Universidade Federal de Campina Grande

Inteligência Artificial [2025.2]

APLICAÇÃO DE ALGORITMO GENÉTICO AO PROBLEMA DOS 100 PRISIONEIROS

Grupo:

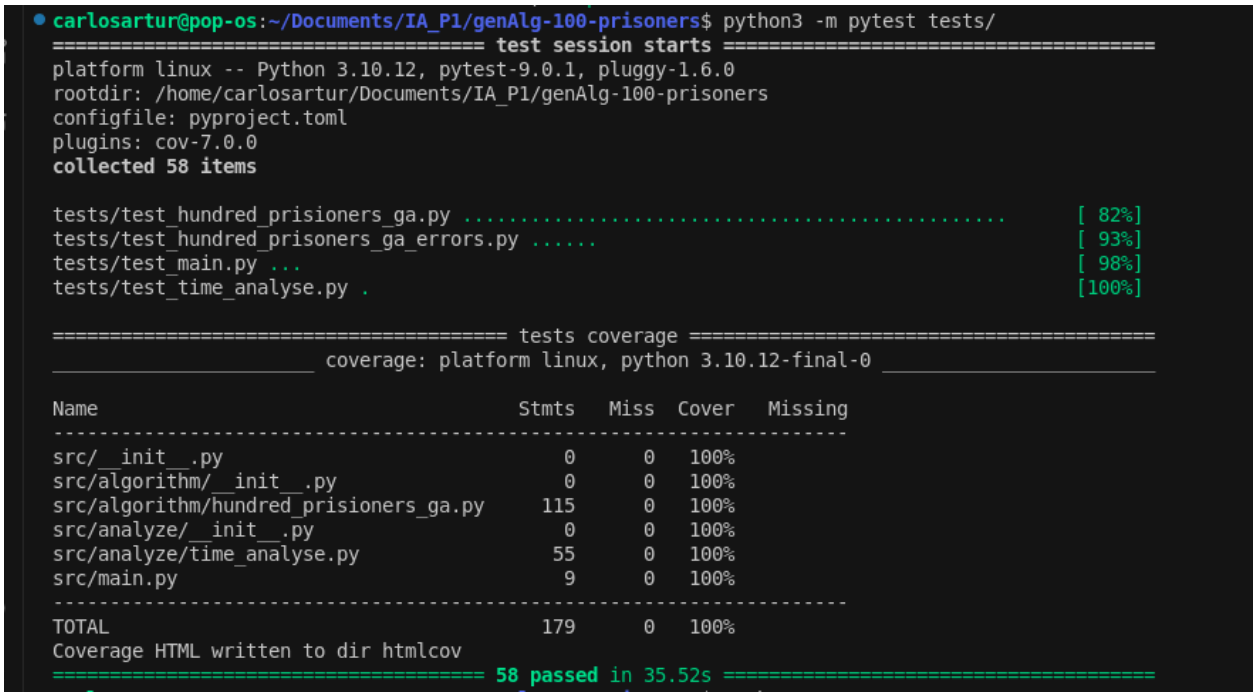
Arthur Vasconcelos Rego Barros | [matrícula: 123110824] - [função: Desenvolvedor]
Carlos Artur Santana Sales| [matrícula: 123110250] - [função: Testes]
Débora Sabrina de Oliveira Pereira | [matrícula: 123111211] - [função: Think Aloud]
Helder Chaves Leite Júnior | [matrícula:] - [função: Testes]
Rafael Morais Cavalcante | [matrícula: 120110265] - [função: Desenvolvedor]
Ruan Rodrigues da SIlva | [matrícula: 123210708] - [função: Desenvolvedor]

TEST-DRIVEN-DEVELOPMENT + COBERTURA

Para o desenvolvimento do projeto, foi adotado o processo de Test-Driven Development (TDD) utilizando pytest como framework principal e pytest-cov para cálculo da cobertura.
A suíte de testes criada validou tanto o comportamento normal das funções quanto o tratamento de erros e cenários limite. Os testes foram organizados em múltiplos arquivos, cobrindo:

Lógica do algoritmo genético, Validação e tratamento de erros, Execução do módulo principal (main), Funções de análise de tempo

Após a execução da bateria completa contendo 58 testes, obteve-se uma cobertura total de 100%, **conforme demonstrado na imagem abaixo:**



O uso disciplinado de TDD combinado com testes bem estruturados possibilitou a criação de um código mais robusto, confiável e facilmente evolutivo. Embora a cobertura de 100% não seja, por si só, um indicador absoluto de qualidade, já que cobertura não garante que todos os cenários reais foram validados, ela evidencia um esforço consistente na testagem do sistema e demonstra que todas as partes do código foram exercitadas durante a execução dos testes, reduzindo a probabilidade de falhas silenciosas. Esse nível de atenção à testabilidade reforça a maturidade do projeto e facilita futuras manutenções ou extensões, garantindo maior segurança nas modificações e no comportamento esperado do sistema.

RELATÓRIO THINK ALOUD

O problema em questão envolve 100 prisioneiros e 100 caixas, cada uma contendo um número de 1 a 100. Cada prisioneiro pode abrir no máximo 50 caixas. Para que todos sobrevivam, todos devem encontrar o número correspondente ao seu próprio número nessas 50 tentativas. Se a escolha fosse aleatória, a chance de sucesso coletivo seria quase zero.
Existe uma solução teórica conhecida (seguir o ciclo), que aumenta a chance de sobrevivência do grupo para aproximadamente 31%.
O grupo aplicou um Algoritmo Genético (AG) para tentar otimizar a ordenação das 100 caixas. O Algoritmo Genético busca uma estratégia de ordenação com maior taxa de sucesso, utilizando etapas como geração inicial aleatória, uma função de *fitness* para avaliar a qualidade, seleção, cruzamento e mutação.

Explicação detalhada no readme do projeto: https://github.com/arthur-vasco7/genAlg-100-prisoners?tab=readme-ov-file#algoritmo-gen%C3%A9tico-problema-dos-100-prisioneiros

Tarefa:

A tarefa principal era implementar e refatorar um Algoritmo Genético capaz de gerar uma ordenação das 100 caixas na qual, utilizando a lógica circular, todos os prisioneiros conseguissem encontrar seus respectivos números.
As responsabilidades foram divididas, focando nos componentes chave do AG:

1. **Geração e Mutação:** Criação de permutações iniciais e garantia de que a mutação preservasse a validade dessas permutações.
2. **Fitness e Seleção:** Desenvolvimento da função de aptidão (*fitness*) para quantificar quantos prisioneiros sobreviveram em uma dada estratégia, e a seleção dos melhores indivíduos para gerar a próxima população.
3. **Estrutura e Refatoração:** Consolidar o código em uma estrutura de classe única, tornando-o flexível e configurável, e implementar mecanismos de melhoria, como o elitismo.
4. **Testes:** Criar testes para validar o ambiente, a mutação e a função de *fitness*, lidando com a natureza probabilística da solução.

Processo de Pensamento:

O processo de pensamento foi dominado pela necessidade de aprender e adaptar. Muitos membros do grupo não tinham conhecimento prévio sobre Algoritmos Genéticos, o que exigiu alguns dias de pesquisa (vídeos, livros) para entender a teoria e como aplicá-la ao problema.
A abordagem inicial de um dos desenvolvedores (Ruan) para entender a implementação do AG foi se basear na lógica de um problema binário mais simples (maximizar a soma de bits).

A adaptação da lógica foi crucial, diferente dos problemas clássicos onde cada candidato tem uma "nota" de desempenho, neste problema, o sucesso é totalmente dependente e exige que todos os microproblemas trabalhem em conjunto. O *fitness* não poderia ser baseado em uma pontuação individual, mas sim na contagem de quantos prisioneiros conseguiram encontrar seus números na ordenação da caixa.
Durante a refatoração, a decisão foi encapsular toda a lógica em uma única classe, abandonando a estrutura anterior de múltiplos arquivos com funções isoladas (como uma função de mutação em um arquivo próprio). Essa refatoração também introduziu flexibilidade, permitindo configurar o número de prisioneiros, o tamanho da população, o número de gerações, e as taxas de mutação e cruzamento (*crossover*) como atributos.
Para a mutação, o desenvolvedor (Rafael) pensou na restrição de permutações. Como a sequência de números não pode se repetir, a mutação simples não funcionaria. A solução adotada foi a *swap mutation*, que escolhe dois índices aleatórios e troca seus valores, garantindo que a permutação permaneça válida.
Na seleção dos pais (*select parents*), o raciocínio foi que, para convergir rapidamente, era necessário escolher os melhores indivíduos da população atual. A lógica é que, quanto melhor for o desempenho dos pais, mais rápido o algoritmo chegará à solução ótima de 100 acertos.

Conclusões:

A solução utilizando Algoritmo Genético foi implementada com sucesso, apesar da curva de aprendizado íngreme para a maioria dos membros do grupo. A principal dificuldade foi a adaptação conceitual do AG para um problema que exige sucesso coletivo. A refatoração e a implementação de mecanismos de controle (como elitismo e taxas de cruzamento flexíveis) garantiram que o algoritmo conseguisse buscar a solução ótima de forma eficiente. O código é considerado funcional e atende aos requisitos, encontrando uma solução possível usando a lógica circular. Os testes validaram que as permutações são mantidas válidas e que a lógica de seleção funciona, embora a dificuldade em testar a aleatoriedade tenha sido uma limitação notada.

10 Itens mais Importantes Observados nas Entrevistas:

1. **Compreensão parcial, mas suficiente, do problema dos 100 prisioneiros por todos os entrevistados;**

2. **Domínio firme do código apenas pelos desenvolvedores diretamente envolvidos na escrita das funções centrais;**

3. **Uso de IA (LLM) para gerar testes;**

4. **Os testadores tiveram dificuldade em compreender completamente o algoritmo genético;**

5. **Mudanças entre branchs resultaram em testes quebrados;**

6. **A função de fitness foi ponto de maior dúvida entre alguns membros;**

7. **Participantes demonstraram maior segurança ao explicar sua própria parte do código, mas pouca visão global;**

8. **Boa capacidade de identificar erros ao vivo durante a entrevista;**

9. **Percepção de que parte do código poderia ser reorganizada ou modularizada;**

10. **A estratégia de desenvolvimento colaborativo funcionou, mas gerou assimetrias de entendimento;**

Resumo das entrevistas:

Ruan Rodrigues (Função: Desenvolvedor/Refatoração):
Foco e Contribuição: Ruan Rodrigues foi o principal responsável pela refatoração do código, consolidando a lógica de quatro arquivos separados em uma única classe. Ele estruturou a classe para ser flexível, permitindo configurar parâmetros importantes do Algoritmo Genético (AG), como o número de prisioneiros, tamanho da população, número de gerações, e as taxas de mutação e cruzamento (*crossover*).
Processo de Pensamento e Dificuldades: Ele se baseou na lógica de um problema de AG mais simples (maximizar a soma de bits binários) para modelar o processo neste projeto. Ruan identificou a necessidade crítica de implementar o elitismo, pois percebeu que o algoritmo, após encontrar uma solução ótima (100 acertos), frequentemente a perdia devido à aleatoriedade das operações genéticas. Outra dificuldade técnica foi a necessidade de realizar cópias explícitas no Python (como na linha 59 do *crossover*) para evitar erros de referência e garantir que o elitismo funcionasse corretamente, o que ele notou que pode tornar o código mais lento.
Uso de IA: Ele consultou a Inteligência Artificial principalmente para fins de diagnóstico e aprendizado conceitual, como quando a IA sugeriu o conceito de elitismo para resolver o problema de perda de soluções ótimas.
Conclusão: A refatoração resultou em uma estrutura mais organizada e flexível. Ruan concluiu que aprendeu a modelar a resolução de problemas utilizando o Algoritmo Genético, internalizando o modelo mental de como o *fitness*, a mutação e a seleção devem ser aplicados.

Artur Vasconcelos (Função: Desenvolvedor - Fitness/Seleção):
Foco e Contribuição: Artur foi o responsável, principalmente, pelos métodos de avaliação de aptidão (*fitness*) e seleção da elite. Ele definiu a função de *fitness* para quantificar quantas vezes cada indivíduo teve sucesso em cenários aleatórios.
Processo de Pensamento e Dificuldades: Ele destacou que a principal dificuldade foi adaptar o Algoritmo Genético para o Problema dos 100 Prisioneiros. Nos problemas clássicos de AG, cada

candidato recebe uma nota individual, mas neste caso, a solução exige que "todos micro problemas trabalhem em conjunto para todos darem certos" (dependência mútua). Ele precisou de pesquisa para entender o problema e o AG, pois não tinha conhecimento prévio.
Uso de IA: Consultou a IA para ajudar a conceituar como relacionar o problema ao algoritmo, pois estava "muito difícil" adaptar a contagem de sucessos. No entanto, as respostas da IA eram em texto (não código) e ele precisou adaptar as ideias, já que a IA, em um momento, sugeriu representar a solução como um grafo em vez de uma lista.

Rafael Morais (Função: Desenvolvedor - Mutação/Geração de População):
Foco e Contribuição: Rafael implementou, principalmente, a função de mutação e a geração da população inicial.
Processo de Pensamento e Dificuldades: Ele demonstrou um entendimento aprofundado do problema e da solução teórica, incluindo a probabilidade baixa de sucesso aleatório e a taxa de 31% com a estratégia cíclica. Ele enfatizou a restrição de que a mutação precisava preservar a propriedade de permutação (sem repetição de números). Por isso, adotou a troca de valores entre duas posições aleatórias.
Design e Estabilidade: Rafael notou que os métodos de mutação e geração de população são simples, têm baixa dependência de outros métodos e se mantiveram inalterados desde a primeira implementação. Ele não vê custos ou efeitos colaterais significativos em seus métodos.
Uso de IA: Usou a IA apenas para gerar os comentários e documentação do código.
Ambiguidades: Sentiu que a especificação inicial do problema no documento era ambígua, o que exigiu pesquisa adicional para entender o problema clássico e a necessidade de usar o AG.

Carlos Artur (Função: Teste):
Foco e Contribuição: Carlos foi responsável pela criação de testes para validar o ambiente, o *fitness* e a mutação.
Processo de Pensamento e Dificuldades: A principal dificuldade que encontrou foi testar a parte randômica do código, pois não havia "muito controle" sobre a aleatoriedade. Sua solução para isso foi rodar os testes "1000 vezes para ficar mais próximo" do comportamento esperado. Ele criou testes para verificar se o genoma sofreu mutação e se as caixas geradas estavam no *range* correto.
Uso de IA e Ferramentas: Utilizou o ChatGPT principalmente para **configurar o ambiente de testes** (pytest, cobertura de código) e para tirar dúvidas sobre como configurar testes para coletar erros (assertions). Ele teve que adaptar manualmente algumas das configurações fornecidas pela IA.
Aprendizado: Relatou ter aprendido a trabalhar com a questão da aleatoriedade em testes e a usar ferramentas como pytest.

Helder Chaves (Função: Teste):
Foco e Contribuição: Hélder foi responsável pelos testes, e utilizou uma LLM (Large Language Model) para gerar os testes da classe principal, alcançando uma cobertura de quase 100%.

Processo de Pensamento e Dificuldades: Ele relatou que sua experiência com Python estava "enferrujada" e usou a LLM para auxiliar na criação do código de teste. Ele viu a alta cobertura como uma métrica de confiança nos testes gerados. Ele não teve que adaptar ou corrigir os testes gerados pela IA, exceto por ter que pedir casos adicionais para levantar erros específicos.
Visão do Testador: Ele não tinha conhecimento prévio de Algoritmos Genéticos. Como testador, não percebeu grandes desafios em testar o AG em comparação com o software tradicional; a chave foi entender o problema e a solução. Ao analisar o código, notou que a função de *fitness* era complexa, mas não encontrou problemas de ambiguidade ou fragilidade na solução em si, apenas questões de estilização.

Link entrevistas (Entrevistas Think Aloud)

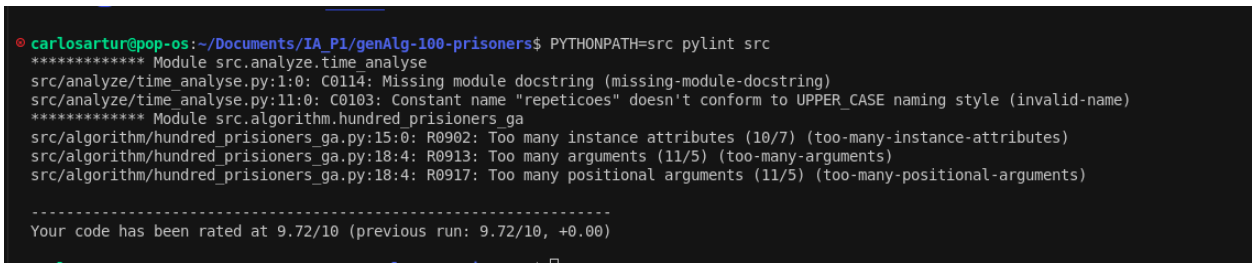
TESTES ESTÁTICOS DE CÓDIGO

Ao executar a aplicação do Pylin no código fonte foram obtidos os seguintes resultados na planilha abaixo:

Código	Descrição
C0301	Line too long
C0303	Trailing whitespace
C0305	Trailing newlines
C0114	Missing module docstring
R0902	Too many instance attributes
R0913	Too many arguments
R0917	Too many positional arguments

A partir desse resultado inicial, foram observados os **C0300**, que correspondem a problemas de formatação e estilo do código e vão contra as convenções recomendadas pelo Pylint. Esses pontos incluíam linhas longas, espaços em excesso, quebras de linha incorretas e outros ajustes de formatação, os quais foram devidamente corrigidos. Também foi adicionada a docstring ausente para atender ao código **C0114**, garantindo que o módulo possua uma descrição adequada de sua finalidade e funcionamento.

Por outro lado, os avisos das categorias **R0900** o **R0902**, **R0913** e **R0917** não foram solucionados. Esses códigos estão relacionados à estrutura e arquitetura do código, como complexidade de classe, número excessivo de atributos, quantidade elevada de argumentos em funções e uso excessivo de argumentos posicionais. A correção desses problemas exigiria uma refatoração mais profunda, que implicaria mudanças significativas na lógica implementada e na forma como os componentes do código se relacionam. Como esse tipo de modificação poderia impactar funcionalidades já estabelecidas e demandar uma revisão estrutural abrangente, optou-se por não aplicar essas alterações.



MÉTRICAS ESTÁTICAS DE FUNCIONAMENTO DO CÓDIGO

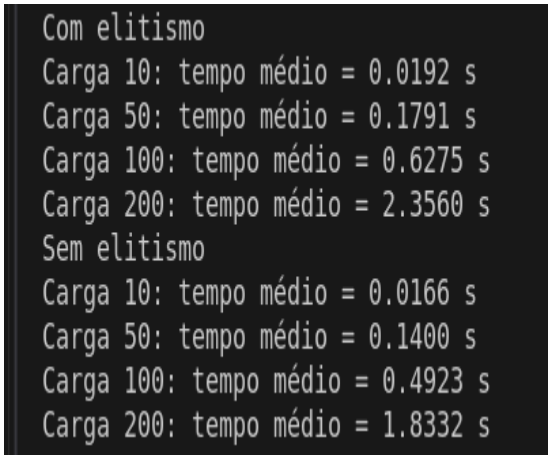
Complexidade de algoritmo:

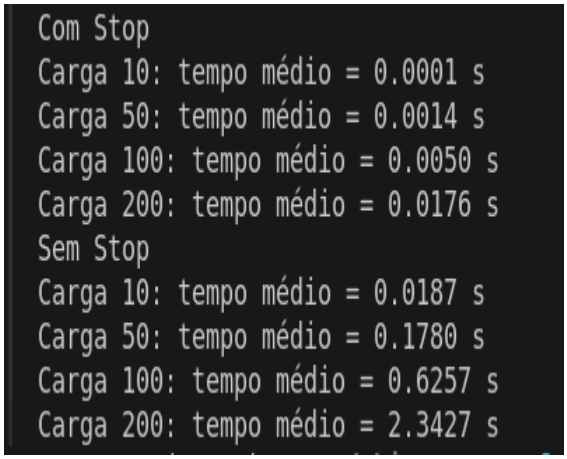
Para analisar a complexidade temporal do algoritmo, chamaremos de “n” o tamanho do genoma (lista de inteiros, nesse problema). Abstraindo custos de tempo insignificantes em comparação à magnitude dos processos mais demorados, ficamos com:

- Inicializar a população: O(n) por indivíduo para gerar a permutação, significando um custo geral de O(n * <POPULAÇÃO>);
- Avaliar fitness do indivíduo: Envolve duas iterações de laços for aninhados, custando $O(n^2)$ por genoma. Para o conjunto todo, temos $O(n^2 * <POPULAÇÃO>)$;
- Para avaliar a fitness do objetivo, apenas um loop é utilizado, gerando custo de O(n). Para a população completa, temos O(n * <POPULAÇÃO>);
- Na seleção dos pais, temos uma iteração de laço for, com O(n), assim como um custo significativamente menor de O(log n), em razão da otimização da função sorted de python;
- Para o cruzamento de indivíduos, temos um loop for, contendo uma verificação de pertencimento, fazendo com que seu desempenho piore para $O(n^2)$;
- No loop principal, temos a ação de avaliar fitness e cruzamento como principais tarefas custosas, ou seja:
  $O(n^2 * <POPULAÇÃO>) + O(n^2 * <POPULAÇÃO>) = 2 * O(n^2 + <POPULAÇÃO>)$.
  Como o custo quadrático é consideravelmente mais agudo, abstraímos para:
  $O(n^2 + <POPULAÇÃO>)$.

No custo espacial, a maior estrutura armazenada é a população. Logo, como cada indivíduo é um vetor de elementos, a complexidade se torna O(n * <POPULAÇÃO>).

Tempo de processamento:





Como podemos observar, os resultados mostram diferenças significativas no desempenho do algoritmo, dependendo das estratégias utilizadas. Quando o elitismo está ativado, os tempos médios de execução aumentam um pouco em todas as cargas, o que reflete o esforço extra para preservar os melhores indivíduos ao longo das gerações. Em contrapartida, a falta de elitismo

diminui um pouco o tempo, porém pode afetar a qualidade e a estabilidade das soluções. A estratégia de **Stop**, que interrompe a execução ao encontrar uma solução ótima, demonstra uma melhoria de desempenho significativa, com tempos praticamente insignificantes mesmo em cargas maiores. Por outro lado, sem o Stop, os tempos permanecem semelhantes aos observados com ou sem elitismo. Isso mostra que o critério de parada antecipada é altamente eficaz para agilizar a execução sem afetar a solução final.