Relatório da Atividade 02 - Agente Inteligente

Celso Vinícius S. Fernandes[1]

[1]Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Divinópolis – MG – Brasil

[2]Bacharelado em Engenharia de Computação

celso.23@aluno.cefetmg.br

Resumo. *Este trabalho apresenta o desenvolvimento de um agente inteligente aspirador robô para simulação de limpeza de um ambiente matricial, utilizando C++. A aplicação permite aos usuários inserir o tamanho da matriz quadrática desejada e simular o comportamento do agente, gerando um 'output' e um 'relatório' do desempenho do agente em determinados níveis de sujeira do ambiente. Destinado a introdutórios de Inteligência Artificial, a implementação pode facilitar a visualização e compreensão dos conceitos básicos de funcionamento de um agente inteligente.*

1. Introdução

A Implementação, basicamente gira em torno de um agente inteligente para o mundo do aspirador de pó no qual o agente possui um sensor de sujeira, um método de locomoção e limpeza. Considerações:

1. O agente implementado será avaliado em um abiente de matriz NxN, inicialmente 4x4, porém o tamanho do ambiente e deve ser flexível para uso em outras dimensões.
2. O agente não sabe a disposição da sujeira nem mesmo o tamanho do ambiente.
3. O agente é iniciado em um local aleatório a cada experimento.
4. O agente pode se movimentar para direita, esquerda, cima e para baixo.
5. O agente pode armazenar o estado de até 3 movimentos realizados. No entanto, ele não pode armazenar sua localização.
6. Após ser realizada a limpeza, o ambiente não volta a ficar sujo.
7. Se bater na parede o agente deve fazer um movimento pseudo-aleatório. Ele pode usar as informações dos estados anteriores para fazer esse movimento.
8. A disposição do local de sujeira deve ser gerada de forma aleatória antes de iniciar o agente. Porém, as configurações de quantidade serão as seguintes:
    (a) 16 quadrados sujos, 12 quadrados sujos, 8 quadrados sujos e 4 quadrados sujos.
9. Registre a pontuação do Aspirador:
    (a) ganha 3 pontos para cada ambiente limpo;
    (b) perde 1 ponto para cada movimento;
    (c) perde 20 pontos para cada quadrado sujo remanescente após o término da limpeza;
10. O Critério de parada do agente deve ser definido pelo aluno.

Para experimentação, deve-se executar 10 repetições de cada configuração e apresentar o melhor resultado, o pior, a média e o desvio padrão. Assim, a intenção global do problema não é encontrar uma solução ótima, mas sim, tentar deixar o ambiente limpo e reproduzir um agente eficiente.

2. Metodologia

Este projeto foca na criação de um agente "aspirador" que deve operar sob a restrição de uma memória limitada e sem conhecimento prévio de sua localização espacial. O agente inteligente foi projetado para operar em um ambiente simulado, enfrentando desafios de navegação e limpeza com uma abordagem que une aleatoriedade e memória de curto prazo. A frente, será discutida a descrição dessa estratégia.

2.1. Seleção de Tecnologias

Para desenvolver a implementação com base nos requisitos, optou-se pela linguagem C++, de compatibilidade do aluno. Desenvolveu-se o projeto na plataforma Visual Studio Code em busca de garantir uma codificação eficiente e organizada.

2.2. Solução Implementada

A solução implementada é dividida em classes principais que serão esclarecidas adiante.

- `'Agent'`: Opera com lógica reativa para navegar e limpar o ambiente baseado em sua percepção imediata e uma memória curta de ações passadas.
- `'Environment'`: Define o espaço de atuação do agente, gerando configurações aleatórias de sujeira em uma matriz NxN.
- `'Utility'`: Fornece suporte para execução do agente e geração de relatórios, simplificando o fluxo principal de atividades.

Em primeira análise, ressalta-se, brevemente, a função `'GenerateInputData'`, responsável por preparar previamente o ambiente. Ela constrói uma matriz de tamanho predefinido (NxN), onde cada célula tem uma probabilidade de estar suja, baseada em um percentual de sujeira que é passado como parâmetro para a função.

Adiante, a classe `'Environment'` modela o ambiente como uma matriz bidimensional em que cada célula pode conter sujeira. Métodos cruciais como `'IsDirty'` e `'Clean'` são usados para verificar e modificar o estado das células, respectivamente, simulando a percepção e interação do agente com o ambiente. Já dentro da classe `'Agent'`, o método `'Act'` é o principal da lógica operacional do agente. A cada passo da simulação, este método é invocado para determinar a próxima ação do agente, seja limpar uma célula suja ou se mover para uma nova posição.

Quando invocado, o método `'Act'` executa primeiramente uma verificação da célula atual em que o agente se encontra. Se a célula está suja, o agente executa a ação de limpeza, o que resulta em uma pontuação positiva e a célula é marcada como limpa. Esta ação é registrada em um *log* para futura análise. A 'limpeza' é tratada como um movimento especial, onde não ocorre mudança de posição, mas ainda assim é importante atualizar a memória do agente para refletir que a ação foi tomada.

Se a célula não está suja, o agente precisa decidir para onde se mover. A princípio, o agente tenta escolher uma nova direção aleatória, a decisão de movimento so é influenciada pela memória de curto prazo do agente, quando ocorre uma colisão com as paredes do ambiente. Nesse contexto, o agente opta por escolher uma direção que não tenha sido tomada recentemente, armazenada nos últimos três movimentos realizados, e que não resulte novamente em colisão, como é possível observar adiante.

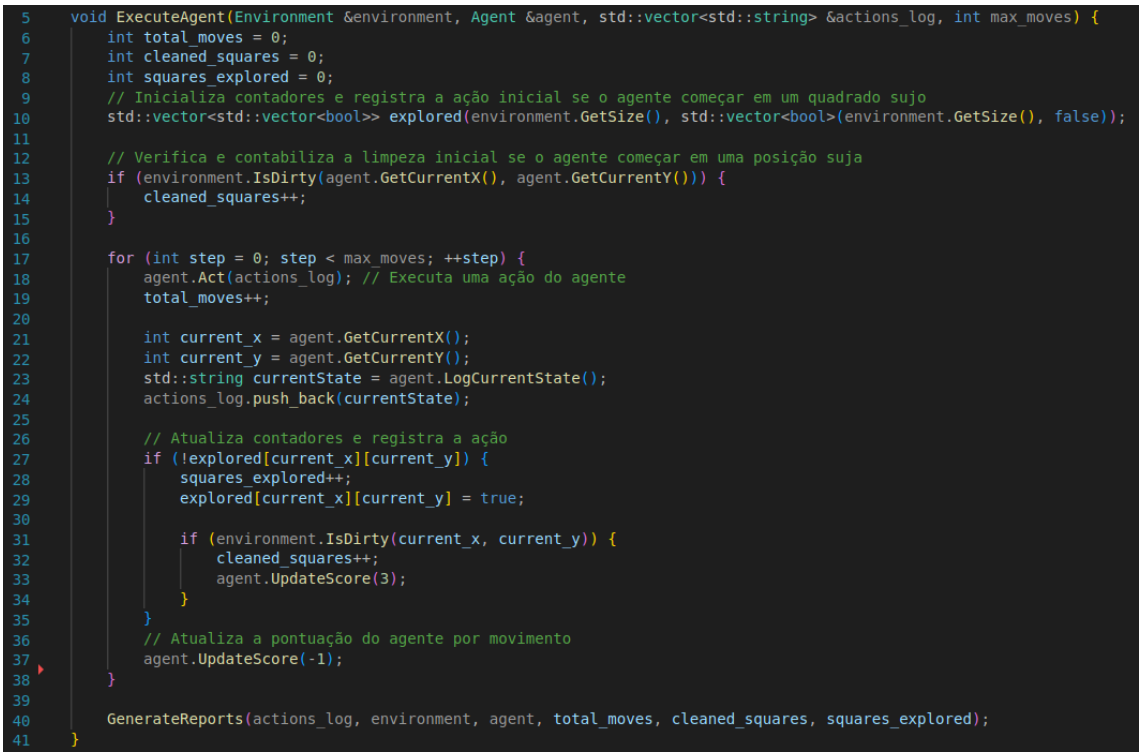
```cpp
// Define o comportamento do agente em cada movimento
void Agent::Act(std::vector<std::string> &log) {
    std::stringstream logEntry;
    // Verifica se a posição atual está suja
    if (environment.IsDirty(current_x, current_y)) {
        // Se sim, ele executa a ação de limpeza
        CleanCurrentPosition();
        // Registra a ação de limpeza no log para análise posterior
        logEntry << "Limpou em (" << current_x << ", " << current_y << ")";
        log.push_back(logEntry.str());
        // Atualiza a memória para indicar que uma ação de limpeza ocorreu, sem mudança de posição
        UpdateMemory(0, 0);
    } else {
        // Se a posição atual não está suja, o agente decide para onde se mover
        // Cria um vetor de direções possíveis (Cima, Direita, Baixo, Esquerda)
        std::vector<std::pair<int, int>> directions = { {-1, 0}, {0, 1}, {1, 0}, {0, -1} };
        std::random_shuffle(directions.begin(), directions.end());

        bool moved = false;
        for (auto &dir : directions) {
            // Para cada direção possível, verifica se o movimento é válido
            int dx = dir.first, dy = dir.second;

            // Verifica se o movimento foi recentemente realizado, usando a "memória" do agente
            if (std::find(memory.begin(), memory.end(), std::make_pair(dx, dy)) != memory.end()) {
                continue; // Ignora a direção se já foi tomada recentemente
            }

            int new_x = current_x + dx;
            int new_y = current_y + dy;

            // Checa se o novo movimento é válido e não resulta em colisão com a parede
            if (new_x >= 0 && new_x < environment.GetSize() && new_y >= 0 && new_y < environment.GetSize()) {
                // Se o movimento é válido, atualiza a posição do agente e sua memória
                UpdatePosition(dx, dy);
                moved = true;
                logEntry.str("");
                logEntry << "Move para (" << new_x << ", " << new_y << ")";
                log.push_back(logEntry.str());
                break; // Sai do loop após um movimento bem-sucedido
            } else {
                // Caso o movimento resulte em uma colisão (tentativa de sair do ambiente), registra a tentativa
                UpdateMemory(dx, dy);  // Importante: atualiza a memória para incluir a tentativa falha
                logEntry.str("");
                logEntry << "Colisão com parede ao tentar mover para (" << new_x << ", " << new_y << ")";
                log.push_back(logEntry.str());
            }
        }

        if (!moved) {
            log.push_back("Agent preso, sem movimentos válidos.");
        }
    }
}
```

Figura 1. Códificação do método `Act` em C++, o principal da lógica operacional do agente.

Além disso, a implementação conta com algumas funções auxiliares que são utilizadas pelo método `Act` para facilitar a movimentação e a manutenção da memória do agente. Após determinar um movimento, o `UpdatePosition` é chamado para alterar a posição atual do agente. A nova posição é calculada com base no movimento escolhido e a pontuação do agente é ajustada para refletir o custo desse movimento. Cada vez que o agente se move ou tenta se mover, o `UpdateMemory` é chamado para registrar o movimento na memória do agente. Movimentos recentes são mantidos no início da lista, enquanto os mais antigos são descartados se a memória atingir sua capacidade máxima.

Ademais, a execução da simulação e a avaliação do desempenho do agente são gerenciadas pela classe `Utility`. A função `ExecuteAgent` é responsável por orquestrar o processo de simulação, chamando o método `Act` do agente repetidamente e compilando as ações. Este registro das ações do agente é vital para análises subsequentes, permitindo o rastreio o comportamento e desempenho do agente como será discutido

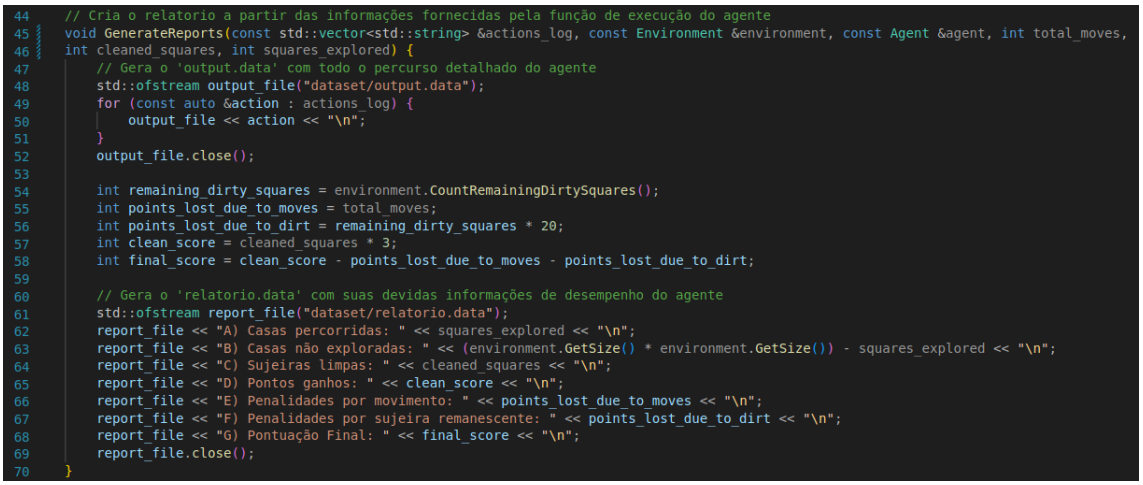
na seção de Resultados e Discussão.



```cpp
void ExecuteAgent(Environment &environment, Agent &agent, std::vector<std::string> &actions_log, int max_moves) {
    int total_moves = 0;
    int cleaned_squares = 0;
    int squares_explored = 0;
    // Inicializa contadores e registra a ação inicial se o agente começar em um quadrado sujo
    std::vector<std::vector<bool>> explored(environment.GetSize(), std::vector<bool>(environment.GetSize(), false));

    // Verifica e contabiliza a limpeza inicial se o agente começar em uma posição suja
    if (environment.IsDirty(agent.GetCurrentX(), agent.GetCurrentY())) {
        cleaned_squares++;
    }

    for (int step = 0; step < max_moves; ++step) {
        agent.Act(actions_log); // Executa uma ação do agente
        total_moves++;

        int current_x = agent.GetCurrentX();
        int current_y = agent.GetCurrentY();
        std::string currentState = agent.LogCurrentState();
        actions_log.push_back(currentState);

        // Atualiza contadores e registra a ação
        if (!explored[current_x][current_y]) {
            squares_explored++;
            explored[current_x][current_y] = true;

            if (environment.IsDirty(current_x, current_y)) {
                cleaned_squares++;
                agent.UpdateScore(3);
            }
        }
        // Atualiza a pontuação do agente por movimento
        agent.UpdateScore(-1);
    }

    GenerateReports(actions_log, environment, agent, total_moves, cleaned_squares, squares_explored);
}
```

Figura 2. Códificação do método `ExecuteAgent` em C++, responsável pelo processo de simulação do agente.

Por fim, após a conclusão da simulação, a função `GenerateReports`, compila um relatório detalhando o desempenho do agente, incluindo o número de movimentos realizados, células limpas e a pontuação final.Estes dados são importantes para medir a eficiência do agente e para avaliar se as estratégias de movimentação implementadas estão atingindo os objetivos desejados de limpeza eficaz.



```cpp
// Cria o relatorio a partir das informações fornecidas pela função de execução do agente
void GenerateReports(const std::vector<std::string> &actions_log, const Environment &environment, const Agent &agent, int total_moves,
int cleaned_squares, int squares_explored) {
    // Gera o 'output.data' com todo o percurso detalhado do agente
    std::ofstream output_file("dataset/output.data");
    for (const auto &action : actions_log) {
        output_file << action << "\n";
    }
    output_file.close();

    int remaining_dirty_squares = environment.CountRemainingDirtySquares();
    int points_lost_due_to_moves = total_moves;
    int points_lost_due_to_dirt = remaining_dirty_squares * 20;
    int clean_score = cleaned_squares * 3;
    int final_score = clean_score - points_lost_due_to_moves - points_lost_due_to_dirt;

    // Gera o 'relatorio.data' com suas devidas informações de desempenho do agente
    std::ofstream report_file("dataset/relatorio.data");
    report_file << "A) Casas percorridas: " << squares_explored << "\n";
    report_file << "B) Casas não exploradas: " << (environment.GetSize() * environment.GetSize()) - squares_explored << "\n";
    report_file << "C) Sujeiras limpas: " << cleaned_squares << "\n";
    report_file << "D) Pontos ganhos: " << clean_score << "\n";
    report_file << "E) Penalidades por movimento: " << points_lost_due_to_moves << "\n";
    report_file << "F) Penalidades por sujeira remanescente: " << points_lost_due_to_dirt << "\n";
    report_file << "G) Pontuação Final: " << final_score << "\n";
    report_file.close();
}
```

Figura 3. Códificação do método `GenerateReports` em C++, responsável pelo geração de um relatório detalhando o desempenho do agente

3. Resultados e Discussão

3.1. Modelo e Reprodução

A fim de atestar a transparência e a reprodutibilidade dos resultados apresentados neste estudo, todo o código-fonte e a documentação detalhada deste estudo estão disponíveis em um repositório GitHub público. O diretório `'src/'` contém os scripts C++ desenvolvidos para a análise, permitindo que outros pesquisadores validem, reproduzam ou expandam a implementação. A URL do repositório Git é fornecida na seção de Referências[Celso 2023] deste artigo.

3.2. Experimentação e Análise

A tabela comparativa à frente avalia o desempenho do agnete desenvolvido, apresentando Pior e Melhor Resultados, Média e Desvio Padrão para cada umas das quatro configurações de sujeira. Todas simulam o mesmo agente sob condições idênticas de tamanho de matriz, sendo esta 4x4. Nesse contexto, os resultados obtidos após 10 repetições com critério de parada de 32 movimentos para cada configuração de sujeira no ambiente são exibidos à seguir.

Configuração de Sujeira	Pior Resultado	Melhor Resultado	Média	Desvio Padrão
25% Sujos	-112	-54	-79.3	15.7
50% Sujos	-192	-100	-138.4	26.9
75% Sujos	-249	-88	-163.9	55.2
100 % Sujos	-260	-191	-166.0	134.7

Tabela 1. Medição do desempenho do agente implementado em determinados níveis de sujeira.

Os resultados destacam a variação na performance do agente com diferentes níveis de sujeira no ambiente. Aumentos na quantidade de sujeira tendem a dificultar a tarefa do agente, refletido nos valores de pontuação mais baixos e na maior variação (desvio padrão) entre os resultados. O movimento, muitas vezes, repetitivo caracterizado pelo padrão aleatório de locomoção do agente é o principal responsável pela penalização do agente, uma vez que, limitado a um critério de parada de "x" movimentos, acaba por finalizar sua execução sem cobrir grande parte do ambiente e perder diversos pontos por sujeiras remanescentes.

4. Conclusão

A solução alcançou o objetivo de criar um agente capaz de limpar o ambiente de certa forma eficiente, porém com suas devidas limitações. Observa-se que o agente mostrou capacidade de lidar com ambientes variados, mas os resultados indicam uma clara variação no desempenho. O aumento da complexidade do ambiente, com mais sujeira, leva a pontuações mais baixas, demonstrando os desafios adicionais que o agente enfrenta. A estratégia de movimento do agente, embora efetiva em certas situações, encontra limitações conforme a complexidade do ambiente aumenta. Isso é evidenciado pelos resultados mais baixos (mais negativos) em ambientes com maior percentual de sujeira.

Os experimentos sugerem áreas para futuras melhorias, especialmente na otimização da estratégia de movimento e na eficiência da limpeza em ambientes mais

desafiadores. Em suma, a experimentação revelou tanto o potencial quanto as limitações da abordagem atual. Embora o agente demonstre uma capacidade básica de navegação e limpeza, há uma certa necessidade de aprimoramento, visando aumentar a eficiência e adaptabilidade em diversos cenários de limpeza.

5. Compilação e Execução

5.1. Arquivo de Entrada

Para começar a simulação, é essencial que o usuário gere um arquivo de entrada chamado `'input.data'` que configura o estado inicial do ambiente de simulação. Este arquivo é criado de forma automática pela função `'GenerateInputData'`, contendo seus parâmetros ajustáveis de acordo com o usuário como mostram as figura à seguir.

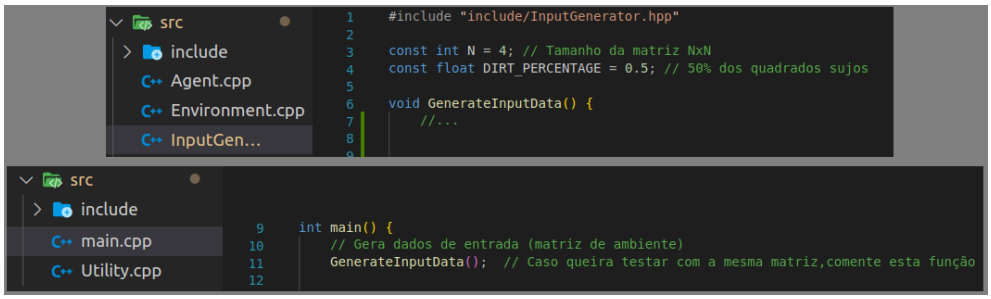


Figura 4. Imagem exemplificativa dos possíveis ajustes na criação do ambiente matricial.

Assim, a estrutura do arquivo gerado seguirá o formato demonstrado na figura adiante. Contendo em sua primeira linha os valores NxN do tamanho da matriz e em seguida a matriz, onde 0 representam quadrados limpos e 1 os quadrados sujos.

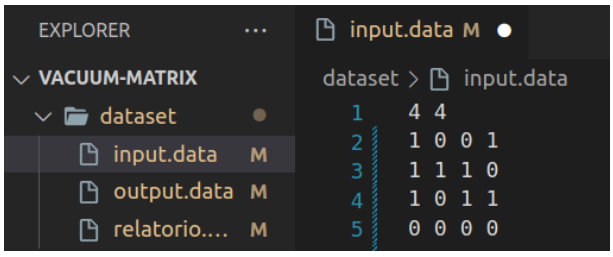


Figura 5. Imagem exemplificativa do arquivo 'input.data' que deve ser inserido ou alterado.

5.2. Arquivos de saída

Ao final de sua execução, o programa gera um arquivo `'output.data'` dentro da pasta `'dataset'`, o qual contém todo o percurso do agente ao longo de sua execuação e outro arquivo `'relatorio'`, contendo o desempenho do agente. Certifique-se de verificá-los ao final da compilação.

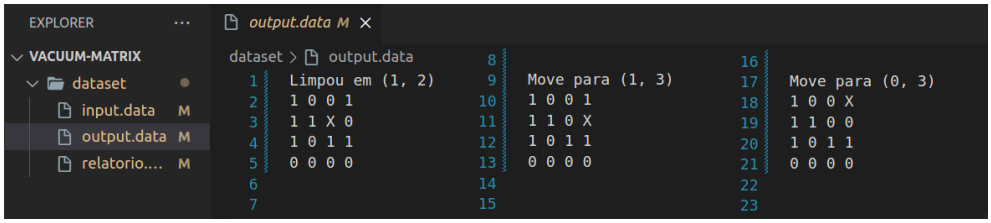


Figura 6. Imagem exemplificativa do arquivo 'output.data' gerado após a compilação do programa.

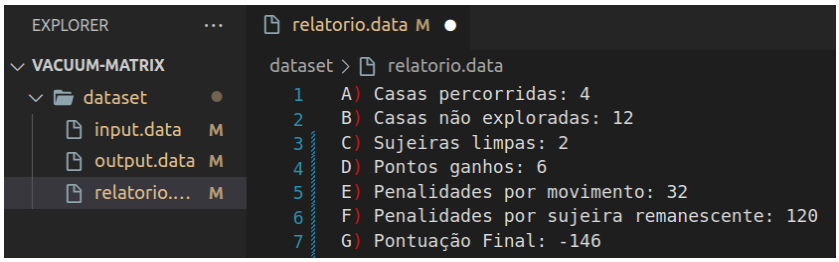


Figura 7. Imagem exemplificativa do arquivo 'relatorio.data' gerado após a compilação do programa.

5.3. Execução

Por fim, esse programa possui um arquivo `'Makefile'` e um `'CMakeLists'`, os quais realizam todo o procedimento de compilação e execução. Para tanto, cabe ao usuário escolher o de sua preferência, porém se atente ao fato de que para cada um temos as seguintes diretrizes de execução.

Makefile	
Comando	Função
make clean	Apaga a última compilação realizada contida na pasta build.
make	Executa a compilação do programa utilizando o gcc, e o resultado vai para a pasta build.
make run	Executa o programa da pasta build após a realização da compilação.
make c	Apaga, compila e executa o programa.

Tabela 2. Comandos em sequência de execução para o compilador Makefile.

CMake	
Comando	Função
cmake -B ./build	Apaga a última compilação realizada contida na pasta build.
cmake –build ./build	Executa a compilação do programa utilizando o cmake, e o resultado vai para a pasta build.
./build/app	Executa o programa da pasta build após a realização da compilação.

Tabela 3. Comandos em sequência de execução para o compilador CMake.

Referências

Celso, V. (2023). Agente aspirador inteligente. `https://github.com/celzin/Vacuum-Matrix`. Acesso em: 29 mar. 2024.