U.PORTO
FEUP ENGENHARIA

Match the Tiles

FEUP – Inteligência Artificial 2020/21

Grupo 34

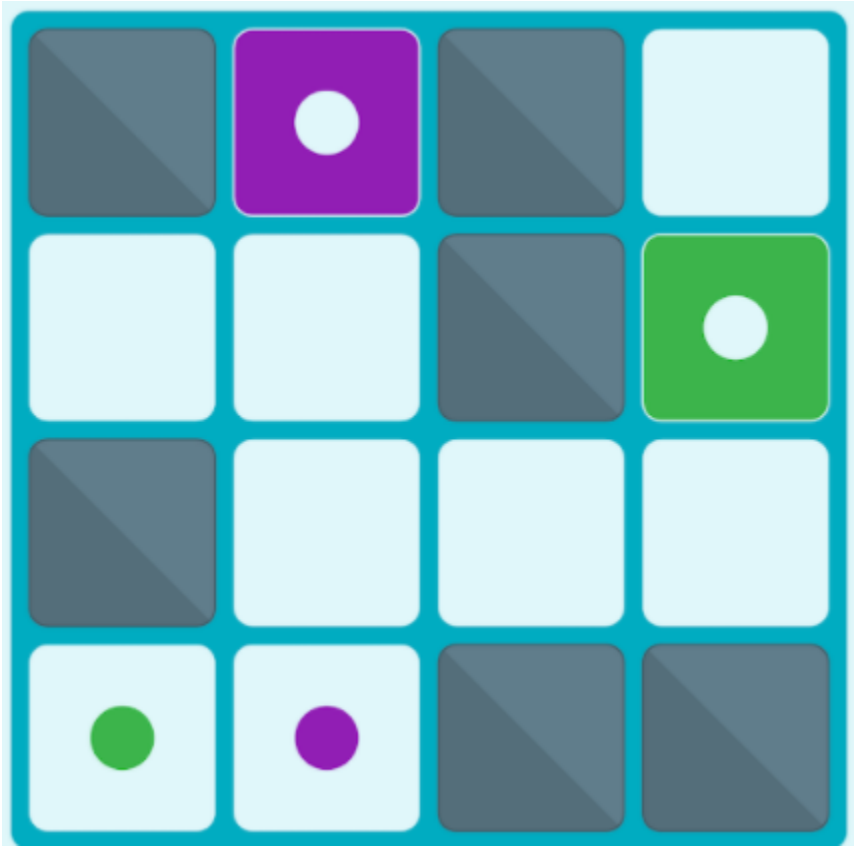
Caio Nogueira - up201806218
Carlos Lousada - up201806302
Miguel Silva - up201806388

Match the Tiles

Match the Tiles é um jogo de um jogador no qual o objetivo consiste em colocar as diferentes peças (tiles) coloridas nas respetivas células objetivo. Para tal o jogador pode mexer as suas peças (simultaneamente) nas 4 direções possíveis. Ao serem movidas, as peças deslocam-se na direção desejada até que embatam contra um obstáculo ou atinjam o limite do tabuleiro.

O objetivo do programa a desenvolver é resolver puzzles deste tipo (4x4, 5x5, 6x6) usando o menor número de jogadas possível.

Para tal, serão exploradas diferentes algoritmos de pesquisa, de modo a compará-los quanto à sua eficiência.



Formulação do problema

- **Representação do estado**

Matriz 4x4, 5x5 ou 6x6, preenchida com pares que representam o tipo (ou ausência de peça presentes): ["grass","empty"] para uma célula vazia, ["block","block"] para um obstáculo, ["grass","color1piece"] para uma célula de cor "color1", ["color1center","empty"] para uma célula que contem o objetivo da célula de cor "color1", ["color1center", "color1piece"] para uma célula com cor "color1" que se encontra na célula objetivo

- **Estado inicial**

Matriz bidimensional de dimensão N (4 <= N <= 6): Board[N,N,2] com as células na disposição inicial.

- **Estado Objetivo**

Matriz bidimensional de dimensão N (4 <= N <= 6): Board[N,N,2] em que as células se encontrem na distribuição desejada, ou seja, todas as peças se encontrem nas mesmas coordenadas do respetivo objetivo.

- **Operadores**

Os jogadores apenas de movem nas 4 direções, como tal os operadores são move_up(), move_down(), move_left(), move_right().

Formulação do problema

- Pré-condições - assumindo que o tabuleiro tem dimensão NxN e que as peças coloridas estão em (x1,y1), (x2,y2), ...

Nome	Pré-condições	Efeitos
Left	(x1 > 0 /\ (B[x1-1][y][1] = "empty")) \/ (x2 > 0 /\ B[x2-1][y2][1] == "empty") \/ (...)	B[x1][y1][1] = "empty"; B[x1-d][y1][1] = "colorPiece"; (...)
Right	(x1 < N /\ (B[x1+1][y][1] != "empty")) \/ (x2 < N /\ B[x2+1][y2][1] != "empty") \/ (...)	B[x1][y1][1] = "empty";B[x1+d][y1][1] = "colorPiece"; (...)
Up	(y1 > 0 /\ (B[x1][y-1][1] != "empty")) \/ (Y2 > 0 /\ B[x2][y2-1][1] != "empty") \/ (...)	B[x1][y1][1] = "empty";B[x1][y-d][1] = "colorPiece"; (...)
Down	(y1 < N /\ (B[x1][y+1][1] != "empty")) \/ (y2< N /\ B[x2][y2+1][1] != "empty") \/ (...)	B[x1][y1][1] = "empty";B[x1][y+d][1] = "colorPiece"; (...)

Dado que o objetivo será minimizar o número de jogadas para resolver o puzzle, o custo será unitário para todos os operadores.

Se nenhuma das peças coloridas satisfizer as pré-condições, o operador será inválido e a ação não é executada. Caso contrário, a operação é válida e será efetuada.

Formulação do problema

Heurísticas utilizadas:

Foram desenvolvidas três heurísticas para serem usadas pelo Greedy Search e A* algorithm: simple heuristic, complex heuristic e fast heuristic.

De modo a estimar a distância de cada estado ao estado objetivo, será usada como função de avaliação a soma do número mínimo de jogadas até atingir o objetivo de cada peça.

Deste modo, a função de avaliação complexa (complex heuristic) atribui pontos de penalização caso as peças se encontre em linhas/colunas diferentes da solução:

- Se a peça estiver na mesma linha e mesma coluna, a penalização será nula;
- Se a peça estiver na mesma linha e colunas diferentes ou vice-versa, a penalização será de 1 ou 2 pontos, dependendo da posição do centro.
- Se a peça estiver em linhas e colunas diferentes, a penalização será de 2 ou 3 pontos, dependendo da posição do centro.

Caso a peça se encontre na mesma linha/coluna do seu destino, é necessário verificar também a existência de obstáculos, a qual será penalizada em 3 pontos.

```
def admissible_heuristics(self):
    gameStateBoard = self.board
    piece_points = []
    for pieces in self.grouped_pieces:
        points_list = []
        permutations = self.permutations[tuple(pieces)] #  *Finds best combination between pieces and respective centers*
        for permutation in permutations:
            list_aux = []
            for piece, center in permutation:
                pointsAux = 0
                if center[0] == piece.x: pointsAux += check_obstaclesX(piece, center[0], gameStateBoard.board)
                if center[1] == piece.y: pointsAux += check_obstaclesY(piece, center[1], gameStateBoard.board)
                pointsAux += estimateCenterDifference(piece, center, gameStateBoard.board)
                list_aux.append(pointsAux)
            points_list.append(max(list_aux))

        piece_points.append(min(points_list))

    return max(piece_points) #*evaluates only the worst piece (otherwise it would not be optimistic)
```

Algoritmos implementados

Foram implementados todos os algoritmos de pesquisa dados nas aulas, ou seja, BFS (breadth-first-search), DFS (depth-first-search), Iterative Deepening, Greedy Search e A*.

```
def greedy_search(self, generate = False):
    self.dfs_visited = []
    start_node = copy.deepcopy(self)
    current_node = copy.deepcopy(self)
    starttime = datetime.now()
    while True:
        self.dfs_visited.append(current_node.board.board)
        if current_node.board.check_game_over():
            solution = getPath(current_node.board, [])
            return solution
        timer = datetime.now()-starttime
        if(generate and (timer.microseconds // 1000) > 500):
            return None
        neighbours = [x for x in current_node.neighbours() if x.board.board not in self.dfs_visited]
        self.stats.operations+=len(neighbours)

        if len(neighbours) == 0:
            #self.dfs_visited.append(current_node.board.board)
            current_node.board = current_node.board.parentBoard #BACKTRACK
            continue

        if self.settings.heuristic == 1:
            current_node = min(neighbours, key = lambda x: x.simple_heuristics())
        elif self.settings.heuristic == 2:
            current_node = min(neighbours, key = lambda x: x.admissible_heuristics())
        elif self.settings.heuristic == 3:
            current_node = min(neighbours, key = lambda x: x.heuristics())

    return None
```

```
def iterative_deepening(self, gameState):
    depth = 1
    self.iddfs_result = None
    while True:     #BEWARE OF IMPOSSIBLE PUZZLES
        self.dfs_visited = []
        self.iddfs(gameState, depth)
        if self.iddfs_solution != None:
            result = getPath(self.iddfs_solution.board, [])
            return result
        depth += 2
        #print(depth)


def iddfs(self, gameState, depth):
    if (depth == 0):
        return None

    gameStateBoard = gameState.board
    self.dfs_visited.append(gameStateBoard.parentBoard)
    if gameStateBoard.board not in self.dfs_visited:
        neighbours = gameState.neighbours()
        self.stats.operations += len(neighbours)
        for neighbour in neighbours:
            if neighbour.board.check_game_over():
                self.iddfs_solution = neighbour
                return neighbour

            self.iddfs(neighbour, depth-1)
```

```
def dfs(self, gameState):
    gameStateBoard = gameState.board
    if gameStateBoard.board not in self.dfs_visited:
        self.dfs_visited.append(gameStateBoard.board)
        neighbours = gameState.neighbours()
        self.stats.operations += len(neighbours)
        for neighbour in neighbours:
            if neighbour.board.check_game_over():
                result = getPath(neighbour.board, [])
                self.dfs_result = result
                return result
            self.dfs(neighbour)
```

Algoritmos Implementados

Para a representação do jogo foram usadas as classes Piece, Center, Game e Board, sendo a board do jogo representada através de uma lista de listas. Usando estas classes os algoritmos de pesquisa tentam encontrar uma solução para o problema dado.

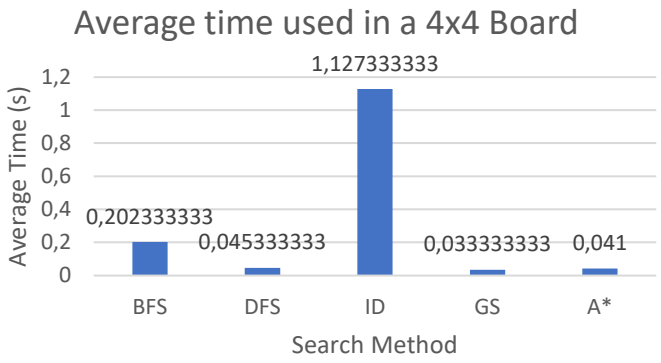
Para a implementação dos algoritmos foram usadas listas, mesmo para algoritmos que usam filas ou filas de prioridade na sua implementação, como é o caso do BFS e do A*, sendo que, nestes dois algoritmos, as listas são operadas de forma diferente.

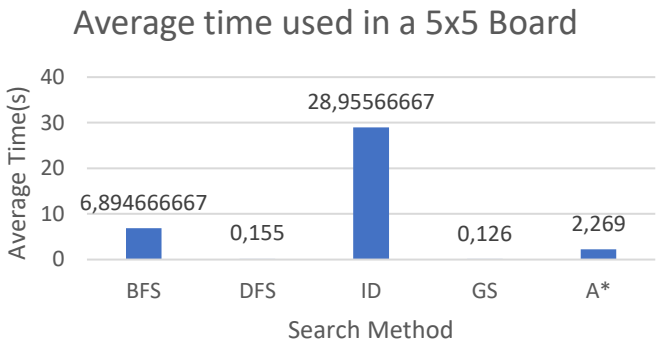
É de notar que, dado que o algoritmo de pesquisa DFS (depth-first-search) é recursivo, pode acontecer que, para puzzles de maior complexidade, este algoritmo não chegue a uma conclusão pois atinge o limite de recursão estipulado pelo Python.

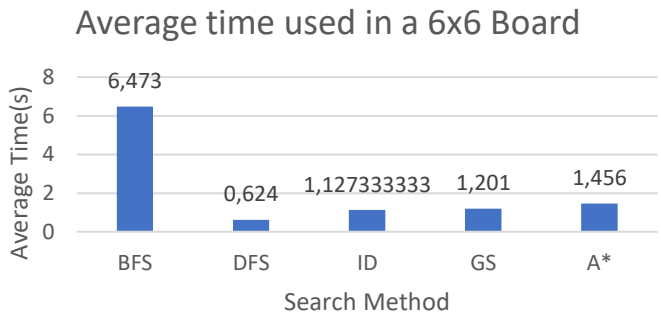
Foi também implementado um sistema de geração de puzzles aleatórios que utilizam o algoritmo Greedy Search, de modo a verificar se o puzzle é possível, assim como o algoritmo A* (com a “fast” heuristics) para verificar a complexidade do tabuleiro gerado.

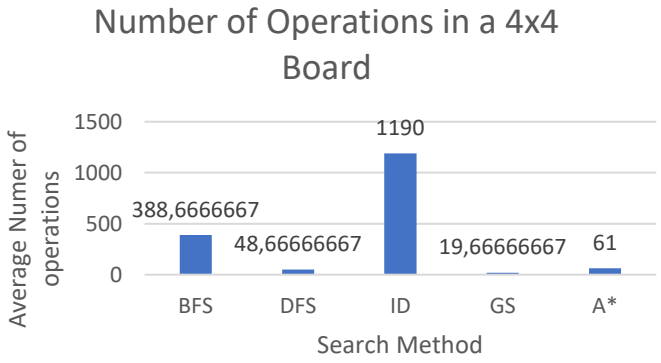
```
def a_star_search(self, limit = False):
    self.dfs_visited = []
    current = copy.deepcopy(self)
    frontier = [current]
    cost_so_far = {current: 0}
    self.nodes_expanded = 0
    self.stats.start_timer()

    while frontier:
        self.stats.update_timer()

        if (self.stats.ms > 150) and limit:
            return []


        if self.settings.heuristic == 1:
            current = min(frontier, key = lambda x: cost_so_far[x] + x.simple_heuristics())
        elif self.settings.heuristic == 2:
            current = min(frontier, key = lambda x: cost_so_far[x] + x.admissible_heuristics())
        elif self.settings.heuristic == 3:
            current = min(frontier, key = lambda x: cost_so_far[x] + x.heuristics())
```

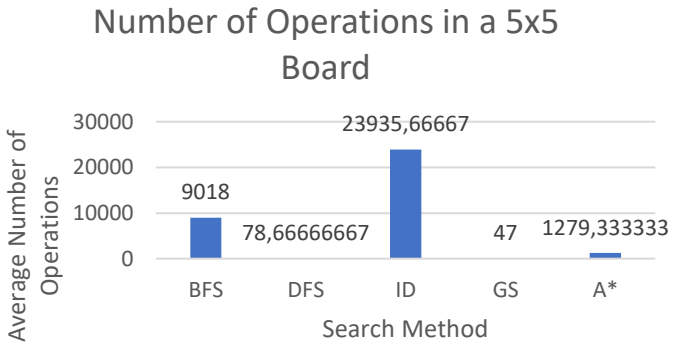
Resultados experimentais

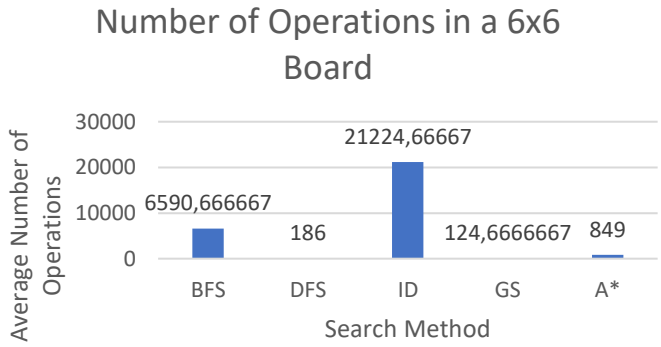






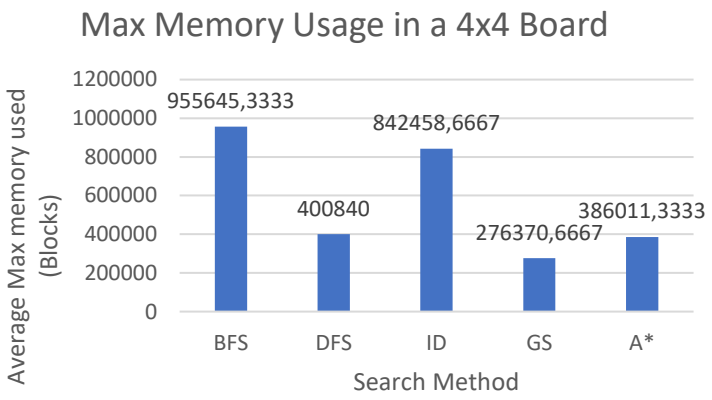


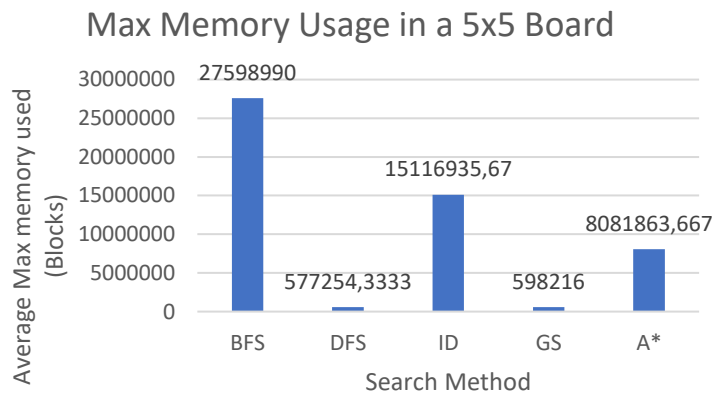


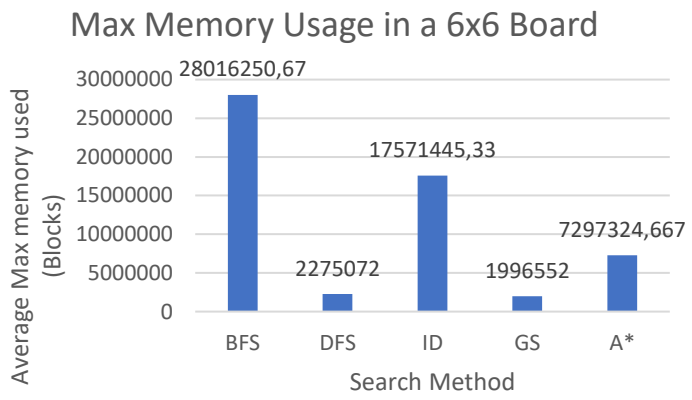


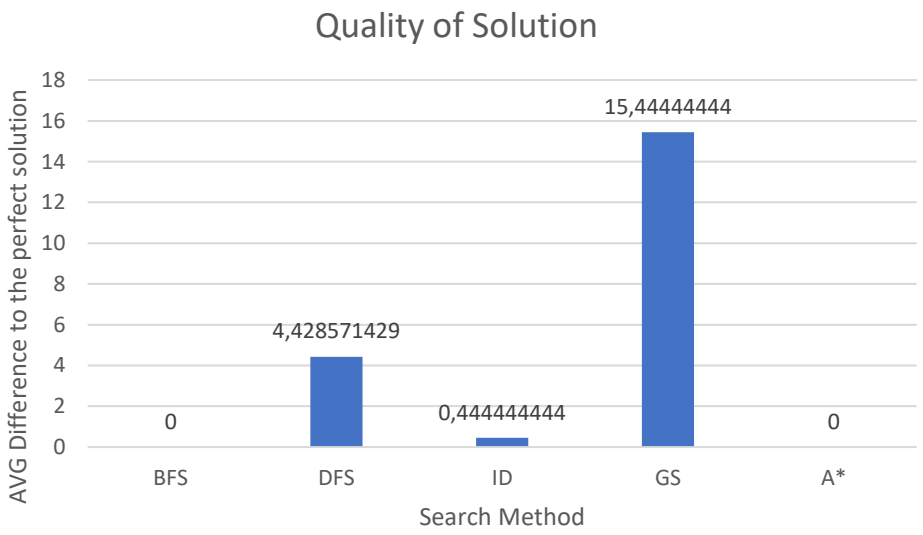
*Estes gráficos foram obtidos atráves do tratamento dos dados das tabelas enviadas em anexo (IART_grupo34_docs.zip)

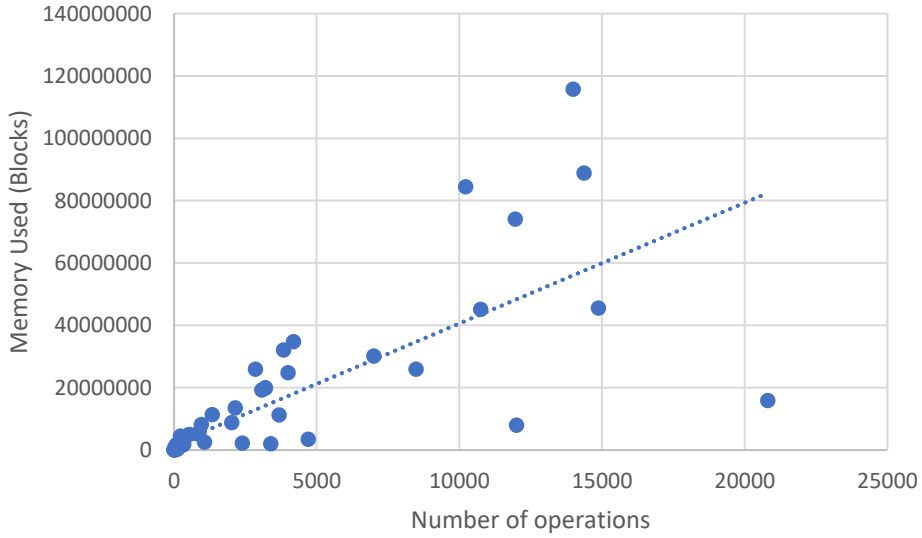
Resultados experimentais











Conclusões

Analisando os resultados experimentais, verifica-se, como seria de esperar, que o algoritmo Greedy Search é o mais rápido a encontrar uma solução, dado procurar uma solução baseando-se numa heurística, sem se preocupar com a sua optimalidade, sendo também o algoritmo que, em média, expande o menor número de nós, pelo que, consequentemente, utiliza menos memória.

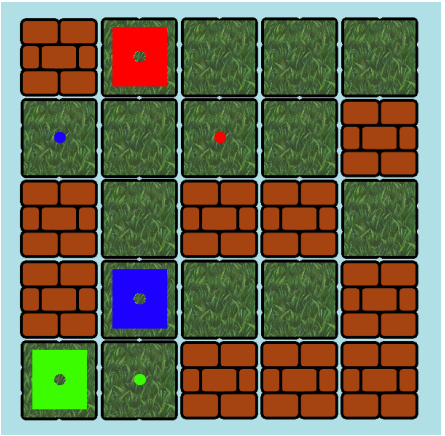
Para a tentativa de encontrar uma solução ótima para o problema dado, o algoritmo A* é, como seria de esperar, o melhor apesar do seu desempenho ser muito afetado pela heurística usada.

Quanto às heurísticas, muitas vezes, dependendo do puzzle em questão, a heurística mais complexa foi mais lenta a encontrar a solução do que a heurística mais simples devido ao elevado custo computacional inerente à consideração dos diversos cenários possíveis. Deste modo a melhor heurística em geral foi o algoritmo fast, sendo este um middle-ground entre a heurística mais simples e a complexa.

Relativamente aos uninformed-search algorithms, o DFS foi o que encontrou soluções em menos tempo e utilizando a menor quantidade de memória.

Para o desenvolvimento do projeto foi implementada a lógica do jogo, usando o Python como linguagem de programação. Para implementar a interface gráfica, foi usado o módulo pygame.

O ambiente de desenvolvimento utilizado foi o VSCode.



Referências

Durante o desenvolvimento do projeto, recorremos às seguintes páginas web:

- https://www.pygame.org/docs/
- https://www.geeksforgeeks.org/a-search-algorithm/
- https://pygame-menu.readthedocs.io/en/4.0.1/
- Material fornecido no Moodle