U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

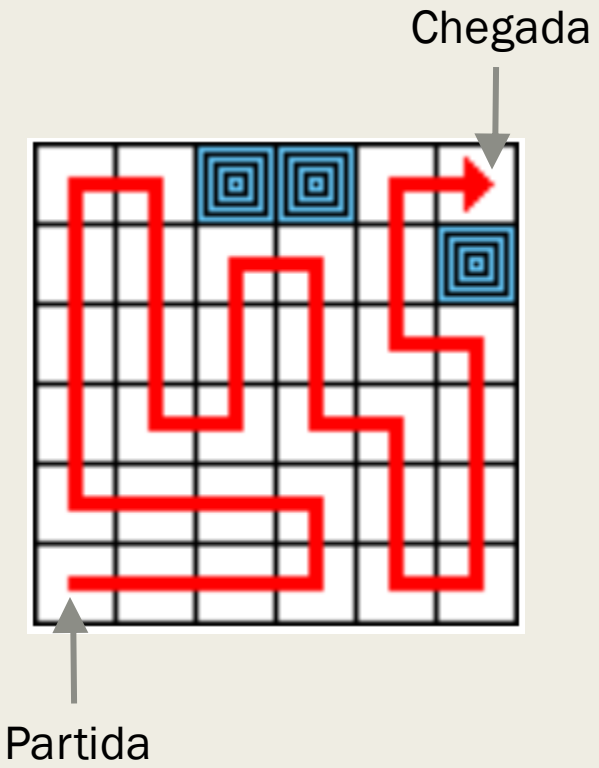
UNEQUAL LENGTH MAZES

Inteligência Artificial, 3LEIC01

Grupo 05_1A:
Henrique Ribeiro Nunes, up201906852
Margarida Assis Ferreira, up201905046
Patrícia do Carmo Nunes Oliveira, up201905427

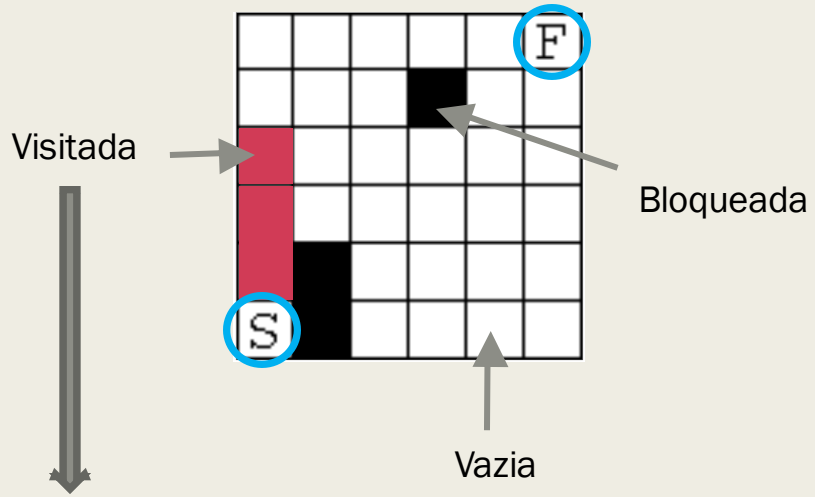
Descrição do Jogo

- O jogo ***Unequal Length Mazes*** consiste num tabuleiro de dimensões arbitrárias com algumas das suas células bloqueadas (obstáculos).
- O objetivo do jogo é encontrar um caminho desde a célula de partida até à célula de chegada que passe por todas as células disponíveis (não bloqueadas) apenas uma vez.
- O caminho deve alternar entre segmentos horizontais e verticais e dois segmentos consecutivos não podem ter o mesmo tamanho.



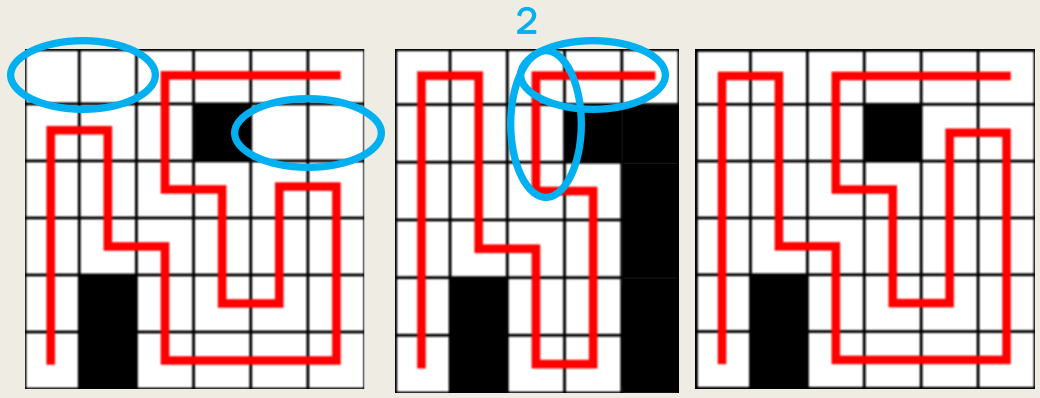
Formulação como Jogo Individual

- **Tipo de tabuleiro:** tabuleiro de dimensões arbitrárias, célula de partida no canto inferior esquerdo e célula de chegada no canto superior direito;
- **Tipo de peças:** células vazias, células bloqueadas e células visitadas;
- **Regras de movimento:** o jogador tem de alternar entre segmentos horizontais e verticais, sendo que dois segmentos consecutivos não podem ter o mesmo tamanho;



Próximo Movimento?
Continuar nesta direção ou posso trocar de direção

- **Condições para terminar jogo derrotado:** ser impossível de resolver, no sentido de não existir nenhum caminho possível que esteja de acordo com as regras do jogo;
- **Condições para terminar jogo com vitória:** chegar à célula do campo superior direito do tabuleiro, sendo que todas as células do tabuleiro que não eram bloqueadas foram visitadas e que com esse movimento final os segmentos consecutivos não ficaram com igual tamanho;
- **Pontuação:** o tempo gasto na resolução do puzzle penalizado pelo número de dicas utilizado (20 segundos por cada dica).



Exemplos de não vitória

Vitória

Formulação como Problema de Pesquisa

- **Representação do Estado:**
  - Matriz que representa o tabuleiro - b[*HEIGHT*][*WIDTH*]:
    - Célula Vazia (*EC*): ***0***
    - Célula de Partida *(VC):* ***8***
    - Célula com Obstáculo (*BC*): ***9***
    - Célula com a indicação da Direção a partir da qual foi visitada (UP | DOWN | LEFT | RIGHT): ***1 | 2 | 3 | 4***
  - Célula atual: ***Row, Col, Dir*** $\in$ *{Up, Down, Left, Right},* ***Length***
  - Comprimento do último segmento: ***LastLenght***

- **Estado Inicial:**
  - Tabuleiro vazio com obstáculos e célula inicial visitada.
  - Célula atual é a célula inicial (sem direção).
  - Não existe informação sobre o comprimento do segmento anterior.
  - Comprimento do segmento atual é zero.

- **Teste Objetivo:**
  - Tabuleiro com obstáculos e as restantes células todas visitadas.
  - Célula atual é a célula final.
  - Todos os segmentos consecutivos têm tamanhos diferentes.

- **Heurísticas:**
  - (1) Inverso da distância de Manhattan à célula final.
  - (2) Soma do peso das células vazias.*
  - (3) Dobro do número de células vazias.*

- **Operadores:**

Nome	Pré-condições	Efeitos	Custos
Up	Row > 0 Dir = Up b[Row-1][Col] = 0	Row -= 1 b[Row][Col] = UP Lenght++	1
Down	Row < HEIGHT - 1 Dir = Down b[Row+1][Col] = 0	Row += 1 b[Row][Col] = DOWN Lenght++	1
Left	Col > 0 Dir = Left b[Row][Col-1] = 0	Col -= 1 b[Row][Col] = LEFT Lenght++	1
Right	Col < WIDTH - 1 Dir = Right b[Row][Col+1] = 0	Col += 1 b[Row][Col] = RIGHT Lenght++	1
SwapToUp	Length != 0 Length != LastLenght Dir=Left v Dir = Right	Dir = Up LastLenght = Lenght Lenght = 0	2
SwapToDown	Length != 0 Length != LastLenght Dir=Left v Dir = Right	Dir = Down LastLenght = Length Lenght = 0	2
SwapToLeft	Length != 0 Length != LastLenght Dir=Up v Dir = Down	Dir = Left LastLenght = Lenght Lenght = 0	2
SwapToRight	Length != 0 Length != LastLenght Dir=Up v Dir = Down	Dir = Right LastLenght = Lenght Lenght = 0	2

* Nestas heurísticas são ainda identificadas células vazias encurraladas que tornam impossível a resolução do problema a partir deste estado.

Implementação (1)

- Estruturas de Dados:

```python
class Node:
  def __init__(self, state, depth, cost, heuristic, parent):
    self.state = state
    self.depth = depth
    self.cost = cost
    self.heuristic = heuristic
    self.parent = parent
```

Árvore de Pesquisa com Nós (contêm informação sobre o estado, profundidade, custo, heurística e antecessor).

```python
class SearchProblem:
  def __init__(self, initState, isFinalState, checkVisited=False):
    self.initState = deepcopy(initState)
    self.queue = [Tree.Node(initState, 0, 0, 0, -1)]
    self.isFinalState = isFinalState
    self.visited = []
    self.checkVisited = checkVisited
```

Fila de Prioridade (o critério de inserção dos nós na fila, já ordenada, varia em função do algoritmo).

- Estado Inicial e Teste Objetivo:

```python
def setInitState(n):
  global initState, H, W

  initBoard = boards.initBoards[n]
  H = len(initBoard)
  W = len(initBoard[0])
  currentCell = (H-1, 0, None, 0)
  lastSegment = None
  initState = (initBoard, currentCell, lastSegment)
```

Inicialização do Estado Inicial.

```python
def isFinalState(state):
  (board, (row,col,dir,length), lastSegment) = state
  for line in board:
    for cell in line:
      if cell == EC:
        return False
  return row == 0 and col == W-1 and length != lastSegment
```

Função de Verificação de Estado Final.

Implementação: Operadores (2)

```python
def move(state, direction):
    (board, (row,col,dir,length), lastSegment) = deepcopy(state)
    step = propDir[direction]['step']
    row += step[0]
    col += step[1]
    if withinBoard(row, col) and dir == direction and board[row][col] == EC:
        board[row][col] = propDir[direction]['value']
        length += 1
        return {"state": (board, (row,col,dir,length), lastSegment), "cost": 1}
    return False
```

Visitar a próxima célula vazia do tabuleiro na direção igual à atual ("andar em frente").

```python
propDir = {
    'up': {"step":(-1, 0), "value": UP},
    'down': {"step":(1, 0), "value": DOWN},
    'left': {"step":(0, -1), "value": LEFT},
    'right': {"step":(0, 1), "value": RIGHT}
}

def withinBoard(row, col):
    return row >= 0 and row < H and col >= 0 and col < W

def canSwap(nextDir, prevDir):
    if nextDir == 'up' or nextDir == 'down':
        return prevDir == 'left' or prevDir == 'right'
    elif nextDir == 'left' or nextDir == 'right':
        return prevDir == 'up' or prevDir == 'down'
    return False
```

Estrutura e Funções Auxiliares para a implementação dos operadores.

```python
def swap(state, direction):
    (board, (row,col,dir,length), lastSegment) = deepcopy(state)
    if ((length != lastSegment and length != 0 and canSwap(direction, dir)) or lastSegment == None):
        dir = direction
        lastSegment = length
        length = 0
        return {"state": (board, (row,col,dir,length), lastSegment), "cost": 2}
    return False
```

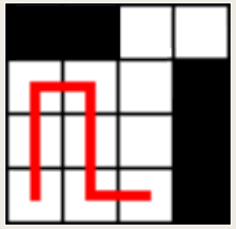
Trocar de direção (vertical para horizontal e vice-versa), desde que válido: segmentos consecutivos não têm igual tamanho.

```python
def newTransitions(node: Tree.Node, heuristic):
    '''Find reachable states from node\'s state and return nodes correspondig to those states.'''
    state = node.state

    transitionsMoves = [move(state, direction) for direction in propDir.keys()]
                       + [swap(state, direction) for direction in propDir.keys()]
    transitionsMoves = list(filter(lambda state : state, transitionsMoves))

    transitions = []
    for movement in transitionsMoves:
        newState = movement["state"]
        newCost = movement["cost"]
        newHeuristic = heuristics(newState, heuristic)
        transitions.append(Tree.Node(newState, node.depth+1, node.cost+newCost, newHeuristic, node))
    return transitions
```

Função que cria novos nós da árvore por aplicação dos operadores

Implementação: Heurísticas (3)

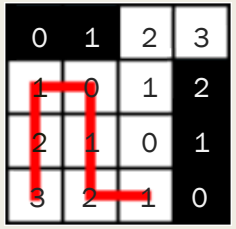
(1) **Inverso da distância de Manhattan** da célula atual à célula final	(2) **Soma do peso das células vazias** (com poda de estados com células encurraladas)	(3) **Dobro do número de células vazias** (com poda de estados com células encurraladas)
Visitar primeiro as células mais longe da célula final, diminuindo a probabilidade de nos aproximarmos da célula final com células ainda por visitar.	Completar a diagonal inferior antes da diagonal superior	Prosseguir na pesquisa com estados com maior número de células já visitadas (caminhos já mais explorados)



Célula Atual: (3, 2)
Célula Final: (0, 3)
Distância: |0-3| + |3-2|

h() = 1/4 = 0.25

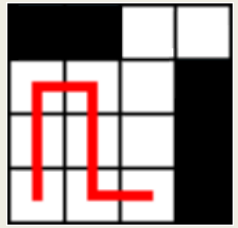
```
86  def manhattan(x1, y1, x2, y2):
87      return abs(x2-x1) + abs(y2-y1)
88
89  (board, (row, col, d, l), last) = state
90  if type == 1:
91      dist = manhattan(row, col, 0, W-1)
92      if dist == 0: return 0
93      else: return 1/dist
```



Soma dos pesos:
= 0 + 1 + 2 + 3
= 6

h() = 6

```
94  elif type == 2:
95      value = 0
96      for i in range(len(board)):
97          for j in range(len(board[i])):
98              if (board[i][j] == EC):
99                  value += abs(i-j)
100                 if (i != 0 and j != W-1) and emptyAdjacents(state, i, j) < 2:
101                     return 99999
102     return value
```

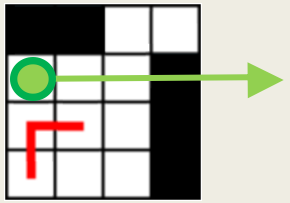


Células Vazias: 3
(excluindo célula final)
Fator Multiplicativo: 2

h() = 3*2 = 6

```
104 elif type == 3:
105     emptyCells = 0
106     for i in range(len(board)):
107         for j in range(len(board[i])):
108             if (board[i][j] == EC and (i != 0 and j != W-1)):
109                 emptyCells += 1
110                 if emptyAdjacents(state, i, j) < 2:
111                     return 99999
112     return emptyCells * 2
```

- **Poda de estados com células encurraladas:**
  - Reconhecer estados que não conduzem de certeza à solução: atribui-se uma heurística de valor elevado de forma a não serem explorados.
  - Uma célula vazia é considerada encurralada se não tiver pelo menos duas células adjacentes vazias.*



h() = 99999

*célula final ignorada e célula atual considerada vazia para efeitos de cálculo

Implementação: Algoritmos (4)

```python
def search(self, newTransitions, algorithm, heuristic=0, limit=-1):
  totalNodesVisited = 0
  while True:
    if not self.queue:
      print("No solution found!")
      return (None, totalNodesVisited)

    currentNode = self.queue.pop(0)
    totalNodesVisited += 1
    currentState = currentNode.state

    if self.checkVisited: self.visited.append(str(currentState))
    if self.isFinalState(currentState): break
    if (algorithm == algorithmTypes["depth_limited"] and currentNode.depth >= limit): continue

    currTransitions = newTransitions(currentNode, heuristic)
    if self.checkVisited:
      currTransitions = list(filter(lambda transition : str(transition.state)
                                    not in self.visited, currTransitions))

    for transition in currTransitions:
      self.queue.insert(self.getInsertPosition(algorithm, transition), transition)

  path = self.getPath(currentNode)
  return (path, totalNodesVisited)
```

Algoritmo genérico de pesquisa da solução: até encontrar o estado final, calcula todos os descendentes do estado atual e insere-os na fila de prioridade com o auxílio das funções à direita.*

*Uma vez que neste problema não há estados repetidos não foram guardados os estados visitados de forma a melhorar a performance ('*checkVisited*' é sempre *False*).

```python
def lessThanNode(self, algorithm, node1, node2):
  if algorithm == algorithmTypes["breadth"]:
    return node1.depth < node2.depth
  elif algorithm == algorithmTypes["depth"]:
    return -node1.depth < -node2.depth
  elif algorithm == algorithmTypes["depth_limited"]:
    return -node1.depth < -node2.depth
  elif algorithm == algorithmTypes["iterative_deepening"]:
    return -node1.depth < -node2.depth
  elif algorithm == algorithmTypes["uniform"]:
    return node1.cost < node2.cost
  elif algorithm == algorithmTypes["greedy"]:
    return node1.heuristic < node2.heuristic
  elif algorithm == algorithmTypes["A*"]:
    return node1.cost + node1.heuristic < node2.cost + node2.heuristic

def getInsertPosition(self, algorithm, node):
  if not self.queue: return 0
  lower, higher = 0, len(self.queue)-1
  while lower <= higher:
    mid = (lower + higher) // 2
    if self.lessThanNode(algorithm, self.queue[mid], node):
      lower = mid + 1
    elif self.lessThanNode(algorithm, node, self.queue[mid]):
      higher = mid - 1
    else:
      return mid
  return lower
```

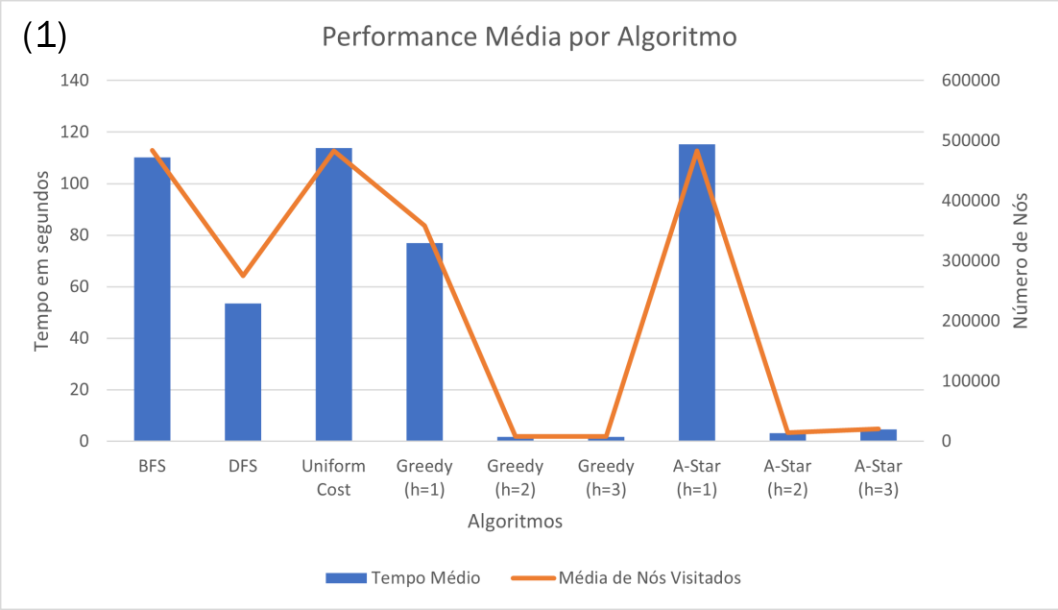
Obtenção do **índice para inserção na fila de prioridade** com pesquisa binária segundo o critério apropriado para cada algoritmo.

- Pesquisa em Largura;
- Pesquisa em Profundidade;
- Pesquisa em Profundidade Limitada;
- Aprofundamento Progressivo;
- Pesquisa de Custo Uniforme;
- Pesquisa Gananciosa;
- Algoritmo de Pesquisa A*.

Resultados (1)

Análise da média obtida em tempos de execução e número de nós explorados para 27 cenários diferentes (27 puzzles).

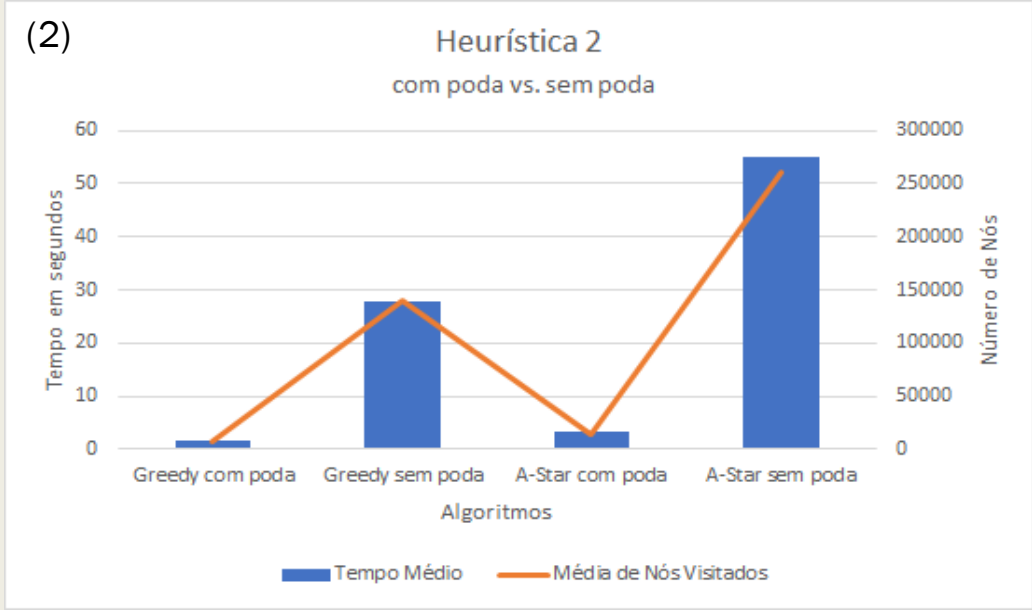
(1)



Tendo o problema uma única solução, constatamos que é sempre melhor continuar o **caminho atual mais explorado**, em vez de analisar os vários caminhos a tomar em cada ponto de decisão.

Tal explica a **performance do *DFS*** comparando-a com a **performance do *BFS***, apresentada no gráfico (1).

(2)

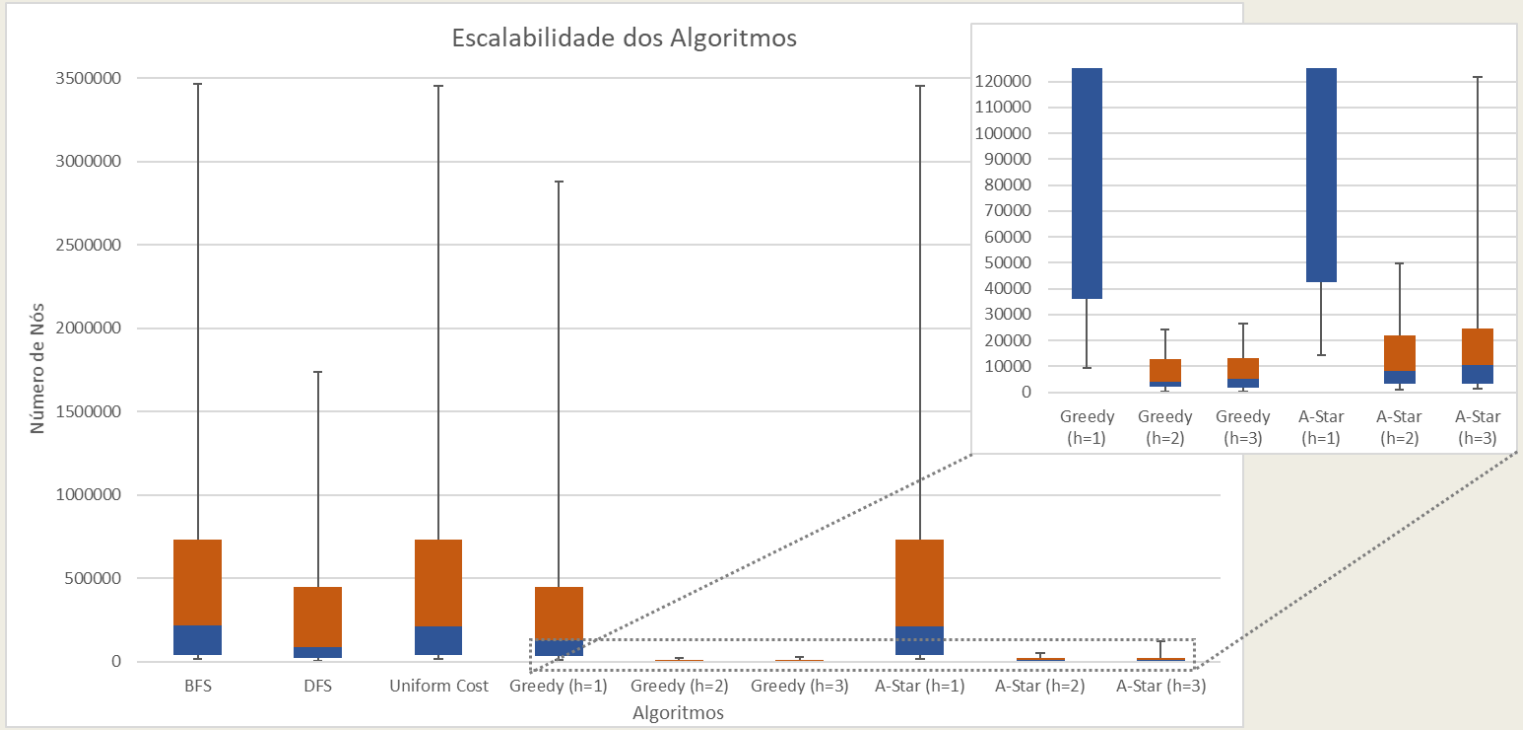


Os **Algoritmos *Greedy* e *A* com* heurísticas 2 e 3**, utilizam o mesmo princípio do *DFS*, explorando estados de pesquisa mais avançados, com a **adição de um critério de poda** que deteta antecipadamente caminhos que não levam à solução.

Este aprimoramento tem elevado impacto na performance e eficiência dos algoritmos referidos. Como se verifica no gráfico (2).

Resultados (2)

Análise da escalabilidade dos algoritmos para os 27 cenários diferentes (27 puzzles).



A **performance** pode **variar muito com o puzzle** utilizado, independentemente do algoritmo, de forma por vezes não determinística.

A performance é influenciada pelo número e localização de obstáculos mas o seu escalamento é mais significativo com o aumento do tamanho do tabuleiro.

Conclusões e Referências

- Os métodos de pesquisa mais convencionais, por exemplo, *BFS* e *DFS,* mostraram-se pouco escaláveis a problemas de maior dimensão e daí a **importância de utilizar heurísticas que permitam melhorar a eficiência na obtenção da solução**.
- Com este problema chegamos à conclusão de que é **necessário conhecer bem o jogo**, bem como as principais estratégias para o vencer para ser possível determinar heurísticas que produzam bons resultados.
- Neste jogo, existe apenas uma solução e os custos dos operadores são semelhantes, tornando-se complicado encontrar boas heurísticas admissíveis e otimistas. Isto deriva, também, do facto de uma transição de estado neste problema ter pouco significado quando vista de forma isolada.
- Assim, mesmo com alguma imprevisibilidade causada pela possibilidade de diferentes puzzles terem soluções muito distintas, os melhores resultados são obtidos quando todo o puzzle é avaliado e é dada uma menor importância aos custos que levam até ao estado corrente.

- **Trabalhos relacionados e referências:**
  - *Pathfinding e resolução de labirintos com algoritmo de pesquisa A**
  - *Algoritmo de pesquisa A* para resolução de labirintos*
  - *"Rato no Labirinto" (jogo semelhante)*