Rook Jumping Maze Generation for AI Education

Todd W. Neller
Gettysburg College
Department of Computer Science
Gettysburg, PA 17325
tneller@gettysburg.edu

Abstract

Rook Jumping Maze design provides a number of good opportunities for experiential learning of AI concepts, including uninformed search, stochastic local search, machine learning, and objective/utility function design. In this paper we will define the maze and present a collection of exercises that allow exploration of several AI topics in the context of an engaging, fun, and unifying task.

Rook Jumping Mazes

Definitions

Figure 1 provides an example of a Rook Jumping Maze.[1] Let $r_{\max}$ and $c_{\max}$ be the number of rows and columns, respectively. In this case, $r_{\max} = c_{\max} = 5$. A state s of the maze (i.e. current location) is denoted by the row-column coordinate (r, c), where $r \in \{1, \ldots, r_{\max}\}$ and $c \in \{1, \ldots, c_{\max}\}$. For example, a maze puzzler located at $(1, 1)$ is located in the upper-left corner cell of the grid. The set of all states is denoted S. Let functions $\text{row} : S \rightarrow \mathbb{N}$ and $\text{col} : S \rightarrow \mathbb{N}$ map a state to its row and column, respectively.

The circled starting state of this example maze, denoted s_{start}, is $(1, 1)$. The goal state of this example maze, denoted s_{goal} and marked with a "G", is $(2, 4)$.

Each state of the maze has an associated *jump number* that provides the exact number of cells one may move horizontally or vertically in a straight line to change states. In Figure 1, the first move from $(1, 1)$ may either be 3 cells right to $(1, 4)$, or 3 cells down to $(4, 1)$. From $(4, 1)$, there is only one legal *forced move* 4 cells right to $(4, 5)$. From $(4, 5)$, one may move 3 cells left to $(4, 2)$ or 3 cells up to $(1, 5)$. A jump must be in a single orthogonal direction, and may neither stop short of the number of required cells at edges, nor may it wrap around edges toroidally (see Project Variations).

Let *jump function* $j : S \rightarrow \mathbb{N}$ map a state to its jump number. Define $j(s_{\text{goal}}) = 0$. Let the *successor function* $\sigma : S \rightarrow 2^S$ map a state to its possible successor states, that is:

$$\sigma(s) = \left\{ s' \in S \;\middle|\; \begin{array}{lcl} s' & = & (\text{row}(s) + j(s), \text{col}(s)), \text{or} \\ s' & = & (\text{row}(s) - j(s), \text{col}(s)), \text{or} \\ s' & = & (\text{row}(s), \text{col}(s) + j(s)), \text{or} \\ s' & = & (\text{row}(s), \text{col}(s) - j(s)) \end{array} \right\}$$

Let the *predecessor function* $\pi : S \rightarrow 2^S$ map a state to its possible predecessor states, that is:

$$\pi(s) = \{s' \in S | s \in \sigma(s')\}$$

Define a *path* of length n from s_{from} to s_{to} as a sequence of states $(s_1, s_2, \ldots, s_n)$ such that $s_1 = s_{\text{from}}$, $s_n = s_{\text{to}}$, and for all $1 \leq i < n$, $s_{i+1} \in \sigma(s_i)$. The *optimal* or *shortest solution path* is a path of minimal length from s_{start} to s_{goal}. Let $|p|$ denote the length of path p. Let $P_{s_{\text{from}}, s_{\text{to}}}$ be the set of all paths from s_{from} to s_{to}. Then an optimal solution path p^* is $\arg\min_{p \in P_{s_{\text{start}}, s_{\text{goal}}}} |p|$.

3	4	1	3	1
3	3	3	G	2
3	1	2	2	3
4	2	3	3	3
4	1	4	3	2

Figure 1: Example Rook Jumping Maze. Starting at the circled cell, each *jump number* indicates the exact number of cells one may move in a straight line horizontally or vertically. The object is to find a path to the goal marked "G".

Origin

The origin of Rook Jumping Mazes is unknown, but some attribute its creation to the great puzzle innovator Sam Loyd. Loyd's 1898 Queen Jumping Maze, which additionally allows diagonal moves, is shown in Figure 2. It appears on page 106 of the *Cyclopedia of Puzzles* (Loyd 1914), a collection of Loyd's work compiled by his son.[2]

[1] Minimum 13-move solution for Figure 1: down, right, left, up, down, left, right, up, left, left, right, down, up

[2] Public domain scans available from `http://www.mathpuzzle.com/loyd/`

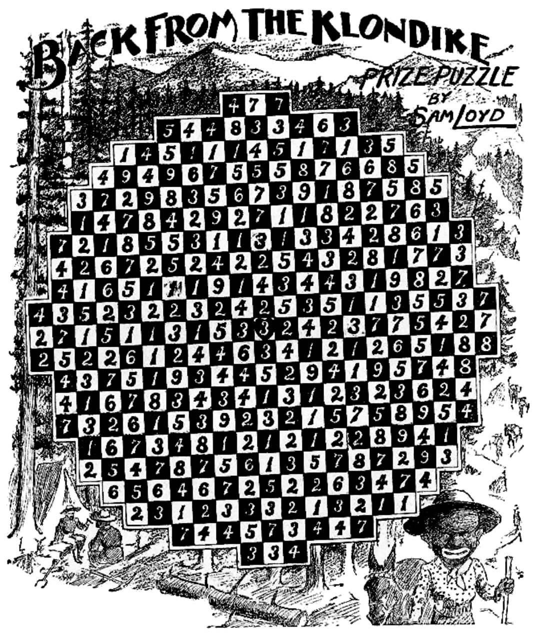

Figure 2: Loyd's puzzle "Back from the Klondike"

The puzzler is directed to a heart-marked start location at the center of a gridded circle. The object is to find a path to a cell from which one can jump one cell beyond the circle's edge. Loyd writes that the puzzle "…was built purposely to defeat Euler's [working backwards] rule and out of many attempts is probably the only one which thwarts his method."

Rook Jumping Maze Generation Assignments

The process of generating Rook Jumping Mazes (RJMs) offers a number of introductory AI teaching opportunities with regards to state representation, uninformed search, stochastic local search, machine learning, and objective function design.

Like programming contest problems, each assignment presents a problem description and an input/output specification independent of programming language. Some assignments require other assignments as prerequisites, and dependencies are clearly described in a table on the RJM Generation home page[3]. Further, a variety of tracks through the dependencies (e.g. Basic, Machine Learning) are suggested for various pedagogical purposes. The basic track can be completed in about 100 lines of Java code, and all exercises can be completed in about two weeks. Relevant online resources and text readings are also suggested.

Representation and Breadth-First Search

We begin by having students first represent a RJM and then find an optimal solution minimizing the number of jumps. The latter exercise provides an opportunity to implement and apply breadth-first search.

[3]`http://modelai.gettysburg.edu/2010/rjmaze/`

Maze Representation: Generate and print a random n-by-n RJM ($5 \leq n \leq 10$) where there is a legal move (jump) from each non-goal state.
Input: an integer $5 \leq n \leq 10$
Output: Initially prompt the user with "Rook Jumping Maze size (5-10)? ". Given valid input, print the randomly generated RJM 2D-array of jump numbers, with jump numbers separated by a single space.

Maze Evaluation: Generate and print a random 5-by-5 RJM where there is a legal move (jump) from each non-goal state. Then, for each cell, compute and print the minimum number of moves needed to reach that cell from the start cell, or "- -" if no path exists from the start cell, i.e. the cell is unreachable.

First, generate and print a random 5-by-5 RJM according to the Maze Representation specification.

There are many features of a good RJM. One obvious feature is that the maze has a solution, i.e. one can reach the goal from the start. One simple measure of maze quality is the minimum number of moves from the start to the goal. For simplicity, we will limit our attention to these, although consideration of other features is an interesting exercise (see Maze Evaluation II).

Using breadth-first search, or some other suitable graph algorithm, compute the minimum distance (i.e. depth, number of moves) to each cell from the start cell. Create an objective function (a.k.a. energy function) that returns the negated distance from start to goal, or a large positive number (e.g. 1,000,000) if no path from start to goal exists. Then the task of maze generation can be reformulated as a search through changes in the maze configuration so as to minimize this objective function.
Input: (no input)
Output:

- Print a randomly generated 5-by-5 RJM 2D-array of jump numbers, with jump numbers separated by a single space.
- Print a blank line.
- Print "Moves from start:" on a single line.
- Print the 2D-array of corresponding cell depths (i.e. minimum moves from start) separated by spaces. For unreachable cells, print "- -". For other cells, print the depth right-justified in a width of two characters. Print a blank line.
- Print the output of your objective function on a single line.

Since uninformed search topics are sometimes taught in the context of graph algorithms in a Data Structures (CS2) course, these first exercises can be useful in that context as well. We would suggest, however, the alternative goal of solving an interesting *given* RJM rather than a random RJM. A related programming contest problem specification and a source of daily RJMs are available online[4].

Stochastic Local Search

Having represented the RJM and applied breadth-first search to the computation of the shortest solution path (or the fact

[4]`http://tinyurl.com/rjmaze`

no such path exists), we can now easily apply stochastic local search (Hoos & Stützle 2005) to the task of generating "better" mazes. Our simple objective function defines "better" as having a solution and preferring a longer shortest solution path. Without loss of generality, our stochastic local searches will minimize this function.

It is important to stress the difference between a *state* in maze solving (i.e. a maze cell) and a *state* in maze generation (i.e. an entire maze configuration). With stochastic local search, we search the space of possible maze *configurations* or *designs* for the best one according to our objective function measure.

Hill Descent: Using a form of stochastic local search called Hill Descent, search the space of 5-by-5 RJMs for a given number of iterations and print the best maze evaluated according to the objective function in the Maze Evaluation specification.

Given a number of iterations from the user, first generate a random 5-by-5 RJM according to the Maze Representation specification and evaluate it according to the Maze Evaluation specification. Then for the given number of iterations:

- For a random, non-goal cell, change the jump number to a different, random, legal jump number.
- Re-evaluate the objective function according to the Maze Evaluation specification.
- If the objective function has not increased (worsened), accept the change and store the RJM if its evaluation is the best (minimum) evaluated so far. Otherwise, reject the change and revert the cell to its previous jump number.

Finally, print the RJM with the best (minimum) objective function evaluation according to the Maze Evaluation output specification.

More formally, let jump function j be a mapping from cells to legal jump numbers, or 0 in the case of the goal cell. Let objective function (or energy function) e be a function we seek to minimize over jump functions. Let step be a function that takes a jump function j, and generates a "neighboring" jump function j' by making a single, stochastic change: function step chooses from non-goal cells with equal probability, and for that cell c chooses $j'(c)$ from all legal jump numbers not equal to $j(c)$ with equal probability. For all other cells, $j'(c) = j(c)$. Then we may describe our algorithm as follows:

```
Let $j$ be chosen at random, and $j_{\mathrm{best}} \leftarrow j$.
For a given number of iterations:
    $j' \leftarrow \mathrm{step}(j)$
    if $e(j') \leq e(j)$
        $j \leftarrow j'$
        if $e(j) \leq e(j_{\mathrm{best}})$
            $j_{\mathrm{best}} \leftarrow j$
return $j_{\mathrm{best}}$
```

Input: a positive integer number of iterations for hill descent optimization

Output:

- Initially prompt the user with "Iterations? ".
- After the hill descent iterations, print the RJM with the best (minimum) objective function evaluation according to the Maze Evaluation output specification.

Hill Descent with Random Restarts: Using a form of stochastic local search called Hill Descent with Random Restarts, search the space of 5-by-5 RJMs for a given number of iterations and print the best maze evaluated according to the objective function in the Maze Evaluation specification.

One problem with pure hill descent is that stochastic local search may become trapped in local minima, where every local step is uphill, making things worse. One escape strategy is to restart search periodically. Another way of viewing this is that we iteratively perform pure hill descent, starting each descent at a random state. The end result is the best result from all descents.

```
Let $j$ be chosen at random, and $j_{\mathrm{best}} \leftarrow j$.
For a given number of searches:
    For a given number of iterations:
        $j' \leftarrow \mathrm{step}(j)$
        if $e(j') \leq e(j)$
            $j \leftarrow j'$
            if $e(j) \leq e(j_{\mathrm{best}})$
                $j_{\mathrm{best}} \leftarrow j$
    Let $j$ be chosen at random.
return $j_{\mathrm{best}}$
```

Input:

- a positive integer number of iterations for hill descent optimization
- a positive integer number of hill descents

Output:

- Initially prompt the user with "Iterations? " and "Hill descents? ".
- After all hill descents, print the RJM with the best (minimum) objective function evaluation according to the Maze Evaluation output specification.

Hill Descent with Random Uphill Steps: Using a form of stochastic local search called Hill Descent with Random Uphill Steps, search the space of 5-by-5 RJMs for a given number of iterations and print the best maze evaluated according to the objective function in the Maze Evaluation specification.

Another strategy for escaping local minima is to allow uphill steps with some small probability. In this exercise, you will modify hill descent to allow uphill steps with a given fixed probability p.

In the Hill Descent algorithm, change "if $e(j') \leq e(j)$" to "if $e(j') \leq e(j)$ or with probability p". Note: With $p = 0$, this degenerates to pure hill descent. With $p = 1$, this degenerates to random walk.

Input:

- a positive integer number of iterations for hill descent optimization
- a probability for the acceptance of an uphill step

Output:

- Initially prompt the user with “Iterations? ” and “Uphill step probability? ”.
- After the hill descent iterations, print the RJM with the best (minimum) objective function evaluation according to the Maze Evaluation output specification.

Simulated Annealing: Using a form of stochastic local search called Simulated Annealing, search the space of 5-by-5 RJMs for a given number of iterations and print the best maze evaluated according to the objective function in the Maze Evaluation specification.

One problem with Hill Descent with Random Uphill Steps is that all uphill steps are equally likely. A small uphill step would generally be more desirable than a large uphill step. Simulated annealing is a stochastic local search technique based on an analogy to energy distributions in heated materials as they cool (i.e. anneal) and “seek” a lower energy state. For example, blacksmiths long ago observed that quenching, i.e. rapid cooling, would lead to a harder, more brittle metal than annealing, i.e. slow cooling, which yields a more malleable metal with a lower energy and more crystalline configuration.

When the material is heated (high energy input), atoms reconfigure among different possible energies much like a random walk. When the material is rapidly cooled (low/no energy input), atoms reconfigure to lower energy states often getting trapped in local minima. The local minima escape strategy of simulated annealing concerns a temperature schedule, called an annealing schedule, that gradually shifts search from a free random walk to a final descent, while favoring smaller uphill steps over larger ones. In a nutshell, for local minima, there are temperatures where one is more likely to escape than reenter. For more on simulated annealing, see the recommended background readings online.

The practical application of simulated annealing here involves very few modifications to Hill Descent with Random Uphill Steps. Using the definitions of hill descent, we may describe our modified algorithm as follows:

```
Let $j$ be chosen at random, and $j_{\text{best}} \leftarrow j$.
Let $T \leftarrow T_0$, where $T_0$ is the initial temperature.
For a given number of iterations:
    $j' \leftarrow \text{step}(j)$
    if $e(j') \leq e(j)$
        $j \leftarrow j'$
        if $e(j) \leq e(j_{\text{best}})$ or with probability $\frac{\exp(e(j)-e(j'))}{T}$
            $j_{\text{best}} \leftarrow j$
    $T \leftarrow T \times d$, where $d$ is the iteration temperature decay.
return $j_{\text{best}}$
```

Thus, simulated annealing in this form takes three parameters: number of iterations, initial temperature, and temperature decay rate. Note the acceptance probability for uphill steps: $\frac{\exp(e(j)-e(j'))}{T}$. When the temperature is high, this is close to $\exp(0) = 1$ and acceptance of any uphill step is very likely. As the temperature drops to zero, this approaches $\exp(\infty) = 0$ so any uphill step would be rejected. In between, a larger uphill step leads to lower probability acceptance.

Input:

- a positive integer number of iterations for hill descent optimization
- a positive floating-point initial temperature
- a positive floating-point geometric decay rate (usually chosen slightly less than 1.0)

Output:

- Initially prompt the user with “Iterations? ”, “Initial temperature? ”, and “Decay rate? ”.
- After the simulated annealing iterations, print the RJM with the best (minimum) objective function evaluation according to the Maze Evaluation output specification.

Finally, we note that this project nicely integrates with the method for teaching stochastic local search described in (Neller 2005).

Machine Learning

There is a time-quality tradeoff in RJM generation. The stochastic local search algorithms described are anytime algorithms that incur definite computational costs for possible quality improvements, so this presents a learning challenge for metalevel control of stochastic local search. In these exercises, reinforcement learning techniques are used to find optimal parameters for balancing time and quality when generating large numbers of RJMs.

Restart Bandit: Use ϵ-greedy and softmax strategies to learn an approximately optimal restart period that maximizes generated maze utility per unit of computation.

“One-armed bandit” is a slang reference to a slot machine. A well-known problem in psychology and machine learning is the n-armed bandit problem where the learner is faced with a set of single actions with variable payouts and must balance explorative actions that gain information about expected action utility, versus exploitative actions that use such information to maximize utility.

In this case, let us suppose that our goal is to create a RJM generator that uses hill descent with random restarts in order to yield mazes with the longest solutions most efficiently. Further, let us suppose that we value a maze with goal depth d twice as much as a maze with goal depth $(d-1)$. Finally, suppose that computation has a uniform cost per iteration. Let $d = 0$ for a maze without a solution. Then we can measure the utility of a single hill descent of i iterations as $\frac{2d}{i}$.

In this exercise, we will implement two simple techniques for learning the approximate best number of iterations i that maximizes the expected utility $\frac{2d}{i}$ for hill descent. Then one can restart after i iterations and expect to maximize expected utility of the generator per unit of computation.

For each method, we will create, in effect, a 6-armed bandit by allowing 6 choices for the number of iterations between each restart: 100, 200, 400, 800, 1600, 3200, that is, $100 \times 2^a, a \in \{0, \ldots, 5\}$. Each method will learn the utility of actions by using a action selection strategy that allows explorative actions while largely taking actions that are expected to be superior.

The ϵ-greedy strategy for action selection chooses a random action with probability ϵ, and otherwise chooses an action with the maximum average utility experienced so far. If $\epsilon = 1$, this is purely explorative. If $\epsilon = 0$, this is purely exploitative.

The softmax strategy for action selection assigns a probability to the choice of each action a according to the normalized weight $\exp(U(a)/\tau)$ for some given constant τ. That is, we compute the sum of $\exp(U(a)/\tau)$ for all a, and divide each individual term of the sum by the sum to get the action probability.

Evaluate ϵ-greedy strategy and softmax strategy for Hill Descent generation of RJMs. Evaluate three different values of ϵ and τ for ϵ-greedy strategy and softmax strategy, respectively, for 10000 iterations each. Print tables of the data collected for each 10000 iterations of learning.

Restart SARSA: Use the SARSA algorithm to learn an approximately optimal restart policy that maximizes the expected generation of long-solution 5x5 RJMs per unit of computation.

Let us suppose that, for the purposes of this exercise, we wish to maximize the generation, per unit of computation, of RJMs with minimum solution path lengths of at least 18 moves, i.e. with goal depth ≥ 18. When we terminate optimization at a threshold of satisfaction for our evaluation function, this is called satisficing. At every decision point, we will choose between two actions:

Go - Execute another 250 iterations of Hill Descent.

Restart - Randomize the maze state.

In our decision making we will consider two factors:

- computational time since restart - Let time t denote the number of successive Go actions leading to the current state. Thus, the number of iterations since the last random restart is $250 \times t$.
- quality of current maze state - Let d be the current maze goal depth. Let $d = 0$ for a maze without a solution.

Thus, the current state can be described by the pair (t, d). To limit the size of our learning state space, we consider actions only for 5 values of $t \in (0, \ldots, 4)$ and 6 values of $d \in (17, 16, 15, 14, 13, 12)$. If $d > 17$, we have generated a satisficing maze and terminate search. Let us treat all mazes with $d < 12$ as being in a state with $d = 12$. In other words, $d = 12$ actually represents the set of all maze depths ≤ 12. If $t > 4$, we consider the hill descent to have taken too long, and we force the choice of the Restart action.

The goal of this exercise is to apply the SARSA algorithm to learn the optimal policy for Go/Restart actions so as to maximize the expected number of satisficing mazes (with $d \geq 18$) per unit of computation, i.e. number of 250 iteration hill descent stages.

The SARSA algorithm is an on-policy temporal difference control method for estimating the expected utility for the current policy for each state and action pair. The policy we will use is an ϵ-greedy policy with $\epsilon = 0.1$.

After taking an action in a state, there is an immediate reward. This reward can be negative (i.e. a cost). For each Go action, there is reward of -1 reflecting computational cost. For each satisficing maze, there is a reward of $+1$. Thus, a Restart action that (with very low probability) yields a satisficing maze would have a reward of 1, whereas the Restart reward would be 0 otherwise. A Go action that yields a satisficing maze would have a reward of $-1 + 1 = 0$, whereas the Go reward would be -1 otherwise.

For the SARSA algorithm (Sutton & Barto 1998), we use a learning rate $\alpha = 0.1$ and a discount rate $\gamma = 1$. Execute SARSA learning for 2000 episodes. (Use fewer episodes for testing.) Print out the table of Q value estimates and how many updates occurred for each state. Finally evaluate the policy by comparing the number of hill descent iterations needed to generate 100 satisficing puzzles for a greedy policy (i.e. $\epsilon = 0$), and policies that only restart at $t = 1, 2, 3, 4, 5$.

Maze Evaluation II

Students, faculty, and staff at Gettysburg College voluntarily participated in two iterations of testing and critique of mazes generated by stochastic local search. Each iteration yielded new insights and maze features that may be of use when constructing an improved energy function. A complete description of our design considerations was presented at the International Conference on Computers and Games 2010 (Neller *et al.* 2011), and are the basis for the suggestions in this advanced design exercise.

Our initial design experiences and those of our students completing this assignment suggest that Rook Jumping Maze generation presents an excellent opportunity for design creativity. After student(s) have solved and studied a number of puzzles generated with the simple objective/energy function, the time is ripe to ask how the objective/energy function may be improved to generate better mazes. In this exercise, student(s) seek to improve subjective quality of generated mazes by forming subjective preferences and skillfully balancing such preferences in the definition of an improved objective/energy function.

Problem: Define a better maze evaluation function and argue why it leads to improved maze quality. Compare and contrast the results obtained with your new evaluation function versus the previous evaluation function using the same stochastic local search technique for each.

There are many features of a good RJM. Previously, we evaluated RJMs according to solvability and the shortest distance to the goal. However, there are many other features that may be considered as well. Here, we describe a few:

- *Black holes*[5] - A black hole is a dead-end. Define a *reachable* cell to be a cell one can reach from the start through a sequence of legal moves. Define a *reaching* cell to be a cell from which one can reach the goal through a sequence of legal moves. A cell is part of a black hole if it is a reachable, non-reaching cell.

[5]The descriptive maze terms "black hole" and "white hole" were coined by maze designer Adrian Fisher.

- *White holes* - Some puzzlers seek to trace backwards from the goal to the start. A white hole is a back-tracing dead end. A cell is part of a white hole if it is an unreachable, reaching cell.
- *Start and goal positions* - Is anything gained/lost by allowing the position of the start and/or goal cell to vary?
- *Shortest solution uniqueness* - How does knowledge that there is a unique shortest solution affect the maze solving experience?
- *Forward/backward decisions* - Sometimes there is only a single legal "forced" move from a cell. Sometimes only a single move can lead to a cell. How do the number of forward/backward decisions affect the maze solving experience?
- *Same jump clusters* - Same jump clusters are sets of cells that all have the same jump number and reach each other without leaving the cluster. How do same-jump clusters affect the maze solving experience? Are some better than others?
- *Solution direction variability* - How do back-and-forth move sequences within the same row or column affect the maze solving experience?

Select one or more of the features above that you consider most important, compute measures for maze evaluation, and seek to improve upon the previous RJM evaluation function. Generate sets of puzzles using the old and new evaluation functions, compare and contrast the puzzles, and argue why your new evaluation function improves the average quality of your mazes.

Project Variations

Many rich possibilities for creative variants exist, so it is not difficult to craft a unique (i.e. not easily plagiarized) assignment experience for your students.

One may vary *tiling* of the maze, using different regular tilings, e.g. triangular or hexagonal. Semiregular and other tilings present different interesting possibilities at the risk of yielding movement instructions that are difficult for many to grasp.

Additional *topological* constraints may be added or removed, such as allowing toroidal wrap-around grid boundaries, or creating additional graph connectivity as in the abstract strategy board game Surakarta.[6] Simple means of adding constraints include the addition of impassable walls between tiles, impassable tiles, or tiles which may be passed over but cannot be a move destination.

Movement constraints may be varied as well. With the addition of diagonal moves, the Rook Jumping Maze becomes a Queen Jumping Maze. Robert Abbott's "no-U-turn" rule[7] increases state complexity so that the current state must be described as the product of the row, the column, and the previous move direction.

Most of the design considerations we outline remain relevant to these variations.

[6]`http://en.wikipedia.org/wiki/Surakarta_(game)`

[7]`http://www.logicmazes.com/n4mz.html`

Conclusion

From these descriptions and experience in solving such mazes, one can see that the fun challenge of generating Rook Jumping Mazes allows a variety of learning experiences that bring together several AI topics, including problem representation, breadth-first search, stochastic local search algorithms, reinforcement learning algorithms, and the design of objective/utility functions.

In game/puzzle-related assignment design, it is important to achieve a high fun-to-SLOC (Source Lines of Code) ratio so that the experiential learning objectives are not lost in the midst of programming complex rules and constraints. Rook Jumping Mazes, with their extremely simple rules and representation yet difficult and engaging play, indeed offer much fun learning experience in relatively few lines of code.

These assignments are archived online as a 2010 Model AI Assignment of the First Symposium on Educational Advances in Artificial Intelligence[8], with further exercise details, sample transcripts, and recommended online and text readings.

[8]`http://modelai.gettysburg.edu`

References

Hoos, H. H., and Stützle, T. 2005. *Stochastic Local Search: foundations and applications*. San Francisco: Morgan Kaufmann.

Loyd, S. 1914. *Sam Loyd's Cyclopedia of 5000 Puzzles, Tricks, and Conundrums with Answers*.

Neller, T.; Fisher, A.; Choga, M.; Lalvani, S.; and McCarty, K. 2011. Rook jumping maze design considerations. In van den Herik, H. J.; Iida, H.; and Plaat, A., eds., *LNCS 6515: Proceedings of the Computers and Games, 7th International Conference, CG 2010, Kanazawa, Japan, Sept. 24-26, 2010, revised selected papers*, 188–198. Springer.

Neller, T. 2005. Teaching stochastic local search. In *Proceedings of the 18th International FLAIRS Conference (FLAIRS-2005), Clearwater Beach, Florida*, 8–13. AAAI Press.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: an introduction*. Cambridge, Massachusetts: MIT Press.