Using logic to win a Dating Game Show

Algorithmic techniques for solving permutation mastermind and the related "Are you the one?" variant.

Denise Graafsma

January 2022

1 Introduction

In the game show "Are you the one?", twenty contestants have to work together to find their perfect match and win a cash prize. Ten men and ten women have been paired up by production beforehand, they have ten weeks to find out who their partner is. At the end of each week, the contestants form couples during the matching ceremony. After they have all chosen a partner, it is revealed how many of the pairings are correct. In order to win, they have to find all the perfect matches by the end of the tenth week.

Before each matching ceremony, the contestants also have the opportunity to send one couple into the "truth booth". This booth confirms whether the pairing is a perfect match or not.

While the show itself encourages the contestants to play using their heart, we take a more logical approach. In this report, we present several algorithmic techniques for both Are you the one? and the related game of permutation mastermind.

1.1 Mastermind

The game played in Are you the one? (AYTO) is closely related to the well-known code-breaking game mastermind. Mastermind is played by two people, a codemaker and a codebreaker. The codemaker chooses a secret code consisting of four coloured pegs, which the codebreaker tries to guess in as few turns as possible. There are six colours for the codemaker to choose from and repetition is allowed. On each turn, the codebreaker guesses a code and receives feedback from the codemaker, which consists of black and white pegs. A black peg is given for each peg that is correct in both colour and position and a white peg for each guess that is correct in colour but not in the right position. This process of guessing and receiving feedback continues until the codebreaker receives four black pegs.

1.2 Permutation Mastermind

Since its invention, many different variants of the game have been created. For example, static mastermind where all the guesses have to be done at once, without receiving feedback in between, and the codebreaker has to deduce the secret code from all the feedback.

But the variant that is most closely related to AYTO is permutation mastermind. In this variant, all available colours must be included and no repetition is allowed. In other words, the secret code is a permutation of the n available colours. Since all the colours in the code are already known, the answer to each query only consists of black pegs: the number of colours that are in the correct position. In Section 2, we will discuss how AYTO can be modelled as a permutation mastermind variant.

1.3 Outline of this report

In this report, we will examine four different algorithms: two for permutation mastermind and two for the AYTO variant. For permutation mastermind, we will discuss an existing binary search algorithm, and present our own algorithm based on a swap operation in Section 4. The latter will also be extended to the AYTO variant in Section 5. In addition, we contribute a second algorithm for this variant by introducing an inference table.

All four algorithms were implemented and tested for secret codes of different lengths. We will present our results in Section 6 and discuss our findings in Section 8.

2 Problem Description

We can model the different pairings of the twenty contestants as permutations of ten different numbers. If the men and women are both numbered from one to ten, we consider the men the positions and the women the number in the permutation. So for permutation $(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$, woman 10 is paired with man 1, woman 9 with man 2, and so on. The aim of the game now becomes to uncover a secret permutation $s = (s_1, s_2, \ldots, s_{10})$ of the numbers 1 to 10. Each matching ceremony, the contestants decide on a query $q = (q_1, q_2, \ldots, q_{10})$ that is also a permutation of the numbers 1 to 10 and are given an answer $a(q, s)$. This answer is a number from 0 to 10, indicating how many of the numbers in q are in the correct position:

$$a(q, s) = |\{i \in [1, n] | q_i = s_i\}|. \tag{1}$$

We will refer to these correct entries as hits. The game is won when $a = 10$ and all the women are with the correct man.

Without the addition of the truth booth, the game is equivalent to permutation mastermind with 10 different colours.

The truth booth takes a number k and corresponding position i and returns whether s has k in position or not:

$$b(i,k) = \begin{cases} 1 & \text{if } s_i = k \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

3 Related Literature

Many algorithms for solving mastermind have been published. In this section, we will discuss the most relevant results. Permutation mastermind, on the other hand, has not been studied as extensively. To the best of our knowledge, only two algorithms have been published for this mastermind variant, both employing a binary search technique. We will discuss both of these algorithms.

3.1 Mastermind

In [2] Knuth presents an algorithm for the classic version of mastermind, with six colours and a secret code length four. It requires a maximum of five turns to win, and is often cited as the first published algorithm for solving mastermind.

The algorithm is derived in the following manner. After every turn, a test code is chosen that minimizes the maximum number of remaining possibilities over all possible feedback from the codemaker. If several codes qualify as the next test code, preference is given to a "valid" code, that does not contradict any of the earlier feedback. If this still leaves multiple options, the first in numerical order is selected.

The resulting strategy is given in the form of a table where the next test code can be determined by the number of possibilities still available and the feedback from the current test code.

Rao [4] published an algorithm for solving mastermind that keeps track of the remaining possible positions of a colour. For each colour, first its frequency and then the corresponding positions are determined. For example, if we know 1 appears once in the secret code, we can determine the frequency of 2 by guessing $(1,2,2,2)$. In addition, if we do not have any white pegs, we know 1 is in the correct position. Otherwise, our next guess would be $(3,1,3,3)$ to test for the position of 1, and the frequency of 3 simultaneously. Suppose we find this is indeed the correct position for 1, then the possible positions of 2 and 3 are updated to be $(2,(1,3,4))$ and $(3(1,3,4))$, meaning both 2 and 3 cannot be in the second position anymore.

3.2 Permutation mastermind

Ko and Teng [3] introduced the permutation mastermind variant and presented an algorithm that identifies a permutation of size n by $O(n \cdot \log_2 n)$ turns. The algorithm uses a binary search technique to find all the hits in a query.

Suppose we have a query q with a correct entry which that not yet been identified. The algorithm then creates a new query with the first half consisting of entries known to be wrong and the second half identical to q. From the answer, we can derive whether the correct entry was in the first or second half of q. This process is then repeated, but with only half of the section of q known to contain the correct entry. By repeatedly halving the number of possible hits, we end up with only one option, which we then know to be a hit.

To find a wrong section for the search, the algorithm requires some extra testing involving swapping two entries. El Ouali and Sauerland [1] improved on this by cleverly choosing a so-called pivot that separates the wrong sections and the section identical to q. We will further discuss their algorithm in Section 4.1.

4 Permutation mastermind algorithms

We present two different algorithms for solving permutation mastermind. The algorithm from [1] employs a binary search and identifies the secret code in $O(n \cdot \log_2 n)$ turns. We also present our first own contribution: the swap search algorithm. This algorithm requires $O(n^2)$ turns to identify the secret code, but was found to be more suitable for the AYTO variant than the binary search algorithm.

4.1 Binary search

In [1] an algorithm for permutation mastermind using a binary search is presented. Let n be the number of colours and $s = (s_1, s_2, \ldots, s_n)$ the secret code. The algorithm starts by querying for ten different permutations q^1 to q^n, with

$q^1 = (1, 2, \ldots, n)$

$q^2 = (n, 1, 2, \ldots, n-1)$

$q^3 = (n-1, n, 1, 2, \ldots, n-2)$

$\ldots$

$q^n = (2, 3, \ldots, n, 1).$

For each of these queries, the algorithm determines the number of hits $a(q^j, s)$. It then keeps track of how many of these still need to be identified. We define a partial solution $p = (p_1, p_2, \ldots, p_n)$, where $p_i = s_i$ when we have identified s_i and $p_i = 0$ when s_i is still unknown. Since $a(q, p)$ returns the entries for which $q_i = p_i$, this is the number of hits in q that have already been identified. When can then define the number of unidentified hits in q as

$$a(q, s, p) := a(q, s) - a(q, p). \tag{3}$$

We should note that, since p is known, a(q,p) can be determined without using any turns.

For the binary search, we need we need a pair of consecutive queries that satisfy $a(q^j, s, p) > 0$ and $a(q^{j+1}, s, p) = 0$, or $a(q^n, s, p) > 0$ and $a(q^1, s, p) = 0$. We will only discuss the former case, as the latter results in a very similar procedure. For the sake of readability, we define $r := j + 1$. In addition to this pair of queries, we require an entry c, referred to as the pivot, for which the position has already been identified: $p_i = c$ for some i. We denote the position of c in q^j and q^r as l_j and l_r, respectively. The aim of our search is to find the index m of an unidentified hit in q^j: $q^j_m = s_m$ and $p_m = 0$. We first construct a new permutation

$$q^{j,0} = (c, (q^r)_{i=2}^{l_j}, (q^j)_{i=l_j+1}^{n}).$$

Since $a(q^r, s, p) = 0$, the first l_j entries of $q^{j,0}$ cannot contain any unidentified hits. Thus, if $a(q^{j,0}, s, p) > 0$ we know $m > l_j$. On the other hand, if $a(q^{j,0}, s, p) = 0$, then we must have that $m \leq l_j$. Suppose we find the latter, then for our next query we define

$$q^{j,l} = ((q^j)_{i=1}^{l-1}, c, (q^r)_{i=l+1}^{l_j}, (q^j)_{i=l_j+1}^{n})$$

where $l = \lceil l_j/2 \rceil$. This time, we know only the first $l - 1$ entries can possibly contain an unidentified hit. Hence, if $a(q^{j,l}, s, p) > 0$ then $m < l$ and if $a(q^{j,l}, s, p) = 0$ we have $l \leq m \leq l_j$. By repeating this process, we continue halving the section containing a hit until we have identified m.

Since this search requires a pivot c that has already been identified, a modified version is used to find the first hit. We will not go into detail but want to specify that it contains a swap operation that inspired the swap search algorithm discussed in the next section.

The binary search algorithm identifies a secret code length n in $O(n \cdot \log_2 n)$ turns.

4.2 Swap search

The swap search algorithm identifies the location of hits by swapping two entries and observing how the number of hits changes. It consists of two main phases, that are repeated until all the entries of secret code s have been identified. An overview can be found in Algorithm 1.

We first define the unidentified subsequence of query. If we have query q and partial solution p, then

$$u(q, p) = \{q_i \mid p_i = 0\}. \tag{4}$$

So if $q = (1, 2, 3, 4)$ and $p = (1, 0, 0, 4)$ then we have unidentified subsequence $u(q, p) = (2, 3)$.

In the first phase, the procedure NEXTQUERY is called to find a query with unidentified hits. Suppose we have a query q with no unidentified hits: $a(q, s, p) = 0$. To obtain a new query, a circular shift to the right is applied to

Algorithm 1 Swap Search Algorithm

Input: Secret code s
Output: Solution p

```
q := (1, 2, ..., n)
p := (0, 0, ..., 0)                          ▷ A sequence of n zeros
while p contains 0 do
    while a(q, s, p) = 0 do
        nextQuery                            ▷ Find query with unidentified hits
    while a(q, s, p) > 0 do
        swapSearch                           ▷ Find unidentified hits in query
return p
```

the entries in q_p. More specifically, for the unidentified subset $q_p = u(q, p)$ we define the shift

$$shift(q_{p_i}) = \begin{cases} q_{p_n} & \text{if } i = 1 \\ q_{p_{i-1}} & \text{if } i > 1. \end{cases} \tag{5}$$

Then by applying this shift, we find our next query $q' = next(q, q_p)$ with

$$q'_i = \begin{cases} shift(q_{p_j}) & \text{if } q_i = q_{p_j} \text{ for some } j \\ q_i & \text{otherwise.} \end{cases} \tag{6}$$

We repeat this until we find a q' with $a(q, s, p) > 0$. An overview of the nextQuery procedure can be found in Algorithm 2.

Once we have found such a query, we move on to the second phase. To find the unidentified hits in a query q, we use the procedure swapSearch, as specified in Algorithm 3[1]. For this, we define the swap operation, where a new query is created by swapping two entries. Let $q = (q_1, q_2, \ldots, q_n)$, then we create a new permutation $q' = swap(q, x, y)$ with

$$q'_i = \begin{cases} y & \text{if } q_i = x \\ x & \text{if } q_i = y \\ q_i & \text{otherwise.} \end{cases} \tag{7}$$

The entries of q_p are tested in pairs by employing the swapTest. The full details on this procedure can be found in Appendix A.2, but we will illustrate the general idea behind it.

Suppose for simplicity that $q_p = q$. First, we obtain a new query $q' = swap(q, q_1, q_2)$. If $a(q', s) = a(q, s)$ then neither q_1 or q_2 is a hit. If $a(q', s) = a(q, s) - 2$, then both q_1 and q_2 are hits. If $a(q', s) = a(q, s) - 1$ then either q_1 or q_2 is a hit, and we create a third query $q'' = swap(q, q_1, q_3)$. Suppose we find

[1] In the actual implementation, some changes were made to further reduce the number of turns. The details on this can be found in Appendix A.1

Algorithm 2 Function nextQuery

Input: Query q, secret code s, partial solution p
Output: New query q with unidentified hits and total number of hits a_s

procedure nextQuery(q, s)
  $q_p := u(q, p)$
  $a_s := a(q, s)$
  $a_p := a(q, p)$
  while $a_s = a_p$ **do** ▷ $a(q, s, p) = 0$
    $q := next(q, q_p)$
    $q_p := u(q, p)$
    $a_s := a(q, s)$
  return q, a_s

$a(q'', s) = a(q, s)$, then q_1 can't be a hit, and thus it has to be q_2. However, in some cases, we require a fourth and sometime even a fifth query to identify all hits. Note that we could also have cases where $a(q', s) > a(q, s)$ and have accidentally identified a new hit with the swap.

If q still has unidentified hits, we repeat the procedure for the next two entries. This process continues until all the hits in q have been identified.

After identifying all the hits, we go back to the first phase.

In general, we know only q_p can contain unidentified hits. So we limit our search and only apply swapTest and nextQuery to the entries of q_p. This also means that once we have identified a hit, it remains in the correct position for the rest of the algorithm.

Algorithm 3 Function swapSearch

Input: Query q with number of hits a_s, secret code s, partial solution p
Output: Partial solution p

procedure swapSearch(q, a_s, s)
  $q_p := u(q, p)$
  $a_p := a(q, p)$
  $i := 1$
  while $a_s > a_p$ & $i \leq |q_p| - 2$ **do**
    $p :=$ swapTest$(q, q_{p_i}, q_{p_{i+1}}, q_{p_{i+2}})$
    $a_p := a(q, p)$
    $i := i + 2$
  if $a_s > a_p$ **then** ▷ *Hits are at the end of q*
    $p :=$ swapTest$(q, q_{p_{n-1}}, q_{p_n}, q_{p_0})$
  return p

Concerning the number of turns needed to win, we have the following result.

Lemma 1. *The swap search algorithm for permutation mastermind identifies a secret code length n in at most $\frac{7}{4}n(n+1)$ turns.*

Proof. To find the first query with more than zero hits, we need at most n turns. For the swap search to find the hits, we need to test at most $\frac{n}{2}$ pairs and each pair requires at most 5 turns.

Since we have now identified at least one entry of s, to find the next permutation with an unidentified hit, we require at most $n-1$ turns. To identify the hits, we now need to test at most $\frac{n-1}{2}$ pairs with at most 5 turns each.

From this, we find that the amount of turns needed to identify the secret code has upper bound

$$\sum_{i=0}^{n}(n-i+5\frac{n-i}{2}) = \sum_{i=1}^{n}\frac{7}{2}(n-i) = \frac{7}{2}(n\frac{n+1}{2}) = \frac{7}{4}n(n+1).$$

□

Both NEXTQUERY and SWAPSEARCH require $O(n)$ turns. The outer loop that repeats the two phases also iterates $O(n)$ times. We conclude that the swap search algorithm identifies the secret code in $O(n^2)$ turns.

4.3 Comparing binary and swap search

Given their complexity, we expect the binary search algorithm to perform better than the swap search algorithm for larger n.

On the other hand, when considering the AYTO variant (where $n = 10$) the binary search algorithm does not seem like a fitting choice. Since we are trying to win within ten rounds, the first phase where we query for q_1 to q_{10} guarantees we will always lose. Although we have the addition of the truth booth, there is no straightforward way of using it during this phase that significantly reduces the number of turns needed.

Therefore, we will use the swap search algorithm to construct an algorithm for the AYTO variant.

5 AYTO algorithms

Using the swap search algorithm described in Section 4.2, we construct an algorithm for the AYTO variant. We also present a second algorithm, which employs an inference table in a similar way to [4].

5.1 Swap search

For the AYTO variant, the truth booth can be used to significantly reduce the number of turns needed in the swap search algorithm.

As mentioned in Section 4.2, the swap test can take up to five turns to identify the hits in a pair of entries. For the AYTO variant, we only need one turn to test the pair. In Algorithm 4, the SWAPTEST for the AYTO variant can be found.

Suppose we have query q and $q' = swap(q, q_i, q_j)$. We distinguish five different cases:

- Case 1: $a(q', s) = a(q, s) - 2$
- Case 2: $a(q', s) = a(q, s) + 2$
- Case 3: $a(q', s) = a(q, s) - 1$
- Case 4: $a(q', s) = a(q, s) + 1$
- Case 5: $a(q', s) = a(q, s)$.

In cases 3 and 4, we use the truth booth to identify the one hit in the pair. Suppose we have case 3, that is, we find $a(q', s) = a(q, s) - 1$. Then either q_i or q_j is a hit. If $b(i, q_i) = 1$ then clearly q_i is the hit, if $b(i, q_i) = 0$ then it has to be q_j. Similarly, in case 4 we can find the hit by testing $b(i, q'_i)$.

Algorithm 4 Function SWAPTEST (AYTO Variant)

Input: Query q with number of hits a_s, entries q_i and q_j, secret code s, partial solution p
Output: Partial solution p, Boolean truthBooth

```
procedure SWAPTEST(q, a_s, q_i, q_j, s, p)
    q' := swap(q, q_i, q_j)
    a'_s := a(q_2, s)
    if a'_s = a_s − 2 then
        p_i := q_i, p_j := q_j,
        truthBooth := true                          ▷ The truth booth is available again
    if a'_s = a_s + 2 then
        p_i := q'_i, p_j := q'_j,
        truthBooth := true
    if a'_s = a_s − 1 then                          ▷ Either q_i or q_j is a hit
        if b(i, q_i) = 1 then
            p_i := q_i
        else
            p_j := q_j
        truthBooth := false                         ▷ We have used the truth booth
    if a'_s = a_s − 1 then                          ▷ Either q'_i or q'_j is a hit
        if b(i, q'_i) = 1 then
            p_i := q'_i
        else
            p_j := q'_j
        truthBooth := false
    if a'_s = a_s then                              ▷ No hits were found, p remains unchanged
        truthBooth := true
    return p, truthBooth
```

In the other cases, we do not need any further truth booth for the pair. Instead, we use it to check the next entry in the unidentified subset, so we do not need to test it with the swapTest anymore. The details of the entire swap search algorithm for the AYTO variant can be found in Appendix B.

With this addition of the truth booth, we find a new upper bound for the required number of turns. While the swap search algorithm for permutation mastermind required at most $\frac{7}{4}n(n+1)$ turns, we find the following result for the AYTO variant.

Lemma 2. *The swap search algorithm for the AYTO variant identifies a secret code length n in at most $\frac{3}{4}n(n+1)$ turns.*

Proof. To find the first query with more than zero hits, we need at most n turns. For the swap search to find the hits, we need to test at most $\frac{n}{2}$ pairs which can be done in 1 turn.

Since we have now identified at least one entry of s, to find the next permutation with an unidentified hit, we require at most $n-1$ turns. To identify the hits, we now need to test at most $\frac{n-1}{2}$ turns.

From this, we find that the amount of turns needed to identify the secret code has upper bound

$$\sum_{i=0}^{n}(n - i\frac{n-i}{2}) = \sum_{i=1}^{n}\frac{3}{2}(n-i) = \frac{3}{4}n(n+1).$$

□

Even though swapSearch for the AYTO variant requires fewer turns than for permutation mastermind, the search is still done in $O(n)$ turns. The complexity of the outer loop also remains $O(n)$. Consequentially, the total algorithm identifies a secret code length n in $O(n^2)$ turns.

5.2 Inference table

In the swap search algorithm, much of the obtained information remains unused. For example, if $a(q,s) = 0$ then we know none of the entries are correct.

To make use of this information, we introduce an inference table T, similar the the one presented in [4]. This is an n by n matrix where the entries are either zero or one. A one in position (i,j) indicates s_i could still be j, a zero indicates this is not possible. So if, for example, we have truth booth result $b(1,5) = 0$, then $T_{1,5}$ is set to zero. However, if we would have found $b(1,5) = 1$, then all $T_{1,j}$ with $j \neq 5$ and $T_{i,5}$ with $I \neq 1$ are set to zero.

If, at any point, we have that $T_{i,j}$ is the only non-zero entry in either its row or column, then we know that $s_i = j$ (if we hadn't yet discovered this).

An overview of the swap search algorithm with inference table can be found in Algorithm 5. After every swapSearch, the procedure checkTable updates T for the new identified hits, and checks if we can derive any new hits from the table. The details on checkTable can be found in Algorithm 6.

Algorithm 5 Swap Search Algorithm (Inference Table)

Input: Secret code s
Output: Solution p

$q := (1, 2, ..., n)$
$p := (0, 0, ..., 0)$ ▷ *A sequence of n zeros*
truthBooth := true
$T :=$ n by n matrix of ones
while p contains 0 **do**
    while $a(q, s, p) = 0$ **do**
        nextQuery ▷ *Find query with unidentified hits*
    while $a(q, s, p) > 0$ **do**
        swapSearch ▷ *Find unidentified hits in query*
    checkTable
return p

We should note that checkTable is also called during the nextQuery procedure, and that for certain cases of the swapTest, additional changes are made to T. The details on this can be found in Appendix C.

Preferably, the table would also be checked during swapSearch, to prevent checking for information we could already have derived. However, this would make the algorithm more involved, as will be further discussed in Section 8.2.

Algorithm 6 Function checkTable

Input: Partial solution p, inference table T
Output: Updated partial solution p and inference table T

procedure checkTable(p, T)
    change:=false
    for all entries in p **do** ▷ *Update table for current p*
        if $p_i > 0$ **then**
            $T_{i,j} := 0$ for all $j \neq p_i$
            $T_{k,p_i} := 0$ for all $k \neq i$
    for all rows of T **do**
        if there is an x s.t. $T_{i,x} > 0$ and $T_{i,j} = 0$ for all $j \neq x$ **then**
            $p_i = x$
            change:=true ▷ *There is a change in p*
    for all columns of T **do**
        if there is a k s.t. $T_{k,j} > 0$ and $T_{i,j} = 0$ for all $i \neq k$ **then**
            $p_k = j$
            change:=true
    while change = true **do** ▷ *Update and check table again*
        checkTable
    return p, T

6 Results

We implemented the binary search and swap search algorithm for permutation mastermind, and the swap search with and without inference table for the AYTO variant. In this section, the result for all four algorithms will be presented.

6.1 Permutation mastermind

We tested both the binary search and the swap search algorithm for 100 000 uniformly at random selected secret codes of length $n = 10$. The results can be found in Table 1.

We found that, on average, the swap search algorithm performs better. It does have a larger standard deviation and the maximum and minimum are further apart. This is not surprising, since the performance of the swap search algorithm is more dependent on the distribution of the hits over the queries. It starts at the beginning of the query and identifies all the hits by running through the query once. As a result, it will perform better if we find many hits in one query and they are towards the beginning of the unidentified set.

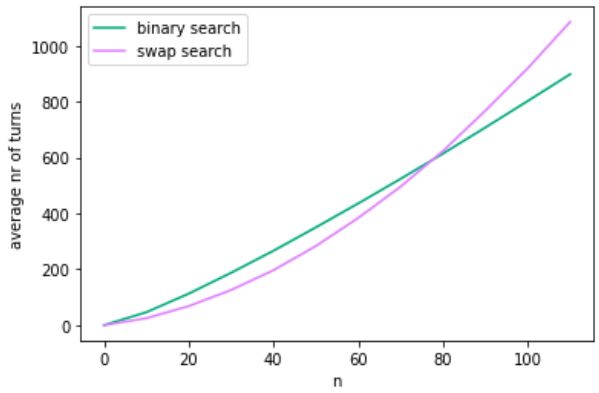
The binary search on the other hand, only identifies one hit per run. It also takes about the same amount of turns to identify each hit, with little influence from its location in the query. Hence, the total amount of turns needed will not vary as much.

	binary search	swap search
mean	46.528	25.284
std	2.078	4.162
max	55	43
min	38	8

Table 1: Comparison of the binary and swap search algorithm for permutation mastermind.

In addition, we also tested the two algorithms for different permutation lengths. We tested for lengths ranging from 0 to 110, each for 1000 secret codes. The results can be seen in Figure 1. While the swap search performs better for smaller n, the number of turns grows much faster as n becomes larger. This is what we would expect, considering the swap search algorithm grows with $O(n^2)$, compared to the binary search algorithm with $O(n \cdot \log_2 n)$.

Figure 1: Binary search and swap search for different code lengths.



6.2 AYTO variant

In Table 2 the results for both the swap search algorithm and the swap search algorithm with inference table are presented. We tested the number of turns needed to identify the secret code and considered it a win if this was ten or lower. For both algorithms, we tested 100 000 uniformly at random selected secret codes.

	swap search	inference table
win rate	16.97%	50.03%
mean	12.581	10.542
std	2.208	1.802
max	23	19
min	4	1

Table 2: Comparison of the swap search algorithm with and without inference table.

We can see the win rate is more than three times larger when including the inference table. The mean, maximum and minimum number of turns are also lowered. For the swap search algorithm, we found upper bound $\frac{3}{4}n(n+1)$. For $n = 10$ this amounts to 82.5 turns. This is much larger than the maximum amount we found testing.

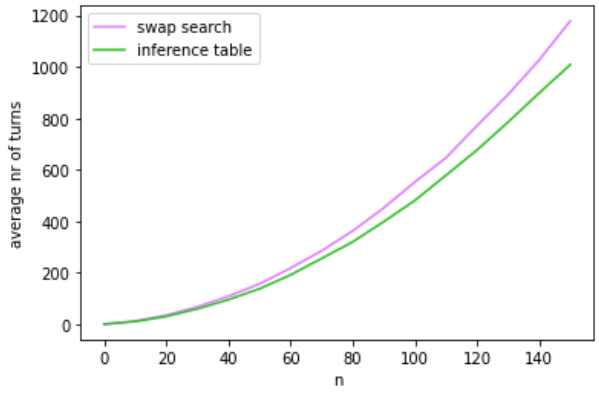
In addition, we tested the algorithm with inference table for all possible secret codes. These are all permutations of the number one to ten, which is a total of $10! = 3\,628\,800$ secret codes. The results can be seen in Table 3.

	inference table
win rate	49.88%
mean	10.545
std	1.804
max	19
min	1

Table 3: Swap search algorithm with inference table for all possible secret codes.

In Figure 2, the results for different n for swap search with and without inference table can be seen. We tested 100 random secret codes with lengths ranging from 0 to 150. The algorithm without inference table grows faster as n grows larger. However, the algorithm with inference table still appears to grow much faster than $O(n)$.

Figure 2: Swap search with and without inference table for different code lengths.



7 Conclusion

For permutation mastermind, we found the swap search algorithm performs better for secret codes length $n = 10$. For longer permutations, say $n = 100$, the binary search algorithm performs better.

Introducing an inference table improves the swap search algorithm for the AYTO variant. With inference table, the algorithm has a win rate of 49,88% and takes a maximum of 19 turns.

8 Discussion

We discuss the results for both permutation mastermind and the AYTO variant and mention possible improvements.

8.1 Permutation mastermind

We concluded that the binary search algorithm is not a fitting choice for a secret code of length $n = 10$. This is, however, for the specific binary search algorithm that was presented in [1].

We could change the first phase in a way that would require fewer turns. For example, like the one in [3] where, if the first query has $a(q, s) > 0$, the hits are identified similarly to swap search. We can then always guarantee a pair of queries with $a(q^i, s, p) > 0$ and $a(q^{j+1}, s, p) = 0$ exists. This removes the need to start by testing ten queries.

In addition, we could limit the binary search to the unidentified subset, reducing the number of turns needed to identify a hit.

Finally, the algorithm could be adjusted to keep track of the locations of all the unidentified hits. For example, assume we find the section from 1 to $l - 1$ and the section from l to n both have one unidentified hit. We could first search in entries 1 to $l-1$ and then, after identifying the hit, return to searching in the section from l to n instead of considering the entire query all over again. This would improve the algorithm, but also require a significant amount of additional bookkeeping.

With the mentioned improvements, it could very well be that the binary search algorithm also outperforms the swap search algorithm for $n = 10$.

Concerning the implementation of the binary search algorithm, the maximum found while testing exceeds the theoretical upper bound of $(n-3)\lceil log_2 n\rceil + \frac{5}{2}n - 1$ as found in [1]. The observed maximum over 100 000 random secret codes was 55, while the theoretical upper bound would be 52. This could indicate that our implementation was not fully identical to the original algorithm, but we were not able to identify any differences.

8.2 AYTO variant

We are certain that the algorithm with inference table is not yet optimal, and could still be improved upon. For example, by checking the table after each iteration in SwapSearch. However, this improvement would introduce some complications. Say, for example, that we have query $q = (1, 2, 3, 4, 5)$ with $a(q, s, p) = 2$ and unidentified subset $q_p = (2, 3, 4, 5)$. Suppose that we check the inference table and identify $s_2 = 5$. If we simply remove 5 from the unidentified subset, we obtain $q_p = (2, 3, 4)$ with partial solution $p = (1, 5, 0, 0, 0)$. But this would correspond to a query $q = (1, 5, 2, 3, 4)$, which is not the original query that we know contains another unidentified hit.

A solution would be to swap 2 and 5 in q, but then we would also have to consider the possibility that the new $q'_{p_5} = 2$ is a hit.

Although these improvements would make the algorithm more involved, they are likely to improve the win rate even further.

Interestingly enough, while we obtained a win rate of 49.88%, the actual show has a win rate of 87,5%, with 7 out of the 8 seasons being won by the contestants. A possible, but improbable explanation would be that the contestants have discovered a much more clever approach to playing the game. It is, perhaps, more likely that their personalities do help with identifying their perfect match.

However, we believe the most probable cause of this high win rate is that the show is not entirely truthful. Especially considering that, in quite a few seasons, the number of hits increases by a surprising amount from the 9^{th} to the 10^{th} week.

References

[1] Mourad El Ouali and Volkmar Sauerland. "Improved approximation algorithm for the number of queries necessary to identify a permutation". In: *International Workshop on Combinatorial Algorithms*. Springer. 2013, pp. 443–447.

[2] Donald E Knuth. "The computer as master mind". In: *Journal of Recreational Mathematics* 9.1 (1976), pp. 1–6.

[3] Ker-I Ko and Shia-Chung Teng. "On the number of queries necessary to identify a permutation". In: *Journal of Algorithms* 7.4 (1986), pp. 449–462.

[4] T Mahadeva Rao. "An algorithm to play the game of mastermind". In: *ACM SIGART Bulletin* 82 (1982), pp. 19–23.

A Swap search algorithm for permutation mastermind

We will discuss the specifics of the swap search algorithm and the utilized swapTest, as discussed in Section 4.2.

A.1 Swap search

In Algorithm 3, we always swap test the entries at the end of the query, if there are still unidentified hits. In the actual implementation, for the cases where $t = |q_p|$ with $a(q, s, p) = 1$, and $t = |q_p| - 1$ with $a(q, s, p) = 2$ no swap test was performed, since the remaining entries to check and the remaining unidentified hits are equal. Thus, in these cases, we identify the hits to be the last one or two entries of q_p, respectively.

In some cases, swapTest ends up identifying $q_{p_{i+2}}$ in addition to testing q_{p_i} and $q_{p_{i+1}}$. In this case, we can skip $q_{p_{i+2}}$ and move on to entries $q_{p_{i+3}}$ and $q_{p_{i+4}}$. We should point out that in some cases, $q_{p_{i+2}}$ is not identified, but has been checked. In this case, we could also skip it, but this was not implemented in the swap search algorithm and remains a possible improvement.

A.2 Swap test

The swapTest used in swapSearch, creates a new permutation $q^2 = swap(q, q_i, q_j)$, and tests if q_i and q_j are hits by considering several cases

- Case 1: $a(q^2, s) = a(q, s) - 2$
- Case 2: $a(q^2, s) = a(q, s) + 2$
- Case 3: $a(q^2, s) = a(q, s) - 1$
- Case 4: $a(q^2, s) = a(q, s) + 1$
- Case 5: $a(q^2, s) = a(q, s)$

an overview of conclusions from each cases can be found in Algorithm 7.

In case 3 and 4, extra testing is needed to determine the location of the hits. Since the procedure for both cases is almost identical, we will only discuss the details of case 3, which can also be found in Algorithm 8. We create $q^3 = swap(q, q_i, q_j)$, and once again distinguish several cases

- Case 3.1: $a(q_3, s) = a(q, s) - 2$
- Case 3.2: $a(q_3, s) = a(q, s) + 2$
- Case 3.3: $a(q_3, s) = a(q, s) - 1$
- Case 3.4: $a(q_3, s) = a(q, s) + 1$
- Case 3.5: $a(q_3, s) = a(q, s)$.

Algorithm 7 Function swapTest (Permutation Mastermind)

Input: Query q with number of hits a_s, entries q_i, q_j, q_k, secret code s partial solution p

Output: Partial solution p

procedure swapTest$(q, a_s, q_i, q_j, q_k, s, p)$

- $q^2 := swap(q, q_i, q_j)$
- $a_s^2 := a(q_2, s)$
- **if** $a_s^2 = a_s - 2$ **then**
  - $p_i := q_i,\ p_j := q_j$
- **if** $a_s^2 = a_s + 2$ **then**
  - $p_i := q_i^2,\ p_j := q_j^2$
- **if** $a_s^2 = a_s - 1$ **then** ▷ *Either q_i or q_j is a hit*
  - Case 3 test
- **if** $a_s^2 = a_s - 1$ **then** ▷ *Either q_i^2 or q_j^2 is a hit*
  - Case 4 test
- **if** $a_s^2 = a_s$ **then**
  - Do nothing
- **return** p

Algorithm 8 Case 3 test ($a_s^2 = a_s - 1$)

$q^3 := swap(q, q_i, q_k)$
$a_s^3 := a(q^3, s)$
if $a_s^3 = a - 2$ **then**
  $p_i := q_i,\ p_k := q_k$
if $a_s^3 = a_s + 2$ **then**
  $p_i := q_i^3,\ p_j := q_j^3,\ p_k := q_k^3$
if $a_s^3 = a_s - 1$ **then** ▷ *Either q_i or both q_j and q_k are hits*
  Case 3.3 test
if $a_s^3 = a_s + 1$ **then** ▷ *q_j and either q_i^3 or q_k^3 are hits*
  $q^4 := swap(q^3, q_j, q_k)$
  $a_s^4 := a(q^4, s)$
  if $a_s^4 = a_s^3 - 2$ **then**
    $p_j := q_j^3,\ p_k := q_k^3$
  if $a_s^4 = a_s^3 - 1$ **then**
    $p_i := q_i^3,\ p_j := q_j^3$
if $a_s^3 = a_s$ **then**
  $p_j := q_j$
return p

Once again, case 3.3 and 3.4 require further testing with a fourth query q^4. For case 3.3, we could even end up in a case requiring a fifth query q^5. The details on this can be found in Algorithm 9.

For case 4, we apply the same procedure as for case 3, except we have q^2 instead of q.

Algorithm 9 Case 3.3 test ($a_s^3 = a_s - 1$)

```
if a_s = 1 then
    p_i := q_i
else
    q^4 := swap(q, q_j, q_k)
    a_s^4 := a(q^4, s)
    if a_s^4 = a_s − 2 then
        p_j := q_j^3, p_k := q_k^3
    if a_s^4 = a_s + 2 then
        p_i := q_i^3, p_j := q_j^3, p_k := q_k^3
    if a_s^4 = a_s + 1 then                ▷ q_i and either q_j^4 or q_k^4 are hits
        q^5 := swap(q^4, q_i^4, q_j^4)
        a_s^5 := a(q^5, s)
        if a_s^5 = a_s^4 − 2 then
            p_j := q_i^4, p_k := q_j^4
        if a_s^5 = a_s^4 − 1 then
            p_j := q_i^4, p_k := q_k^4
    if a_s^4 = a_s then
        p_i := q_i
return p
```

B Swap search algorithm for AYTO variant

For the AYTO variant, the swap search algorithm requires some slight adjustments to incorporate the truth booth. Since the truth booth is available before each matching ceremony, we introduce a Boolean truthBooth, which is true when we still have the truth booth available.

The NEXTQUERY procedure alternately checks the first entry of the unidentified subset in the truth booth and tests for hits in the matching ceremony. We should note that, after using the truth booth, we do not test the same query for hits but immediately cycle to the next one. We found this method to be slightly more efficient than using the same query for the truth booth and matching ceremony. Presumably, this is because, if the truth booth does not return a hit, the remaining entries that can be hits is reduced by one. Since we eventually either find a query with unidentified hits or find a new hit with the truth booth, we do not have to worry about queries remaining unchecked and the loop continuing forever.

For the swapSearch procedure, we either use the truth booth to check the next entry in the unidentified subset, or we use it in the swapTest

Algorithm 10 Swap Search Algorithm (AYTO Variant)

Input: Secret code s
Output: Solution p

$q := (1, 2, ..., n)$
$p := (0, 0, ..., 0)$ ▷ *A sequence of n zeros*
truthBooth := true
while p contains 0 **do**
    while $a(q, s, p) = 0$ **do**
        nextQuery ▷ *Find query with unidentified hits*
    while $a(q, s, p) > 0$ **do**
        swapSearch ▷ *Find unidentified hits in query*
return p

Algorithm 11 Function nextQuery (AYTO Variant)

Input: Query q, secret code s, Boolean truthBooth, partial solution p
Output: New query q with number of hits a_s, Boolean truthBooth

procedure nextQuery$(q, s, \text{truthBooth}, p)$
    $q_p := u(q, p)$
    $a_s := a(q, s)$
    $a_p := a(q, p)$
    while $a_s = a_p$ **do** ▷ $a(q, s, p) = 0$
        $q := next(q, q_p)$
        $q_p := u(q, p)$
        if truthBooth **then**
            $j := index(q, q_{p_1})$ ▷ *Retrieves the index of q_{p_1} in q*
            if $b(j, q_{p_1}) = 1$ **then**
                $p_j := q_{p_1}$
                $q_p := u(q, p)$
            truthBooth := false
        else
            $a_s := a(q, s)$
            truthBooth := true
    return q, a_s, truthBooth

Algorithm 12 Function swapSearch (AYTO Variant)

Input: Query q with number of hits a_s, secret code s, boolean truthBooth partial solution p
Output: Partial solution p, Boolean truthBooth

procedure swapSearch($q, a_s, s,$ truthBooth$, p$)
  $q_p := u(q, p)$
  $a_p := a(q, p)$
  while $a_s > a_p$ & $i \leq |q_p| - 1$ **do**
    if truthBooth **then**
      $j := index(q, q_{p_i})$
      if $b(j, q_{p_i}) = 1$ **then**
        $p_j := q_{p_i}$
      truthBooth := false
      $i := i + 1$
    else
      swapTest$(q, q_{p_i}, q_{p_{i+1}})$
      $a_p := a(q, p)$
      $i := i + 2$
  if $a_s > a_p$ **then** ▷ *A hit at the end of q*
    $j := index(q, q_{p_i})$
    $p_j := q_{p_i}$
  return p, truthBooth

C Inference table

In Section 5.2, we presented the swap search algorithm with inference table. We will discuss the changes in the nextQuery and swapSearch, compared to the algorithm without inference table.

First, we introduce a new subset of $q_{p_T} = u_T(q, p, T)$, with only the entries of the unidentified subset q_p that could still be a hit by T. We define

$$u_T(q, p, T) = \{q_i \mid p_i = 0 \; and \; T_{i,q_i} = 1\}. \tag{8}$$

The procedure nextQuery now only applies to truth booth to entries in q_{p_T}, to avoid checking entries that we know cannot be a hit. In addition, if we find $a(q, s, p) = 0$, we update in T that none of the entries in q_p are hits.

The swapSearch is now limited to q_{p_T}. In addition, whenever it uses the truth booth and does not identify a hit, we change the corresponding entry of T to zero.

During the swapTest, we have several cases where T can be updated. For example if we find $a(q', s) = a(q, s)$, we know the pair of entries we are testing cannot be hits in either q or q'. Hence all four corresponding entries of T can be set to zero. A complete overview off the adjusted swapTest, can be found in Algorithm 15

Algorithm 13 Function NEXTQUERY (Inference Table)

Input: Query q, secret code s, Boolean truthBooth, inference table T
Output: New query q with number of hits a_s, Boolean truthBooth partial solution p, inference table T

```
procedure NEXTQUERY(q, s)
    q_p := u(q, p)
    a_s := a(q, s)
    a_p := a(q, p)
    while a_s = a_p do                                   ▷ a(q, s, p) = 0
        q := next(q, q_p)
        q_p := u(q, p)
        if truthBooth then
            q_{p_T} := u_T(q, p)
            j := index(q, q_{p_{T_1}})                   ▷ Retrieves index of q_{p_{T_1}} in q
            if b(j, q_j) = 1 then
                p_j := q_j
                q_p := u(q, p)
            else
                T_{j,q_j} = 0
            truthBooth := false
        else
            a_s := a(q, s)
            if a_s = a_p then                            ▷ No entry of q_p is a hit
                for all entries in q_p do
                    k := index(q, q_{p_i})
                    j := q_{p_i}
                    T_{k,j} = 0
                truthBooth := true
        p, T = CHECKTABLE
    return q, a_s, truthBooth, p, T
```

Algorithm 14 Function SWAPSEARCH (Inference Table)

Input: Query q with number of hits a_s, secret code s, boolean truthBooth inference table T

Output: Partial solution p, Boolean truthBooth, inference table T

procedure SWAPSEARCH(q, a_s, s)
- $q_{p_T} := u_T(q, p)$ ▷ *Only check entries possible by T*
- $a_p := a(q, p)$
- **while** $a_s > a_p$ & $i \leq q_{p_T} - 1$ **do**
  - **if** truthBooth **then**
    - $j := index(q, q_{p_{T_i}})$
    - **if** $b(j, q_j) = 1$ **then**
      - $p_j := q_j$
    - **else**
      - $T_{j,q_j} = 0$
    - truthBooth := false
    - $i := i + 1$
  - **else**
    - SWAPTEST($q, q_{p_i}, q_{p_{i+1}}$)
    - $a_p := a(q, p)$
    - $i := i + 2$
- **if** $a_s > a_p$ **then** ▷ *A hit at the end of q*
  - $j := index(q, q_{p_i})$
  - $p_j := q_{p_i}$
- **return** p, truthBooth

Algorithm 15 Function SWAPTEST (Inference Table)

Input: Query q with number of hits a_s, entries q_i and q_j, secret code s, inference table T

Output: Partial solution p, Boolean truthBooth, inference table T

procedure SWAPTEST(q, a_s, q_i, q_j, q_k, s)

- $q' := swap(q, q_i, q_j)$
- $a'_s := a(q_2, s)$
- **if** $a'_s = a_s - 2$ **then**
  - $p_i := q_i,\ p_j := q_j,$
  - T
  - truthBooth := true ▷ *The truth booth is available again*
- **if** $a'_s = a_s + 2$ **then**
  - $p_i := q'_i,\ p_j := q'_j,$
  - truthBooth := true
- **if** $a'_s = a_s - 1$ **then** ▷ *Either q_i or q_j is a hit*
  - **if** $b(i, q_i) = 1$ **then**
    - $p_i := q_i$
    - $T_{j,q_j} := 0$
  - **else**
    - $p_j := q_j$
    - $T_{i,q_i} := 0$
  - truthBooth := false ▷ *We have used the truth booth*
- **if** $a'_s = a_s - 1$ **then** ▷ *Either q'_i or q'_j is a hit*
  - **if** $b(i, q'_i) = 1$ **then**
    - $p_i := q'_i$
    - $T_{j,q'_j} := 0$
  - **else**
    - $p_j := q'_j$
    - $T_{i,q'_i} := 0$
  - truthBooth := false
- **if** $a'_s = a_s$ **then** ▷ *No hits were found*
  - $T_{i,q_i} := 0,\ T_{i,q'_i} := 0,\ T_{j,q_j} := 0,\ T_{j,q'_j} := 0$
  - truthBooth := true
- **return** p, truthBooth, T