**The Computer Journal Advance Access published March 6, 2007**

doi:10.1093/comjnl/bxm006

# A Two-Phase Optimization Algorithm For Mastermind

SHAN-TAI CHEN[1], SHUN-SHII LIN[2,*] AND LI-TE HUANG[2]

[1]*Department of Computer Science, Chung Cheng Institute of Technology, National Defense University, Tao-Yuan, Taiwan, R.O.C.*

[2]*Graduate Institute of Computer Science and Information Engineering, National Taiwan Normal University, No. 88, Sec. 4, Ting-Chow Rd., Taipei, Taiwan, R.O.C.*

**Corresponding author: linss@csie.ntnu.edu.tw*

**This paper presents a systematic model, two-phase optimization algorithms (TPOA), for Mastermind. TPOA is not only able to efficiently obtain approximate results but also effectively discover results that are getting closer to the optima. This systematic approach could be regarded as a general improver for heuristics. That is, given a constructive heuristic, TPOA has a higher chance to obtain results better than those obtained by the heuristic. Moreover, it sometimes can achieve optimal results that are difficult to find by the given heuristic. Experimental results show that (i) TPOA with parameter setting ($k$, $d$) = (1, 1) is able to obtain the optimal result for the game in the worst case, where $k$ is the branching factor and $d$ is the exploration depth of the search space. (ii) Using a simple heuristic, TPOA achieves the optimal result for the game in the expected case with ($k$, $d$) = (180, 2). This is the first approximate approach to achieve the optimal result in the expected case.**

*Keywords: algorithm/Mastermind/search strategies/deductive game*

*Received 6 January 2006; revised 30 November 2006*

## 1. INTRODUCTION

Deductive games are *two-player zero-sum* games of *imperfect information*. Player I, called the codemaker, chooses a secret code. Player II, the codebreaker, does not know the choice player I made and has to guess the secret code. After each guess, Player II will get a hint about the accuracy of the guess from Player I. The goal of Player II is to discover the secret code, according to the hints, in the fewest number of guesses.

The game of Mastermind [1] is a well-known deductive game. In general, the Mastermind Satisfiability Problem is NP-complete [2]. A secret code consists of four pegs out of six possible colors. Repeated colors are allowed, so the number of possible secret codes is $6^4 = 1296$. A hint consists of black and white pegs; a black peg means that a peg in the codebreaker's guess is correct in both position and color; a white peg means that a peg in the guess is correct in color but not in position; and finally, no pegs means that there are no pegs in the guess, which are correct in color. Now we restate the game with a more precise description. The codemaker chooses a secret code ($s_1$, $s_2$, $s_3$, $s_4$). After each guess ($g_1$, $g_2$, $g_3$, $g_4$) made by the codebreaker, the codemaker responds with a pair of numbers [$B$, $W$]. The symbols, $B$ and $W$, denote the number of black pegs and white pegs, respectively. More precisely, $B = |\{i\colon s_i = g_i\}|$ and $W = \sum_{j=1}^{6} \min(p_j, q_j) - B$, where $p_j = |\{i\colon s_i = j\}|$ and $q_j = |\{i\colon g_i = j\}|$. For example, if the secret code is (1, 4, 4, 3) and the guesses are (3, 1, 5, 4) and (4, 1, 4, 5), then the responses are [0, 3] and [1, 2], respectively.

Merelo *et al.* [3] transferred the optimal strategy for Mastermind game to a combinatorial optimization problem. It resembles other computational problems such as circuit testing, differential cryptanalysis, on-line models with equivalent queries and additive search problems. For the example of differential cryptanalysis, playing the game is similar to submit combinations of letters in an alphabet to a 'black box' encrypting device, and the encrypted output is analysed to crack the key; the problem is to compute a set of combinations that allow players to extract maximal information from the 'black box'.

Over the past three decades, much research has been done on this kind of game. Knuth [1] demonstrated a strategy for the Mastermind game that requires at most five guesses in the worst case and 4.478 in the expected case. The strategy is to choose the guess that minimizes the maximum number of remaining candidates at every stage. Later, Irving [4] and Neuwirth [5] used sophisticated heuristic strategies to improve the bound in the expected case to 4.369 and 4.364, respectively. Finally, Koyama and Lai [6] used an exhaustive depth-first search on a supercomputer to determine the optimal strategy for Mastermind, where the expected number of guesses is 4.34. Variants of the Mastermind game have been studied in [7, 8]. Furthermore, in [3, 9, 10], the authors used *evolutionary algorithms* and *genetic algorithms* to solve related problems. Roche [11] proved that the number of guesses needed is $O(M(\log(\log M)))$, where $M$ is the number of pegs. Kabatianski, Lebedev and Thorpe [12] investigated the Mastermind game and its related applications based on coding theory. More recently, a graph-partition approach was introduced to determine the optimal strategies for various games with two-digit secret code [13]. Barteld [14] analysed four well-known strategies and presented a novel heuristic strategy for the game, where the expected number of guesses is 4.373.

Algorithms that deal with this kind of problems can be classified as either *complete* or *approximate* algorithms [15]. *Complete algorithms*, such as A* and branch-and-bound, are guaranteed to find an optimal solution; however, they might need exponential computation time in the worst case. This often leads to a high computation time for practical purposes. Thus, the use of approximate methods has received more and more attention in the last three decades. *Approximate algorithms* sacrifice the guarantee of finding optimal solutions for the sake of getting feasible solutions in a significantly reduced amount of time.

Approximate methods usually are distinguished between *constructive* methods and *local search* methods. Constructive methods generate solutions from the scratch by adding *components* (or called *moves*) until a solution is complete. On the other hand, local search methods start from some initial solution and iteratively try to replace the current solution by a better solution. However, both methods may easily be trapped into local optima.

To escape from local optima, a new kind of approximate algorithms has emerged in the past three decades. These algorithms try to combine basic heuristic methods in a higher-level framework aimed at efficiently and effectively exploring a search space. Examples of these algorithms based on *local search* are genetic algorithms [16], simulated annealing [17], Tabu search [18, 19], Ant Colony optimization [20] and iterated greedy [21]. On the other hand, examples of algorithms based on constructive methods are iterative sampling [22], HBSS [23], sampling and clustering [24], selective-sampling simulation [25], adaptive sampling [26, 27], GRASP [28, 29], block search [30] and Monte-Carlo tree search [31]. The main difference between these algorithms is the mechanisms used to guide the tree search. Due to the problem of huge search space, however, no approximate algorithm developed has achieved the optimal result for the Mastermind game in the expected case.

In this paper, we propose an approximate algorithm based on constructive methods, called two-phase optimization algorithms (TPOA). TPOA introduces a systematic mechanism to approach global optima for Mastermind.

This paper is organized as follows. Section 2 formulates the optimization problems for deductive games. The general structure of TPOA is introduced in Section 3. In Section 4, we demonstrate how to apply TPOA to solve the game of Mastermind. Comparison results for different methods are given in Section 5. Section 6 contains our concluding remarks.

## 2. OPTIMIZATION PROBLEMS FOR DEDUCTIVE GAMES

Optimization problems for deductive games are classified into two categories, i.e. to find an optimal strategy for the games in *the worst case* and in *the expected case*. Optimal strategies for these two cases can be defined as follows.

(i) *An Optimal strategy in the worst case* is a strategy that minimizes the maximum number of guesses required to discover an arbitrary secret code given by the codemaker.
(ii) *An Optimal strategy in the expected case* is a strategy that minimizes the total number of guesses required to discover all possible secret codes, assuming a uniform distribution over the possible secret codes.

We now introduce some properties of deductive games by a simple *number guessing game*, denoted $1 \times N$ games [32]. Figure 1 shows a game tree for a $1 \times 16$ game. From the game tree, we can easily obtain the following two observations, which show how to derive the number of guesses required in the two cases. These observations can be applied to analyse other deductive games.

***Observation 1.*** The number of guesses required *in the worst case* for a game is $n$, where $n$ is the height of the game tree, i.e. the length of a longest path from the root to a leaf in the game tree. For example, $n = 5$ in Fig. 1.

***Observation 2***. The number of guesses required *in the expected case* for a game is $L/N$, where $L$ is the *external path length* [33] of the game tree, i.e. the sum of the distances from the root to each leaf in the game tree. For example, in Fig. 1, $L = 1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 8 + 5 \times 1 = 54$ and the number of guesses required in the expected case is $L/N = 54/16 = 3.375$.

We now investigate how large the search space is for a deductive game. For a game tree with height $n$ and branching factor $b$, the search space will be $b^n$. For a deductive game, the branching factor for each *move* depends on the number of *possible responses* $r$ and *possible codewords* $c$ of the game. Accordingly, the branching factor $b = r \times c$, and hence, the search space for a game with $M$ pegs out of $N$ colors, i.e. an $M \times N$ game, is equal to $(r \times c)^n$, where $r = (M^2 + 3M)/2$ and $c = N^M$. Therefore, the search space for an $M \times N$ game is $(N^M(M^2 + 3M)/2)^n$. For example, the search space for the

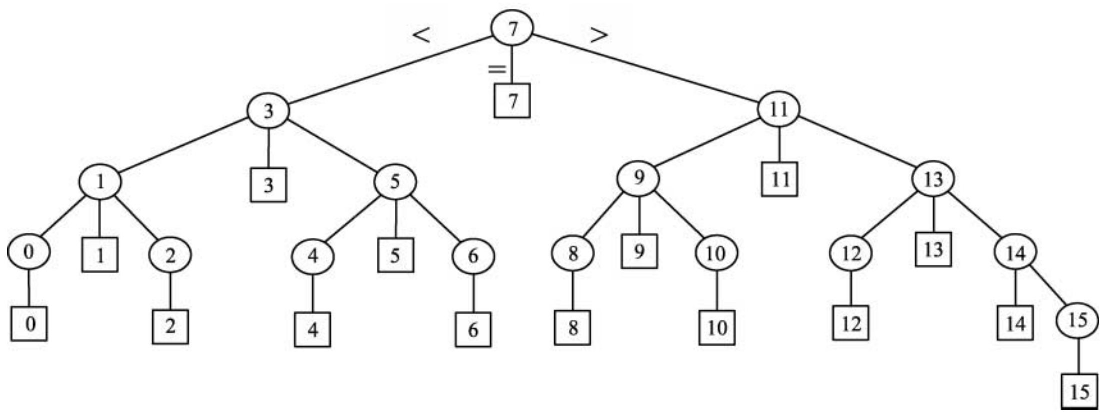

**FIGURE 1.** A game tree for a 1 × 16 game, where the binary search strategy is used [32].

game of Mastermind is $(6^4(4^2 + 12)/2)^n = (1296 \times 14)^n$, where $n = 5$ according to the strategy proposed by Knuth [1]. In Section 4, we will apply TPOA to solve deductive games. The search space will be reduced to a manageable size.

## 3. THE TWO-PHASE OPTIMIZATION ALGORITHM

TPOA is an approximate algorithm for solving Mastermind and is able to discover results with higher quality. We can also think of TPOA as a general improver for heuristic strategies. That is, given a heuristic, TPOA has higher chance to obtain results better than those obtained by the heuristic. Moreover, it sometimes can achieve optimal results that are difficult to find by the given heuristic.

### 3.1. Related research

The fundamental ideas of TPOA are (i) to combine *multi-way search* and *sampling* and (ii) the utilization of *clustering techniques*. Similar ideas have been employed in previous studies.

The sampling techniques are mechanisms trying to guide the tree search to areas that seem to contain promising solutions. In GRASP [28, 29], the next component is chosen at random from a *candidate list*, which keeps the best-ranked solution components according to a greedy function. HBSS [23] makes use of a node-ordering heuristic to guide search in a tree. The probability of choosing a next move is determined by a bias function. SAGE [26] uses random sampling techniques as a heuristic during beam search. The heuristic estimates the fitness of internal nodes by performing a random sampling from the node to a leaf. Selective-sampling simulation [25] and Monte-Carlo tree search [31] use on-line simulation results to guide the tree search. Adaptive probing [27] proposes *a learning model* during the search, which uses the gradient decent method to update the estimated cost of a leaf. Instead of using randomized-based [23, 26, 28, 29], simulation-based [25, 31] or learning-based [27] sampling schemes, TPOA uses *deterministic* heuristics cooperated with clustering techniques to guide the multi-way tree search. The higher-ranked branches will be explored during the search. Besides, the number of branches to be explored in TPOA could be tuned according to the solution quality required, as well as the time and space allowed.

Similar clustering techniques are employed in sampling and clustering [24] and block search [30]. Sampling and clustering [24] uses clustering rules based on the idea of distance between points, to identify clusters of initial points which would lead to the same local optimum. Block search [30] is a *complete* search algorithm, which takes advantage of the relative independence between parts of a solitaire card game. It improves on depth-first search by grouping several positions in a block, and searching only on the boundaries of the blocks. Rather than clustering rules [24] and independence analysis [30] required, TPOA could take advantage of the existing heuristics and design a corresponding hash function to perform the clustering.

### 3.2. The structure of TPOA

The search tree of TPOA, abbreviated to *TPOA tree*, is divided into two phases, exploration and exploitation. The objective of exploration phase is to discover promising *partial* solutions; on the other hand, the exploitation phase is to choose the way that leads each of the partial solution to a ‘best’ *complete* solution. Two parameters, the *branching factor k* and the *exploration depth d*, are used to decide how large the search space TPOA intends to explore. That is, the parameters determine how many potential (promising) solutions that TPOA will exploit. We denote TPOA with the branching factor $k$ and the exploration depth $d$ as TPOA$^+$ $(k, d)$. The TPOA$^+$ $(k, d)$ tree is shown in Fig. 2a. Given a TPOA tree with arbitrary height $n$, after level $d$ the algorithm does a greedy search from that node on. The number of potential solutions exploited in a TPOA$^+$ $(k, d)$ *tree* will be $k^d$.

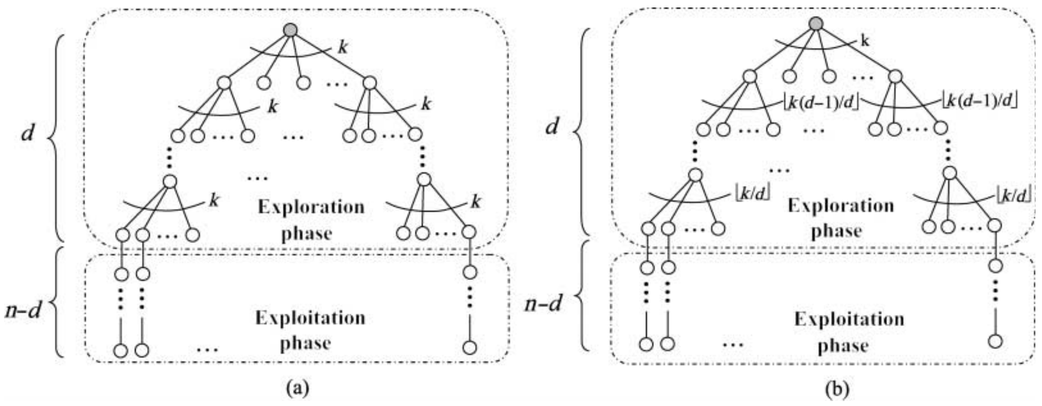

**FIGURE 2.** Search trees of different constructions for TPOA. (a) TPOA$^+$ ($k$, $d$) tree. (b) TPOA*($k$, $d$) tree. The number of solutions exploited are $k^d$ and $k^d \times d!/d^d$, respectively.

Taking advantage of the concept of 'annealing', the branching factor $k$ in TPOA could be evenly decreased at each descending level. We define this construction of algorithm as TPOA* ($k$, $d$). That is, in TPOA* ($k$, $d$) the branching factors at levels 1, 2, …, $i$, …, and $d$ are $k$, $\lfloor k \times (d-1)/d \rfloor$, $\lfloor k \times (d-2)/d \rfloor$, …, $\lfloor k \times (d-i+1)/d \rfloor$, …, and $\lfloor k \times 1/d \rfloor$, respectively. Therefore, the number of potential solutions exploited by TPOA* ($k$, $d$) is

$$\prod_{i=1}^{d}\left(\left\lfloor \frac{k \times i}{d} \right\rfloor\right) \le \prod_{i=1}^{d}\left(\left\lfloor \frac{k \times i}{d} \right\rfloor\right) = k^d \times \frac{d!}{d^d}, \qquad (1)$$

which is much smaller than $k^d$. The search tree for TPOA* ($k$, $d$) is shown in Fig. 2b. The two constructions of TPOA have been implemented for solving the game of Mastermind. A comparison of experimental results for these two constructions will be given in Section 5.

We now describe the structure and properties of TPOA. Given the parameters ($k$, $d$), the sketch of a recursive procedure for TPOA is shown in Fig. 3.

TPOA can be implemented by a *modified exhaustive depth-first search* on a TPOA tree. The main modification to depth-first search is that at each visited node in the exploration phase (with depth $d$), we consider only $b$ branches and ignore other branches. In Fig. 3, lines 3–5 show the variations of branching factor $b$ in TPOA$^+$ and TPOA*. In the exploration phase, TPOA$^+$ has a fixed $b$ ($=k$), but the value of $b$ in TPOA* is evenly decreased at descending levels, as shown in lines 3 and 4. In the exploitation phase, both TPOA$^+$ and TPOA* have a fixed $b = 1$, as shown in line 5. Therefore, TPOA$^+$ ($k$, $d$) [or TPOA*($k$, $d$)] is able to prune a huge search space

```
    TPOA(k, d, b, c) {                                  // k, d: the given constants
1   l = Current_level();                                // get the current level in the TPOA tree
2   If (c is a complete solution) then return c;
3   +If (l < d) then b = k;                             // in the exploration phase, this statement + is for TPOA+
4   *If (floor(( k × l ) / d)<k) then b = (floor(( k × l ) / d));   // this statement * is for TPOA*
5       else b = 1;                                     // in the exploitation phase
6   For (each move m ∈ M)                               // M: the set of all next potential components
7       i = Hash(m);                                    // classify possible next moves to HCGs by a hashing
8       HCG_i ← HCG_i ∪ {m};                            function
9   B = {HCG_j | HCG_j is the top b groups that could obtain promising results};
10  For (each HCG_i ∈ B)                                // B: the set of b selected HCGs
11      c_i = Choose(HCG_i);                            // c_i: the selected representative for HCG_i
12      C = C ∪ { c_i };                                // C: the set of b representatives c_i in B
13  S ← ∅;                                              // S: the set of potential solutions from descendant nodes
14  For (each c_i ∈ C)                                  // recursively b-way search to find the best solution from
15      s_i = TPOA(k, d, b, c_i);                       descendant nodes
16      S ← S ∪ { s_i };                                // select the best solution discovered in S
17  c = Max_{s_i ∈ S} (eval(s_i));                      // return c to the parent node
18  return c;
    }
```

**FIGURE 3.** The sketch of TPOA.

to a manageable size $k^d$ (or $k^d \times d!/d^d$) as shown in Fig. 2. For Mastermind, since the 14 response nodes at each level should be kept, the search space is reduced to $(14\ k)^d$ or $(14)^d \times k^d \times d!/d^d$.

Given two constants ($k$, $d$), the time complexity of TPOA$^+$ ($k$, $d$), in terms of number of nodes exploited, is $k^d$ $(n-d)$, where $n$ is the height of the game tree, i.e. the number of guesses required in the worst case. Roche [11] proved that the number is O($M$(log(log$M$))), where $M$ is the number of pegs. This means that no matter how large an instance of a problem is given, TPOA can always obtain an approximate result by appropriately selecting the parameters ($k$, $d$). Furthermore, depending on the execution time and space allowed, we can increase the value of parameters ($k$, $d$) to approach the optimal result. Now, we summarize the fundamental components of TPOA as follows: *a constructive heuristic* for the problem at hand; *a hash function* according to the heuristic; *two parameters* ($k$, $d$) to decide how large the search space TPOA intends to explore.

### 3.3. Hash collision groups

In TPOA, how to select the (most likely) best $b$ next potential components is a critical issue. The problem can be effectively and efficiently solved by a *clustering approach*. TPOA performs clustering using a concept of *hash collision groups* (HCG) [34]. The next potential components of solutions with similarity are clustered together in an *HCG* by a given *hash function* to the problem at hand. That is, the potential components with the same hash value will be clustered together. Sections 4 and 5 will give detail examples of how the clustering mechanism works. We now describe properties of *HCGs*. Figure 4 illustrates the relation between *HCGs* and equivalent classes in a search space of next potential components. There are several advantages of using *HCGs* in TPOA. The important properties of *HCGs* include:

(i) For two components in the same *HCG*, they are most likely equivalent. On the other hand, for two equivalent components, they are definitely in the same *HCG*.
(ii) Given a hash function, it is efficient to obtain the $b$ best *HCGs*.
(iii) Without losing the generality, an arbitrary component can be chosen to represent its *HCG*.

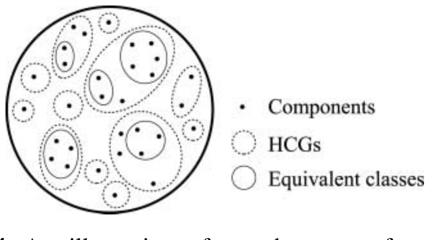

**FIGURE 4.** An illustration of search space of next potential components.

Therefore, TPOA is able to efficiently and effectively select the $b$ 'best' representatives among all next potential components. On the other point of view, if an evaluation function is used in TPOA, each HCG can be regarded as a set of the next potential components which have a tie on the return value of the function. Note that most ties are equivalent but equivalent solutions will produce ties.

## 4. TPOA FOR THE GAME OF MASTERMIND

In this section, we will apply TPOAs to a popular deductive game, Mastermind. Two hash functions are designed for optimization problems of the game in the worst case and the expected case, respectively. Furthermore, both TPOA$^+$ ($k$, $d$) and TPOA* ($k$, $d$) are applied to solve the problems.

### 4.1. TPOA for the game in the worst case

From Observation 1, we have to minimize the height of the game tree so as to obtain the optimal strategy for the game in the worst case. To achieve this goal, we employ the following simple heuristic, which is modified from Knuth's [1].

*Heuristic for the game in the worst case*. Minimize the number of remaining candidates for the largest response classes after each guess. There are 14 response classes for the game as shown in Table 1.

According to the heuristic, we design the corresponding hash function as follows:

*Hash function for the game in the worst case*. Let the number of remaining candidates of the 14 response classes after a guess $g$ be $C_g = \langle C_{g,1}, C_{g,2}, \ldots, C_{g,14}\rangle$. We define the *hash function*:

$$\begin{aligned}&\mathrm{Hash_w}(C_g = \langle C_{g,1}, C_{g,2}, \ldots, C_{g,14}\rangle)\\&\quad = (C'_g = \langle C'_{g,1}, C'_{g,2}, \ldots, C'_{g,14}\rangle),\end{aligned} \tag{2}$$

where $C'_{g,1} \geq C'_{g,\ 2} \geq \ldots \geq C'_{g,14}$. That is, the hash function sorts the original sequence $C_g$ into a nonincreasing sequence $C'_g$. For the example in Table 1, if the guess $g = (0, 0, 1, 2)$, we have $C_g = \langle 1, 20, 5, 40, 105, 4, 84, 230, 182, 2, 44, 222, 276, 81\rangle$ and $C'_g = \langle 276, 230, 222, 182, 105, 84, 81, 44, 40, 20, 5, 4, 2, 1\rangle$. If two nonincreasing sequences $C'_g$ and $C'_h$ are the same, that is, $C'_{g,1} = C'_{h,1}, \ldots$, and $C'_{g,14} = C'_{h,14}$, then the guess $g$ and the guess $h$ are classified into the same HCG.

Table 1 shows the number of remaining candidates of the 14 response classes after the first guess. Note that after the first guess, all the 1296 candidates will *automatically* be classified into only five HCGs by the hash function Hash$_\mathrm{w}$. In addition, all the candidates in the same HCG are equivalent. Unfortunately, after the second guess, the situation becomes more complicated. That is, some guesses that are not equivalent are possibly classified into the same HCG. However, we

**TABLE 1.** The number of remaining candidates in each class after the first guess [14]

| | Class | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Guess | [4,0] | [3,0] | [2,2] | [2,1] | [2,0] | [1,3] | [1,2] | [1,1] | [1,0] | [0,4] | [0,3] | [0,2] | [0,1] | [0,0] | Number of parts |
| 0000 | 1 | 20 | 0 | 0 | 150 | 0 | 0 | 0 | 500 | 0 | 0 | 0 | 0 | 625 | 5 |
| 0001 | 1 | 20 | 3 | 24 | 123 | 0 | 27 | 156 | 317 | 0 | 0 | 61 | 308 | 256 | 11 |
| 0011 | 1 | 20 | 4 | 32 | 114 | 0 | 36 | 208 | 256 | 1 | 16 | 96 | 256 | 256 | 13 |
| 0012 | 1 | 20 | 5 | 40 | 105 | 4 | 84 | 230 | 182 | 2 | 44 | 222 | 276 | 81 | 14 |
| 0123 | 1 | 20 | 6 | 48 | 96 | 8 | 132 | 252 | 108 | 9 | 136 | 312 | 152 | 16 | 14 |

also guarantee the fundamental properties that (i) for two components in the same HCG, they are most likely equivalent, and that (ii) for two equivalent components, they are definitely in the same HCG. Therefore, we can arbitrarily choose a guess to represent its HCG, rather than exhaustively explore all guesses in the HCG, and obtain an approximate result.

### 4.2. TPOA for the game in the expected case

In this case, we will minimize the external path length of the game tree to achieve the optima. The simple heuristic used in this case is Barteld's 'most-parts strategy' [14].

*Heuristic for the game in the expected case.* Maximize the number of parts caused by partitioning the remaining candidates according to the guess. As shown in Table 1, the number of non-zero parts from guesses 0000, 0001, 0011, 0012 and 0123 are 5, 11, 13, 14 and 14, respectively. The guess chosen in the 'most-parts' heuristic is 0012 or 0123. Note that the two guesses are in the same HCG. In reality, 0012 is the best guess.

The corresponding hash function can be defined as follows:

*Hash function for the game in the expected case.* Let the number of remaining candidates of the 14 response classes after a guess $g$ be $C_g = \langle C_{g,1}, C_{g,2}, \ldots, C_{g,14}\rangle$ and let $|C_g| = \sum_{i=1}^{14} x_{g,i}$, where $x_{g.i} = 1$ if $C_{g,i} > 0$ and $x_{g,i} = 0$ otherwise. We define the following hash function.

$$\mathrm{Hash_e}(g) = |C_g|. \tag{3}$$

## 5. EXPERIMENTAL RESULTS

In this section, we first compare TPOA against A* algorithms to show (i) the effectiveness and (ii) the ability to discover optimal solutions of TPOA. Then, the comparison between previous results and our results for the Mastermind problem is given. All experiments were run on a Pentium IV 1.6 GHz computer.

In our experiment, an A* strategy has been developed to solve the Mastermind problem in the expected case. It evaluates nodes by combining $g(n)$, the path length from the root of the game tree to node $n$, and $h(n)$, the external path length of the 'full' subtree rooted at node $n$. The full subtree guarantees that after each guess, there is a remaining candidate in the class [4, 0]. It is easy to show that $h(n)$ is *admissible* [35]. That is, the solution obtained by the A* strategy must be an optimal one.

Comparison results for the two algorithms in solving a test case, the 3 × 5 Mastermind problem, are given in Table 2.

**TABLE 2.** Comparison results for the A* strategy and TPOA for 3 × 5 Mastermind

| Experiments | | External path length ($L$) | The number of guesses in the expected case ($L/5^3$) | The number of guesses in the worst case ($n$) | Within % of the optima in the expected case ($451/L$) (%) | Run time (s) |
|---|---|---|---|---|---|---|
| **A*** | | **451** | 3.608 | 5 | 100 | **125.34** |
| | $(k, d) = (1, 1)$ | 460 | 3.680 | 5 | 98.043 | 0.01 |
| | $(k, d) = (5, 2)$ | 460 | 3.680 | 5 | 98.043 | 0.02 |
| TPOA* with | $(k, d) = (10, 2)$ | 460 | 3.680 | 5 | 98.043 | 0.04 |
| $\mathrm{Hash_w}$ | $(k, d) = (20, 2)$ | 452 | 3.616 | 5 | 99.778 | 0.09 |
| | $(k, d) = (30, 2)$ | 452 | 3.616 | 5 | 99.778 | 0.15 |
| | $(\boldsymbol{k}, \boldsymbol{d}) = \mathbf{(40, 2)}$ | **451** | 3.608 | 5 | 100 | **0.23** |

The optimal solution for the game in the expected case is $L = 451$, which can be discovered both by the A* strategy and TPOA*(40, 2). The A* strategy spends much longer time than TPOA*(40, 2) does, so TPOA is quite efficient. However, it is worth to mention that TPOA cannot guarantee to yield the optimal strategies even though a large parameter setting $(k, d)$ is given. The performance of TPOA critically depends on the choice of heuristic and the corresponding hash function to the problem at hand.

A comparison between previous results and our results for Mastermind is shown in Table 3. We performed four series of experiments denoted as series (1–4). From the experimental results, we have the following observations.

(i) In series (1), when $(k, d) = (1, 1)$, our result is optimal in the worst case, i.e. $n = 5$, and is within 97.38% of the optima, which is better than the result [1] in the expected case.

(ii) In series (4), when $(k, d) = (5, 2)$, the result $L = 5647$ is within 99.61% of the optima, which outperforms the best results of the previous heuristic strategy [5].

(iii) In series (1) and (2), since the hash function $\text{Hash}_w$ is tailored to the game in the worst case, all experiments achieve the optima, i.e. $n = 5$, for the game in the worst case.

(iv) On the other hand, in series (3) and (4), all experiments obtain better results in the expected case than those obtained by the corresponding experiments in series (1) and (2). For the example of $(k, d) = (1, 1)$, the experiment in series (3) obtains $L = 5674$, which is smaller than 5776 obtained by the experiment in series (1).

(v) Compared to $\text{TPOA}^+$ $(k, d)$, TPOA* $(k, d)$ exploits much smaller search space (and hence spends less time) but the quality of results is comparable to those obtained by $\text{TPOA}^+$ $(k, d)$, particularly for experiments in series (3) and (4). For example, TPOA*(30, 2) and $\text{TPOA}^+$ (30, 2) discover the same result, i.e. $L = 5632$, but the latter takes much longer time than the former does.

**TABLE 3.** A comparison among different strategies for Mastermind

| Previous methods and our experiments | | External path length ($L$) | The number of guesses in the expected case ($L/6^4$) | The number of guesses in the worst case ($n$) | Within of the optima in the expected case (**5625**/$L$) (%) | Run time (s) |
|---|---|---|---|---|---|---|
| Knuth [1] | | 5803 | 4.4776 | **5** | 96.9326 | N/A |
| Barteld [14] | | 5668 | 4.3735 | 6 | 99.2414 | N/A |
| Neuwirth [5] | | 5656 | 4.3642 | 6 | 99.4519 | N/A |
| Koyama [6] | | 5625 | 4.3403 | 6 | 100 | N/A |
| (1) $\text{TPOA}^+$ with $\text{Hash}_w$ | $(k, d) = (1, 1)$ | **5776** | **4.4568** | **5** | **97.3857** | 2.12 |
| | $(k, d) = (5, 2)$ | 5749 | 4.436 | 5 | 97.8601 | 20.58 |
| | $(k, d) = (10, 2)$ | 5711 | 4.4066 | 5 | 98.4941 | 71.24 |
| | $(k, d) = (20, 2)$ | 5669 | 4.3742 | 5 | 99.2238 | 263.29 |
| | $(k, d) = (30, 2)$ | 5659 | 4.3665 | 5 | 99.3992 | 589.63 |
| (2) TPOA* with $\text{Hash}_w$ | $(k, d) = (1, 1)$ | 5776 | 4.4568 | 5 | 97.3857 | 2.12 |
| | $(k, d) = (5, 2)$ | 5760 | 4.4444 | 5 | 97.6563 | 14.37 |
| | $(k, d) = (10, 2)$ | 5716 | 4.4105 | 5 | 98.408 | 40.06 |
| | $(k, d) = (20, 2)$ | 5676 | 4.3796 | 5 | 99.1015 | 137.32 |
| | $(k, d) = (30, 2)$ | 5665 | 4.3711 | 5 | 99.2939 | 297.56 |
| (3) $\text{TPOA}^+$ with $\text{Hash}_e$ | $(k, d) = (1, 1)$ | **5674** | 4.3781 | 6 | 99.1364 | 2.08 |
| | $(k, d) = (5, 2)$ | 5643 | 4.3542 | 6 | 99.6810 | 17.29 |
| | $(k, d) = (10, 2)$ | 5640 | 4.3519 | 6 | 99.7340 | 59.81 |
| | $(k, d) = (20, 2)$ | 5634 | 4.3472 | 6 | 99.8403 | 226.59 |
| | $(k, d) = (30, 2)$ | **5632** | 4.3457 | 6 | 99.8757 | **504.71** |
| (4) TPOA* with $\text{Hash}_e$ | $(k, d) = (1, 1)$ | 5674 | 4.3781 | 6 | 99.1364 | 2.06 |
| | $(k, d) = (5, 2)$ | **5647** | 4.3573 | 6 | 99.6104 | **12.24** |
| | $(k, d) = (10, 2)$ | 5641 | 4.3526 | 6 | 99.7164 | 34.10 |
| | $(k, d) = (20, 2)$ | 5637 | 4.3495 | 6 | 99.7871 | 119.59 |
| | $(k, d) = (30, 2)$ | **5632** | 4.3457 | 6 | 99.8757 | **255.18** |

(vi) Compared with the latest published *genetic algorithm* [10], TPOA obtains higher quality of results both in the worst and expected cases of the game. In [10], although the average run time is shorter, the average number of guesses is 4.75, which is larger than 4.4568 obtained by TPOA$^+$ (1, 1). Furthermore, the number of guesses in the worst case has no upper bound by using genetic algorithms.

Now, TPOA$^+$ (180, 2) with Hash$_e$ has achieved the optimal result, $L = 5625$, for the game in the expected case in 1464.69 s. This is the first approximate approach to obtain the optimal result (to our knowledge). According to the optimal game tree developed above, we have also implemented an interactive program for the game of Mastermind, which is available on the website [36]. For the game with larger size, $5 \times 7$ Mastermind, TPOA* (30, 2) with Hash$_e$ has obtained approximate results within about 27 hours, where the number of guesses required for the worst and expected cases are 7 and 5.1381, respectively.

From the above observations, we conclude some important characteristics of TPOA as follows.

(i) The larger the parameter setting ($k$, $d$), the more accurate the solution is. Tuning the parameter setting ($k$, $d$) is able to balance the tradeoff between the run time and the solution quality.
(ii) As parameters ($k$, $d$) increase, the TPOA tree becomes larger. Any kind of conventional tree pruning algorithms can be applied to TPOA.
(iii) TPOA could be applied to some other combinatorial optimization problems, e.g. the set covering problem, provided that an appropriate heuristic and the corresponding hash function for each of the problems is given.

## 6. CONCLUDING REMARKS

In this paper, we propose a systematic model, called TPOA. TPOA is able to obtain (near-)optimal results that cannot be discovered by the previously published heuristic strategies. Compared with genetic algorithms, TPOA obtains higher quality of results both in the worst and expected cases for the game of Mastermind. Moreover, TPOA achieves the optimal result for the game in the expected case, where $L = 5625$. This is the first approximate approach to obtain the optimal result (to our knowledge).

TPOA could be applied to related problems with larger sizes, which are difficult to solve by complete algorithms within a reasonable time. Depending on the characteristics of each problem, how to define an appropriate *hash function* for the problem at hand is a critical issue. Furthermore, with the excellent ability to discover solutions with diversity, TPOA could cooperate with other *local search* methods, such as *genetic algorithms* and *Tabu search*. This could be further studied in the future.

## 7. ACKNOWLEDGEMENTS

This research was supported in part by a grant NSC94-2213-E-003-004 from National Science Council, R.O.C. We thank the anonymous referees for providing constructive comments and help in improving the contents of this paper.

## 8. REFERENCES

[1] Knuth, D.E. (1976) The computer as Mastermind. *J. Recr. Math.*, **9**, 1–6.

[2] Stuckman, J. and Zhang, G.Q. (2006) Mastermind is NP-Complete. *INFOCOMP J. Comput. Sci.*, **5**, 25–28.

[3] Merelo, J.J., Carpio, J., Castillo, P., Rivas, V.M. and Romero, G. (1999) Finding a needle in a haystack using hints and evolutionary computation: the case of genetic Mastermind. *Genetic and Evolutionary Comput. Conf. Late Breaking Papers Books*, Orlando, Florida, July, 13–17, 184–192. Morgan Kaufmann, San Francisco.

[4] Irving, R.W. (1978–79) Towards an optimum Mastermind strategy. *J. Recr. Math.*, **11**, 81–87.

[5] Neuwirth, E. (1982) Some strategies for Mastermind. *Z. Oper. Res.*, **26**, 257–278.

[6] Koyama, K. and Lai, T.W. (1993) An optimal Mastermind strategy. *J. Recr. Math.*, **25**, 251–256.

[7] Ko, K.I. and Teng, S.C. (1986) On the number of queries necessary to identify a permutation. *J. Algorithms*, **7**, 449–462.

[8] Flood, M.M. (1988) Sequential search strategies with Mastermind variants – Part 1. *J. Recr. Math.*, **20**, 105–126.

[9] Bento, L., Pereira, L. and Rosa, A. (1999) Mastermind by evolutionary algorithms. *Proc. 1999 ACM Symp. Applied Computing*, San Antonia, Texas, 28 February–2 March. 307–311. ACM Press, New York.

[10] Kalisker, T. and Camens, D. (2003) Solving Mastermind using genetic algorithms. *Lect. Notes Comput. Sci.*, **2724**, 1590–1591.

[11] Roche, J. R. (1997) The value of adaptive questions in generalized Mastermind. *Proc. 1997 IEEE Int. Symp. Information Theory*, Ulm, Germany, 29 June–4 July. 135–135. IEEE Press, New Jersey.

[12] Kabatianski, G., Lebedev, V. and Thorpe, J. (2000) The Mastermind game and the rigidity of the Hamming space. *Proc. 2000 IEEE Int. Symp. Information Theory*, Sorrento, Italy, 25–30 June, pp. 375–375. IEEE Press, New Jersey.

[13] Chen, S.T. and Lin, S.S. (2004) Optimal algorithms for $2 \times n$ Mastermind games–a graph-partition approach. *Comput. J.*, **47**, 602–611.

[14] Barteld, K. (2005) Yet another Mastermind strategy. *Int. Comput. Games Assoc. J.*, **28**, 13–20.

[15] Blum, C. and Roli, A. (2003) Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Comput. Surv.*, **35**, 268–308.

[16] Holland, J.H. (1975) *Adaptation in Natural and Arstificial Systems*. University of Michigan Press, Ann Arbor.

[17] Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P. (1983) Optimization by simulated annealing. *Science*, **220**, 671–680.

[18] Glover, F. (1986) Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, **13**, 533–549.

[19] Glover, F. (1990) Tabu search – Part II. *ORSA J. Comput.*, **2**, 4–32.

[20] Dorigo, M. and Gambardella, L.M. (1997) Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Trans. Evol. Comput.*, **1**, 53–66.

[21] Ruiz, R. and St Äutzle, T. (2007) A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *Eur. J. Oper. Res.*, **177**, 2033–2049.

[22] Harvey, W.D. and Ginsberg, M.L. (1995) Limited discrepancy search. *Proc. Fourteenth Int. Joint Conf. Artificial Intelligence*, Québec, 20–25 August, pp. 607–613. Morgan Kaufmann, San Francisco.

[23] Bresina, J.L. (1996) Heuristic-biased stochastic sampling. *Proc. Thirteenth Nat. Conf. Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conf.*, Oregon, Portland, 4–8 August, pp. 271–278, AAAI Press, Menlo Park.

[24] Colorni, A., Dorigo, M., Maffioli, F., Maniezzo, V., Righini, G. and Trubian, M. (1996) Heuristics from nature for hard combinatorial optimization problems. *Int. Trans. Oper. Res.*, **3**, 1–21.

[25] Billings, D., Papp, D., Peña, L., Schaeffer, J. and Szafron, D. (1999) Using selective-sampling simulations in poker. *Proc. AAAI Spring Symp. Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, Stanford, CA, USA, March, 13–18. AAAI Press, Menlo Park.

[26] Juillé, H. and Pollack, J.B. (1998) A sampling-based heuristic for tree search applied to grammar induction. *Proc. 15th Natl Conf. Artificial Intelligence*, Wisconsin, 26–30 July, 776–783. American Association for Artificial Intelligence, Menlo Park.

[27] Ruml, W. (2001) Incomplete tree search using adaptive probing. *Proc. 17th Int. Joint Conf. Artificial Intelligence*, Washington, 4–10 August, Seattle, pp. 235–241. Morgan Kaufmann, San Francisco.

[28] Feo, T.A. and Resende, M. G. C. (1995) Greedy randomized adaptive search procedures. *J. Glob. Optim.*, **6**, 109–133.

[29] Pitsoulis, L.S. and Resende, M.G.C. (2002) Greedy randomized adaptive search procedure. In Pardalos, P. and Resende, M. (eds), *Handbook of Applied Optimization*. Oxford University.

[30] Helmstetter, B. and Cazenave, T. (2003) Searching with analysis of dependencies in a solitaire card game. In van den Herik, H.J., Iida, H. and Heinz, E.A. (eds), *Advances in Computer Games 10*, Kluwer Academic Publishers, Netherlands.

[31] Coulom, R. (2006) Efficient selectivity and backup operators in Monte–Carlo tree search. *Proc. Fifth Conf. Computers and Games*, Italy, May, 29–31, Springer-Verlag, New York.

[32] Chen, S.T. and Lin, S.S. (2004) Optimal algorithms for 2 × n AB games – a graph-partition approach. *J. Inf. Sci. Eng.*, **20**, 105–126.

[33] Sedgewick, R. (1988) *Algorithms* (2nd edn). Addison-Wesley, Boston.

[34] Chen, S.T. (2004) On the study of optimization algorithms for deductive games and related problems. PhD Dissertation, National Taiwan Normal University, Taipei, R.O.C.

[35] Russell, S. and Norvig, P. (2003) *Artificial Intelligence: A Modern Approach* (2nd edn). Prentice-Hall, New Jersey.

[36] Lin, S.S. (2005). Mastermind Games & AB Games. Available at: http://www.csie.ntnu.edu.tw/~linss/deductivegame/index.htm.