INSTITUTO FEDERAL ESPÍRITO SANTO Campus Serra	**IFES** **Inteligência Artificial** Exercício de Programação 2: Problemas de Otimização	Nota

Professor: Sérgio Nery Simões Data: 20/11/2022

Nome: _____ Turma: _____

Informações:

- **O trabalho deve ser submetido até a data limite na atividade do AVA.**
- **Deve ser submetido um arquivo compactado contendo o código em Python dos programas desenvolvidos e o relatório em formato pdf.**
- **É permitido que os alunos conversem a discutem suas soluções, mas os códigos-fonte devem ser produzidos individualmente. Se forem identificados casos de cola, serão abertos processos que podem culminar no 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 são:

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

Desenvolvimento: Os algoritmos devem ser usados para tentar encontrar a solução ótima para 3 problemas (conforme especificação). Na descrição dos problemas são definidas a função objetivo e as funções para geração de vizinhos, e no caso específico do GA: mutação e *crossover*. Também são definidos também os parâmetros que devem ser usados pelos métodos. Os algoritmos de otimização devem ser executados 10 vezes (no mínimo) e o relatório deve apresentar os resultados.

(1) **TSP** (Travelling Salesman Problem – Problema do Cacheiro-Viajante);

(2) Minimização de Função Objetivo univariada;

(3) Problema das 8-rainhas (*8-queens Problem*).

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)** para busca local da solução aplicado ao problema TSP e algumas funções para geração de gráficos comparativos.

(b) Implementação das funções de base do Algoritmo GA aplicados ao problema das 8-rainhas.

A partir dos ambientes (a) e (b), os algoritmos deverão ser adaptados de acordo com as especificações de cada problema, fornecida mais a frente. Deve-se buscar implementar de forma estruturada, ou seja, desenvolver os algoritmos principais de forma mais macro e concentrar as alterações específicas de cada problema nas funções mais básicas.

Bom trabalho!

Importante mencionar que, para o desenvolvimento do trabalho, não é permitido o uso de bibliotecas que implementem os algoritmos, mas é permitido usar bibliotecas auxiliares, e.g., que implementem estruturas de dados. Esses algoritmos bem como sua como os problemas de aplicações podem ser úteis em entrevistas de programação em empresas de grande porte, então considere este tempo de implementação como um investimento na sua carreira.

Relatório

Formato do Relatório: O relatório deverá conter para cada problema, tabelas e gráficos sumarizando os resultados obtidos e apresentando uma breve discussão.

- Formato da discussão: A discussão deve descrever os resultados (qual algoritmo obteve o melhor resultado, etc.) e prover uma explicação para o que foi observado com base nos conhecimentos adquiridos.
  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 3 algoritmos, plote um gráfico contendo os boxplots de cada um, ou seja, para visualização dos valores máximos, mínimos, percentis 25, medianas, e percentis 75. Ver exemplo na Figura 1.
- 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: Para cada algoritmo deve ser produzido um gráfico no qual o eixo x represente o número de chamadas à função objetivo e o eixo y represente o melhor valor da função objetivo encontrado até aquele momento. A Figura 2 traz um exemplo do gráfico. Cada curva representa apenas uma das 10 execuções do algoritmo. Note que diferente do exemplo, a curva nunca subiria depois de descer, uma vez que estamos mostrando o melhor valor da função objetivo até aquele momento. Para produzir esse gráfico utilize a função *lineplot* da biblioteca seaborn [4].

Pontuação: a pontuação será proporcional a realização das atividades propostas

Questões	**Pontuação**
Problema 1: TSP	30 pontos
Problema 2: Otimização de Função (univariada)	30 pontos
Problema 3: Problema das 8-rainhas.	40 pontos
Obs: dentro de cada problema cada algoritmo tem um valor percentual:	HC (20%), SA (30%) e GA (50%)

Pontos Extras: O estudante receberá pontos extras se desenvolver alguma forma de visualizar os estados visitados ao longo do processo de otimização. Será atribuído +10% por problema. Exemplos de visualizações são apresentados na descrição dos problemas. Esta visualização permite que os estudantes observem o comportamento dos algoritmos ao longo do processo de otimização.

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.

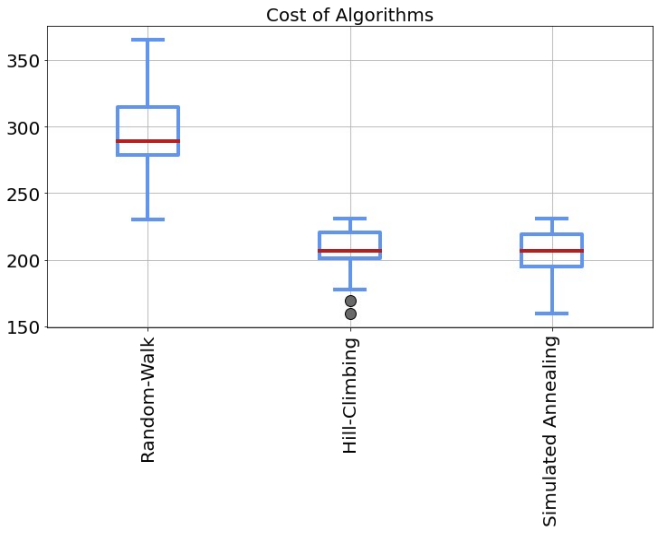
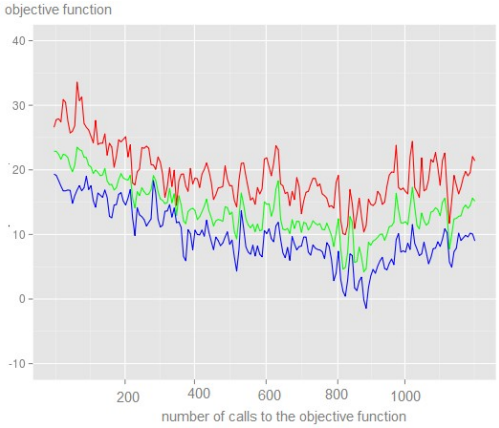


Tabela 1: Tabela de exemplo com estatísticas das execuções dos algoritmos de otimização A, B, C e D para o problema X.

Algoritmo	Max	Min	Mediana	Desvio Padrão
Hill-Climbing	10.123	5.788	7.321	1.551
Hill-Climbing w restart	3.442	0.123	1.235	1.982
Simulated Annealing	1.554	0.012	0.777	0.156
Algoritmo Genético	0.998	0.001	0.230	0.098

Figura 2: Exemplo de gráfico comparando a evolução da função objetivo nas 10 execuções de um algoritmo de otimização.



Bom trabalho!

Problemas

Problema 1: O problema de otimização consiste em encontrar uma solução para o problema do caixeiro viajante (*travelling salesman problem* - TSP). A entrada do TSP é um conjunto de pontos representando posições de cidades. A solução é o trajeto mais curto para, partindo de uma cidade inicial, visitar todas as cidades uma vez e depois retornar à cidade inicial. 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**: A função objetivo a ser minimizada é o tamanho total do trajeto.
- **Representação do estado**: Uma solução para o problema será dada por uma sequência de números inteiros representando a ordem em que as cidades serão visitadas. 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**: A geração de vizinhos e a operação de mutação no algoritmo genético serão dadas pela troca da posição de duas cidades escolhidas aleatoriamente no trajeto. Por exemplo, o estado [2, **4**, 5, **6**, 3] poderia ser transformado em [2, **6**, 5, **4**, 3] trocando 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 isto, usaremos um operador de *crossover* especial chamado ***Order 1 Crossover*** ou (OX Crossover). Explicações dele são apresentadas em [5] e [6].
- **Parâmetros**: O número de iterações assim como o tamanho população e o número de gerações do genético devem ser ajustados dependendo do 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. A partir daí, mantenha a probabilidade constante em 0. 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!

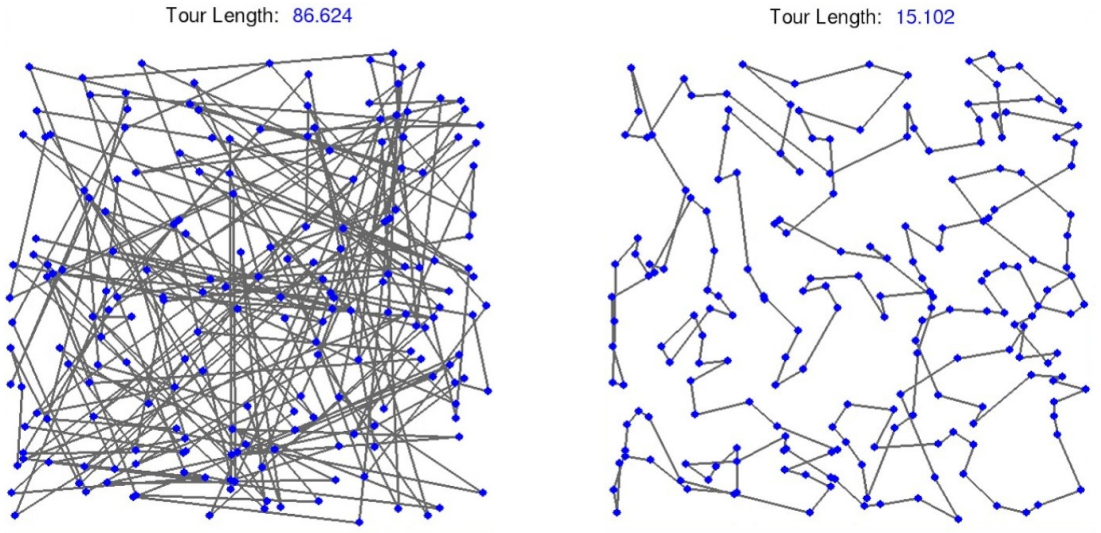


Figura 3: As figuras apresentam soluções para o problema do caixeiro viajante (*travelling salesman problem* – TSP). A solução (a) tem custo de caminho inferior à solução (b). Observe que o número de cruzamentos de trechos em (b) é menor que em (a).

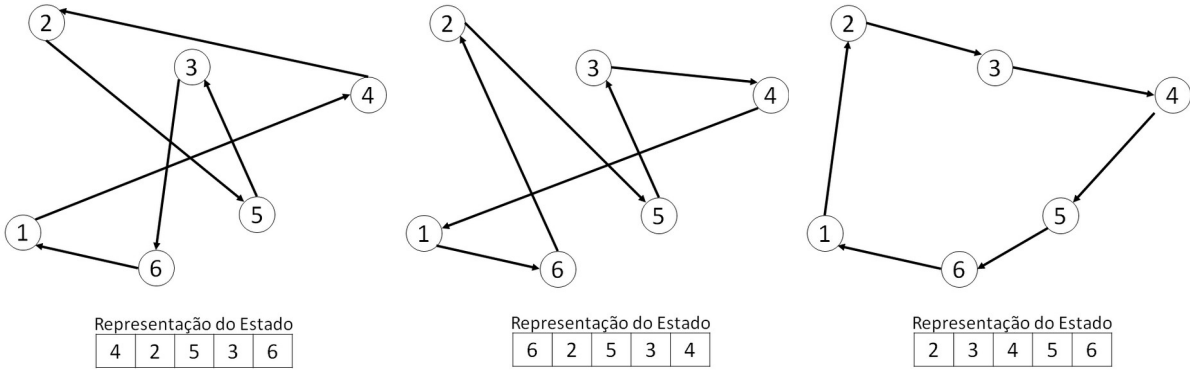


Figura 4: Exemplos de como as soluções do TSP serão representadas em nosso projeto.

Sugestões Gerais para a Realização dos Testes

- **Sugestão**: Não é obrigatório, mas bastante recomendável fazer um programa para rodar o processo de otimização. O programa receberia como entrada o nome do algoritmo de otimização e o nome do problema e geraria como saída um arquivo com os estados visitados e os valores da função objetivo (no caso do genético, os estados devem ser organizados em populações). Tendo estes arquivos, produzir o relatório será mais rápido e fácil. A biblioteca argparse [3] é uma boa alternativa para produzir interfaces de linha de comando.

- **Sugestão**: Não é obrigatório, mas recomendável fazer um *script* para fazer as 10 execuções programaticamente. Se vocês encontrarem bugs, podem ter que refazer os testes e ter que rodar tudo de novo manualmente toma bastante tempo.

Bom trabalho!

Problema 2: Minimizar a função objetivo $f(x) = \frac{x^2}{100} + 10\sin\left(x - \frac{\pi}{2}\right)$ no intervalo [-100, 100]. A Figura 5 apresenta o gráfico da função.

- **Produção de vizinhos e mutação**: Para produzir o vizinho de um estado e para realizar mutação no algoritmo genético, **some** ao estado um valor aleatório amostrado de uma distribuição normal com média 0 e desvio padrão 0.1 (veja [1]). Nesta distribuição, valores próximos de 0 possuem alta probabilidade e a probabilidade de escolher um número diminui à medida que o número se afasta de zero.
- ***Crossover***: Sejam p1 e p2 dois pontos escolhidos para produzirem filhos. A operação de *crossover* será dada pela média ponderada c1 = p1 α + p2 (1 - α) e c2 = p2 α + p1 (1 - α), onde α é um número aleatório amostrado de uma distribuição uniforme entre 0 e 1 (veja [2]).
- **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 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.

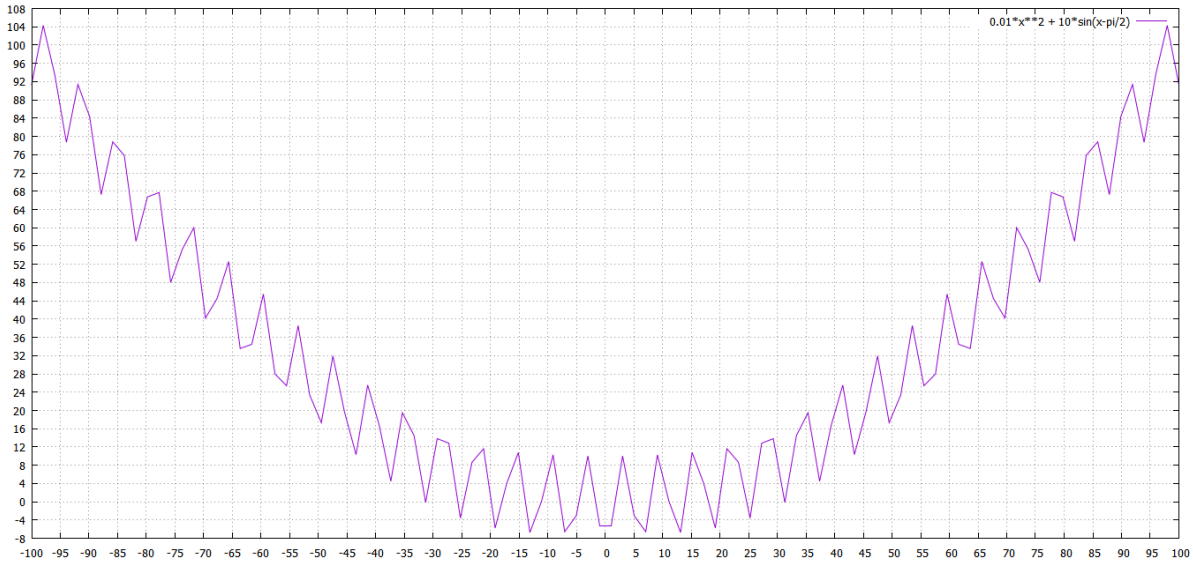


Figura 5: Gráfico da função $f(x) = \frac{x^2}{100} + 10\sin\left(x - \frac{\pi}{2}\right)$.

Bom trabalho!

Problema 3: O problema de otimização consiste em encontrar uma solução para o problema 8 rainhas (*8-Queens Problem*). Esse problema consiste em dispor 8 rainhas em um tabuleiro de Xadrez de forma que nenhuma rainha ataque a outra. Visto que neste problema existe apenas rainhas, num primeiro momento a representação poderia ser uma matriz quadrada de 0's e 1's, em que 0 representa um espaço vazio e 1 representa um espaço ocupado por uma rainha. Mas para otimizar esta representação, o tabuleiro poderia ser representado por um vetor de 8 posições em que cada posição representa uma coluna do tabuleiro e o respectivo valor representa a linha ocupada por aquela rainha. Para ilustrar, a Figura 6 mostra um tabuleiro que pode ser representado pelo vetor **[8, 4, 8, 5, 5, 2, 7, 8]**.

As funções para a geração de vizinhos, cálculo da quantidade de rainhas que estão se atacando (custo/fitness), mutação e crossover, já foram apresentadas pelo código fornecido no Notebook colab e comentadas em aula. Desta forma, você deverá apenas adaptar o código para implementar os algoritmos [HC-R] (*Hill-Climbing with Restart*), [SA] (*Simulated Annealing*), e [GA] (*Genetic Algorithm*).

- **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 30%.

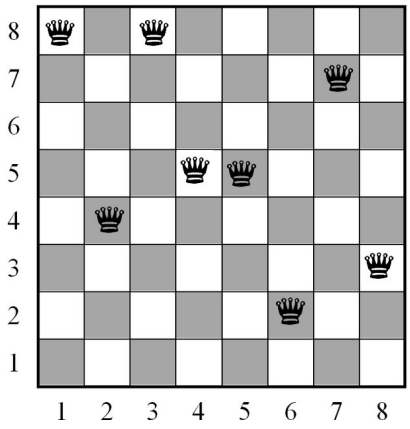


Figura 6: Exemplo de visualização do problema das 8-Rainhas para a representação pelo vetor **[8, 4, 8, 5, 5, 2, 7, 8]**.

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!