Algoritmo Genético para Decifragem de Cifra de Substituição: Efeitos de Tamanho do Texto, Dicionário e Operadores de Crossover

Caio Vilor Brandão orientado por Mateus Coelho Silva
[1]*Centro de Matemática, Computação e Cognição, Universidade Federal do ABC, Brazil*
[2]*Laboratório de Inteligência Artificial, Robótica e Algoritmos, Universidade Federal do ABC, Brazil*
caio.brandao@aluno.ufabc.edu.br, coelho.mateus@ufabc.edu.br

Keywords: Algoritmo Genético, Criptoanálise, Cifra de Substituição, OX, PMX, Função de *fitness*.

Abstract: This work investigates the use of Genetic Algorithms (GAs) for breaking classical monoalphabetic substitution ciphers without a known key. We empirically discuss the impact of (i) the size of the ciphertext, (ii) the size of the dictionary (vocabulary) used in the *fitness* function, and (iii) the choice of *crossover* operator (OX vs. PMX). Based on successive exploratory decryptions, we observed that longer texts tend to make the *fitness* function more informative, whereas shorter texts increase the chance of stagnation in local optima and require larger dictionaries. In a set of 10 runs with OX and 10 runs with PMX, keeping the same text, dictionary, and GA parameters, the PMX operator got stuck in local optima 3 times, while OX got stuck only once. On the other hand, in the successful runs, PMX typically converged slightly faster. These results support the recommendation of using OX as the default *crossover* operator in GAs applied to monoalphabetic substitution ciphers, as it combines robustness with satisfactory search time, leaving PMX as an alternative for scenarios in which one can afford fine-tuning of parameters.

1 INTRODUÇÃO

Cifras de substituição monoalfabéticas são um alvo clássico para técnicas heurísticas. Sem pistas adicionais, a busca exata sobre o espaço de permutações (26! para o alfabeto latino) é inviável. Algoritmos Genéticos (AG) oferecem uma abordagem estocástica e paralelizável que, quando bem configurada, consegue aproximar a chave ótima em tempo prático.

Apesar da extensa literatura sobre AG, faltam orientações simples de "engenharia de uso diário" para ajustar o método quando a *informatividade* do texto varia (poucas vs. muitas palavras distintas) e quando o *fitness* depende de vocabulários de tamanhos diferentes. Este artigo contribui descrevendo uma implementação direta de AG para cifra de substituição e relatando observações empíricas sobre o efeito do tamanho do texto, do tamanho do dicionário e da escolha entre OX e PMX. Ao fim, com base em um conjunto de 10 execuções com OX e 10 com PMX, defendemos o uso de OX como operador padrão por se mostrar mais robusto a mínimos locais, ao mesmo tempo em que o PMX, quando bem ajustado, pode convergir ligeiramente mais rápido.

2 TRABALHOS RELACIONADOS

AGs já foram aplicados à criptoanálise de cifras clássicas combinando operadores de permutação e funções de avaliação baseadas em n-gramas, léxicos e estatísticas de língua. Operadores de *crossover* como OX (Order Crossover) e PMX (Partially Mapped Crossover) são padrões para problemas de permutação (ex.: TSP) e aparecem com frequência em criptoanálise heurística. Nosso foco não é propor novos operadores, e sim relatar, de forma prática, como configuramos um AG simples para cifra de substituição, discutindo empiricamente o efeito de OX e PMX e por que, ao final, optamos por OX como escolha padrão, apoiados em 20 execuções comparativas (10 por operador).

3 MÉTODO

3.1 Modelo de Cifra e Representação

Tratamos a chave como uma permutação do alfabeto latino (26 letras maiúsculas: A–Z). Cada indivíduo no AG é uma permutação; a decifragem aplica a correspondência inversa dessa permutação sobre o texto

cifrado. O texto cifrado é representado como uma sequência de caracteres ASCII (apenas letras e espaço), reconstruída a partir de uma sequência de bytes em formato textual.

3.2 Funcionamento do Algoritmo Genético

O AG segue o ciclo clássico de evolução, adaptado a um espaço de permutações:

1. **Inicialização**: gerar aleatoriamente uma população inicial $P^{(0)}$ de N permutações do alfabeto.
2. **Avaliação**: para cada indivíduo $x \in P^{(g)}$, decifrar o texto e calcular o *fitness* $f(x)$ com base na contagem de palavras válidas.
3. **Seleção de pais**: selecionar pares de indivíduos com probabilidade proporcional a $f(x)^{\alpha}$, onde α é um expoente que controla a pressão evolutiva.
4. **Crossover**: para cada par, aplicar OX ou PMX, produzindo dois filhos (permutações válidas).
5. **Mutação**: em cada filho, com probabilidade fixa p_m, trocar duas posições aleatórias da permutação.
6. **Formação da nova população**: a nova população $P^{(g+1)}$ é formada apenas pelos filhos (sem elitismo explícito); o melhor indivíduo global é armazenado separadamente.
7. **Critério de parada**: repetir de 2 a 6 até atingir o número máximo de gerações G.

Um fluxograma resumindo esse ciclo é mostrado na Fig. 1.

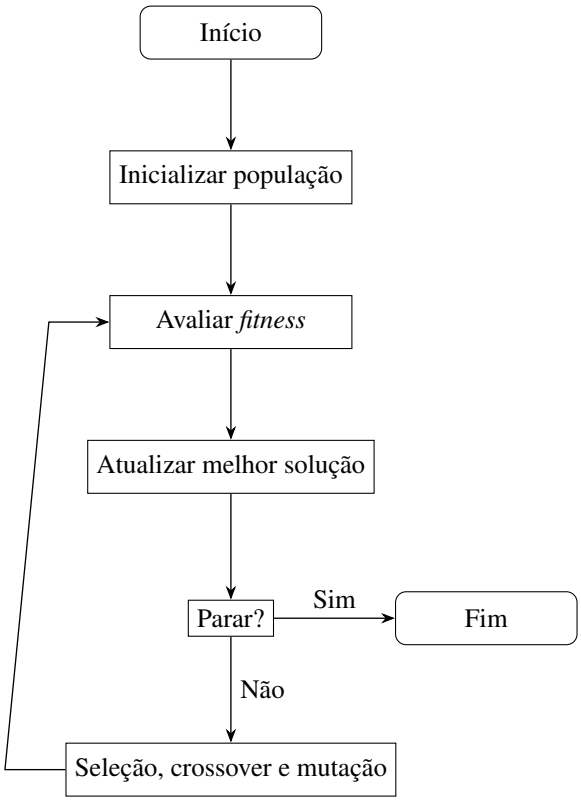

Figura 1: Fluxograma simplificado do algoritmo genético.

3.3 Função de *Fitness*

O *fitness* conta quantas palavras *distintas* do texto decifrado pertencem a um dicionário (vocabulário) pré-definido. Seja $W(x)$ o conjunto de palavras distintas obtidas ao decifrar com a chave x, e D o dicionário usado. Definimos:

$$f(x) = |W(x) \cap D|. \tag{1}$$

Usamos dois dicionários: *menor* (subconjunto das palavras mais frequentes) e *maior* (conjunto mais amplo). Quanto maior o dicionário, maior a sensibilidade do *fitness* para reconhecer decifrações parcialmente corretas, porém também maior a chance de falsos positivos (palavras curtas e ambiguidades).

3.4 Operadores Genéticos

Seleção: utilizamos seleção por roleta (*roulette wheel*). Dada a população $P^{(g)} = \{x_1, x_2, \ldots, x_N\}$ com *fitness* $f(x_i)$, atribuímos a cada indivíduo um peso

$$w_i = \max(0, f(x_i))^{\alpha}, \tag{2}$$

onde α é o expoente de pressão evolutiva. Em seguida, normalizamos w_i para obter uma distribuição de probabilidade e sorteamos pais de forma proporcional a esses pesos. Quando todos os $f(x_i)$ são nulos, a seleção degenere para uniforme (todos têm a mesma chance), evitando viés artificial no início da busca.

Crossover OX (Order Crossover): o OX é um operador clássico para permutação, projetado para preservar a ordem relativa de elementos. No nosso caso, cada indivíduo $x = (x_1, x_2, \ldots, x_{26})$ é uma permutação das 26 letras. O procedimento para gerar um filho c a partir de dois pais $p^{(1)}$ e $p^{(2)}$ é:

1. Sortear dois pontos de corte a e b tais que $1 \leq a < b \leq 26$.
2. Copiar o segmento central do primeiro pai para o filho:
   $$c_i = p_i^{(1)} \quad \text{para } i \in [a, b).$$
   As demais posições de c são inicialmente marcadas como vazias.
3. Percorrer o segundo pai $p^{(2)}$ a partir da posição b até o final, e depois circularmente do início até $b-1$. Para cada gene g visitado:

- se g ainda não aparece em c, inserir g na próxima posição vazia de c (começando em b e avançando circularmente);
- se g já está presente (dentro do segmento copiado), ignorá-lo.

4. Ao final, todas as 26 posições de c estarão preenchidas, sem duplicação de símbolos e preservando a ordem relativa induzida por $p^{(2)}$ fora do segmento de $p^{(1)}$.

Esse mecanismo garante que o filho seja uma permutação válida, ao mesmo tempo em que preserva um bloco contínuo de mapeamentos de $p^{(1)}$ e a ordem relativa dos demais mapeamentos de $p^{(2)}$. Em termos de busca, observamos empiricamente que isso introduz bastante diversidade estrutural: a combinação de segmento fixo + preenchimento circular tende a gerar arranjos de letras bem diferentes dos pais, o que se mostrou benéfico para escapar de mínimos locais, mesmo quando o *fitness* é relativamente informativo.

Crossover PMX (Partially Mapped Crossover): o PMX também trabalha com dois pontos de corte, mas enfatiza a preservação de "pares de mapeamento" entre pais e filhos. O procedimento é:

1. Sortear dois pontos de corte a e b tais que $1 \leq a < b \leq 26$.
2. Copiar o segmento central do primeiro pai para o filho c:
$$c_i = p_i^{(1)} \quad \text{para } i \in [a,b).$$
3. Construir um mapa de correspondência entre os genes do segmento:
$$M\left(p_i^{(1)}\right) = p_i^{(2)}, \quad M^{-1}\left(p_i^{(2)}\right) = p_i^{(1)} \quad \text{para } i \in [a,b).$$
4. Para cada posição j fora do segmento ($j < a$ ou $j \geq b$):
   (a) definir $g = p_j^{(2)}$ como gene candidato;
   (b) enquanto g já estiver presente no segmento copiado de c (ou seja, $g = p_i^{(1)}$ para algum $i \in [a,b)$), atualizar $g \leftarrow M^{-1}(g)$, navegando pelo mapa de correspondência;
   (c) após resolver o conflito, atribuir $c_j = g$.

Essa lógica de "mapa parcial" garante que cada símbolo apareça exatamente uma vez no filho e que relações de mapeamento dos pais sejam preservadas quando possível. Na prática, observamos que o PMX tende a fazer uma *exploração local* mais conservadora: blocos de mapeamentos corretos são mantidos com mais fidelidade, o que pode ser útil quando a população já está bem posicionada em torno de uma solução boa. Por outro lado, em nossas execuções comparativas, essa mesma característica frequentemente levou a travamentos em mínimos locais dos quais o PMX tinha dificuldade de sair, em comparação com o OX.

Mutação por troca (*swap*): a mutação é responsável por introduzir variações locais na permutação, evitando que a população convirja precocemente para uma região estreita do espaço de busca. Utilizamos um operador de *swap* simples:

1. Para cada indivíduo gerado por crossover, sortear um número aleatório $u \sim \mathcal{U}(0,1)$.
2. Se $u < p_m$ (com $p_m = 0{,}35$), escolher duas posições distintas i e j uniformemente em $\{1,\ldots,26\}$.
3. Trocar os símbolos dessas posições:
$$(x_i, x_j) \leftarrow (x_j, x_i).$$

Esse operador é minimalmente invasivo (apenas duas posições são alteradas), preserva a natureza de permutação e, ao mesmo tempo é suficientemente forte para "sacudir" a chave, permitindo que o algoritmo escape de ótimos locais. A escolha de aplicar a mutação em nível de indivíduo (e não de gene) com probabilidade relativamente alta (35%) foi motivada empiricamente: valores menores faziam o AG estagnar com maior frequência em mapeamentos subótimos, especialmente em cenários de baixo poder discriminativo do *fitness*.

4 DISCUSSÃO

4.1 Poder Discriminativo do *Fitness*

O ponto central para o bom desempenho do AG é o poder discriminativo do *fitness* — a capacidade de distinguir chaves melhores de piores de forma consistente. Textos maiores e dicionários amplos tendem a aumentar essa capacidade; textos curtos e dicionários pequenos a reduzem. Em nossas decifrações exploratórias, esse padrão apareceu de forma robusta: quando o texto cifrado continha muitas palavras distintas e o dicionário cobria uma fração razoável do vocabulário, o *fitness* fornecia um gradiente claro para a busca; em textos curtos e/ou com dicionário muito restrito, o AG passava a ver muitas chaves com valores de *fitness* iguais ou muito próximos, o que favorece a estagnação.

Nesses cenários de baixo poder discriminativo, a busca precisa de mais *exploração* (diversidade). O OX ajuda nesse aspecto: sua forma de preservação de ordem com preenchimento por rotação manteve mais padrões variados na população, o que na prática se traduziu em maior capacidade de escapar de chaves claramente subótimas. Já o PMX, ao tentar preservar

mapeamentos de forma mais rígida, funcionou bem apenas quando a população já estava relativamente próxima de uma solução boa; caso contrário, ficava "preso" em estruturas ruins.

4.2 Pressão Evolutiva, Tamanho de Texto e Dicionário

De maneira empírica, observamos os seguintes comportamentos:

- **Textos grandes** facilitam a busca: mais contexto significa mais chances de o *fitness* recompensar mapeamentos corretos e penalizar, ainda que indiretamente, mapeamentos ruins. Nesses casos, tanto dicionários menores quanto maiores funcionam.
- **Textos pequenos** tornam a busca mais frágil: com poucas palavras distintas, muitos mapeamentos diferentes acabam produzindo o mesmo valor de *fitness*. Aumentar o tamanho do dicionário (isto é, permitir mais palavras candidatas) ajuda a recuperar parte do poder discriminativo, mas também introduz mais ruído (falsos positivos).
- **Pressão evolutiva** muito alta (expoente grande na roleta) tende a ser perigosa quando o *fitness* é ruidoso: a população pode se concentrar muito rápido em chaves medianas e não sair mais dali. Pressões moderadas funcionaram melhor, especialmente em combinação com OX.

Em todos esses cenários, o OX se mostrou mais tolerante a configurações não ideais: mesmo quando a escolha de dicionário ou pressão evolutiva não era a melhor possível, o AG ainda conseguia, após muitas gerações, encontrar decifrações qualitativamente razoáveis. O PMX, ao contrário, era mais sensível a tais escolhas.

4.3 Comparação Empírica entre OX e PMX

Para tornar essas observações mais concretas, realizamos um conjunto simples de 20 execuções: 10 execuções com OX e 10 execuções com PMX, mantendo fixos o texto cifrado, o dicionário, o tamanho da população, o número de gerações, a pressão evolutiva e a taxa de mutação. Em cada execução, registramos se o AG convergiu para uma decifração qualitativamente boa ou se ficou claramente preso em um mínimo local.

Dos 10 testes com OX, apenas 1 execução ficou presa em um ótimo local ruim, enquanto as demais encontraram decifrações úteis. Já com PMX, 3 das 10 execuções ficaram presas em ótimos locais, com pouca ou nenhuma melhoria relevante após um certo número de gerações. Nas execuções em que ambos os operadores tiveram sucesso, o PMX normalmente convergiu em um número ligeiramente menor de gerações (do ponto de vista qualitativo), coerente com sua natureza mais conservadora e focada em refinamento local.

Esses resultados sugerem o seguinte compromisso prático:

- OX apresenta **menor taxa de travamento** em mínimos locais (1/10 nas nossas execuções) e, portanto, é mais adequado como escolha padrão, pois facilita o ajuste de parâmetros (é menos sensível a pequenos erros de configuração) e já oferece um tempo de busca satisfatório.
- PMX pode **convergir mais rápido** quando encaixado em uma boa configuração (3/7 casos bem-sucedidos, nas nossas execuções), mas é mais sujeito a travamentos (3/10 execuções). Assim, ele é uma opção interessante para cenários em que se pode investir em ajuste fino de parâmetros, possivelmente em conjunto com alguma estratégia de *restart* ou mutação mais forte.

4.4 Regra Prática

Com base nessas observações empíricas e no pequeno conjunto de 20 execuções comparativas, chegamos à seguinte recomendação pragmática:

- **Texto pequeno**: usar dicionário maior para ganhar informatividade; OX como *crossover* padrão; pressão evolutiva moderada ou baixa (ex.: 7); e número de gerações maior (ex.: 22.000).
- **Texto grande**: OX também como *crossover* padrão; dicionário menor, pois já é suficiente; pressão evolutiva em nível moderado (ex.: 9); número de gerações menor (ex.: 5.000), pois é suficiente e reduz o tempo de execução.

O PMX pode ser visto como um operador mais conservador, interessante em situações em que se deseja explorar finamente uma região já boa do espaço de chaves. Entretanto, em nossas 10 execuções com PMX ele se mostrou mais sujeito a travamentos em ótimos locais do que o OX (3 contra 1 travamento), de modo que não o recomendamos como escolha padrão para decifragem de cifra de substituição, especialmente quando o tamanho do texto ou a qualidade do dicionário são limitados. Ainda assim, ele pode ser útil se somado a algum mecanismo de mutação mais forte do que o escolhido em nossos testes, ou se empregado após uma busca inicial feita com OX.

5 CONCLUSÃO

Apresentamos uma implementação simples de AG para decifragem de cifras de substituição monoalfabética e discutimos, a partir de observações empíricas, como o comportamento do algoritmo depende do tamanho do texto, do dicionário e do operador de *crossover*.

Os principais pontos são:

1. Textos maiores simplificam a busca, pois aumentam o poder discriminativo do *fitness*; textos muito pequenos tornam a tarefa sensivelmente mais difícil.
2. Dicionários maiores podem recuperar parte da informatividade perdida em textos pequenos, mas à custa de mais ruído (maior chance de falsos positivos).
3. Em um conjunto de 10 execuções com OX e 10 com PMX, o OX ficou preso em mínimos locais apenas 1 vez, enquanto o PMX ficou preso 3 vezes, indicando que OX é mais robusto à estagnação, tanto em textos maiores quanto menores, e independentemente do tamanho do dicionário usado.
4. Nas execuções em que ambos os operadores tiveram sucesso, o PMX tendeu a convergir um pouco mais rápido, reforçando seu papel como operador de refinamento local potencialmente eficiente, porém mais sensível à configuração de parâmetros e mais sujeito a travamentos.

Diante disso, nossa conclusão prática é clara: **para AGs aplicados à cifra de substituição monoalfabética, recomendamos o uso de OX como operador padrão de *crossover***, por combinar bom desempenho, robustez a mínimos locais e tempo de busca satisfatório. O PMX permanece como uma alternativa promissora para cenários em que se pode investir em ajuste cuidadoso de parâmetros e, possivelmente, combiná-lo com OX (por exemplo, usando OX na fase inicial de exploração e PMX em uma fase posterior de refinamento).

Trabalhos futuros incluem incorporar n-gramas (bigramas, trigramas) à função de *fitness*, testar esquemas dinâmicos de ajuste de mutação e pressão evolutiva e estudar estratégias de *restart* e combinação adaptativa de OX com PMX.

REPRODUTIBILIDADE

Todo o material necessário para reproduzir as decifrações e observar empiricamente o comportamento do algoritmo está disponível em um repositório público no GitHub:[1]

- dois arquivos de entrada com os textos cifrados em bits, um *texto pequeno* (`cifrado_menor.txt`) e um *texto grande* (`cifrado_maior.txt`);
- dois programas em Python para decifragem via AG: `decifrar_menor.py` (configurado com parâmetros adequados para texto pequeno) e `decifrar_maior.py` (configurado com parâmetros adequados para texto grande), que podem ser usados tanto para inspeção qualitativa das chaves quanto para repetir múltiplas execuções com OX e PMX.

A partir desse repositório, o pesquisador pode executar diretamente `decifrar_menor.py` ou `decifrar_maior.py`, inspecionar qualitativamente as chaves encontradas, ajustar parâmetros como número de gerações, pressão evolutiva e taxa de mutação, e comparar empiricamente o uso de OX e PMX em diferentes textos e dicionários, inclusive reproduzindo o cenário de 10 execuções por operador descrito neste artigo.

REFERENCES

D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

N. M. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*. Springer, 2008.

A. Clark and M. Dawson, "Optimisation heuristics for the automated cryptanalysis of classical ciphers," *Journal of the Operational Research Society*, 2000.

[1] Disponível em: https://github.com/Caio-VB/Algoritmo-Genetico-em-Cifra-de-Substituicao.