Mindefuse - Hostage Crisis Defusal Agent

Henrique Araújo
Instituto Superior Técnico
henriquecmaraujo
@tecnico.ulisboa.pt

Nuno Amaro
Instituto Superior Técnico
nuno.amaro
@tecnico.ulisboa.pt

Tiago Gonçalves
Instituto Superior Técnico
tiago.r.goncalves
@tecnico.ulisboa.pt

ABSTRACT

In a hostage negotiation setting, humans have always been preferred over computational agents, mostly due to the unpredictability of the captors. A human agent, capable of empathy, active listening and persuasion can be able to overcome this unpredictability and successfully defuse an hostage situation in a non-violent manner. A computational-based agent has yet to demonstrate the capability to perform at the required level to replace its human counterpart. Our proposal aims to study a promising solution for the computational performance of an autonomous agent attempting to defuse an hostage crisis, using an approach analogous to the popular board game Mastermind [1].

1. INTRODUCTION

The proposed agent system encompasses an autonomous *negotiator* agent able to uncover a secret sequence, detained by another, reactive, *captor* agent. This secret sequence represents the mindset of the *captor* during the negotiation. The goal is to determine the correct sequence by gathering clues from the *captor*. However, this goal must be achieved within a reasonable time frame, as the *captor* will eventually run out of patience for the negotiation attempts and harm the hostage. Different *captors* might be more or less permissive than others, bringing another element of difficulty for the agent to overcome. In order to accomplish this, we will develop and compare several condition satisfiability strategies like **Knuth**, **Genetic** and **Swaszek**, considering both effectiveness and time efficiency. Afterwards, a **Hybrid** strategy will be devised, based on the evaluation of the previously cited strategies.

2. PROPOSAL

A Mastermind game involves two players where one defines a secret sequence with 4 of the 6 different coloured pegs available, while the other tries to uncover the sequence throughout a set of 12 rounds. To uncover the secret sequence, in each round, the player proposes a sequence and the adversary responds to the attempt by placing a white marker for every correct coloured peg in a wrong position and a red marker for all correct coloured pegs in a correct position, without stating which marker refers to each peg.

Considering the size of the sequence, s, and the available colours c, the number of possible secret sequences would result in c^s. On the original game, it corresponds to $s = 4$ and $c = 6$, which results in 1296 possible sequences. The guesses and feedback must be used to successfully restrict the pool of possible sequences, as to reduce the number of rounds required to uncover the secret.

Bondt [3] proved that solving a Mastermind board is an NP-complete problem, by proving that any one-in-three 3-SAT can be represented in it, thus, although a solution can be verified in polynomial time, there is no known way to find a solution in polynomial time.

Providing that the *negotiator* does not know the number of available rounds, which corresponds to the permissiveness of the *captor*, we mainly intend to minimise the number of guesses required to discover the secret combination.

Considering that the *captor* agent is an abstraction that only provides clues for his secret in a deterministic and simple manner, we will only define the properties of the *negotiator*, which possesses more complex behaviour.

Table 1: Agent Properties

Autonomy	**Adaptivity**	**Rationality**	**Curiosity**
✓	✗	✓	✓
Reactivity	**Proactivity**	**Sociability**	**Collaboration**
✓	✗	✓	✗
Believability	**Mobility**	**Personality**	**Veracity**
N/A	N/A	N/A	N/A

Table 2: Environment Properties

Accessibility	**Determinism**	**Dynamism**
Accessible	Deterministic	Static
Continuity	**Memory**	
Discrete	Non-Episodic	

Each of the developed strategies will be evaluated based on the time and number of rounds taken to successfully uncover a secret, and compared with each other to find the best algorithm for each secret sequence type and size.

3. STRATEGIES

The presented strategies have been developed following the specifications defined in the original articles. We present the default algorithm order for explanation clarity, minor

0

algorithm reordering and other optimisation methods have been used for enhanced computational performance.

A white marker corresponds to a correct coloured peg in a wrong position, while a red marker corresponds to a correct coloured peg in a correct position. In each round, a sequence is proposed as an attempt to guess the secret sequence. For each guess a (w, r) tuple is returned, where w is the number of white markers and r is the number of red markers. For a given secret sequence c of size s and some guess g the number of whites, w, can be given by

$$w = (\sum_{i=1}^{s} \min(c_i, g_i)) - r$$

where c_i is the number of times the element i is in the secret sequence and g_i is the number of times it is in the guess.

The comparison of two equally sized sequences will be made with procedure $COMPARE(s_1, s_2)$, which returns a (w, r) tuple. The submission of a guess g will be represented by procedure $PLAY(g)$, which uses $COMPARE(c, g)$ to match a guess with the secret sequence, returning a (w, r) tuple.

3.1 Knuth

The original algorithm proposed by Donald Knuth [4] can uncover a secret of size 4, with 6 available colours in 5 or fewer rounds by progressively submitting the guess with the maximum expected utility value. Taking into account that on our base case we consider that 10 colours are available, the secret is uncoverable in less than 7 rounds instead of 5.

The *initialGuess* proposed by Knuth was always *0011* for the default game configuration. However, considering that we intend to generalize the solution, we use the initial guess given by $\sum_{i=0}^{s} E_i * (i + 2)$, right cropped to match the secret size, where E corresponds to the ordered set of all the possible elements of a sequence. Concretely, for a secret of size 3, the first guess would be *001*, while for a secret of size 6 the first guess would be *001112*.

The expected utility is based on the returned (w, r) tuple received upon each guess commit, concretely, $\mu((w, r)) = w + r$ and is represented by μ in algorithm 1. Minimax, algorithm 1, is used to obtain the set of guesses of expected maximum utility.

Algorithm 1 Minimax

```
1: procedure UTILITYSCORE(c, S)
2:     scores ← empty dictionary
3:     for all s ∈ S do
4:         score ← μ(COMPARE(c, s))
5:         scores[score] ← scores[score] + 1
6:     end for
7:     return max scores
8: end procedure
9:
10: procedure MINIMAX(S, A)
11:     scores ← empty dictionary
12:     for all c ∈ A do
13:         scores[c] ← UTILITYSCORE(c, S)
14:     end for
15:     return arg min scores
16: end procedure
```

Algorithm 2 Knuth

```
1: procedure BESTGUESS(B, S, A)
2:     G ← {g ∈ B | g ∈ S}
3:     if G = ∅ then
4:         G ← {g ∈ B | g ∈ A}
5:     end if
6:     return min G
7: end procedure
8:
9: procedure KNUTH(p)
10:     S ← S_n
11:     A ← S_n
12:     g ← initialGuess
13:     while GAME do
14:         r ← PLAY(g)                          ▷ play guess
15:         S ← S \ g                  ▷ remove used guess from S
16:         A ← A \ g                  ▷ remove used guess from A
17:         S ← {x ∈ S | COMPARE(x, g) = r}          ▷ prune S
18:         B ← MINIMAX(S, A)          ▷ best scored sequences
19:         g ← BESTGUESS(B, S, A)     ▷ choose best guess
20:     end while
21: end procedure
```

Knuth demonstrated that the code can be uncovered using algorithm 2, which progressively reduces the number of possible patterns and can be described as:

1. Create a set S, the solution space, and a set A, all the unused codes, from the set of all possible codes S_n, which contains c^s elements.
2. Generate an initial guess.
3. Play the guess, g, and obtain the corresponding value of whites and reds, r.
4. Verify if the game is over, if the maximum number of rounds has passed or if number of reds is equal to the size of the secret, if it is, the game is won. Thus, $GAME$ will become $False$.
5. Remove g from S and from A.
6. Prune from S any code that would not give the same response, r, if it was the secret sequence.
7. Apply $miniMax$ to find the set of next possible guesses. For each code in A, calculate how many possibilities in S would be eliminated for each possible white + red peg score, μ.
8. Choose the next guess g, from B, by selecting, if possible, the lowest value also present in S, otherwise from A.
9. Repeat from step 3.

The score of a code represents the maximum number of sequences that would still need to be tested if the worst case scenario (w, r) was returned, after that code was played as a guess. Reversely, a low score indicates that, for the worst case, the solution space would still get very narrowed down, thus, the minimum number of possibilities it might eliminate from S. The most desirable score is the lowest one, as it also

represents the highest utility score, since it assures that the solution space got as narrowed down as possible.

The algorithm is able to attain an optimal result, but at the cost of high time and memory complexity. Considering the initial solution space, S_n of size c^s, the time complexity of the algorithm corresponds to $\mathcal{O}((S_n)^2)$, considering that for obtaining the expected utility of each guess, it must be compared with all other possible sequences, which implies $\mathcal{O}(N^2)$ complexity. The theoretical memory complexity corresponds to $\mathcal{O}(S_n)$. Most of the guesses will end up being pruned, but it will only happen after the first guess is played.

Considering these efficiency values, one of the implemented improvements consists in distributing the workload of utility estimation among several cores. With this improvement each core will concurrently compute the score only for a portion of the combinations of A from algorithm 1, line 12. Thus, we are able to substantially amortize the algorithm complexity in accordance to the number of available cores.

3.2 Swaszek

The algorithm proposed by P.F. Swaszek [5], described in Algorithm 3, consists in generating a list of all possible sequences, randomly picking a guess among them, submitting the guess and remove all the choices that could not have given the same outcome, by comparing them with the chosen proposal and checking if the outcome is the same as the evaluation.

Algorithm 3 Swaszek

```
1: procedure SWASZEK(p)
2:     S ← S_n                                  ▷ generate all possible choices
3:     while GAME do
4:         g ← RANDOMGUESS(S)                   ▷ get random guess
5:         r ← PLAY(p, g)                       ▷ play guess
6:         S ← S \ g                            ▷ remove used guess from S
7:         S ← {x ∈ S | COMPARE(x, g) = r}      ▷ prune S
8:     end while
9: end procedure
```

Algorithm 4 Multi-Agent Swaszek

```
1: procedure MULTIAGENTSWASZEK(p)
2:     S ← S_n                                  ▷ generate all possible choices
3:     agents ← GENERATEAGENTS                  ▷ create list of agents
4:     p_best ← p                               ▷ variable to store best problem
5:     while GAME do
6:         r_best ← empty                       ▷ variable to store best proposal
7:         for agent in agents do
8:             p_a ← p                          ▷ duplicate the problem
9:             g ← RANDOMGUESS(S)               ▷ get random guess
10:            r ← PLAY(p, g)                   ▷ play guess
11:            if BESTGUESS(r_best, r) then
12:                p_best, r_best ← p_a, r
13:            end if
14:        end for
15:        S ← S \ g                            ▷ remove used guess from S
16:        S ← {x ∈ S | COMPARE(x, g) = r}      ▷ prune S
17:    end while
18:    RETURN(p_best)                           ▷ return when finished or solved
19: end procedure
```

On our implementation of the Swaszek algorithm we developed a Multi-Agent system composed of three agents, described in Algorithm 4, each with a different method of choosing a proposal from the list of possible choices.

The first agent chooses a random proposal from the list of possibilities. The second agent always chooses the first element in the list. The third agent iteratively chooses the elements on the list,starts with the first element, then the second, then the third and so on, until it reaches the end of the list, in which he starts from the first element again.

Each proposal is evaluated, and the best choice out of the three is the one which will be applied to the problem, updating the list of choices, thus counting as if only the agent with the best proposal was the one playing that round, after they all debated which of the proposals would be better, as opposed to having a single agent making a choice each round.

Algorithm 5 Proposal Comparison

```
1: procedure BESTGUESS(p1, p2)
2:     p1_pegs ← TOTALNUMBEROFPEGS(p1)
3:     p2_pegs ← TOTALNUMBEROFPEGS(p2)
4:     if p2_pegs > p1_pegs then
5:         RETURN(True)
6:     end if
7:     if p2_pegs == p1_pegs then
8:         if WHITEPEGS(p2) > WHITEPEGS(p1) then
9:             RETURN(True)
10:        end if
11:    end if
12:    RETURN(False)
13: end procedure
```

The heuristic used to determine the best choice is the sum between the amount of red and white pegs. In case of a draw the heuristic used is the amount of white pegs. This means that we choose the proposal with the most amount of pegs, and if there are multiple proposals with the same amount of pegs, we choose the one with the most white pegs. If it still ends in a draw the chosen proposal is the first one to be made. The algorithm used to compare the guesses is described in Algorithm. 5.

3.3 Genetic

The algorithm described by Berghman et al. [2] presents a solution for the problem using an embedded *genetic algorithm*, which is inspired by the process of *natural selection*. This kind of algorithm belong to a larger class of *evolutionary algorithms* and *evolutionary computation*. Being inspired by natural selection and biological evolution, genetic algorithms rely on bio-inspired operators such as crossover, mutation and selection. Beyond the presented strategy, these algorithms can be used in many different applications, such as implementations of ant and bee colonies with the goal of optimizing the agents' behaviour, known as swarm algorithms, and other population-based metaheuristic problems such as wolf pack hunting methods.

As with the algorithm by Knuth, our genetic algorithm strategy, described as Algorithm 6, also uses a fixed initial guess, which in our implementation follows the behaviour described in 3.1.

For each round of guessing, we first start by initializing a population with randomly generated sequences. This initial population will then go through the evolution process,

Algorithm 6 Genetic Strategy

Set $i = 1$;
Play fixed initial guess $\mathbf{g_1}$;
Get response X_1 and Y_1;
while $X_i \neq P$ **do**
  $i = i + 1$;
  Set $\widehat{E}_i = \{\}$ and $h = 1$;
  Initialize population;
  while ($h \leqslant maxgen$ AND $|\widehat{E}_i| \leqslant maxsize$) **do**
    Generate new population using crossover, mutation, inversion and permutation;
    Calculate fitness;
    Add eligible combinations to $\widehat{E}_i$ (if not yet contained in $\widehat{E}_i$);
    $h = h + 1$;
  end while
  Play guess $\mathbf{g}_i \in \widehat{E}_i$;
  Get response X_i and Y_i;
end while

where elements of this population will be seen as parents to the newly generated offspring guesses. Our implementation of the evolution follows the same process described by Berghman et al.: To start, either a two-point or one-point crossover (both with a probability of 0.5) is applied to two parent guesses in order to generate their children. These can then go through a mutation (with a probability of 0.03), where an element in a randomly chosen position is replaced by a random possible element. Additionally, children can go through a permutation, where two elements of the guess are switched, with a probability of 0.03. The final possible bio-operator applied, with a 0.02 probability, is an inversion whereupon two positions of the guess are randomly chosen and the order of elements between those positions is inverted. If after all these operations the child is already present in the population, we randomly generate a guess in order to diversify the offspring population. The *fitness score*, computed by a fitness function, corresponds to the utility of a given guess. Berghman et al. defined this fitness function as:

$$f(c,i) = a(\sum_{q=1}^{i}|Red'_q(c) - Red_q|) + (\sum_{q=1}^{i}|White'_q(c) - White_q|) + bP(i-1)$$

where b and a are empirical constants, P the number of positions, i the number of turns played, and the sums are of the differences between the pegs between the trial code if the previous guesses were the secret. In our implementation, however, we have opted for a simpler fitness function, only accounting for the sum of the sums of differences of each pegs if the previous guesses were used as secret:

$$f(c,i) = \sum_{q=1}^{i}|Red_q(c) - Red_q| + \sum_{q=1}^{i}|White'_q(c) - White_q|$$

This fitness score is used to gauge whether a child, , is an eligible code or not. We consider a code to be eligible when $f(c,i) = 0$. To account for higher complexities, we also scale the initial base $maxsize$ and $maxgen$ from the original 60 and 100, respectively, until we find a list of eligible guesses we can use. Finally, a decision must be made about which of the eligible codes will be played as the following guess. Berghman et al. described three alternatives to this decision: choose the most similar code to the other eligibles, choose the most different code to the other eligibles or choose a random eligible code. For our implementation we have chosen the third alternative in order to minimise the computational cost.

4. EVALUATION

Taking into account that a secret of size 1 will never provide the number of whites, and thus is not compliant with the default game rules we will refrain of taking secrets of that size into during evaluation. Intuitively, the best strategy to uncover those kind of secrets is *Swaszek*, as *Genetic* is dependant on the number of whites to calculate the fitness function and *Knuth* would require, for the worst case, a number of rounds equivalent to the number of possible elements of the secret, thus $\mathcal{O}(S)$.

The strategies will be evaluated against the 3 different types of secrets available, which are formed by elements of the one of these sets:

numeric numeric values [0...9], 10 elements

lstring lower case strings [a...z], 26 elements

string lower and upper case strings [a-zA-Z], 52 elements

For each of these secret types, a set of 10 sequences of sizes 2 to 4 will be randomly generated.

4.1 Results and Analysis

Given the implied computational time constraints of the algorithm *Knuth* can not be timely tested and evaluated for secrets of type *string*, and secrets of type *lstring* of size superior to 2, thus those cases will be refrained from analysis. Furthermore, due to lack of space in accordance to the defined size restrictions of the report, we are unable to provide data about the strategies before applying our improvements, even though some of these results will be referred to during this analysis.

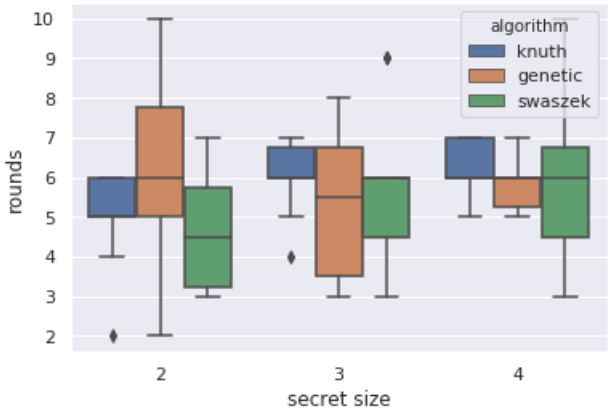

Figure 1: Numeric secret - Distribution of rounds per secret size

By relating the secret size with the number of rounds we may infer which algorithm is more fit for situations where the *captor* tends to be impatient, thus, the number of available rounds is low. Overall *Knuth* presents the most narrow intervals, and good results for numeric secrets, fig. 4.1, however, as previously mentioned, when the size of the solution

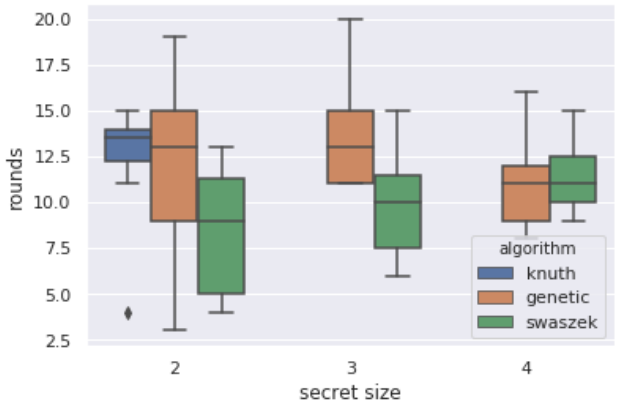

Figure 2: Lstring secret - Distribution of rounds per secret size

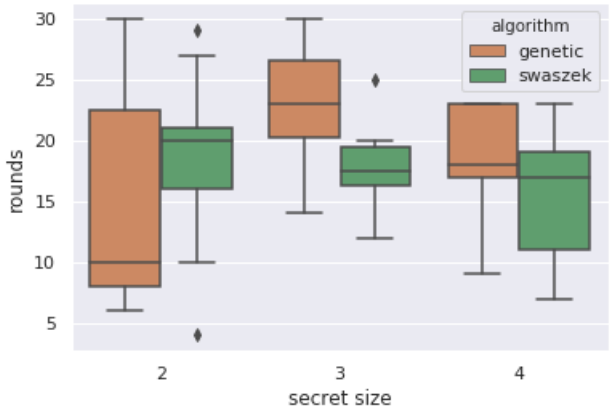

Figure 3: String secret - Distribution of rounds per secret size

space increases, it begins to under perform. *Genetic* seems to be unfit for solving secrets of size two, which is aligned with the rationale that its crossover and mutation can not be properly applied for individuals of size two. However, it seems to perform successively better as the secret size increases, considering that the interval of required rounds starts to converge and mainly reduce, when compared with the other two strategies. From fig. 4.1 we can verify that by a secret of size 4, *Genetic* is already starting to outperform *Swaszek*. *Swaszek* presents low number of rounds from the start, however, as the size of the secret increases, so does the interval of rounds required to find a solution, which is most notable in for secrets of size 4 in fig. 4.1.

The time taken to answer, even if not directly related to the problem, is of the utmost importance in the real life setting. An agent should not take too long to reply or it would not be useful when considering a real time conversation with the captor. As we can see in fig. 4.1, *Knuth* clearly suffers from its time complexity value, since it starts to rise past *Genetic* for secrets of size superior to 3. *Genetic* time value visibly differs between lstring and string types of secret. *Swaszek* presents a seemingly constant increase of time taken to uncover the secret, as one can see by compar-

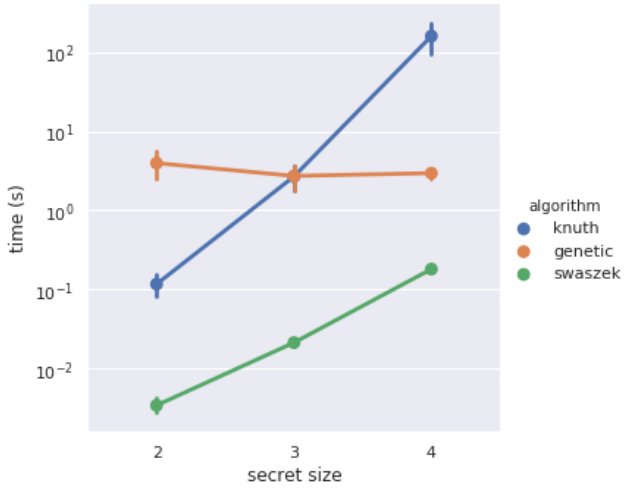

Figure 4: Numeric secret - Average time per secret size

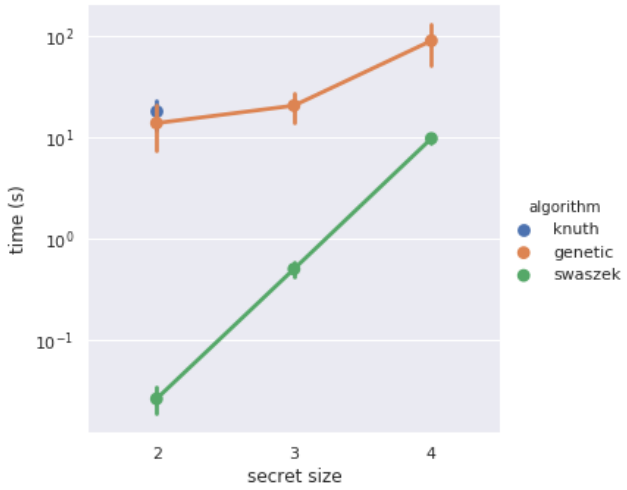

Figure 5: Lstring secret - Average time per secret size

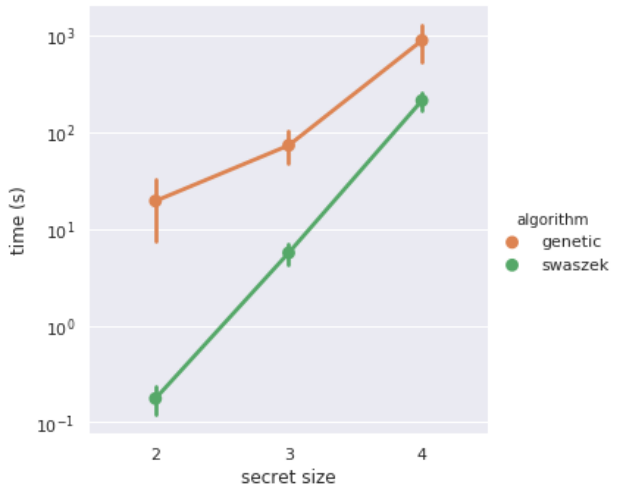

Figure 6: String secret - Average time per secret size

ing fig. 4.1 with fig. 4.1.

In order to feasibly evaluate our solutions and provide more precise metrics, cluster or cloud based computational power would be required, however, that option was unavailable, given the existing time constraints and context of the project.

Note that, *Swaszek* with multiple agents now has increased performance compared to the unimproved strategy. Previously, the best approach was to use *Genetic* for secret sizes superior to 3, but now, further testing would be required to uncover which of them would have the upper hand for secrets bigger than 4.

4.2 Hybrid Agent

Based on the gathered metrics and results, an hybrid agent, which evaluates the problem and chooses the best strategy to use in order to uncover a secret, was defined using the following set of rules:

- secrets of size 1 and 3 are solved with Swaszek
- secrets of size 2, of types numeric and lstring are solved with Knuth
- secrets of size 4 are solved with Swaszek, unless numeric types, which are solved with Genetic
- secrets of size superior to 4 are solved with Genetic

By using these rules, the agent is expected to perform as well as the best of all 3 strategies would when facing the same problem. Considering that it may fall out of the scope of the project, no evaluation tests were performed to assure that the optimal rules have been applied to define this agent.

4.3 Possible Improvements

Genetic could use multiprocessing or apply an approach similar to what happens in *Swaszek*, using communication between several agents running the strategy. Furthermore, a more precise population size estimation function, based on the size and expected complexity of the problem could be developed.

The *Hybrid* agent could employ *machinelearning* techniques and learn to choose the best strategy to employ in each received problem, instead of relying on previously defined thresholds and empirical values.

5. CONCLUSION

In this project we were able to successfully relate and employ an emotional and logical abstraction of what the human mind. We devised 3 strategies that employed several concepts lectured during the course, concretely, decision theory with *Knuth*, *Genetic* with reinforcement learning and communication between agents in our *Swaszek* strategy. An *Hybrid* agent was devised based with the intention of harnessing the best properties and behaviour of the other strategies. Furthermore, during the implementation of the architecture, the fundamental concepts of agent design methodologies were followed as to correctly devise functional, modular and extensible agents.

REFERENCES

[1] Mastermind board game. https://en.m.wikipedia.org/wiki/Mastermind_(board_game). Accessed on 2019-03-29.

[2] L. Berghman, D. Goossens, and R. Leus. Efficient solutions for mastermind using genetic algorithms. *Comput. Oper. Res.*, 36(6):1880–1885, June 2009.

[3] M. de Bondt. "p-completeness of master mind and minesweeper. *onbekend : Radboud Universiteit Nijmege*, 2004.

[4] D. E. Knuth. The computer as mastermind. *Journal of Recreational Mathematics*, 9:1–6, 01 1976.

[5] P. F. Swaszek. The mastermind novice. *journalId:00001601*, 30, 01 2000.