INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SÃO PAULO Campus Birigui	**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA** **Campus Birigui** **Bacharelado em Engenharia de Computação**

Disciplina: Inteligência artificial	**Atividade**
Professor: Prof. Dr. Murilo Varges da Silva	**Data:** 18/08/2023
Nome do Aluno: Henrique Akira Hiraga	**Prontuário:** BI300838X

TRABALHO - BUSCA LABIRINTO

ESTUDO DO FUNCIONAMENTO

Geração de labirinto

A geração do labirinto é realizada pelo código denominado maze_generator.py. A inicialização do labirinto é através de um array cheio de 0's, indicando células vazias.

```
# initialize the grid array full of zeros
num_rows = 41
num_columns = num_rows
grid = np.zeros((num_rows, num_columns))
```

A posição inicial será demarcada em verde "2", sendo então o ponto de partida do labirinto. O ponto que determina a conclusão do labirinto é mostrado como um ponto vermelho "3". Esses pontos de início e fim do labirinto podem ser alterados no código, porém deve-se tomar cuidado para não quebrar o intuito do labirinto. As cores também podem ser trocadas, mas para manter de uma forma mais intuitiva mantém as mesmas cores.

```
# define start and goal
grid[0, 0] = 2
grid[-1, -1] = 3
```

O programa entra em um loop principal e permanece lá até o labirinto estar completo. A geração do labirinto é feita passo a passo através da função 'generate_step' que irá determinar o próximo passo na geração. Quando ela recebe a posição atual, o histórico das posições percorridas e o contador de passos de retorno como entrada. Ela determina as próximas posições em que o algoritmo pode avançar. Se houver pelo menos uma posição válida para se mover, o algoritmo escolhe uma delas aleatoriamente e avança para a posição. As posições escolhidas indicarão o caminho em que é possível

caminhar, sendo eles na cor branca “1”. A posição anterior é atualizada para a nova posição e adicionada ao histórico de posições visitadas.

Esse algoritmo conta com uma funcionalidade de dead end, ou também conhecido como beco sem saída. Se caso não houver posições válidas para onde se mover a partir da posição atual, o algoritmo retrocede na trilha do labirinto, voltando para uma posição anterior que tenha caminhos não explorados. Os números de passos do retrocesso são rastreados para evitar retroceder demais.

Mantendo toda essa lógica anteriormente apresentada, o algoritmo continuará gerando passo a passo até que todas as células tenham sido visitadas e o labirinto esteja completamente gerado.

Estratégias: Busca em largura, profundidade e A*

Antes de iniciarmos falando sobre como os algoritmos resolvem esse problema, devemos entender o propósito de cada um. O algoritmo BFS, ou também conhecido como busca em largura, explora os nós em camadas, começando pela posição inicial e expandindo gradualmente para nós vizinhos antes de prosseguir para camadas subsequentes. Garantindo que encontre o caminho mais curto. Para a resolução do labirinto, o algoritmo utiliza o BFS para controlar a fronteira e os nós explorados. Todo o processo de resolução pode ser exibido se caso habilitado, mostrando o caminho em que o BFS percorre.

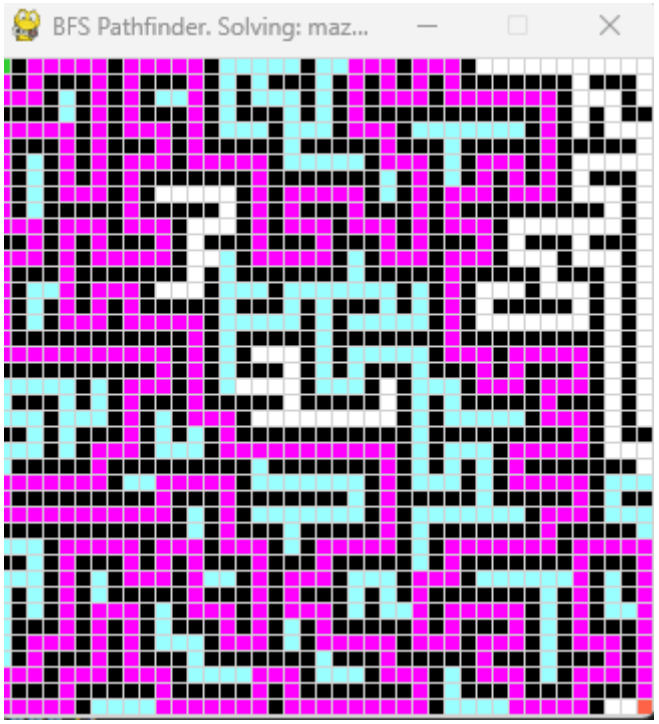
O algoritmo DFS, também conhecido como busca em profundidade, explora o máximo possível em uma direção antes de retroceder. No problema do labirinto, ele começa na posição inicial e explora a ramificação de caminhos até alcançar a posição final. O foco do DFS está na profundidade, o que significa que ele pode seguir um caminho até o final antes de voltar atrás e explorar outros caminhos. A classe DFS implementada mantém uma fronteira entre nós a serem explorados e uma lista de nós já explorados. Ele itera pela fronteira, expandindo os nós para encontrar caminhos até o objetivo.

O algoritmo A* utiliza as técnicas do algoritmo BFS e a heurística para encontrar o melhor caminho possível. Ele avalia os nós com base em uma função de custos estimados que combina o custo acumulado até o nó e uma estimativa do custo restante até o objetivo. O código em que é implementado o A* possui uma classe ‘Node’ para representar os nós e uma classe ‘PriorityQueue’ para manter a fronteira de

nós a serem explorados.

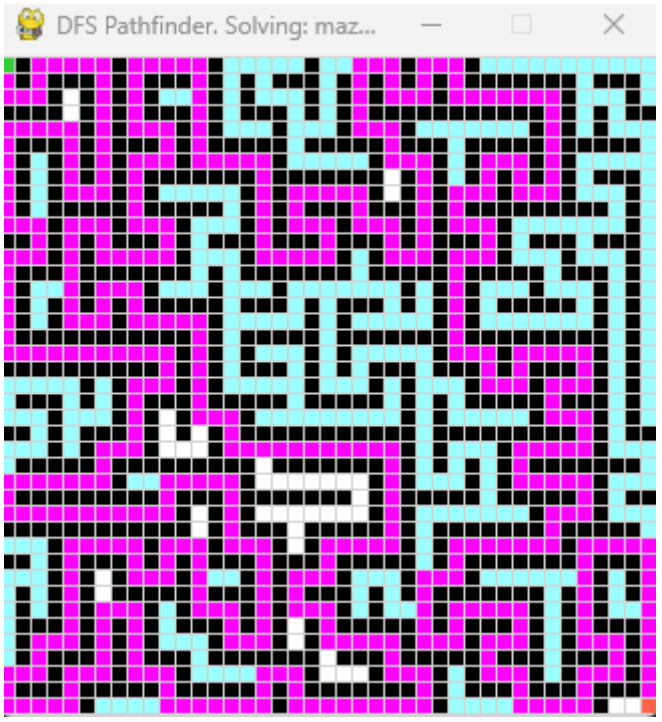
Estudo com tempo de execução e soluções encontradas cada estratégia de busca

BFS:



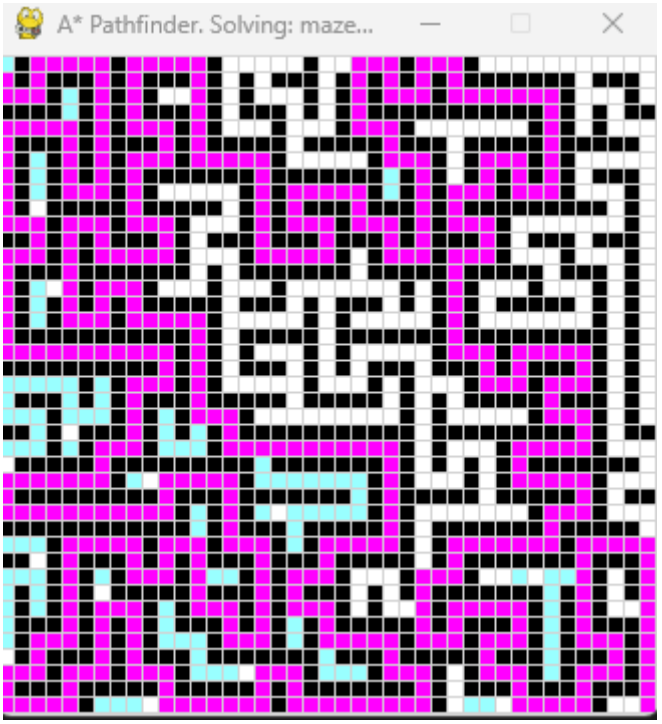

```
Pathfinder BFS. solving: mazes/maze_1.csv
--- finished 0.117 s---
```

DFS:




```
Pathfinder DFS. solving: mazes/maze_1.csv
--- finished 0.133 s---
```

A*:




```
A* Pathfinder. Solving: mazes/maze_1.csv
--- finished 0.048 s---
```

Após a realização dos testes e o conhecimento de cada algoritmo por trás da resolução do labirinto, podemos concluir que o algoritmo A* é o mais eficiente entre todos. Em seguida temos o BFS mostrando uma melhor eficiência do que o DFS. O DFS demonstrou o pior desempenho entre os testados, sendo perceptível até no mapa gerado, onde ele explorou uma maior quantidade de nós que não tinham necessidade. Em termos de tempo, no geral todos foram rápidos, porém em problemas mais complexos, os algoritmos que apresentaram o melhor tempo podem ter um desempenho muito superior aos demais.

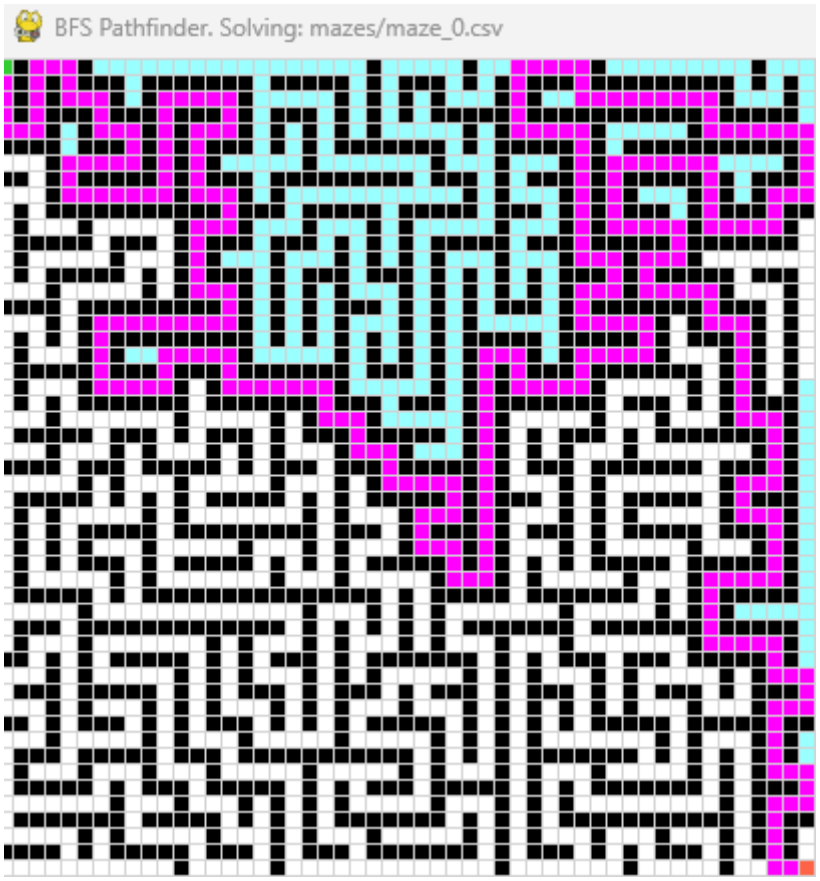
Alterando o tamanho do labirinto e realizando experimentos

Para alterar o tamanho do labirinto a ser gerado deve-se alterar o valor da variável ‘num_row’, vale ressaltar que os valores inseridos nessa variável tem que ser ímpares, caso contrário, apresentará erro na geração do labirinto.

```
# initialize the grid array full of zeros
num_rows = 51
num_columns = num_rows
grid = np.zeros((num_rows, num_columns))
```

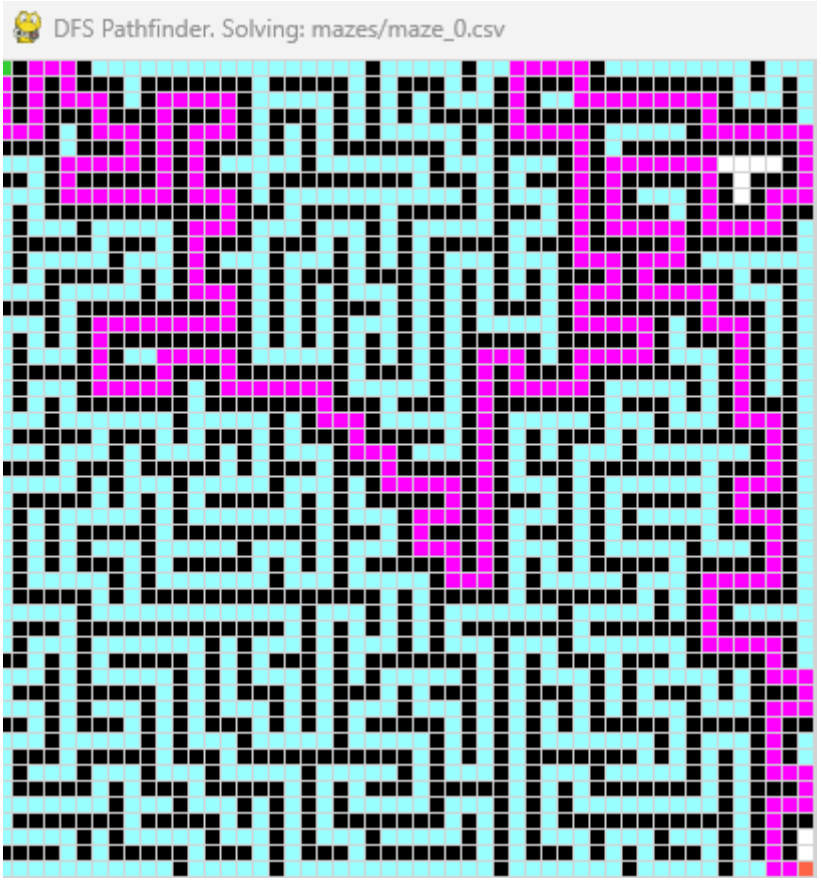
O valor em questão foi 51, aumentando em mais 10 linhas e 10 colunas em comparação com o labirinto anterior. A seguir serão apresentados os resultados dos 3 algoritmos.

BFS:



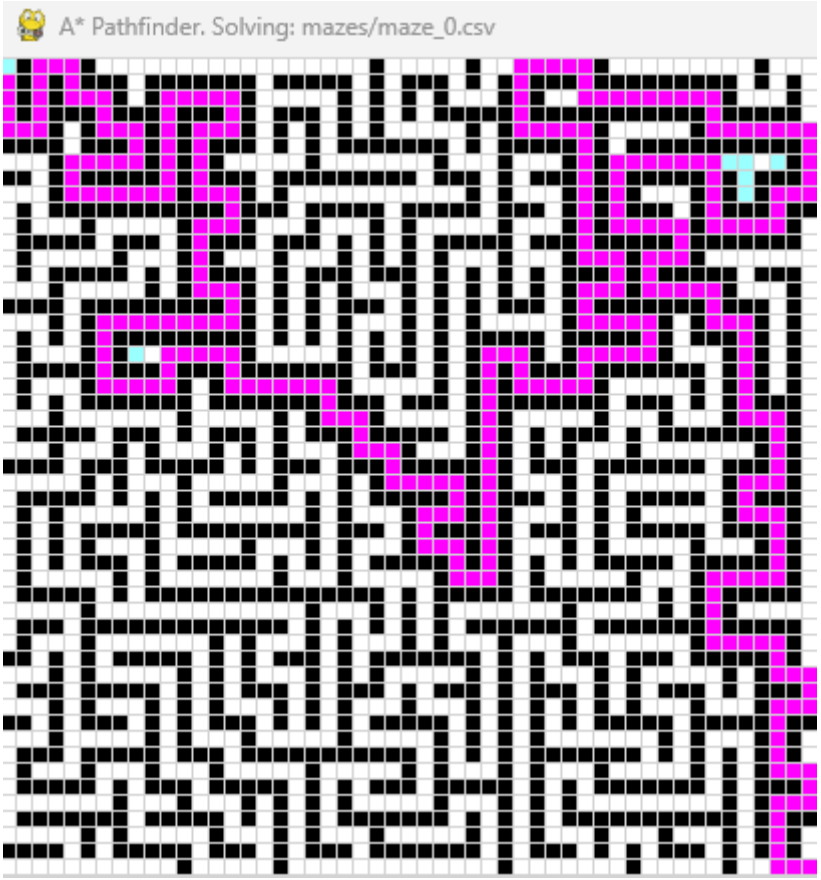

```
Pathfinder BFS. solving: mazes/maze_0.csv
--- finished 0.070 s---
```

DFS:




```
Pathfinder DFS. solving: mazes/maze_0.csv
--- finished 0.320 s---
```

A*:

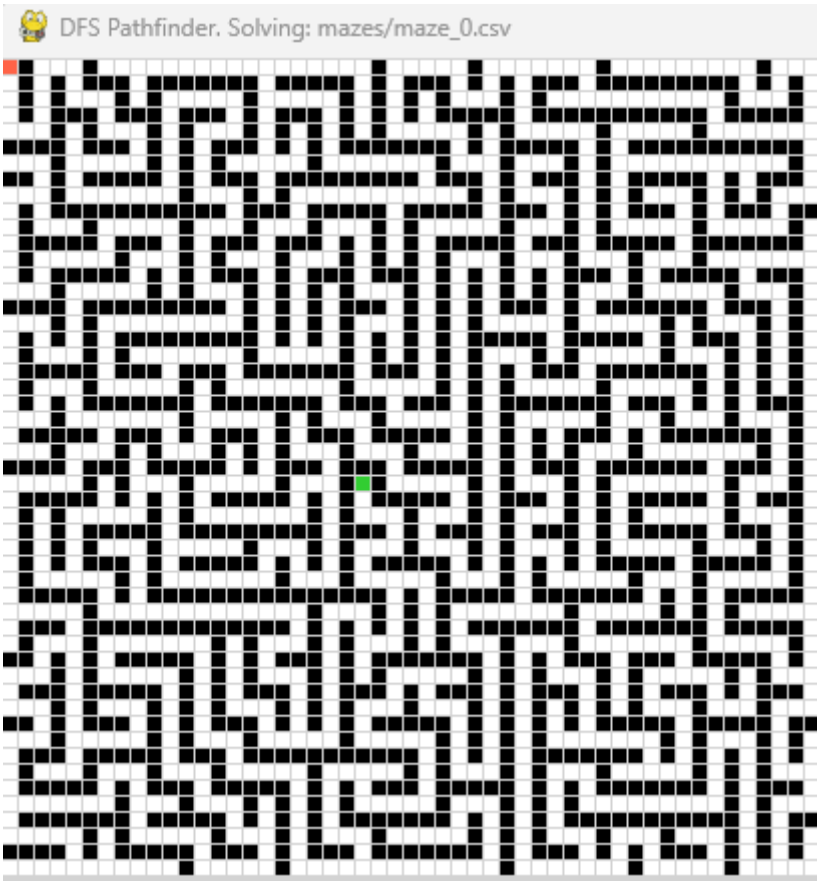


Com os experimentos realizados, constatamos que a eficácia do algoritmo A* é completamente superior aos demais, pois como podemos ver pela Figura x, o labirinto foi concluído com um caminho praticamente sem desvio do objetivo. Em contrapartida, os resultados dos outros dois algoritmos foram ineficientes em comparação ao A*. Sendo o BFS em segundo lugar, ele consegue completar em um tempo interessante, porém como visto na Figura x ele acabou realizando uns desvios para o canto superior do labirinto. Em último lugar temos o algoritmo DFS, sendo o pior resultado, ele concluiu o labirinto em um tempo muito acima dos demais e em quesito a caminhos explorados, foi praticamente o labirinto inteiro, isso se dá pelo seu comportamento de explorar até o fim um caminho para que vá para os demais.

Alterando a posição de início e fim do labirinto

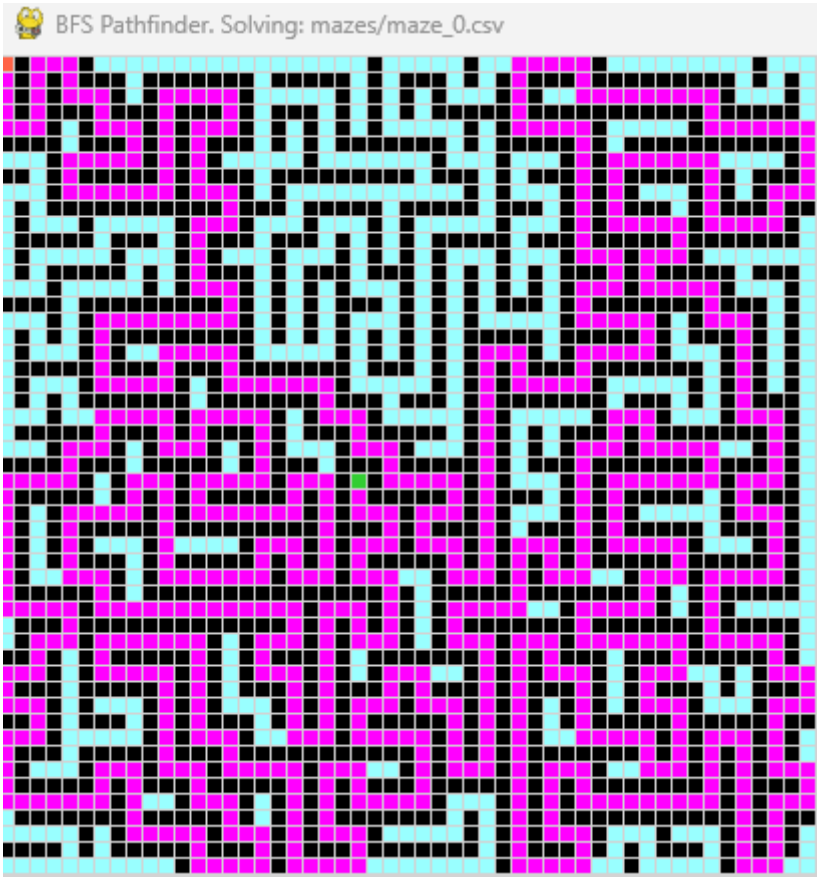
Como experimento, a posição de início e fim do labirinto foi trocado. O início

agora será próximo do centro do labirinto, na linha 26 e coluna 22 e o fim passará a ser a posição inicial original, linha 0 e coluna 0. Com isso obtemos o labirinto da seguinte forma:



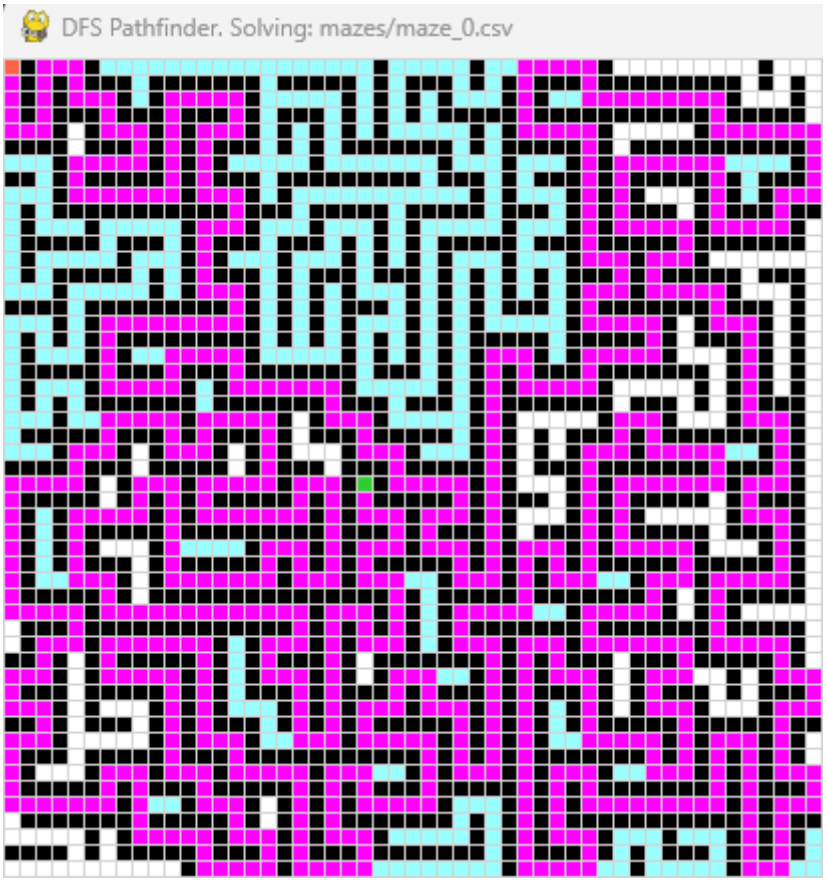
A partir deste labirinto, serão feitos testes com os três algoritmos e seus resultados serão apresentados a seguir:

BFS:



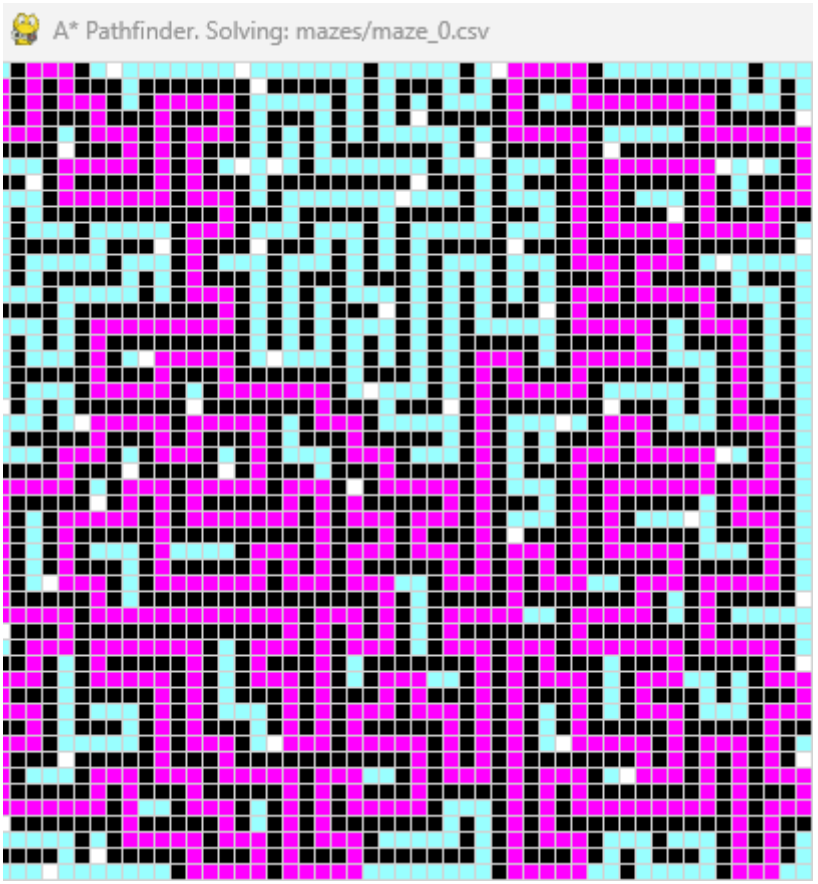

```
Pathfinder BFS. solving: mazes/maze_0.csv
--- finished 0.327 s---
```

DFS:




```
Pathfinder DFS. solving: mazes/maze_0.csv
--- finished 0.235 s---
```

A*:

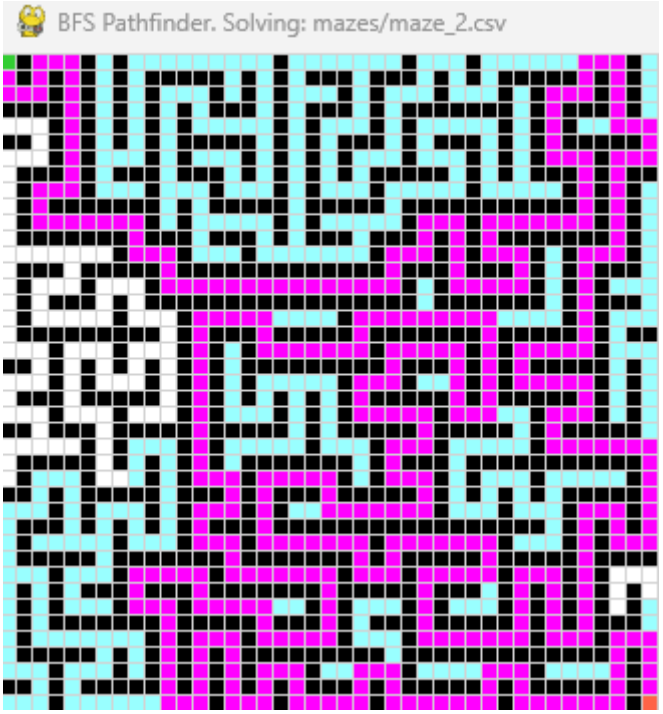


A alteração no ponto de início e fim influenciou de maneira negativa os algoritmos para a conclusão do problema. Como visto, o A* permaneceu sendo o melhor de todos mesmo tendo que percorrer uma grande distância. Em seguida temos o DFS que pela primeira vez não obteve o pior resultado, ele conseguiu ser o algoritmo que percorre a menor distância, visto que há muitos nós não visitados. E por último temos o BFS que teve o pior tempo e visitou todos os nós possíveis.

Implementando um contador de nós visitados

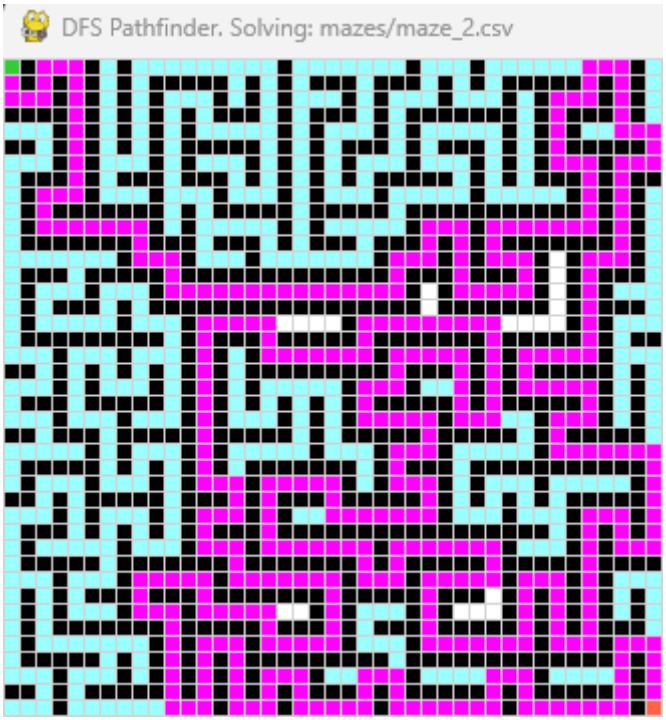
Para o teste a seguir, será selecionado um dos mapas gerados no arquivo original. O nome do labirinto se chama “maze2_csv” e nele será feito o teste de quantos nós cada algoritmo tem que visitar para solucionar o problema. A seguir será apresentado o resultado dos algoritmos:

BFS:




```
Pathfinder BFS. solving: mazes/maze_2.csv
--- finished 0.115 s---
Nodes visited: 783
```

DFS:




```
Pathfinder DFS. solving: mazes/maze_2.csv
--- finished 0.140 s---
Nodes visited: 859
```

A*:


```
A* Pathfinder. Solving: mazes/maze_2.csv
--- finished 0.036 s---
Nodes visited: 505
```

Como possível analisar, novamente o A* foi o mais eficiente para encontrar a saída do labirinto devido a sua natureza ele conseguiu percorrer a menor quantidade de nós. Em segundo lugar temos o BFS que visitou uma quantidade maior do que o A* e por fim levou um tempo maior para a conclusão. Por último temos o DFS que percorreu a maior quantidade de nós e levou o maior tempo para a conclusão.

Função para o cálculo de custo originalmente implementado

Dentro dos arquivos é possível encontrar o "helper_aStar.py", ele é o responsável por toda a lógica envolvendo o algoritmo A*. Nele é disponibilizado a função "compute_node_cost".

```
def compute_node_cost(pos, goal):
    """
    Parameters
    ----------
    pos : tuple of 2 ints
```

```
        position of node whos cost want to compute.
    goal : tuple of 2 ints
        position of goal.
    Returns
    -------
    cost : float
        euclidean distance pos-goal
    """
    x, y = pos
    x_goal, y_goal = goal
    cost = np.sqrt((x_goal-x)**2 + (y_goal-y)**2)
    return cost
```

Como podemos ver, ele utiliza de um cálculo de custo conhecido. Utilizando a fórmula euclidiana para calcular a distância percorrida, essa é uma heurística comumente utilizada nos algoritmos A*. Essa fórmula representa a distância direta (em linha reta) entre dois pontos em um espaço bidimensional.

Implementado a distância cityblock na busca A* e realizando testes

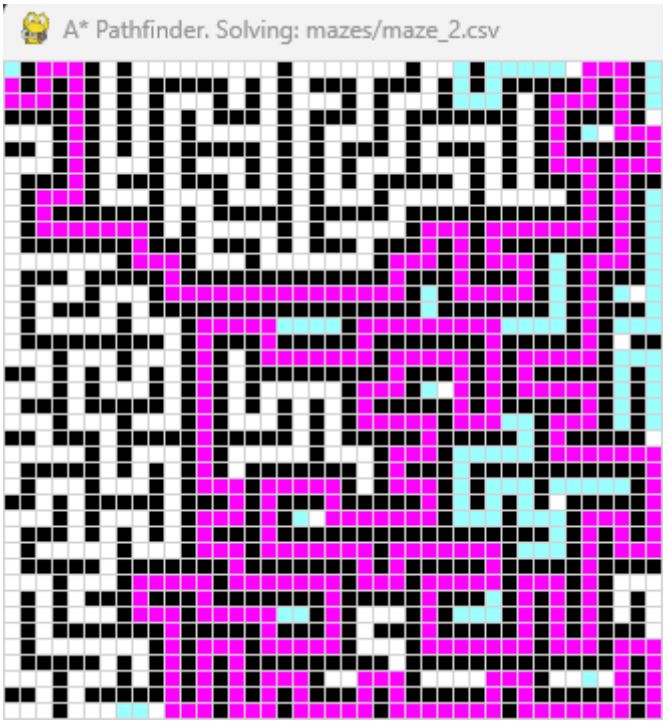
Para implementar esse cálculo de distância no algoritmo de busca A* basta importarmos uma biblioteca scipy.spatial.distance cityblock. Após isso, é necessário ir a função “compute_node_cost” e alterar o código presente para:

```
def compute_node_cost(pos, goal):
    """
    Parameters
    ----------
    pos : tuple of 2 ints
        position of node whos cost want to compute.
    goal : tuple of 2 ints
        position of goal.
    Returns
    -------
    cost : float
        euclidean distance pos-goal
    x, y = pos
```

```
    x_goal, y_goal = goal
    cost = np.sqrt((x_goal-x)**2 + (y_goal-y)**2)
    return cost
    """
    cost = cityblock(pos, goal)
    return cost
```

A seguir será feito uma comparação entre o método de cálculo da distância original e a cityblock.

A* cityblock:




```
A* Pathfinder. Solving: mazes/maze_2.csv
--- finished 0.038 s---
Nodes visited: 494
```

A* original:


```
A* Pathfinder. Solving: mazes/maze_2.csv
--- finished 0.036 s---
Nodes visited: 505
```

Em primeiras impressões podemos notar que o tempo para resolver é praticamente o mesmo, tendo apenas milésimos de segundos de diferença entre os dois. Mas se olharmos na quantidade de nós visitados é perceptível que o método cityblock conseguiu ser mais eficiente do que o método original, visitando uma menor quantidade de nó para a solução do problema. O cityblock possui como característica a soma das diferenças absolutas das coordenadas x e y entre dois pontos. Em um ambiente onde os movimentos só podem ocorrer na horizontal e na vertical (sem diagonais), a distância de Manhattan representa o número mínimo de movimentos necessários para mover-se de um ponto para outro, seguindo apenas direções perpendiculares.