CMU-CS-83-144

ROG-O-MATIC: A Belligerent Expert System

Michael Mauldin, Guy Jacobson,
Andrew Appel and Leonard Hamey

July 8, 1983

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

Rog-O-Matic is an unusual combination of algorithmic and production systems programming techniques which cooperate to explore a hostile environment. This environment is the computer game **Rogue**, which offers several advantages for studying exploration tasks. This paper presents the major features of the Rog-O-Matic system, the types of knowledge sources and rules used to control the exploration, and compares the performance of the system with human Rogue players.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

List of Figures

1. Introduction

Rog-O-Matic[1] plays the computer game Rogue, wherein the object is to explore and survive a complex and hostile environment. Rogue is an example of what we have called an *exploration task*.[2] Studying exploration requires two things: terrain to be explored, and an explorer to search it. There are three major advantages for choosing Rogue as a terrain generator:

- It is a pre-existing game designed for human play.
- It has an objective, scalar measure of performance (*i.e.* the score).
- There is an abundance of human volunteers to calibrate the performance measure.

The Rogue exploration task is complicated by adversaries whose goal is to prevent the explorer from reaching the lower levels. Carbonell has studied the problem of planning in an adversary relationship [2], but planning in Rogue is hampered by the randomness of the adversary. Where the probabilties are known, search trees can be built with each possible next state labelled with its transition probability. The action with the highest expected value can then be selected [1]. In Rogue, however, these probabilities are unknown to the player, and can change from game to game. *Scenarios* have also been used to analyze combat situations [7], but when unseen opponents can appear at any time, and when many of the parameters of the combat are unknown, choosing the right scenarios can be very difficult.

Rog-O-Matic is designed as an expert system because of the intractibility of search in the complicated Rogue environment (there are as many as 500 possible actions at any one time; for each action there are several thousand possible next states).

Rog-O-Matic differs from other expert systems in the following respects:

- It solves a dynamic problem rather than a static one.
- It plans against adversaries.
- It plays a game in which some events are determined randomly.
- It plays despite limited information.

In this paper we introduce the Rogue environment and discuss the architecture, knowledge sources, and production rules used by Rog-O-Matic to explore that environment. We also discuss the system's implementation and compare its performance with that of human Rogue experts.

---

[1] Copies of the Rog-O-Matic source are publicly available using FTP over the ARPAnet. These are only likely to be useful at Unix sites. For more information, send mail to Michael Mauldin (Mauldin@CMU-CS-A).

[2] Given an undirected planar graph, a starting node, and a visibility function which maps each node into a subset of nodes, exploration entails traversing edges so as to see all of the nodes. Minimizing the number of nodes visited is a subgoal.

2. The Rogue Environment

The Rogue game is described in more detail in [6]; an overview is given here. The environment is a cave composed of levels; each level contains between six and nine rectangular rooms connected by tunnels. The game player takes the role of explorer, searching for gold and fighting the monsters which guard the cave. Magical items such as potions or weapons are also scattered about, and these can be used to advantage in combat. Traps are hidden throughout the rooms to thwart careless players.

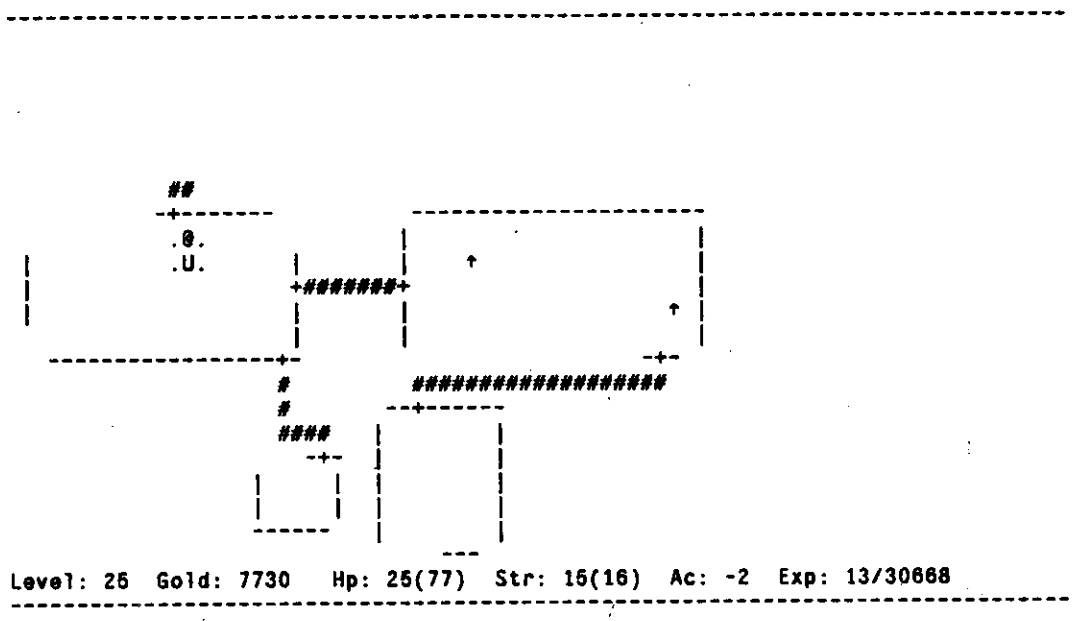

Figure 2-1: A Sample Rogue Screen

Figure 2-1 shows a sample screen from one of Rog-O-Matic's games against Rogue 5.2. Rog-O-Matic is represented by the "@" sign, and the "U" is a monster called an *Umber Hulk*, which is about to force the program to quit (rather than face certain death). The carats "^" are traps which the program has previously sprung. The bottom line displays Rog-O-Matic's fighting abilities or *stats*; they represent his *Gold*, *Hit Points*, *Strength*, *Armor Class*, and *Experience Level*. Numbers in parentheses represent the maximum value of the *Hit Point* and *Strength* attributes.

The object of the game is to explore to the 26th level and retrieve the *Amulet of Yendor*. As the explorer goes deeper into the cave, the gold becomes more plentiful and the monsters become more difficult to defeat. As the explorer kills monsters in combat, he becomes a better fighter. On the average, monsters increase in ferocity faster than the player increases his skill, so reaching the 26th level is almost impossible. The player's score is the amount of gold he obtains, with bonuses given for retrieving the amulet. Being killed costs the player ten percent of his gold, so in some cases it is advisable to quit rather than to fight "to the death".

3. The Expert System Architecture

Rog-O-Matic is a combination of algorithmic knowledge sources and production rules. Where uncertainty is low, algorithmic methods allow fast calculation of relevant information. Where uncertainty is high, heuristic production rules provide reasonable behavior. Knowledge sources and production rules interact through a world model similar to the blackboard used by the Hearsay-II system [3]. Because the system is implemented in an imperative language, the production rules are statically ordered; dynamically ordered rules would have been much more complicated to code.

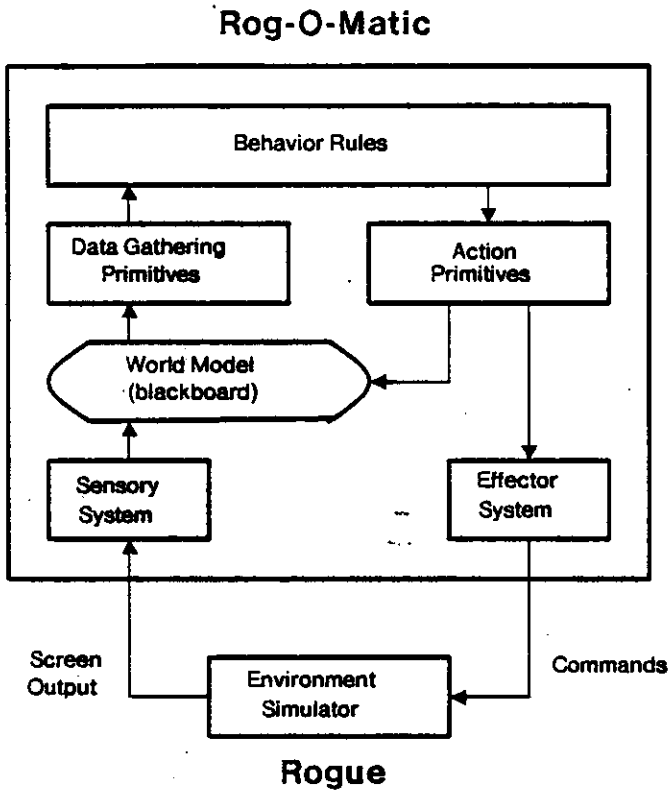

Figure 3-1: Rog-O-Matic System Architecture

Figure 3-1 shows the Rog-O-Matic high-level system architecture. Rog-O-Matic plays the game by intercepting characters from the terminal stream, and sending command characters back to the Rogue process. The input interface must convert a stream of characters designed to draw a meaningful picture on a CRT screen into a representation of the world state. This function is performed by the *sensory* module. The less difficult task of converting motion and action directives into Rogue commands is done by the *effector* module, which is shown mainly for symmetry. Langley *et al* have described the advantages of simple sensory/effector interfaces in studying a reactive environment [5].

The world model is operated on by a variety of modules, most of which take low level data from the world model and process it into higher level information. Eventually, the information is encoded at a

sufficiently high level for the rules to operate directly on it. The action primitives can also change the world model, allowing production rules to encode inference mechanisms. Short term memory is also implemented by using a portion of the world model to store internal state information. A partial list of knowledge sources is given here:

- Sensory system (**sense**) Builds low level data structures from Rogue output.
- Object map (**objmap**) A data structure which tracks the location and history of objects in the environment (such as weapons or monsters).
- Inventory handler (**invent**) A database of items in Rog-O-Matic's pack.
- Terrain map (**termap**) A data structure recording the terrain features of the current level.
- Connectivity analyzer (**connect**) Finds cycles of rooms (loops).
- Path calculation (**pathc**) Performs weighted shortest path calculations.
- Internal state recognizer (**intern**) Tracks the health and combat status of Rog-O-Matic.

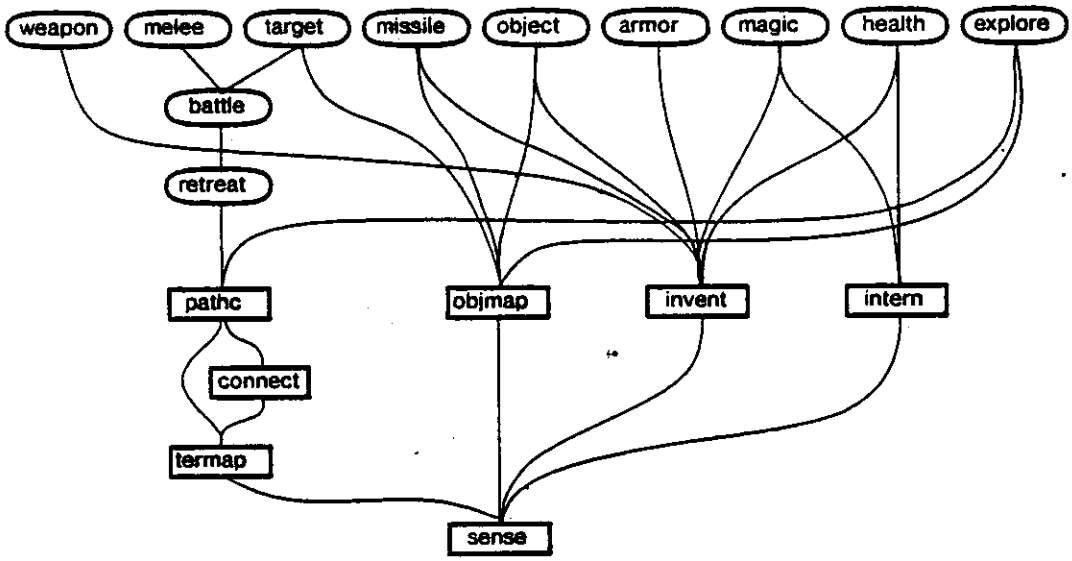

Figure 3-2: Experts (ovals) and Knowledge Sources (boxes)

The rules are grouped hierarchically into experts, each of which performs a related set of tasks. These experts are ordered statically (in rough order of contribution to survivability). For example, if the melee expert decides that a monster must be dispatched, its decision overrides the advice of the object acquisition expert calling for an object to be picked up. If the melee expert suggests no action, then the object acquisition expert's directive is acted upon. The basic structure resembles a directed acyclic graph (DAG) of *IF-THEN-ELSE* rules. Figure 3-2 shows the relationships between these experts.

Although we consider the static ordering a drawback (for reasons described in section 6) it does have two advantages. The first is ease of coding, and the second is the prevention of looping. When subtle features of the situation can reorder the rules, the system may oscillate between two courses of

action (in Rogue 5.2, the player would starve to death). This is similar to many chess programs which can be drawn by taking advantage of the three move rule. The richness of the Rogue environment makes loop detection difficult, but static rules facilitate reasoning about the code to verify the absence of loops. All that is needed is to verify that each expert makes progress toward its own goals, and that each goal is abandoned only for a higher priority goal. These are the most important experts:

- Weapon handler (**weapon**) Chooses weapon to wield.
- Melee expert (**melee**) Controls fighting during combat.
- Target acquisition expert (**target**) Controls pursuit of targets.
- Missile fire expert (**missile**) Fires missiles (arrows, spears, rocks, *etc.*) at distant targets.
- Battlestations expert (**battle**) Performs special attacks, initiates retreat.
- Retreat expert (**retreat**) Uses the terrain map and connectivity analysis to choose a path for retreat.
- Object acquisition expert (**object**) Picks up objects.
- Armor handler (**armor**) Chooses armor to wear.
- Magic item handlers (**magic**) Manipulates magic items.
- Health maintenance (**health**) Decides to eat when hungry and to heal damage by resting.
- Exploration expert (**explore**) Chooses next place to explore, and controls movement.

Of the algorithmic knowledge sources, the path calculator is the most interesting. It reads the terrain map and determines weighted shortest paths from Rog-O-Matic to the nearest location satisfying specified criteria (such as the nearest unknown object, or the nearest escape route). The edge costs are small, bounded integers which encode known or possible dangers (such as traps or unexplored squares). A breadth-first search explores outward from the current location, and any square with a non-zero cost (an *avoidance* value) has its cost decremented and then square is placed unexamined at the end of the search queue. The result is a weighted shortest path obtained in time linear to the number of terrain squares. Since exploration requires the system to spend most of its time moving, a fast path calculator is essential.

Another algorithmic knowledge source is the connectivity analyzer. *Connect* performs a depth-first search of the terrain to detect loops in corridors. This information is a resource for the retreat expert. Since the Rogue player can heal damage sustained in combat, while most monsters do not heal, these monsters can be defeated regardless of their strength if only the player can retreat far enough (and heal his injuries while doing so). The combination of the *connect* knowledge source and the *retreat* expert contribute heavily to Rog-O-Matic's success.

Most of the actions taken by Rog-O-Matic are simpler than movement, and these actions are performed directly by the production rules. For example, when the *melee expert* has determined that a battle is underway, it puts certain key parameters of the battle, such as the strength of the monster, the number of turns which the monster will require to reach the player, and its direction, into the

blackboard and invokes the *battlestations* expert. *Battlestations* decides whether to attack with the weapon currently in hand, attack with a special magic item, or to retreat. Figure 3-3 shows some sample rules extracted from *battlestations*.

```
/*
 *    Retreat
 *
 * IF we are not confused (so we can be sure of our direction) AND the monster
 *    is not a fungus (which can hold us) AND we could die in one melee round AND
 *    the RETREAT EXPERT finds a retreating move
 *
 * THEN (retreat expert makes move), display the current state to the user.
 */

       if (!confused && monc != 'F' && die_in (1) && runaway ())
       { display ("Run away! Run away!");
         return(1); }

/*
 *    Can't run, use a wand of teleport to banish a very mean monster
 *
 * IF we could die in one melee round AND we have a line of sight (mdir) AND
 *    the monster is adjacent (turns=0) AND we have a wand of "teleport away"
 *
 * THEN point the wand at the monster.
 */

       if (die_in (1) && mdir >= 0 && turns == 0 &&
          (obj = havewand ("teleport away")) >= 0)
       { return (point (obj, mdir)); }

/*
 *     No wand, so use a scroll of teleport to move us out of danger
 *
 * IF we could die in one melee round AND the monster is adjacent (turns=0) AND
 *    we have a scroll of "teleportation"
 *
 * THEN read the scroll.
 */

       if (die_in (1) && turns == 0 && (obj = havenamed (scroll, "teleport")) >= 0)
       { return (reads (obj)); }
```

Figure 3-3: Sample Rules from the *Battlestations* Expert

4. Implementation

Rog-O-Matic runs on a Vax 11/780 under the Berkeley Unix[3] operating system. It is composed of 12,000 lines of C code [4]. By comparison, the Rogue game requires 8,900 lines of C code. C was chosen for two reasons: convenience and speed. It offers direct access to the operating system primitives required to interface to the Rogue game (a necessity since we did not want to change the Rogue game for this project). C code also runs much faster than most production systems languages, and this speed was necessary for Rog-O-Matic to play enough games for its performance to be measured. The program takes about 30 CPU seconds (generating about 400 actions) to explore one level, and Rogue takes an additional 15 CPU seconds calculating its part of the simulation. To date, Rog-O-Matic has played more than 5000 games of Rogue.

---

[3] Unix is a trademark of Bell Laboratories.

Knowledge Source	Expert	Date	10 percentile Score
		Dec-81	
termap		Dec-81	
objmap		Dec-81	
intern		Dec-81	
	target	Dec-81	~1000
pathc		Mar-82	
invent		Mar-82	
explore		Mar-82	~1500
	object	Apr-82	
	health	Apr-82	
	magic	Apr-82	
	melee	Apr-82	~2000
connect		May-82	
	retreat	May-82	~3000
	weapon	Nov-82	
	missile	Nov-82	
	armor	Nov-82	~3500
With subsequent improvements		Jul-83	~4000

Figure 4-1: History of the Rog-O-Matic Program

Rog-O-Matic was started in October of 1981 as a simple project, but it has grown greatly since then. Initially, little thought given to artificial intelligence techniques, but as the Rogue task became better understood, more and more subproblems were discovered which required either search or heuristic methods for solution. We eventually realized that the Rog-O-Matic program had become a large, competent expert system. Figure 4-1 shows this development as various knowledge sources and experts were added to the system. Although this table gives an approximate view of the value of each expert and knowledge source, it should be noted that each part of the system was undergoing continual improvement. In particular, the *explore* expert has been completely rewritten four times—once by each author.

The initial experts and knowledge sources were designed using the authors' knowledge of Rogue, but subsequent modules were developed by a trial and error. Each version of Rog-O-Matic was carefully watched as it played hundreds of games. Human observations, and analysis of game logs would point out specific situations where the program lacked sophistication. These observations were often comparisons made by human experts, usually in the form *"When I'm in that situation, I generally..."*. Whenever a new expert was added, the system's performance (as measured by both score and deepest penetration) was compared to the previous version, and the new expert was retained if performance improved. This methodology was feasible both because the score was an obvious measure of performance, and because a detailed log was kept of each game showing the interactions between the Rog-O-Matic and Rogue processes.

5. Performance

Rog-O-Matic plays as well as human experts, and is particularly good at playing out marginal games which human players often abandon. Figure 5-1 shows percentile plots for 16 of the best Rogue players at Carnegie-Mellon University, sorted by median score. Boxes are drawn from the lower quartile to the upper quartile, with a line at the median. The whiskers are drawn at the 10^{th} and 90^{th} percentile marks, and extreme games are plotted as asterisks. The vertical scale is the score (drawn logarithmically) and the horizontal scale is the player's rank.

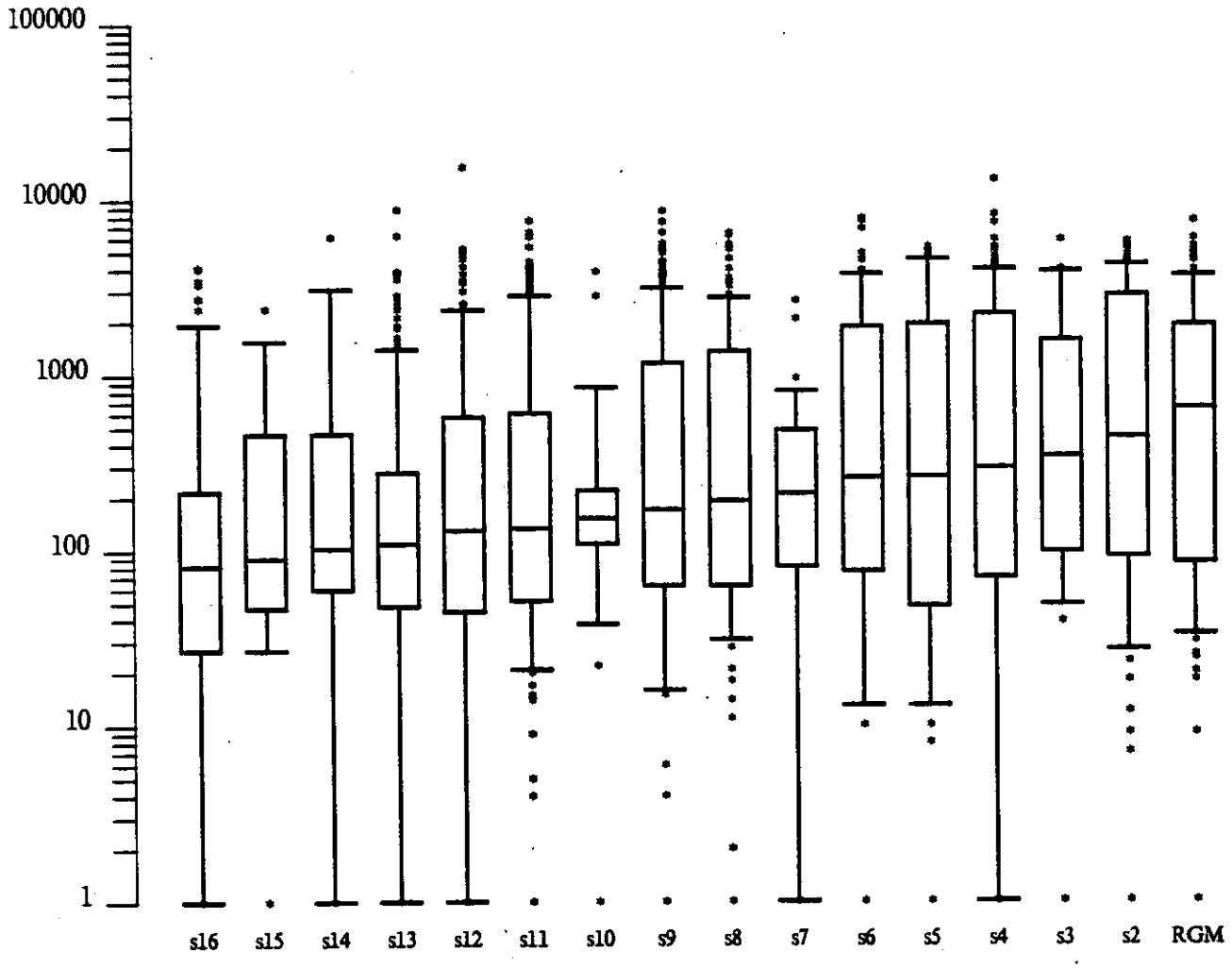

Figure 5-1: Log Scores *vs.* Player, Sorted by Median

The plot includes players with ten or more games against Rogue (version 5.2) on the GP-Vax at CMU. The data were collected over a two month period from January 1, 1983 to February 21, 1983. During this time, Rog-O-Matic played 106 games, and achieved the highest median score of any player. It was also on the Rogue *Top Ten* during most of this time (*i.e.* it was among the ten players with the highest scores). Since February, Rog-O-Matic's median score has increased to 1400, and its

10^{th} percentile score is now 3950. The highest score obtained against Rogue 5.2 so far is 7730. In that game, Rog-O-Matic had found the Amulet and was defeated while returning it to the surface. The final position is, in fact, that shown in Figure 2-1.

6. Problems

The major drawback of the static rule ordering required by the implementation is dealing with the changing nature of the Rogue task. As the explorer goes deeper, new dangers appear. Additional rules are required to cope with these dangers, each with slightly different activation conditions. Dynamic ordering might permit fewer rules to achieve the same effect, and the rules could be grouped together for readability. For example, on levels 16 to 25 there are monsters called *Invisible Stalkers*. The possibility of running into an invisible monster requires that special precautions be taken when resting on these levels. This knowledge must be spread about all rules which are affected, rather than being grouped into one set of "invisible monster strategies".

Another problem is single-mindedness. For example, when running from a monster, it is often possible to pick up objects on the way to an escape route. Since Rog-O-Matic does not consider that a single action may satisfy multiple goals, the program fails to take advantage of such situations. In other cases the presence of a secondary factor requires more creative actions than the static rule ordering can achieve. This problem is most severe when Rog-O-Matic has to combat multiple monsters. Its heuristic is to fight the meanest monster first, but there are situations in which this rule fails miserably. In Rogue, failing miserably usually results in death.

7. Conclusion and Future Work

By combining efficient algorithmic knowledge sources with a rule based control mechanism, we have produced a program demonstrably expert at the game of Rogue. The result is a system capable of performing exploration tasks while also fulfilling the goal of self-preservation. The system can function well in the face of uncertain dangers, make well considered *fight or flight* decisions, and retreat successfully when it faces overwhelming opposition. In addition to producing high median scores, it also has been a regular member of the Rogue *Top Ten* players at CMU for several months. By encoding knowledge of the Rogue game in a procedural form, we have been able to test many tactics and strategies, resulting in higher scores for Rog-O-Matic (and also for several players who helped in developing Rog-O-Matic).

Although the technology used in Rog-O-Matic has not been exhausted, it is becoming more and more difficult to improve its play, both because of Rog-O-Matic's increasing complexity, and because we are approaching the limit of obvious strategies. It is hoped that the addition of a statistical learning module (being implemented by Leonard Hamey) will allow automatic selection between tactics. By testing various tactics proposed by the programmers, the program will assume part of the burden of improving its own performance.

8. Acknowledgements

We would like to thank the Rog-O-Matic user communities at both CMU and at Rice University for their thousands of hours of Rog-O-Matic testing. Murray Campbell, Gordon Goetsch, David Johnson, Paul Milazzo, Walter Van Roggen, Mary Shaw, Hank Walker and Robert Wilber provided many carefully documented observations which were invaluable in debugging the code. The players who provided copious data on human Rogue performance included David Ackley, Thomas Anantharaman, Jim Anderson, Mike Foster, Stephen Hancock, David McDonald, Robert Sansom, Mark Stehlik, Robert Stockton and Alan Sussman.

We are very grateful to Jaime Carbonell and Dave Touretzky for their inspiration and helpful criticisms of drafts of this paper. And of course, special thanks go to Ken Arnold and Michael Toy for creating a very interesting (and very, very hostile) environment.

9. References

1. Hans Berliner, "Experiences in Evaluation with BKG—A Program that plays Backgammon," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* 1977.

2. Jaime Carbonell, "Counterplanning: A Strategy-Based Model of Adversary Planning in Real-World Situations," *Artificial Intelligence,* Vol. 16, 1981.

3. Lee Erman, Frederick Hayes-Roth, Victor Lesser and Raj Reddy, "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," *Computing Surveys,* Vol. 12, No. 2, June 1980.

4. Brian Kernighan and Dennis Ritchie, *The C Programming Language,* Prentice-Hall, 1978.

5. Pat Langley, David Nicholas, David Klahr and Greg Hood, "A Simulated World for Modelling Learning and Development," *Proceedings of the Third Annual Conference of the Cognitive Science Society,* 1981.

6. Michael Toy and Kenneth Arnold, "A Guide to the Dungeons of Doom," Unix Documentation, Department of Electrical Engineering and Computer Science, University of California,.

7. Rajendra Wall and Edwina Rissland, "Scenarios as an Aid to Planning," *Proceedings of the second meeting of the American Association for Artificial Intelligence,* 1982.