Lineage Guide-Matrix Algorithms

Robin Wopaleński, Natan Arnold
December 4, 2023

Citationless Copy

Abstract

In this paper, we propose two lightweight machine learning algorithms, comprehensively exploring an implementation of Ant Colony Optimisation (ACO) for global fitness based cryptanalysis. The first LGM algorithm is with the initial purpose of improving on the performance and utility of the genetic algorithm and/or simulated annealing in cracking a monoalphabetic substitution cipher. Presented are some helpful modifications to the basic method to improve performance specifically for the cryptanalsysis of substitutions. We also use a variant algorithm that can be used to effectively solve large permutation ciphers (and other simple related transpositions), and adapt it to solve larger (NP-hard) permutations.

---

1 Position Probability Matrix (ppM) Algorithm

1.1 Introduction

In the cryptanalysis of substitution ciphers, the keyspace of the cipher is very large (grows with $n!$), so brute-force attacks are impractical. A widely used algorithm for cracking a monoalphabetic (size 26) substitution cipher is the hill-climbing algorithm. While this algorithm significantly improves on the brute-force attack, it leaves much to be desired. Firstly, it has trouble with time complexity, failing to solve substitutions with larger character spaces (e.g. size 676 substitutions on alphabetic bigrams) in any meaningful time. Secondly, its solving rate decreases as the algorithm approaches the correct answer, meaning it is unlikely to get the exact answer reliably in some given number of generations, and will instead get a key that is a few transpositions away. Finally, the number of generations required to get a desirably close key is typically large.
The main pitfalls of the hill-climbing algorithm are its carelessness with information and the random nature of its evolution. When a series of children keys are spawned and their fitness tested, only the child with the highest fitness is used to iterate the algorithm. All of the information about the fitness change with the other spawned children is completely disregarded. This is an inefficiency which our method aims to address. Additionally, the spawning of the children keys is based on a uniform sampling distribution: all children are equally likely. This results in the convergence problem mentioned before: as the parent key gets closer to the correct key, the likelihood of spawning a better child decreases greatly, so the solving rate decreases.
Lineage Guide-Matrix algorithms are a subset of the more general ACO method, and propose an enhancement of the hill-climbing algorithm to solve monoalphabetic substitutions. Though the implementation perspective is slightly different, later the link between ACO and LGM is made clear. The LGM algorithm makes use of the previously discarded information and the child sampling process is no longer uniformly stochastic, but instead evolves dynamically over generations.

1.1.1 Substitution Ciphers

Substitution ciphers replace tokens in a text string. These tokens can, for example, be the individual characters of the English alphabet, in which case there are 26 tokens to substitute and the substitution is known as monoalphabetic. Another valid substitution cipher is one that substitutes all possible English bigrams (pairs of letters). Substitutions form a group under the operation of composition, which is isomorphic to the Permutation group. All substitutions can be expressed as a permutation, in two line notation, with an enumeration of the constituent tokens. For instance, the following monoalphabetic substitution:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	A	B	C	M	N	O	P	Q	R	S	Z	Y	V	X	W	U	T

encrypts *"It was a bright cold day in April and the clocks were striking thirteen"* to become *"Lz xds d erljkz focg gdu ln Dprlc dng zkh fcofbs xhrh szrlblnj zklrzhhn"* and can be represented as the permutation (using the enumerated alphabet $A \to 1$, $B \to 2$...):

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 \\ 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 1 & 2 & 3 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 26 & 25 & 24 & 23 & 22 & 21 & 20 \end{pmatrix}$$

... and given the enumeration, the first line isn't necessary at all, and the second line can be considered the key. It follows that there are $n!$ possible size-n substitution keys. For this paper, reference to $\phi(s) = \{z \mid \text{transpositiondist}(s, z) = 1\}$ is the set of all substitution keys resulting from swapping a single pair of tokens in s.

1.1.2 The Hill-climbing Algorithm

The hill-climbing optimisation algorithm is used in breaking monoalphabetic substitutions. It involves a lineage, or sequence, of candidate keys s_n. The first parent key s_0 is initialised. For each generation n, the current parent s_n is used to create a new set of children keys. The fitness of these children keys, along with the parent, is tested and the key (out of all children and the current parent) with the highest fitness becomes the new parent s_{n+1}. The child spawning method is a swap function, that chooses two random positions in the key and swaps their entries (a sequence transposition). A commonly used fitness function for determining how close text is to English is quadram logarithm fitness (quadgramlog for short). Quadgramlog checks quadgrams (groups of four consecutive letters) in the text against an expected frequency database (calculated from a large English text corpus). The quadgramlog score of a string is the average of the logarithm of the expected frequency f_q for each quadgram q in the string:

$$F_Q = \frac{1}{N} \sum_q \log(f_q) \tag{1}$$

... where N is the number of quadgrams (length(text) − 3)

Algorithm 1 The Hill-climbing Algorithm

```
Initialise parent
F ← fitness(parent)
for g generations do
    children ← generate s mutations of parent
    F_best ← max(fitness(children))
    if F_best > F then
        F ← F_best
        parent ← child whose fitness is F_best
    end if
end for
return parent
```

1.1.3 Genetic Algorithm and Simulated Annealing

Genetic algorithms are also commonly used to break substitution ciphers. Genetic algorithms differ to the regular hill-climbing because there are multiple parents in a lineage and their attributes can be merged to make a new key. The ppM algorithm is tested against hill-climbing rather than the genetic algorithm because the ppM can only make one modification to the key per generation, whereas the genetic algorithm can make multiple. The performance of genetic algorithms is later addressed, once the ppM algorithm is tested against hill-climbing. The genetic algorithm can be modified with a simulated annealing component, which adds a temperature variable that decreases with generations. Choosing the next parent key is no longer deterministic; a higher temperature means that the key with the highest fitness is less likely to be selected as the next parent. This provides a good solution to escaping local maxima, but is not needed in the solution to monoalphabetic substitutions, as local maxima do not pose a large issue with quadgramlog and the simulated annealing adds some computing cost.

1.2 Motivation

However many children are spawned, the hill-climbing algorithm only extracts information from one key each generation - the next parent - and this information is only that it is the best key of that generation. This discards not only the information in the magnitude of the fitness change, but also the resulting fitness changes from all of the discard spawned keys. The ppM algorithm aims to use that discard information to converge to the solution faster.

The basis of the algorithm is that, as in the hill-climbing case, there exists a solution lineage, but this lineage is "supervised" by a guide matrix, which learns from the information revealed in the lineage's progression. At first, the guide matrix has a low variance and "guides" the lineages progression with little certainty, so the lineage progresses as a stochastic hill-climb search. As the lineage progress and new keys are spawned and discarded or approved, the guide matrix will learn and its variance will increase. Thus, the guide matrix will influence the lineage's progression and "steer" it towards the solution with the information it has gained from supervising it. When the lineage nears the solution, it enters a fitness maximum, so the guide matrix continues to "reinforce" its standing guidance, until the variance approaches one, and the algorithm has permanently converged.

To spawn the next generation's children, the guide matrix is used as a probability weighting for sampling a space with all possible swaps between two tokens in the substitution key, each producing an element of $\phi(s_n)$. If the guide matrix has a large value for a particular swap - because the information gathered suggests it is correct and appears in the solution - that swap is more likely to be chosen; this is how the guide matrix can "steer" the lineage. Additionally, as the variance in the guide matrix increases, the probability of choosing swaps that the matrix deems to be incorrect decreases, so the algorithm is self-annealing. As the algorithm progresses and the guide matrix converges on the solution, the sampling becomes more deterministic.

1.2.1 The Position Probability Matrix

The ppM (or Position Probability Matrix) is introduced as the guide matrix for this method. It is an $n \times n$ matrix of values in the interval $[0, 1]$. Denoting the ppM with P, the element P_{xy} represents the estimated probability that item x is in position y in the solution of a general ordering problem, where $x \in X = \{1, 2, ..., n\}$ is the set of enumerated items to order and $y \in Y = \{1, 2, ..., n\}$ are the possible positional indices. For example, in the ordering of four words with the enumeration "He" $\to 1$, "Big" $\to 2$, "Brother" $\to 3$, "loved" $\to 4$, the target solution...

He	loved	Big	Brother
1	4	2	3

... is represented by the (permutation) mapping $f : X \to Y$

$$f = \{1 \to 1 \quad 4 \to 2 \quad 2 \to 3 \quad 3 \to 4\}$$

which says that item 4 is in position 2, item 2 is in position 3 and so on. Since the ppM represents probabilities, the constraint $\sum_Y P_{xy} = 1$ is enforced for each $x \in X$, though the converse $\sum_X P_{xy} = 1$ is not required. The value of P_{xy} is used a sampling weight when spawning children, such that P_{xy} weights the event "move item x to position y", swapping the item x and the item in position y. For instance, in the above ordering example, P_{23} represents the probability of:

Big	He	loved	Brother	$\to$	loved	He	Big	Brother
2	1	4	3		4	1	2	3

At generation 0, the ppM can be initialised either to a uniform probability matrix $P_{xy} = \frac{1}{n}\ \forall x, y$, or normalised data from some fast speculative fitness function $F_s(x, y)$ which gives an efficient estimate of the fitness of item x appearing in position y in the solution. As the algorithm progresses, the ppM is updated based on lineage information. The aim is for the ppM to converge to:

$$P_{xy} = \begin{cases} 1 \text{ if } x \text{ is in position } y \text{ in the solution} \\ 0 \text{ otherwise} \end{cases} \tag{2}$$

The variance $\mathrm{Var}_y[P_{xy}]$ is a good measure of how certain the ppM is of the position of x in the solution.

1.2.2 Update Functions

If a swap (x_1, x_2, y_1, y_2) - that is x_1 goes to position y_2, replacing x_2, which goes to y_2 - elicits a positive change in fitness ($\Delta F > 0$), the ppM should "reinforce" the likelihood that that particular arrangement is correct. If it is negative then it should reduce the likelihood. However, one sample corresponds to four ppM values. For example, if $\Delta F > 0$,

- $P_{x_1 y_2}$ should increase
- $P_{x_1 y_1}$ should decrease
- $P_{x_2 y_1}$ should increase
- $P_{x_2 y_2}$ should decrease

because the previous arrangement yielded a worse fitness. Furthermore, to preserve probability sums ($\Sigma_x P_{xy}$), the change in $P_{x_1 y_1}$ should equal the negative of that of $P_{x_1 y_1}$, so we define an *update function* U, and the following update mechanism.

- $P_{x_1 y_2}$ += $U(\Delta F, P_{x_1 y_1}, P_{x_1 y_2})$
- $P_{x_1 y_1}$ −= $U(\Delta F, P_{x_1 y_1}, P_{x_1 y_2})$
- $P_{x_2 y_1}$ += $U(\Delta F, P_{x_2 y_2}, P_{x_2 y_1})$
- $P_{x_2 y_2}$ −= $U(\Delta F, P_{x_2 y_2}, P_{x_2 y_1})$

The conditions for the update function $U(\Delta F, p_0, p_1)$ - where p_1 is a probability associated with the new arrangement and p_0 is a probability associated with the old arrangement - are as follows:

1. It must be strictly increasing for increasing ΔF
2. The new values of p_0, p_1 must stay bounded by $[0, 1]$ to represent probabilities - the range of U must be $(-p_0, p_1)$
3. $U(0, p_0, p_1) = 0$

One such a function is a transformation of tanh, which maps $U : \mathbb{R} \to (-p_0, p_1)$ with a free parameter M that controls the update rate:

$$U(x, p_0, p_1) = \frac{2 p_0 p_1}{(p_0 + p_1)\coth(M \cdot x) + (p_0 - p_1)} \tag{3}$$

A disadvantage of this function is that probabilities with the value 0 stay stuck with the value 0.

Algorithm 2 The ppM Algorithm

Initialise $\mathbf{P}$ as an $n \times n$ matrix
Normalise $\mathbf{P}$ such that $\sum_Y P_{xy} = 1 \; \forall x \in X$
function $\mathrm{U}(x, p_0, p_1) = \frac{2p_0p_1}{(p_0+p_1)\coth(M \cdot x)+(p_0-p_1)}$

function generateSwap($\mathbf{P}$) $= (x_1, x_2, y_1, y_2)$ chosen with probability weighting $P_{x_1y_2}$ (that is, x_1 goes from its position y_1 to new position y_2, and x_2, which was previously in position y_2, goes to position y_1)

Initialise *parent*
$F \leftarrow$ fitness(*parent*)

for g generations **do**

    swaps $\leftarrow$ generateSwap($\mathbf{P}$) s times, without replacement
    children $\leftarrow$ apply *swaps* to *parent*
    $F_{best} \leftarrow \max(\text{fitness}(children))$

    for $swap : (x_1, x_2, y_1, y_2)$ in *swaps* **do**
        $\Delta F = \text{fitness}(swap) - F$
        $P_{x_1y_2}$ += $U(\Delta F, P_{x_1y_1}, P_{x_1y_2})$
        $P_{x_1y_1}$ −= $U(\Delta F, P_{x_1y_1}, P_{x_1y_2})$
        $P_{x_2y_1}$ += $U(\Delta F, P_{x_2y_2}, P_{x_2y_1})$
        $P_{x_2y_2}$ −= $U(\Delta F, P_{x_2y_2}, P_{x_2y_1})$
    end for

    if $F_{best} > F$ **then**
        $F \leftarrow F_{best}$
        parent $\leftarrow$ child whose fitness is F_{best}
    end if

end for

return *parent*

1.3 Algorithm

In our implementation, in Julia, we created an object-oriented approach, in order to internalise all the mapping states. The `::PosProbMatrix` type contains the ppM, as well as instances of the current $X \to Y$ order mapping - as a vector where `ppm.xy[x] = y` - and the inverse order $Y \to X$ mapping, both of which are updated when the new parent is chosen to match the new parent's mapping. These internalised mapping states improve performance and make the algorithm safer to use with different ordering problems. Another basic optimisation is to forbid identity swaps - that is, swap (x_1, y_2) is not allowed if x_1 is already in position y_2. Since the `::Substitution` object stores the inverse mapping (`sub[y] = x` means x substitutes y, but x is in position y), the internalised mappings are intialised with `set_yx!(ppm)` not `set_xy!(ppm)`. The substitution solve also solves for the *inverse* substitution (the one that returns the ciphertext to its plaintext) and then inverts it, because this avoids inverting every child before application to measure its fitness.

1.3.1 ACO and Travelling Salesman

In reality, the Position Probability Matrix plays the role of 'pheromone weights' in ACO algorithms, though the probability is precalculated in our implementation, hence the restrictions on entry sums. In regular ACO, the pheromone weights are unbounded and a probability function is used to normalise/generate mutation probabilities. The lineage is equivalent to the 'ants' at a certain iteration, though specifically the set of 'ants' has a linear, child/parent tree structure and children are discarded after each iteration.
The problem of finding a substitution cipher key is equivalent to a Travelling Salesman problem (under the assumption that the global fitness is 'simple' - that is, just the sum of some partial fitnesses for individual tokens/positions in the key), where the nodes are labelled 1 to 26 (indices) and the following node is the permutation mapping of the previous node, then the ppM is directly related to the pheromone weight matrix.

1.3.2 Limiting Cases

- $M = 0$: This case amounts to the studied hill-climbing algorithm (if the ppM is initialised with uniform distribution), or hill-climbing with weighted probabilities if the ppM is non-homogeneous
- $M = \infty$: The update function becomes a transformed step function. This case can increase computational performance, but will fall flat to any mismatch between ΔF information and global fitness.

1.4 Research Objectives

- Is the proposed algorithm able to solve ordering problems with its learning properties only, excluding hill-climbing behaviour?
- Does the proposed algorithm solve the key exactly in fewer generations - on average - than the hill-climbing algorithm?
- Does the proposed algorithm's solve time (in generations) have a lower variance than that of the hill-climbing algorithm (i.e. is it more certain that the key is solved within some number of generations)?
- Is the proposed algorithm able to solve substitutions with a larger keyspace (specifically, English bigram substitution)?
- Where else does the algorithm apply, and where - and why - does it fail?
- Hyperparameter tuning of spawn number and update rate

1.5 Results

To reliably benchmark the progression of the algorithm over generations - without bias - the candidate solving algorithm is run with a fixed number of generations G_{set}, with data recorded for each run, and the runs are appropriately averaged to produce an average run: this removes the stochastic variation in the algorithm's performance on individual runs. To remove bias from the target substitution, a random substitution is generated for each run. To remove bias from the plaintext, runs receive plaintext from different English sources, all cut to a length of 2000 characters (counting alphabetic only). With all the above variations, the algorithm is run a set number of times and all runs are averaged.
Another specifier is the parent initialisation: the benchmark was run both with starting from an identity substitution and from a frequency matched substitution. The frequency matched substitution simply extracts the empirical text token frequencies and consults a database of expected frequencies, pairing tokens by frequency rank ("E" ranks first and substitutes the most empirically common token, "T" second...). Starting with a frequency matched substitution provides the algorithm with a 'headstart', beginning at a higher relative fitness.
Additionally, the benchmark for the ppM algorithm was run with an extra parameter. The initialisation of the Position Probability Matrix is either uniform, or the algorithm can be given an 'information headstart', using batch-binomial probabilities (`bbin` for short). This involves using a binomial model for the appearance of a token in the text and, for each token w_i, working out the probability that it appears as many times as some other token w_j does in the ciphertext, then normalising. That is,

$$P_{\text{bbin}}(w_i \text{ has substituted } w_j) = \frac{\binom{L}{N_i}(f_j)^{N_i}(1-f_j)^{L-N_i}}{\sum_k \binom{L}{N_i}(f_k)^{N_i}(1-f_k)^{L-N_i}} \tag{4}$$

where f_i is the expected frequency of w_i, N_i is the empirical number of appearances and L is the length of text. This initialisation function is called only once and provides the ppM with a dose of information about the likelihoods of certain mappings, serving as a fast approximation function for the converged ppM state.

1.5.1 ppM Convergence

To verify that the theoretical method works as intended, there are two criteria to satisfy:

1. Does the ppM converge to the expected matrix?
2. Does the ppM algorithm converge to the solution, without relying on the hill-climbing behaviour of the algorithm?

Condition 1 requires a fitness function for the Position Probability Matrix. A fitting choice is the Kullback-Leibler divergence between the target matrix (Equation 2) and the ppM, as the matrix represents a probability distribution approximating the true distribution to an increasing degree of accuracy. This does, however, fall short if any of the elements that are supposed to converge to 1 display a value of 0 (or some other vanishingly small float value), giving `NaN`s or `Inf`s. Throughout analysis, this has rarely ever happened, so the KL divergence remains a suitable measure of loss in the ppM. The column variance (as mentioned before) can be used to approximate the 'certainty' of the guide matrix, and thus the progress of the algorithm.

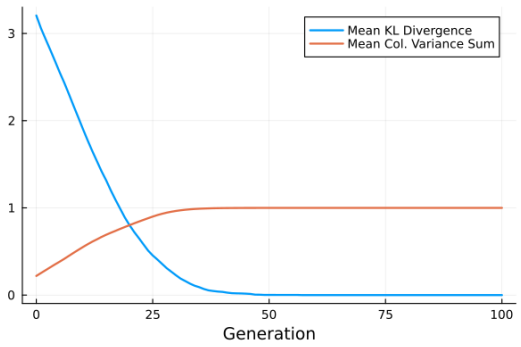


Figure 1: Run-averaged convergence statistics

Figure 1 confirms that the guide matrix is converging as the algorithm progresses, both the sum of the column variances tends to 1, and the KL-divergence of the ppM to its target tends to 0[1]. Another useful data set is the animated heatmap of the ppM over generations - plotting the $n \times n$ matrix with time as the 'generations' dimension - which also confirms the convergence of the ppM (see Fig 2)[2]. The animations unfortunately cannot be embedded into this paper, but can be reproduced simply with the codebase[3]. An extra behaviour, revealed on inspecting the heatmaps, is that the guide matrix will 'query' ambiguity in token positioning: if a token has a pair of highest valued candidate positions in the guide matrix, the subsequent draws will likely choose these positions and the resulting data allows the guide matrix to discriminate between the two, choosing the better mapping.

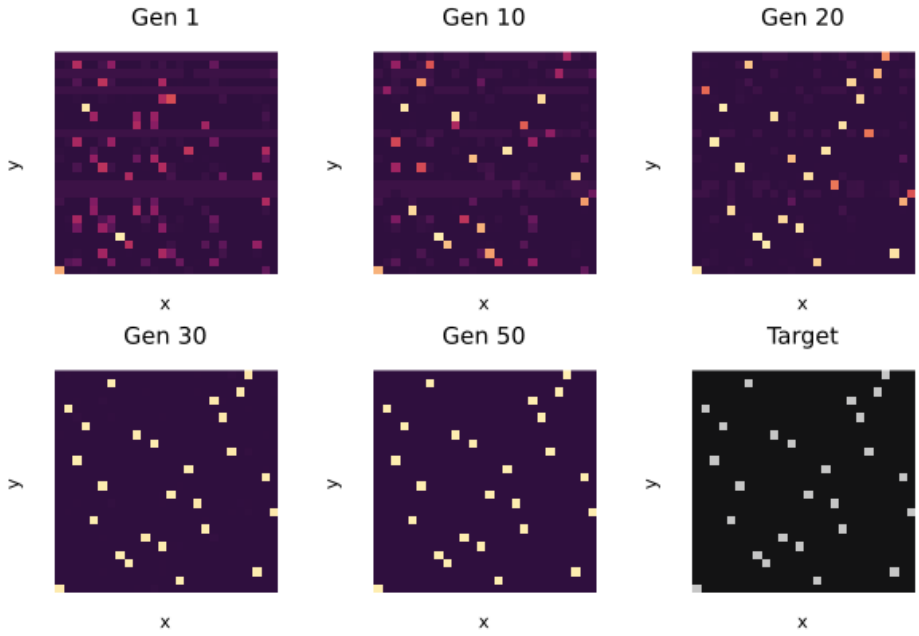


Figure 2: Heatmap of the ppM (brighter colours are closer to 1, darker colours closer to 0) at differing stages of the progression.

Condition 2 means the algorithm cannot simply rely on 'shotgun' hill-climbing. To ensure this, a parameter called the `lineage habit` was added, which controls how the next parent is selected from the children:

- `fascent` or floored ascent - select the best candidate from all children, including the parent
- `ascent` - select the best candidate from all children, excluding the parent
- `random` - select a random candidate from all children
- `stationary` - select the parent
- `descent` - select the worst candidate from all children

The convergence of the pure hill-climbing algorithm relies on a `fascent` lineage habit, otherwise as fitness increases the chance of selecting a child with a lesser fitness increases hugely. Figure 3a shows the lineage fitness benchmark for a pure hill-climbing algorithm[4]. Only `fascent` leads to convergence on the solution. `ascent` comes close but is statistically very unlikely to ever make the last few swaps required to reach the solution. `random` and `descent` both cause massive divergence, settling at equilibrium fitness values far below that of the initial state. The ppM algorithm shows a different performance[5] in Figure 3b, converging with `ascent` almost as well as `fascent` (the lines are so close together that they are superimposed). This is because the ppM algorithm, through the interaction of the guide matrix with the lineage, has a more targeted lineage progression which it can dynamically change. As it climbs the fitness curve and approaches what would have been the `ascent` equilibrium for the hill-climbing algorithm, it has converged sufficiently to be able to predict the necessary remaining swaps, and has set the probabilities of these swaps high enough so as to converge on the solution. Additionally, the self-annealing property means that when the lineage has approached the target, swaps that would elicit a lower fitness become less and less likely and the guide matrix 'confirms' its decision. Even with the `random` lineage habit, the ppM algorithm manages to converge by gaining information during its period of random progression. The `descent` lineage habit fails to meaningfully converge within the 500

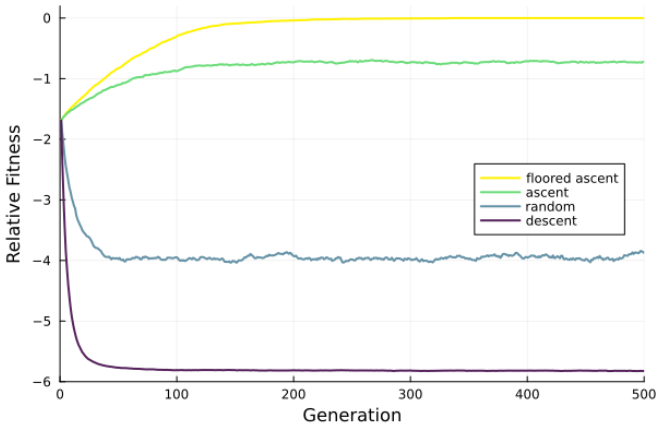
[1] Results reproducible in `ppm convergence.jl`. Parameters: 150 runs, 100 generations, 10 children spawned per gen, $M = 25.0$, `bbin`, frequency matched start.

[2] Results reproducible in `ppm heatmap frames.jl`. Parameters: 10 spawns, $M = 25.0$, `fascent` habit and `bbin = true`.

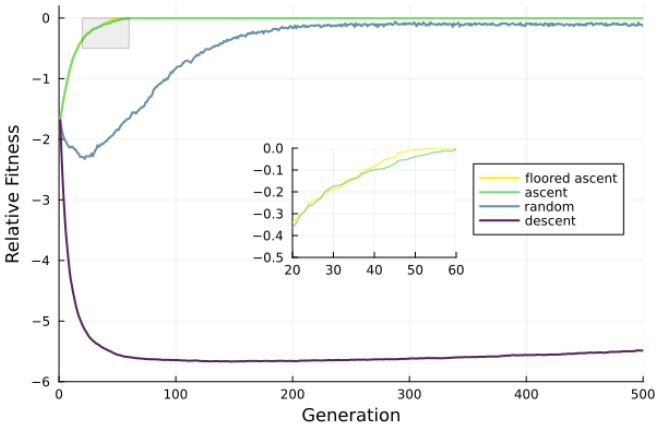
[3] Results reproducible in `ppm heatmap.jl`

[4] Parameters: 150 runs, 500 generations, 10 spawns, frequency matched start.

[5] Run with `bbin` and $M = 10$



(a) Run-averaged performance of the hill-climbing algorithm with different lineage habits

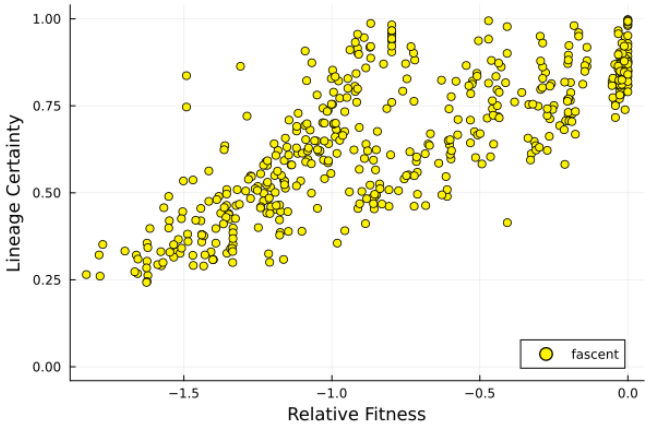


(b) Run-averaged performance of the ppM algorithm with different lineage habits

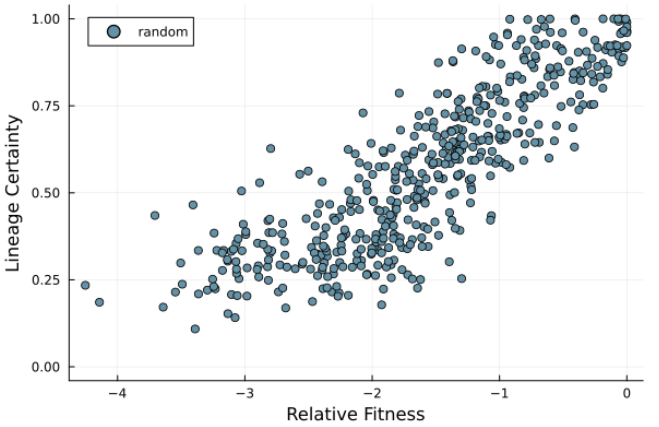
generations of this test. When the lineage habit is switched from `descent` to `fascent`, `ascent` or `random`, the lineage recovers and the ppM converges within 200 generations. This indicates that `descent`'s failure is not due to unreliability in the fitness information at very low fitnesses, the likes of which are achieved after 50 generations of `descent`. It may be the case that - since the swap sampling process is exhaustive and without replacement - a `descent` run with a high number of spawns is likely to always find a worse version, though this doesn't explain the divergence of the ppM. Runs with lower spawns (2 and 3) keep a relatively constant KL Divergence value across generations (runs with 1 spawn are effectively just `ascent` with 1 spawn).

1.5.2 Annealing

To verify that the algorithm does follow the predicted self-annealing, data was gathered about the lineage fitness over generations and the *lineage certainty*, which is the average value of P_{xy}, where x, y are a pair given by the parent solution's mapping. This statistic indicates how sure the guide matrix is of the parent solution's configuration being the correct configuration, and since higher P_{xy} values anneal the algorithm, the correlation between lineage certainty and lineage fitness is an indicator of whether the algorithm anneals correctly. Plotted are 500 random data points from runs, for different lineage habits, with `bbin = true`[6]. `fascent`, `ascent` and `random` all exhibit a strong positive correlation (as well as with `bbin = false`), which confirms that self-annealing works as expected there. `descent` has a very weak positive correlation, which may point to flaws in its annealing, possibly explaining its failure to converge. However, it is also a less reliable statistic as the majority of the data (non-outliers) are situated in a much smaller range than $(0, 1)$, so it is difficult to extrapolate and predict its failure to correlate at higher lineage certainty values.

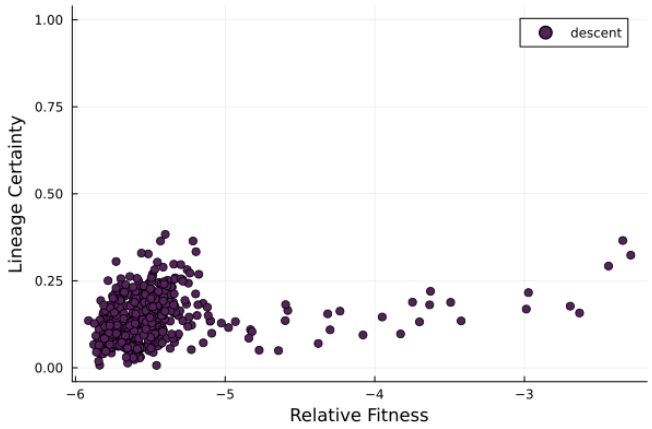


(a) 1000 runs, $G_{\text{set}} = 150$, Pearson Correlation Coefficient = 0.7735, Spearman's Rank Correlation = 0.7722



(b) 500 runs, $G_{\text{set}} = 300$, Pearson Correlation Coefficient = 0.8111, Spearman's Rank Correlation = 0.8228

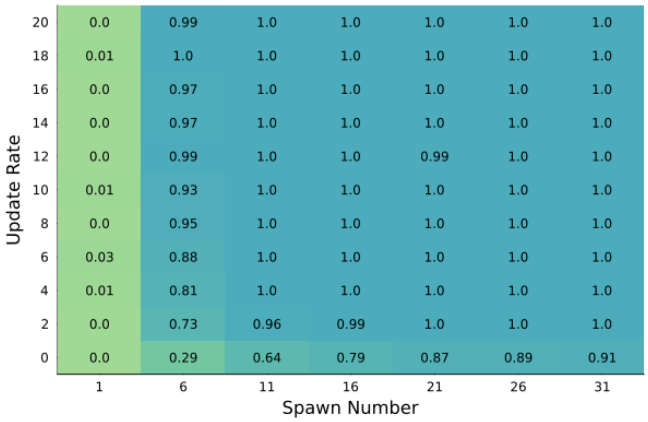
[6]Results are reproducible with `ppm anneal.jl`



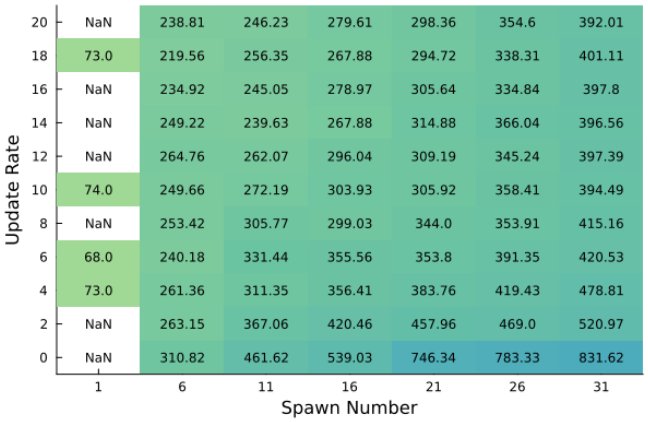
(a) 500 runs, $G_{\text{set}} = 300$, Pearson Correlation Coefficient = 0.1780, Spearman's Rank Correlation = 0.2378

1.5.3 Hyperparameter Tuning

To deduce the optimal parameters for the ppM algorithm for a 2000 token text, specifically with the aim of solving monoalphabetic substitutions, a benchmark was taken[7] with 75 generations for a variety of parameter combinations, testing spawns in the range $\{1, 11, 21, 31, ..., 91\}$ and M in the range $\{1, 11, 21, 31, ..., 41\}$. These ranges were chosen because we expect to see the most sensitivity in the performance of the algorithm in these ranges. Each run records three statistics: the finishing fitness, the finishing KL-divergence and the solved generation (set to 76 if unsolved). The search was then sent through a second iteration, choosing a new parameter set to search: the results are shown in Figures 6a and 6b.



(a) Fraction of runs which reached the solution in the 75 generations



(b) Average number of searched keys for successful runs

Figure 6

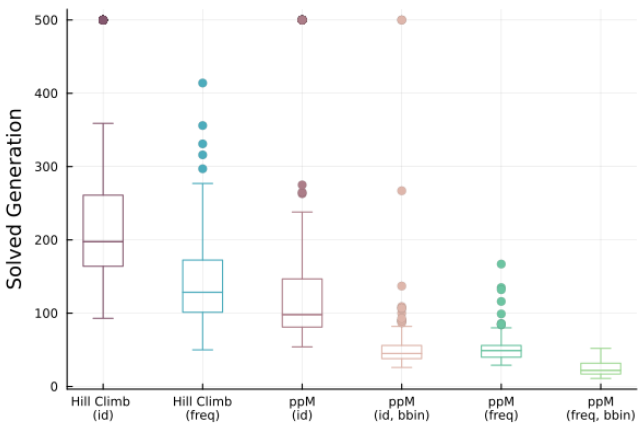
`NaN`s occur where the number of solves is zero. The average number of keys searched is the average number of generations to solve, multiplied by the spawn number. Predictably, increasing the spawn number will decrease the average solved generation, so introducing this statistic serves to give a standardised measure of algorithm performance with variable spawn number. Runs with 1 spawn struggle to converge within 75 generations, because they can only search a maximum of 75 keys, where an average of > 200 are necessary. Looking at spawn numbers above 10, as the update rate increases, the solved generation decreases. Critically, the increase from $M = 0$ to $M = 2$ offers by far the largest performance increase, going by the searched keys metric. This is evidence that it is not just `bbin` which is responsible for the success of the ppM algorithm, since the $M = 0$ case is just `bbin` hill climbing. To minimise the number of searched keys, lower spawn numbers with higher update rates are favourable. This suggests that the ppM tends to exact the correct swaps, leading the other swaps that are drawn (without replacement) to be a waste of search. Taking the reliability of the runs in mind (with success rate in Figure 6a), a good parameter set is ~ 10 spawns with $M > 20$. As M increases from $20 \to \infty$, the change in the number of searched keys is very small. There must be a critical point in that region, as runs with $M = \infty$ have an unsatisfactory performance within the 75 generations (56.6% success

[7] Results reproducible with `ppm hyperparam.jl`. Parameters: 75 runs, 75 generations, frequency matched, `bbin`

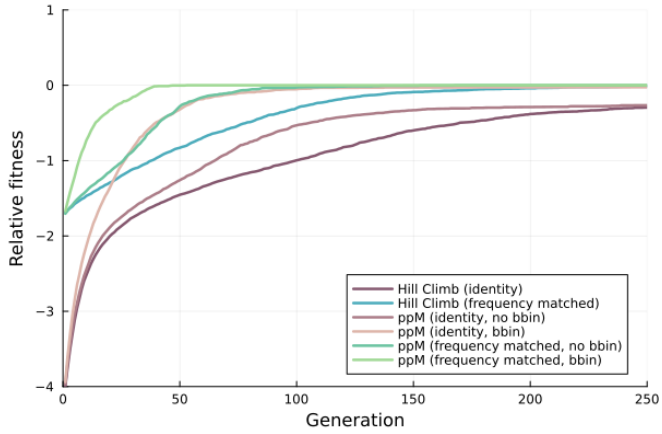
rate). The lowest searchspace of 220 keys with reliable (>97% solve rate) was achieved with 6 spawns and $M = 25.0$[8] when tested over 600 runs.

1.5.4 Lineage Fitness Benchmarking

Here, parent fitness is recorded for each generation in the lineage, and the runs are averaged elementwise to produce an average run. An extra statistic is gathered: the generation number by which the algorithm's lineage reaches the correct (target) substitution, for each run (aka. the solved generation). Each run therefore produces a vector of lineage fitness (of length G_{set}) and a solved generation (between 0 and G_{set}), and the set of these data over all runs gives a better conclusion[9].



(a) The distribution of solutions across generations



(b) The average parent fitness over the first 250 generations

Figure 7b shows the lineage fitness benchmark for different algorithm variants (150 runs, 500 generations, 10 spawns) and $M = 10$ for the ppM algorithm. Starts from an identity substitution are shown in brown colours and from a frequency matched substitution, in green colours.

1.5.5 The Effect of Text Length on Convergence

Using shorter ciphertexts (e.g. 1000 tokens) reduces the solve rate of the ppM algorithm with the aforementioned best parameters to roughly 80%, and this solve rate doesn't change when the algorithm is left to run for more (<600) generations. The average keys searched doesn't change much either. This indicates that the solving becomes more unstable, but the performance of a success is comparable to that with longer text. Inspecting heatmaps and other statistics reveals that this is because the algorithm either is missing one swap to the solution (two tokens, usually infrequent ones, are wrong) or converges more slowly, on account of the decreased uniqueness of the solution's fitness (shorter texts are more likely to have other configurations that are similarly 'English').

1.6 Considering other applications

The ppM algorithm struggles with solving a bigram substitution (which can be phrased as a substitution of length $26^2 = 676$). It quickly runs into local maxima - which may involve the simplicity of the quadgramlog fitness function - and the guide matrix does not converge meaningfully.

Another interesting case we have yet to consider is the application of the ppM algorithm for solving ciphers that have a substitution-equivalent key, such as the Playfair cipher and Polybius square cipher. Since the Polybius cipher can quickly be reduced to a substitution cipher problem in O(1) by assigning tokens to bigrams, the ppM algorithm would be able to solve it. The Playfair cipher is more complex in how its key is used and the encryption algorithm is very convoluted. The lineage/guide-matrix idea suggests that the ppM algorithm performs safely with non-diffusive cipher problems, with a fitness function that is mostly truthful over the entire domain of fitnesses (that is, it more often than not rates better substitution keys with a higher fitness, no matter how good the key is). Another potential area for development is parallel guide matrices to solve polyalphabetic substitution ciphers. Whether its success extends to tougher problems remains to be tested.

Some experimental expansions to the implementation which were not properly tested include:

- using an adaptive learning rate (M) to improve lineage convergence
- trying different update functions for more stable results with 'stuck' ppM element values, such as $P_{xy} = 0.0$
- allowing the lineage to open a parallel 'preemptive spawn' child in each generation of the lineage, which takes on multiple of the most highly probable swaps, to prevent bottlenecking when the guide matrix is sure enough of the solution but must still expend computational effort in exploring one swap at a time

[8] Results are reproducible with the `avg_keys_searched` function in `ppm keysearch.jl`

[9] Results reproducible with `ppm fitness & solve.jl`

- multiplying the guide matrix by a 'choice weight' vector that determines what tokens are more likely to be picked at some stage in the algorithm's progression, with the hope of creating more stable convergence by prioritising more important tokens at the start of the solving (tokens that appear most frequently, and thus affect the fitness most) and leaving less frequent tokens to be investigated later in the solving

2 Successor Probability Matrix (spM) Algorithm

2.1 Introduction

The second guide matrix algorithm was designed as a variant of the ppM algorithm for solving permutation ciphers.

2.1.1 Permutation Ciphers

Permutation ciphers have the same keyspace as substitution ciphers. They are transposition ciphers, maintaining the (unordered) set of characters in text but changing their order. For example the permutation cipher with key $(3, 1, 2, 5, 4)$ encrypts *"ITWAS ABRIG HTCOL DDAYI NAPRI LANDT HECLO CKSWE RESTR IKING THIRT EEN"* to become *"WITSA RABGI CHTLO ADDIY PNAIR NLATD CHEOL SCKEW SRERT IIKGN ITHTR NEE"*, permuting the text within blocks of 5 characters.
The cryptanalysis of permutation ciphers is certainly more difficult than that of substitution ciphers of the same size. This is because there are more local maxima in the keyspace, especially with quadgramlog, which is more sensitive to frequency information than ordering information. It is expected that the solving rate of optimisation algorithms at lower frequencies will be lesser than the solving rate for substitution problems, because small changes such as swaps no longer have a diffusive effect: swapping two items in a substitution affects all instances of the involved tokens, whereas swapping two items in a permutation only has local effects in the places where characters were interchanged.
A local fitness function approach would be favourable, as algorithms would benefit more from knowing which parts of the permutation key are correct, not how correct the key is overall (as in global fitness). Using a local fitness function for such problems is hugely beneficial, as changes in the permutation key apply locally in the corresponding tokens of the text, whereas in the substitution cipher case the changes applied globally across the whole text. As such, a local fitness function is a more efficient use of information. The spM algorithm will be based on a global fitness function, to sit in line with the ppM method, though a local fitness variant would perform better.
The morphism to a Travelling Salesman problem is even more obvious here: the route in the problem is just the bottom row of the two line representation, because the following node is just the successor mapping of the previous node.

2.2 Modifications

2.2.1 Relative Positional Encoding

The first major discrepancy between substitution keys and permutation keys is their treatment of position: substitution ciphers value absolute position, whereas permutation ciphers value relative position. For instance, the ambiguous key $(5, 4, 3, 2, 1)$, when shifted circularly to produce $(3, 2, 1, 5, 4)$ would elicit a drastically large ΔF for a substitution, because the absolute positions of all items have changed. On the contrary, for a permutation key, the ΔF for this shift would be much smaller, because items that belong together - on the whole - stay together, and almost all *relative positions* are conserved.
To modify the ppM algorithm, the guide matrix must encode the *relative* positions of items, instead of their absolute positions. A simple way to do this is to store the *successor* of each item. Denoting the spM with S, the element S_{xy} represents the estimated probability that item x is succeeded by item y in the solution, where $x \in X = \{1, 2, ..., n\}$ and $y \in Y = \{1, 2, ..., n\}$ is the set of enumerated items to order. In the previously used example, with the enumeration "He" $\to 1$, "Big" $\to 2$, "Brother" $\to 3$, "loved" $\to 4$, the target solution...

He	loved	Big	Brother
1	4	2	3

... is represented by the (succession) mapping $f : X \to Y$

$$f = \{1 \to 4 \quad 4 \to 2 \quad 2 \to 3 \quad 3 \to 1\}$$

which says that item 1 is followed by item 4, item 2 is followed by item 3 and so on. Again, since the spM represents probabilities, the constraint $\sum_Y S_{xy} = 1$ is enforced for each $x \in X$, though the converse $\sum_X S_{xy} = 1$ is not required. This encoding is lossy, however, since the end and start of the key are not specified: $(4, 2, 3, 1)$ also fits the same succession mapping f. A fix is proposed later.

2.2.2 Spawning Child Solutions

As before, the matrix elements S_{xy} are used as sampling weights to randomly draw a mutation. Previously, the selection of $S_{x_1 y_2}$ caused the swapping of items y_1, y_2. The spM algorithm takes a different approach. Take the mapping $(1, 4, 5, 6, 7, 8, 2, 3, 9)$. Using the mapping f to denote the current parent successor mapping, sampling now involves choosing an item and a new successor (x_1, y_2), from it generating the loose ends $(x_2, y_1) = (f^{-1}(y_2), f(x_1))$, but also generating an x_3, y_3 through successive Bernoulli sampling. The x_3, y_3 items mark the end of a *chain* of items to move. Instead of moving only one item to change its

successor (which wouldn't improve greatly on the regular absolute position swap), spM spawning chooses a whole chain of data to move, preserving structures whose fitness data suggests they appear in the target solution. The new mutation is a 'splice', removing the chain of items (beginning with y_2 and ending in x_3) and reinserting it at the target position x_1:

$$\begin{array}{cccccccccc} x_2 & y_2 & & & & x_3 & y_3 & x_1 & y_1 & \\ (1 & 4 & 5 & 6 & 7 & 8 & 2 & 3 & 9) & \end{array} \rightarrow \begin{array}{cccccccccc} x_2 & y_3 & x_1 & y_2 & & & & x_3 & y_1 & \\ (1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9) & \end{array}$$

The chain end and its successor x_3, y_3 begin as the chain start and its successor $y_2, f(y_2)$. Until a failure is drawn from the Bernoulli variable (with probability $p = S_{y_3\ f(y_3)}$), x_3, y_3 are advanced by one mapping $x_3 \to f(x_3) = y_3, y_3 \to f(y_3)$, extending the chain of items to splice by one. The motivation behind generating a chain length by sampling from the spM values for the current parent successors is that when the spM has a high certainty that the item following the splice start item y_2 succeeds that item, suggesting this is true in the solution, the succeeding item is more likely to be added to the chain, and thus kept in the arrangement that the spM sees fit, and so on for further successors. This results in a self-precipitation mechanism, where, at the start of the algorithm, the spM will choose short chains to splice due to its low certainty, but as it converges, it will begin to splice larger and larger chains, assembling the correct solution in parts that coalesce, just like a human assembles a puzzle. Towards the end of algorithm, the certainty will be high enough so the chain is almost always chosen to be the entire set of items, and the algorithm self-anneals, unable to make any further changes.
The splicing mutation does require an extra set of restrictions: y_3 must stop advancing, before it coincides with x_1, and x_1 should never be chosen to equal y_2 (set $S_{ii} = 0 \quad \forall i$ when sampling).

2.2.3 Update Protocol

Where previously a swap concerned only four values, now a splice concerns an extra two. The update functions are now:

- $P_{x_1 y_2}$ += $U(\Delta F, P_{x_1 y_1}, P_{x_1 y_2})$
- $P_{x_1 y_1}$ −= $U(\Delta F, P_{x_1 y_1}, P_{x_1 y_2})$
- $P_{x_2 y_1}$ += $U(\Delta F, P_{x_2 y_2}, P_{x_2 y_1})$
- $P_{x_2 y_2}$ −= $U(\Delta F, P_{x_2 y_2}, P_{x_2 y_1})$
- $P_{x_3 y_1}$ += $U(\Delta F, P_{x_3 y_3}, P_{x_3 y_1})$
- $P_{x_3 y_3}$ −= $U(\Delta F, P_{x_3 y_3}, P_{x_3 y_1})$

2.2.4 Start/End Encoding

To restore information about the start/end lost by the successor mapping, a size-n ordering problem becomes analogous to a size-$n+1$ problem, with an extra item called the split item. The successor of the split item is the first (starting) item, and the successor of the last item is the split item. The "He loved Big Brother" example from before has the new split-successor mapping:

$$f = \{\mathrm{S} \to 1 \quad 1 \to 4 \quad 4 \to 2 \quad 2 \to 3 \quad 3 \to \mathrm{S}\}$$

For ease of indexing, the split item can be included in the enumeration, as the last item $n+1$ (5 in the case of the example). Now, start/end information is restored, and the spM algorithm can freely adjusted the location of the split item during its progression just like any other item, resulting in the effect of circularly shifting the other items (unless it is included in a chain).

2.2.5 Speculative Initialisation

Since the batch-binomial function no longer applies to the problem, a new speculative fitness function is necessary. For the case of permutation ciphers, the 'follow score' is an applicable function. The tokens are arranged into a size $(n, \cdot)$ matrix, and split into columns of tokens with identical positioning in the permutation block:

I	T	W	A	S
A	B	R	I	G
H	T	C	O	L
⋮	⋮	⋮	⋮	⋮

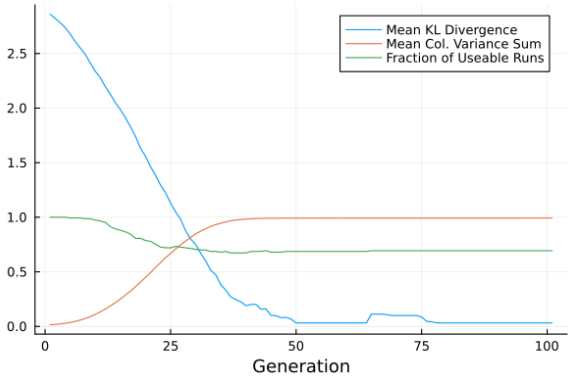
The Follow matrix F_{ij} is the average bigramlog score of column j following column i, excluding any bigrams containing null tokens, which come from incomplete permutation ciphers ($L \not\equiv 0 \pmod n$). We use the `exp10()` function on all the follow score values, both to make them positive, and because it is beneficial to inflate differences between higher scoring entries more than lower scoring ones. In initialising the spM, these values are normalised.

2.3 Results

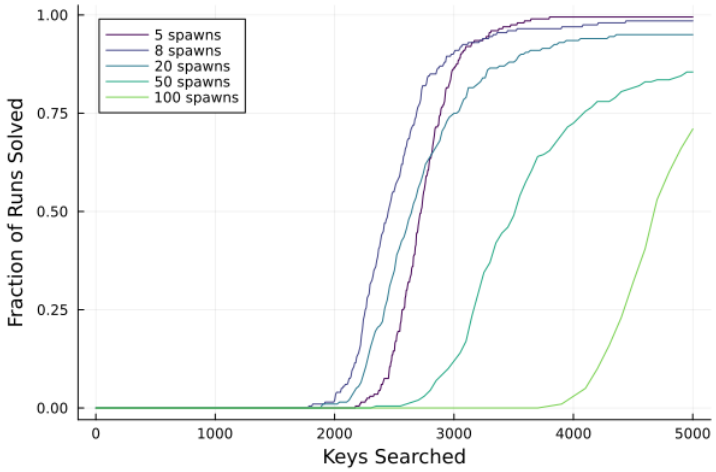
2.3.1 spM Convergence

In running the same test as for the ppM, it is discovered that the average KL Divergence is mostly comprised of undefined/irregular floats (`NaN` and `Inf`). Most of the individual runs contain infinities in the KL Divergences. On closer inspection, the lineage succeeds and produces the correct solution, but the guide matrix gives up, failing to investigate and correct any

final tokens which are misplaced after the lineage has succeeded. A zero in the spM where the solution should have a 1 causes undefined behaviour in the KL Divergence. Some runs are excluded in order to produce finite data and plot a graph[10] (the fraction of used runs is also plotted in Fig 8a)



(a) 150 runs averaged, solving a 20-permutation, plotting the convergence statistics



(b) Cumulative plot of 200 runs each, with different spawn numbers

Figure 8

The plot[11] in Fig 8b suggests that having too large a number of spawns slows down convergence. There is a critical point between 5-10 spawns (10 and 16 were both tested and converged more slowly than 8 spawns) where the optimal number of spawns for this particular configuration lies. The product of spawn number and number of generations contributes to the keys searched, and thus the computational cost. It is sensible to suppose that the algorithm favours the forward 'momentum' of the lineage in order to converge, since high spawn numbers means the number of lineage advances per key searched decreases.

2.3.2 Chain Spawning Evidence

To verify that the second self-annealing mechanism, of splicing larger chains when there is more certainty, is correct, a useful graphical data set is the permutation required to get from the current parent permutation to the desired solution (not between split token & successor encodings, but between regular permutation encodings). Visually, it is easy to see what steps must be taken to finish the solution, because when solved a diagonal appears in this permutation matrix. Fig 9 is interesting, displaying the previously described matrix[12]. It is evident that the algorithm first connects singular successors into small blocks that work well together (something the 'follow score' metric aims to do, but fails to achieve due to the complexity of the problem). As the solving progresses, the algorithm connects larger chunks to fit, moving them in one go with the chain spawning mechanism. By Gen 20, there are only a few large chunks to rearrange before the solution by Gen 50.

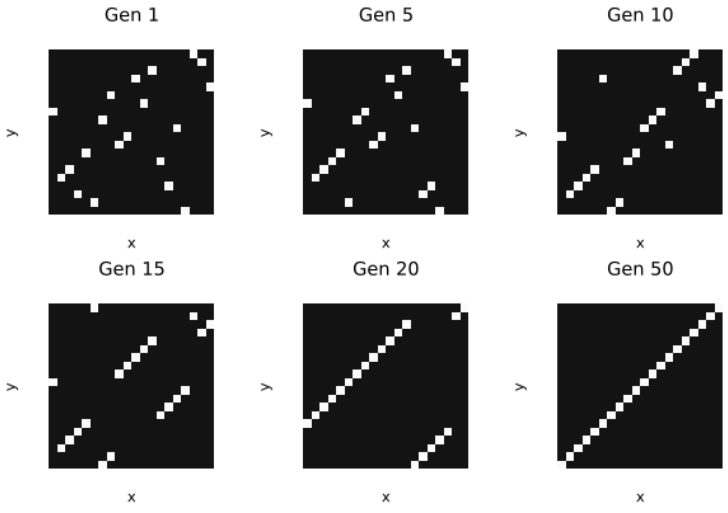


Figure 9: Permutation matrix to solution from current parent

2.3.3 Solving m10b (Cadenus) as a 130-Permutation

The National Cipher Challenge 2022 Mission 10b challenge is a ciphertext consisting of a modified Cadenus cipher. This Cadenus cipher is a transposition cipher operating

[10] Results reproducible in `spm convergence.jl`

[11] Results reproducible with `spm cdf.jl`. Parameters: Permutation length = 50, Text length = 2000, $M = 10.0$, `fscore = true`, `ascent`, no amnesia

[12] Results reproducible in `spm solving frames.jl`. Parameters: Permutation length = 20, 10 spawns, $M = 25.0$, `fascent`, follow score initialisation (`fscore`) `= true`

periodically on blocks of $26 \times N$ (where N is the length of the permutation key) The blocks are arranged into matrices, as with an N-permutation cipher, and the columns are rearranged according to the permutation key ($N!$ combinations). Each column is circularly shifted by a number from 1 to 26, forming another key (26^N combinations). In the case of M10b, $N = 5$ and there were $26^5 \times 5! \approx 10^9$ keys (ignoring the weakening of the key when the circshift key also dictates the permutation: e.g. (25,12,1,3,9) gives permutation (5,4,1,2,3)). Since the transpositions are confined to blocks of $26 \times 5 = 130$, this cipher is technically congruent to a 130-permutation cipher. A 130-permutation has $\approx 10^{220}$ keys.

Challenges begin to arise with an spM matrix of a size as large as 130×130. Since the number of draw elements increases with n^2, the number of spawns should also increase. For the benchmark with Mission 10b, there is one ciphertext of length 3250 tokens and 200 runs are averaged[13]. Inspection of the heatmaps and lineage certainty when running `fascent` reveals that the lineage reliably climbs to a high relative fitness (approx. > -0.5) but that the final solving stages are wildly unpredictable. Sometimes, the solution is found almost immediately after the climb, sometimes the lineage fitness plateaus and does not advance further. This is not a question of the fitness function's blindness, as if it was then the lineage would attain a relative fitness greater than zero. This final roadblock is also independent of generations: no matter how long it is left to run for, the probability of crossing the final hurdle does not meaningfully increase. The 'to-solve permutation' animations (as in Fig 9) display an almost solved final state in these failure cases, with long completed chains and a minority of splices left to do in order to construct the solution. An extra modification to the spawning or 'draw' mechanism was added to combat this stagnation: the spM is multiplied element-wise with a 'choice weights' vector to produce the sampling matrix ($M_{xy} = S_{xy}c_x$). The vector $c_x = 2 - S_{x\ f(x)}$ - where f is the forward parent mapping, aka `spm.xy[...]` - gave larger values for tokens whose current successor under the parent mapping corresponded to a low certainty value in the spM. This was in an attempt to channel the new spawns and computing power into the remaining tokens that disagreed with the guide matrix. The benchmark (with 'choice weights') resulted in a 15.5% success rate, with the runs that ended in solution averaging 332 generations to solve.

While it was impressive that some runs ended in solution, it was obvious that the algorithm could perform better. A crucial improvement ended up being 'amnesia', which draws parallels to pheromone 'dispersion' in ACO. When the lineage certainty ($\frac{1}{N}\sum_x S_{x\ f(x)}$) rises above a threshold value, the spM values are compressed towards each column's mean (which by $\sum_y S_{xy} = 1$ is always $\frac{1}{N}$) by a constant coefficient a:

$$x_i^* = a(x_i - \bar{x}) + \bar{x} \tag{5}$$

This has the effect of reducing the lineage certainty while maintaining the relative values of spM squares. 'Amnesia' is an antidote for over-annealing and is designed to help when the lineage is stuck. The benchmark was run with multiple configurations, detailed in Table 1 below. The amnesia threshold was set at 0.95 and the coefficient as $a = 0.8$.

Lineage Habit	Choice Weights	Amnesia	Solve Percentage	Avg. Solve Gen.
`fascent`	✓		15.5%	332
`fascent`	✓	✓	27.5%	450
`ascent`	✓	✓	97.5%	373
`ascent`	✓		63.0%	320
`ascent`			63.0%	337

Table 1: Benchmark of different configurations on the Mission 10b ciphertext (200 runs)

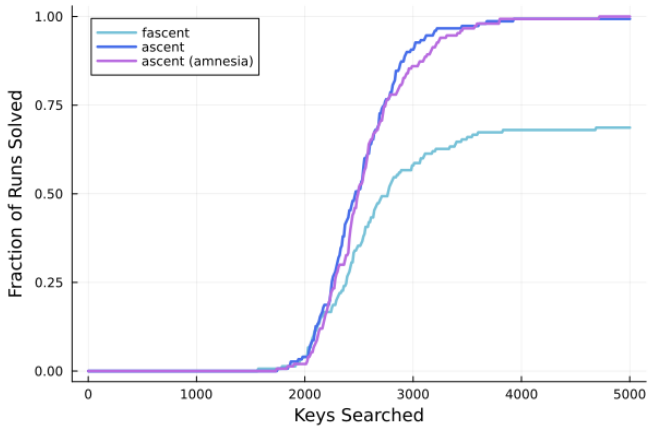
Firstly, in both lineage habit cases, the use of amnesia increases the solve percentage and the solve generation (up ~ 120 with `fascent` and ~ 50 with `ascent`, which are significant in the context of 200 trials). This suggests that amnesia is responsible for dislodging the stagnation in the late stages of the solving (when the lineage certainty is high), producing solutions past the plateau point, which would explain the increased solve percentage and why the average solve generation shifts upwards. The graph[14] Fig 10b condones this explanation: for large permutation problems, `ascent` plateaus but with amnesia runs continue to be solved past this point. For the smaller permutation of 50 (see Fig 10a) `ascent` is successful enough on its own and amnesia plays a negligible role. There is insignificant evidence that the use of 'choice weights' increases the performance for `ascent`, though it does increase the computational workload.

Moreover, the most noticeable difference is in the performances of `fascent` and `ascent`. While we expected `fascent` would perform better, because the lineage fitness is floored and progress can never be undone, it turns out that - by a significant margin - `ascent` is better suited to this problem, yielding a satisfactory 97.5% solve rate. Two candidate explanations are:

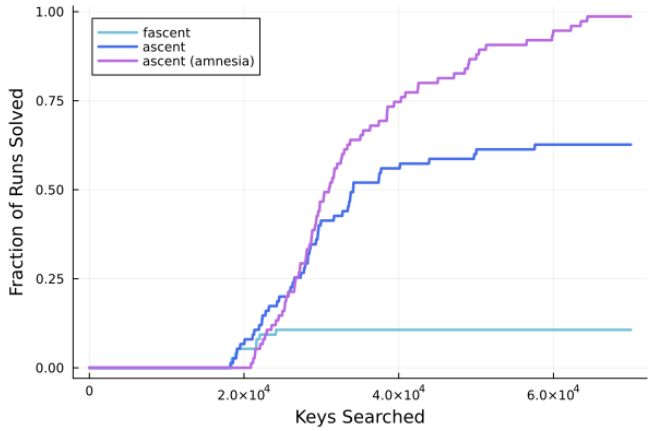
1. `fascent` anneals too quickly and gets frozen to an suboptimal lineage parent. `ascent` takes longer to settle on a solution and manages to avoid this over-annealing. This may also involve 'stranded' squares in the spM - with values of exactly 0.0 or 1.0 that arise due to floating point constraints and rounding - though it is not entirely responsible for this disparity, because amnesia would remove this factor, yet the performance difference is still apparent with amnesia.

[13] Results from this benchmark are reproducible with `spm m10b bench.jl`. Parameters: $G_{\text{set}} = 600$, 200 runs, 150 spawns, $M = 10.0$, `fscore = true`

[14] Results reproducible with `spm cdf.jl`. Parameters: Text length 2000, $M = 10.0$, `fscore = true`, amnesia coefficient $= 0.8$ and threshold $= 0.95$ (where applicable)



(a) $N = 50$, 10 spawns, 200 runs



(b) $N = 100$, 70 spawns, 75 runs

Figure 10: Cumulative plot of runs

2. The fact that a new parent is chosen every generation in `ascent` removes some unknown 'parental bias' mechanism by which a parent that stays reigning for multiple generations wrongly influences the convergence of the guide matrix. One simple concrete explanation would be that the spM squares for a highly rated (by fitness) parent receive a disproportionate amount of updates increasing their value, because at least two parent squares are guaranteed to be involved in any mutation, so they outweigh the merit of stronger squares/solutions.

It is interesting to note that `random` also manages to converge and solve the M10b ciphertext, over around 2000 generations and with a smaller amnesia effect (coefficient of 0.96 and threshold 0.98) to aid its annealing. The success of `ascent` indicates that the best solver algorithm for similar sizes of problem is to run `ascent` and keep an external log of the parent with the highest fitness, which would return the solution if `ascent` comes across it at any point during the run. This would remove the necessity of annealing, and decrease the number of generations necessary to guarantee a particular solve percentage. Without hyperparameter tuning, the average number of keys searched with the best configuration is around 56,000, which is a fraction of the keyspace.

2.3.4 Solution Limits

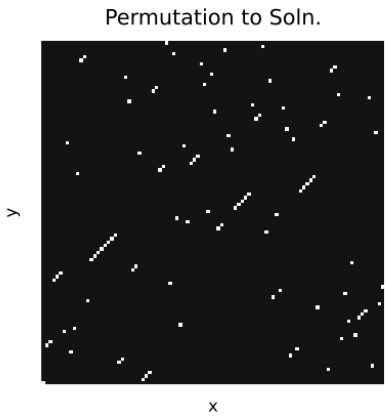


Figure 11

While increasing the permutation length N increases the number of possible keys greatly, a more affecting factor is the text length L. With the restriction of quadgramlog as a fitness function, there exists a 'specific unicity number' r_0 (based on Shannon's unicity distance), and values of $r(L, N)$ below r_0 make the correct key impossible to distinguish from other spurious keys. For a crude analysis, we assume $r(L, N) = L/N$, since the floor function of this quantity is the minimum length of a column in the matrix arrangement of a permutation cipher, thus being the number of elements that must be *simultaneously* correct for a good solution. It can be reasoned that the algorithm's difficulty in solving depends closely - if not linearly - on this number. An experimental lower bound can be found by executing runs and seeing the highest r value for which the highest ranked parent in any lineage has a relative fitness exceeding 0, since that means quadgramlog ranks an incorrect key as more fit than the true solution (fitness = 0). An upper bound is more difficult to find; of greater interest to this research is the lowest r value for which it is *possible* for spM to converge.

To gauge this number in a computationally unintensive way, one run of 6000 generations[15] was repeated a few times with a small amnesia coefficient for $N = 100$ with $L = 100, 200, 300, 400 \ldots$. While this may not be the most reliable data collection, it serves only to get a quick lower bound on this critical r value. r_0 is bounded below by 3.0, as runs with $L \leq 300$ yield spurious keys (relative fitness >0)[16]. The critical r value for the spM algorithm with the parameters used is bounded below by 6.0 ($L = 700$ resulted in a solution after <3900 generations). Fig 11

[15] Text was drawn from a $\sim$ 28000 token extract of George Orwell's *Nineteen Eighty-Four*. Parameters: 350 spawns, $M = 10.0$, `ascent`, `fscore = true`, $a = 0.7$, amnesia threshold = 0.85

[16] Results with $N = 50$ similarly bound $2.8 < r_0$

shows the permutation between the target and parent at the conclusion of the $L = 600$ run. Short chains of correct tokens are visible but the algorithm is not able to converge.
Looking in the other direction, the largest tested permutation that the algorithm solved was $N = 300$, with $L = 5000$ ($r \approx 16.67$), after searching just over 2 billion keys.

2.3.5 Keysearch Complexity across Permutation Length

3 Limitations / Future Points of Research

3.1 Parent Bias

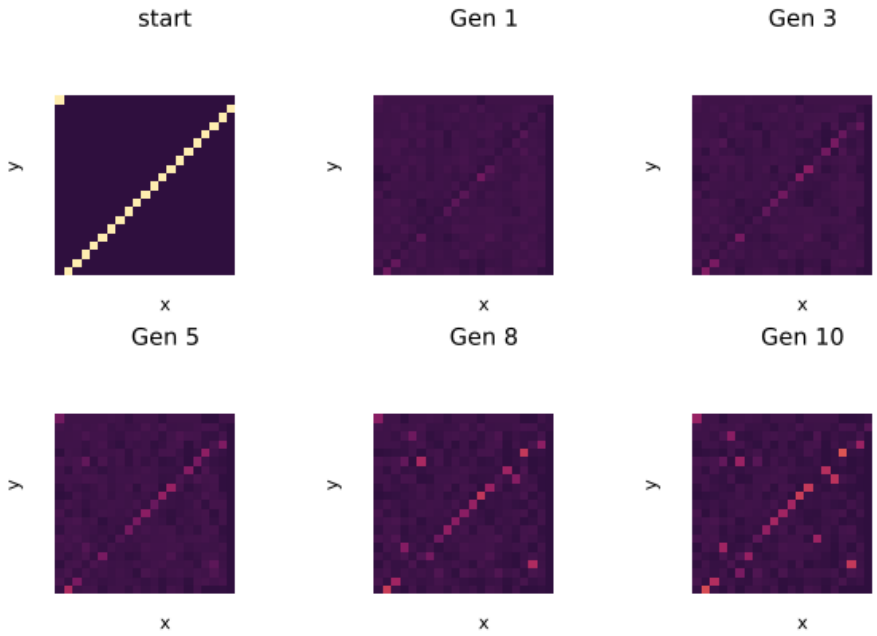


Figure 12: The first few generations of a lineage guide-matrix algorithm run, as well as the 'start', or first parent

A problem with the LGM algorithms is that a lineage parent's squares (labelled $(x_1, y_1), (x_2, y_2), \dots$) are expected to be updated s times more frequently than other squares, where s is the number of spawns. This is because, for every swap, the parent is used as a reference to check against and so the parent squares are updated. In certain situations, the parent squares can receive a larger increase - from being evaluated as better than a collection of candidate children - than the squares corresponding to a more fit arrangement. So, the parent is made more likely to stay as parent. The problem is particularly bad with `fascent`, as due to the floored lineage, the parent will sometimes stay as parent for many generations (this is never a problem with the other lineage habits). This parent bias can be directly observed in the heatmaps (Fig 12), where over the first few generations, the parent arrangement emerges with higher values but then dissolves as the algorithm converges. Noticeably, this is a problem with low variance guide matrices, as it is not as prominent with the very certain `bbin` initial matrix in the 26-substitution problem. Potential remedies include:

- Using a 'leaky' update function $\tilde{U}(\Delta F, p_0, p_1) = \{U(0.1\Delta F, p_0, p_1)$ if $\Delta F < 0,\ \tilde{U}(\Delta F, p_0, p_1)$ otherwise$\}$ to decrease the learning rate for a parent key outperforming its children and maintain the learning rate when a child outperforms the parent key.
- Connecting one guide-matrix to multiple parallel lineages, starting from different parents, in the hope that the parent biases cancel out.
- Removing probability sum constraints from the guide matrix (using a 'free' matrix) and calling the `softmax()` function columnwise to recover probabilities. This would allow for custom update protocols, where parent squares could simply receive smaller updates without breaking the sum constraints. This is equivalent to splitting the parent squares' update changes over every square in the column in a constrained matrix architecture.

3.2 Descent mode divergence

According to the ideas of lineage guide-matrix theory, there is no reason for `descent` to diverge like it has been shown to do experimentally. The implementation may have a discrepancy, or the specific problems/situations have some unpredicted behaviour that is disregarded in theoretical assumptions.

3.3 Local Fitness Approaches for spM

As discussed before, local fitness functions are more applicable for solving permutation ciphers. Although the use of quadgramlog in spM is pseudo-local, since the algorithm assumes (correctly) that all of the change in the global fitness value is attributed only to the changes in tokens/locations that are involved in a splice, the algorithm could potentially be improved by incorporating information about which specific tokens/locations have changed for the better and which for the worse. This distinction between whether the new successor y_2 does not fit after the old token x_1 or whether the old successor y_1 does not fit after the new token x_2 give more accurate information to the solving process.

4 Conclusion

The Lineage Guide-Matrix perspective on the ACO algorithm is a useful simplification tailored to solving cryptanalytic problems, and our findings on the behaviour of solving runs - presented as interpretable graphs and data points - led to interesting developments in improving LGM performance.
These implementations compromise on conceptual simplicity (making them a convincing alternative to other 'natural' algorithms such as the genetic algorithm) and experimental success. The versatility of different ACO implementations in attacking other ciphers has been demonstrated previously and our findings conform to this consensus.