Dylan Hutchison
CS 541 Artificial Intelligence
Midterm Project due 10 October 2012

Mastermind AI

Contents

Background, Definitions and Notation

Mastermind, derived from an earlier game called bulls and cows, has been one of the most popular games of study in the field of AI due to its large but not overwhelming solution space size. (Wikipedia) The main goal of the game is to guess a secret word (the target) in as few turns as possible. Before attempting to characterize and solve Mastermind, let's fix the game's notation and rules.

The two principle parameters of Mastermind are the length of the secret word, also known as the number of pegs or positions, and the number of digits each position can take on, also known as the number of colors. We will operate with 4 places and 10 possible digits, denoted '0'-'9'. No two digits in T_s may be the same. The pegs and colors naming refer to the board game versions of Mastermind. Call the secret word T_s and some guess for the secret word x. Let's refer to all possible secret words as τ.

Upon making a guess x_i against the target, the player will receive feedback (also called a response) in the form of $r_i = b.c$, where b is the number of *bulls* and c is the number of *cows*. The set of all possible responses is R; there are 14 of them, including 4.0 and excluding 3.1 (an impossible response because the 1 cow must be in the correct position if the other 3 digits are bulls in their correct positions). The board game version calls the bulls black pegs and the cows white pegs. I adopt the former naming because it is more intuitive. A bull signifies that a digit in the guess is present in T_s in the correct position while a cow signifies that a digit in the guess is present in T_s, but in a different position than is in the guess. The game ends when the player makes a guess and receives feedback 4.0.

As guesses are made and feedback received, we can construct a game *history* in the form of a list of turns where each turn is a pair of a guess and response (x_i, r_i). A possible two-turn history is [(0123, 2.0) (2145, 1.2)].

At the beginning of the game, all guesses τ are possible targets T_s. With feedback from each guess, we can reduce the number of possible guesses. Call guesses that could still be possible targets given some history *consistent* and denote the set of them as τ*. Given τ*, it is possible to divide τ* into up to |R| = 14 partitions, where each partition corresponds to the guesses ϵ τ* that remain consistent after receiving a response r ϵ R. We will elaborate on this further later in the algorithm discussion.

Inspired by (Slovesnov 4), I will denote a *transformation* from guess p to guess q as Ψ, consisting of a substitution (mapping of digits to digits) and a permutation (mapping of places to places). There are 10! possible substitutions and 4! possible permutations. Given a history h, if there exists a transformation from guess p to q that *respects* h, we can say that p ~ q (p is equivalent to q). Again, I will discuss the details of this equivalence class later on.

Environment Description

For Mastermind, the state has a single variable: whether the agent correctly guessed the solution or not.

- Partially Observable – responses provide limited feedback
- Single Agent (one could treat the codemaker as another agent, but this is silly since the codemaker randomly generates targets)
- Deterministic – Next state is completely determined by my action
- Episodic – the agent's experience is divided into episodes, in each of which it must decide what to guess next
- Static – the target code never changes
- Discrete – the target code has a finite domain

Agent Description

Fundamental agent function: $\text{f(percept history)} = \text{f(past guesses and responses)} = \text{the guess that will generate the most information}$

- Performance measure – the number of turns it takes to guess a codeword (for both a specific target code and the average value of all target codewords)

- Environment – just the codemaker
- Actuators – making a guess
- Sensors – the feedback received from the guess

Algorithm Overview

The game begins with a call to `playGame()`, possibly with a reference to a file containing precomputed information to aid later on in the game. The precomputed information is not strictly necessary, but it will speed the algorithm's operation later on.

The algorithm will always make a first guess of 0123. Because the target can be anything, it makes no difference what guess we use at first. We receive feedback from this guess: r_1.

Based on the feedback, the program fills the array `consisT` with all possible guesses that are consistent with the response the first turn (0123,r_1) if the guess were indeed the secret target. **The smaller the number of remaining consistent guesses, the closer the program is to uniquely identifying the target codeword.** When the size reaches 1, the program will guess the codeword on the next turn. Of course, it is perfectly possible that the program will guess the target solution before that happens, but if it does, it is due to luck.

After the first guess, the program will continue making the "best possible" guesses until it guesses the codeword (signified by receiving a response of 4.0). The true intelligence of the program is in the algorithm it uses to find the "best" guess given a history of past guesses and responses. In the following sections, I will exactly define my working definition of "best" and how to evaluate the worth of a guess. Then I will discuss how to prune away redundant and inferior guesses so that we can minimize the number of guesses we actually have to evaluate, speeding up runtime computation.

Note that the program only looks at the partitioning of the consistent set of solutions after making a single additional guess. This makes the program a **greedy local search** algorithm, in that it chooses the guess that provides the most information now without consideration of turns past the next. The algorithm is not optimal, but it is much faster than those that traverse the entire solution space. The method of finding the best guess acts as the heuristic for a guess in that it estimates the worthiness of a guess for selection.

Heuristics for the Best Guess

There are many well-studied methods of choosing what guess to make given a past history of guesses and responses. For an overview, see (Pepperdine 3). From the performance measurements of (Merelo, Mora and Cotta), the entropy method and most parts method seem to be most viable. Both methods examine the division of the current set of consistent solutions (`consisT`) into 14 partitions corresponding to the 14 different possible responses from guessing that guess.

Entropy

The entropy method is based on information theory. The entropy of a discrete random variable that can take on values x_1, ..., x_k with probability p_1, ..., p_k has entropy $H(X) = -\sum_{i=1}^{k} p_i \ln p_i$ (Wikipedia). One can think of entropy as the measure of the information content of a system. Take a look at these two extremes:

- All the guesses are in a single partition. The other partitions have no candidate guesses (and therefore we can never receive any other response). The entropy of this partition split is $-1\ln 1 = 0$. This kind of a partition split gives no information and it is the worst kind of split imaginable.
- The guesses are evenly divide among all 14 paritions, such that $p_i = \frac{1}{n}$, where n is the total number of guesses. It can be shown that this leads to a maximum entropy. If the base of the logarithm is n, then the entropy is 1; otherwise the entropy will differ by a constant multiple. Here is the derivation:

$$\max(H) = \min(\sum_{i=1}^{n} n_i \ln n_i) \quad \textbf{Take } \frac{d}{dn_i} = 0$$

$$\sum_{i=1}^{n} (\ln n_i + 1) = n + \sum_{i=1}^{n} \ln n_i = n + \ln \prod_{i=1}^{n} n_i = 0$$

$$b^{-n} = \prod_{i=1}^{n} n_i$$

This equation is satisfied when the base of the logarithm b = n and $n_i = \frac{1}{n}$ uniformly. In addition, we know this is a minimum because the second derivative is $\sum_{i=1}^{n} \frac{1}{n_i} > 0$.

When considering guesses to choose, what we're really interested in is the change in entropy (which will be positive) when dividing the consistent solution set into 14 partitions. This change is {entropy of all the consistent solutions together in 1 category} – {entropy of the consistent solutions divide among the 14 categories} = $\ln n - \frac{1}{n} \sum_{i=1}^{14} n_i \ln n_i$. We will choose the guess that has the highest increase in entropy. Of course, when it comes to actually computing the entropy of each guess, we only need to minimize $\frac{1}{n} \sum_{i=1}^{14} n_i \ln n_i$.

Entropy is a commonly used measure in machine learning to derive the criteria that divides test cases into categories such that the gain in information from the division is maximal. In that respect, the entropy approach makes this algorithm very similar to a machine learning algorithm.

Most Parts

The most parts method is simpler but equally valid. Count the number of nonempty partitions a guess divides `consisT` into. A guess that has a higher number of nonempty partitions is better than another guess, as it provides more classes to split the consistent guesses among n matter what response we receive.

Testing Results: Entropy Outperforms Most Parts

While the entropy and most parts methods are correlated and similar, they provide information on different levels that makes considering both of them worthwhile. Most parts is more coarse in that it considers the number of nonempty partitions and not the composition of each partition. Entropy, on the other hand, fully considers the composition of each partition, ranking those partition splits closer to the uniform partition split higher.

Because both methods are viable, let's combine the two instead of choosing one over the other. The question then becomes: which should we choose as the primary criterion for ranking guesses? One could either rank guesses by most parts and break ties by entropy or vice versa. As it is not immediately clear which way is better, I decided to try both methods, compute the average game length across all possible guesses for each and seeing which version would produce the lower average game length (see the method `computeAverageGameLength()` in the program). Here are the results:

		Avg. Game Length	
Response	Count	MostParts	Entropy
0.0	360	5.025	5.03056
0.1	1440	5.54722	5.54792
0.2	1260	5.43968	5.42222
0.3	264	5.0303	4.87879
0.4	9	3.55556	3.55556
1.0	480	5.00625	4.99792
1.1	720	5.20694	5.18333
1.2	216	4.96296	4.77315
1.3	8	3.75	3.75
2.0	180	4.67222	4.7
2.1	72	4.36111	4.36111
2.2	6	3.66667	3.66667
3.0	24	4.04167	4.04167
4.0	1	1	1
Overall Stats	5040	**5.26587**	**5.24286**

Count is the number of guesses that fall into the corresponding response class. Breaking the 5040 guesses into their respective response classes in this way helps one see the differences between different approaches. For example, the entropy first then most parts method has a lower average game length for the 0.2 response category but a higher length in the 0.0 response category.

The most important number, however, is the overall average game length across all 5040 guesses. Ranking by entropy first proves a better measure because it has a lower average game length. Furthermore, the expected game length is pretty good relative to the

Based on analyzing the best case partition splits for all possible responses, John Francis shows that the strict lower bound on the average game length is 5.12 (Francis 10). Considering that the program reaches an average game length of 5.24286 using only a 1-level-deep greedy local search method, the program stands well among other Mastermind AIs.

Reducing the number of guesses to evaluate

First, a caveat from (O'Geran, Wynn and Zhiglyavsky 37): “On closer inspection, it is clear that an inconsistent set can sometimes serve as a better test set than a consistent set.” In other words, we should not restrict our search for best guesses to the set of consistent guesses; there may very well be an inconsistent guess that provides more information than a consistent one!

While it is possible to evaluate all 10*9*8*7=5040 possible guesses at each turn and choose the best guess according to the method above, there is a clear waste in CPU cycles because we will be evaluating lots of guesses that we will end up not choosing. Can we identify some of these bad guesses beforehand without incurring more computational expense than it takes to just check the guess? Yes, in two ways:

Redundant Guesses

Given a past history of guesses and repsonses, many guesses will provide the same information. We call these guesses equivalent, and we can find equivalent guesses by searching for a transformation between them. The main thrust of this idea along with several improvements is found in (Slovesnov 4-6, 11).

Let’s illustrate this idea with examples. On our first guess, there is no past history. It’s reasonable to see that any guess at this point will convey the same information. By same information, I mean that the partition sets created by guessing the guess are the same. All guesses are equivalent.

Now suppose we make the guess 0123 and obtain some response. We have no information about the digits 4 through 9 because they are not contained in our first guess. Therefore, the guess 8645 will provide the same information as 4567, 9865, 7694, and all other guesses involving some permutation of the digits 4 - 9. The guess 2641 will provide a different set of information, however, because it contains the digits 1 and 2, which we have some knowledge about from our first guess.

Two guesses are equivalent if there exists a transformation, consisting of a substitution and a permutation, between them that respects the past history of guesses. By ‘respects’ I mean that the transformation must not change the past history of guesses. One can think of the transformation as a renaming and reordering of the digits of the guesses that does not affect the past history of guesses. We can always find a substitution and permutation taking us from one guess to another, but it is not guaranteed that the transformation will respect past history (this can be proven by exhaustion; just write a computer program to find the transformation between every possible pair of guesses).

If the responses in the past history are all unique, any transformation that respects the past must map each response to itself. Suppose instead we have a past history of [(0123,2.0) (2145,2.0)]. Any transformation Ψ that respects this history must satisfy either

Ψ(0123)=0123, Ψ(2145)=2145

or

Ψ(0123)=2145, Ψ(2145)=0123

Thus, in the case that we have repeated responses in the past history, the past history respecting restriction is a bit more lenient, meaning that it is easier to find transformations leading to more equivalence relationships ultimately reducing the size of the guesses we will have to evaluate. So incorporating runtime response information can help when we have repeated responses. When we can take advantage of this phenomenon, it is logged in the program output by the statement "Based on repeated responses in game history, reduced the number of representative guesses to __".

One can denote the existence of a past-respecting transformation between p and q as p ~ q because the relationship truly forms an equivalence class; it is reflexive, symmetric, and transitive. For further details of the properties of transformations, please see Slovesnov's paper.

We can construct the set of representative guesses by taking one member from each equivalence class of all equivalence classes present given a past history. The information we obtain from each response should differ, and so it is worthwhile to test each one. This lack of redundancy can speed things up a lot.

Computing the set of representative guesses, however, is an expensive process. It's better to do it in advance and save the results in a precomputed file properly indexed. The file can then be referred to by the program to lookup the set of representative guesses for a past history. When the program uses the file in such a way, it outputs the statement "From precomputed file: __ representative guesses (narrowed down from 5040)". Remember that if the past history contains repeated responses, we can further reduce the number of representative guesses for a bit of extra computation.

Finally, because the number of representative guesses we must evaluate grows at an exponential rate with the number of turns deep we want to process, this method is only useful up to and including turn 3. In addition, as the length of the past history increases, the payoff of finding the set of representative guesses is subject to diminishing returns; turn 1 can reduce the number of guesses we need to try to a single guess, turn 2 might reduce our guesses to try to the tens or hundreds, and in the worst case we may only be able to reduce our guesses to the thousands range. Fur subsequent turns, we need can still reduce the number of guesses to try using the methods below.

Approximating Redundant Guesses

There are two measures used to reduce the number of guesses without the aid of representative guesses: absent digits and uncalled digits (Slovesnov 10).

An absent digit is one we know is not present in the solution from a 0.0 response. There is no point in guessing a guess with an absent digit because it will not provide as much information as other guesses. We can prune those right away.

An uncalled digit is one that we have not guessed yet in the past. Guesses with uncalled digits in the same positions automatically belong to the same equivalence class, so we can prune all but one of each class. The class we keep is the one lexicographically smallest (when ordered by the natural ordering of digits).

Testing Results: Speedup[1]

To prove the effectiveness of using precomputed representative guesses and other redundant guess reduction techniques, I benchmarked the time it took to compute the average game length for both methods. When run with the techniques enabled, it took 950497 ms = 15.84161 minutes of CPU time (not real time) to finish playing the game over all 5040 possible guesses (on a regular, non-production build, it took 1202999 ms = 20.04998 min).

When run without any guess pruning at all such that the program evaluated all 5040 guesses before selecting the best one, it took a considerable 21140200 ms = 352.33667 minutes of CPU time, over 22x as long! The effectiveness of the aforementioned measures is clear.

Perhaps more importantly, the average game length results without the redundant and inferior guess pruning is *exactly the same* as including the pruning. This means that the pruning techniques never prune away the true best guess, proving their correctness.

Program Usage

SolveMaster.exe [--repGuessFile rep_guess_file] [--target codeWord | --computeAvgGameLength] [--benchmark]

Starts a new game with the specified target code (or a random one if unspecified) using the representative guess file. If computeAvgGameLength is specified, outputs a table with the average game length classified according to first response.

SolveMaster.exe --genRepGuess --repGuessFile rep_guess_file --depth depth_level [--benchmark]

Generates a representative guess file x levels deep and saves it to the file

Running the program is fairly straightforward from the information above. In addition to runtime switches there are several compiler options. To enable more levels of debugging output, compile the code with higher version identifiers.

- `-unittest` runs the unit tests to ensure the integrity of the utility functions
- `-version=0` has minimal output
- `-version=1` will output more details about guess selection, including every time a new best guess replaces an old one
- `-version=2` will provide details about every guess considered. Please redirect your standard output to a file when compiling with this switch.
- `-version=3` will include information about pruned guesses too.
- `-release -O -inline –noboundscheck` will compile the code in production mode, optimizing and inlining functions along with disabling runtime array bounds checking.

[1] I ran these measurements on a 64-bit Windows 7 laptop with an Intel Core i7 CPU rated at 1.73GHz. The number of cores doesn't matter because the benchmarking was single-threaded sequential.

Please do not run the program with a malformed representative guess file. I did not design the program for security.

The actual code of the program is a bit of a mess due to a bad habit of coding a method right next to where I want to use it instead of laying out the organization of methods ahead of time. In addition, when I need to code a method, I try to code it as generically as possible so that I can reuse it under similar circumstances. This leads to a seemingly random arrangement of the code that actually makes logical sense to me. Rely on the comments and names of the methods to guide you through interpreting my implementation.

Further Work

There are two main ways the program can be improved: in timing performance and game playing performance.

The program can solve any single game in a second or two, but it would be nice to obtain average game length results faster. Because each game play is independent of another and we must conduct all 5040 game plays, it is very reasonable to consider parallelizing the average game length computation. No inter-task communication = "embarrassingly parallel" = great speedup. With a bit of additional effort, parallelizing the algorithm should theoretically not be difficult.

To obtain better average game lengths, we could either search for better heuristics to choose guess that offers the best information gain across all possible solution codewords, or we could analyze information gain deeper than 1 turn. The former approach seems difficult as the entropy and most parts heuristics are regarded as the best heuristics known and they particularly for entropy they are well rooted in both theory and application.

We can extend the algorithm to take a minimax approach, analyzing the information gain $n > 1$ levels deep. Such an approach will deliver better results, but the added computational expense is considerable. If we had infinite computing resources, we could analyze every possible guess sequence from the initial guess to the end state solution for all possible target codewords. In the worst case, we may have to evaluate 5040^7 possible guess sequences. While we could reduce the number of evaluations through the alpha-beta pruning techniques previously mentioned, I am not sure if that will bring solving the Mastermind game exactly (without the use of approximating heuristics) into the feasible region. Rather than attempt it, I chose to implement an algorithm that will run many, many times quicker that obtains average game lengths within 0.15 turns of the proven ideal lower bound. To the ambitious, I challenge you to do better.

Works Cited

Francis, John. "Strategies for playing MOO, or "Bulls and Cows"." January 2010. <http://www.jfwaf.com/Bulls%20and%20Cows.pdf>.

Merelo, J. J., et al. "An experimental study of exhaustive solutions for the Mastermind puzzle." *arXiv* (2012). <http://arxiv.org/abs/1207.1315>.

O'Geran, J H, H P Wynn and A A Zhiglyavsky. "Mastermind as a Test-Bed for Search Algorithms." *Chance: New Directions for Statistics and Computing* 6.1 (1993). <http://www.cf.ac.uk/maths/subsites/zhigljavskyaa/pdfs/search/mastermind.pdf>.

Pepperdine, Andy. "The game of MOO." 29 January 2010. 3 October 2012. <http://www.pepsplace.org.uk/Trivia/Moo/Moo.pdf>.

Slovesnov, Alexey. "Search of optimal algorithms for bulls and cows game." 1 November 2011. *Bulls and cows game.* 3 October 2012. <http://bulls-cows.sourceforge.net>.

Wikipedia. *Mastermind (board game)*. 3 October 2012. 3 October 2012.

—. *Maximum entropy probability distribution*. 10 May 2012. 3 October 2012. <http://en.wikipedia.org/wiki/Maximum_entropy_probability_distribution>.