Noname manuscript No.
(will be inserted by the editor)

A Genetic Algorithm for the Zen Puzzle Garden Game

Martyn Amos · Jack Coldridge

Abstract In this paper we present a novel genetic algorithm (GA) solution to a simple yet challenging commercial puzzle game known as Zen Puzzle Garden (ZPG). We describe the game in detail, before presenting a suitable encoding scheme and fitness function for candidate solutions. We then compare the performance of the genetic algorithm with that of the A* algorithm. Our results show that the GA is competitive with informed search in terms of solution quality, and significantly out-performs it in terms of computational resource requirements. We conclude with a brief discussion of the implications of our findings for game solving and other "real world" problems.

Keywords Genetic Algorithm · Transport Puzzle · Game · A*

1 Introduction

Zen Puzzle Garden (ZPG) [17] is a one-player puzzle game involving a Buddhist monk raking a sand garden. It is inspired by Japanese garden design, and one common feature of such gardens is a flat region of sand or small pebbles, which is raked into a pattern. ZPG is one example of a *transport puzzle*; these are problems that involve the player moving entities around a given domain (e.g., boxes around a warehouse), starting at some initial configuration, until they attain pre-defined goal conditions. Entities must move according to the constraints of the puzzle, and may only move between connected positions (that is, an entity may not be "lifted" off the board and replaced at a position perhaps far from its initial location). A graphical representation of the problem may use vertices to represent the set of positions an entity may occupy, with connecting edges determined either from any explicitly named connections, or from those implied by arrangement on the board or within a grid.

The rest of the paper is organized as follows: We first describe related work in Section 2 and give an in-depth description of the problem in Section 3, before describing two solution methods (genetic algorithm and A*) for ZPG in Section 4.

School of Computing, Mathematics and Digital Technology,
Manchester Metropolitan University, Manchester M1 5GD, UK.
E-mail: M.Amos@mmu.ac.uk

Experimental results are presented in Section 5, before we conclude with a brief discussion of the implications of our findings in terms of broader applicability.

2 Previous work

Many transport puzzles require the player to make a trip around a board between given start and end positions, and puzzles may be extended via the introduction of objects or obstacles to the board, which must be collected or moved to satisfy given constraints. A well-studied example of the transport puzzle is *Sokoban* [5, 11]. In this game the player takes control of a warehouse keeper whose job it is to push boxes around a maze and into designated target locations; only one box may be pushed at any one time, and boxes may not be pulled. The challenge of this puzzle is derived from this latter condition, since if a box is pushed into a corner of any construction then the game is lost. Sokoban is known to be NP-hard [5]. Various AI-based techniques have been applied to its solution, including multi-agent systems [1], abstraction and decomposition [2], embedding domain-specific knowledge [12], and heuristic search [13]. The applicability of such methods to ZPG is, however, not clear, since the problem features additional complicating factors (described in the next Section).

According to an in-depth search of the literature, no attempts to automatically solve this particular problem have been previously documented. We believe the genetic algorithm (GA) [6] to be a good candidate method for its solution. We therefore describe preliminary work on applying this method to a new variant of the transport problem. We present a comparison between the GA and a "base-line", search-tree-based method, in the hope that this will motivate further study in the future.

3 Zen Puzzle Garden

The ZPG game takes place on a garden board comprised of a two-dimensional grid of *sand* squares surrounded by a *perimeter region*. Boards may also contain *rocks*, *leaves* and *ornaments*, the purpose of which we explain shortly. Two of the boards supplied with the game [17] are partitioned into different sections separated by path regions; these are not considered here. The objective of the game is to move a *monk* character around the garden, causing him to completely rake the available surface. A puzzle board is completed when all initially empty squares (i.e., any sand square not covered by either a rock or an ornament) have been raked and the monk has stepped onto the perimeter region. This last condition is necessary, as a square is not considered to be raked until it has been vacated.

The monk always begins a game at the same point on the perimeter, regardless of the board being played, and the following rules apply:

- The monk may move freely around the perimeter region.
- When on the sand the monk may only move within the von Neumann neighbourhood (i.e., no diagonal movements are allowed).
- The perimeter is left by entering the sand on any un-raked (i.e., empty) square.

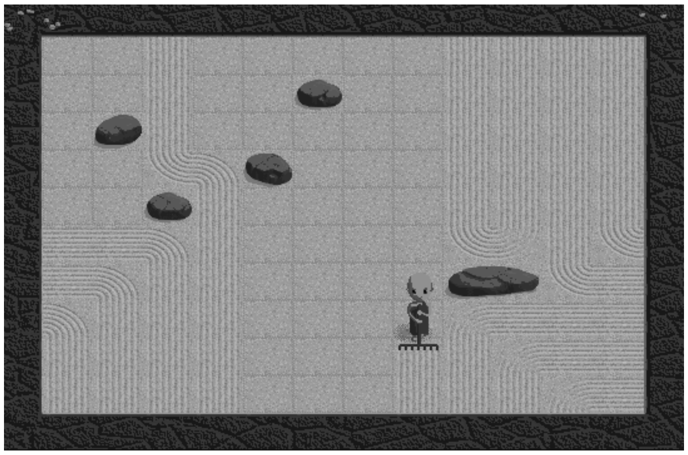

Fig. 1 Example Zen Puzzle Garden board.

- Vacating an empty sand square causes that square to be permanently raked. The monk cannot move from the perimeter onto an empty sand square and then immediately back onto the *same* perimeter square, as this is considered a "step back".
- Once moving on sand, the monk continues to move in a straight line until he encounters either the perimeter (in which case he moves onto it), a raked square or an *object* (in both cases, he stops moving). The monk may not turn corners during a single move while moving on sand. These two rules are central to the challenge of the game – if the monk could be moved over the sand on a square-by-square basis then most boards would be trivial to solve.
- Any *moveable* objects may only be pushed onto an *empty* sand square. They may not be pushed off the garden into the perimeter region. A single move pushes such an object one square, if possible. When pushing an object the monk does not continue moving until he is no longer able to (unlike in normal movement), but the monk may use multiple moves to push an object a number of squares.
- The current game ends if the monk is moved into a position such that he is in a "dead end" (i.e., unable to make a legal move).

Objects may be one of three types:

- *Rocks* are placed in a fixed position at the start of the game, and may not be moved.
- *Ornaments* also start in the same place at the beginning of each game. They may only be pushed into an empty square, and not into the perimeter (see above).
- *Leaves* are coloured either yellow, orange or red, and must be collected (i.e., moved over) in that order, at which time they are removed from the board. An orange or red leaf is classed as an immoveable object until the preceding leaves have been collected.

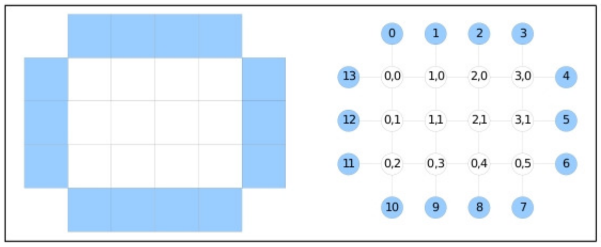

Fig. 2 Graph-based representation of ZPG board.

An example board is depicted in Figure 1. In this situation the monk is halfway through completing a move, and will completely rake the current column before moving onto the perimeter.

Previous work [4] suggests that ZPG is likely to be NP-complete. Although a formal proof has yet to be published, we are aware of evidence that this is, indeed, the case [10]. The authors of [4] study "pushing block" puzzles (like Sokoban), and introduce variants in which blocks must slide their maximal extent when pushed, and where a player's path must not cross itself. These variants are demonstrated to be NP-hard.

4 Two methods for ZPG

In this section we describe two solution methods for ZPG; a search-tree method and a genetic algorithm (GA). As a single board may admit several different solutions, in what follows, we seek the *optimal* (i.e., shortest) sequence of moves needed to solve each puzzle instance. We first give a representation scheme for the game, before describing a generic simulator we built for the game. We then give details of the A* and GA algorithms.

4.1 Representation

In Figure 2, we show a graph-based representation of the ZPG board. Perimeter tiles are represented by coloured nodes, and sand tiles labelled according to their coordinated on the board. Entities representing the player and obstacles may occupy vertices on the graph or travel (where allowed) along connecting edges between them. Although not shown in Figure 2, the monk may move freely between perimeter nodes (that is, the perimeter subgraph is fully connected).

Standard game theory [20] defines a *game tree* as a graph in which nodes represent game states, with each branch corresponding to a move. The *complete* game tree for a problem is the game tree starting at the initial state and containing every possible move. Terminal nodes represent the possible states that may end the game; either a goal state or deadlock. The *branching factor* is the number of children at each node; an exhaustive search of the tree will follow every branch at every node and the total number of vertices will increase exponentially to the depth in the tree.

A game tree is an example of state-space search whereby successive configurations or states of an instance are considered, with the goal of finding a state with a desired property. Obtaining a complete game tree for a problem can be very computationally expensive [9], and has resulted in search algorithms such as A* playing an important role in recent research.

4.2 Simulator

Attempts to solve commercial games are often hampered by the reluctance of the authors to release commercially-sensitive source code. Therefore, in order to test candidate scripts against game boards, we use the same fundamental approach as Kendall and Spoerer [14], which is to write a simplified version of the game engine. This engine retains the essential characteristics of the game in terms of its rules, but omits the graphical user interface and other "playability" features. A representation of the game state is modified by an external controller program, and moves and their results may therefore be assessed without access to the source of the main game itself. Care is taken to ensure that all restrictions listed in Section 1 are enforced. The same simulator is used by both the genetic algorithm and informed search.

To determine the quality of a (partial) solution to a given garden, the simulator includes an $AreaFitness()$ method to calculate and return a value describing how close it is to a *full* solution. This method compares the unraked (available) area of the garden before and after a given path has been explored. By dividing the number of unraked tiles after the path has traversed the garden by the *initial* number of unraked tiles, a quality metric may be obtained. As the value approaches zero the proportion of the garden that has been raked rises, with a value of zero indicating the path has covered all available tiles.

4.3 A* solution

The A* algorithm is an extensively studied best-first search method. It is best-first in that it takes an *informed* approach when deciding which node is most likely to provide the least-cost distance to a goal state; the order in which nodes are visited is determined by a distance-plus-cost heuristic, given by $f(x) = g(x) + h(x)$, where:

– $g(x)$: The shortest distance from the root node to the node being evaluated;
– $h(x)$: The estimated distance from the node being evaluated to a goal state.

The heuristic function should be optimistic in that it will never overestimate the cost of a path from the root node to a goal state; as the algorithm will never overlook the possibility of a lower-cost path it is therefore admissible. Hart *et al.* [8] first discussed this algorithm in 1968, then calling it just Algorithm A. It has since been shown by Dechter and Pearl [3] to be optimal in that it considers fewer nodes than any other admissible search algorithm with the same heuristic.

The A* algorithm displays characteristics of the breadth-first search; it is complete in that it will always return a solution if one exists, and it will also visit all equal-cost vertices at a given depth before continuing further along a path at greater cost. Many enhancements have been made to the basic algorithm that allow it to be tailored to the problem domain, such as iterative deepening and pattern databases. These enhancements have provided some impressive reductions in the search effort required to solve challenging problems such as Rubik's Cube [16]. The algorithm is implemented using a priority queue, the ordering determined by the heuristic function, with a lower value indicating a better candidate. The following pseudo-code illustrates the basic operation of the algorithm:

```
function A* (start, goal) {
       add start to open_queue
       while open_queue not empty {
          x = poll open_queue for lowest f(x)

          if x == goal {
             return x
          }
          else {
          for y in neighbours(x) {
                if y not in closed_list and y not in open_queue {
                   add y to open_queue
                }
             }
         }
         add x to closed_list
       }
       return not_found_ERROR
}
```

The heuristic function comprises the distance from the root node, $g(x)$, given by the number of moves completed, and the estimate $h(x)$ is provided by the $AreaFitness()$ method described earlier.

The algorithm is implemented with a conditional loop which polls the fittest state from the open queue and proceeds to create leaf nodes for each move available from that state. As the nodes are created their quality is analysed, and they are inserted into the queue accordingly. Once the first complete solution has been found the queue is pruned of all nodes requiring a greater number of moves, and only nodes with an move count less than or equal to the best solution are inserted into the queue.

This process is repeated until all nodes in the queue have been examined, at which point the solutions found are analysed and the set of unique solutions returned. Pruning reduces the completeness of the solution set generated by the algorithm for solutions requiring more than the optimal number of moves, but this behaviour is acceptable, as only the optimal solutions are of interest in this investigation.

4.4 Genetic algorithm solution

Genetic algorithms [6] have been studied extensively in the context of combinatorial optimisation problems. With respect to the current problem, Hong *et al.* [9] discuss the application of genetic algorithms to game search trees, specifically that of the Latin square problem devised by Leonhard Euler. Mantere and Koljonen [18] perform a similar analysis of the efficacy of genetic algorithms for devising and solving Sudoku problems.

When choosing the representation for our GA, we first consider the basic operation of the game. A successful move consists of the selection of a point on the garden perimeter and a transition onto the garden surface, followed by zero or more in-move choices (the number of choices to be made is only non-zero if a raked square or object is encountered), leading to the monk finishing back on the perimeter. If a choice of direction is required at a raked square, then *at most* two options are available, as the monk can neither move backwards nor onto the raked square in front of him. If an ornament is encountered, then the monk must choose a number of times (≥ 0) to push it in the current direction.

For a rectangular board of dimension $x \times y$, there are $C = 2x + 2y$ unique starting points on the perimeter, ordered by starting at 1 at the north face of the upper-left square and moving "clockwise" to the west face of the same square. C is therefore a measure of the "circumference" of the board. We assume, based on extensive personal experience of playing the game, that no move will contain more than 20 direction choices or opportunities to push, and we therefore define m as the maximum number of moves allowed. A candidate solution is naturally made of a sequence of clauses

$$(c_1, p_{1,1}, d_{1,1}, d_{1,1}, \ldots, p_{1,8}, d_{1,8}), \ldots$$
$$(c_i, p_{i,1}, d_{i,1}, \ldots, p_{i,8}, d_{i,8}), \ldots$$
$$(c_m, p_{m,1}, d_{m,1}, \ldots p_{m,8}, d_{m,8}),$$

where each clause encodes a move, $1 \leq c \leq C$, $1 \leq p_{i,j} < max(x, y) - 1$ and $d_{i,j} \in \{1, 2\}$. Values for c_i encode a starting point on the board perimeter. Values for $p_{i,j}$ encode a distance to push an object (if encountered). We use an indirect encoding scheme for values of $d_{i,j}$ to encode direction choices, using the notion of *available moves*. When a move is required, the list of available moves is constructed and one of two chosen (remembering that only two moves will be possible), according to the value of d_i. This genome sequence therefore encodes a "script" of moves, which is then fed into the simulator for evaluation.

4.4.1 Fitness Function

The fitness function is given below. It accepts a sequence of moves, and returns the overall fitness of the sequence based on (a) its length, and (b) the quality of the solution it encodes. Fitness values are in the range $0 \ldots 500$. $GENE\_LENGTH$ is equivalent to m, above, and is set to a default value of 20. $Moves$ represents the number of valid moves executed. The first component of the fitness function therefore rewards short sequences.

```
fitness()
{
  fitness=0 ;
  fitness += (GENE_LENGTH-Moves)*
                (200/GENE_LENGTH)

  if (AreaFitness>0)
    fitness += (1-AreaFitness)*200
  else if (garden is full solution)
    fitness += 300
  else
    fitness = 0
}
```

The second component of the fitness value is calculated by the *AreaFitness*() function; a complete solution gains an absolute value of 300 for this component; if deadlock is reached (i.e., the monk is unable to move) a value of zero is awarded for this component.

5 Results

We tested both methods on 24 different ZPG game boards, including one engineered to have no solution. We selected nine "retail" boards (supplied with the game, labelled R1-R9) and constructed an additional 14 "test" boards (T1-T14). We could not simply select all retail boards, because we were restricted to using smaller boards due to the resource limitations we imposed (an informed search would be terminated if it either consumes more that 100Gb of disc space or runs for longer than 72 hours). Boards were selected/constructed in order to be "tractable" in this sense, but we also ensured that the full range of board features was represented in our test suite[1]

We initially ran both methods on 30 different ZPG game boards. Of these, 15 were "retail" boards (i.e., supplied with the game), and 15 were hand-designed by us. Of the latter, one "illegal" board was specifically engineered to have no solution, in order to demonstrate the exhaustive nature of the A* algorithm. The other 14 non-retail boards were designed to represent the range of obstacles contained in all 64 retail boards. Of the retail boards, eight were selected specifically because we expected them to exceed (due to their size) the A* resource limits that we impose. The 14 hand-designed boards were designed to be "tractable" in this respect. The boards used ranged in size from 6×4 to 9×6.

The genetic algorithm (GA) was implemented in Java, using the JGAP [19] Java package; it used a constant population size, rank-based selection (with 95% of the population considered for the next generation), and simple one-point crossover. The parameter values used were as follows:

[1] The board files, along with the code for the simulator, are available on request from the corresponding author.

Instance	**Opt.**	**GA best**	**GA av.**	**GA excess %**
T1	3	3	3.38	12.67
R1	4	4	4	0
R2	4	4	4	0
T2	4	4	4.12	3.0
T3	4	4	4	0
T4	4	4	4.62	15.5
T5	4	4	4.24	6.0
R3	5	5	5.14	2.8
R4	5	5	5	0
R5	5	5	5	0
R6	5	5	5	0
T6	5	5	5.06	1.2
T7	5	5	5.94	18.8
T8	5	5	5.88	17.6
T9	5	5	5.76	15.2
T10	5	5	6.36	27.2
T11	5	5	5.08	1.6
R7	6	6	11.41	90.17
R8	6	6	6.7	11.67
T12	6	7	8.98	49.67
T13	6	6	6.06	1.0
R9	6	6	6.22	3.67
T14	6	9	9.57	59.5
			Average	**14.66**

Table 1 Move count comparison for A* and GA.

– Population size: 1000
– Generations: 100
– Chromosome length: 20
– Mutation rate: 0.07

The GA was run 50 times on each board. Neither A* nor the genetic algorithm found valid solutions for the “illegal” board, for which no solutions exist. The following results therefore describe comparisons over 23 ZPG boards. In Table 1, we first show the results in terms of solution quality. For each board, we give the the optimum number of moves for completion (as found by A*), the length of the *best* solution found by the GA, the *average* solution length for the GA, and the average GA “quality overhead” in terms of excess moves. The table is ordered by the number of moves required by the optimal solution. We observe that the GA fails to find the optimal solution in only two cases (T12 and T14), and solves to optimality all of the retail boards tested (see graphical depiction of quality results in Figure 3). Over all boards, on average the GA finds solutions that require roughly 15% more moves than the optimal solution.

We now consider the computational effort required by each algorithm. We measure this in terms of *board evaluations required*, as we believe this to be the best practical metric. Both algorithms use the same board evaluation code, so this metric avoids complications arising from differential efficiency of *implementation* rather than the *inherent quality* of the respective algorithms. In addition, in order to ensure a fair comparison we only consider boards where the GA finds the *optimal solution*, so we discount boards T12 and T14.

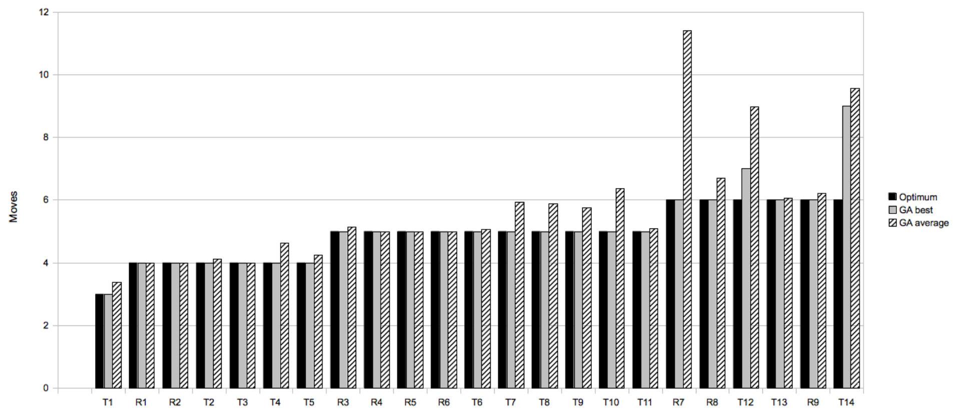

Fig. 3 Graphical depiction of quality results.

Instance	**A* evals.**	**Av. GA evals.**	**% of A***
T1	903,634	27000	2.99
R1	123,084	1000	0.81
R2	152,265	1000	0.66
T2	77,272	30000	38.82
T3	109,284	1000	0.92
T4	2,962,349	48000	1.62
T5	956,861	35000	3.66
R3	30,759,145	25000	0.08
R4	2,983,478	1000	0.03
R5	2,983,134	2000	0.07
R6	912,676	1000	0.11
T6	1,137,751	13000	1.14
T7	158,756,106	50000	0.03
T8	15,092,790	45000	0.3
T9	33,414,980	53000	0.16
T10	128,639,420	47000	0.04
T11	9,896,968	32000	0.32
R7	5,888,672	31000	0.53
R8	15,728,328	48000	0.31
T13	22,742,096	12000	0.05
R9	5,888,672	31000	0.53
		Average	**2.53**

Table 2 Number of evaluations for A* and GA.

The results are given in Table 2. The second column gives the number of boards evaluated by A* to find the optimal solution. The third column gives the average number of fitness evaluations required by the GA to find the optimal solution, and the fourth shows this number as a proportion of the number of A* evalutions (i.e., anything less than 100 shows an improvement over A*).

We observe that, with the exception of a single problem instance (T2), compared to informed search the GA requires significantly fewer evaluations to find

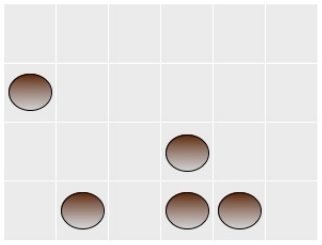

Fig. 4 The "problematic" board T2.

the optimum. Over all boards, the GA requires, on average, under 3% of the evaluations needed by A*. If we discount board T2, this falls to 0.65%.

Board T2 contains only rocks, and it is not immediately apparent from inspection why the GA struggles so much with this particular instance. The board is shown in Figure 4; we note its small size, and relatively dense population of rocks (21% of the board is occupied at the outset). We believe that this density reduces the number of possible paths through the board, which is why this board has by far the smallest search space. It may be that, in cases such as these, it is more efficient to solve the instance using informed search.

6 Conclusions

In this paper we have described a novel genetic algorithm solution to a block-based puzzle game. The game poses significant challenges in terms of the size of its search space, but our solution is competitive with informed search in terms of solution quality, and significantly out-performs it in terms of its computational resource requirements.

We presented ZPG in order to both demonstrate the efficacy of a genetic algorithm solution, and to encourage further study of its properties. Inspired by [15], we wish to investigate questions such as "Is it possible to automatically generate hard and easy instances of the problem?", as well as considering the notion of an *aesthetically pleasing* solution. Both of these questions could be addressed by considering the effect (in terms of both instance difficulty and the "artistic merit" of a solution) of the introduction (or omission) of various features, such as rocks, leaves and ornaments.

In addition to providing a useful testbed for new solution methods, the problem domain has real significance if we consider the problem of mobile robotics, where a self-avoiding path must be chosen whilst also considering possible obstacles and moveable objects. Future work will focus on formally establishing the difficulty of the ZPG game, as well as further investigations into its solution.

Acknowledgements The authors thank Joseph White (author of ZPG) for invaluable assistance with the game, David Corne for useful advice on representation schemes, and David Goldberg [7] and Robert M. Pirsig [21] for titular inspiration.

References

1. M.S. Berger and J.H.Lawton. Multi-agent Planning in Sokoban. *Multi-Agent Systems and Applications V*, Lecture Notes in Computer Science 4696, p.334, Springer, 2007.
2. A. Botea and M. Muller and J. Schaeffer. Using abstraction for planning in Sokoban. *Computers and Games*, Lecture Notes in Computer Science 2883, pp. 360–375, Springer, 2003.
3. R. Dechter and J. Pearl. Generalized best-first strategies and the optimality of A*. *Journal of the ACM*, 32(3), pp. 505–536, 1985.
4. E.D. Demaine and M. Hoffman. Pushing blocks is NP-complete for noncrossing solution paths. *Proc. 13th Canad. Conf. Comput. Geom*, pp. 65–68, 2001.
5. D. Dor and U. Zwick. SOKOBAN and other motion planning problems. *Computational Geometry: Theory and Applications* 13(4):215–228, 1999.
6. D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.
7. D.E. Goldberg. Zen and the art of genetic algorithms. *Proceedings of the 3rd International Conference on Genetic Algorithms*, pp. 80–85, 1989.
8. P.E. Hart and N.J. Nilsson and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), pp. 100–107, 1968.
9. T.-P. Hong, J.-Y. Huang, and W.-Y. Lin. Applying genetic algorithms to game search trees. *Soft Computing* 6(3–4):277–283, 2002.
10. R. Houston and J. White. Personal communication, August 2010.
11. A. Junghanns and J. Schaeffer. Sokoban: A challenging single-agent search problem. *IJCAI Workshop on Using Games as an Experimental Testbed for AI Research*, pp. 27–36, 1997.
12. A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence* 129(1-2), pp. 219–251, 2001.
13. A. Junghanns and J. Schaeffer. Sokoban: Improving the search with relevance cuts. *Theoretical Computer Science* 252(1-2), pp. 151–175, 2001.
14. G. Kendall and K. Spoerer. Scripting the game of Lemmings with a genetic algorithm. In *Proceedings of the 2004 Congress on Evolutionary Computation*, pp. 117–124. IEEE Press, 2004.
15. G. Kendall, A. Parkes and K. Spoerer. A survey of NP-complete puzzles. *ICGA Journal* 31(1), pp. 13–34, 2008.
16. R.E. Korf. Finding optimal solutions to Rubik's cube using pattern databases. *Proc. National Conference on Artificial Intelligence*, pp. 700–705, Wiley, 1997.
17. Lexaloffle Games. Zen Puzzle Garden, trial version downloadable at http://www.lexaloffle.com/zen.htm.
18. T. Mantere and J. Koljonen. Solving, rating and generating Sodoku puzzles with GA. *Proc. IEEE Congress on Evolutionary Computation (CEC) 2007*, pp. 1382–1389, IEEE Press, 2007.
19. K. Meffert and N. Rotstan and C. Knowles and U. Sangiorgi. JGAP: Java Genetic Algorithms and Genetic Programming Package, http://jgap.sf.net
20. M.J. Osborne and A. Rubinstein. *A course in game theory.* MIT Press, 1999.
21. Robert M. Pirsig. *Zen and the art of motorcycle maintenance.* Corgi, 1976.