SOLVING SUDOKU PUZZLES WITH THE COUGAAR AGENT ARCHITECTURE

by

Michael Ray Emery

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

July 2007

APPROVAL

of a thesis submitted by

Michael Ray Emery

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Dr. John Paxton

Approved for the Department of Computer Science

Dr. Michael Oudshoorn

Approved for the Division of Graduate Education

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U. S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Michael Emery

July, 2007

ACKNOWLEDGEMENTS

I deeply thank Dr. John Paxton, my advisor, for providing direction to my research, reviewing my work, and editing this thesis; my wife, Ginny, for her endless love, support, encouragement, and thesis editing; SRI International for their flexibility in allowing me to take 2 weeks off work to complete this thesis; the Meyers for providing hot meals and a place to stay during my final push for completion; and God, with whom, all things are possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The Cougaar distributed agent architecture, originally a DARPA-funded research project, provides a platform for developing distributed agent systems. Testing its ability to solve complex problems using a large number of agents in an interesting topic of research.

In this thesis, the Cougaar distributed agent architecture is studied from the standpoint of Sudoku. Through analysis and experimentation, insight is gained into both the properties and weaknesses of Cougaar. Cougaar's performance when solving Sudoku puzzles is then compared with other Sudoku solving techniques. The Cougaar agent approach solves Sudoku puzzles in a human-like fashion, reaching solutions using both analysis and guessing.

Cougaar is shown to be capable of solving Sudoku puzzles in a distributed agent architecture. Although not as fast as traditional techniques, the Cougaar distributed agent approach is able to provide solutions to Sudoku puzzles using fewer guesses. Additionally, solving Sudoku puzzles with Cougaar exposed some reliability issues and demonstrated the overhead required for communication. These results open the way for additional study of Cougaar.

PROBLEM STATEMENT

The purpose of this thesis is to study the Cougaar distributed agent architecture by applying it to the Sudoku problem and to understand some of the properties and weaknesses of Cougaar. Cougaar provides a stable platform for distributed agent work by handling many of the basic requirements for an agent system, such as agent-to-agent communication and service discovery.

Sudoku is an interesting problem to solve with the Cougaar distributed agent architecture. Each square on the Sudoku board can be treated as an autonomous agent. These agents then communicate with neighbors to gather enough information to solve the Sudoku puzzle as a community. This distribution creates a large number of agents and provides an environment to study Cougaar. Traditional, single CPU approaches to solving Sudoku puzzles are also discussed. These provided a baseline for comparison in the experiments.

Each agent in the Cougaar system only had a limited amount of analytical power. More difficult Sudoku puzzles required an additional agent to make guesses when needed. Three Sudoku problem solving methods were tested: a random choice from a random worst agent, a random choice from any random agent, and a random choice from a random best agent. Here, best indicates an agent is from the set of agents with the fewest possibilities, and worst is the set with the most possibilities.

Two sets of Sudoku puzzles were tested. The first consisted of actual Sudoku puzzles from WebSudoku.com [1]. The second was randomly generated. For both sets, the single CPU approaches performed faster and more reliably than the Cougaar distributed agent Sudoku problem solving techniques. Although the Cougaar Sudoku problem solving techniques were much slower, they consistently solved each Sudoku puzzle with significantly fewer guesses. The testing showed that the Cougaar distributed agent architecture is capable of being applied to solve Sudoku puzzles.

During testing, Cougaar was shown to have some reliability issues. Agents would become unresponsive and would no longer receive messages sent to them. Restarting the agent was the only way to return the system to a working state. Additionally, messages between functional agents would not be delivered. The overhead required to send messages among agents and ensure the delivery of commands and information greatly outweighed any work done by individual agents focused toward solving the Sudoku puzzle itself. These findings show some of the weaknesses of Cougaar and open the way for additional research.

RELATED WORK

Introduction to Sudoku

Sudoku is often thought to have originated in Japan, due to the name and its popularity as a Nikoli-published puzzle game in Japan since the 1980's [2]. However, Dell Magazines previously published it under the name Number Place in the late 1970's [3]. The game also has roots in Latin Squares, a game thought to have been invented by Leonhard Euler in the 1700's [4]. Regardless of origin, the game has recently become internationally popular and can be found in a number of mainstream newspapers.

Solving Sudoku

Sudoku is a number logic game where the goal is to fill a square grid with numbers relating to the size of the puzzle. The typical Sudoku puzzle is a nine by nine grid that must be filled using the numbers one through nine. The grid is divided into rows, columns, and subsquares. A subsquare is an n by n square, where n is the square root of the number of columns or rows. For a standard Sudoku puzzle, n is three. All standard Sudoku puzzle will be already partially filled in (Figure 1, left puzzle). The information provided by these preexisting numbers determines the general difficulty level of the puzzle. The goal is to arrange numbers on the grid so that every number appears only once in each row, column, and subsquare (Figure 1, right puzzle).

	1	3		9			8	6
		2				9	4	
	4		1				3	
	2		8		7			
		5	9	4	1	6		
			5		3		1	
	5				8		6	
	3	1				4		
6	9			3		1	7	

5	1	3	7	9	4	2	8	6
8	7	2	3	5	6	9	4	1
9	4	6	1	8	2	7	3	5
1	2	4	8	6	7	5	9	3
3	8	5	9	4	1	6	2	7
7	6	9	5	2	3	8	1	4
4	5	7	2	1	8	3	6	9
2	3	1	6	7	9	4	5	8
6	9	8	4	3	5	1	7	2

Figure 1. A Sudoku Puzzle and its Solution.

A properly formed Sudoku puzzle has only one correct solution. Showing that a candidate Sudoku puzzle has more than one solution was proved to be NP-Complete by Yato and Seta [5]. The total number of properly formed Sudoku puzzles is 6670903752021072936960, as enumerated by Felgenhauer and Jarvis [6]. Algorithms for solving Sudoku generally follow two styles: rapid computer algorithms and human techniques.

Computer Algorithms

Popular rapid solvers are based on backtracking and dancing links. Rapid style Sudoku solvers are preferred for verifying large sets of randomly generated Sudoku puzzles due to their ease of implementation and quick results. Backtracking, a common technique in artificial intelligence, is a refined brute force search that explores a constrained set of possibilities until reaching a solution [7]. Donald Knuth suggested

Dancing Links (DLX) as an implementation of his Algorithm X which solves the exact cover problem [8].

Since Sudoku is a subset of the exact cover problem, DLX provides a quick and efficient Sudoku solver. Each space is treated as a vertex on the Sudoku grid, yielding 81 vertices. The driving force behind DLX is the concept that nodes in a circular, doubly-linked list can be quickly inserted or removed. DLX operates on a matrix of possible decisions where each column and row is a circular, doubly-linked list. The way DLX inserts and removes nodes from the matrix was the inspiration for the name Dancing Links.

Human Techniques

Human style solving is generally used as a way to estimate the difficulty level of a puzzle. Human solvers commonly employ techniques such as cross-hatching, counting, and marking up [9]. Cross-hatching is a scanning technique that uses the process of elimination to identify lines where a number must be. Counting is the opposite of cross-hatching because it eliminates numbers that cannot be assigned to a cell. Marking up is used after applying cross-hatching and counting. A cell is marked up by listing all possible remaining valid numbers. After all cells have been marked up, they can be compared and analyzed.

6

COUGAAR

Introduction

COGnitive Agent Architecture (Cougaar) is a Java-based agent architecture [10]. Cougaar is capable of handling large-scale distributed applications and was developed as part of the solution to the DARPA UltraLog project, a distributed logistics application comprised of more than 1000 agents distributed over 100 physically dispersed hosts [11]. A primary goal of UltraLog was to maintain communication among all hosts even with 40% of the communication network disabled by asymetrical attacks. The resulting system is completely open-source. Due to the size and complexity of Cougaar, only the components used for this research will be discussed. For a more complete description, see the Cougaar Developer's Guide [12].

The Decision to Use Cougaar

The decision to investigate Cougaar, instead of another agent architecture, was based on the author's previous work with Cougaar. A grant by the Transportation Security Administration [13] to Rocky Mountain Agile Virtual Enterprises at Montana Tech [14] funded research and development of an airport security system distributed across multiple agents using Cougaar. The author worked on a part of the project devoted to door access control using distributed agents. A paper presented at the

Computational Intelligence conference in Calgary in 2005 provides details of the door access control project and how it was tested [15].

Alternative Agent Architectures

Although the Cougaar distributed agent architecture is the focus of this thesis, other alternative agent architectures are available. The Open Agent Architecture (OAA), developed by SRI International, primarily focuses on the concept of a delegated computing model where agents are assigned tasks by a controller agent based on their capabilities [16]. The Common Object Request Broker Architecture (CORBA), developed by OMG, is a widely used platform that provides the infrastructure and architecture needed to develop a distributed agent architecture [17]. Jini, an open source project supported by Sun Microsystems [18] and SolutionsIQ [19], is a service oriented architecture that can support intelligent agent systems [20]. The Reusable Environment for Task-Structured Intelligent Networked Agents (RETSINA), developed at Carnegie Mellon, is a research platform for studying agent architecture related problems such as reliable communication and agent reuse [21]. Finally, The Knowledgeable Agent-oriented System (KAoS), developed by Boeing, provides an agent architecture that attempts to address the common distributed agent architecture problem caused by a lack of semantics and lack of extensibility for agent communication languages [22].

The Cougaar Problem Domain

Cougaar is suited for use in a wide range of distributed or agent related problems. Any autonomous group of robots could utilize an agent architecture to provide software infrastructure for tasks such as communication and task delegation. The adaptive logistics required for inventory management dispersed across multiple physical locations would be another problem suited to an agent architecture. Other problems that could be explored with an agent architecture such as Cougaar are security systems, data mining, load balancing, and certificate authorities.

Cougaar Components

The Community

Cougaar supports the segmenting of agents into communities [23]. A community is composed of logically related agents. An agent can belong to an unlimited number of communities. It may join or leave a community as its goals change. From a communication standpoint, this function allows a single message to be addressed to an entire community without regard to a community's current membership.

The Node

A Node is a single Java Virtual Machine [24] that may contain and maintain multiple agents [12]. Generally, Nodes share a 1:1 correspondence with a CPU. For example, a node may contain all agents requiring access to a restricted resource. A node can be treated logically as a community restricted to a single CPU. A node is

similar to a community because an agent on one node can move to another. A node provides several services to its agents, such as message routing, logging, and resource access.

The Agent

Each Cougaar agent is composed of plugins, a blackboard, and support services (Figure 2). The plugins provide the behavior of an agent. They communicate with other plugins, agents, and communities using the blackboard. The blackboard supports the publish/subscribe semantic for object handling [25]. A plugin indicates the types of object it is interested in by registering a subscription with the blackboard. After an object is published to the blackboard, the agent checks all subscriptions and activates relevant plugins. The plugin can then retrieve the object and perform any tasks. Any plugin may publish objects to the blackboard. Plugins normally only activate when one of their subscriptions is triggered.

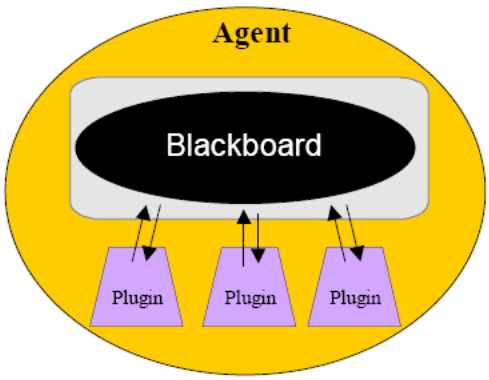

Figure 2. Internal Agent Structure.

Inter-Agent Communication

Agents communicate with each other using a built-in, asynchronous message-passing protocol called the Message Transport Service (MTS) [26]. The MTS provides an API for sending messages to agents by name. For solving Sudoku puzzles, messages take the form of a relay. A relay combines send and receive blackboard interfaces to share messages between agents. Source object data appears on the blackboard of the target agent and can return to the source agent's blackboard as a reply [27]. A white pages service, similar to DNS, performs address resolution for the agents [28]. Additional nodes can run the white pages service to provide redundancy.

THE AGENT APPROACH TO SUDOKU

The agent approach to Sudoku is based on dividing the workload evenly among many agents. One agent is assigned to each of the 81 cells on the board. A final agent, the board master, has the task of maintaining a complete picture of the board's state and makes any decisions that require a knowledge of each agent's status. Each cell agent is only allowed to communicate directly with the board master and the cell agents in its row, column, and three by three square. For example, cell agent A00 communicates with 20 other cell agents and the board master (Figure 3).

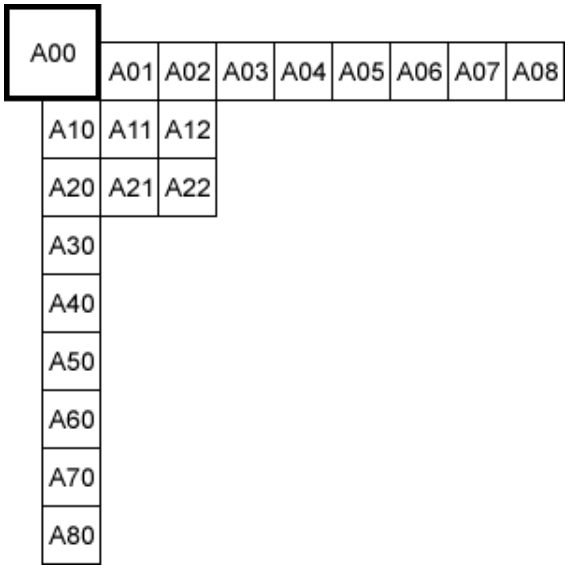

Figure 3. Communication Points for Agent A00.

Key Objects

There are three key objects that play an important role in how agents relate to each other and how decisions are made: the number, the status, and the command.

Number

The Number object is a pair of integers. The first represents a possible value on the Sudoku board, ranging from one to nine. The second is a guess ID that indicates the point at which the Number was removed as a candidate. An active Number is any Number object that has not been eliminated or uniquely selected as the choice for a particular cell. Active Numbers are designated by a guess ID of -1. For example, a cell might have the following un-eliminated choices: 1, 5, and 7. The active Number associated with 7 would be [7,-1]. If it were eliminated as a candidate for the cell after the first guess was made, it would become [7,1]. The purpose of storing the guess ID along with the candidate number is discussed in the Rollback subsection of The Board Master Agent section.

Status

The Status object represents all important external information about a cell agent to other cell agents and the board master. Depending on the situation, a Status object may contain any or all of the following elements:

- The cell agent's name. For example, the top-left cell's name is "A00" and the bottom right's is "A88."

- A list of the nine Number objects, reflecting their current status.
- The message ID. This is an increasing number that is never reset. The message ID is used to eliminate out-of-order messages, only the latest message is important.
- The reset ID. This is an increasing number that is never reset. After a puzzle is completed, every cell agent must be reset to its starting configuration for the next puzzle. A cell agent with the wrong Reset ID is clearly still referencing an old puzzle and can be corrected by the board master.
- The guess ID. This is a varying number that resets on completion of a puzzle. The guess ID indicates the cell agent's understanding of the current guess level. This is used in combination with the command ID to ensure that cell agents are following the guess and roll-back sequence correctly.
- The command ID. This is an increasing number that resets on completion of a puzzle. Since the guess ID can increase or decrease, the command ID is used to guarantee that the cell agent has rolled back appropriately before analyzing data from the current guess ID.
- The change counter. This is used internally by the board master to count the number of consecutive times a cell agent has sent the same Status.
- Neighbor completeness. This is set by the cell agent to indicate that it has received Status objects from all of its neighbors. The board master will not signal for solving to begin until all cell agents report this as true.

A Status contains almost all relevant data for a cell agent and is capable of producing guesses based on the currently active Numbers. Therefore when multiple guesses are made for the same cell, it will not produce the same guess twice.

Command

The Command object is the only message passed from the board master to the cell agents. The board master uses five types of commands:

- Startup: Before a cell agent can begin its part in solving a Sudoku puzzle, the board master must tell it whether has a starting value or not.
- Reset: After a puzzle is complete, every cell agent is sent the same reset command. The command contains an ID to be stored by the cell agents.
- Guess: Whenever the board master makes a guess, it sends this command to the chosen cell agent, along with the guessed value for that cell agent and an increased guess ID.
- Rollback: If the puzzle enters an invalid state, the board master instructs all cell agents to rollback to the guess ID provided.
- Report: The board master instructs the cell agent to respond with its current status.

The Cell Agent

For a 9x9 Sudoku grid, there are 81 total cell agents, one for each space on the Sudoku board. Each is only aware of the neighbors in its row, column, and three by

three square. The cell agents are completely unaware of the overall board status. A cell agent's only goal is to reduce its set of active Numbers to one. As a result, the system exhibits an emergent behavior by solving the Sudoku puzzle. In this case, the emergent behavior is known formally as a weak emergence because the new property (the completion of a Sudoku puzzle) comes directly from the interaction of individual elements (the cell agents) [29]. Arrangement of the cell agent within Cougaar is shown in Figure 4.

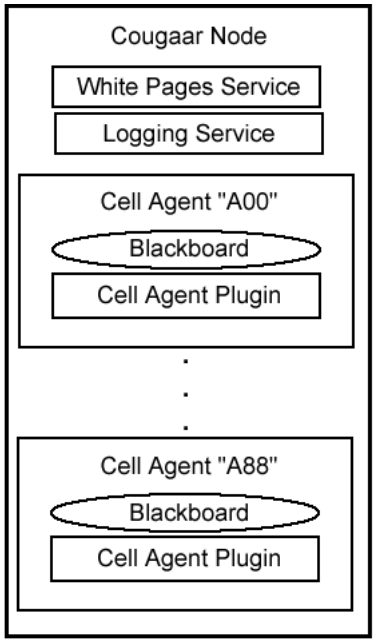

Figure 4. Internal Structure of a Cell Agent.

Starting

Before starting work on a Sudoku puzzle, all cell agents need to be communicating successfully with their neighbors. Each cell agent is explicitly aware of other cell agents in its row, column, and three by three square. Communication lists are composed at startup based on the cell agent's name. Each cell agent sends a message to the other cell agents on its lists. For efficiency, cell agents appearing on more than one list only receive one message. A cell agent has two opportunities to receive a response. First, when the target cell agent starts and sends messages of its own, and second, when it responds to a request. To prevent endless respond-respond looping, each cell agent keeps a list of other cells it knows to be responsive. When the list contains all cells a cell agent needs to communicate with, it sends a Status message to the board master indicating it is ready to begin. At this stage, all cells consider 1-9 to be valid choices.

Solving

Work on the puzzle begins when the board master sends replies to the first round of Status messages. If this response indicates the cell agent is constrained to a specific number, the cell agent sends the updated Status to its neighbors (row, column, and three by three square). As other cell agents receive these messages, they deactivate the appropriate Number in their local Status. The cell agent then sends the updated Status to all its neighbors. These two types of Status messages (one with a single active Number, one with many) allows for two types of analysis. The first is the

elimination of all choices but one for a specific cell. The second type is when the cell agent compares the local Status to the received Status from its neighbors. By process of elimination, a cell agent may constrain itself to a single active Number if no other cell agents in its row, column, or three by three square show that Number as active.

Rolling Back

When a cell agent reports to the board master with a duplicate value already contained in its row, column, or three by three square, the entire puzzle has entered an invalid state. The board master issues a rollback to restore the board to a state where a solution is still possible. Also, if a cell agent reports that it has eliminated all candidate Numbers as valid choices, a rollback is needed. The board master triggers a rollback by sending a rollback command to all cell agents. Each cell agent then reverts itself to the state before the most recent guess was made. Describing how guesses and rollbacks are handled at the cell agent level is best shown by example.

A cell agent's Number list, before startup:

[1, -1] [2, -1] [3, -1] [4, -1] [5, -1] [6, -1] [7, -1] [8, -1] [9, -1]

A constrained cell agent's Number list (constrained to 5), after startup:

[1, 0] [2, 0] [3, 0] [4, 0] **[5, -1]** [6, 0] [7, 0] [8, 0] [9, 0]

A cell agent's Number list with 3, 5, and 6 as candidates, before the board master makes a guess:

[1, 0] [2, 0] **[3, -1]** [4, 0] **[5, -1]** **[6, -1]** [7, 0] [8, 0] [9, 0]

The same cell agent's Number list, after the board master makes a guess (ID 1) which sets a neighbor to 3:

[1, 0] [2, 0] [3, 1] [4, 0] **[5, -1]** **[6, -1]** [7, 0] [8, 0] [9, 0]

The same cell agent's Number list, after the board master makes a guess (ID 2) which sets a neighbor to 5:

[1, 0] [2, 0] [3, 1] [4, 0] [5, 2] **[6, -1]** [7, 0] [8, 0] [9, 0]

Now the cell agent is constrained to 6. If the board master issues a rollback for guess ID 2, the cell agent reverts the Number list by reactivating any Number with a guess ID of 2 or higher. The board master can issue multiple simultaneous rollbacks by using a lower guess ID. The same cell agent's Number list, after rollback with guess ID 2:

[1, 0] [2, 0] [3, 1] [4, 0] **[5, -1]** **[6, -1]** [7, 0] [8, 0] [9, 0]

Cell agents constrained before a given guess ID have no rollback work to do.

The Board Master Agent

The board master agent is responsible for keeping the system synchronized, handing out starting and guess values, rolling back guesses, and tracking the overall state of the board. The board master agent only receives Status object messages from the cell agents, but the state of the board determines how the messages will be used. Arrangement of the board master agent within Cougaar is shown in Figure 5.

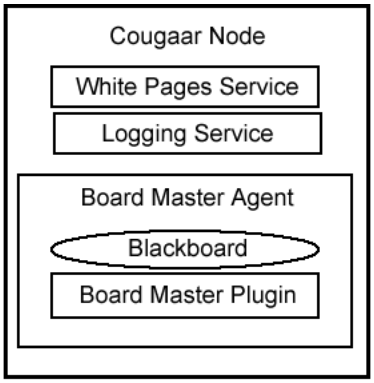

Figure 5. Internal Structure of the Board Master Agent.

Starting

The board master creates a list of cell agents as they request their startup values in order to ensure that all cell agents are online. When the list reaches 81 (for a nine by nine Sudoku board), the board master agent sends all starting values to the cell agents. If a cell agent is supposed to be constrained to a single starting value, but reports a list of several values, the board master agent sends the startup message again.

Solving

As cell agents become constrained, they send status messages to the board master that reflect their constraints. The board master keeps a 2D array of all cell agents' status. A timer periodically requires all cell agents to report to the board master with their Status. The timer only triggers if there have been no Status reports for a period of time, set at the start of the system. The slowest CPU in the system determines the

necessary wait time, to allow for late coming messages. If the cell agents are reporting updates to the board master, the timer will not trigger to avoid unnecessary traffic.

Detecting a Stalled Puzzle

If all cell agents are responding with unchanging Status, none are able to make further restrictions to their number lists, and the board is considered stalled. When the board has stalled, the board master starts guessing.

Guessing

The board master uses several methods to choose which cell agent should make a guess.

1. Random Guess from a Random Agent: The board master randomly selects a cell agent from the set of all cell agents with any active Numbers. This is the most basic Sudoku problem solving method.

2. Random Guess from a Random Best Agent: The board master randomly selects a cell agent from the set of cell agents with the lowest amount of active Numbers.

3. Random Guess from a Random Worst Agent: The board master randomly selects a cell agent from the set of cell agents with the highest amount of active Numbers.

Once the board master has selected a cell agent to make a guess, the cell agent's Status is added to a guess stack. The size of the stack is the board's guess ID. Each

Status provides a method to return possible guesses until none remain. Once the board master has exhausted all guesses for a cell agent without reaching a completed puzzle state, the cell agent's Status is removed from the guess stack. The next Status on the stack will provide the next guess. Whenever a guess leads to an invalid board state, the board master issues a rollback that reverts the board to its state before the current guess.

Rolling Back

All cell agents perform rollbacks if needed and respond with a Status that reflects the new guess ID. After all cell agents have reported back, the board master selects the next guess from the Status on top of the guess stack. If no more guesses remain for a given Status, it issues another rollback before moving to the next Status on the stack. Once the board master finds a suitable guess, it sends the guess to the corresponding cell agent and solving resumes. By process of elimination, the first chosen cell agent must yield a guess that leads to a correct solution. If the board master exhausts all guesses for this cell agent, the Sudoku puzzle has no solution.

Completion

Once the system has solved the board correctly, all cell agents become dormant. If the board master is loaded with another puzzle, it sends a reset command to the cell agents, and the system returns to the Startup stage.

Illustrative Example

Although all testing and design is centered around the standard nine by nine Sudoku puzzle, it is more insightful to use a smaller four by four Sudoku puzzle as an example. The guessing method used will be Random Guess from a Random Agent. Throughout this example, italics indicate changes between previous and current figures. At startup, A11 and A30 are constrained to 2. A23 is constrained to 1. All cell agents not provided an initial constraint have all active Numbers, as shown in Figure 6.

1234	1234	1234	1234
1234	2	1234	1234
1234	1234	1234	1
2	1234	1234	1234

Figure 6. Example Sudoku Puzzle: Cell Agents at Startup.

Immediately following startup, non-constrained cell agents determine which of their active Numbers can be deactivated. 7 shows this change in all active cell agents except A02.

Now that all cell agents have an accurate representation of their neighbors, analysis can occur. For this, each cell agent scans its neighbor's active Numbers against its own. If a Number is active for the current cell agent, but not for any of that cell

134	134	1234	234
134	2	134	34
34	34	234	1
2	134	34	34

Figure 7. Example Sudoku Puzzle: Cell Agents After Round One.

agents neighbors in a row, column, or two by two square, the current cell agent is the only remaining candidate. For this example, A23 uses this process to constrain itself to 2 (Figure 8).

134	34	134	234
134	2	134	34
34	34	2	1
2	1	34	34

Figure 8. Example Sudoku Puzzle: Cell Agents After Round Two.

A second round of messages and analysis yields one more constrained cell agent, A03 to 2 (Figure 9).

At this point, no further work can be done by the cell agents with their current analysis tools. The puzzle will not move closer to a solution without a guess from the

134	34	134	**2**
134	2	134	34
34	34	2	1
2	1	34	34

Figure 9. Example Sudoku Puzzle: Cell Agents After Round Three.

board master. Selecting a random Number from a random agent, the board master chooses 3 from agent A10 (Figure 10).

14	***4***	134	2
****3***	2	***14***	***4***
4	34	2	1
2	1	34	34

Figure 10. Example Sudoku Puzzle: Cell Agents After Guess.

After a final round of messages, every cell agent has constrained itself to a single active Number, and the Sudoku puzzle has been successfully solved (Figure 11).

1	4	***3***	2
*3	2	***1***	4
4	***3***	2	1
2	1	***4***	***3***

Figure 11. Example Sudoku Puzzle: Cell Agents At Completion.

EMPIRICAL ANALYSIS

Methodology

Metrics

Because it is difficult to measure space in a distributed agent-based architecture, only time and guesses were considered in this study. For backtracking, the raw time to solve the problem was measured, and each recursive call was treated as a guess. For DLX, only the measurement of time could be accurately compared to the other Sudoku solving techniques. Work done by DLX was measured by recording the removal or addition of links to the matrix. For consistency, each of these changes to the matrix was referred to as a guess.

For the Cougaar approach, measurements included the raw time to solve the problem, the number of messages passed, and the number of guesses made. Although message counts cannot be effectively compared to backtracking or DLX, they provide a useful metric for comparing Sudoku problem solving strategies within the Cougaar approach.

Each Sudoku solver had 10 recorded attempts at every puzzle. On each puzzle, averages and standard deviations of the metrics were recorded. For backtracking and DLX, this applied to time and guesses. For the Cougaar Sudoku system, this applied to time, guesses, and messages.

Test Sets

Two test sets were used to measure the performance of the Agent-based Sudoku solver. The first consisted of verified Sudoku puzzles, and the second consisted of randomly generated, unverified puzzles. A verified Sudoku puzzle has been shown to only have one solution. All of the verified Sudoku puzzles used for testing were taken from WebSudoku.com. Ten puzzles were taken from each of the available difficulty levels, Easy, Medium, Hard, and Evil, yielding 40 total puzzles for testing. For reference, the difficulty levels are classified by WebSudoku.com as follows:

> "The difficulty of a puzzle is related to the depth of thinking required. An easy Sudoku can be solved using more simple logic, whereas an evil puzzle needs deeper analysis. Obviously there is also variation between different puzzles, so even within a level some puzzles may seem harder than others." [1]

A Sudoku puzzle was selected at random to provide the base puzzle for generating additional puzzles. The Sudoku puzzle starts in a completed state and cells are blanked at random in groups of nine. This is repeated, saving the resulting Sudoku puzzle at each step, until the entire Sudoku puzzle is blank. The entire reduction from a completed Sudoku puzzle to a blank one was repeated four times to yield a total of forty puzzles. These randomly reduced Sudoku puzzles will be referred to by the number of blanks they contain. For example, the 0 puzzle contains no blank cells and the 81 puzzle is completely blank.

Cougaar Environment

One of Cougaar's key features is the ability to connect agents regardless of their location on a network. During development, the agent system was tested on networks of up to five desktop computers. For the experiments, 82 agents ran on a single workstation. The GUI proved an effective tool for developing, testing, and debugging the agents on single boards but was impractical for any large scale analysis. To enable repetitive batch-processing for large numbers of puzzles with varying decision making constraints, a separate board master was developed. At the Cougaar level, the second board master is identical to the first. In the place of a GUI, all commands came from the command line and all results were logged in separate files according to puzzle number, date, and time. Finally, several scripts were used to compile the results into meaningful formats.

Experiments on Verified Sudoku Puzzles

Each of the Sudoku puzzles used in this section were verified by WebSudoku.com to only have one solution. During the testing of the Cougaar agent approach, one or more Cougaar agents occasionally became unresponsive. This seemingly happens at random and appears to be a messaging break down within Cougaar while under high messaging loads. Each of the times this occurred, testing had to be restarted manually.

Backtracking

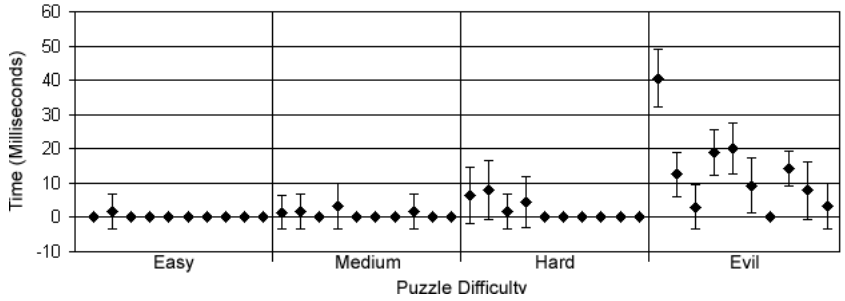

Figure 12. Experiment One Time Results for Backtracking.

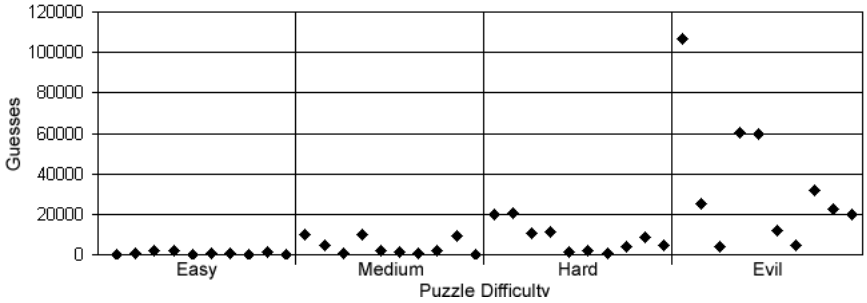

Figure 13. Experiment One Guess Results for Backtracking.

Table 1. Backtracking Experiment One.

	Time		Guesses	
	Average	σ	Average	σ
Easy	0.16	0.51	816.70	837.78
Medium	0.79	1.12	4027.90	4067.09
Hard	2.06	3.10	8320.50	7308.03
Evil	12.99	11.79	34694.30	32137.32

The backtracking solver successfully found a solution for all of the Sudoku puzzles in this test set. For all but the Evil puzzles, backtracking was able to solve the Sudoku puzzles within 10 milliseconds using a low number of guesses. For most of the puzzles, backtracking reached a solution in an amount of time that was too small to be measured by the testing computer. Figure 12 and Figure 13 show a strong correlation between time to solve and guesses made. This is especially apparent in the results of Evil 1: 40 milliseconds and over 100,000 guesses. This correlation between times and guesses exists for all of the solving methods tested.

DLX

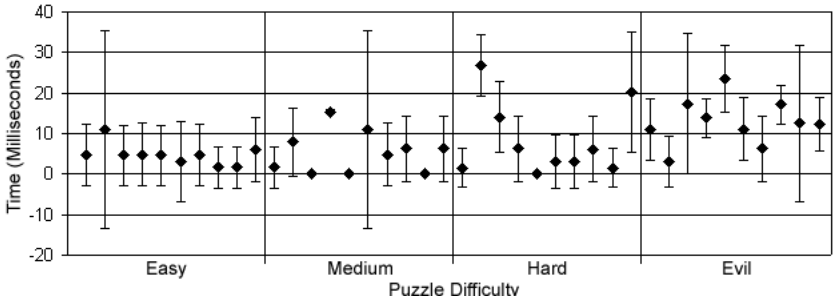

Figure 14. Experiment One Time Results for DLX.

The DLX solver successfully found a solution for all of the Sudoku puzzles in this test set. DLX took longer than backtracking in almost all cases. A notable exception is DLX's overall time on the Evil puzzles (Table 2), which was very close to backtracking (Table 1). This is likely attributable to DLX being a more complicated algorithm than straightforward backtracking. For the easier Sudoku puzzles, this

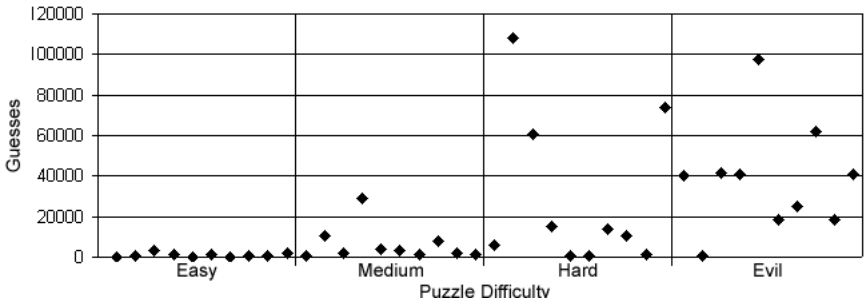

Figure 15. Experiment One Guess Results for DLX.

Table 2. DLX Experiment One.

	Time		Guesses	
	Average	σ	Average	σ
Easy	4.67	2.63	949.50	998.27
Medium	5.32	5.17	6163.30	8679.90
Hard	8.27	9.05	29154.10	38082.75
Evil	12.77	5.75	38513.40	26903.39

slowed DLX down when compared to backtracking. On the more difficult Evil puzzles, the same complexity allowed it to quickly solve the puzzles. Backtracking, on the other hand, experienced a significant time and guess increase when compared to the other three categories of puzzle. Note that although Figure 14 scales into negative time, negative values are only shown for the sake of graphing the standard deviation. This occurs throughout the results and should be ignored.

Random Choice, Random Worst Agent

Random Choice with Random Worst Agent (Random Worst) is the first of the Cougaar Sudoku problem solving methods. Random Worst experienced a messaging

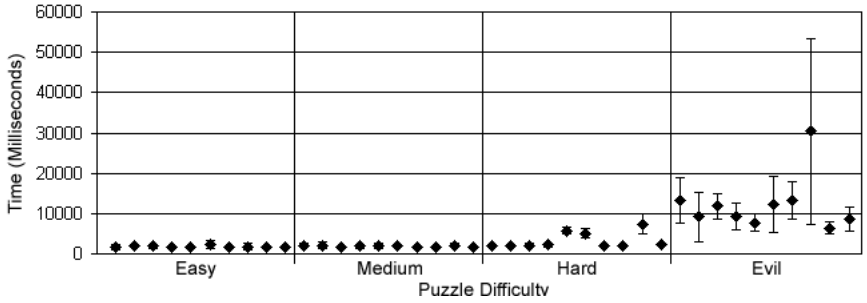

Figure 16. Experiment One Time Results for Random Worst Agent.

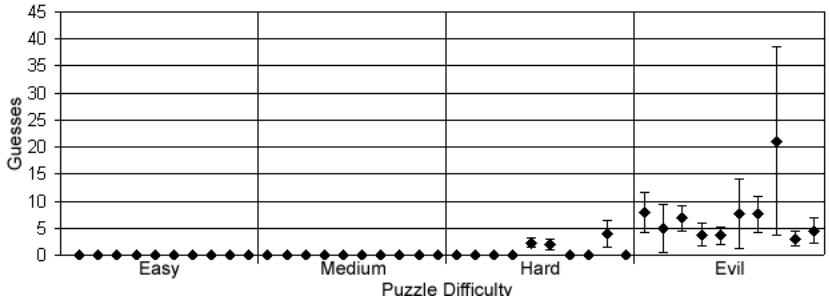

Figure 17. Experiment One Guess Results for Random Worst Agent.

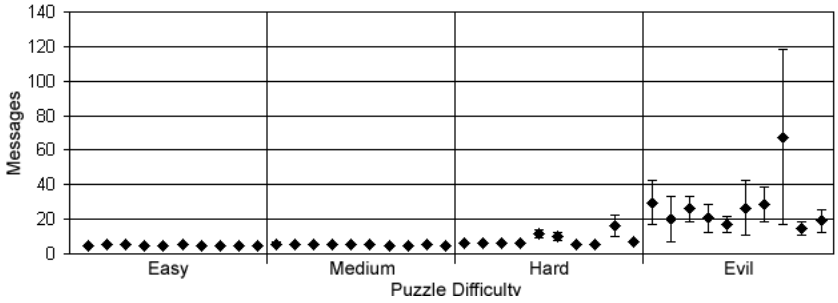

Figure 18. Experiment One Message Results for Random Worst Agent.

Table 3. Random Worst Agent Experiment One.

	Time		Guesses		Messages	
	Average	σ	Average	σ	Average	σ
Easy	1753.34	236.50	0.00	0.00	4.75	0.33
Medium	1891.06	186.36	0.00	0.00	5.29	0.31
Hard	3296.10	1987.08	0.81	1.40	8.00	3.60
Evil	12221.46	6806.97	7.09	5.26	26.90	15.17

failure on four occasions: three Evil boards and one Medium board. Easy and Medium required no guessing, so interesting results do not occur until the second half of Hard and all of Evil. In sparsely populated initial Sudoku puzzles, Random Worst is very likely to make an incorrect guess early on. Figure 17 shows that Evil 8 was particularly difficult for this Sudoku problem solving method. The large deviation from average indicates that some attempts made good guesses, while others made very bad ones. Backtracking and DLX greatly outperformed Random Worst with regard to time, but Random Worst used fewer guesses in all cases. This is attributable to the analysis the Agent Method performs at each step. It is interesting to note that the most difficult puzzle for Random worst (Evil 8) was different from the most difficult puzzle for backtracking (Evil 1) and DLX (Evil 5). Clearly, subtleties in these puzzles favor one solving method over another. Finding the specific reasons for this is mentioned in the Future Work chapter.

Random Choice, Random Agent

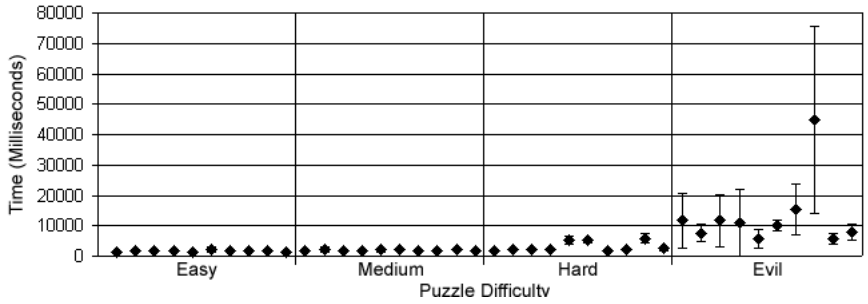

Figure 19. Experiment One Time Results for Random Agent.

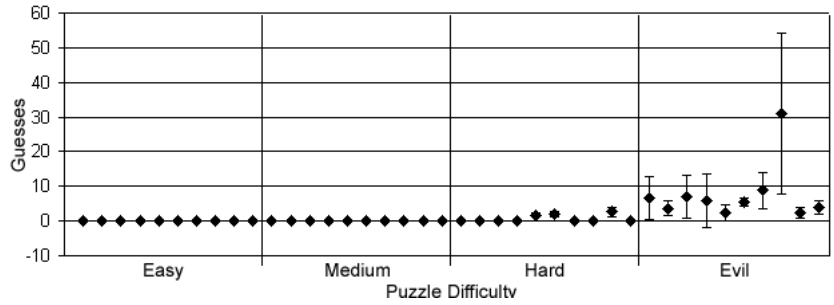

Figure 20. Experiment One Guess Results for Random Agent.

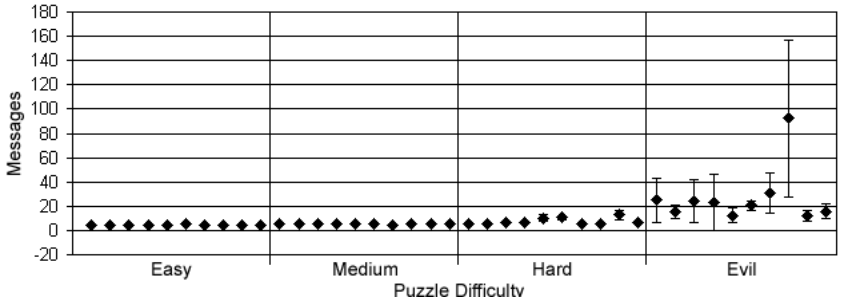

Figure 21. Experiment One Message Results for Random Agent.

Table 4. Random Agent Experiment One.

	Time		Guesses		Messages	
	Average	σ	Average	σ	Average	σ
Easy	1603.27	241.87	0.00	0.00	4.48	0.32
Medium	1862.89	192.81	0.00	0.00	5.16	0.28
Hard	3102.60	1627.65	0.63	1.04	7.44	2.75
Evil	13156.62	11521.59	7.65	8.43	27.16	23.73

Random Choice with Random Agent (Random) failed to solve the Sudoku puzzle on two Evil boards. Random performed very similarly to Random Worst. Of particular interest is Random's average number of guesses (Figure 20) compared to Random Worst's average number of guesses (Figure 17) on the third-to-last Evil board. The difference of 10 guesses shows that in some cases, guessing from a smaller set of agents (Random Worst) could reach a solution more quickly than guessing from a larger set of agents. Compared to backtracking, Random's performance is very similar to Random Worst's.

Random Choice, Random Best Agent

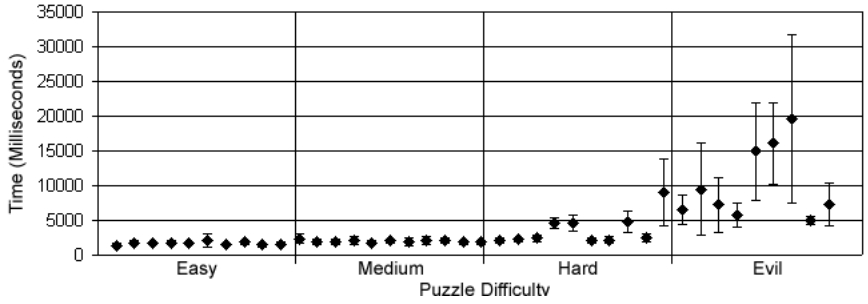

Figure 22. Experiment One Time Results for Random Best Agent.

Random Choice with Random Best Agent (Random Best) failed to solve the Sudoku puzzle on two occasions: one Evil board and one Hard board. Random Best predictably shows the best overall performance of the three Cougaar Sudoku problem solving methods. It consistently outperforms the other two Cougaar Sudoku problem solving methods for every metric. On Evil 8 puzzle, Random Best makes the fewest

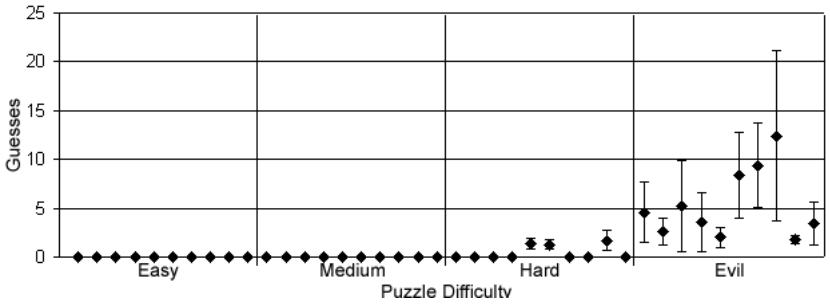

Figure 23. Experiment One Guess Results for Random Best Agent.

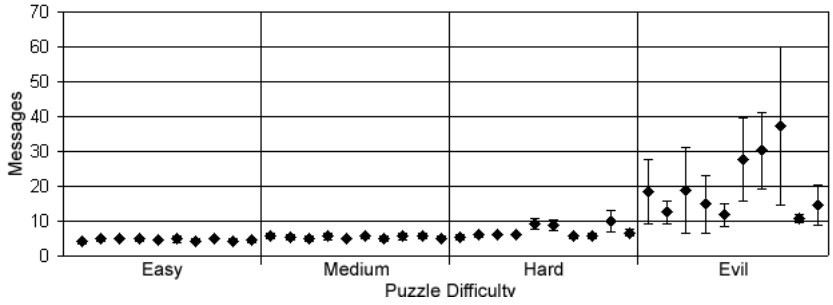

Figure 24. Experiment One Message Results for Random Best Agent.

Table 5. Random Best Agent Experiment One.

	Time		Guesses		Messages	
	Average	σ	Average	σ	Average	σ
Easy	1703.64	223.13	0.00	0.00	4.65	0.30
Medium	2009.71	177.80	0.00	0.00	5.42	0.30
Hard	2955.28	1184.41	0.44	0.72	7.01	1.71
Evil	10090.55	5006.66	5.34	3.56	19.77	9.04

guesses. This hierarchy (Random Best, Random Worst, Random) indicates that the Cougaar Sudoku problem solving methods might perform better with smaller sets of agents to guess from. Compared to backtracking, Random Bests's performance is very similar to the other two Cougaar Sudoku problem solving methods.

Conclusions

Backtracking and DLX were substantially quicker than any of the Cougaar Sudoku problem solving methods. All of the Cougaar Sudoku problem solving methods required fewer guesses to reach a solution than backtracking. Random and Random Worst performed very similarly. Random Best was predictably the clear winner among the Cougaar methods.

Experiments on Random Sudoku Puzzles

In the previous section, the verified Sudoku puzzles were all known to have only one solution. In this section, the Sudoku puzzles might have one or more solutions. For example, an initially empty nine by nine Sudoku puzzle has a very large number of solutions. Due to the large number of unconstrained cells in some of these Sudoku puzzles, it is possible for the random Sudoku problem solving methods to take a very long time. A time limit of five minutes was set for each solver to reach a solution before moving on to the next Sudoku puzzle. For each of the Cougaar Sudoku problem solving types tested, this time limit failure is mentioned along with any failures in

message communication. The full set of puzzles in this section will be referred to by the number of blanks they contain: 0, 9, 18, 27, 36, 45, 54, 63, 72, and 81.

Backtracking

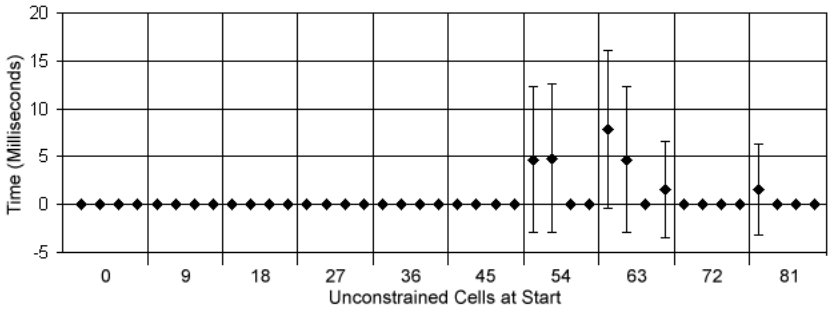

Figure 25. Experiment Two Time Results for Backtracking.

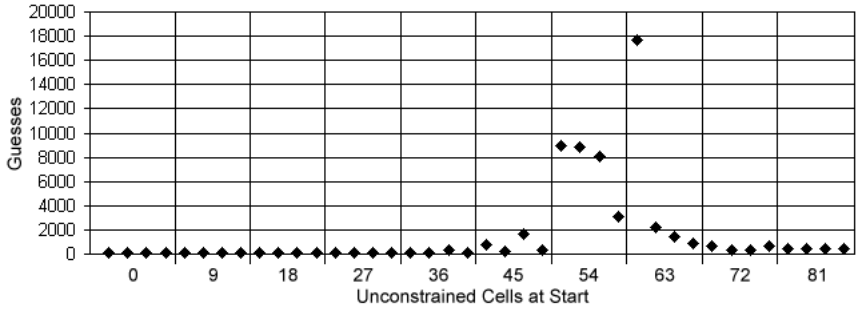

Figure 26. Experiment Two Guess Results for Backtracking.

The backtracking solver successfully found a solution for all of the Sudoku puzzles in this test set. Figure 25 and Figure 26 show that none of the randomly-reduced Sudoku puzzles posed as much solving difficulty for backtracking as did the Easy/Medium/Hard/Evil Sudoku puzzles (Figure 12 and Figure 13). In Table 6, the

Table 6. Backtracking Experiment Two.

	Time		Guesses	
	Average	σ	Average	σ
0	0.00	0.00	81.00	0.00
9	0.00	0.00	81.00	0.00
18	0.00	0.00	82.25	1.89
27	0.00	0.00	93.00	15.77
36	0.00	0.00	159.75	103.03
45	0.00	0.00	744.50	682.69
54	2.38	2.74	7260.50	2775.53
63	3.52	3.45	5523.75	8100.84
72	0.00	0.00	506.25	191.37
81	0.38	0.75	391.00	0.00

0 puzzle results show the minimum number of recursive calls backtracking performs on an already-completed Sudoku puzzle. It is interesting that only the 45, 54, and 63 puzzles caused backtracking to do any work. These results indicate that backtracking is extremely effective at solving over-constrained and under-constrained Sudoku puzzles.

DLX

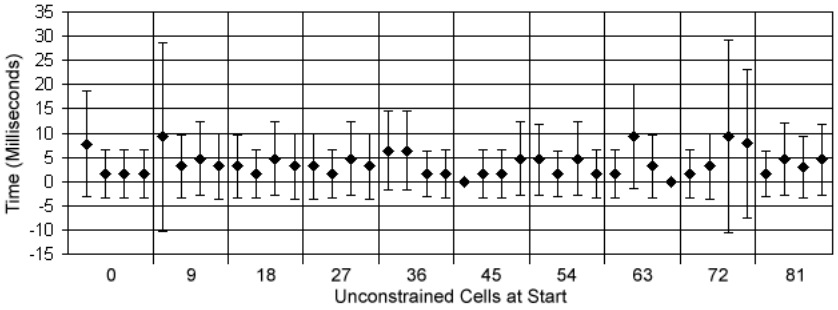

Figure 27. Experiment Two Time Results for DLX.

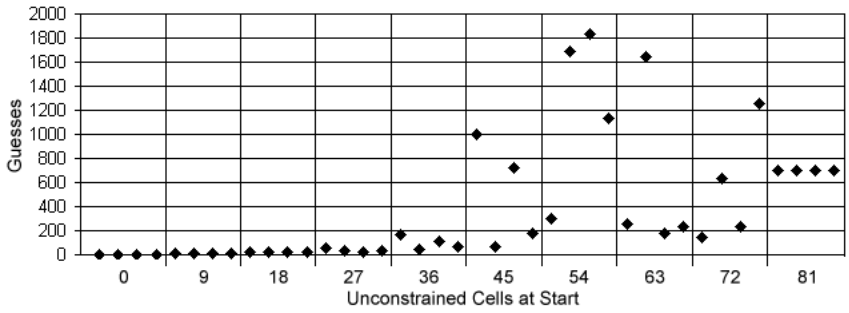

Figure 28. Experiment Two Guess Results for DLX.

Table 7. DLX Experiment Two.

	Time		Guesses	
	Average	σ	Average	σ
0	3.12	3.05	0.00	0.00
9	5.05	2.86	9.00	0.00
18	3.15	1.27	19.00	1.15
27	3.17	1.27	34.50	11.12
36	3.92	2.74	95.00	55.73
45	1.98	1.97	492.00	445.52
54	3.07	1.76	1240.50	696.27
63	3.47	4.02	576.50	712.93
72	5.50	3.70	565.00	507.24
81	3.40	1.46	701.00	0.00

The DLX solver successfully found a solution for all of the Sudoku puzzles in this test set. DLX took longer to solve each puzzle (Figure 27) than backtracking (Figure 25). In Table 7, the 0 puzzle results show that DLX performs some analysis of a puzzle before doing any work. Puzzles 0, 9, 18, and 27 on Table 7 show that DLX is capable of making about one change to the matrix (guess) per blank on the board, when the choices are from a sufficiently constrained set.

Random Choice, Random Worst Agent

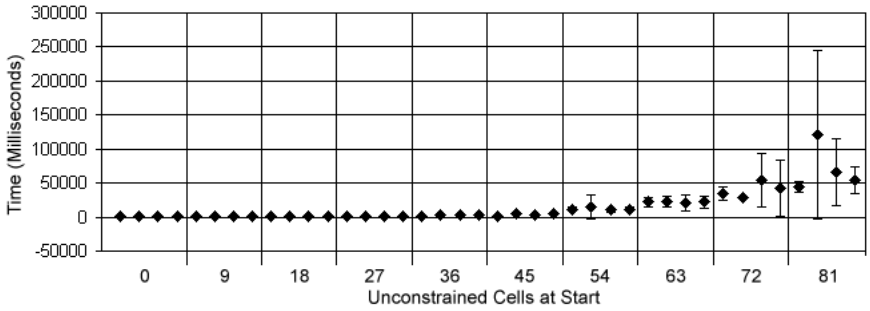

Figure 29. Experiment Two Time Results for Random Worst Agent.

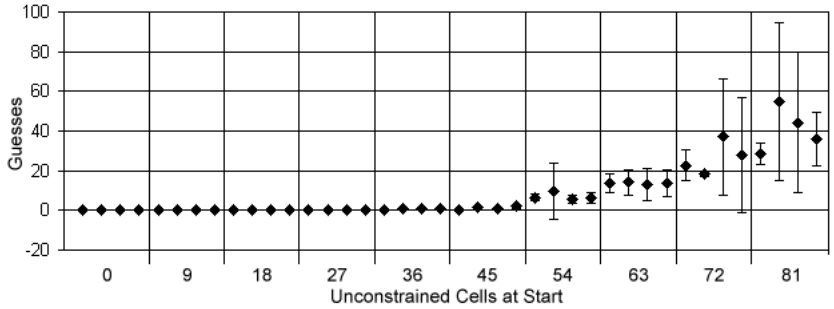

Figure 30. Experiment Two Guess Results for Random Worst Agent.

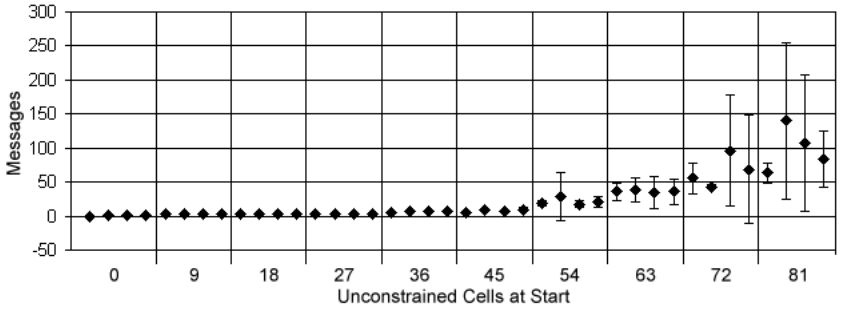

Figure 31. Experiment Two Message Results for Random Worst Agent.

Table 8. Random Worst Agent Experiment Two.

	Time		Guesses		Messages	
	Average	σ	Average	σ	Average	σ
0	354.72	143.03	0.00	0.00	0.51	0.33
9	1081.80	52.67	0.00	0.00	3.16	0.09
18	1148.40	204.42	0.00	0.00	2.98	0.31
27	1351.92	120.73	0.00	0.00	3.73	0.20
36	2981.22	997.17	0.75	0.50	5.82	1.10
45	3853.52	1449.05	1.10	0.84	7.51	2.11
54	12155.10	2051.22	7.08	1.68	21.71	4.63
63	21971.80	610.34	13.60	0.49	36.34	1.60
72	40208.98	10824.08	26.49	8.03	65.88	22.69
81	71155.77	34313.10	40.96	11.32	98.74	32.79

Random Worst failed to solve the Sudoku puzzle on seven occasions: twice for 9, three times for 72, and twice for 81. Additionally, 24 attempts were halted after reaching the time limit. For each of these failures, the puzzles were from the 63, 72, and 81 sets. Unlike backtracking, the number of guesses made by Random Worst (Figure 30) did not peak at the 53 and 63 puzzles but increased steadily throughout. The exact reason for this is unknown, but the steady increase is a common theme throughout the Cougaar random Sudoku problem solving techniques.

Random Choice, Random Agent

Random failed to solve the Sudoku puzzle on four occasions. Additionally, nine attempts were halted after reaching the time limit. For each of these failures, puzzles were from the 63, 72, and 81 sets. The overall performance of Random (Table 9) was slightly better than Random Worst (Table 8) for a majority of sets. This is possibly

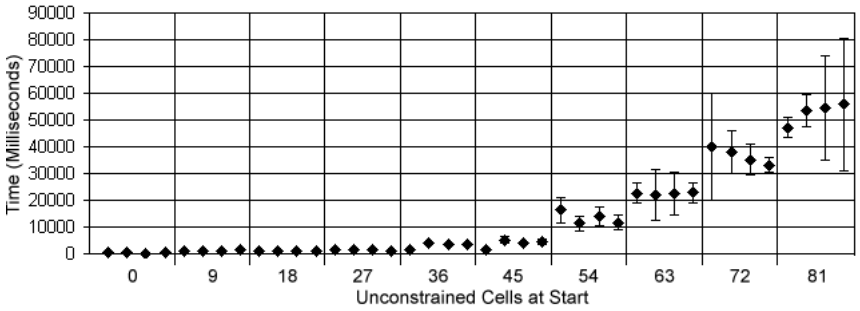

Figure 32. Experiment Two Time Results for Random Agent.

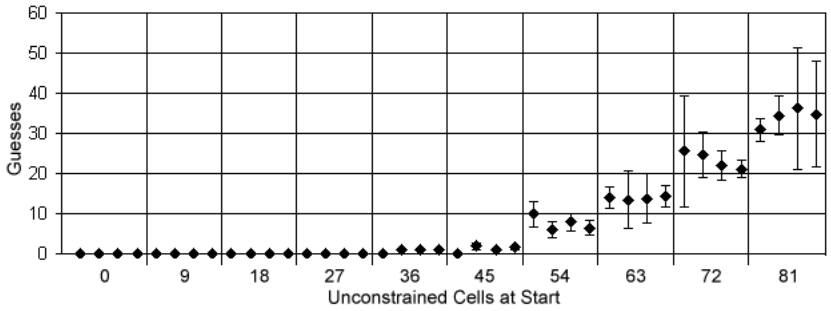

Figure 33. Experiment Two Guess Results for Random Agent.

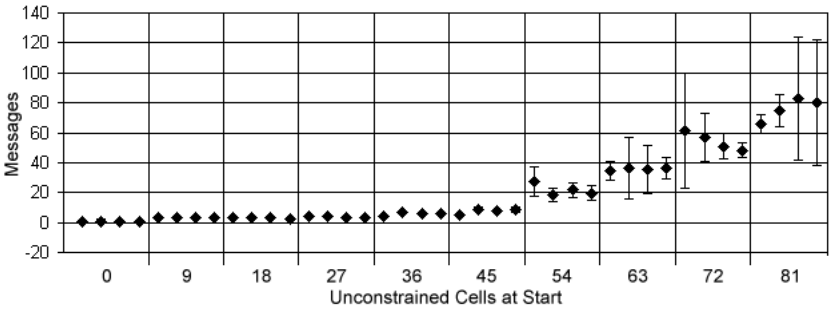

Figure 34. Experiment Two Message Results for Random Agent.

Table 9. Random Agent Experiment Two.

	Time		Guesses		Messages	
	Average	σ	Average	σ	Average	σ
0	304.12	102.64	0.00	0.00	0.59	0.33
9	1064.60	133.90	0.00	0.00	3.12	0.14
18	1082.86	91.45	0.00	0.00	2.91	0.11
27	1307.31	164.90	0.00	0.00	3.72	0.22
36	3032.95	1104.55	0.75	0.50	5.78	1.27
45	3803.89	1555.45	1.12	0.83	7.35	1.87
54	13299.98	2327.06	7.60	1.71	21.71	3.83
63	22461.45	367.69	13.93	0.35	35.75	0.80
72	36548.60	3036.95	23.39	2.12	54.41	5.97
81	52775.97	3917.06	34.11	2.27	76.10	7.33

caused by Random's access to slightly better agents to guess from than Random Worst.

Random Choice, Random Best Agent

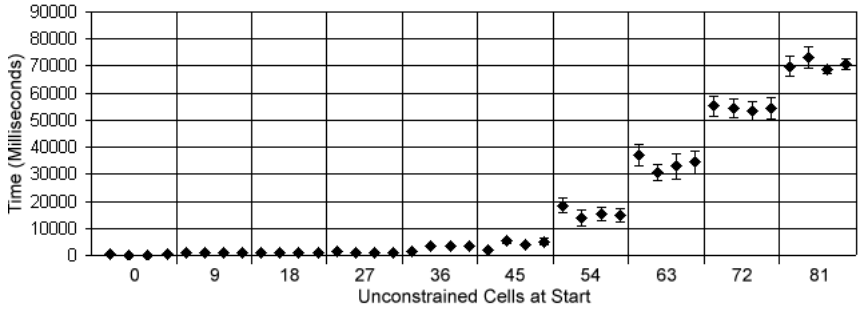

Figure 35. Experiment Two Time Results for Random Best Agent.

Random Best failed to solve the Sudoku puzzle on four occasions. No attempts were halted due to reaching the time limit. Random Best successfully solved each of the Sudoku puzzles in this test set within 80 seconds. Random Best appears to

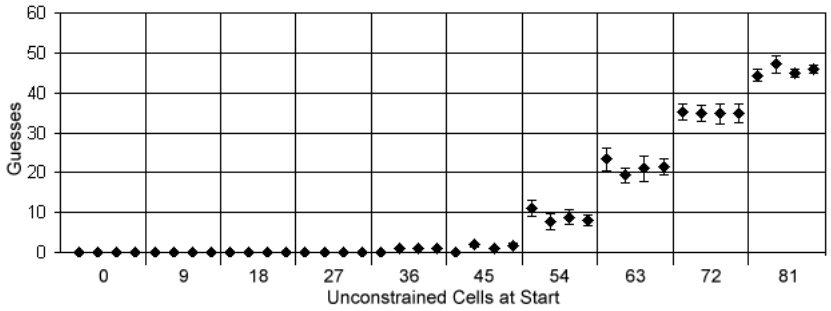

Figure 36. Experiment Two Guess Results for Random Best Agent.

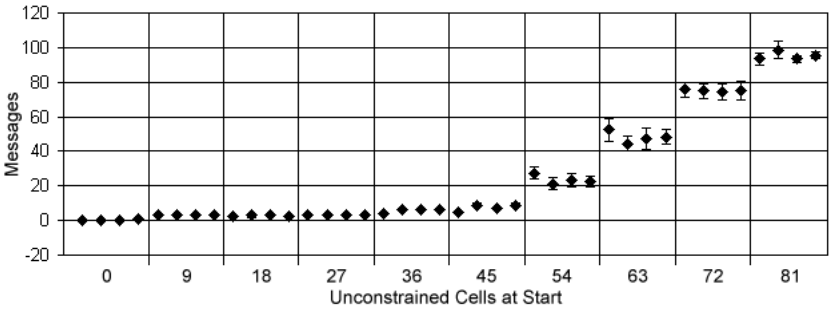

Figure 37. Experiment Two Message Results for Random Best Agent.

Table 10. Random Best Agent Experiment Two.

	Time		Guesses		Messages	
	Average	σ	Average	σ	Average	σ
0	263.83	79.57	0.00	0.00	0.35	0.18
9	1020.47	23.97	0.00	0.00	2.96	0.08
18	1020.03	79.94	0.00	0.00	2.79	0.09
27	1207.97	92.48	0.00	0.00	3.47	0.12
36	2924.35	874.62	0.75	0.50	5.69	1.07
45	3991.12	1594.27	1.15	0.86	7.45	1.92
54	15594.60	2055.84	8.90	1.47	23.61	2.67
63	33836.05	2551.76	21.25	1.64	48.10	3.35
72	54349.30	729.20	34.95	0.21	74.99	0.67
81	70580.57	1807.07	45.60	1.20	95.33	2.40

perform more poorly (Table 10) than Random Worst (Table 8) and Random (Table 9), but these tables to not take into consideration attempts prematurely ended by reaching the time limit.

Table 11. Time Limit and Communication Failures.

	Time Limited Failures	Communication Failures
Random Worst	24	7
Random	9	4
Random Best	0	4

Table 11 shows that Random Best has a better overall performance than Random Worst or Random, once time-limited failures are taken into consideration. In the set of solutions excluding time-limited failures, Table 8 shows that selecting guesses from the set of least-constrained cell agents tends to lead to solutions with fewer guesses.

Conclusions

As with the Easy/Medium/Hard/Evil Sudoku test set, backtracking and DLX reached a solution much more quickly than any of the random methods. The Cougaar random methods required significantly fewer guesses than backtracking. Random Best was again the superior random Sudoku problem solving method, but only after time-limited failures were considered.

Final Analysis

These tests clearly show the brute force solving method used by backtracking was the fastest. Random Best consistently produced solutions using the least number of guesses. The performance of Random Worst and Random Best on the randomly generated Sudoku puzzles indicated that a good guessing strategy could be devised by restricting the set of agents available for guessing. Although several of Random Worst's attempts exceeded the time limit, the rest were completed with relatively few guesses. Differing performances of DLX, backtracking, and the Cougaar approach showed that certain Sudoku puzzles were more suitable for a particular solving method than others. However, the exact cause of this was not apparent. Overall, the Cougaar distributed agent approach demonstrated itself capable of solving all Sudoku puzzles tested.

The Cougaar approach showed weakness in one area specifically. Occasionally, messages would not be delivered to the intended agent, or would arrive later than usual. To compensate for this, time delays and the resending of messages was needed. In some cases, an agent would cease to receive messages entirely. Only restarting the node containing the agent could restore communication with the agent. These factors caused the overhead needed for message passing to greatly outweigh the time spent actually executing code at the agent level. Additional research is needed to search for a solution to these problems.

FUTURE WORK

Role of the Board Master

Currently, the board master fulfills several roles in the system. It loads new boards, logs results, keeps cell agents synchronized, provides interfaces for user control, and handles guessing. To create a more interesting environment for study, some of these tasks could be shifted from the board master to cell agents or communities of cell agents. For example, the cell agents could be individually responsible for making sure their neighbors are synchronized with each other. Communities of cell agents could handle stall detection, guessing, and rolling back.

Pre-guess Analysis

Some Sudoku fans are of the opinion that any well-formed Sudoku puzzle can be solved without guessing. The current implementation of the Agent Method resorts to guessing early on. Certainly the board master or individual cell agents could do more analysis to delay or eliminate the need for a guess. Certain types of analysis, such as looking at groups of related cells which share the same n-choices, would be particularly suited to an agent-based solver.

Guess Techniques

The potential for solutions with fewer guesses was demonstrated for the Random Worst method. This could be explored by changing the random number selected from a random worst agent to some fixed selection, such as first or last. Undocumented tests on a completely empty Sudoku puzzle seem to indicate that a first choice from a random worst agent leads to a solution with fewer guesses than any of the random Sudoku problem solving methods discussed in this paper. A possible cause of this is the potential guess size reduction resulting from taking the first or last guess.

Performance Measurement

As mentioned in the Measurable Results section, measuring the time required by the Agent Method to solve a Sudoku puzzle currently does not yield a useful metric for measuring performance. Time spent waiting for additional messages quickly overwhelms time spent actually solving the puzzle. Separating the recorded work times from the recorded wait times may potentially yield meaningful data that could be compared to other Sudoku solving programs.

Fault Tolerance

Fault tolerance is critical to the success of any agent system where communications may be lost or individual agents may become unresponsive. Message sending in Cougaar is generally very fault tolerant. The board master keeps track of what

it expects to hear from each cell agent, so that a lost message can be re-sent or an out-of-sync cell agent can be reset. In what is likely to be a Cougaar implementation issue, a random agent may become unresponsive. The agent will continue to operate but can no longer send or receive messages. When this problem occurs, the system completely fails. Every agent must be responsive for a solution to be reached. Implementing a function where the board master detects dead cell agents and restarts them could be a potential solution to the problem. Another solution would be to use an environment where each cell agent resides in a separate Cougaar instance, so that an agent's entire Cougaar instance could be restarted. For the experiments, all 82 agents were in the same Cougaar instance therefore restarting the instance meant restarting the entire test sequence.

Puzzle Subtleties

The experiments showed that specific Sudoku puzzles were more easily solved by one Sudoku problem solving technique over another. For example, Evil 1 was most difficult for backtracking while Evil 5 was most difficult for DLX. The exact reasons causing a Sudoku puzzle to be more easily solved using one technique instead of another would be an excellent topic for further research.

REFERENCES CITED

[1] Websudoku.
Available at: http://www.websudoku.com.

[2] Puzzle communication nikoli.
Available at: http://www.nikoli.co.jp/en/.

[3] Dell magazines.
Available at: http://www.dellmagazines.com/.

[4] M. Kraitchik. *Mathematical Recreations*, page 178. W. W. Norton, 1942.

[5] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(5):1052–1060, 2003.

[6] Bertram Felgenhauer and Frazer Jarvis. Mathematics of sudoku i. *Preprint*, 2006.

[7] Roberto Tamassia and Michael Goodrich. *Algorithm Design*, pages 627–631. PWS Publishing Company, 2002.

[8] Donald Knuth. Dancing links. *Millennial Perspectives in Computer Science*, pages 187–214, 2000.

[9] Peter Gordon and Frank Longo. *Mensa Guide to Solving Sudoku*. Sterling, 2006.

[10] Aaron Helsinger, Michael Thome, and Todd Wright. Cougaar: A scalable, distributed multi-agent architecture. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 1910–1917, 2004.

[11] Michael Thome, Todd Wright, and Sebastien Rosset. Ultralog: The cougaar component model. Technical report, BBN Technologies, 2004.

[12] BBN Technologies. Cougaar developersǵuide. Technical report, BBN Technologies, 2004.

[13] The transportation security administration.
Available at: http://www.tsa.gov/.

[14] Rocky mountain agile virtual enterprises at montana tech.
Available at: http://www.mtech.edu/.

[15] Michael Emery, John Paxton, and Richard Donovan. Application and testing of a cougaar agent-based architecture. In *Computational Intelligence: Multi-Agent Systems*, pages 153–157, 2005.

[16] Philip Cohen, Adam Cheyer, Michelle Wang, and Soon Choel Baeg. An open agent architecture. In *AAAI Spring Symposium*, pages 1–8, 1994.

[17] Jon Currey. Real-time corba theory and practice: A standards-based approach to the development of distributed real-time systems. In *Embedded Systems Conference San Jose*, 2000.

[18] Sun microsystems.
Available at: http://www.sun.com/.

[19] Solutionsiq.
Available at: http://www.solutionsiq.com/.

[20] Frank Sommers. Dynamic clustering with jini technology. *Artima Developer*, 2006.

[21] K. Sycara, M. Paolucci, M. van Velsen, and J. A. Giampapa. The retsina mas infrastructure. *Autonomous Agents and MAS*, 7(1, 2), 2003.

[22] Jeffrey Bradshaw, Stewart Dutfield, Pete Benoit, and John Woolley. *KAoS: Toward an Industrial-strength Open Agent Architecture*, pages 375–418. MIT Press, 1997.

[23] Ronald Snyder and Douglas MacKenzie. Murdoch: Publish/subscribe task allocation for heterogeneous agents. In *Proceedings Open Cougaar*, pages 143–147, 2004.

[24] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[25] Brian Gerkey and Maja Mataric. Murdoch: Publish/subscribe task allocation for heterogeneous agents. In *Proceedings of Autonomous Agents*, pages 203–204, 2000.

[26] The Foundation for Intelligent Physical Agents. Fipa agent message transport service specification. Technical report, The Foundation for Intelligent Physical Agents, 2001.

[27] BBN Technologies. Cougaar architecture document. Technical report, BBN Technologies, 2004.

[28] Todd Wright. Naming services in multi-agent systems: A design for agent-based white pages. In *Proceedings 3rd International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1478–1479, 2004.

[29] Mark Bedau. Weak emergence. *Philosophical Perspectives: Mind, Causation, and World*, 11:375–399, 1997.