HEURISTIC SOLUTION METHODS FOR THE 1-DIMENSIONAL AND 2-DIMENSIONAL MASTERMIND PROBLEM

By

ANDREW M. SINGLEY

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2005

Copyright 2005

By

Andrew M. Singley

ACKNOWLEDGEMENTS

I would most like to thank my committee chair, Dr. Elif Akcali, without whom this thesis would not have been possible. Her course in metaheuristics provided me with both the inspiration and the knowledge necessary to pursue my hobby study of the problem and turn it into a thesis. I further thank the University of Florida for providing a quality education in all areas. Without supporting courses in statistics, programming, technical writing, and others, this would not have been possible.

Additionally, I thank Microsoft for creating most of the programs used in the study and in the preparation and presentation of this document.

Finally, I thank my fiancée for her support through this period. Over the past two years she has put up with my obsession with education that has proven both financially and emotionally draining for us both.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS……iii

LIST OF TABLES……..vi

LIST OF FIGURES……...……vii

ABSTRACT……...……..viii

CHAPTER

1 INTRODUCTION……...1

1.1 Combinatorial Problems……...1
1.2 Heuristics……..2
1.3 Game Problems……..5
1.4 MasterMind……...…6
1.5 Outline……...6

2 MASTERMIND PROBLEM……...7

2.1 Motivation……7
2.2 Mastermind……...8
2.3 Variations……..9
2.4 Definitions……...12

3 LITERATURE REVIEW……..13

3.1 Exhaustive Search (Decision Trees) ……..13
3.2 Genetic Algorithms and Simulated Annealing……...16
3.3 Stochastic Methods……20
3.4 Information Theory……21
3.5 Complexity and Bounds……..22

4 ONE-DIMENSIONAL MASTERMIND……..25

4.1 Greedy Construction……...25
4.2 Preprocessing……..26
4.3 Local Search……28

4.4 Tabu Search……29
4.5 Computer Implementation……..30

5 TWO-DIMENSIONAL MASTERMIND……32

5.1 Preprocessing……..34
5.2 Tabu Search……34
5.3 Computer Implementation……...35

6 EXPERIMENTATION AND RESULTS……..37

6.1 Experimental Design……..37
6.2 1-D MasterMind: Results and Discussion……..38
6.3 2-D MasterMind: Results and Discussion……..41

7 CONCLUSION AND FUTURE WORK……..45

7.1 Conclusion……..45
7.2 Future Work……46

REFERENCES……..48

APPENDIX STATISTICAL RESULTS OF EXPERIMENTATION……...50

BIOGRAPHICAL SKETCH…….……53

LIST OF TABLES

Table page

2-1. An example game of MasterMind……...…9

3-1. Number of codes by pattern and the expected number of guesses to deduce……..15

6-1. Run times for 1-D Tabu search method…….40

6-2. Run times for 2-D Tabu search method…….44

LIST OF FIGURES

Figure page

2-1. A triangular 2-D MasterMind board with diagonal feedback……...11

3-1. Number of remaining possible codes after Knuth's (1977) first guess of 1122……14

4-1. Pseudocode for Greedy 1……...…25

4-2. Pseudocode for Greedy 3……...28

4-3. Pseudocode for Local Search……29

5-1. Example of a 2-D MasterMind code and guess with feedback……32

6-1. Mean number of guesses required to solve the 1-D problem (n = 4)….....38

6-2. Maximum number of guesses required to solve the 1-D problem (n = 4)….....38

6-3. Mean number of guesses required to solve the 1-D problem (m = 8)….....39

6-4. Maximum number of guesses required to solve the 1-D problem (m = 8)….....40

6-5. Mean number of guesses required to solve the 2-D problem (n = 4)….....42

6-6. Maximum number of guesses required to solve the 2-D problem (n = 4)….....42

6-7. Mean number of guesses required to solve the 2-D problem (m = 8)….....43

6-8. Maximum number of guesses required to solve the 2-D problem (m = 8)….....43

7-1. Proposed new preprocessing step……..46

Abstract of Thesis Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Master of Science

HEURISTIC SOLUTION METHODS FOR THE 1-DIMENSIONAL AND 2-DIMENSIONAL MASTERMIND PROBLEM

By

Andrew M. Singley

May 2005

Chair: Elif Akçali
Major Department: Industrial and Systems Engineering

MasterMind is a game in which the Player attempts to discover a secret code in a limited number of guesses by submitting guesses and receiving feedback on how similar those guesses are to the secret code. This problem can be formulated as a combinatorial optimization problem and a restricted search problem. Due to the combinatorial structure of the problem, the number of possible solutions is prohibitively large as a function of the problem parameters. Due to the large search space, heuristic methods have been devised to break the secret code with a minimal number of guesses.

In this thesis we develop our several greedy constructive, local search, and Tabu search heuristics to solve the traditional one-dimensional (1-D) problem. The performance of each methods is then compared with the others and with those from the literature.

We then extend the problem to two dimensions, expanding the secret code from a vector of n stones to an $n \times n$ matrix of stones. This new problem provides the Player additional challenges and advantages over the standard 1-D problem. We propose a Tabu search solution to this new problem.

CHAPTER 1
INTRODUCTION

We begin this thesis with a brief description of combinatorial problems. We then describe different heuristic solution methods used in these problems. Next, we describe how games and puzzle can be formulated as combinatorial problems. Finally, we provide an outline of the remaining chapters.

1.1 Combinatorial Problems

Combinatorial problems are those which involve making discrete choices. These include choosing a subset from, arranging a permutation of, or defining an assignment on discrete elements from a finite set. Solving these problems often involves minimizing or maximizing the value of an objective value function defined for the problem or finding a feasible solution given certain constraints or both. Classical examples of such problems include the Vehicle Routing, Quadratic Assignment, and Traveling Salesman problems.

These problems are known to have a large number of possible feasible solutions, typically exponential or factorial in problem parameters, and thus are difficult to solve. There is usually no guarantee that a given solution is the best possible or "global optimal" solution without enumerating and testing all possible solutions. However, a branch of problem solving called "heuristics" provides tools with which combinatorial problems can be solved more efficiently than complete enumeration.

1.2 Heuristics

With the rapid increase in computing power over the past 20 years, we are able to solve problems with thousands of variables instead of tens of variables, and problems which once could take years to solve can now be solved in minutes. Despite these advances, however, many real-life problems remain intractable. This is especially true for combinatorial problems, which tend to have a high number of variables, and incredibly large search spaces. Also, many problems do not yet have efficient algorithms available to find an optimal solution.

Heuristics offer a way to circumvent the limitations in computing power. Instead of using a direct approach to calculate an optimal solution, which might require too many resources, one can use more general rules or methods, called *heuristics*, to obtain a solution of the problem. These are general solution methods which attempt to find an optimal or near-optimal solution which can be modified to fit a variety of problems. The trade-off, however, is that heuristics do not usually guarantee to find an optimal solution.

We shall now offer brief descriptions of some of the more popular classes of heuristics. To provide an example we will use a standard problem from the literature called the Traveling Salesman Problem (TSP).

The TSP involves a traveler who wishes find the lowest cost (time, money, etc) way to visit n cities. This problem is represented by a graph where each node represents a city, and each edge represents a possible path to/from that city. The weight of edge e_{ij} is the cost to travel between cities i and j. The objective is to minimize the total cost of the path. This is subject to the constraints that each city must be visited exactly once, and that the tour must end at the same node from which it began.

Probably the most common and intuitive class of heuristics is the *greedy construction* variety. Greedy methods attempt to an optimal or near-optimal solution quickly by seeking to maximize (or minimize, depending on the problem) the value added at each step. These methods are often simple to understand and implement, but not always as successful as one might hope in terms of solution optimality. One difficulty is choosing the correct attribute value to maximize, as the most intuitive choice may not be the most effective choice. A greedy construction heuristic for the TSP might involve sorting the edges in non-decreasing order by weight and adding edges to the graph if neither end node of that edge already has two edges connected.

Greedy methods almost always guarantee *a* solution, but it is usually suboptimal. One can exploit these two facts by using a greedy method to create an *initial* solution, which can then be modified by other heuristic procedures. This method is commonly used in *improvement heuristics*, methods which involve a *search* among the problem's solution space. The idea is that one can more easily find the optimal solution by starting with a "good" solution and looking for better solutions than by trying to create the optimal solution directly. This could also be used to look for the *best available* solution given a limited amount of time to find a solution.

The simplest methods in this class are called *local searches*. A local search is another intuitive heuristic wherein one begins at a current solution and looks at nearby solutions in the local *neighborhood*. This follows from the logic that good solutions tend to be near other good solutions. If we already have one good solution, then the nearby solutions are also likely to be good, and thus are worth the time to search to see if one is better.

If one were searching for an optimal solution, one might begin at the best available solution so far. The search would then expand outward from that point since those locations would have a higher probability of success than a randomly chosen location. Similarly, if one wished to maximize the value of a function, one might take an existing solution and try changing the value of each variable independently by a small amount. In this way one could determine which variable to change to create the largest increase in the objective function value.

The effectiveness of a local search depends on the assumption that there is some path on which solutions will continue to improve until the optimal solution is reached. If the search space is not contiguous, or there are multiple solutions which are each locally optimal for that particular section of the search space, then the local search is likely to get stuck at a suboptimal solution.

Another drawback is that local search methods require an initial solution from which to search. Some problems may offer a trivial solution from which to begin, but starting from a better solution might allow the search to be completed faster, and may reach a better solution at the end. This is where a greedy construction method can be useful to create the initial starting solution.

For our TSP example, it might mean swapping the order in which two cities are visited. Whether these two cities must be sequential on the current tour or not is up to the researcher.

There are many other popular search heuristics, including Tabu Search [?] and evolutionary methods (such as Genetic Algorithms, Neural Networks, and Ant Colonies). Tabu search seeks to overcome the difficulties of local search methods. This method

does not become trapped at locally optimal points, and can even be designed to handle non-contiguous search spaces.

Tabu search is similar to local search, except the move to make is chosen randomly instead of looking for the best neighboring solution. The direction of that move is then added to a list of recently used moves which are now unavailable or "tabu." Moves on this list can only be made again if the resulting solution has a high *aspiration* (for example, if the value of that solution would be better than the value of the best solution found thus far.) Each time a new move is added to the tabu list, the oldest entry in the list is removed, so the length of the list remains constant.

For the TSP, this might mean randomly swapping the ordering of two randomly selected cities on the tour. These two cities would then be added as a pair (not individually) to the tabu list and would not be allowed to be swapped with each other unless the aspiration level is high enough.

1.3 Game Problems

Games and puzzles can be formulated as combinatorial problems. These generally require that the player(s) choose from a finite number of possible moves. The rules that govern what moves are allowed or not serve as a set of constraints on the choice. The objective of the game is to maximize the points received or to meet some other success criterion. This collection of a discrete set of elements, a set of constraints, and an optimization criterion allow one to formulate a game or puzzle as a combinatorial optimization problem.

1.4 MasterMind

The MasterMind problem is one of choosing a subset of elements from a larger finite set. This subset is considered optimal if it matches a preselected subset and thus maximizes the value of an objective function. Constraints are developed during play based on the feedback received describing the relationship between a previously chosen subset and the optimal subset.

The number of possible solutions is exponential with respect to the problem parameters. Heuristics have been developed to more efficiently search this set of solutions. We develop additional heuristics.

1.5 Outline

In the next chapter we formally define the MasterMind problem and present some variations. Chapter 3 presents a review of the literature on the problem. Chapters 4 and 5 examine solution approaches to the 1-D and 2-D MasterMind problem respectively. In Chapter 6 we examine the results of our computational experiments. Finally, in Chapter 7 we draw conclusions from our results and suggest possible directions for future work.

CHAPTER 2
MASTERMIND PROBLEM

In this chapter we will formally define the MasterMind problem and consider several variations which have been proposed.

2.1 Motivation

Millions of people around the world play the game of MasterMind in one form or another. Many players have developed their own strategies for successfully winning, although they might have some difficulty articulating the exact rules used. The traditional game of MasterMind, though it is a combinatorial problem, is still very small and simple. However, the rules are simple enough that the game can be generalized into more difficult problems without needing to change them.

These larger, more difficult problems would probably prove too difficult for human players to play (or at least to enjoy playing.) Computers, however, excel at solving large problems with well-defined rules. Thus we can create computer simulations of the generalized MasterMind problem in order to study the effectiveness of different solution methods. This also allows us to test the sensitivity of different methods to changes in the problem parameters.

We begin by testing how well greedy, local search, and Tabu search methods perform in solving the one-dimensional problem. We then extend the problem to multiple dimensions and test how well Tabu search performs vis-à-vis this generalization. In this

thesis we shall directly implement and test two-dimensional problems, and will discuss the effectiveness that one might expect in high dimensions.

2.2 MasterMind

The game of MasterMind was invented in 1970 by Israeli postmaster Mordechai Meirovitz. Invicta Plastics, Ltd. purchased the intellectual property rights and has contracted with various toy and game manufacturers to produce products based on the game. Pressman Toy, Inc. currently produces and markets standard MASTERMIND™ as well as "super," children's, and travel versions of the game in the U.S.

MasterMind is a game in which a Player (i.e. "codebreaker") attempts to deduce a secret code generated by the Oracle (i.e. "codemaker") in as few turns as possible by submitting guesses and receiving feedback based on how each guess is related to the secret code. Each turn, the Player creates a permutation of colored stones. There are m possible colors for each stone, and n stones in a valid code. We shall denote a game of MasterMind as $MM(n, m)$, with n and m defined as above. When the Player is satisfied with his/her guess, it is submitted to the Oracle. The Oracle then places b black pegs, called "bulls" next to the code, one for each stone which has the correct color in the correct location. The Player does not know *which* stones have matched, only how many. The Oracle then places w white pegs, called "cows" next to the code, one for each stone which matched the color of an as-yet unmatched stone in the code, but was not in the correct position. This definition has traditionally been vague, but we will see how it can be formalized in the literature review.

Play continues in this manner until the maximum number of guesses (10 in the traditional game) have been played, or the player has received feedback of n black pegs.

Note that this winning condition is different from simply "knowing" the secret code. If the player "knows" what the secret code must be (the remaining search space includes only one code) after his/her turns are up, but did not receive *n* black pegs by the last turn, then the game is still lost.

A typical game (not using any specific strategy) might look like this:

Table 2-1. An example game of MasterMind.

Secret Code:	1251	
Guess #1:	1111	BB__
Guess #2:	1122	BWW_
Guess #3:	1123	BWW_
Guess #4:	1124	BWW_
Guess #5:	1125	BWWW
Guess #6	2511	BWWW
Guess #7:	2151	BBWW
Guess #8:	1251	BBBB

2.3 Variations

Below we present a list of variations on the MasterMind problem which have been proposed thus far.

1. **"Super Mastermind":** In addition to the regular MASTERMIND™, Pressman also produces a larger "super" version of the game with more stones and colors. This most-often studied variant of MasterMind involves varying the parameters *m* and *n*. Because the search space is m^n, the size and complexity of the problem increases rapidly.
2. **No Repeated Colors:** This is actually quite similar to the game "Cows and Bulls" from which the original MasterMind was derived (Knuth, 1976-77). The only change is that each color in the code must be unique (no repeated colors al-

lowed). Generally, the ratio of colors to stones will be greater than with repeats allowed because the search space is smaller: $\binom{m}{n} < m^n$.

3. **<u>Static Mastermind:</u>** In the "static" version of the game, the Player must submit all of his/her guesses simultaneously. After receiving feedback for all submitted guesses the Player then announces the secret code deduced from the feedback or loses the game. Because feedback is not provided between guesses, it can be expected that the Player will require more guesses in order to acquire sufficient information. For research on this problem, see Greenwell (1999-2000) and Chvatal (1983).

4. **<u>Dynamic MasterMind:</u>** Bestavros and Belal (1986) suggest a new variant of the game wherein the Oracle plays a more competitive role. After the feedback is given for each guess, the Oracle has the option of changing the secret code to a new one so long as it is still consistent with the feedback given for all previous guesses.

5. **<u>"MathMind":</u>** Reddi (2002) proposed two new variations "that will be of interest to mathematicians/engineers/scientists." The first variant made the standard game easier. This version replaced the white feedback pegs with two types of pegs called "higher" and "lower." This reduced the ambiguity of the near-miss guesses by signaling how many stones need to be increased in value, and how many decreased. It seems logical to surmise that this version would indeed be much easier to solve as the number of dimensions increased.

The second proposed variant involves calculating an integer "distance" between the current guess and the secret code as the only unit of feedback. The pa-

per considered using Euclidean distance and commented that a maximum of two guesses would be sufficient for the *MM(4,6)* game. Such a tight upper bound on the worst case makes this version of little interest to research. However, it would provide an interesting challenge to a human player. Reddi also suggests that other distance criteria are possible which should make for a more complex problem.

6. <u>Different shapes:</u> Ungór (personal communication, 2004) suggested the possibility of creating other shapes beyond the traditional square such as triangles, circles, etc. One can argue that other shapes could be simulated on a square board as below, using the number (i.e. -1) to represent known "invalid" stones and solving the resulting matrix as one would in the standard game. This transformation would also hold for the normal game with $|c| \neq |r|$ (i.e. a rectangle).

An interesting note about using different shapes is that they lend themselves well to different feedback directions (diagonal for triangle, concentric circles for circular, etc). However, one can transform these into a standard square matrix with column and row feedback. Indeed, one could transform any *d*-dimensional non-square game into a *d'*-dimensional (possibly sparse) square game, where *d'* is the number of feedback directions. Consider this example:

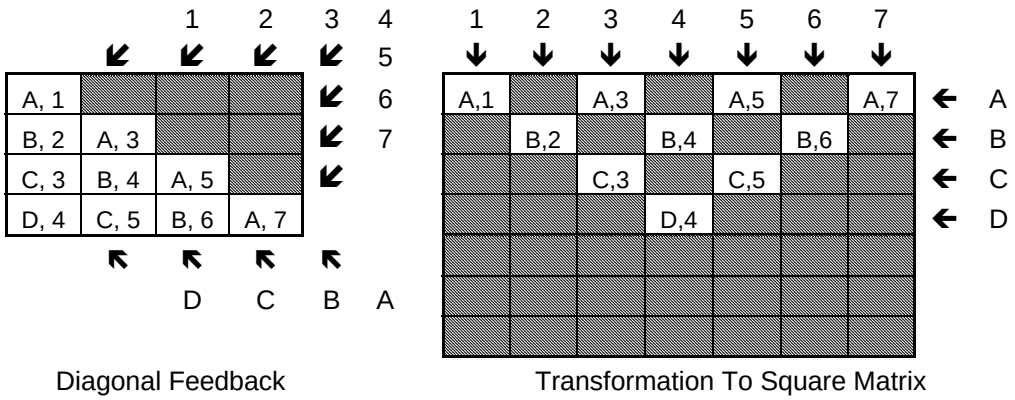

Figure 2-1: A triangular 2-D MasterMind board with diagonal feedback

2.4 Definitions

Bulls: Black feedback pegs designating a "hit."

Colors: The set of possible values for each element in a valid code. Denoted as m.

Cows: White feedback pegs designating a "near miss."

Hit: A match between the ith elements of the guess and the secret code.

MasterMind: A game in which the player attempts to deduce a secret code by asking questions and receiving feedback. Denoted here as $MM(n,m)$ where n is the number of elements in the code, and m is the number of possible values for each element.

MASTERMIND™: A product sold by Pressman Toys based on the standard $MM(4,6)$.

Near Miss: A match between the ith element of the guess and the jth element of the secret code where there was not a "hit" in either location. ($i \neq j$)

Oracle: One who generates the secret code and provides feedback.

Player: One who attempts to deduce the secret code by submitting guesses.

Stones: The set of elements which comprise a valid guess or code. Denoted as n.

CHAPTER 3
LITERATURE REVIEW

3.1 Exhaustive Search (Decision Trees)

The first formal paper published on the MasterMind problem was written by renowned computer scientist Donald Knuth (1976-77). This paper provided solid groundwork for future research on the MasterMind problem, and has been cited in almost every paper since. It also provided a surprisingly tight upper bound to the average and worst-case number of turns required to win the game when compared against later research.

He was the first to give a formal definition for the number of bulls and cows received as feedback. The definition of the bulls was straightforward, but the cows had been a vague and contended point before now. His definition is given below (where n_i is the number of times color i appears in the secret code, and n'_i is the number of times it appears in the current guess.):

$$w = \#\text{ of cows} = \sum_{i=1}^{M} \min(n_i, n'_i) \quad - \quad (\#\, of\ bulls)$$

His method was a simple greedy heuristic with a single decision rule: minimize the maximum number of remaining possibilities after each submitted guess. This was done by starting with the observation that there are only fourteen possible feedback combinations, and each has an easily calculated maximum number of remaining possible codes in the search space. (The reader is invited to think about why the fifteenth possible feedback, namely "BBBW," is not feasible.) This simplified the search for the optimal

next guess at each turn by using the fourteen possible outcomes and choosing the code which has the smallest maximum number of possible codes remaining.

Feedback	Remaining Codes
(None)	256
B	256
BW	208
BWW	36
BWWW	0
BB	114
BBW	32

BBWW	4
BBB	20
BBBB	1
W	256
WW	96
WWW	16
WWWW	1

Figure 3-1: Number of remaining possible codes after Knuth's (1977) first guess of 1122

He generated a decision tree with which to play the standard game (*MM(4,6)*) through an exhaustive computer search. Rather than drawing out a complete tree, he created a condensed method to represent each decision in the tree. It reads similar to the following (where *n* is the number of remaining possibilities, *y1y2y3y4* is the next code to be guessed, and a_i represents a similar nested structure representing the next guess if the *i*th possible feedback is received):

$n(y1y2y3y4: a_1, \ldots, a_{14})$

Despite the simplicity of the greedy decision rule, it proved quite successful. The average number of turns required to win the game was 4.478, with a worst-case of 5 turns required (the best-case is always 1). These were good results given the requirement that the player must win in 10 or fewer guesses. He noted that his strategy was not likely to be optimal, but proposed that it was likely to be very close.

Irving (1978-79) took Knuth's work one logical step further. Instead of trying the minimize the *maximum* number of possible codes after each turn, he chose to minimize the *expected* number remaining after each of the first two turns. In so doing, he reduced

the average number of guesses required to 4.369, but with a single code requiring a worst-case of 6 guesses. As a trade-off he also offered a slight modification to his strategy to reduce the worst-case back to 5 guesses. This also meant increasing the average number of guesses to 4.370. The importance of this result was to demonstrate the dichotomy which can be present in a strategy wherein one must choose between either minimizing the average or worst-case performance.

As an extension, he challenged the existing assumption that the Oracle would choose the secret code at random. The Oracle wishes the maximize the number of guesses required by the Player (and ideally prevent the code from being broken within the number of turns available). Certainly not all codes would prove equally challenging. His study began with the observation that there are only five possible classes or patterns of codes, namely *aaaa*, *aaab*, *aabb*, *aabc*, and *abcd*. He then proceeded to calculate the number of codes matching each pattern, and the expected number of guesses for that class of codes using his strategy. His conclusion (see table) was that the Oracle would be most successful by choosing a code of the form *aabc*, but only marginally better than most other codes. As one might expect, the Oracle would be least successful with a code of the form *aaaa*.

Table 3-1: Number of codes by pattern and the expected number of guesses to deduce.

Pattern	Number of Codes	Expected Guesses
aaaa	6	3.667
aaab	120	4.367
aabb	90	4.367
aabc	360	4.367
abcd	720	4.376

From Irving (1978-79).

Unfortunately this information would prove more relevant for a game being played against a human Player than for further research on the problem. In most research on the one-dimensional game the researcher would typically test the algorithm or heuristic being evaluated against all m^n codes. However, the search space becomes so large in the multi-dimensional game, m^{rc}, that the researcher could be expected to randomly generate test cases. Because of the sampling issue, one could consider both an unbiased Oracle who creates random secret codes and a competitive Oracle who has knowledge similar to the above paragraph.

Koyama and Lai (1993) took a slightly different approach to create a decision tree. Borrowing from abstract algebra, they considered equivalence transformations between codes to reduce the set of possible codes to be considered for the next guess. By using equivalences to reduce the search space at each step, they performed a depth-first search to create a decision tree similar to Knuth (1976-77) and Irving (1978-79). This tree produces an average number of guesses of 4.340 with a single worst-case of 6 guess. Alternately, the tree can be modified to give an average of 4.341 guesses, but a worst-case of 5.

3.2 Genetic Algorithms

While decision trees may yet hold the secret to an optimal playing strategy, they require the Player to have all the necessary information beforehand. They are also very rigid in that a new tree must be created for every combination of m and n that the Player might encounter. These limitations, combined with the popularity of Genetic Algorithms in the 1980s and 1990s, lead to their use in solving the MasterMind problem.

Genetic Algorithms (GA) are based on the evolution of DNA through successive generations of life. Those parents who are considered most "fit" are more likely to survive and to create offspring who have an even higher level of "fitness." Because the guesses in MasterMind have an identifiable fitness value (the feedback received or the number of "constraints" met or both) and a generational structure (turns), the MasterMind problem proved a natural fit for such algorithms.

Bernier et al. (1996) convert MasterMind from a restricted search problem to a constrained optimization problem and propose a GA solution. Previous guesses and the feedback received are saved and used as "constraints" or "rules" for future turns. The fitness of a chromosome (potential guess) is related to the number of constraints that it satisfies. A constraint is satisfied if the feedback received by playing a previous guess against the one currently under consideration is the same as the actual feedback received when that guess was submitted.

Unfortunately, the number of chromosomes with the same fitness level (satisfy the same number of constraints) is likely to be high, especially in early rounds. To avoid this problem, Bernier et al. modified the fitness function to emphasize those rules which are "better" than others. This is achieved by setting the fitness of constraint i equal to $10b_i + w_i$. This appears logical because a code with feedback of two bulls would be considered "closer" to the secret code than one with feedback of two cows. However, this does not always imply that the rule provides more "information" about the secret code. Consider if the Player of the *MM(4,6)* game submitted the guess 1234 and received no bulls or cows. This would mean that the secret code consisted solely of 5s and 6s. This

would certainly be valuable information, but would result in a fitness contribution of 0, and hence the constraint could effectively be ignored.

As with many GAs, a bitwise chromosome representation was chosen. Each stone was represented by the minimum number of bits, y, such that $m \leq 2^y$. The length of a chromosome was thus yn bits, less than the $32n$ bits needed for a typical integer representation. Because a modulo method was used to avoid invalid colors, there is a bias for the lower numbers if m is not a power of two. The authors decided that this was not a serious issue and that the space savings and ease of operations were more important.

Three operations were defined for this population to achieve a balance between diversity and fitness. The first was a traditional two-parent double-crossover. Unfortunately this does not consider gene (stone) boundaries, so the crossovers can occur in unnatural places.

The other two operators are single-parent operations designed to provide controlled diversification. "Cloning with mutation" flips single bits in proportion to the remaining stones unaccounted for by feedback in the most recently submitted guess (that is, $n - b - w$). This provides more randomization in the early stages when little information is available, and less randomization near the end when most stones have either matched or "nearly matched" with the secret code.

The second single-parent operator is called "cloning with transposition." This operator is the most natural fit with the way the MasterMind game is played, as entire genes (stones in the code) are transposed instead of randomly selected bits. This operator is used in proportion to the number of cows received, as those mark the number of stones which are in the wrong positions.

For comparison purposes, the authors also implemented a non-optimal simulated annealing (SA) algorithm, and an optimal random search algorithm. The random search generated random guesses until one was found that met all of the feedback rules thus far. Because Rosu (1997) and others have tested the efficacy of random searches and found good results (see following section), this was a logical comparison.

The SA runs, however, were limited in time to force non-optimality. That is, the guess which had met the most feedback rules when time ran out would be submitted. They argued that this algorithm would have become a random search algorithm if allowed to run until an optimal solution was found. Because solutions are chosen at random, the SA method would continue choosing random solutions until on was found that satisfied all previous feedback rules. However, this is the way that the random search functions. By forcing non-optimality it can be expected that the SA method would not perform as well as the other methods which play optimal guesses.

The computational results are fairly straightforward and as expected: SA was the fastest (seconds or less), GA was about two orders of magnitude slower (minutes to hours), and the random search was an additional order slower (minutes to tens of hours.) However the average number of guesses were not as clearly different. In fact, GA and random search showed negligible differences in this measure. Exact numbers are not given, only graphs, but the variance bars suggest that the two approaches are not different in a statistical sense. As expected, the non-optimal SA algorithm required a higher number of guesses on average.

In a follow-up paper, Merelo et al. (1999) test the effects of a different fitness function on GA performance. They consider the “distance” between a guess candidate

and each of the feedback rules provided thus far. This was done to create a "smoother fitness landscape." The changes resulted in an average of 4.132 guesses. As of this writing, this is the second-best average result published.

3.3 Stochastic Methods

One of the most widely cited papers on MasterMind was an undergraduate thesis by Rosu (1997). In this paper, a very simple random heuristic was used which gave very good results. For each turn a random guess was generated and tested for consistency against the previous guesses and the feedback received. If this guess is consistent with all previous guesses, then it is submitted to the Oracle. If the guess was not consistent with the feedback of one or more previous guesses, then it was discarded and a new guess was randomly generated.

This algorithm ran very quickly (on the order of seconds using a 100MHz Pentium.) The time required increases exponentially, but is still reasonable for higher values of m and n than are present in other literature. Comparison of results is limited by the lack of other literature for most combinations of m and n tested here. However, compared to Knuth (1977) on *MM(4,6)*, the random heuristic only performs 4% worse on average number of guesses, at 4.66. Such a good result puts pressure on the more complex methods, both to prove their efficacy compared to a random search and also because they tend to be considerably more time-intensive. Unfortunately, the maximum number of guesses played rose to 8 guesses from the Knuth (1976-77) maximum of 5.

A more restricted random search heuristic was presented by Temporel in (2003). He used the information contained in the bulls and cows to guide the random search. After each submitted guess, the feedback provided is used to determine if that guess was

closer or farther from the secret code than the best guess so far (the Current Favorite Guess or CFG.) If this new guess was the best so far, then it became the CFG. If not, then the old CFG is kept. Either way, the CFG is used to generate the next guess to be submitted.

Each bull corresponds to a stone which is currently in the correct position, so *b* stones were randomly chosen to be kept from the CFG to the next guess. Similarly, each cow corresponds to a stone which is a correct color but in the wrong position, so *w* stones were randomly chosen to be kept from the CFG to the next, and the position of each was randomly changed. To fill in the (n - b - w) remaining stones for the next guess, the author created a probability distribution for the *m* colors. This is done to maintain diversity in the new code by making it less likely that colors which are already present in the partially-completed next guess will be added randomly to complete the guess.

The results were almost identical to Rosu's for the average number of guesses, obtaining a best average of 4.64 for the *MM(4,6)* and 5.888 for the *MM(5,8)* case. The advantage, however, came from the average number of codes evaluated before finding a consistent guess to submit. Temporel's algorithm evaluated an average of 41.2 codes, where Rosu's evaluated an average of 1295 codes for the *MM(4,6)* case and 480.1 instead of 32515, respectively, for the *MM(5,8)* case. In fact, the results suggest that the number of codes evaluated *increases at a decreasing rate* as the problem size increases. This is an important result because the search space increases exponentially with problem size.

3.4 Information Theory

Bestavros and Belal (1986) approached the problem from an information theory perspective. Instead of minimizing the number of possible solutions remaining at each

turn, they looked to minimize the pool size up to L turns after the current turn. In order to achieve this they calculated the amount of information (in bits) to be gained from submitting a particular code given the probabilities of receiving each of the 14 possible feedback results (from chapter 2). In addition, the authors calculated the number of bits necessary to deduce the secret code. The reasoning was then that the sum of the information gathered from each turn should equal the amount of information required to deduce the code.

Two strategies were then tested. The first maximized the minimum amount of information to be gained from the submitted code. The second maximized the expected information to be received. These gave results of 3.860 and 3.835 guesses required on average.

There is a question about whether these results can be directly compared to other published results. The winning conditions were given as having a remaining pool of candidate solutions of size one after the last submitted guess. This is very vague, and implies a different winning condition than used in other papers, namely that $b = n$ after the last guess. If this is indeed the case, then these results may not be directly comparable with results of other reported work.

3.5 Complexity and Bounds

De Bondt (2004) proved that the problem of finding a guess which is consistent with all previous guesses is NP-complete. As with most NP-completeness proofs, this was done by reducing a known NP-complete problem to the current problem. Specifically, the author showed that the (1 in 3)-SAT problem could be reduced to the MasterMind problem.

Most papers thus far have been concerned with how well a given strategy or algorithm performs in practice. However, Chvatal (1983) examined the theoretical bounds on the number of guesses required to solve the problem. He concluded that, if $n \le m \le n^2$, then the maximum number of questions, *f(n,m)*, needed to break a code is given by:

$$\frac{n \log_2 m}{\log_2 \binom{n+2}{2}} \quad \le \quad f(n,m) \quad \le \quad 2n \log_2 m + 4n$$

For *MM(4,6)*, this means that the number of turns needed to break any single code is between 2.65 and 36.68. While this is a loose bound for this case (Knuth (1977) has shown that no more than 5 turns are needed), it is important because $n \log_2(m) + n = \log_2(m^n) + n$ grows more slowly than the search space size, m^n. For example, for *MM(8,16)*, the size of the search space is 4.29E+09, but the number of guesses needed is bounded from above by 96.

Chen et al. (1996) developed a new upper bound by implementing a binary search algorithm. His method removed the previous restriction that $m \le n^2$ and is given by:

$$f(n,m) \quad \le \quad \left\lceil \frac{m}{n} \right\rceil + 2n \log_2(n) + n + 2$$

This improves the number of turns needed to break any code from 36.68 to 28 for *MM(4,6)* and from 96 to 68 for *MM(8,6)*.

Greenwell (2000) solved the *MM(4,6)* static MasterMind problem. The static problem involves finding the minimum size set of codes which, when submitted simultaneously, will provide the Player with sufficient feedback to uniquely determine the secret code in the next turn. For the *MM(4,6)* case, he found a set of six codes which would yield sufficient feedback, which means that the static game can be won in seven turns.

Because the regular game is a simplified version of the static game (the same set of codes could be submitted sequentially instead of simultaneously), the solution of the static game for a given n and m provides an upper bound on the number of turns necessary to win the regular game. This number can be compared to test the worst case performance of a given algorithm. If the worst case is greater than the static bound, then there is room for improvement. For the interested reader, his (possibly not unique) set is: 1221, 2354, 3311, 4524, 5656, 6643.

CHAPTER 4
ONE-DIMENSIONAL MASTERMIND

In this chapter we focus on the traditional 1-D MasterMind problem. We develop three classes of heuristic methods – greedy construction, local search, and Tabu search – to solve the standard MasterMind problem. We propose a preprocessing approach to quickly reduce the search space. We then propose a greedy construction and a local search approach for the problem. Finally, we develop a Tabu search approach for the problem.

4.1 Greedy Construction

We first implement a simple greedy heuristic which we denote as "Greedy 1." All stones except one are held constant, and that one rotates through all available colors until the correct color is determined. This is done by testing to see if the number of bulls has increased between one guess and the next. If this is true, then a match has been found and it is both the correct color and in the correct location. The method then proceeds for each additional stone until the secret code has been determined.

```
 FOR I = I TO N
   FOR J = 1 TO M
      GUESS[I] = J
      IF NEW BULL FOUND
THEN
         EXIT INNER LOOP
      END IF
   END INNER LOOP
```

Figure 4-1: Pseudocode for Greedy 1

There are m possible colors for each stone, and n stones to be determined. This gives a maximum number of guesses needed (worst-case performance) of mn. While this is a large improvement over complete enumeration, there is still much room for improvement. The following two observations were used to improve the method's performance based on the problem structure:

1. Only complete guesses can be submitted. This means that the color c_1 need not be explicitly tested for stones 2 through n because these were implicitly tested when solving for each previous stone. The worst-case is thus reduced by $(n - 1)$.
2. There is exactly one color for each stone. Because we know how many colors are available we need not test for the final color, c_m, for any stone. This value can be deduced from the results of testing the first $(m - 1)$ colors a given stone. The worst-case is further reduced by n.

By implementing these two improvements, which we shall denote as "Greedy 2," the worst-case performance is reduced to $1 + n(m - 2)$. However, the heuristic is still very generic and the upper bound still leaves additional room for improvement. We now attempt to improve the worst-case performance of these constructive heuristics by adding a preprocessing step.

4.2 Preprocessing

Each stone which is an element in the permutation that is the secret code has two attributes: a position and a color. Previously, we fixed the position and searched for the correct color. However, an observation of human players suggests a different solution approach. That is, we can first determine how many of each color are in the secret code, and then determine the correct positions.

The method is simple: the first m turns are to be used to determine the correct number of each color appearing in the secret code. This can be done by submitting first a guess of 1111, then 2222, up to mmmm. By the second observation implemented in

Greedy 2, we do not need to test the final color. We combine this method with a variation of Greedy 1 to create our final "Greedy 3" method.

Such a method could be wasteful because the Player would be submitting codes *known* to be wrong. For example, it would be unwise to submit 2222 if the Player knows that there is exactly one 1 in the code. Instead, we choose to keep each of the previously determined colors in the submitted guess at each round. Thus instead of submitting 2222, the Player would submit 2221. There is a small, but nonzero, probability of guessing the correct code during the first *m* turns with this modification.

After we know the *n* colors present in the code, we proceed to enumerate through that set until the correct value of the first stone is determined (similar to the previous two greedy methods). This proceeds to the second stone, which need only search through (*n - 1*) possibilities, and so on until the last stone has only a single possible color. By the first observation in Greedy 2, we must fill in the stones which are not being tested to create a complete guess. We choose to fill the untested stones with the color which occurs the most times in the code. If there is a tie, then the color with the highest numerical value is chosen because we search through the colors in non-decreasing order. By filling in the blank stones in this way, the last stone need not be explicitly tested.

We can now calculate the worst-case performance as the sum of the preprocessing worst-case and the worst-case of the greedy step.

$$\text{Preprocessing + Greedy} = (m-1) + (\sum_{i=1}^{n} i - 1) \quad = \quad m + \frac{n(n+1)}{2} - 2$$

```
FOR I = 1 TO (M – 1)
  SUBMIT TEST CODE "XXXX"
  COLORSET[I] = # OF COLOR I
END FOR

FOR I = 1 TO (N – 1)
  FOR J = 1 TO N
    FILL EXTRA STONES WITH COLOR
    GUESS[I] = COLORSET[J]
    IF BULL FOUND THEN
      REMOVE ELEMENT J FROM COL-
ORSET
      EXIT INNER LOOP
    END IF
  END INNER LOOP
END OUTER LOOP
```

Figure 4-2: Pseudocode for Greedy 3

4.3 Local Search

We now turn our attention from constructive heuristics to improvement heuristics. First, we consider a *2-swap* local search that is almost as straightforward as the greedy methods. The preprocessing step from Greedy 3 is used to generate an initial solution from which to search. The positions of stones are swapped in a methodical order and improving solutions are kept. Non-improving solutions are discarded, and the stones are returned to the previous positions. To improve the computational efficiency, two stones of the same color are never swapped.

Values are assigned to the feedback of a guess to determine if it is an improvement over a previous guess. Each bull is given 2 points, and each cow is given 1 point. The value of the feedback is then summed and compared with the previous value. We adopt a first-improving strategy in determining which improved solutions to keep. This is chosen to mean strictly improving, so we keep the new guess only if the new feedback has a value strictly greater than the previous value.

```
FOR I = 1 TO (M – 1)
  SUBMIT TEST CODE "XXXX"
  COLORSET[I] = # OF COLOR I
END FOR

GENERATE INITIAL SOLUTION FROM COLORSET[]

FOR I = 1 TO (N – 1)
  FOR J = (I + 1) TO N
    SWAP STONES I AND J
    IF IMPROVED SOLUTION THEN
      KEEP CODE AND CONTINUE
    ELSE
      SWAP STONES BACK
    END IF
  END INNER LOOP
END OUTER LOOP
```

Figure 4-3: Pseudocode for Local Search

The worst-case performance is again the sum of the preprocessing worst case and the 2-swap worst case. We then have:

$$\text{Preprocessing + Local Search} = (m-1) + \binom{n}{2} = m + \frac{n(n-1)}{2} - 1$$

4.4 Tabu Search

We now focus on developing a Tabu search approach for the 1-D MasterMind problem. Instead of iterating through the 2-swaps in a specific order as in the local search, pairs are chosen at random. The positions of the two stones to be swapped are then added as a pair to the Tabu list. Unlike the local search, this Tabu search does not submit a guess after each pair swap. Instead, it continues to search until a consistent guess is found, and only then it the guess submitted.

A pair which is not on the Tabu list is swapped regardless of whether it is an improvement over the current solution or not. If a chosen pair is already on the list and the new guess is consistent, then the feedback after making the swap is compared to the best feedback found so far (the aspiration criterion). The value of a guess is based on the

feedback received by assigning 2 points to each bull and 1 point to each cow and summing the result. If the new solution does not have a higher value than the best found so far, or if the resulting guess would not be consistent then the pair is swapped back. However, a solution which is an improvement over the best value is always kept and the best value is updated to reflect this solution.

The Tabu tenure was experimentally set at (n - 2). This was experimentally determined to be small enough to maintain diversity, but large enough to prevent it from becoming a purely random search. Diversity is important because for small values of n, the number of possible pairs is also small. For $n = 4$, there are only 6 pairs.

The implementation has a mechanism built-in which keeps track of the remaining search space which is consistent with the feedback received from guesses submitted so far. A guess is consistent if it has not yet been eliminated as the possible secret code by the feedback obtained from previous guesses. Each potential guess is screened to test for consistency prior to submission to the Oracle. We believe that this will make the search more efficient, as every submitted guess will have a nonzero probability of being the secret code.

Because the swaps are chosen at random, it is not possible to determine a fixed worst-case bound as we did with the previous methods. One might expect the bound to be reasonably close to that of the local search as a result of the similarities between the two.

4.5 Computer Implementation

The data structures used in the implementation of the solution methods are relatively straightforward. Integers were used for representing colors, and an array of inte-

gers was used to represent each guess or code. The Tabu list stored a pair of integers to represent each pair swap.

The greedy and local search methods were tested on all possible codes for chosen values of n and m. Due to its stochastic nature, the Tabu search method was run 5 times for each combination to assess the average and worst-case performance of the method experimentally.

Because the run times for greedy construction and local search were generally less than one second for all m^n codes to be tested, reliable per-code runtimes could not be measured. However, Tabu search ran for a significantly longer time, so per-code run-times were determined by dividing the total run time by the number of codes tested.

The long runtimes for Tabu search also limited the size of problems that could be tested. For example, the complete testing of *MM(7,8)* was extrapolated to take roughly 117 days, and thus was not tested. Instead, we generated 20,000 random test cases to have some data to compare to the other methods at this level.

CHAPTER 5
TWO-DIMENSIONAL MASTERMIND

In this chapter we present the crux of our research: a study of the two-dimensional MasterMind problem. We begin by briefly comparing and contrasting the 1-D and 2-D problems and methods for the solution of each. We then present a heuristic method for solving the 2-D problem using a preprocessing stage similar to that used in the previous chapter and a Tabu search.

Each 2-D code can be considered a $r \; x \; c$ matrix of stones. For simplicity we choose to study square problems, where $r = c = n$. We denote this 2-D square case as *MM2(n,m)*. Feedback is provided from two directions, for each row and each column. This is essential to differentiate the problem from the 1-D problem (albeit solving several 1-D codes in parallel). If feedback were only provided from one direction, then there would be no additional challenge to the problem, nor any new advantages to exploit.

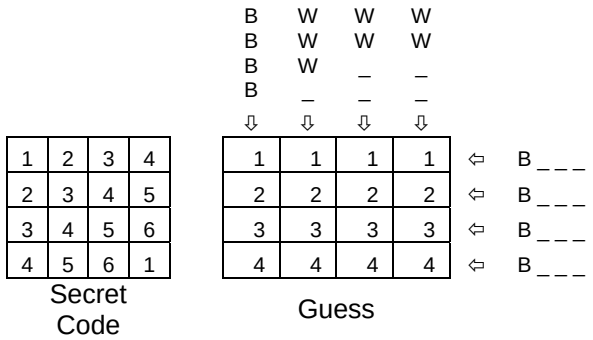

Figure 5-2: Example of a 2-D MasterMind code and guess with feedback

It should be mentioned that the 2-D problem can be deconstructed into r (the number of rows) or c (the number of columns) 1-D problems and solved with methods

discussed previously. This result is useful because it allows us to compare performance of a 2-D method with the performance of existing 1-D methods. The worst-case performance of a 1-D problem would be the same if the method was applied to a 2-D problem. However, the mean is not likely to be the same because the number of guesses needed to win a game is the minimum of the maximum of the number of guesses needed for any row in the set of rows and the maximum of the number of guesses needed for any column in the set of columns.

We wish to exploit the exchange of information between rows and columns provided by the 2-D structure that is absent from the 1-D problem. For example, if we discover that the color c_i does not appear in column j then we can eliminate all guesses which contain c_i in position j from the search space of every row. Additionally, if we discover the secret code for row k then we can fix the colors in position k for every column thus reducing the search spaces for each.

Consistency is the goal of our method. A row (column) is consistent if it has not yet been eliminated as possibility for that row (column) of the secret code by the feedback obtained from previous guesses. The consistency of the entire guess matrix could be defined similarly. However, we choose to define a consistent guess matrix as one in which each row and column is individually consistent within its own search space to better reflect our search method and its implementation. At each turn we wish to submit a guess which is fully consistent with all feedback received so far. We believe that the interaction between rows and columns will provide sufficient implicit constraints to guide our search without the need to generate a set of possible guesses and choose the best based on some additional criteria.

Tabu search was chosen because of its ability to make non-improving moves. From our choice of consistency as the primary factor in choosing a guess, moves through inconsistent space would be considered non-improving moves. As feedback is received from guesses, the remaining consistent search space becomes noncontiguous (that is, one cannot necessarily move from one consistent guess to another by making a single move). By allowing moves through inconsistent space (moves which result in an intermediate solution which is not consistent), the search can progress until it discovers a consistent region.

5.1 Preprocessing

The preprocess or color-counting stage is very similar to the 1-D version described in chapter 4. We begin by testing a matrix composed entirely of color c_1 to count the number of times this color appears in each row and column. We then proceed fill in the correct numbers of c_1 for each row and fill in the remaining stones with color c_2. This will guarantee that each row is consistent with the feedback of the first guess. However, the columns will not be consistent at this point. We then use our Tabu search to rearrange the rows until we create a guess which is consistent for all rows and columns.

The third color, c_3, is tested in a similar way. The correct numbers of c_1 and c_2 are added to each row, and the remaining stones are assigned color c_3. Tabu search is once again used to rearrange the rows until a fully consistent guess is found. The method continues in this way until all m (or possibly fewer) colors have been counted.

5.2 Tabu Search

The Tabu search is implemented in a very similar way to the one used in the previous chapter. First, a row is chosen at random. A pair of stones (which are not the same

color) is then chosen at random and the positions are swapped within that row. Unlike the local search, this Tabu search do not submit a guess after each pair swap. Instead, it continues to search until a consistent guess is found, and only then it the guess submitted.

The value of a guess is the sum of the number of consistent rows and columns. A count of the total number of consistent rows and columns is made after each swap. If a pair is not Tabu, then they are always swapped. If a pair is on the Tabu list, then the swap is only made if it increases the current sum of consistent rows and columns. Based on experimentation, the Tabu tenure was chosen to be $2r$.

Experimentation has shown that search times can be reduced by occasionally re-starting the current search from the best solution found since the previous guess was submitted. We have determined that resetting every $10^{(n-2)}$ moves works well.

5.3 Computer Implementations

Most of the data structures used are straightforward. The secret code and each guess can easily be saved in a (possibly sparse, if not a square game) n-length d-dimensional integer matrix. Integers are not space efficient, but are more easily manipulated. For study purposes, we choose to keep track of the remaining search space for each row and column.

The search space for a square d-dimensional game is m^k, where $k = n^d$, which grows exponentially with parameters. However, we adopt a reduced (for reasonable values of d, m, and n) mapping which has size $dn^{d-1}m^n$. In the 2-D *MM2(4,6)* game, this means that we use only 10,368 Boolean elements instead of 2.821E+12 elements. We accomplish this by not saving each possible guess matrix in the search space. Instead we save the search space for each row and column individually.

It is worth noting that while most papers present the colors as integers from 1 to *m*, we chose to represent them as integers from 0 to (*m - 1*) in the code. This is to save space in the Boolean vectors which represent the search space for an individual column or row. This way we can treat a guess as a base-*m* number, and can easily convert it to a base-10 number for use by the program.

The other data are simple to store. The parameters (*n, m, d*), number of guesses, black pegs, and white pegs are stored as integers. One may wish to keep the feedback counts as arrays in order to keep track of the changing values during the game (as in some implementations described in Chapter 3), but we do not.

For each entry in the Tabu list we choose to save the coordinates of the first D – 1 dimensions and the two positions for the last dimension which were switched. This represents a 2-swap move similar to that described in Chapter 4.

As with any computer implementation there are some issues which much be addressed which are independent of the problem. First, all runs were completed on the same machine so that the execution times could be compared. The times given in the results should only be compared to each other and not compared to results from the same program run on a different machine or a different implementation run on the same machine. Also, computers do not generate truly random numbers because these numbers are generated from mathematical functions. To improve randomness, we have chosen to seed the random number generator with the current time and date information at the beginning of every test case.

CHAPTER 6
EXPERIMENTATION

In this chapter we describe our computational experiments. We then compare the results of the five 1-D methods and examine the sensitivities to changes in problem parameters. Finally, we present the results of the Tabu search method used to solve the 2-D problem.

6.1 Experimental Design

As reflected in previous literature, the most logical parameters to manipulate are the values of m and n. The most common levels to test are *MM(4,6)* and *MM(5,8)*. However, to better understand how well the different methods scale with problem parameters, we choose to change one parameter while holding the other constant. For the first round of testing we keep $n = 4$ constant while varying the value of m. We then hold $m = 8$ constant and vary the value of n.

For the 1-D problem, we test all m^n possible codes for each combination. The exception is for the Tabu search at the level *MM(7,8)*. The run time was extrapolated to be prohibitively long, so we choose to generate a sample of 20,000 randomly generated test cases instead. Due to the incredibly large number of codes for the 2-D problem, we randomly generated 10,000 test cases for each level. An exception was made for *MM2(7,8)* where only 1,000 test cases were run due to the long run times. To improve the randomness of the test problems, we reseed the random number generator with the time and date before each one.

6.2 1-D Results and Discussion

In this section we examine and discuss the results of our computational experiments.

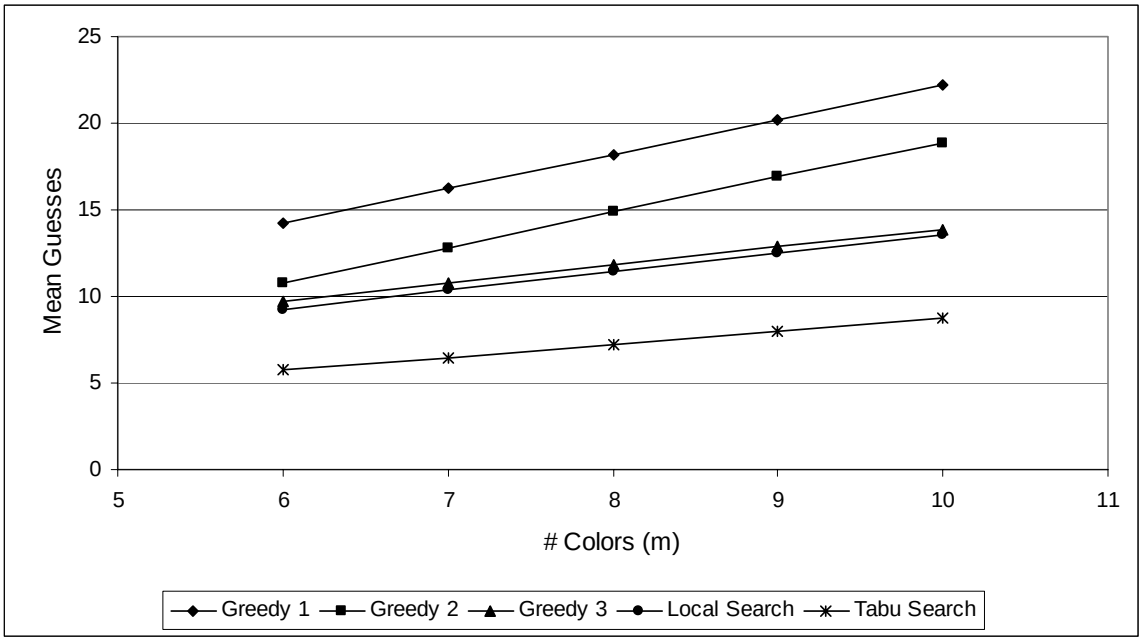

Figure 6-1**:** Mean number of guesses required to solve the 1-D problem (n = 4)

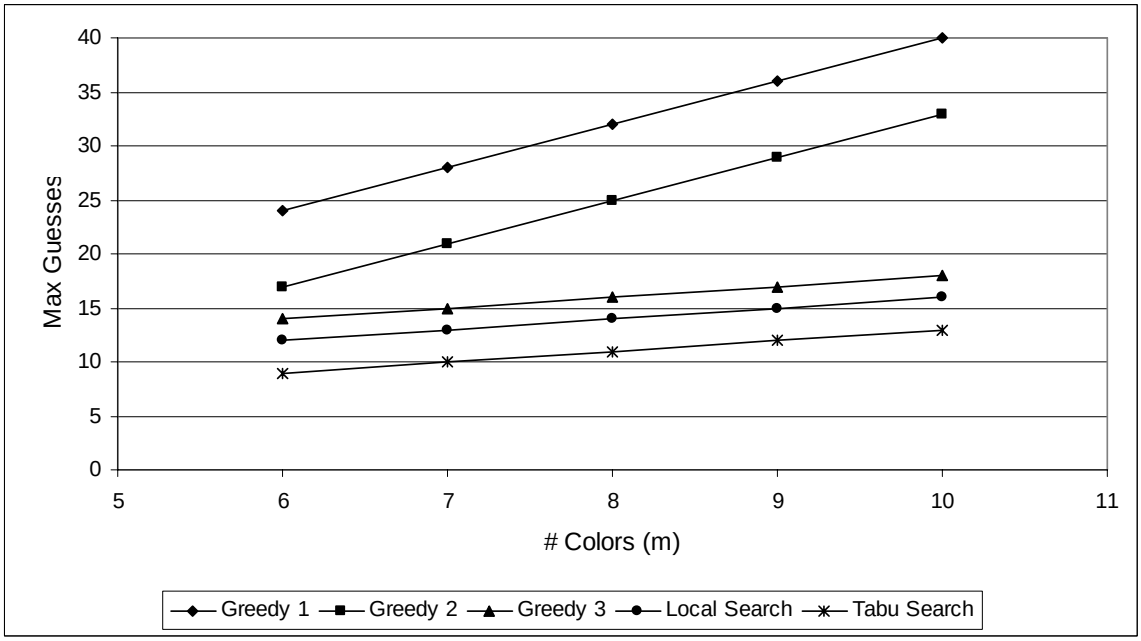

Figure 6-2: Maximum number of guesses required to solve the 1-D problem (n = 4)

We begin our experimentation be holding *n* constant and testing how performances of the different approaches scale with *m*. As Figures 6-1 and 6-2 show, for both the mean and maximum number of guesses the Greedy 1 method performed worst, followed by Greedy 2, Greedy 3, Local Search and finally Tabu Search. The average performance between Greedy 3 and Local Search is almost negligible. Unfortunately, all of the methods tested here performed worse than those presented in the literature (see chapter 3).

All of the solution methods scaled linearly with *m* in both measures. Both Greedy 1 and 2 had the parallel slopes for both measures. Also, Greedy 3, Local Search and Tabu Search had almost the same slope for both measures. This shows that the Greedy methods were more sensitive to changes in *m* than the other methods.

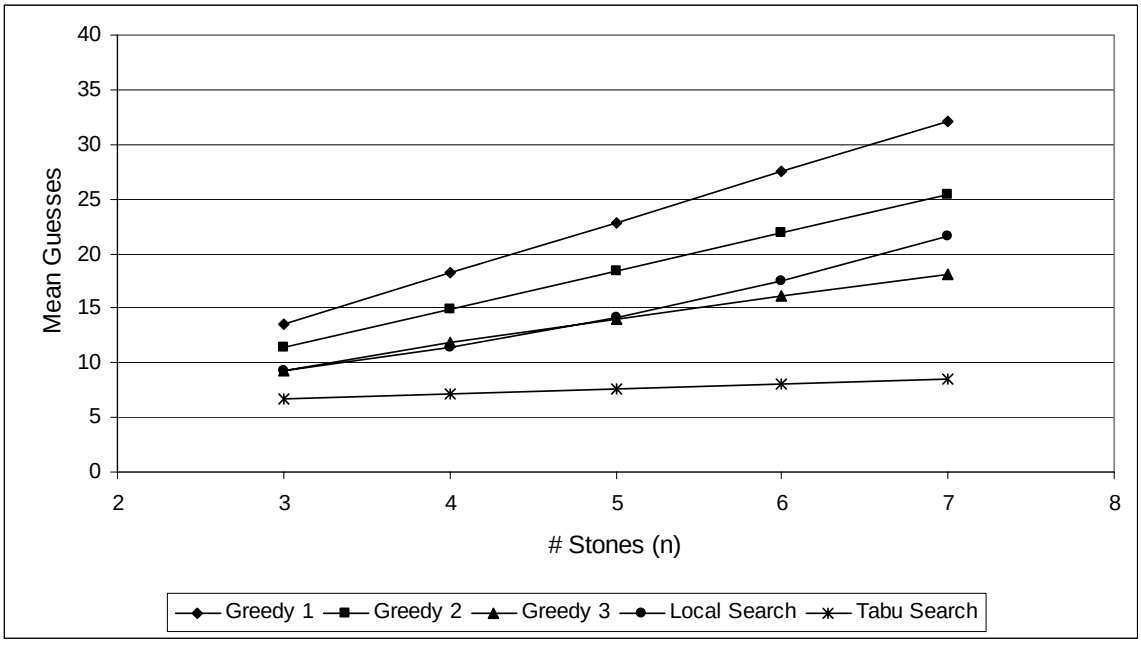

Figure 6-3: Mean number of guesses required to solve the 1-D problem (m = 8)

We now hold $m = 8$ constant and test how the performances of the different solution methods scale with *n*. As Figures 6-3 and 6-4 show, the ranking of the performances of the different methods is the same as for the previous tests with one exception. The Lo-

cal Search performs worse than Greedy 3 in the average number of guesses for larger problems. However, Local Search still performs better than Greedy 3 in the maximum number of guesses. We find that all Greedy methods and Local Search are more sensitive to changes in the parameter *n* than they were to changes in *m*.

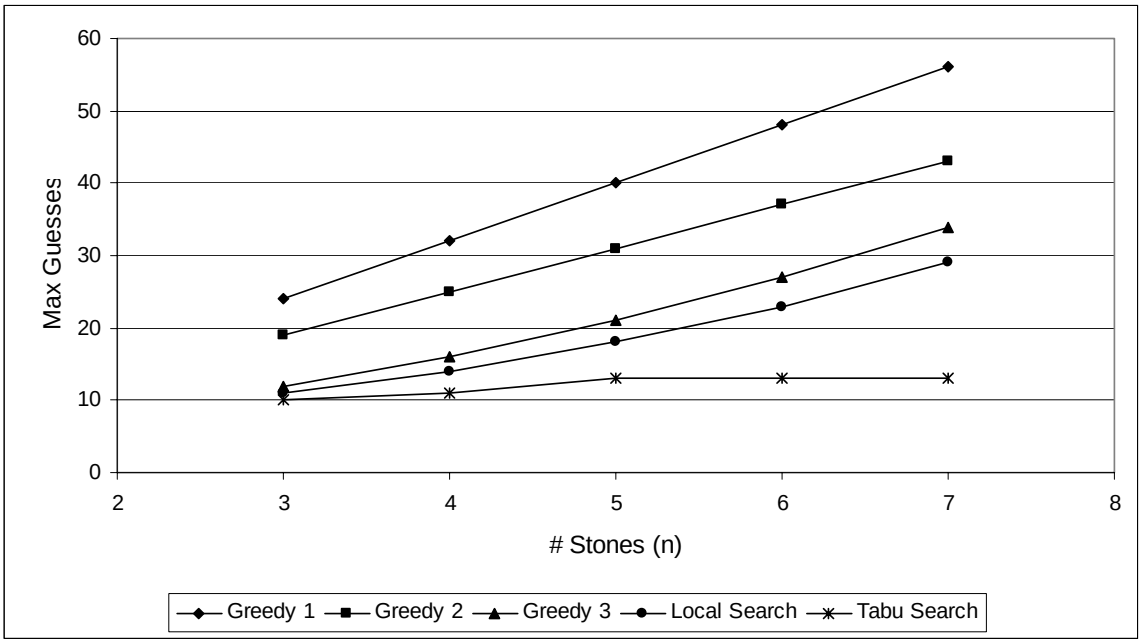

Figure 6-4: Maximum number of guesses required to solve the 1-D problem (m = 8)

Performances of the different methods appear to diverge as *n* increases. The regression slope for the Local Search is higher than that for Greedy 3 for the average number of guesses, but lower for the maximum number of guesses. Interestingly, only Tabu Search has a lower slope with regards to *n* than it did with regards to *m*, and thus was less sensitive to changes in this parameter.

Table 6-1: Run times for 1-D Tabu search method.

	N = 4, M = 6	*N = 4, M = 7*	*N = 4, M = 8*	*N = 4, M = 9*	*N = 4, M = 10*
Time Per Run (ms)	2.266	4.493	6.857	13.778	22.340

	N = 3, M = 8	*N = 4,M = 8*	*N = 5, M = 8*	*N = 6, M = 8*	*N = 7, M = 8*
Time Per Run (ms)	0.892	6.857	59.662	521.835	4382.850

The trade-off for the improved performance of the Tabu Search comes with respect to average run times to solve a single test problem. Times for the Greedy and Local Search methods were less than 1.0 ms for all levels of *MM(n,m)* tested. At that time scale exact measurements are not possible due to operating system overhead and other confounding factors. However, the run times for Tabu Search were significantly higher, as shown in Table 6-3. Run times grew exponentially, which could make Tabu Search a less feasible alternative as the problem size grows.

6.3 2-D Results and Discussion

We now examine the results of our Tabu Search solution to the 2-D problem. For comparison we include the results of the 1-D Tabu Search at the same levels of n and m (i.e. a single row of the 2-D problem).

We begin once again by holding $n = 4$ constant and varying the value of m. As Figures 6-5 and 6-6 show, the 2-D problem requires a higher average number of turns to solve, but a lower maximum number of turns to solve, than the 1-D problem. While we can be confident about the results of the average number of guesses, we must be more wary about the results of the maximum number of guesses. This is because the 1-D problem was solved for all possible codes, while the 2-D problem was tested on a set of randomly generated test cases. A single problem can easily change the maximum number of guesses required so far, but is unlikely to have a large effect on the average number.

We now conclude by holding $m = 8$ constant and varying the value of n. Figure 6-7 illustrates the surprising result that the average number of guesses required with respect to n grows more slowly for the 2-D problem than for the 1-D problem. Figure 6-8

shows a similar result for the maximum number of turns. However, this should be treated the same as Figure 6-6 with regard to the level of uncertainty.

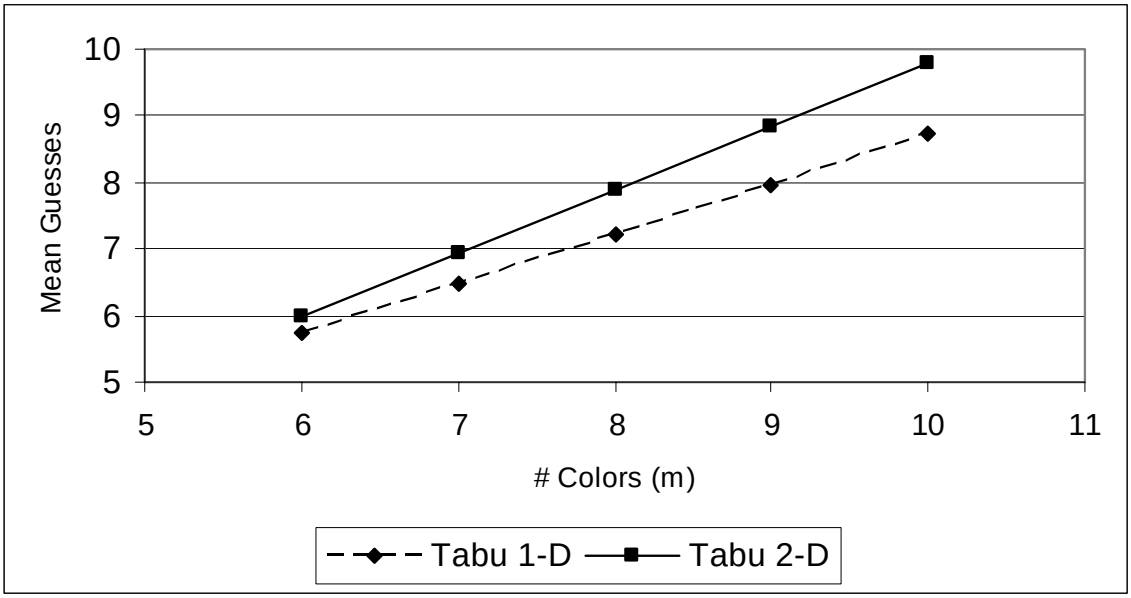

Figure 6-5: Mean number of guesses required to solve the 2-D problem (n = 4)

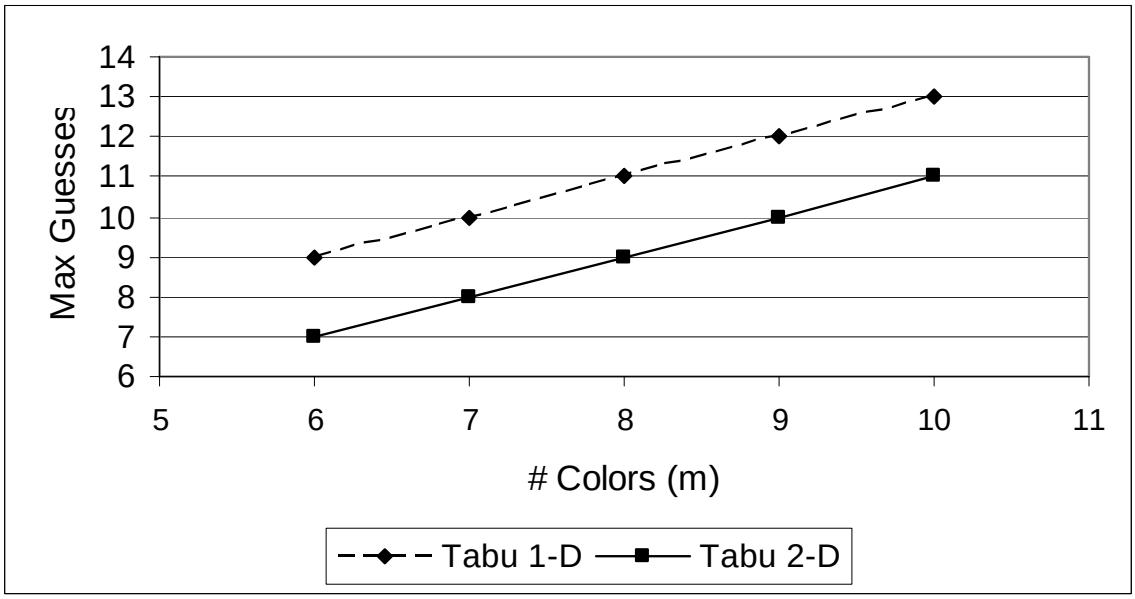

Figure 6-6: Maximum number of guesses required to solve the 2-D problem (n = 4)

Once again there is a trade-off between run time and the average and worst-case performances. Table 6-4 shows the average run times per test problem for the 1-D and 2-D problems in milliseconds. The time required for the 2-D problem is approximately one

to two orders of magnitude greater than that required for the 1-D problem. The run times grow so large that it became infeasible to run 10,000 tests for the *MM2(7,8)* case on our available equipment.

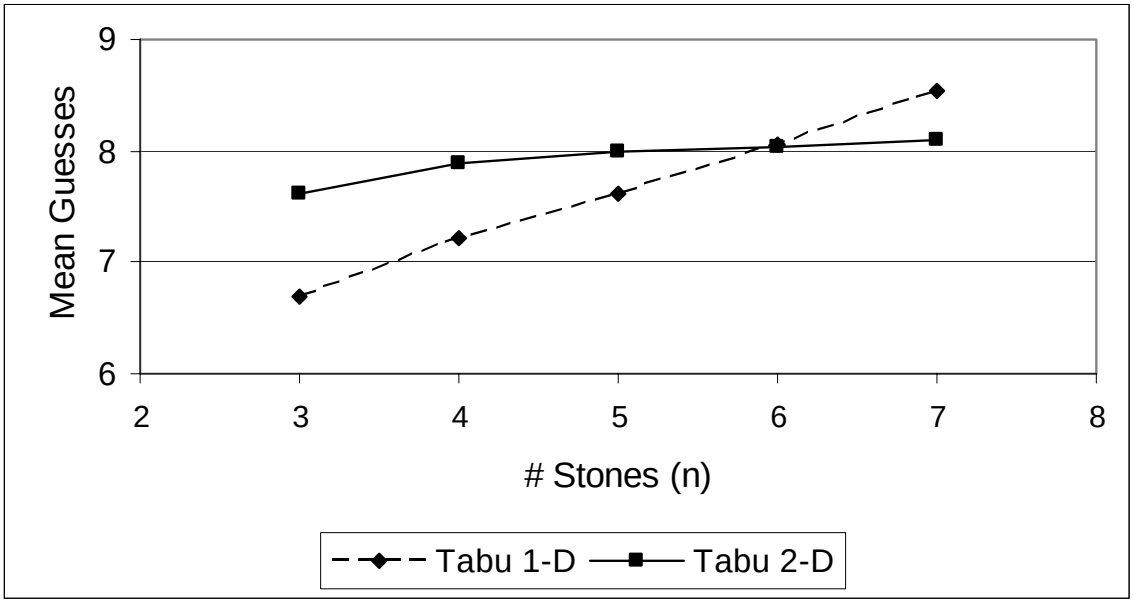

Figure 6-7: Mean number of guesses required to solve the 2-D problem (m = 8)

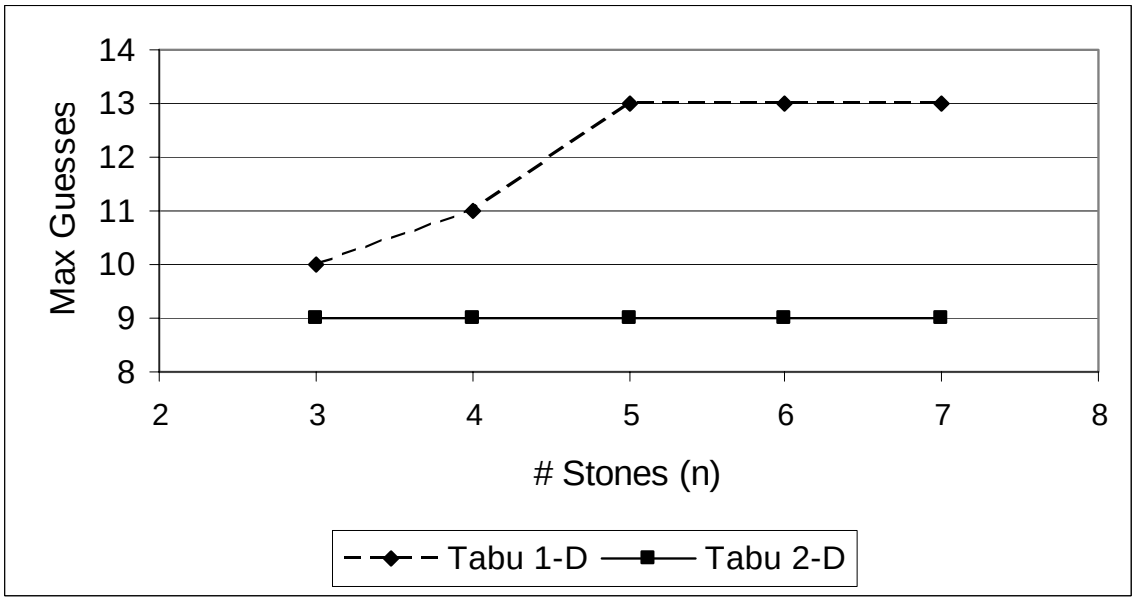

Figure 6-8: Maximum number of guesses required to solve the 2-D problem (m = 8)

Table 6-2: Run times for 2-D Tabu search method.

	N = 4, M = 6	N = 4, M = 7	N = 4, M = 8	N = 4, M = 9	N = 4, M = 10
1-D	2.266	4.493	6.857	13.778	22.340
2-D	218.4	300.5	377.5	524.900	773.800

	N = 3, M = 8	*N = 4, M = 8*	*N = 5, M = 8*	*N = 6, M = 8*	*N = 7, M = 8*
1-D	0.892	6.857	59.662	521.835	4382.850
2-D	9.100	377.500	5885.400	43264.039	395330.100

CHAPTER 7
CONCLUSION AND FUTURE WORK

In this chapter we begin by briefly discussing our conclusions. We then describe some future work which could be derived from the research presented in this study.

7.1 Conclusions

All of the solution methods we proposed we able to solve the 1-D MasterMind problem and did so in a reasonable amount of time. Performance for all methods grew linearly with the problem size despite the search space size growing exponentially. While none of the methods performed as well as those methods described in the literature, we believe that these methods still have room for improvement. The 1-D Tabu Search method performed sufficiently well that a Player could use it to win a traditional game of MasterMind with an imposed maximum allowed number of guesses (ten guesses in *MM(4,6)*).

We introduced the 2-D MasterMind problem and examined how it differs from the 1-D problem. The proposed Tabu Search proved even more effective at solving the 2-D in a minimum number of guesses than it did for the 1-D problem. Performance still grew linearly with problem size despite the search space size growing exponentially. However, there are some aspects of the 2-D problem which show additional room for improvement.

7.2 Future Work

Research can easily generate many questions along with answers. We now present a list of possible future research which extends naturally from this work.

1. **Improved preprocessing.** One of the major issues with the current preprocessing phase is that the overall strategy is known to be suboptimal, with a lower bound on the average number of guesses of 5.245. The strategy illustrated below, derived from (Knuth, 1977), should reduce the lower bound on the average to 4. Some additional logic will be needed to determine the number of each color.

   The first three guesses should be applicable as they appear, but the fourth guess may have to be conditional on the feedback from the first three. For example, the (row or column) guesses 1114 and 1156 would be indistinguishable given the four guesses below. This and similar situations can be handled by adding an appropriate conditional fourth guess. Additional information available from the orthogonal guesses can also be used to help determine which of the two might be correct.

1	1	2	2
1	1	2	2
2	2	1	1
2	2	1	1

Guess 1

3	3	1	1
3	3	1	1
1	1	3	3
1	1	3	3

Guess 2

4	4	5	5
4	4	5	5
5	5	4	4
5	5	4	4

Guess 3

6	6	4	4
6	6	4	4
4	4	6	6
4	4	6	6

Guess 4*

Figure 7-1: Proposed new preprocessing step

2. **Variations.** As mentioned in Chapter 2, there are many existing variations on the MasterMind problem. More importantly, the game will likely lend itself to many new variations as research continues.

3. **Higher Dimensions.** One of the most interesting aspects of MasterMind is the ease with which the game can be generalized to higher dimensions. It is expected that the average number of guesses will decrease as the number of dimensions increases. This is due to the additional information *per stone* which helps to reduce the possible search spaces for each of the possible rows, columns, etc. which contain that stone.

   Many possible questions remain to be answered here. What (if any) is the theoretical lower bound on the average and worst-case number of guesses as the number of dimensions, *d*, becomes very large? Will Tabu Search become considerably more effective in constructing fully-consistent guesses?

4. **New Solution Methods For 2-D MasterMind.** As of this writing there are no published works regarding the 2-D problem. We believe that our Tabu search results can provide an initial metric against which to test the efficacy of other solution methods. Possibilities which are expected to perform well include simulated annealing, genetic algorithms, and stochastic search.

REFERENCES

A. Bestavros, A. Belal. *MasterMind, a game of diagnosis strategies.* Bulletin of the Faculty of Engineering, Alexandria University, Alexandria, Egypt, 1986.

J. L. Bernier, C. Ilia Herraiz, J. J. Merelo, S. Olmeda, and A. Prieto. *Solving MasterMind using GAs and simulated annealing: a case of dynamic constraint optimization*. In Parallel Problem Solving from Nature IV, lecture notes in Computer Science 1141, Springer, Verlag (editors), 1996, 554-563.

V. Chvatal. *MasterMind*, Combinatorica 3 (1983), pp. 325-329.

Z. Chen, C. Cunha, S. Homer. *Finding a hidden code by asking questions.* COCOON`96, Computing and Combinatorics, J. Cai, C. K. Wong (editors), Hong Kong: Springer, 1996, 50-55.

M. de Bondt. Manuscript, Radboud Universiteit Nijmegen, Nijmegen, Netherlands (2005)

D. L. Greenwell. *MasterMind.* J. Recreational Mathematics 30 (1999-2000), 191-192.

R. W. Irving. *Towards an optimum MasterMind strategy*, J. Recreational Mathematics 11 (2) (1978-79) 81-87.

D. E. Knuth. *The computer as MasterMind*, J. Recreational Mathematics 9 (1) (1976-77) 1-6.

K. Koyama; T. W. Lai. *An optimal MasterMind strategy*. J. Recreational Mathematics 25 (1993), 251 - 256.

J. J. Merelo. *Genetic Mastermind, a case of dynamic constraint optimization*. GeNeura Technical Report G-96-1, Universidad de Granada, Granada, Spain, 1996.

J. J. Merelo, J. Carpio, P. Castillo, V. M. Rivas, G. Romero. *Finding a needle in a haystack using hints and evolutionary computation: the case of genetic MasterMind*. Late breaking papers at the GECCO99, A. Wu, S. Brave (editors), 1999, 184-192.

S. Reddi. *MathMind – Variations on MasterMind*. Unpublished paper, 2002. Retrieved 10 Oct. 2004 <http://www.rspq.org/pubs>

R. T. Rosu. *MasterMind.* Master's thesis, North Carolina State University, Raleigh, North Carolina, 1999.

A. Temporel. *A heuristic hill climbing algorithm for MasterMind.* Master's thesis, University of Bristol, Bristol, United Kingdom, 2003.

APPENDIX
STATISTICAL RESULTS

Greedy 1

	N = 4, M = 6	N = 4, M = 7	N = 4, M = 8	N = 4, M = 9	N = 4, M = 10
Mean	14.267747	16.238651	18.2146	20.194635	22.1779
Median	14	16	18	20	22
S.D.	3.380171	3.936171	4.505332	5.080817	5.65944
Maximum	24	28	32	36	40

	N = 3, M = 8	N = 4, M = 8	*N = 5, M = 8*	N = 6, M = 8	N = 7, M = 8
Mean	13.591797	18.2146	22.839325	27.464291	32.089286
Median	14	18	23	27	32
S.D.	3.972431	4.505332	4.97877	5.410493	5.810159
Maximum	24	32	40	48	56

Greedy 2

	N = 4, M = 6	N = 4, M = 7	N = 4, M = 8	N = 4, M = 9	N = 4, M = 10
Mean	10.80015432	12.833403	14.8571777	16.8750191	18.8889
Median	11	13	15	17	19
S.D.	2.700956497	3.3617557	4.001383	4.6316657	5.2541112
Maximum	17	21	25	29	33

	N = 3, M = 8	N = 4, M = 8	*N = 5, M = 8*	N = 6, M = 8	N = 7, M = 8
Mean	11.357422	14.8571777	18.35714722	21.857143	25.357143
Median	11	15	18	22	25
S.D.	3.519077	4.001383	4.441847824	4.8456787	5.218214
Maximum	19	25	31	37	43

Greedy 3

	N = 4, M = 6	N = 4, M = 7	N = 4, M = 8	N = 4, M = 9	N = 4, M = 10
Mean	9.6875	10.77676	11.828857	12.859168	13.8759
Median	10	11	12	13	14
S.D.	2.258991	2.277457	2.30808	2.346503	2.390203
Maximum	14	15	16	17	18

	N = 3, M = 8	N = 4, M = 8	*N = 5, M = 8*	N = 6, M = 8	N = 7, M = 8
Mean	9.322266	11.828857	14.018433	16.065845	18.157969
Median	10	12	14	16	19
S.D.	2.499247	2.30808	2.74636	3.562173	4.432365
Maximum	12	16	21	27	34

Local Search 1

	N = 4, M = 6	N=4, M = 7	N = 4, M = 8	N = 4, M = 9	N = 4, M = 10
Mean	9.266975	10.364015	11.436768	12.49276	13.5367
Median	9	11	12	13	14
S.D.	1.91116	1.952809	1.99118	2.027918	2.063927
Maximum	12	13	14	15	16

	N = 3, M = 8	N = 4, M = 8	*N = 5, M = 8*	N = 6, M = 8	N = 7, M = 8
Mean	9.298828	11.436768	14.112396	17.502697	21.650592
Median	10	12	14	18	22
S.D.	1.963888	1.99118	2.48626	3.1842529	3.9554807
Maximum	11	14	18	23	29

Tabu Search 1-D

	N = 4, M = 6	N=4, M=7	N=4,M=8	N=4, M=9	N=4, M=10
Mean	5.73611	6.46822	7.20825	7.962262	8.72864
Median	6	7	7	8	9
S.D.	1.040400131	1.15467	1.286423	1.442918	1.600704
Maximum	9	10	11	12	13
Time Per (ms)	2.26558642	4.49287797	6.857470703	13.77838744	22.34

	N=3, M=8	N=4,M=8	*N = 5, M = 8*	N = 6, M = 8	N = 7, M = 8
Mean	6.684375	7.20825	7.622833	8.0485992	8.54285
Median	7	7	8	8	8
S.D.	1.5089498	1.286423	1.170798	1.1283003	1.15764
Maximum	10	11	13	13	13
Time Per (ms)	0.891796875	6.8574707	59.66186523	521.8353271	4382.85

Tabu Search 2-D

	N = 4, M = 6	N=4, M=7	N=4,M=8	N=4, M=9	N=4, M=10
Mean	5.9724	6.9261	7.8809	8.8322	9.7814
Median	6	7	8	9	10
S.D.	0.27936722	0.3216351	0.3783246	0.43984373	0.497231097
Maximum	7	8	9	10	11
Minimum	4	5	5	5	6
Time Per (ms)	218.4	300.5	377.5	524.9	773.8

	N=3, M=8	N=4,M=8	*N = 5, M = 8*	N = 6, M = 8	N = 7, M = 8
Mean	7.6136	7.8809	7.9912	8.03627451	8.095
Median	8	8	8	8	8
S.D.	0.67907375	0.3783246	0.2431635	0.237871332	0.299094348
Maximum	9	9	9	9	9
Minimum	4	5	5	7	7
Time Per (ms)	9.1	377.5	5885.4	43264	395330.1

BIOGRAPHICAL SKETCH

Andrew Singley was born in Logansport, Indiana, in February of 1980. His family moved to central Florida when he was in elementary school. Following this move, he became enamored with the manned space program at NASA. He studied math and science during middle and high school in order to one day work at Cape Canaveral.

As an undergraduate at the University of Florida, he could not force himself to choose a single major, and decided to double-major in psychology and mathematics (with a minor in computer science.) This allowed him to learn about technical and non-technical areas of interest. During this period he also volunteered with local 501(c)3 non-profit groups.

After completing undergraduate studies, Andrew decided to take a year off to do something meaningful and to take a break before graduate school. He joined AmeriCorps to do a year of community service work in the Capital Region (DC, VA, WV, PA, OH). During this year he met his future fiancée, Samantha Till, who is from Palmer, Alaska.

Andrew returned to the University of Florida to continue his education, but again chose breadth of knowledge over depth. This time he pursued master's degrees in industrial engineering and business administration. While taking a class in heuristics taught by Dr. Elif Akçali, he worked on a term project which would later become this thesis.