Inteligencia Artificial

Pedro Emanuel Ferreira Paiva das Neves - 510727o

Francisco Breno da Silveira - 511429

David Alves Soares - 511105

November 2023

1 Introdução

Os algoritmos genéticos (AGs) são técnicas de otimização inspiradas na seleção natural que têm sido aplicadas com sucesso em uma variedade de problemas. Este relatório aborda a implementação de um AG para otimizar a alocação de enfermeiros em turnos, considerando diversas restrições. Abordaremos a implementação do crossover, elitismo e mutação, bem como os efeitos desses parâmetros nas soluções obtidas.

2 Algoritmo

O algoritmo construido foi o seguinte. A primeira parte refere-se às restrições constatadas no pdf da prática, ao qual pode ser haver no mínimo 1 enfermeiro e no máximo 3, cada enfermeio deve ser alocado em 5 turnos por semana, não pode haver mais de 3 dias sem folga aos enfeirmeiros e todos eles contêm sustencia a sua carga de trabalho. Na sequência, temos a parte que refere-se a implementação dos métodos utilizados para compor a lógica do algoritmo, como o crossover e o método que realiza a mutação, centralizada no método run_evolution. Por fim, temos a terceira parte do código que refere-se aos métodos utilizados para realizar as experimentações, gravar logs e gerar gráfos.

3 Experimentações

- Implementação do crossover

  O método crossover recebe dois genomas (genomeA e genomeB) como entrada e aplica a lógica de crossover entre eles. A seguir estão os passos e explicações para cada parte do código:

  – Verificação de tamanho - parte 01: Verifica se os genomas têm o mesmo tamanho. Se não, gera um erro.

```
        if len(genomeA) != len(genomeB):
            raise ValueError("Os genomas A e
                B precisam ser do mesmo
                tamanho!")
```

  – Verificação de tamanho - parte 02: Retorna a entrada do programa se os genomas tiverem tamanho 1 ou inferior.

```
        length = len(genomeA)
        if length < 2:
            return genomeA, genomeB
```

  – Conversão para o formato de linha: Converte as matrizes bidimensionais genomeA e genomeB em linhas únicas usando a função convertMatrixToSingleLine.

```
        genA =
            convertMatrixToSingleLine(genomeA)
        genB =
            convertMatrixToSingleLine(genomeB)
```

  – Escolhendo o ponto de crossover: Gera um ponto de crossover (p) aleatório entre 1 e length - 1, onde length é o comprimento dos genomas. Este ponto é utilizado para dividir as duas linhas únicas para formar novos genomas.

```
        p = randint(1, length - 1)
```

  – Criação de novos genomas: Para o novo genoma newGenA, combina a primeira parte de genA até o ponto de crossover p com a segunda parte de genB após o ponto de crossover. Para o novo genoma newGenB, combina a primeira parte de genB até o ponto de crossover p com a segunda parte de genA após o ponto de crossover. Confira o trecho de código a seguir

```
        newGenA = genA[0:p] + genB[p:]
        newGenB = genB[0:p] + genA[p:]
```

- Conversão de volta para matriz e retorno do método: Converte os novos genomas de volta para as matrizes bidimensionais usando a função convertSingleLineToMatrix. As dimensões das matrizes de saída são determinadas pelo comprimento das linhas das matrizes originais (genomeA e genomeB). Esse valor é o valor retornado pelo método.

```
        return
            convertSingleLineToMatrix(newGenA,
            len(genomeA[0])),
            convertSingleLineToMatrix(newGenB,
            len(genomeB[0]))
```

- Implementação do Elitismo e Mutação
  - **Elitismo**: Durante a formação de uma nova geração, é possível transferir alguns indivíduos da geração anterior para a subsequente, com o objetivo de preservar os melhores elementos. O critério para esse deslocamento é estabelecido pela taxa de elitismo, um valor variando entre 0 e 1, conforme implementado no trecho de código explicado a seguir.
  - Condição de Elitismo: Verifica se um número aleatório gerado pela função random() é menor que a taxa de elitismo (elitism_ratio). Se sim, o bloco de código dentro do condicional será executado.

```
            if random() < elitism_ratio:
```

  - Marcação para subtração: Define a variável canSubtract como True. Essa variável é utilizada em outras partes do código para indicar que o elitismo está ativo.

```
            canSubtract = True
```

  - Construir Próxima Geração com Elitismo: Quando o elitismo está habilitado, a formação da próxima geração leva isso em consideração. A função sort_population organiza a população com base em critérios específicos definidos pelos argumentos r1Min, r1Max, r2NumTurn, r3MaxConsecWorkDays. Em seguida, os dois melhores indivíduos dessa população ordenada, selecionados pela expressão [0:2], são incorporados como membros fundamentais da próxima geração.

```
            next_generation =
                sort_population(population,
                r1Min, r1Max, r2NumTurn,
                r3MaxConsecWorkDays)[0:2]
```

  - Esse trecho de código se encontra na implementação do método run_evolution é executado em cada interação.

– **Mutação**: Cada gene no genoma tem uma probabilidade de ser invertido (de 0 para 1 ou de 1 para 0), controlada pela taxa de mutação. O método que aplica essa lógica é explicado a seguir.

– Conversão para formato de lista: Converte a matriz bidimensional genome em uma linha única usando a função convertMatrixToSingleLine e, em seguida, transforma essa linha em uma lista.

```
genomeL = list(
convertMatrixToSingleLine(genome)
)
```

– Seleção aleatória de índice: Gera um índice aleatório dentro do intervalo da lista genomeL. Este índice será usado para selecionar um gene específico para mutação.

```
index = randrange(len(genomeL))
```

– Aplicação da mutação: Avalia se a mutação ocorre com base em um número aleatório comparado com a taxa de mutação (mutationRatio). Se a condição for atendida, o gene no índice selecionado (genomeL[index]) permanece inalterado. Caso contrário, o gene é mutado, alternando entre 0 e 1 (representado pelo cálculo abs(int(genomeL[index]) - 1)).

```
genomeL[index] = genomeL[index]
    if random() > mutationRatio
    else abs(int(genomeL[index]) -
    1)
```

– Conversão para matriz e retorno do método: Converte a lista modificada genomeL de volta para a matriz bidimensional usando a função convertSingleLineToMatrix. As dimensões da matriz de saída são determinadas pelo comprimento das linhas da matriz original (genome). Esse é o valor retornado pelo método.

```
return convertSingleLineToMatrix(
    genomeL,
    len(genome[0])
)
```

Efeitos dos Parâmetros:

– O aumento da taxa de elitismo geralmente leva a uma convergência mais rápida, mas pode resultar em soluções subótimas, pois limita a diversidade genética.

– A taxa de mutação influencia a diversidade genética, evitando que a população fique presa em ótimos locais. No entanto, uma taxa muito alta pode prejudicar a convergência.

Bloco de experimentação 01 - Influência do Elitismo nas Iterações e Qualidade das Soluções

- Número de Iterações: Observou-se que um aumento na taxa de elitismo geralmente reduz o número de iterações necessárias para convergência, tornando a busca mais eficiente.
- Qualidade das Soluções: O elitismo melhora a qualidade das soluções, mas taxas muito altas podem levar a soluções locais ótimas.

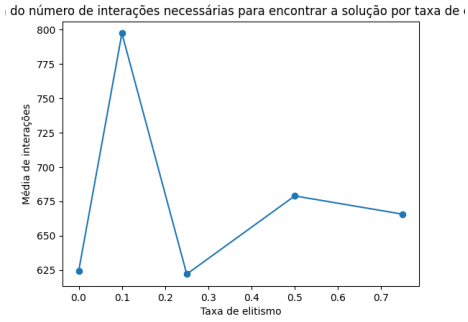

Figure 1: Média de interações por taxa de Elitismo

- **Bloco de experimentação 02** - Influência do Tamanho da População nas Iterações e Qualidade das Soluções
  - Número de Iterações: Aumentar o tamanho da população tende a acelerar a convergência, pois mais indivíduos exploram o espaço de busca.
  - Qualidade das Soluções: Populações maiores proporcionam maior diversidade genética, melhorando a qualidade das soluções, mas também aumentam os recursos computacionais necessários.

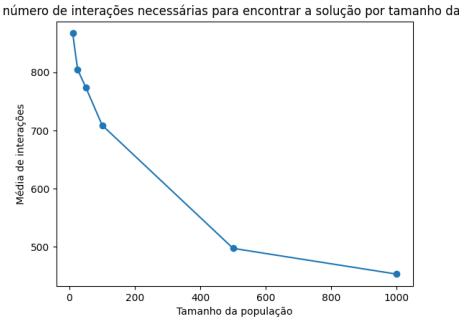

Figure 2: Média de interações por tamanho da população