Name: Yilong Wang
CID: 01362420
yw14218

Mastermind Report

I employed different algorithms in the solver according to the length of the code and possible numbers.

1. If the total amount of possibilities is smaller than 10000, then use Donald Knuth's algorithm.
2. If the total amount of possibilities is larger than 10000, then use algorithms of Controlling Variables to crack the code one by one
3. If the number and length is covered by Dividing Algorithm, then use this to divide the mm into pieces and crack them parts by parts.

Donald Knuth Algorithm

If the total sample space is small, say, around thousands, then I use Donald Knuth's five step algorithm. The algorithm includes:

(1) Create all the possibilities in the space, which have the total amount of number^(length).
(2) According to minimax technique, amongst all the possibilities, pick one that will reduce the most amount of possibilities.
(3) Attempt the code and get feedback, removing all the possibilities that will not give the same feedback, if the current attempt is the correct code.
(4) Repeat step (2), until all black pegs have been found.

However, there will be many limitations in implementing this method.

(1) The minimax technique requires much condensed calculation which makes the running time longer than expected 10s. So in my code, instead of choosing the most optimal attempt, I use the random feature in C++ to pick an attempt for me in the remaining pool of possibilities.
(2) Due to the limitations in memory available for C++ vectors, we cannot store 15^15 possibilities together at the same time; the amount of possibilities generated is simply too large.
(3) In implementing step (2) in Donald Knuth's method, plenty of comparisons will take place, leading to a slow – down in running time and occasionally the programs runs for more than 10s because of this.

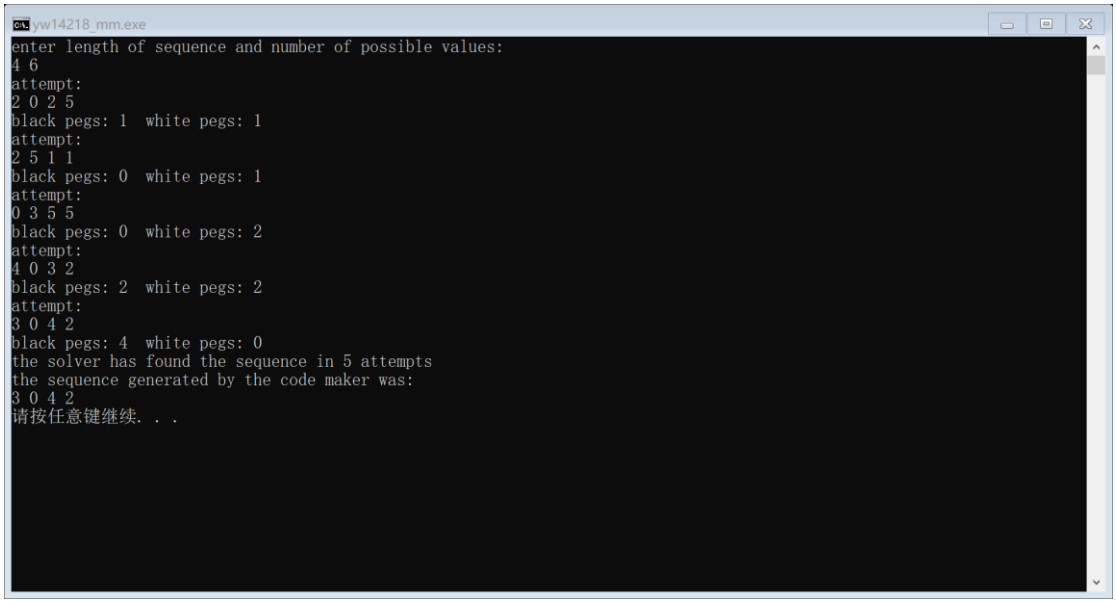
Provided there are 15^15 = 225 possibilities to be tested within a spanned time domain, I tried to experiment them with Donald Knuth's algorithm and recorded the time taken:

Where n represents number and m represents length.

If the time taken was within 5s (multiple attempts and average taken) , the grid was shaded red.

m\n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2															
3															
4															
5															
6															
7															
8															
9															
10															
11															
12															
13															
14															
15															

From the table we can tell that this only steadily covers 83 out of 225 situations, which implies at higher instances, we face a trade-off between running time and attempts required, even within the maximum storage of C++ vectors. It can be noticed that as the total amount of possibilities increase, the running time increases rapidly. Also, if the possibilities are below 5000, the program almost solves the problem instantly. For example, in average, it takes about 5 attempts to solve a 4 X 6 mm.



Dividing Algorithm

Inspired by Donald Knuth's algorithm, I came up with this Dividing Algorithm to solve higher instances. The core concept of this algorithm is to divide a long mm into pieces of smaller mm which could be solved using previous Donald Knuth Algorithm.

If this idea is to be implemented, then the following problems must be solved in advance:

(1) It's difficult to separate a given mastermind code into pieces without violating the rules, since we must enter the whole codes in the attempt and have our feedback influenced by all of them. So we must ensure other variables we enter do not influence the test.

(2) Typically we want to divide a long mm to pieces which should be , within the possible range, as long as possible. Dividing a 4 X 15 into two 4 X 6 and one 4 X 3 is much efficient than dividing it into, say , five 4 X 3 mm. A system of algorithm must exist be automatically allocate the most efficient combination.

One of the methods I came up with is to use a parameter along with several attempts to see whether or not the code is this particular parameter at all index, for example:

Code: 15 X 15

First 15 attempts:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
.....
.....

Since the code can't be 0 and 1 at the same time, we can be assured of one of the following, via the feedback generated regarding black pegs:

1. The code with this index is not 0
2. The code with this index is not 1

Therefore, we can create a vector storing a series of code which would output 0 black pegs if we use it as an attempt. Then, according to the length, we divide this code to smaller pieces, in my case, three pieces, and solve them as a group for each time using reduced Donald Knuth's Algorithm, where the information provided by white pegs are neglected.

I created a sample code that can solve several high mm with specific length and number, experimenting with it and obtained the following results.

m\n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2															
3															
4															
5															
6							X	X	26	23	25	32	30	34	32
7															
8															
9							X	X	33	35	37	41	47	49	48
10															
11															
12							X	X	49	51	55	53	60	63	63
13															
14															
15							X	X	X	X	X	X	X	X	82

(black numbers represent experimental average attempt for 10 trials)

It can be seen that the number contained in the mm hardly affect the efficiency of this solver, provided the length does not change. At higher instances, the added costs of computing several 0 and 1s at the beginning will be reduced.

The areas shaded with "X" means at this instance the program is not steady and occasionally fails, thus these instances are considered outside of the range covered by this algorithm.

Variable Controlling

The final algorithm I use can deal with all the instances and give results within a fairly good time

(1) If it is the first attempt, then attempt with all zeros.
(2) Change the first element to a random number and attempt again, if black pegs increase, then store the random number, if black pegs decrease, then store zero, else, remember this failed attempt so that it won't show up again.
(3) Repeat (2) until a number has been stored in a vector containing potential code, then repeat (2) with the second element and beyond
(4) If all the elements have been through (2), then attempt the code with the potential code stored and win.

This algorithm purely ignores the impacts of white pegs, but is actually a very suitable algorithm for computer implementation.

Taking the same mm into experiments, I obtained the following results in comparison.

m\n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2															
3															
4															
5															
6							20	22	25	26	32	40	38	44	44
7															
8															
9							25	35	38	40	55	58	58	68	65
10															
11															
12							33	47	50	53	64	70	70	80	80
13															
14															
15															95

(black numbers represent experimental average attempt for 10 trials)

If the result is larger than previous counterpart, it is shaded red

The results are all larger than previous case, several further trials are attempted to gain more insights, and the results are shaded blue.

From the two tables, we know that when the Dividing Algorithm covers the range, then it typically works better than the Variable Controlling Algorithm, in particular at higher instances.

Thus , in summary,

1. If the total amount of possibilities is smaller than 10000, then use Donald Knuth's algorithm.
2. If the total amount of possibilities is larger than 10000, then use Variable Controlling.
3. If the number and length is covered by Dividing Algorithm, then use this algorithm instead of Variable Controlling.

Appendix

Data in Dividing Algorithm (attempts with 10 trials):

15 X 15 :	82	91	90	88	75	90	76	73	73	80
12 X 15 :	74	61	56	71	73	48	49	74	59	61
9 X 15 :	56	56	40	37	45	48	46	52	51	50
6 X 15:	34	39	29	34	37	26	33	32	26	31
12 X 14:	69	52	49	69	68	62	63	65	68	66
9 X 14:	43	47	43	48	50	50	47	49	56	54
6 X 14	37	31	34	34	38	28	36	37	33	33
12 X 13:	61	59	54	70	60	58	63	45	62	64
9 X 13:	45	44	40	55	47	49	46	45	46	51
6 X 13:	30	33	29	31	25	34	27	33	31	25

Data in Controlling Variable Algorithm (attempts with 10 trials):

15 X 15:	98	79	108	108	88	105	103	116	144	
12 X 15:	68	84	78	87	72	93	56	105	92	90
9 X 15:	55	55	77	62	44	66	69	81	71	69
6 X 15:	48	37	31	33	52	47	58	40	51	37
12 X 14:	69	69	79	81	79	80	75	85	103	83
9 X 14:	90	65	79	63	46	59	74	86	53	63
6 X 14:	45	41	33	30	40	55	39	51	45	59
12 X 13:	84	44	68	75	77	72	54	75	62	84
9 X 13:	54	45	40	54	66	62	44	73	58	77
6 X 13:	45	18	37	47	35	36	35	45	40	37