MASTERMINDSOLVER

SPRING TERM ASSIGNMENT

SUSHANTH KOLLURU

Algorithms Considered and Used

After looking through several papers, two algorithms were finalised on. Temporel's algorithm, which uses a combination of a 'heuristic' and 'hill climbing' approaches, is being used for smaller values and Singley's algorithm for the larger scaled solution.

Several algorithms were considered, the most obvious of which is Knuth's. The issue with most of these algorithms was either scalability or simplicity. Knuth's algorithm, for example scales very poorly, as calculating the next guess and running through all the possible solutions simply takes too much time and too many resources. Temporel's algorithm was intuitive, scalable and was ne of the better performing algorithms (with respect to the average number of guesses).

However, for larger sets even the hill climbing heuristic algorithm will struggle with both time and moves. For this project, a variation of Singley's algorithm was used. This was primarily down to the speed of the algorithm as well as the simplicity and efficiency. The algorithm is fairly straightforward to understand and has an impressive average rate. Within this project, Singley's algorithm was also used as a 'backup' algorithm for when Temporel's algorithm takes too long. This aspect will be explained in the section later on.

TEMPOREL'S ALGORITHM

Logic (copied from the paper):

1. We submit to the Code maker a random guess that we call the "Current Favourite Guess" (CFG).
2. From the CFG, we induce a new potential code (this is explained in the latter section)
3. If potential code is not consistent with all previous guesses, go to step 2 otherwise submit it as new guess.
4. If submitted guess scores [0,0] then suppress from the pool of colours all the colours present in the last submitted guess. Then find a new random combination (with the new pool of valid colours) consistent with all previous guesses' scores and set this new combination as new CFG and submit it to the code setter.
5. If submitted guess score is as good as or better than CFG score then set this guess to be our new CFG and also set the new score as best score.
6. If submitted guess scores [4,0], stop otherwise go to step 2.

SINGLEY'S GREEDY SEARCH ALGORITHM

1. The first m turns are to be used to determine the correct number of each colour appearing in the secret code.
2. The number with the most black hits is then made into the "base"
3. Each index in then changed incrementally with values determined by whether a black hit has been detected by that colour or not, and then a guess is submitted
4. Depending on the feedback, the next stage is determined:
    a. If the no of black hits increases, the number that the index has been changed to is in the right position
    b. If the no. of black hits stays the same, the number in the index is yet to be determined and more numbers need to be tested
    c. If the no. of black hits decreases, the "base value" belongs at that index

    This is continued till the solution has been found.

GIVE FEEDBACK

Give feedback is a very important part of the program. It works by initially comparing each position from the guess and the sequence and checking if they're equal. If so, the black hit count is increased by one and the elements of the black hits are "used" by being stored in the " -used" vectors.

The next stage calculates white hits, and goes through each possible combination. The function elementPresence is then used. This function goes through a vector and returns true if the tested value is in the vector. If this is passed, the check is then made to see if the two elements match. If they do, the white hits counter is increased by one.

```
void give_feedback(const std::vector<int>& attempt, int& black_hits, int& white_hits){

  black_hits = 0;
  white_hits = 0;

  std::vector<int> guessElementsUsed;
  std::vector<int> answerElementsUsed;
  for (int i = 0; i < sequence.size(); i++) {
    if(attempt[i] == sequence[i]){
      black_hits++;
      guessElementsUsed.push_back(i);
      answerElementsUsed.push_back(i);
    }
  }

  for (int i = 0; i < attempt.size(); i++) {
    for (int j = 0; j < sequence.size(); j++) {
      if((!elementPresence(i, guessElementsUsed)) && (!elementPresence(j, answerElementsUsed)))
      if(attempt[i] == sequence[j]){
        white_hits++;
        guessElementsUsed.push_back(i);
        answerElementsUsed.push_back(j);
      }
    }
  }
}
```

NEW GUESS GENERATION

```
if(cfgBlack>0){
  for (int i = 0; i < cfgBlack; i++) {
    bool replacement = false;
    while(!replacement){
      int element = randn(length);

      if(!elementPresence(element, cfgElementsUsed)){
        attempt[element] = cfg[element];
        attemptElementsUsed.push_back(element);
        cfgElementsUsed.push_back(element);
        replacement = true;
      }
    }
  }
}//randomly chooses black hits
```

The "CFG" and its respective black hit and white hit count is stored. From this, a new guess is generated. First, elements from the CFG (equivalent to the number of black hits) are selected to remain the same by the following code on the left:

Then, elements from the CFG (equivalent to the number of black hits) can possibly be moved in the next part of the code. The counter is present so that if it takes too long to select a white hit (it can actually be impossible if the wrong black hits are chosen)

```
if(cfgWhite>0){
  for (int i = 0; i < cfgWhite; i++) {
    bool replacement = false;
    while(!replacement && counter<1000){

      int cfgElement = randn(length);
      if(!elementPresence(cfgElement, cfgElementsUsed) && counter<1000){
        counter++;
        bool selection = false;
        while(!selection){
          int attemptElement = randn(length);
          if(!elementPresence(attemptElement, attemptElementsUsed)){
            attempt[attemptElement] = cfg[cfgElement];
            attemptElementsUsed.push_back(attemptElement);
            cfgElementsUsed.push_back(cfgElement);
            selection = true;
          }
          replacement = true;
        }

      }
    }
  }
}//randomly chooses white hits
```

```
std::vector<int> probabilityDistribution;

for (int i = 0; i < num; i++) {
  probabilityDistribution.push_back(0);
}

for (int i = 0; i < length; i++) {
  probabilityDistribution[cfg[i]] += 100;

  if(attempt[i] != -1){
    probabilityDistribution[attempt[i]]-= 55;
  }
}

for (int i = 0; i < probabilityDistribution.size() ; i++) {
  probabilityDistribution[i] = 100 - probabilityDistribution[i];
  if(probabilityDistribution[i] <= 0){
    probabilityDistribution[i] = 1;
  }
}

if(blockedElements.size()>0){
  for (int i = 0; i < blockedElements.size(); i++) {
    probabilityDistribution[blockedElements[i]] = 0;
  }
}

std::discrete_distribution<int> probabilities(probabilityDistribution.begin(),probabilityDistribution.end());

for (int i = 0; i < length; i++) {
  if(attempt[i] == -1){
    attempt[i] = probabilities(generator);
  }
}
}
```

At this point, the remaining values that aren't filled are done so using the Fitness Proportionate Scheme(FPS). This is based off the sub algorithm provided in Temporel's algorithm, refer to section 3.4 from the Temporel paper.

The code for the this selection is shown above. After this stage the code is checked to determine if it is a valid guess.

GUESS EVALUATOR AND VALIDATE

The guess evaluator does step 3 of Temporel's algorithm, with all the guesses and feedback being stored in a vector of vectors. The evaluator chooses a guess and its respective feedback and uses the validate function to check whether the guess provides the same response if the "potential guess" is correct. Validate works similarly to give_feedback, simply providing a bool output instead of black hits and white hits.

```
bool guessEvaluator (const std::vector<std::vector<int>>& guessVec, const std::vector<std::vector<int>>& feedbackVec, const std::vector<int>& potentialGuess){
  int i = 0;
  int valid = 1;


  while(valid == 1){
    if(i<guessVec.size()){

      if(!validate(guessVec[i],feedbackVec[i], potentialGuess)){
        valid = 0;
      }
      else {
        i++;
      }
    }
    else{
      valid = 2;
    }
  }
  if(valid == 2){
    return true;
  }
  else{
    return false;
  }
}
```

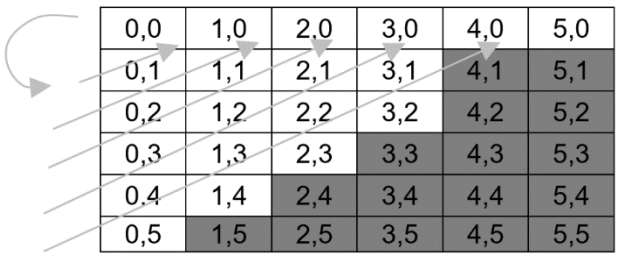
```
bool validate(const std::vector<int>& answer, const std::vector<int>& response,const std::vector<int>& guess){
  int black_hits = 0;
  int white_hits = 0;

  std::vector<int> guessElementsUsed;
  std::vector<int> answerElementsUsed;

  for (int i = 0; i < guess.size(); i++) {
    if(guess[i] == answer[i]){
      black_hits++;
      guessElementsUsed.push_back(i);
      answerElementsUsed.push_back(i);
    }
  }
```

SCORING RESPONSES

As mentioned earlier, the algorithm requires comparing feedback to determine which guess is the CFG , with the scoring system shown in Table 1. Scoring is primarily determined by the total number of hits, followed by black hits and then white hits. The following equation is used by the program to determine the scoring:

$$Ranking = n \times (Total\ Hits) + Black\ Hits$$

White hits do not require a specific term in the equation, as the number of white hits is proportional to the number



0,0	1,0	2,0	3,0	4,0	5,0
0,1	1,1	2,1	3,1	4,1	5,1
0,2	1,2	2,2	3,2	4,2	5,2
0,3	1,3	2,3	3,3	4,3	5,3
0,4	1,4	2,4	3,4	4,4	5,4
0,5	1,5	2,5	3,5	4,5	5,5

Table 1: Possible MM responses-5 peg game

```
int scoringFunction(int black, int white){

  return length*(black+white) + black;
}
```

of black hits (if the total hits remains the same). Alternatively, if the black hits remain constant changing the number of white hits will change the total number of hits. The multiplier used is ‘n’ to ensure that the formula will scale correctly.

After feedback is provided the black and white hits go through ‘scoringFunction’, which will provide an integer response. The score is then compared to the CFG’s scoring integer to determine whether the CFG is replaced by the new guess.

PERFORMANCE EVALUATION

TEMPOREL’S ALGORITHM

The picture above displays a typical hill climbing solution. The initial guess is fairly lucky, providing five white hits. It can then be seen that five of the first pegs have been randomly chosen, with the remainder being chosen by the fitness system. The actual output was worse, but this value is then stored and used to compare any new potential guesses. After this all of the pegs’ colours have been identified, with only their positions remaining. This is then in two moves, thus completing a 7 x 7 in 6 moves.

In general, temporal’s algorithm is quite impressive, particularly when the lengths are lower. The algorithm’s output is almost linear with colour. However, the algorithm then starts to struggle, particularly with larger lengths. One specific case is the 9x9, which would complete the set within ten seconds about two thirds of the time, and not complete it the remaining third. Due to the “risk” being worth it, a time limit has been implemented so that if a solution has not been found after nine seconds, Singley’s algorithm is used.

```
enter length of sequence and number of possible values:
7 7
0 5 3 6 0 4 6
attempt:
6 4 2 3 4 0 0
black pegs: 0  white pegs: 5
attempt:
2 1 4 4 0 3 4
black pegs: 1  white pegs: 2
attempt:
5 0 0 6 6 4 4
black pegs: 2  white pegs: 4
attempt:
3 0 6 0 5 6 4
black pegs: 0  white pegs: 7
attempt:
0 6 0 4 6 5 3
black pegs: 1  white pegs: 6
attempt:
0 5 3 6 0 4 6
black pegs: 7  white pegs: 0
the solver has found the sequence in 6 attempts
the sequence generated by the code maker was:
0 5 3 6 0 4 6
Elapsed time: 0.044876 s
```

SINGLEY'S ALGORITHM

All in all, Singley's algorithm proved to be very easy to implement and consistently provided an output.

The range of the algorithm is somewhat inconsistent. When tested over 1000 times the maximum number of guesses was 88, and the minimum number of guesses was 32. The theoretical worst case is 122 (refer to the paper for explanation).

The example on the right displays the algorithm in action, going through the initial list. It has then selected the number with the largest number of black hits (in this case 3,5,9 and 10 are all equal but since 3 is the first number it is defined as the base).

After this, one can see that the algorithm initially tries one, but then sees that the number of black hits then goes down, meaning that the first index has the value three.

The algorithm then proceeds to check the next few numbers, with two and five being avoided as the initial set of guesses determined that they were not present in the final set.

IMPROVEMENTS

Perhaps one way to improve the algorithm is to only send guesses that are theoretically possible. For example, if we know that there is a single zero in the final set after our first guess, it makes more sense to then place a single zero in the second set rather than the guesses being all ones. This was decided against because of the time it would then require did not seem worth the minor improvement.

If the previous black hit values were placed in the next guess, it could mean that the heuristic could then be used to rearrange the algorithm. The pre-processing would change to what was mentioned in the earlier paragraph, and this could then have been stored in "guessVec" along with the feedback in "feedbackVec". Temporels algorithm would then be introduced to rearrange the vector, with data about the different positions being provided by the random behaviour of how the previous black hits are arranged. This could result in the attempt count reducing significantly.

```
attempt:
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
black pegs: 0  white pegs: 0
attempt:
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
black pegs: 2  white pegs: 0
attempt:
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
black pegs: 1  white pegs: 0
attempt:
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
black pegs: 2  white pegs: 0
attempt:
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
black pegs: 1  white pegs: 0
attempt:
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
black pegs: 1  white pegs: 0
attempt:
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
black pegs: 0  white pegs: 0
attempt:
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
black pegs: 2  white pegs: 0
attempt:
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
black pegs: 2  white pegs: 0
attempt:
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
black pegs: 0  white pegs: 0
attempt:
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
black pegs: 1  white pegs: 0
attempt:
13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
black pegs: 1  white pegs: 0
attempt:
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
black pegs: 1  white pegs: 0
0
attempt:
1 3 3 3 3 3 3 3 3 3 3 3 3 3 3
black pegs: 1  white pegs: 2
1
attempt:
3 1 3 3 3 3 3 3 3 3 3 3 3 3 3
black pegs: 2  white pegs: 1
1
attempt:
3 4 3 3 3 3 3 3 3 3 3 3 3 3 3
black pegs: 2  white pegs: 1
1
attempt:
3 5 3 3 3 3 3 3 3 3 3 3 3 3 3
black pegs: 2  white pegs: 1
1
```

```
enter length of sequence and num
9 9
8 8 5 5 3 1 8 7 2
attempt:
8 8 7 1 6 4 0 4 7
black pegs: 2  white pegs: 2
attempt:
2 3 7 2 5 6 8 6 7
black pegs: 1  white pegs: 4
attempt:
0 2 7 3 6 2 3 0 4
black pegs: 0  white pegs: 3
attempt:
5 8 1 1 2 7 8 5 2
black pegs: 3  white pegs: 4
attempt:
3 8 1 1 5 8 6 2 5
black pegs: 1  white pegs: 6
attempt:
8 8 5 5 2 6 5 1 2
black pegs: 5  white pegs: 1
attempt:
0 0 0 0 0 0 0 0 0
black pegs: 0  white pegs: 0
attempt:
1 1 1 1 1 1 1 1 1
black pegs: 1  white pegs: 0
attempt:
2 2 2 2 2 2 2 2 2
black pegs: 1  white pegs: 0
attempt:
3 3 3 3 3 3 3 3 3
black pegs: 1  white pegs: 0
attempt:
4 4 4 4 4 4 4 4 4
black pegs: 0  white pegs: 0
attempt:
5 5 5 5 5 5 5 5 5
black pegs: 2  white pegs: 0
attempt:
6 6 6 6 6 6 6 6 6
black pegs: 0  white pegs: 0
attempt:
7 7 7 7 7 7 7 7 7
black pegs: 1  white pegs: 0
attempt:
8 8 8 8 8 8 8 8 8
black pegs: 3  white pegs: 0
0
attempt:
1 8 8 8 8 8 8 8 8
black pegs: 2  white pegs: 2
1
attempt:
8 1 8 8 8 8 8 8 8
black pegs: 2  white pegs: 2
2
```

As mentioned earlier, a time limit has been implemented so that if a solution has not been found after nine seconds, Singley's algorithm is used. This example is shown on the left. It means that though taking the risk will cause a greater number of attempts if unsuccessful, the overall average will decrease considerably due to the successful attempts. As a result, the implementation improves efficiency.

The timing method works using the <ctime> library, with the starttime being initiated in the init function.

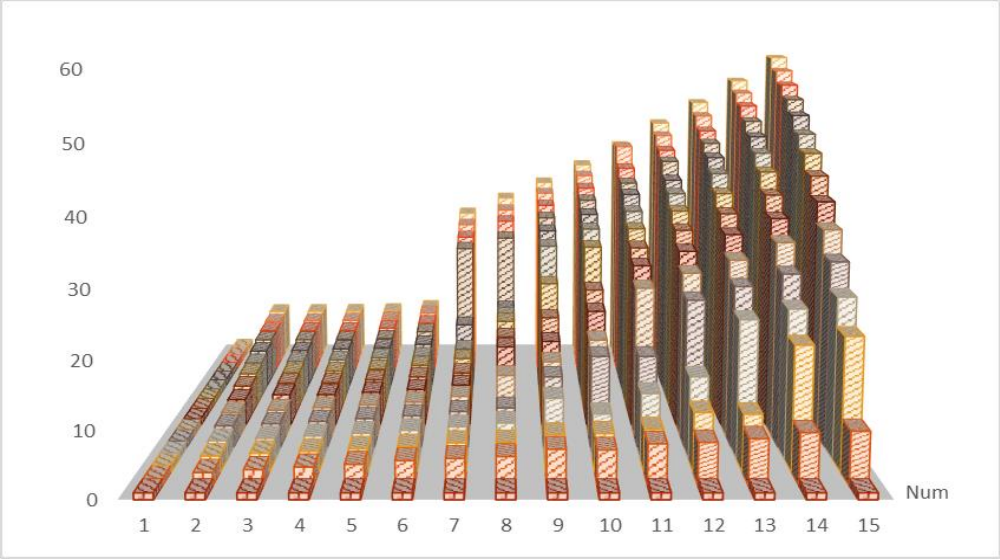
```
finish = std::time(0);
timetaken = std::difftime(finish, start);

if(timetaken > 9){
  switchToSingley = true;
  cfg.clear();
  attempt.clear();
}
```

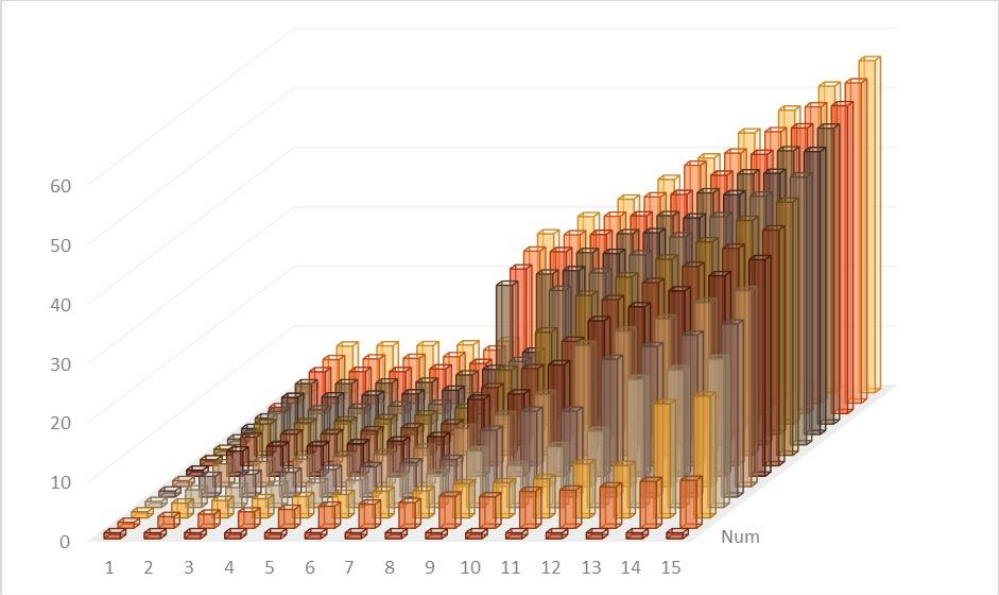
The function works extremely effectively, and means that the latest the program takes to run is around 9.8 seconds, just below the time limit. It takes full advantage of the time limit provided. The pace of Singley's algorithm means that Temporel's algorithm has around 9 seconds to calculate the answer.

In reality, the algorithm would be even better if faster, as some Temporel results do lie within the 9-10 second range. The 0.2 second buffer is there primarily so that the code will always be within the 10 second margin, even in the worst case scenario.

	Num														
Length	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1.96	2.49	2.95	3.46	3.812	4.21	4.815	5.274	5.628	6.2	6.75	7.02	7.32	7.84
3	1	2.37	2.908	3.34	3.8	4.212	5.032	5.32	5.465	5.868	6.3	6.77	7.06	7.432	7.88
4	1	2.76	3.21	3.69	4.17	4.64	5.05	5.36	5.75	6.16	6.59	6.88	7.06	7.519	7.92
5	1	3.18	3.58	4.13	4.61	5.05	5.44	5.87	6.12	6.52	6.81	7.01	7.501	7.808	8.04
6	1	3.71	3.93	4.56	5.05	5.51	5.892	6.36	6.77	6.8	7.42	8.24	8.42	8.92	8.65
7	1	4.04	4.48	5.12	5.72	6.44	6.64	7.04	7.92	8.08	10.9	23.315	24.306	25.551	26.7
8	1	4.32	4.56	5.7	6.3	9.8	12.9	13.1	14.3	14	13.8	25.2	27.2	28.284	29.582
9	1	5.4	5.7	9.5	11.2	12	13.8	16.3	20.7	25.939	27.516	28.8	30.1	31.402	32.52
10	1	5.29	5.9	7.1	14.4	15.5	18.7	20.9	26.88	28.89	30.4	31.95	33.22	34.66	35.84
11	1	6.2	6.56	10.24	14.4	23.74	26.106	27.88	29.96	31.909	33.89	35.01	36.82	39.97	39.45
12	1	6.4	9.05	12.8	23.135	26.123	28.439	30.766	32.938	34.928	36.377	38.842	40.054	42.015	43.655
13	1	6.95	8.8	21.503	25.255	28.174	31.118	33.481	35.85	38.327	40.279	42.025	43.557	45.607	47.477
14	1	7.85	19.131	23.127	27.179	30.929	33.666	36.518	39.464	41.795	43.875	45.902	47.995	49.793	51.52
15	1	8.05	20.429	24.932	29.049	32.926	36.371	39.608	42.527	45.014	47.529	49.697	51.727	53.803	55.794

The table above shows the average number of attempts within each step. Each of the averages has been calculated from at least 250 attempts (with some being calculated from over 5000 iterations) The yellow section denotes the reigion in which Singley's algorithm operates from the very beginning. These were determined experimentally. Below is a visual representation of the attempts distribution





APPENDIX

https://pdfs.semanticscholar.org/2e26/663970ac3e11ba9cd15c2d9671cf6216b5ad.pdf

- Temporel's Algorithm

https://dspace.library.uu.nl/handle/1874/367005

- Comparison of various algorithms

http://etd.fcla.edu/UF/UFE0010554/singley_a.pdf

- Singley's Algorithm