This paper looked at the new puzzle Rogo as an optimisation problem to find the best accumulated reward score and a corresponding loop. The two algorithms presented appear effective for solving Rogos designed for human amusement. The computational experiments also go some way to providing understanding about properties which make instances computationally difficult for the two algorithms. It seems that loop length and reward density are the two main factors. Further investigation into this aspect of the algorithms' performance would make interesting future work.

While this paper makes a contribution to the computational difficulty of Rogo instances, another interesting question is what aspects of a Rogo puzzle makes it difficult (or easy) to solve with pen and paper, without computers. Here the set of Rogo instances drawn from will need to be better defined: those interesting for a human to solve with pen and paper. (Clearly a Rogo with a million grid squares and rewards in each non-forbidden square taking values less than one thousand would be difficult, but not interesting.) As well as those aspects which make Rogo computationally difficult, there are likely to be other aspects which make Rogos difficult for pen and paper solution. For instance, it is not clear that pen and paper difficulty is intransient under rotation and reflection of the Rogo grid. Petty et al. [6] takes an initial look at this.

Another direction for future work is the development of automated processes for generating Rogos for human amusement.

Acknowledgments The authors would like to thank the reviewers for their thoughtful suggestions which have greatly improved this paper. In particular, the NP-hardness proof was suggested by one of the reviewers.

References

[1] Garey, M.R. and Johnson, D.S.: *Computers and Intractability*, W.H. Freeman (1979).
[2] Kant, G.: Drawing planar graphs using the canonical ordering, *Algorithmica*, Vol.16, pp.4–32 (1996).
[3] Khellerstrand, H.: Rogo grid puzzle in Answer Set Programming (Clingo) and MiniZinc, My Constraint Programming Blog, available from ⟨http://www.hakank.org/constraint_programming_blog/2011/01/rogo_grid_puzzle_in_answer_set_programming_clingo_and_minizinc_1.html⟩ (accessed 2011-02-03).
[4] Laporte, G.: A concise guide to the Traveling Salesman Problem, *Journal of the Operational Research Society*, Vol.61, pp.35–40 (2010).
[5] Laporte, G. and Matrello, S.: The selective travelling salesman problem, *Discrete Applied Mathematics*, Vol.26, pp.193–207 (1990).
[6] Petty, N.W. and Dye, S.: Determining degree of difficulty in Rogo, a TSP-based paper puzzle, *Proc. 45th Annual Conference of the ORSNZ*, pp.345–350 (2010).
[7] Trick, M.: Operations Research, Sudoko, Rogo, and Puzzles, Michael Trick's Operations Research Blog, available from ⟨http://mat.tepper.cmu.edu/blog/?p=1302⟩ (accessed 2011-02-03).

Appendix

This NP-hardness proof is thanks to one of the anonymous reviewers. We gratefully acknowledge the contribution. The proof uses a reduction from the NP-hard Hamiltonian circuit problem for planar cubic graphs, see problem [GT37] in Ref. [1].

Given any planar cubic graph $G = (V,E)$ an instance of Rogo is constructed by representing each vertex v as a reward square with reward value 1. Using one of the main results from Ref. [2], these can be embedded in a grid with edges represented by non-overlapping paths of empty grid squares. The embedding runs in polynomial time and the grid width and height are linear in $|V|$. The remaining squares are forbidden. With L sufficiently large, the inequality Rogo instance has maximum collected reward of $|V|$ if, and only if, there exists a Hamiltonian circuit. For the equality version, all loop lengths up to the total grid area can be tested sequentially, solving polynomially many Rogo instances.

Shane Dye obtained his Ph.D. at Massey University. He researches decision-making under uncertainty within the telecommunications, water-resource and electricity industries. He received the Harold W. Kuhn Award and is co-inventor of Rogo, an operations research inspired puzzle. Currently, he lectures operations research at University of Canterbury.

Nicola Ward Petty is a Senior Lecturer in the Department of Management, University of Canterbury and has a Ph.D. in Operations Research. She is a director of Creative Heuristics Ltd, and co-inventor of Rogo. Nicola's research is mainly teaching related as is her blog:
www.learnandteachstatistics.wordpress.com