



UNIVERSIDADE FEDERAL DO MARANHÃO
BACHARELADO INTERDISCIPLINAR EM CIÊNCIA E TECNOLOGIA
INTELIGÊNCIA ARTIFICIAL

ARTHUR FELLIPE LIMA REIS

São Luís

2024

SUMÁRIO

1. INTRODUÇÃO
2. DESENVOLVIMENTO
3. RESULTADOS
4. DISCUSSÕES
5. CONCLUSÃO

INTRODUÇÃO


Um agente em Inteligência Artificial (IA) é um sistema ou entidade capaz de interagir com seu ambiente, tomar decisões e executar ações com base nas informações coletadas. No contexto da IA, um agente pode ser programado para realizar tarefas específicas, como jogar jogos, reconhecer padrões, otimizar processos, ou responder a comandos do usuário. A abordagem de busca em largura (Breadth-first search (Busca em largura)) é uma técnica fundamental em algoritmos de IA que se concentra em explorar todos os possíveis estados de um problema em uma ordem lógica e estruturada, geralmente começando do estado mais inicial e movendo-se para os estados mais próximos.

A aplicação do BFS em sistemas de IA geralmente envolve problemas como o caminho mais curto em grafos, desta forma, utilizaremos o PYMAZE como fundamento de nosso projeto, este é um conjunto de alguns scripts que, por meio de POO, possuem conceitos fundamentais para gerarmos um labirinto que será gerado aleatoriamente, sempre podemos chamar funções e variáveis fundamentais, onde podemos fazer diferentes modificações para adaptar no nosso projeto, será feito um algoritmo aplicado ao conceito de BFS que busque o caminho mais curto até chegar à saída do labirinto. Portanto, o agente será capaz de explorar todas as possibilidades e tomar decisões baseadas em uma busca eficiente para alcançar seu objetivo final.

Detalhando os conceitos teóricos envolvidos, o labirinto é composto por coordenadas de linha e coluna (x,y), o algoritmo BFS analisa cada célula(nó), utilizando um comportamento de lista chamado FIFO(first-in,first-out), onde os primeiros dados que entrarão para ser tratados na lista, serão os primeiros a sair dela, contudo surge os conceitos importantes, onde são bem explicados com esse pseudocódigo:

```
Add start cell in both Frontier and Explored
Repeat until Goal is reached or Frontier is Empty:
    currCell=Frontier.pop(0)=5,5
    for each direction(ESNW):
        childCell=Next Possible Cell
        if childCell already in Explored list→Do nothing
        otherwise→ Append/Push childCell to both Explored & Frontier
```

E→ childCell= X
S→ childCell= X
N→ childCell= 4,5
W→ childCell= 5,4



A célula inicial (x,y), será a primeira célula da lista frontier e explored: frontier é uma lista de onde todas as células que serão analisadas estão, explored, são todas aquelas que foram descobertas, (vai ser importante implementar um raciocínio para que o agente não percorra nós já explorados).

currCell, a lista que armazena as células naquele momento, vai retirar o primeiro elemento da lista frontier, dessa forma, nosso possível novo caminho vai ser verificado: NORTE, SUL,

LESTE ,OESTE serão nossas possíveis childCell, se o caminho estiver bloqueado por uma parede,ou já ter sido explorado, não acontecerá nada, caso contrário, percorra o próximo caminho disponível. Portanto, será feito assim por diante, os nós mais próximos disponíveis vão sendo percorridos até que se consiga chegar ao nosso objetivo.

DESENVOLVIMENTO

Após explicar os conceitos teóricos envolvidos, vamos ao código: primeiro criamos um outro projeto python na mesma pasta, e importamos os principais objetos, labirinto, agente, cor. Dessa forma, conseguimos trabalhar com o básico, podemos não apenas gerar um labirinto mas alterar outras variáveis, quantos caminhos alternativos, qual o tamanho que queremos, se queremos expandir apenas colunas verticais ou horizontais, mudando o objetivo de lugar, criando o agente, podemos até mesmo salvar um arquivo de nosso labirinto para fazer um estudo posterior apenas dele

O INÍCIO

```
from pyamaze import maze, COLOR, agent, textLabel
m = maze(5, 5)
m.CreateMaze()
a = agent(m)
path = BFS(m)
m.tracePath({a: path})
m.run()
```

Assim sempre será gerado um labirinto aleatoriamente de tamanho 5x5, com um agente (quadrado azul por default), onde ele vai percorrer um caminho até o outro: nosso objetivo, mas para que esse algoritmo funcione precisamos determinar nossa função `def BFS(m):`

FUNÇÃO BFS

```
def BFS(m):
    start = (m.rows, m.cols)
    frontier = [start]
    explored = [start]
    bfsPath = {}

    while len(frontier) > 0:
        currCell = frontier.pop(0)
        if currCell == (1, 1):
            break
        for d in 'ESNW':
            if m.maze_map[currCell][d] == True:
                if d == 'E':
                    childCell = (currCell[0], currCell[1] + 1)
                elif d == 'W':
                    childCell = (currCell[0], currCell[1] - 1)
                elif d == 'N':
                    childCell = (currCell[0] - 1, currCell[1])
                elif d == 'S':
                    childCell = (currCell[0] + 1, currCell[1])

                if childCell in explored:
                    continue

                frontier.append(childCell)
                explored.append(childCell)
                bfsPath[childCell] = currCell
```

A função **BFS** implementa a busca em largura (Breadth-First Search, ou BFS) para encontrar o caminho do final (célula `(m.rows, m.cols)`) até o início (célula `(1, 1)`) em um labirinto representado pelo objeto `m`. O algoritmo começa na célula de destino, que é a célula inferior direita do labirinto, e vai explorando as células vizinhas, seguindo a lógica da busca em largura.

A função começa definindo o ponto de partida como `(m.rows, m.cols)` (a célula no canto inferior direito) e inicializa a fila de exploração `(frontier)` com essa célula, bem como a lista de células já exploradas `(explored)`. Além disso, um dicionário `bfsPath` é criado para armazenar a relação entre cada célula e sua célula anterior no caminho.

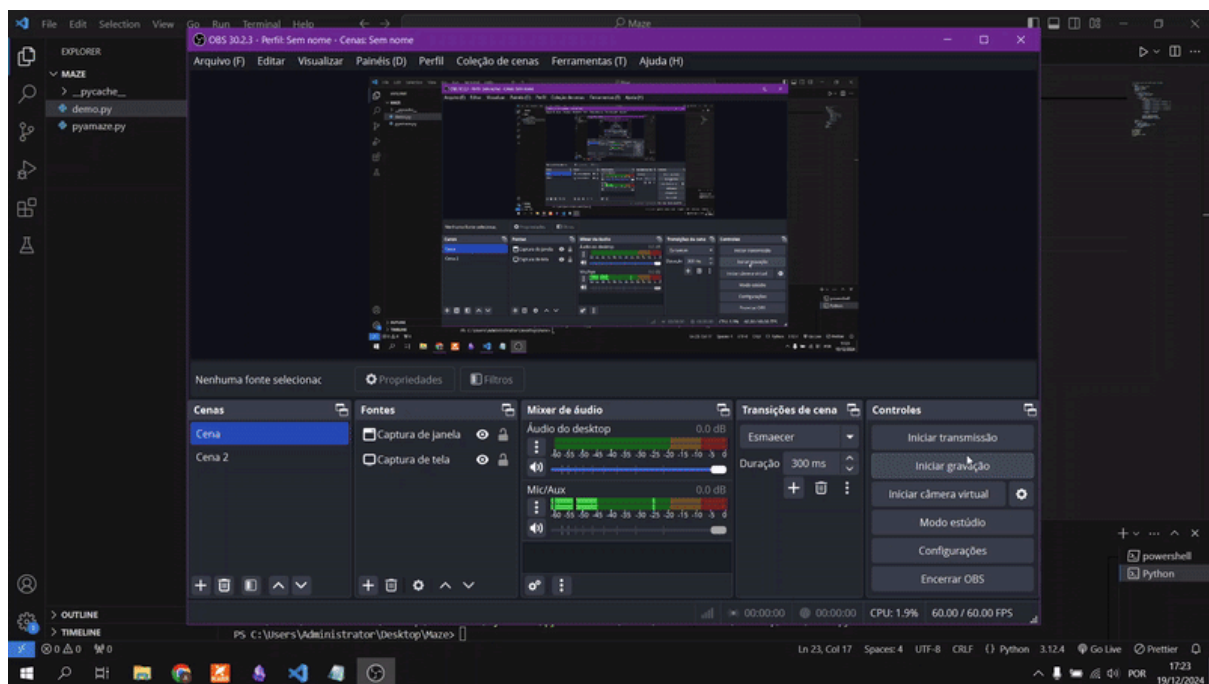
Dentro do laço `while`, o algoritmo continua até que a célula atual seja a célula de início (1, 1), ou até que não haja mais células na fila de exploração. A cada iteração, o algoritmo retira a célula da frente da fila (`currCell`) e explora suas células vizinhas, verificando se há uma parede aberta (onde o valor em `m.maze_map[currCell][d]` é `True`). Dependendo da direção ('E', 'W', 'N', 'S' para leste, oeste, norte e sul, respectivamente), a célula vizinha (`childCell`) é calculada.

Se a célula vizinha (`childCell`) ainda não foi explorada, ela é adicionada à fila de exploração (`frontier`) e à lista de células exploradas (`explored`), além de ser registrada no dicionário `bfsPath` com a célula atual como seu antecessor. Isso garante que o caminho seja rastreado a partir da célula de destino até a célula de início.

O algoritmo continua até encontrar a célula de início, momento em que o caminho é completado. A busca em largura garante que o caminho encontrado seja o mais curto possível, já que explora primeiro todas as células vizinhas antes de avançar para células mais distantes.

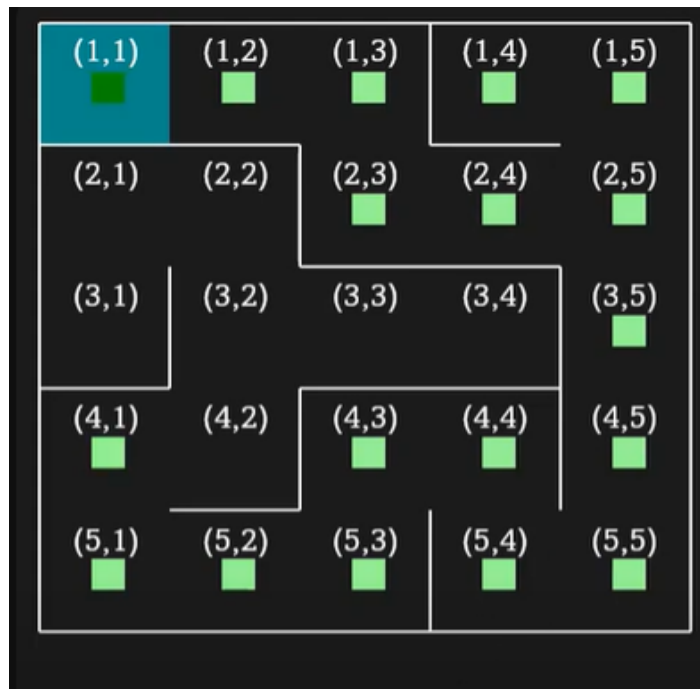
RESULTADOS

Agora, todo labirinto que for gerado, o algoritmo bfs vai calcular todas as rotas possíveis e procurar pelo melhor caminho que estiver disponível, (não é visível no momento)



REPRESENTAÇÃO VISUAL DO BFS

Essa é uma representação visual do labirinto com o valor de cada célula, e da representação do percurso dos diferentes caminhos possíveis, até que se encontre um caminho onde se chega ao objetivo



DISCUSSÕES

O código implementa o algoritmo de Busca em Largura (BFS) para explorar um labirinto e, em seguida, simula a movimentação de um agente que percorre todas as células exploradas pelo algoritmo. O BFS é uma estratégia de busca que garante a exploração de todos os caminhos possíveis de maneira ordenada, começando pelas células mais próximas ao ponto de origem. Isso permite encontrar o caminho mais curto para o destino (caso ele exista).

A BFS funciona utilizando duas estruturas de dados principais: **frontier** e **explored**. A **frontier** armazena as células ainda a serem exploradas e é processada em ordem, começando pela célula inicial. A **explored** armazena as células já visitadas, evitando que o algoritmo explore as mesmas células repetidamente. A busca termina quando o agente encontra o destino ou quando todas as células possíveis são exploradas. Durante o processo, o algoritmo também mantém um mapa (**bfsPath**) que associa cada célula a sua célula anterior, permitindo reconstruir o caminho.

O código de movimentação do agente percorre todas as células exploradas, não necessariamente o caminho mais curto. Isso é feito movendo o agente para cada célula visitada durante a busca e atualizando visualmente o labirinto a cada movimento. A função

`m.update()` é chamada após cada movimento para atualizar a visualização do estado do labirinto, e `time.sleep(0.5)` é usado para criar uma pausa entre cada movimento, facilitando a visualização.

No entanto, algumas melhorias podem ser feitas para otimizar o código. O uso de listas para a `frontier` no BFS pode ser ineficiente, pois a operação `pop(0)` em listas tem complexidade $O(n)$. Substituir a lista por uma fila (`queue`), que tem complexidade $O(1)$ para inserções e remoções, seria mais eficiente. Além disso, o agente poderia ser ajustado para percorrer apenas o caminho mais curto encontrado pela BFS, ao invés de todas as células exploradas, para tornar a simulação mais clara.

Outro ponto de melhoria seria permitir a escolha de diferentes algoritmos de busca, como DFS ou A*, para comparar seus desempenhos em termos de tempo e recursos, além de permitir uma visualização mais interativa. Também é importante observar que armazenar todas as células exploradas pode ser custoso em termos de memória, especialmente para labirintos grandes, então otimizações podem ser necessárias dependendo do tamanho do labirinto.

Em resumo, o código está funcional e simula a movimentação do agente de maneira interessante, mas melhorias podem ser feitas para otimizar o desempenho, a clareza da visualização e a interação com o usuário.

CONCLUSÃO

Em conclusão, o código implementa de forma eficaz o algoritmo de Busca em Largura (BFS) para explorar um labirinto e simula a movimentação do agente ao longo dos caminhos explorados. O BFS, sendo uma abordagem eficiente para encontrar o caminho mais curto em labirintos, garante que o agente explore todas as células acessíveis de maneira ordenada e sem redundância. A visualização da movimentação do agente, apesar de percorrer todos os caminhos, proporciona uma boa demonstração de como o algoritmo examina as possibilidades.

Entretanto, existem áreas para otimização e melhorias. A substituição da lista `frontier` por uma fila (`queue`) pode melhorar o desempenho, especialmente para labirintos grandes, ao reduzir o custo de remoção de elementos. Além disso, a movimentação do agente poderia ser ajustada para seguir apenas o caminho mais curto encontrado, o que tornaria a simulação mais clara e intuitiva. Melhorias na interação e a possibilidade de comparar diferentes algoritmos de busca, como DFS ou A*, também enriqueceriam o código e tornariam a simulação mais dinâmica.

No geral, o código cumpre bem sua função de simular a BFS, mas pode ser aprimorado em termos de eficiência, clareza da visualização e flexibilidade. Ao implementar essas melhorias, o código poderá não apenas ser mais rápido, mas também mais informativo e interativo, oferecendo uma melhor experiência ao usuário.

