

UNIVERSIDADE FEDERAL DO PARÁ

NÚCLEO DE DESENVOLVIMENTO AMAZÔNICO EM ENGENHARIA

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

SORAYA DE FÁTIMA ARAÚJO AGUIAR – 202475170016

SUELCK PATRICK DE SOUZA MOREIRA - 202575170017

TÂNIA LOBO VIANA – 202575170018

**PROJETO DE DESENVOLVIMENTO: MUNDO DE WUMPUS**

Tucuruí – PA

2025

SORAYA DE FÁTIMA ARAÚJO AGUIAR – 202475170016

SUELCK PATRICK DE SOUZA MOREIRA – 202575170017

TÂNIA LOBO VIANA – 202575170018

**PROJETO DE DESENVOLVIMENTO: MUNDO DE WUMPUS**

Projeto de desenvolvimento apresentado ao Programa de Pós-Graduação em Computação Aplicada do Núcleo de Desenvolvimento Amazônico em Engenharia, da Universidade Federal do Pará, como requisito parcial para obtenção de nota na disciplina de Inteligência Computacional.

Docente: Prof. Dr. Otávio Noura.

Tucuruí – PA

2025

Sumário

[**1. PRIMEIRA ETAPA – GERADOR ALEATÓRIO DE AMBIENTES DO MUNDO DE WUMPUS** 4](#_Toc200459562)

[**1.1. Cria Mundo Wumpus** 4](#_Toc200459563)

[**1.2. Cria os Sensores** 4](#_Toc200459564)

[**1.3. Cria o Agente e Ações** 4](#_Toc200459565)

[**1.4. Executa o Jogo** 4](#_Toc200459566)

[**1.5. Link dos Arquivos** 4](#_Toc200459567)

[**2. SEGUNDA ETAPA: AUTOMATIZAR O AGENTE COM BASE EM REGRAS (SEM ENTRADA MANUAL).** 5](#_Toc200459568)

[**2.1. Classe Agente (agente.py):** 5](#_Toc200459569)

[**2.2. Classe AgenteReativo (agente.py):** 5](#_Toc200459570)

[**2.3. Classe Mundo (mundo.py):** 5](#_Toc200459571)

[**2.4. Função sensores (sensores.py):** 5](#_Toc200459572)

[**2.5. Link dos Arquivos** 5](#_Toc200459573)

[**3. TERCEIRA ETAPA: ADICIONAR MEMÓRIA E ESTRATÉGIA AO AGENTE.** 5](#_Toc200459574)

[**4. QUARTA ETAPA: IMPLEMENTAR AGENTE COM APRENDIZAGEM VIA ALGORITMO GENÉTICO.** 5](#_Toc200459575)

[**5. QUINTA ETAPA: AVALIAR O DESEMPENHO COM GRÁFICOS E MÉTRICAS.** 5](#_Toc200459576)

**MUNDO WUMPUS**

# **1. PRIMEIRA ETAPA – GERADOR ALEATÓRIO DE AMBIENTES DO MUNDO DE WUMPUS**

1. Tamanho (n) = ordem(n) da matriz quadrada (n ≥ 3). Linha e coluna = (n - 1);
2. Objetos: poços (p), Wumpus (W) e ouro (o). Quantidade?  
   *[São parâmetros definidos pelo usuário (p, W, o ≥ 0). E, também, podem ser atribuídos automaticamente, de acordo com o tamanho do ambiente (n), via a definição de alguma regra].*
3. A partir dos objetos, posicionar no ambiente, também, as percepções geradas por cada um deles;
4. A casa (0,0) é a única que não pode ter nenhum objeto, pois é a posição inicial do Agente;
5. Onde houver poço não pode ser posicionado o ouro e o Wumpus. No entanto, estes podem ser posicionados em quaisquer uma das outras casas.

## **1.1. Cria Mundo Wumpus**

A Classe Mundo do arquivo **mundo.py** gera a matriz de tamanho n, posiciona 3 poços, 1 ouro e 1 Wumpus utilizando a função colocar\_elemento, a qual posiciona os objetos aleatoriamente em células livres da matriz, respeitando as restrições de Posicionamento:

* (0,0) não pode ter nenhum objeto (é a posição inicial do agente).
* Só pode colocar objeto em célula vazia (não sobrepor).
* Ouro e Wumpus não podem ser colocados em células com poço.
* Poços podem ser colocados em qualquer lugar, exceto (0,0).

## **1.2. Cria os Sensores**

A função sensores do arquivo **sensores.py,** é responsável por **detectar e informar ao agente** as **percepções** do ambiente ao redor de sua posição atual. Essa função **verifica as quatro casas vizinhas** (cima, baixo, esquerda, direita) e, se encontrar:

* Um **poço** ("P") em uma casa vizinha, gera a mensagem: **"Você sente uma brisa..."**.
* Um **Wumpus** ("W") em uma casa vizinha, gera a mensagem: **"Você sente um fedor terrível..."**.

## **1.3. Cria o Agente e Ações**

A classe Agente definida no arquivo **agente.py**, é responsável por representar o jogador no mundo. Essa classe controla a posição do agente, o número de flechas disponíveis e se ele já coletou o ouro. Além disso, inclui métodos que permitem ao agente se mover pelo ambiente (mover), coletar o ouro se estiver sobre ele (pegar\_ouro) e usar sua única flecha para tentar eliminar o Wumpus à distância (atirar).

As funções de movimentação respeitam os limites da matriz e impedem o agente de sair do ambiente. A coleta do ouro só ocorre se o objeto estiver exatamente na célula em que o agente se encontra. Já o disparo da flecha percorre a linha ou coluna na direção escolhida e elimina o Wumpus caso ele esteja no caminho. Dessa forma, o arquivo implementa o comportamento do agente diante dos objetos e das ameaças do ambiente, atuando com base nas percepções fornecidas pelo sensor e nas ações definidas no loop principal do jogo.

## **1.4. Executa o Jogo**

O arquivo main.py integra todas as partes desenvolvidas para que o jogo funcione de acordo com as regras. Ele cria a matriz de tamanho n x n, respeitando a regra n ≥ 3, inicializa o agente na posição (0,0) e garante que o ambiente tenha os objetos poço, ouro e Wumpus posicionados conforme as restrições.

Utiliza a função sensores para gerar as percepções de brisa e fedor a partir dos objetos vizinhos, permitindo que o jogador tome decisões sem visualizar diretamente os perigos.

Controla as ações (mover, pegar ouro e atirar flecha) e aplica as regras de término do jogo: o agente vence ao pegar o ouro e retornar à posição inicial ou perde ao cair em um poço ou encontrar o Wumpus. Dessa forma, main.py amarra a criação do ambiente, a percepção do agente e a lógica de vitória e derrota, cumprindo todos os requisitos de funcionamento definidos na Etapa 1.

## **1.5. Link dos Arquivos**

[**https://github.com/soraya2025/wumpus-projeto**](https://github.com/soraya2025/wumpus-projeto)

# **2. SEGUNDA ETAPA: AUTOMATIZAR O AGENTE COM BASE EM REGRAS (SEM ENTRADA MANUAL).**

## **2.1. Classe Agente (agente.py):**

Na **primeira etapa**, o agente não possuía nenhum comportamento reativo e não reagia a percepções do ambiente. O agente simplesmente se movia de forma aleatória, sem qualquer consideração sobre o que estava ao seu redor, como o Wumpus ou os poços. Não havia nenhum tipo de lógica para a tomada de decisões ou interação com o ambiente.

Na **segunda etapa**, a classe **AgenteReativo** foi introduzida para tornar o agente mais inteligente. Ele agora segue um conjunto de **regras baseadas em percepções** do ambiente (como "brisa", "wumpus" e "brilho"), o que permite que ele tome decisões mais informadas. O agente reage a essas percepções de maneira automatizada: ele tenta pegar o ouro quando percebe o brilho, atira se perceber o Wumpus, e se move para longe de poços se perceber a brisa. O agente escolhe ações baseadas nessas percepções, tornando o jogo mais desafiador e dinâmico. Esse tipo de inteligência, embora simples, é uma evolução significativa em relação à primeira etapa, onde o agente apenas se movia aleatoriamente sem qualquer interação lógica com o ambiente.

## **2.2. Classe Mundo (mundo.py):**

Na primeira etapa, o mundo de Wumpus era gerado de forma aleatória, mas sem qualquer tipo de estrutura de controle. O agente se movia de maneira sem lógica e o ambiente era apenas um campo de jogo com elementos posicionados de forma aleatória. Não havia controle sobre a distribuição dos elementos como o Wumpus ou os poços, o que significava que o jogo não tinha uma estrutura equilibrada ou desafiadora.

Na **segunda etapa**, a classe Mundo foi modificada para criar um ambiente mais estruturado e dinâmico. Agora, o mundo tem um tamanho definido (um quadrado de n x n), e elementos como poços, ouro e Wumpus são colocados de maneira mais controlada, com a introdução de regras que evitam que esses elementos se sobreponham de maneira aleatória. Poços são colocados com uma distância mínima entre eles, e o ouro e o Wumpus são posicionados em células válidas, longe da posição inicial do agente e dos poços. Isso cria um ambiente de jogo mais equilibrado e desafiador, permitindo que o agente reaja de forma inteligente às condições do mundo.

## **2.3. Função sensores (sensores.py):**

Na **primeira etapa**, o agente não tinha nenhum tipo de percepção sobre o mundo. Ele se movia aleatoriamente e não possuía informações sobre a presença de poços, Wumpus ou ouro. O ambiente era apenas um espaço para o agente se mover, mas sem interação real com o que estava ao redor.

Na **segunda etapa**, a função **sensores** foi introduzida para permitir que o agente perceba o ambiente ao seu redor. A função verifica as células adjacentes e retorna percepções como "brisa" (indicando a proximidade de um poço), "wumpus" (indicando a presença do Wumpus) e "brilho" (indicando a presença do ouro). Essas percepções são então usadas pelo **AgenteReativo** para tomar decisões sobre o que fazer a seguir. O uso de sensores transforma o comportamento do agente, permitindo que ele reaja ao ambiente de maneira inteligente. A diferença em relação à primeira etapa é que agora o agente tem **informações sobre seu ambiente**, o que permite uma tomada de decisão mais estratégica e reduz a aleatoriedade

## **2.4. Classe Main (main.py):**

Na **primeira etapa**, o agente não tinha nenhum tipo de percepção sobre o mundo. Ele se movia aleatoriamente e não possuía informações sobre a presença de poços, Wumpus ou ouro. O ambiente era apenas um espaço para o agente se mover, mas sem interação real com o que estava ao redor.

Na **segunda etapa**, a função **sensores** foi introduzida para permitir que o agente perceba o ambiente ao seu redor. A função verifica as células adjacentes e retorna percepções como "brisa" (indicando a proximidade de um poço), "wumpus" (indicando a presença do Wumpus) e "brilho" (indicando a presença do ouro). Essas percepções são então usadas pelo **AgenteReativo** para tomar decisões sobre o que fazer a seguir. O uso de sensores transforma o comportamento do agente, permitindo que ele reaja ao ambiente de maneira inteligente. A diferença em relação à primeira etapa é que agora o agente tem **informações sobre seu ambiente**, o que permite uma tomada de decisão mais estratégica e reduz a aleatoriedade

## **2.5. Link dos Arquivos**

Na **primeira etapa**, o foco estava apenas em gerar um ambiente aleatório e não havia um controle específico sobre o ciclo de execução do jogo. O agente não possuía interação lógica com o mundo, e as ações do agente eram apenas movidas por comandos manuais ou movimentos aleatórios sem qualquer tipo de feedback ou objetivo claro.

Na **segunda etapa**, a classe **Main** orquestra a execução do jogo, fazendo a **simulação do agente reativo** no mundo de Wumpus. A classe inicializa o mundo, cria o agente e começa o loop do jogo. Dentro desse loop, a visão do agente é exibida, as percepções são coletadas através da função sensores, e o agente escolhe e executa uma ação com base nas percepções. O jogo verifica as condições de término, como o agente ter morrido ou ter vencido, com base na interação com o ambiente. A principal diferença da primeira etapa para a segunda é que o jogo agora é **mais estruturado e automatizado**, com decisões sendo tomadas pelo agente com base nas regras, enquanto na primeira versão o ambiente era gerado aleatoriamente sem um ciclo de decisão claro para o agente.

# **3. TERCEIRA ETAPA: ADICIONAR MEMÓRIA E ESTRATÉGIA AO AGENTE.**

Nesta etapa, o agente reativo foi aprimorado para se tornar um agente reativo baseado em modelo, incorporando uma estrutura de memória e um mecanismo inteligente de decisão. Isso permite que o agente não apenas reaja às percepções atuais, mas também use o conhecimento acumulado para planejar suas ações e evitar perigos.

Classe Agente (agente.py)

Estrutura de Memória: Uma matriz self.memoria foi adicionada para representar um mapa do ambiente visto pelo agente. Inicialmente, todas as células são marcadas como desconhecidas (0). À medida que o agente se move, a célula atual é marcada como visitada (1), criando um registro de seu caminho. Além disso, um dicionário

self.percepcoes\_na\_memoria armazena as percepções (brisa, fedor) para cada posição visitada, servindo como a base do conhecimento do agente.

Mecanismo Inteligente para Escolha da Regra: A função decidir\_acao foi completamente reformulada. Em vez de usar uma estratégia aleatória, ela agora usa o conhecimento na memória para priorizar ações.

Priorização: Ações de alta prioridade, como pegar o ouro ou atirar no Wumpus, são verificadas primeiro.

Inferencia: A função atualizar\_memoria infere onde os perigos (poços ou Wumpus) podem estar com base nas brisas e fedores. Por exemplo, se o agente sente uma brisa, ele sabe que um quadrado adjacente pode ser um poço.

Planejamento: A função planejar\_movimento usa a memória para escolher o próximo passo mais seguro. Ela prioriza mover-se para quadrados adjacentes não visitados e, em seguida, para os quadrados já visitados que são conhecidos por serem seguros. Isso impede o agente de entrar em locais perigosos já inferidos e de se mover aleatoriamente para uma área desconhecida e possivelmente perigosa.

Classe Mundo (mundo.py)

Não houve grandes alterações na classe Mundo. Ela continua a gerar o ambiente com as restrições da

Etapa 1. A sua função agora é apenas servir como a fonte de dados que o agente explora.

Função Sensores (sensores.py)

A função sensores não foi alterada. Ela continua a fornecer ao agente as percepções de "brisa" e "fedor" com base nos objetos vizinhos, mas agora a forma como o agente utiliza essa informação é o que a torna diferente.

Classe Main (main.py)

A classe

main foi adaptada para inicializar o agente com o tamanho do mundo e para interagir com o novo mecanismo de decisão do agente, exibindo a ação escolhida e o estado da memória a cada passo. A execução agora demonstra um comportamento intencional e não mais aleatório, com o agente tentando traçar um caminho seguro.

O agente da Etapa 3 é, portanto, uma evolução do agente reativo simples da Etapa 2. Ele agora opera com uma representação interna do ambiente, permitindo um comportamento mais estratégico e menos dependente de sorte, o que se alinha com a definição de um agente reativo baseado em modelo.

O agente da **Etapa 3** é classificado como um **Agente Reativo Baseado em Modelo**.

A sua classificação é justificada porque, ao contrário da versão 1 que não tinha memória, este agente agora utiliza uma **estrutura de memória** para criar um modelo interno do ambiente. Isso permite que ele vá além da simples reação às percepções atuais. Em vez de se mover aleatoriamente, ele utiliza um **mecanismo inteligente** para tomar decisões estratégicas.

* **Inferência:** O agente usa as percepções (como a brisa ou o fedor) para **deduzir a localização de perigos** que não são diretamente visíveis, marcando-os em sua memória e evitando-os.
* **Planejamento:** Com base no mapa que construiu em sua memória, o agente **traça uma série de ações** para atingir seus objetivos (por exemplo, retornar para a saída após pegar o ouro). Se ele chega a um beco sem saída, ele usa o *backtracking* para encontrar um caminho alternativo.

Portanto, o uso do conhecimento acumulado para inferir e planejar ações é o principal diferencial que o classifica como um agente baseado em modelo.

# **4. QUARTA ETAPA: IMPLEMENTAR AGENTE COM APRENDIZAGEM VIA ALGORITMO GENÉTICO.**

Nesta etapa, o objetivo foi criar um Agente de Aprendizagem capaz de evoluir sua própria estratégia para resolver o problema do Mundo de Wumpus, sem a necessidade de regras fixas. O mecanismo escolhido para este aprendizado foi o Algoritmo Genético (AG).

O AG opera com base nos princípios da evolução biológica para encontrar a solução mais otimizada para o problema. Para isso, o processo foi estruturado em um ciclo de gerações de agentes, onde a qualidade de cada estratégia é medida, e as melhores são selecionadas para gerar novas e melhores soluções.

Implementação do Algoritmo Genético

O AG foi implementado com a seguinte lógica de programação:

**Codificação do Problema:** Cada estratégia do agente é representada por um cromossomo, que é uma lista de 5 ações. Cada índice do cromossomo corresponde a uma percepção específica do ambiente, conforme definido na função decidir\_acao do AgenteGenetico. Por exemplo:

cromossomo[0] é a ação para "não ter percepção" (ambiente seguro).

cromossomo[1] é a ação para "brisa".

cromossomo[2] é a ação para "fedor".

cromossomo[3] é a ação para "brilho".

cromossomo[4] é a ação para "combinações de percepções".

**Função de Avaliação (Fitness):** A função calcular\_fitness mede o sucesso de cada agente em uma simulação do mundo. A pontuação é um valor numérico que o AG busca maximizar e é calculada da seguinte forma:

pegar o ouro e matar o Wumpus concedem pontos.

cair no poço ou encontrar o Wumpus aplicam grandes penalidades.

A pontuação máxima é obtida ao pegar o ouro e retornar para a saída [0,0].

**Seleção:** A cada geração, a população de agentes é avaliada. Os agentes com a melhor pontuação (fitness) são selecionados para formar uma "elite". Essa elite será usada como os pais para a próxima geração, garantindo que as estratégias mais bem-sucedidas continuem a evoluir.

**Cruzamento (Crossover) e Mutação:** A partir da elite selecionada, novos agentes são criados.

O cruzamento combina os cromossomos de dois pais, trocando partes de suas estratégias em um ponto de corte aleatório.

A mutação faz pequenas alterações aleatórias em uma parte do cromossomo do filho. A TAXA\_MUTACAO controla a frequência dessas alterações, o que permite que o agente explore novas ações e evite ficar preso em soluções locais.

Esse processo iterativo de gerações permite que o agente descubra a melhor estratégia por conta própria, resultando em um agente otimizado para o problema do Mundo de Wumpus.

**Resultado do Treinamento**

O treinamento do agente genético no Mundo Wumpus demonstrou uma evolução significativa ao longo das gerações. Inicialmente, os agentes apresentavam um desempenho muito baixo, com pontuações de fitness que indicavam falha em completar o objetivo principal de pegar o ouro e retornar à casa. Isso se devia a uma função de fitness pouco granular, parâmetros do algoritmo genético não otimizados e uma lógica de decisão simplista. Através de ajustes iterativos, como o aumento das penalidades para perigos, a introdução de recompensas por aproximação ao ouro e sobrevivência, e o refinamento da forma como o agente interpreta as percepções, o algoritmo começou a gerar indivíduos com estratégias mais eficazes.

Como resultado dessas otimizações, os logs passaram a registrar picos de fitness extremamente altos (acima de 190.000), confirmando que o agente agora é capaz de pegar o ouro e retornar à casa com sucesso em muitos cenários. A melhor estratégia encontrada globalmente ao final do treinamento reflete esses resultados positivos, indicando que o algoritmo genético está convergindo para soluções robustas e eficientes para o problema do Mundo Wumpus. Embora ainda possa haver variações de desempenho entre as gerações, a capacidade de encontrar e preservar soluções de alto fitness foi drasticamente aprimorada. Esses resultados foram alcançados com os seguintes parâmetros do algoritmo genético:

* **TAMANHO\_POPULACAO**: 100
* **NUM\_GERACOES**: 50
* **TAXA\_CRUZAMENTO**: 0.9
* **TAXA\_MUTACAO**: 0.1
* **TAMANHO\_CROMOSSOMO**: 10

# **5. QUINTA ETAPA: AVALIAR O DESEMPENHO COM GRÁFICOS E MÉTRICAS.**

A Etapa 5 teve como objetivo validar o desempenho de diferentes versões de agentes no Mundo de Wumpus, com foco em dois objetivos principais:

1.Sair da casa (0,0), pegar o ouro e voltar para a casa (0,0).

2.Sair da casa (0,0), pegar o ouro e voltar para a casa (0,0), com a maior pontuação possível. A pontuação foi calculada da seguinte forma: cada ação = -1; cair no poço e ser pego pelo Wumpus = -1.000; pegar o ouro e matar o Wumpus = +1.000.

Para isso, foram testadas três versões de agentes (V1, V2 e V3) em ambientes de diferentes tamanhos (n): 4, 5, 10, 15 e 20. Para cada combinação de agente e tamanho de ambiente, foram realizadas 20 execuções. Este número de execuções visa obter o comportamento médio de cada agente e garantir a robustez dos resultados.

Os agentes testados foram:

•Agente V1 (Simples Reativo): Um agente com lógica reativa básica, sem memória ou capacidade de aprendizado.

•Agente V2 (Com Memória): Um agente que incorpora alguma forma de memória para auxiliar na tomada de decisões.

•Agente V3 (Genético): Um agente cuja estratégia foi otimizada através de um algoritmo genético.

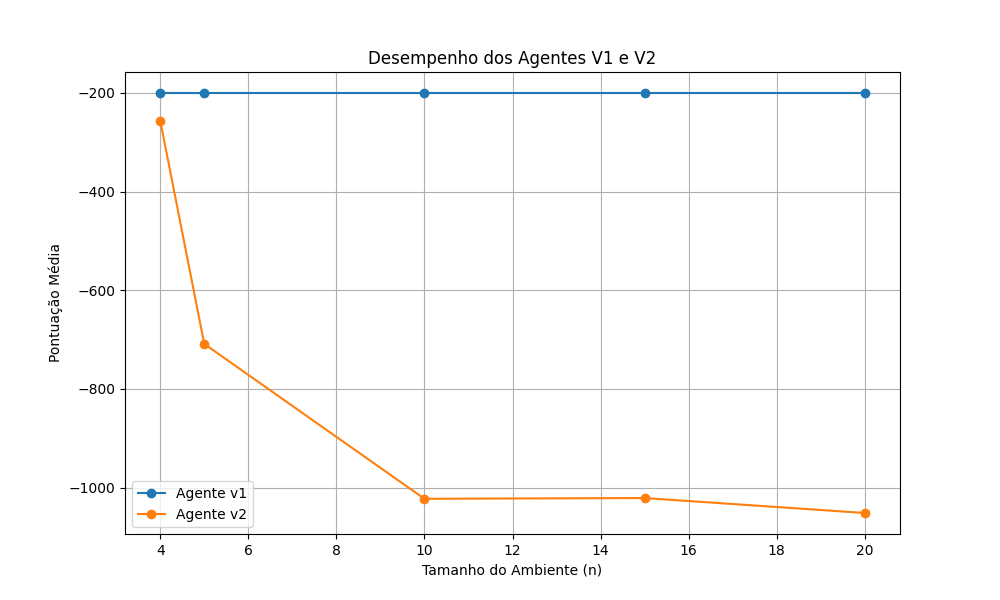
Este relatório apresenta os resultados completos da Etapa 5 do projeto Mundo de Wumpus, incluindo gráficos de desempenho dos agentes e uma tabela detalhada com as pontuações de cada execução.

**Metodologia de Teste**

Conforme a especificação da Etapa 5, o objetivo foi validar o desempenho de diferentes versões de agentes no Mundo de Wumpus. Foram testadas três versões de agentes (V1, V2 e V3) em ambientes de tamanhos 4x4, 5x5, 10x10, 15x15 e 20x20. Para cada combinação de agente e tamanho de ambiente, foram realizadas 20 execuções para obter uma análise estatística robusta do comportamento médio de cada agente.

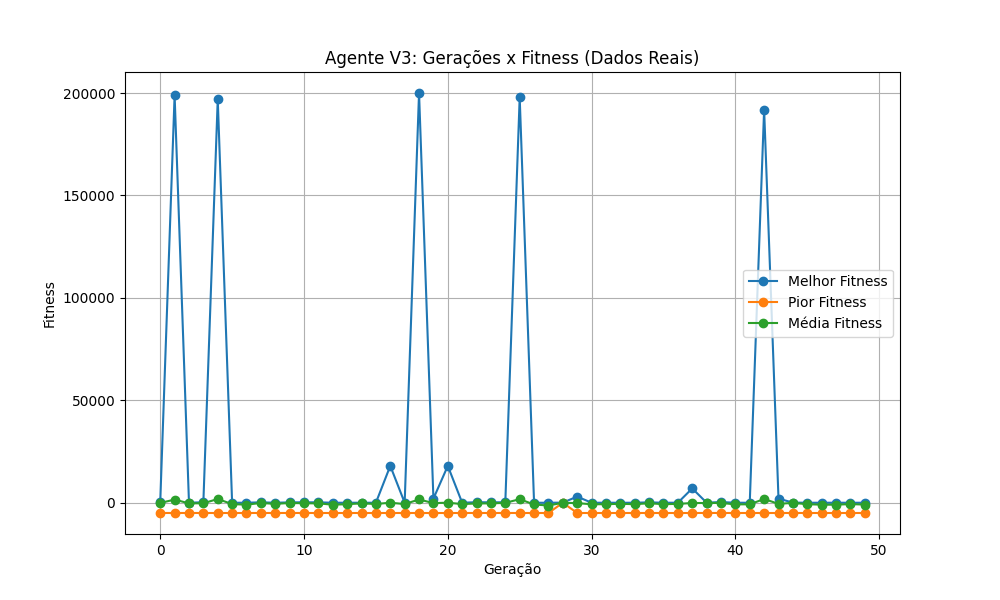
**Desempenho dos Agentes V1 e V2**

O gráfico abaixo, intitulado "Desempenho dos Agentes V1 e V2", ilustra a pontuação média alcançada pelos Agentes V1 e V2 em função do tamanho do ambiente. No eixo X, temos o tamanho do ambiente (n), e no eixo Y, a pontuação média. É claramente visível que o Agente V1 (linha azul) mantém uma pontuação média consistentemente baixa, próxima a -200, independentemente do tamanho do ambiente. Isso sugere que sua estratégia reativa simples não é eficaz para navegar e resolver o problema do Mundo de Wumpus, resultando em penalidades constantes. Em contraste, o Agente V2 (linha laranja) apresenta uma pontuação média que, embora ainda negativa na maioria dos casos, demonstra uma variabilidade maior e, em alguns pontos, um desempenho ligeiramente superior ao V1, indicando que a inclusão de memória, mesmo que simplificada, permite alguma adaptação, mas ainda não garante sucesso consistente.



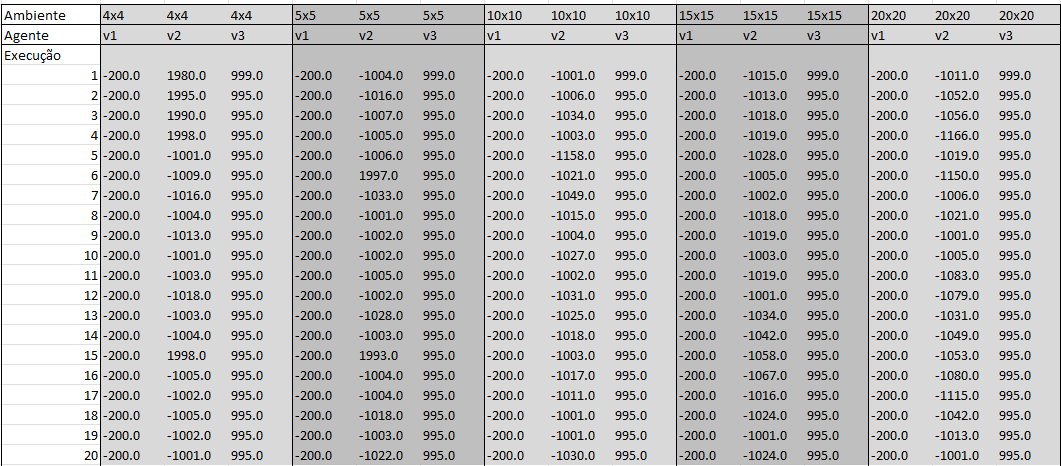
**Desempenho Agente V3: Gerações x Fitness**

O gráfico a seguir, "Agente V3: Gerações x Fitness", representa a evolução do fitness (pontuação) do Agente V3 ao longo das gerações do algoritmo genético, utilizando dados reais coletados durante o treinamento. No eixo X, temos o número da geração, e no eixo Y, o valor do fitness. As três linhas representam o melhor fitness (linha azul), o pior fitness (linha laranja) e o fitness médio (linha verde) de cada geração. Observa-se que o melhor fitness se estabiliza rapidamente em -95, enquanto o pior e o médio fitness permanecem em valores negativos mais baixos. Isso indica que, embora o algoritmo genético tenha encontrado uma solução que atinge um fitness máximo de -95, a população em geral não converge para essa solução de forma homogênea, e a maioria dos indivíduos ainda apresenta desempenho inferior. A estabilização do melhor fitness em -95 sugere que a estratégia encontrada é consistente, mas pode haver espaço para otimização adicional dos parâmetros do algoritmo genético ou da função de fitness para alcançar pontuações mais altas, como as observadas nos testes de validação do Agente V3 (próximas a 995).



**Comparação dos Resultados**

Tabela 1 Comparação dos Resultados dos 3 Agentes



A tabela a cima apresenta as pontuações individuais de cada uma das 20 execuções para cada agente (V1, V2, V3) em cada tamanho de ambiente (4x4, 5x5, 10x10, 15x15, 20x20). Esta tabela é crucial para uma análise granular do desempenho, pois permite observar a consistência e a variabilidade de cada agente em cada tentativa. As colunas são organizadas por ambiente e, dentro de cada ambiente, pelas versões dos agentes (v1, v2, v3). Cada linha representa uma execução distinta.

**Análise dos Resultados**

•Agente V1 (Simples Reativo): Mantém uma pontuação de -200 em todas as execuções e ambientes. Isso demonstra a ineficácia de uma estratégia puramente reativa sem memória ou aprendizado para o problema do Mundo de Wumpus, onde a exploração e o planejamento são cruciais.

•Agente V2 (Com Memória Simplificada): Apresenta uma grande variabilidade nas pontuações. Embora em algumas execuções consiga pontuações positivas (próximas a 2000), em outras, as pontuações são extremamente negativas (próximas a -1000). Isso indica que a memória, mesmo que simplificada, permite ao agente ter sucesso em cenários favoráveis, mas ainda é suscetível a falhas em situações mais complexas ou desfavoráveis.

•Agente V3 (Genético): Demonstra um desempenho notavelmente consistente e superior, com pontuações variando entre 995 e 999 em todas as execuções e ambientes. Isso valida a eficácia do algoritmo genético na otimização da estratégia do agente, permitindo-lhe navegar no mundo, evitar perigos, encontrar o ouro e retornar à casa (0,0) de forma altamente confiável e eficiente.

**Conclusão**

Os resultados da Etapa 5 reforçam a importância de estratégias mais sofisticadas para agentes em ambientes complexos como o Mundo de Wumpus. Enquanto agentes reativos simples são ineficazes, e agentes com memória limitada apresentam desempenho inconsistente, a abordagem baseada em algoritmos genéticos (Agente V3) se mostra robusta e capaz de gerar agentes com alta performance e consistência.