Descrição Geral do Projeto

O objetivo deste projeto é implementar um algoritmo genético para resolver o problema do **Multiple Sequence Alignment (MSA)**, utilizando como base a matriz de substituição **BLOSUM62** e operadores genéticos adaptados a este contexto.

O programa inicia-se pela leitura e preparação dos dados, seguindo-se a geração da população inicial, o cálculo do *fitness* (score) de cada indivíduo e, posteriormente, os processos de seleção, cruzamento (*crossover*) e mutação que permitem a evolução das gerações.

Leitura e Processamento dos Dados

A leitura do dataset é efetuada pela função *read_fasta*, responsável por processar o ficheiro de entrada linha a linha.
Cada vez que é encontrada uma linha iniciada pelos caracteres “<” ou “&”, considera-se que esta marca o início de uma nova sequência. Estes símbolos indicam o início das linhas que contém informação sobre o nome e o tipo de proteína, bem como a espécie de origem. Assim, ao detectar uma destas linhas, o programa inicializa uma nova lista vazia que representará a sequência subsequente.

As linhas seguintes, que contêm apenas os caracteres da proteína, são adicionadas à sequência em construção até ser novamente encontrado um identificador de nova espécie, altura em que a sequência anterior é considerada concluída e então é armazenada.

O resultado final é uma **lista de listas**, em que cada elemento corresponde a uma sequência completa de aminoácidos.
Durante o desenvolvimento, foi inicialmente considerada a utilização da estrutura *MySeq*, fornecida nas aulas, mas optou-se por listas nativas de Python por questões de simplicidade e de flexibilidade funcional.

Após a leitura das sequências, procede-se à importação da matriz de substituição **BLOSUM62** através da biblioteca *substmatrix*. Esta matriz é fundamental para o cálculo do *score* dos alinhamentos, uma vez que é utilizado no módulo *PairwiseAlignment*, utilizado para avaliar a similaridade entre pares de sequências.

Inicialização da População

Com os dados preparados, é chamada a função principal *run_genetic_algorithm*, responsável por gerir a evolução da população ao longo das gerações.

A inicialização da população é realizada através da criação de múltiplos indivíduos.
Cada indivíduo corresponde a um alinhamento inicial composto pelas várias sequências do dataset, às quais são adicionados gaps aleatórios (carácter “-”) no início. Este processo tem como objetivo simular deslocamentos iniciais entre as sequências.

Após a inserção dos gaps iniciais, adicionam-se gaps adicionais no final de cada sequência, de modo a garantir que todas têm o mesmo comprimento. Por fim, são removidas as colunas compostas exclusivamente por gaps, uma vez que não representam posições válidas num alinhamento múltiplo.

O resultado deste processo é um indivíduo representado por uma lista de sequências alinhadas. Este procedimento é repetido *population_size* vezes para gerar a população inicial completa.

Cálculo do Score (Função de Avaliação)

Terminada a criação da população, cada indivíduo é avaliado através de uma função de *score*, que mede a qualidade do seu alinhamento.
Para tal, é construída uma estrutura de dados composta por tuplos, onde cada tuplo contém um indivíduo e o respetivo *score*.

O cálculo do *score* segue uma abordagem baseada na soma de pares (*sum of pairs*).
Para cada indivíduo, são consideradas todas as combinações possíveis de pares de sequências.
Cada par é alinhado (aqui não precisamos de fazer nada pois todas as sequências têm o mesmo tamanho) e, para cada posição, é consultado o valor correspondente na matriz BLOSUM62, que quantifica a similaridade entre os aminoácidos nessa posição.
Os valores obtidos para todas as posições de um par são somados, e este processo é repetido para todos os pares possíveis.
O somatório global de todas as sequências define o *score total* do indivíduo, quanto maior o *score*, melhor o alinhamento. Por fim, a população é ordenada com base no score para facilitar no passo seguinte.

O artigo de referência menciona que o cálculo do *score* deve envolver duas componentes principais:

- uma **componente de custo** (*COST(Ai, Aj)*) que avalia a correspondência entre pares de sequências;
- uma **componente de peso** (*Wij*) que atribui importância relativa a cada par.

Na nossa implementação, a matriz BLOSUM62 foi utilizada como a componente *COST*, representando a afinidade entre aminoácidos, enquanto o custo de gap foi incorporado como penalização associada a inserções ou deleções.
No entanto, devido à falta de informações claras sobre o cálculo da componente *Wij*, o *score* final foi determinado exclusivamente com base na função *COST*.

Fase de evolução

Após a avaliação inicial da população, o algoritmo entra na fase de evolução, onde são geradas novas gerações de indivíduos.
Cada geração tem como objetivo melhorar a qualidade média dos alinhamentos,

promovendo a sobrevivência dos melhores indivíduos e substituindo os menos adaptados por novos candidatos obtidos através de operações genéticas.

Este processo é implementado na função *create_next_generation*, responsável por construir a geração seguinte a partir da atual.

No início de cada iteração, procede-se à aplicação da taxa de substituição, isto é, a proporção de indivíduos da geração anterior que será eliminada para dar lugar a novos indivíduos.
De acordo com o método de overlapping generations descrito no artigo de referência, apenas uma parte da população (tipicamente 50%) é substituída, garantindo assim a continuidade das soluções mais promissoras entre gerações consecutivas.

Com base nesta proporção, os indivíduos com *scores* mais baixos são então removidos, uma operação simples, uma vez que basta eliminar os últimos elementos da lista até restarem apenas os mais aptos.

Os indivíduos sobreviventes representam a elite da população, ou seja, as soluções de melhor qualidade que são preservadas sem alterações para a próxima geração.
Os restantes indivíduos necessários para completar a população são gerados através das fases seguintes do algoritmo: seleção de pais, cruzamento e mutação.
Estas operações introduzem diversidade e permitem a exploração de novas combinações de alinhamentos, assegurando o equilíbrio entre exploração (gerar novas soluções) e exploração local (aperfeiçoar soluções existentes).

Seleção de Pais

A seleção dos pais é uma das partes mais importantes do algoritmo genético, pois define quais os indivíduos da geração atual que vão ser usados para criar os indivíduos da próxima geração. A forma como os pais são escolhidos tem impacto direto na qualidade e na diversidade das soluções obtidas.

De forma geral, indivíduos com scores mais altos representam soluções melhores. Por isso, é esperado que os seus descendentes também tenham bons resultados. No entanto, se escolhermos sempre apenas os melhores indivíduos, como o de maior score ou os dois melhores no caso de operações que requerem mais de um indivíduo, a população perde diversidade rapidamente. Isso pode fazer o algoritmo ficar preso numa solução que não é a ideal, deixando de evoluir, ficando preso num máximo local ao invés de atingir um máximo global.

Para evitar este problema, usamos um método de seleção aleatória, mas com preferência pelos melhores indivíduos, tal como descrito no artigo. Isto significa que todos os indivíduos têm uma hipótese de serem escolhidos, mas os que têm *score* mais alto têm uma probabilidade maior.

O processo funciona da seguinte forma: se existirem scores negativos, é adicionado um valor a todos, positivos ou negativos, para que fiquem positivos e a distância entre scores

se mantenha a mesma. Depois, calcula-se a soma de todos os *scores* da população. Por fim, a probabilidade de cada indivíduo ser escolhido é obtida dividindo o seu score pela soma total.

Desta forma, os indivíduos com scores mais altos têm mais hipóteses de gerar descendentes, mas os restantes continuam a poder ser escolhidos, o que garante alguma variedade.

Operators

Depois de escolher os pais, o passo seguinte é aplicar os operadores genéticos, que servem para introduzir novas variações nas gerações seguintes. No nosso projeto implementámos dois operadores: o crossover e o operador “Split a randomly selected gap block”. De acordo com a nossa interpretação do artigo, o crossover funciona de forma semelhante aos outros operadores, sendo selecionado de forma aleatória, ou seja, não haverá indivíduos que sejam obtidos através de uma operação de crossover seguida de uma operação de “Split a randomly selected gap block”. É importante notar que cada operador precisa de um número diferente de país. O crossover utiliza dois indivíduos, enquanto a divisão de um bloco de gaps precisa apenas de um. Por isso, a escolha do operador é feita antes da seleção dos pais, mas a operação em si só é executada depois de os pais terem sido definidos.

Split a randomly selected gap block

Depois de escolhido o pai, é chamada a função *mutate_split_gap*. O primeiro passo nesta função é criar uma cópia do pai, que é representado como uma lista de listas, tal como foi referido anteriormente. Esta cópia é necessária porque não queremos alterar diretamente o indivíduo original, mas sim devolver uma nova versão modificada que representa o *offspring* com a mutação aplicada.

O processo de mutação é relativamente simples. Primeiro, é escolhida uma sequência aleatória dentro do indivíduo. Apenas essa sequência será alterada. De seguida, percorremos a sequência à procura de blocos de gaps, considerando como bloco qualquer conjunto com pelo menos dois gaps consecutivos. Cada bloco encontrado é guardado numa lista de tuplos que contêm o índice inicial e final do bloco.

Se a sequência selecionada não contiver blocos de gaps, o algoritmo passa para a sequência seguinte de forma circular, isto é, se estiver na última sequência, volta à primeira. O processo continua até ser encontrada uma sequência que possa ser mutada. Caso não exista nenhuma, o *offspring* resultante será igual ao pai, pois não há alterações a aplicar.

Quando é encontrado pelo menos um bloco, será escolhido um destes aleatoriamente para ser dividido. A divisão também é feita de forma aleatória, ou seja, um bloco de seis gaps, por exemplo, pode ser dividido em dois blocos de três gaps, mas também em blocos de

tamanhos desiguais, como um e cinco gaps. As chances de ambos os cenários se realizarem são idênticas.

Depois de definidos os dois sub-blocos, o algoritmo calcula os locais onde estes serão inseridos na sequência. Estes locais são determinados por *offsets* limitados a valores entre um e três. O sub-bloco da esquerda é deslocado para uma posição anterior (*offset* negativo), enquanto o sub-bloco da direita é inserido mais à frente (*offset* positivo).

Por fim, a sequência é reconstruída com os dois novos blocos colocados nas respectivas posições e o bloco original é removido. É importante garantir que os novos blocos não ultrapassam os limites da sequência. Quando o bloco original está no início ou no fim da sequência, o *offset* tem de ser ajustado para evitar posições inválidas.

Crossover

Nós implementámos dois crossovers simples, um deles é implementado segundo o que está descrito no pdf da preparação para o projeto e o outro fizemos nós, e baseia-se apenas em offsets das sequências.

Crossover "offsets"

O crossover com offsets implementa uma forma alternativa de crossover entre dois alinhamentos múltiplos de sequências, baseada na troca das posições relativas das sequências (*offsets*) em vez de operar diretamente sobre os resíduos. A função *extract_offsets_and_residues* separa cada sequência em duas partes: o número de gaps iniciais (*offset*) e a sequência de resíduos sem gaps. Assim, obtêm-se duas listas paralelas, uma com as posições iniciais e outra com as sequências puras.

O crossover é então realizado apenas sobre os offsets. A função *generate_offspring* verifica primeiro se as sequências de resíduos são idênticas entre os dois alinhamentos, assegurando que o cruzamento é válido. Caso sejam, define-se um ponto de corte (aleatório, se não for especificado) e trocam-se os *offsets* entre os dois alinhamentos a partir desse ponto. Desta forma, as sequências mantêm exatamente os mesmos resíduos, mas assumem novas posições relativas dentro do alinhamento.

Por fim, a função *reconstruct_from_offsets* reconstrói os novos alinhamentos a partir dos *offsets* e das sequências de resíduos, ajustando os comprimentos e removendo colunas compostas apenas por gaps. O resultado são dois novos alinhamentos que contêm as mesmas sequências biológicas, mas com diferentes distribuições de gaps.

Crossover "lab"

O funcionamento do crossover depende de um conjunto de funções auxiliares que asseguram a consistência e a validade biológica do resultado. A função *count_residues*

conta o número de resíduos reais numa sequência, ignorando os gaps. Já *index_at_residue* converte um número de resíduos para o respetivo índice na sequência alinhada, ou seja, identifica a posição exata onde se encontra o *n*-ésimo resíduo. A função *split_at* é responsável por dividir cada sequência em duas partes, à esquerda e à direita do ponto de corte, e devolve também o número de resíduos presentes na parte esquerda. Por sua vez, *residues_to_indexes* utiliza essa informação para calcular os índices de corte equivalentes no segundo alinhamento, garantindo que o corte é feito após o mesmo número de resíduos em ambos os alinhamentos, mesmo que a disposição dos gaps seja diferente.

Para manter a compatibilidade entre as duas metades, a função *pad_alignment* adiciona gaps à direita ou à esquerda das sequências, conforme necessário, de forma a que todas as linhas dentro de um alinhamento tenham o mesmo comprimento. A função *merge* concatena as partes correspondentes de cada sequência, unindo as metades esquerda e direita e formando novas combinações. Finalmente, as funções *is_all_gaps* e *clean_alignment* verificam se existem colunas compostas apenas por gaps e removem-nas, produzindo um alinhamento final mais limpo e válido.

A função principal, *generate_offspring*, integra todos estes passos de forma coordenada. Quando o ponto de crossover não é fornecido, ele é escolhido aleatoriamente entre 1 e o comprimento total do alinhamento. Em seguida, o primeiro alinhamento (*align1*) é dividido nesse ponto, e o número de resíduos da parte esquerda é registado para cada sequência. Com base nesse número de resíduos, o segundo alinhamento (*align2*) é dividido em posições equivalentes, calculadas de forma a manter a correspondência biológica entre os dois conjuntos de sequências. Após as divisões, as partes são padronizadas em comprimento com o uso de gaps, o que permite a recombinação correta.

O primeiro descendente (*offspring1*) é então formado ao combinar a parte inicial de *align1* com a parte final de *align2*, enquanto o segundo descendente (*offspring2*) resulta da combinação oposta, isto é, a parte inicial de *align2* com a parte final de *align1*. No final, ambos os descendentes passam por um processo de limpeza que remove as colunas constituídas apenas por gaps, obtendo assim alinhamentos finais coerentes e biologicamente consistentes.

Uso de ferramentas de inteligência artificial

Durante o desenvolvimento do projeto, recorremos a ferramentas de inteligência artificial para apoiar e acelerar algumas fases do trabalho. Estas ferramentas foram usadas sobretudo para nos ajudar a compreender melhor o artigo científico, permitindo identificar rapidamente as partes mais relevantes e esclarecer dúvidas sobre determinados conceitos.

Também serviram de apoio durante a implementação do código, nomeadamente na criação de testes, detecção de erros e correção de pequenos detalhes durante o processo de debugging.

No entanto, é importante reforçar que todas as decisões de design, lógica e implementação do algoritmo foram tomadas por nós. As ferramentas de inteligência artificial funcionaram

apenas como suporte técnico, nunca substituindo o nosso raciocínio nem a autoria do trabalho desenvolvido.