INSTITUTO FEDERAL ESPÍRITO SANTO Campus Serra	**INTELIGÊNCIA ARTIFICIAL** **EP2: Estudo Comparativo de Algoritmos de Otimização** **PPCOMP – IFES – Campus Serra**	Nota

Professor: Sérgio Nery Simões Data de Entrega: 15/01/2025

Nome: _____ Turma: 2024/2

Informações:

- **Deve ser submetido um arquivo compactado contendo o código desenvolvido em Python e o relatório em formato PDF.**
- **É permitido aos alunos discutirem suas soluções, mas os códigos-fonte devem ser produzidos individualmente. Casos de plágio serão investigados e podem resultar em sanções, incluindo o desligamento do aluno.**

Descrição do Trabalho

Contexto: Este projeto de programação consiste em comparar o desempenho dos algoritmos de otimização aplicados a diferentes problemas. Os resultados do projeto serão (i) o código desenvolvido para cada problema e (ii) um breve relatório apresentando e discutindo os resultados comparativos. Os algoritmos a serem comparados incluem:

- **[HC-C] – *Hill-Climbing* (clássico);**
- **[HC-R] – *Hill-Climbing with Restart*;**
- **[SA] – *Simulated Annealing* e**
- **[GA] – *Genetic Algorithm*.**

Desenvolvimento: Os algoritmos devem ser aplicados na busca pela solução ótima de dois problemas definidos a seguir. Cada problema inclui a definição de uma função objetivo e as funções específicas para geração de vizinhos e, no caso do GA, para mutação e *crossover*. Também são definidos também os parâmetros que devem ser usados pelos métodos. Os algoritmos de otimização devem ser executados pelo menos **10 vezes**, para permitir a análise de **variabilidade** dos resultados no relatório.

(1) **Problema 1: *Travelling Salesman Problem* (TSP)** – Resolver o problema do caixeiro-viajante, otimizando o trajeto.

(2) **Problema 2: Minimização da Função de Rastrigin** – Minimizar uma função matemática complexa frequentemente usada em benchmarks de otimização.

Para ajudá-los a começar o desenvolvimento e prover um ambiente para testes dos algoritmos, foram fornecidos os seguintes códigos:

(a) Implementação do algoritmo *Hill-climbing* (clássico) aplicado ao TSP, junto a funções para geração de gráficos comparativos.

(b) Funções base do Algoritmo Genético (GA), desenvolvidas para o problema das 8-rainhas.

Bom trabalho!

A partir dos ambientes (a) e (b), os algoritmos deverão ser adaptados de acordo com as especificações de cada problema, fornecida mais a frente. Os algoritmos principais devem ser implementados de forma **modular**, com alterações específicas concentradas em funções dedicadas a cada problema.

É importante ressaltar que é proibido utilizar bibliotecas que implementem diretamente os algoritmos de otimização. No entanto, bibliotecas auxiliares para manipulação de estruturas de dados são permitidas. A implementação desses algoritmos e sua aplicação em problemas reais são úteis em entrevistas técnicas para empresas de grande porte, representando um investimento valioso na sua carreira profissional.

Relatório

Formato do Relatório: Para cada problema, o relatório deve incluir tabelas e gráficos que sintetizem os resultados obtidos, acompanhados de uma breve discussão.

- Formato da discussão: A discussão deve apresentar os resultados obtidos (e.g., identificação do algoritmo com melhor desempenho) e explicar as observações feitas, com base nos conceitos aprendidos durante o curso.
    - Exemplo: "*O algoritmo de Hill-Climbing obteve uma performance inferior aos outros porque ficou preso em um mínimo local. Isso pode ser observado na Figura X que mostra a evolução da função objetivo. O valor da função objetivo parou de diminuir depois de N iterações*".

Os gráficos e tabelas a serem produzidos são os seguintes:

- Gráfico Box-plot com a sumarização dos resultados: Para comparar os resultados dos algoritmos, crie um gráfico box-plot que mostre os valores máximos, mínimos, medianas, e percentis 25 e 75 de cada algoritmo. Consulte a Figura 1 como exemplo.
- Tabela com sumarização de resultados: Para cada algoritmo, separe o valor da função objetivo ao final de cada uma das 10 execuções do processo de otimização. A tabela deve comparar os valores máximos, mínimos, medianos e o desvio padrão dos 10 valores (ver ex. da Tabela 1).
- Gráfico da evolução da fitness em N iterações do algoritmo: Produza um gráfico para cada algoritmo, onde o eixo $\boldsymbol{x}$ indica o número de chamadas à função objetivo e o eixo $\boldsymbol{y}$ apresenta o melhor valor encontrado até o momento. A Figura 2 traz um exemplo do gráfico: cada curva deve representar apenas uma das 10 execuções do algoritmo. Note que as curvas monotonicamente decrescentes, pois mostram apenas o melhor valor alcançado até aquele ponto.

Pontuação: será proporcional a realização completa das atividades descritas.

Questões	**Pontuação**
Problema 1: TSP	40 pontos
Problema 2: Otimização da Função ***Rastringin***	60 pontos
Obs: pontuação de cada algoritmo em cada problema	HC (25%), SA (30%) e GA (45%)

Bom trabalho!

Exemplos de Figuras e Tabelas para serem usados na discussão dos resultados

Figura 1: Exemplo de gráfico com os boxplots dos custos finais após 10 execuções dos algoritmos A, B e C.

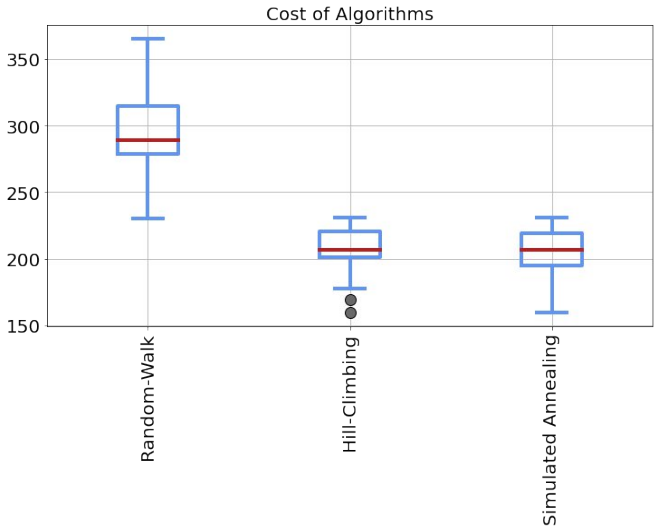
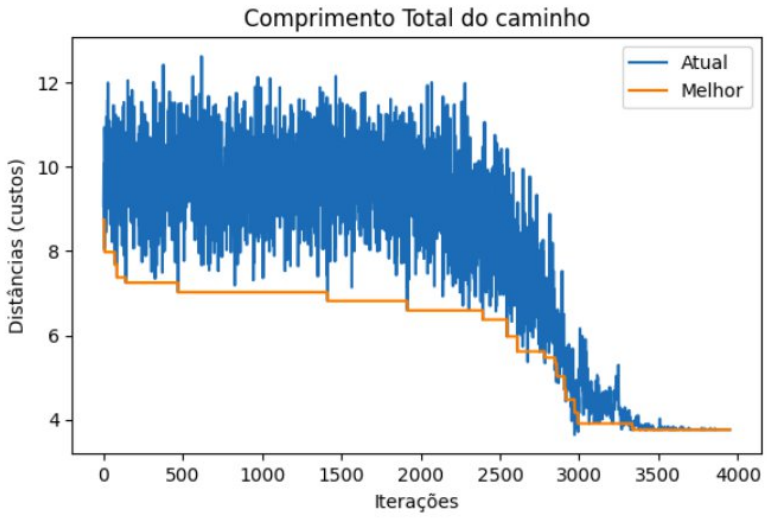


Tabela 1: Tabela de exemplo com estatísticas das execuções dos algoritmos de otimização A, B, C e D para o problema X.

Algoritmo	Max	Min	Mediana	Desvio Padrão
HC	10.123	5.788	7.321	1.551
HC-R	3.442	0.123	1.235	1.982
SA	1.554	0.012	0.777	0.156
GA	0.998	0.001	0.230	0.098

Figura 2: Exemplo de gráfico comparando a evolução da função objetivo em uma das 10 execuções do algoritmo de otimização (SA).



Bom trabalho!

Problemas

Problema 1: Este problema de otimização consiste em resolver o ***Travelling Salesman Problem*** (TSP), conhecido como o problema do caixeiro-viajante. A entrada do TSP é um conjunto de pontos que representam as posições das cidades. A solução ideal é o trajeto mais curto que começa em uma cidade inicial, visita todas as outras uma vez, e retorna ao ponto de partida. A Figura 3 exemplifica duas soluções para o TSP para um conjunto de pontos dado. Anexo à esta especificação são dados alguns conjuntos de pontos para realização de testes do algoritmo de otimização e um conjunto de pontos para o teste final e escrita do relatório. Os arquivos contêm as coordenadas (x, y) de um ponto por linha. Assuma que a cidade inicial do trajeto é dada pelo primeiro ponto do arquivo.

- **Função Objetivo**: minimizar o tamanho total do trajeto.
- **Representação do estado**: A solução será representada como uma sequência de números inteiros que define a ordem de visitação das cidades. Como a primeira cidade da lista é sempre o início e fim do trajeto, não é necessário adicionar ela ao estado (mas ela deve ser considerada no cálculo da função objetivo). Como cada cidade deve ser visitada apenas uma vez, não devem existir repetições de cidades no estado. A Figura 4 ilustra trajetos e o valor de estado de acordo com a representação proposta.
- **Geração de Vizinhos e Mutação**: Para gerar vizinhos ou realizar mutação no algoritmo genético, será feita a troca de posição entre duas cidades selecionadas aleatoriamente no trajeto. Por exemplo, o estado [2, **4**, 5, **6**, 3] pode se transformar em [2, **6**, 5, **4**, 3] ao trocar as posições das cidades 4 e 6.
- **Crossover**: A operação de *single-point crossover* não é adequada para estados representados como permutações de números porque ela pode gerar repetição ou ausência de números (cidades) nos estados filhos. Por exemplo, assuma que os estados pais são **p1** = [**1, 3, 5, 2, 6, 4**] e **p2** = [**6, 4, 1, 3, 2, 5**] e que a posição escolhida para *crossover* é a posição 1 (assumindo índices que começam de 0). Então, usando o *single-point crossover*, os filhos seriam **c1** = [**1, 3, 1, 3, 2, 5**] e **c2** = [**6, 4, 5, 2, 6, 4**]. Observe que em **c1**, as cidades 1 e 3 são visitadas 2 vezes e as cidades 4 e 6 nunca são visitadas. Já em **c2**, as cidades 4 é visitada 2 vezes e a cidade 5 nunca é visitada. Para evitar problemas com duplicação ou ausência de cidades, será utilizado um operador especial de *crossover*, chamado ***Order 1 Crossover*** (***OX Crossover***). Detalhes sobre este método estão disponíveis em [5] e [6].
- **Parâmetros**: O número de iterações, o tamanho da população e a quantidade de gerações no algoritmo genético devem ser ajustados conforme o número de cidades. Quanto mais cidades, mais difícil o problema e, portanto, mais chamadas à função objetivo podem ser necessárias. **Importante**: para uma comparação justa, garanta que todos os algoritmos façam um mesmo número de chamadas à função objetivo. Teste diferentes número de ***restarts*** até encontrar um que leve a bons resultados no [HC-R] (*Hill-Climbing with Restart*). Para o [SA] (*Simulated Annealing*), defina a probabilidade de aceitar um vizinho pior que o atual inicialmente como 1 e reduza a probabilidade linearmente de forma a chegar a 0 após 90% iterações. Para o algoritmo genético, use probabilidade de mutação entre 10% e 20%.
- **[Opcional] Visualização**: A visualização consiste em apresentar os pontos representando cidades e o trajeto dado pela solução atual. As Figuras 3 e 4 são exemplos desse tipo de visualização

Bom trabalho!

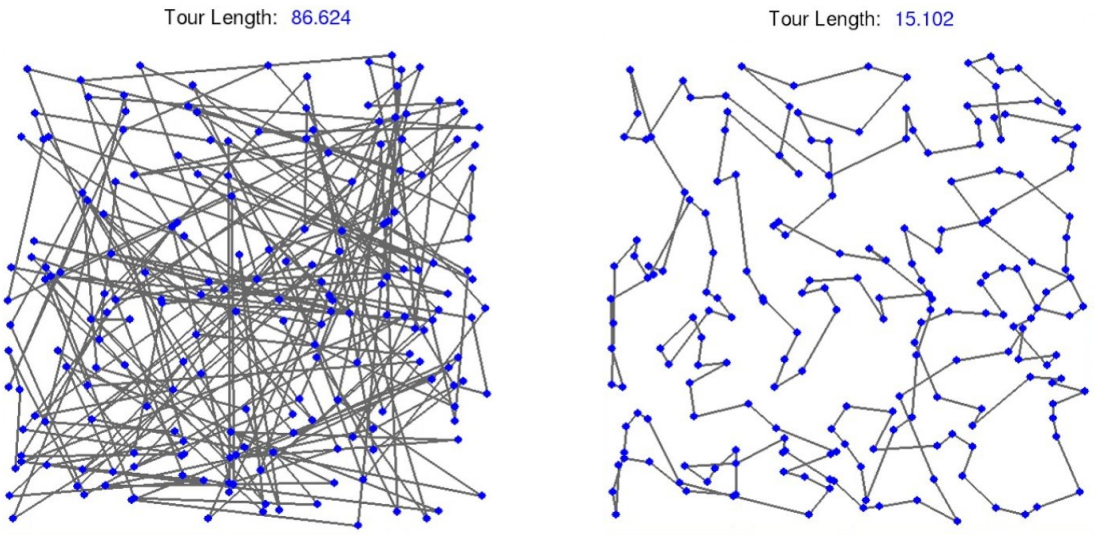


Figura 3: As figuras apresentam soluções para o problema do caixeiro viajante (*travelling salesman problem* – TSP). A solução (a) tem custo de caminho inferior à solução (b). Observe que o número de cruzamentos de trechos em (b) é menor que em (a).

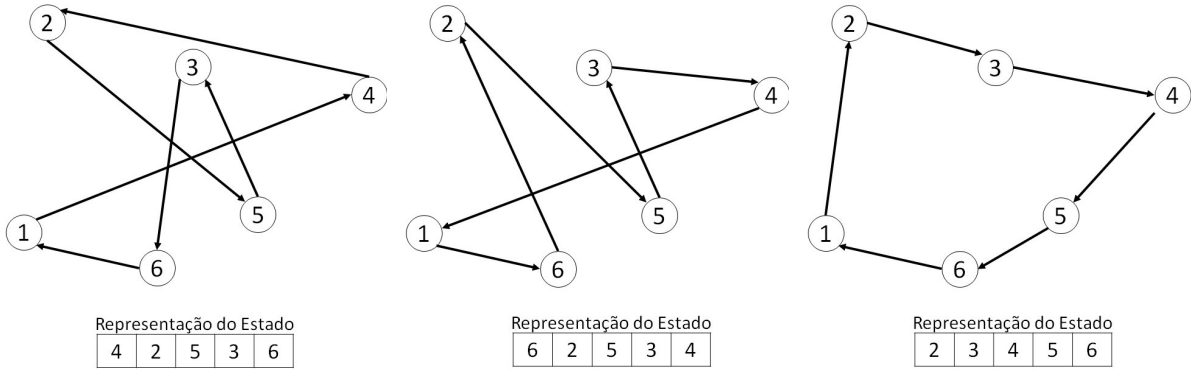


Figura 4: Exemplos de como as soluções do TSP serão representadas em nosso projeto.

Sugestões Gerais para a Realização dos Testes

Embora não seja obrigatório, é altamente recomendável implementar um programa para automatizar o processo de otimização. Esse programa deve aceitar como entrada o nome do algoritmo de otimização e o nome do problema, gerando como saída um arquivo que registre os estados visitados e os valores da função objetivo. Para o algoritmo genético, os estados devem ser organizados em populações. Esses arquivos tornam a produção do relatório mais rápida e eficiente. Para criar interfaces de linha de comando, a biblioteca argparse [3] é uma excelente opção.

Além disso, é recomendável desenvolver um script para realizar automaticamente as 10 execuções do algoritmo. Isso facilita a repetição dos testes, especialmente em caso de *bugs*, evitando a necessidade de refazer todas as execuções manualmente, o que pode ser trabalhoso e demorado.

Bom trabalho!

Problema 2: Minimizar a função de ***Rastrigin*** definida por:

$$f(x,y) = 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y)$$

no intervalo [-5.12, 5.12]. A Figura 6 apresenta o gráfico da função.

- **Produção de vizinhos e mutação**: A geração de vizinhos e as mutações no algoritmo genético são realizadas somando-se valores aleatórios (***x+Δx*** e ***y+Δy***) para as coordenadas ***x*** e ***y***, retirados de uma distribuição normal com **média 0** e desvio **padrão 0.2**, às coordenadas ***x*** e ***y*** do ponto.

- **Crossover**: Para cada coordenada, a operação de *crossover* entre dois pontos (p1 e p2) será realizada por meio de uma média ponderada: **c1 = p1 α + p2 (1 - α)** e **c2 = p2 α + p1 (1 - α)**, onde **α** é retirado de uma distribuição uniforme entre 0 e 1 (veja referência [2]).

- **Sugestão de Parâmetros:** Para todos os algoritmos exceto o genético, executar 1000 iterações. Para o [HC-R] (*Hill-Climbing with Restart*), fazer um *restart* a cada 50 iterações (20 *restarts* no total). Para o [SA] (*Simulated Annealing*), defina a probabilidade de aceitar um vizinho pior que o atual como 1 e reduza a probabilidade linearmente de forma a chegar a 0 após 900 iterações. A partir daí, mantenha a probabilidade constante em 0. Para o algoritmo genético, use uma população de 20 indivíduos, 50 gerações, e probabilidade de mutação de 10-30%.

- **[Opcional] Visualização:** Para todos os algoritmos exceto o genético, desenhar um ponto no gráfico da função objetivo representando o estado atual. Para o algoritmo genético, desenhar vários pontos no gráfico representando os indivíduos da população.

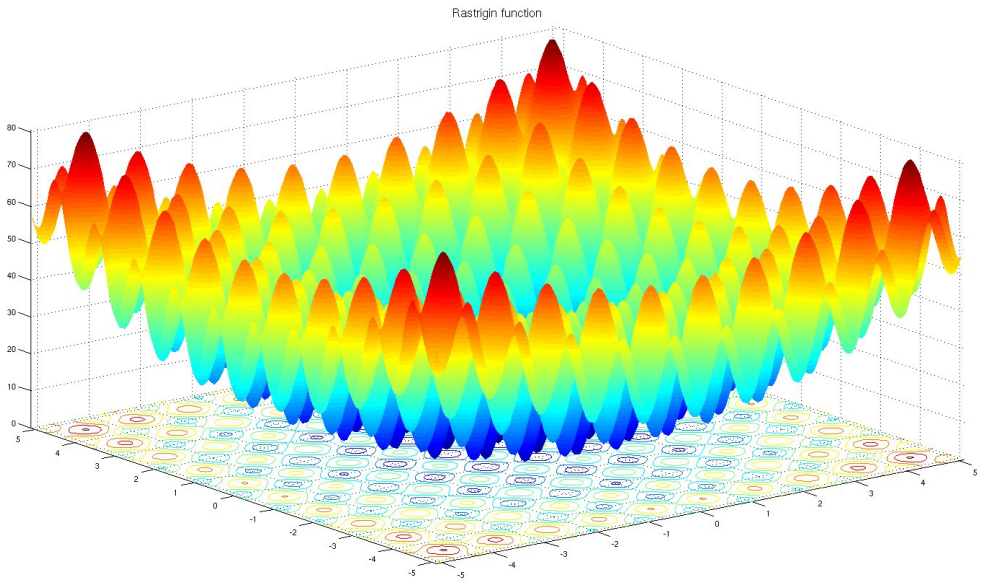


Figura 6:Visualização da função de Rastrigin.

Bom trabalho!

Referências

[1] Use a função numpy.random.randn da biblioteca numpy para amostrar o valor aleatório.
https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html?highlight=randn#numpy.random.randn

[2] Use a função numpy.random.uniform da biblioteca numpy para amostrar o valor aleatório.
https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.html?highlight=uniform#numpy.random.uniform

[3] Biblioteca argparse para criação de interfaces de linha de comando.
https://docs.python.org/3/library/argparse.html

[4] Use a função seaborn.lineplot (ou equivalente) para produzir o gráfico das curvas médias da função objetivo com os intervalos de variação.
https://seaborn.pydata.org/examples/errorband_lineplots.html

[5] https://www.youtube.com/watch?v=HATPHZ6P7c4

[6] Seção Order crossover (OX).
http://creationwiki.org/pt/Algoritmo_gen%C3%A9tico

Obs: Nos notebooks enviados também tem alguns exemplos de usos de algumas dessas funções para facilitar a solução dos problemas.

Bom trabalho!