

# RELATORIO TRABALHO 1 – BUSCA

## Bibliotecas Utilizadas

As principais bibliotecas utilizadas foram:

- **Tkinter / Matplotlib:** Para a criação de interfaces gráficas e visualização de gráficos animados.
- **NetworkX:** Para a modelagem dos grafos, permitindo a manipulação e criação mais fácil.
- **NumPy:** Ajuda em algumas Funções matemáticas.

## Graph.py

O arquivo Graph.py implementa uma classe para modelar grafos com base em uma matriz de entrada, que no caso do trabalho representa o mapa e seus chãos. Utilizando a biblioteca NetworkX construímos um grafo e adicionamos seus vizinhos e peso de forma fácil, sendo os vizinhos todos os nós adjacentes e o peso o próprio valor na matriz (qualquer valor 0 da matriz é excluído do grafo)

```
rows, cols = self.matrix.shape # Pega o tamanho da matriz
for i in range(rows): # Para cada linha
    for j in range(cols): # Para cada coluna
        if self.matrix[i, j] != 0: # Se o valor da matriz for diferente de 0, 0 é água/parade
            self.G.add_node((i, j)) # Adiciona o nó ao grafo

for i in range(rows): # Para cada linha
    for j in range(cols): # Para cada coluna
        if self.matrix[i, j] != 0: # Se o valor da matriz for diferente de 0, 0 é água/parade
            if j + 1 < cols and self.matrix[i, j + 1] != 0: # Se a próxima coluna não for água
                # Adiciona a aresta entre o nó atual e o nó da próxima coluna
                self.G.add_edge(u_of_edge: (i, j), v_of_edge: (i, j + 1), weight=self.matrix[i, j + 1])
            if i + 1 < rows and self.matrix[i + 1, j] != 0: # Se a próxima linha não for água
                # Adiciona a aresta entre o nó atual e o nó da próxima linha
                self.G.add_edge(u_of_edge: (i, j), v_of_edge: (i + 1, j), weight=self.matrix[i + 1, j])
```

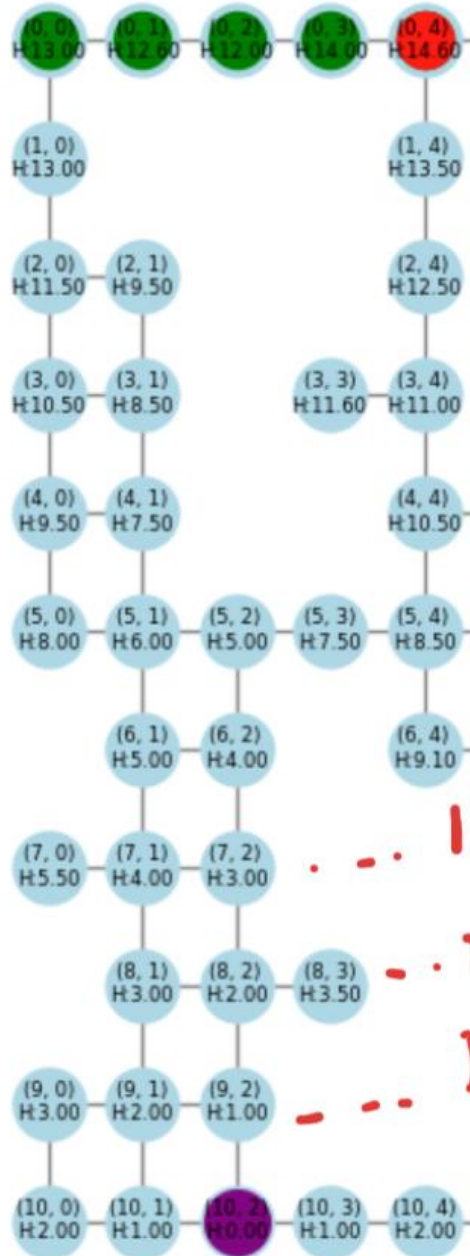
Também implementa o cálculo da heurística baseado no objetivo solicitado pelo usuário. O cálculo da heurística se dá em duas etapas, primeiro calculamos a quantidade de paredes que existem entre o objetivo e o nó atual na linha e coluna do nó atual, isso serve para evitar de ir para caminhos que até podem mais próximo do objetivo porém se existem paredes neles significa que eu irei precisar contornar elas, e isso gera mais custo.

Segue um exemplo para o nó (0, 4) com objetivo em (10, 2)

LINHA



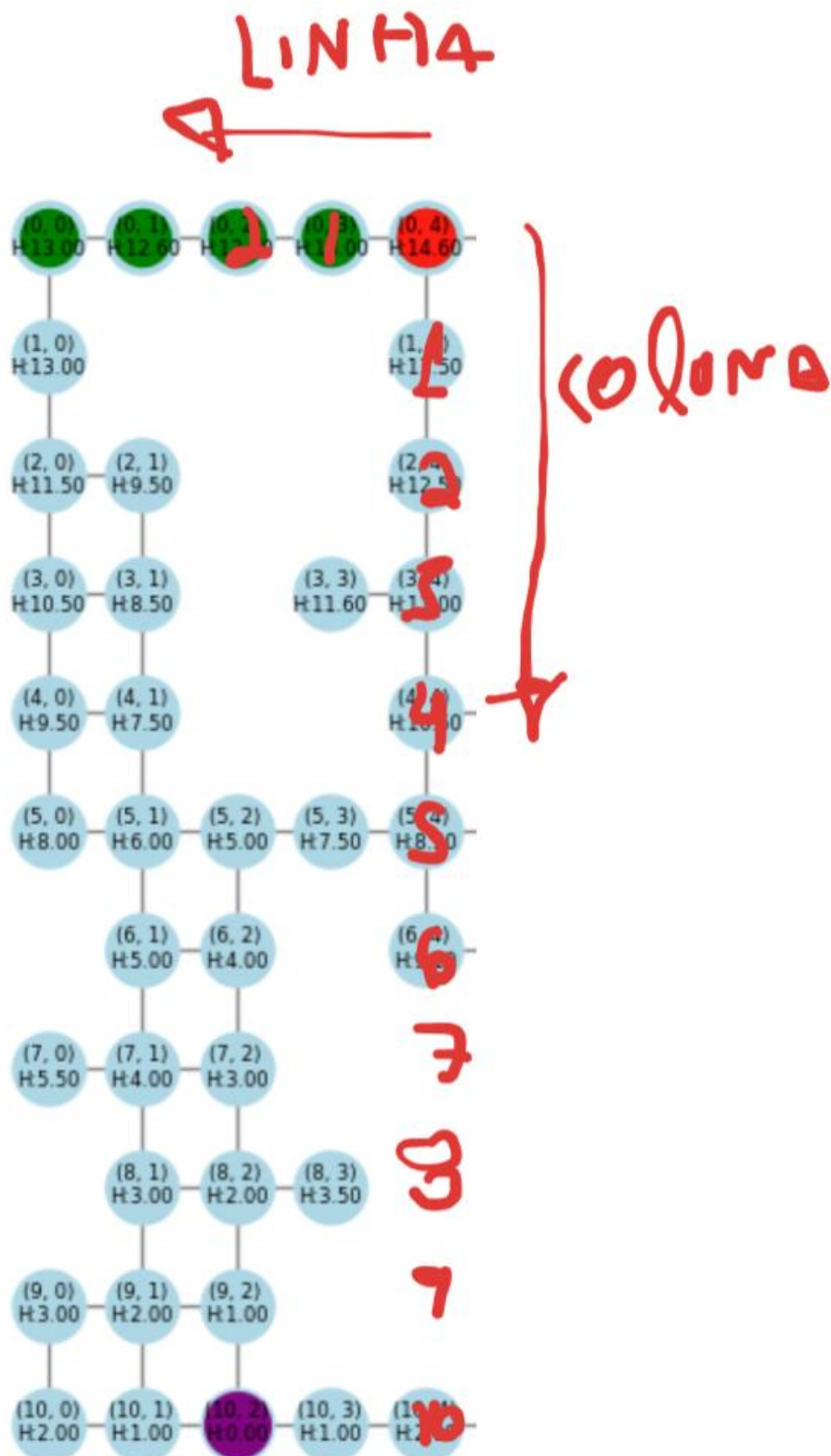
0



COLUNA

3

Depois calculamos a distância real daquele nó até o objetivo tanto pra linha quando pra coluna



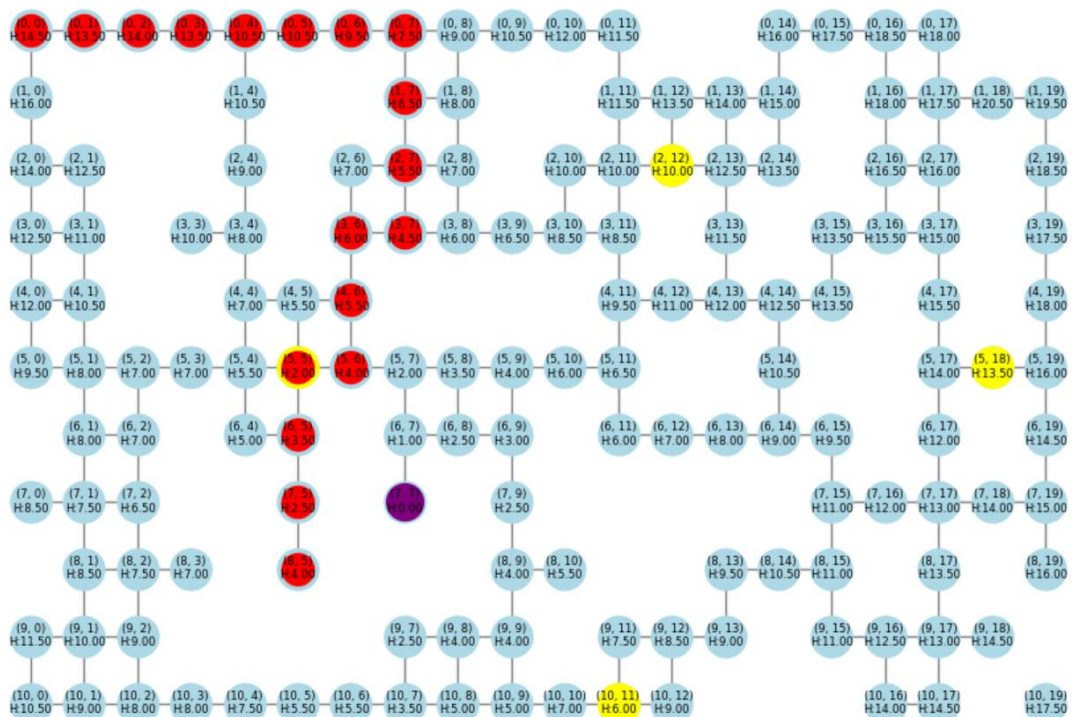
O primeiro cálculo então se dá por:

```
# Calcula a heurística para o nó
heuristic = (abs(i - goal[0]) + rows_water / 2) + (abs(j - goal[1]) + cols_water / 2)
```

O Peso das colunas é dividido por dois, pois são menos relevantes que o caminho real. Além de diminuir em -2 caso seja uma moeda para o algoritmo dar prioridade caso esteja passando.

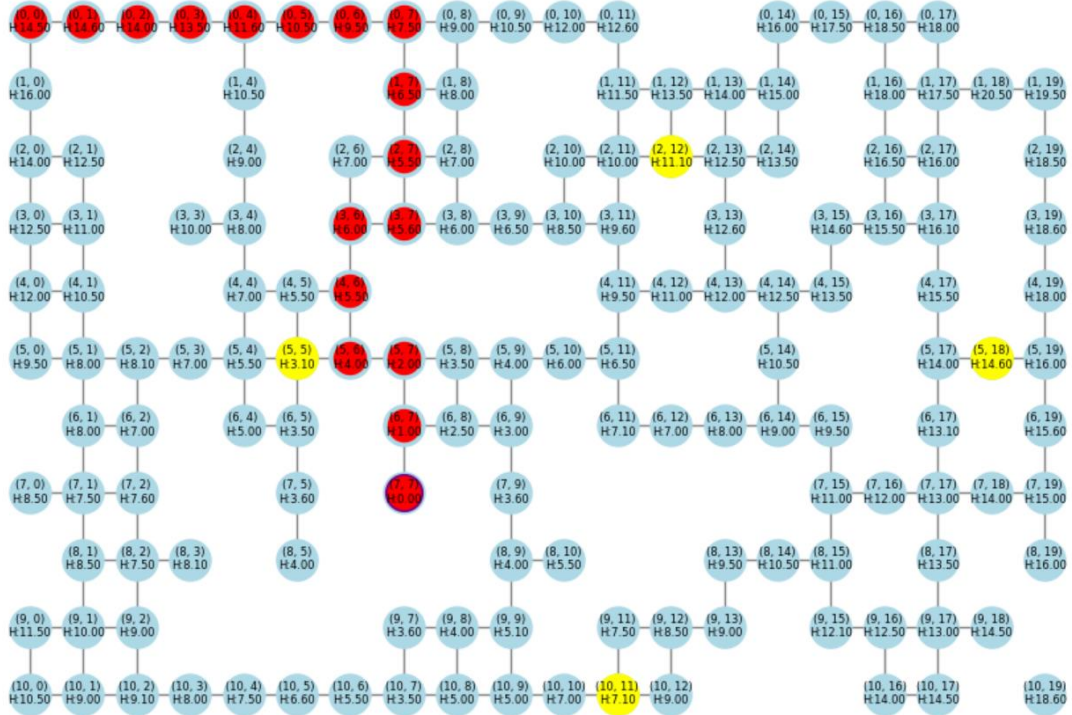
Depois vamos para a segunda etapa do cálculo da heurística e a mais importante. Nela para cada nó existem iremos verificar seus nós vizinhos, para ver se eles possuem uma heurística melhor que a do nó atual e se não existir aumentamos o custo da heurística do nó atual, isso é extremamente útil para evitar falsos caminhos promissores, pois um nó pode até ser melhor que outro, porém seus próximos vizinhos são todos piores que o vizinho do nó que deixou de ser escolhido, ou seja o algoritmo foi por um caminho que na hora era melhor, mas depois se tornou pior. Por exemplo na Busca Gulosa, no seguinte caso foi removido essa etapa do cálculo da heurística e vemos que quando o algoritmo chegou em (5, 6) ele podia ir tanto para (5, 4) ou (5, 7), ele foi pra (5, 5) pois foi o primeiro analisado mas se olharmos para as heurísticas dos vizinhos percebe-se que todos os vizinhos de (5, 5) tem heurísticas piores que ele, ou seja eles não eram um bom caminho, agora (5, 7) tem um vizinho com heurística melhor ou seja um caminho promissor.

Objetivo não alcançado.



Agora se adicionar essa etapa, o algoritmo consegue chegar numa solução

Objetivo alcançado!



Então ao final o cálculo da heurística fica

$$h'(n) = \begin{cases} \left( \left( |i - g_i| + \frac{\text{rows\_water}}{2} \right) + \left( |j - g_j| + \frac{\text{cols\_water}}{2} \right) - 2 \cdot \text{is\_coin}(n) \right) \cdot 1.1, & \text{se } \forall v \in V(n), h(v) \geq h(n) \\ \left( |i - g_i| + \frac{\text{rows\_water}}{2} \right) + \left( |j - g_j| + \frac{\text{cols\_water}}{2} \right) - 2 \cdot \text{is\_coin}(n), & \text{caso contrário.} \end{cases}$$

## Algoritmos

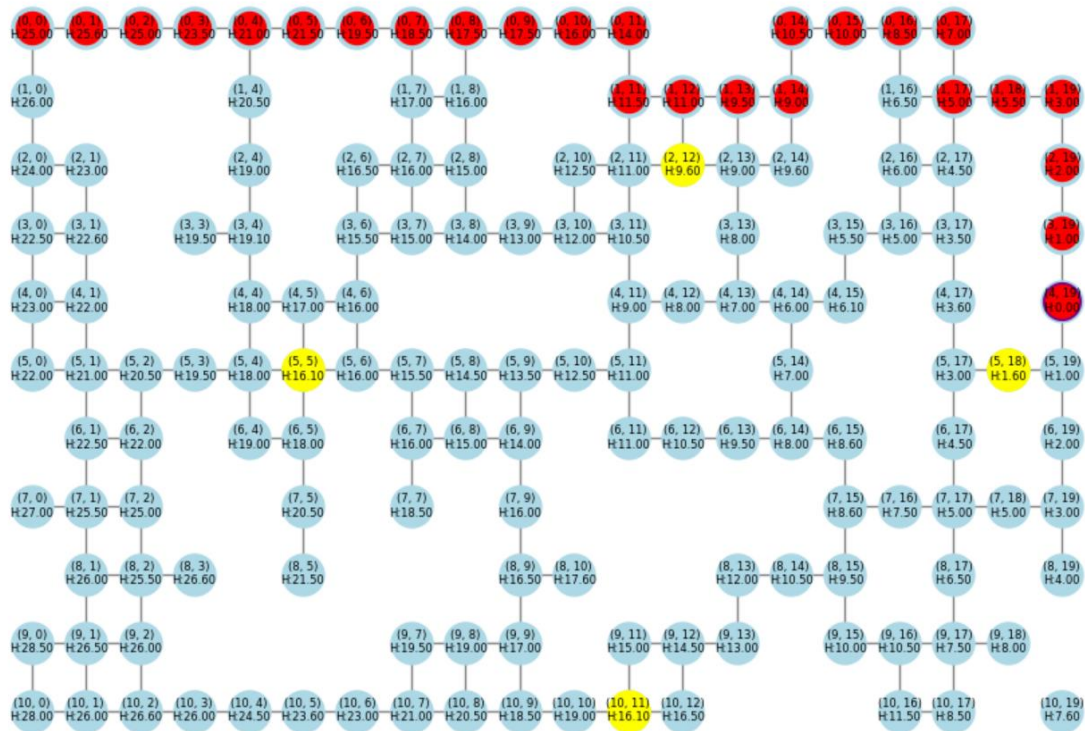
### Busca em Largura

Implementação clássica da busca em largura, com algumas pequenas modificações para ele trabalhar junto com os objetos da biblioteca networkx e a adição de uma variável 'pai' para rastrear o caminho mínimo.

Esse código sempre irá rodar no grafo todo, independente se já achou a solução e sempre retorna o caminho mínimo em relação a quantidade de nós, não leva em considerações os pesos ou heurísticas



Objetivo alcançado!

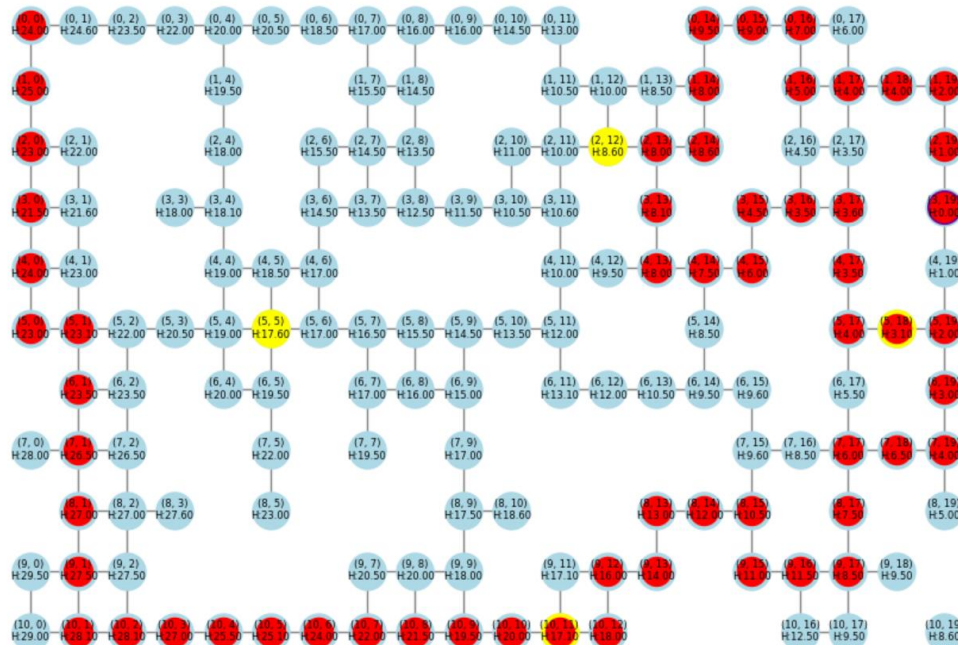


## Busca em Profundidade

Implementação clássica da busca em profundidade, com algumas pequenas modificações para ele trabalhar junto com os objetos da biblioteca networkx e a adição de uma variável 'pai' para rastrear o caminho mínimo.

Esse código roda até achar a solução, e sempre retorna um caminho, mas nem sempre é o mínimo. Não leva em considerações os pesos ou heurísticas

Objetivo alcançado!

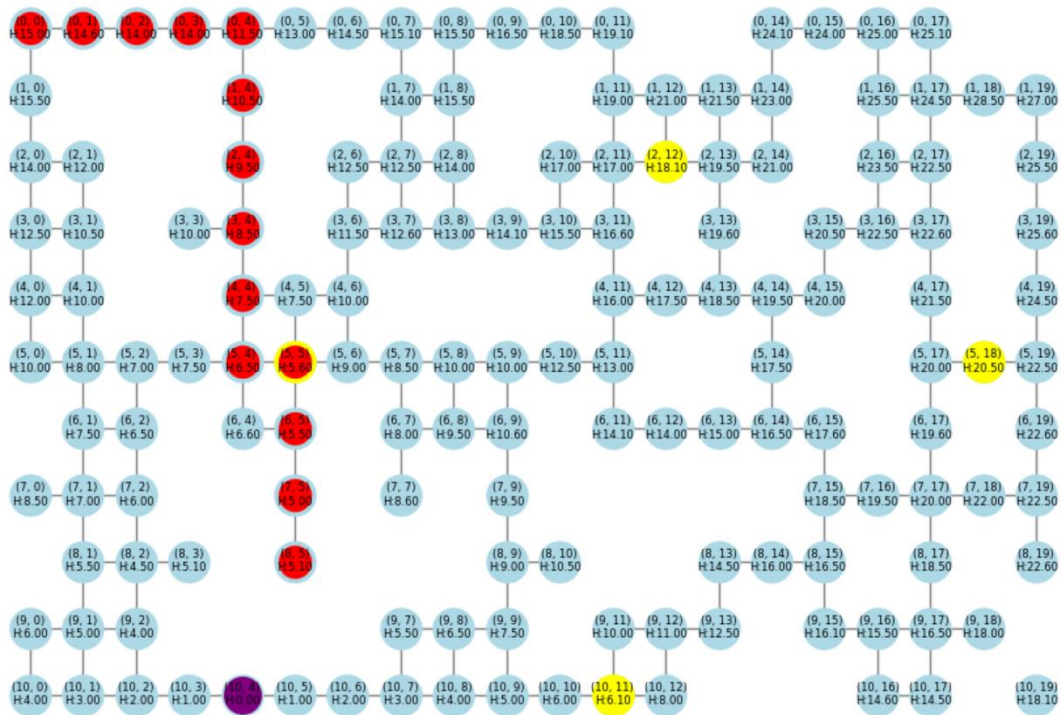


## Busca Gulosa

Foi implementado um algoritmo de busca gulosa, onde para cada vizinho do nó atual ele verifica o com menor heurística e vai até ele, foi adicionado uma lista de nós visitados para evitar loops infinitos, e caso ele entre num nó já visitado o algoritmo para e consideramos que não achamos a solução.

Esse código roda até achar a solução ou voltar para um nó já visitado, ele nem sempre acha uma solução, e leva em consideração somente a heurística dos nós

Objetivo não alcançado.



## Busca A\*

Foi implementando um algoritmo A\*, que utiliza uma fila com os nós atuais + os pesos acumulados até eles + seus heurísticas para determinar o melhor caminho

A cada execução a fila é ordenada para extrair o nó com menor peso. O nó extraído é colocado na lista de visitado e não será mais visitado

```
queue = sorted(queue, key=lambda x: x[1] + self.graph.G.nodes[x[0]]['heuristic'])
```

A seguinte parte do código é o que garante que vamos chegar num nó pelo melhor caminho, pois existem vários caminhos de chegar a um nó e o algoritmo pode até expandir ele uma primeira vez por um nó x, porem esse pode não ser o melhor e posteriormente ele vai expandir novamente esse nó por um nó y que é melhor. Porém esse nó já está na fila e não seria analisado novamente, para evitar isso ao analisar os vizinhos ele verifica se o peso para aquele vizinho é menor que aquele na fila, se sim ele adiciona esse novo caminho na fila também, os dois ficaram

na fila porem a função de cima garante que o menor será escolhido por primeiro sempre e mesmo que o peso acumulado desse novo nó é menor, se o somatório da heurística der maior que o nó que já estava na fila, então o nó que já estava na fila vai ser o visitado.

```
# Verifica os vizinhos
for neighbor in self.graph.G.neighbors(node):
    # Se o vizinho não foi visitado
    if neighbor not in visited:
        # Novo custo para o vizinho, soma do custo acumulado com o custo da aresta
        new_cost = cost + self.graph.G[node][neighbor]['weight']

        # Se for a primeira vez que encontramos o vizinho ou se o caminho é melhor (evita que o um nó na fila não seja atualizado)
        if neighbor not in self.pai or new_cost < self.pai[neighbor][1]:
            self.pai[neighbor] = (node, new_cost) # Armazena o pai e o custo

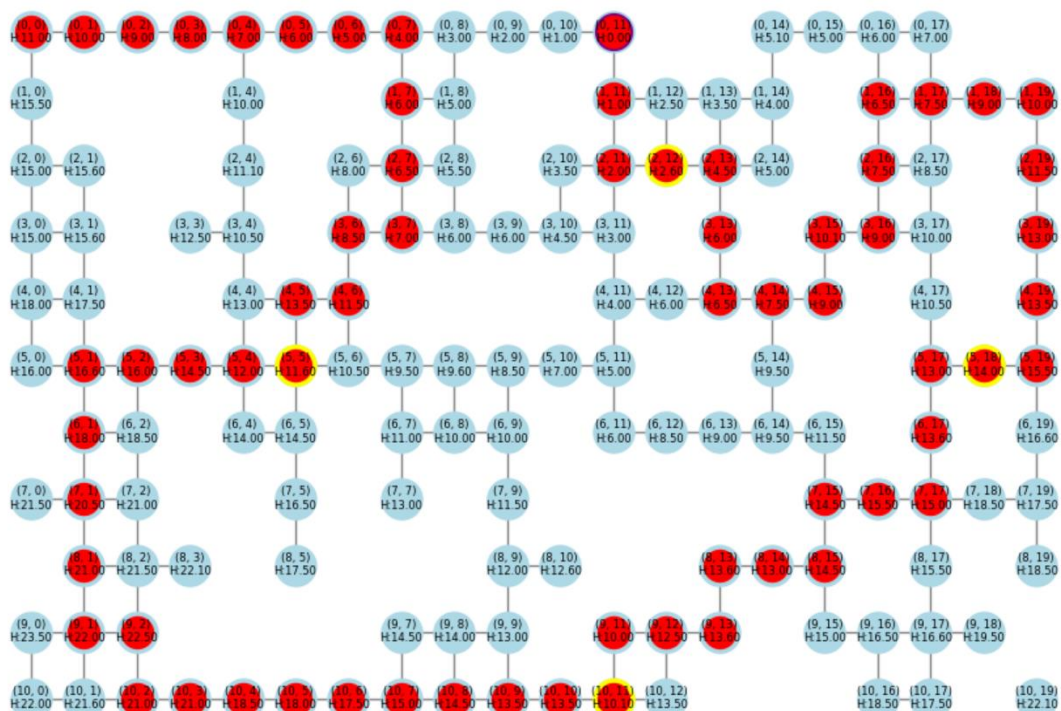
        # Adiciona o vizinho a fila
        queue.append((neighbor, new_cost)) # Pode adicionar um nó mais de uma vez, mas com custos diferentes, o menor custo é escolhido
```

E quando esse nó repetido ser tirado da lista a seguinte verificação garante que o algoritmo não vai repetir ele, pois ele já foi visitado.

```
if node not in visited:
    visited.add(node) # Adiciona o nó ao conjunto de visitados
    self.path.append(node) # Adiciona o nó ao caminho
```

E por fim tem a função para extrair o melhor caminho, usando a variável pai que guardou os antecessores de cada nó durante a execução.

Objetivo alcançado!





## relatorio.py

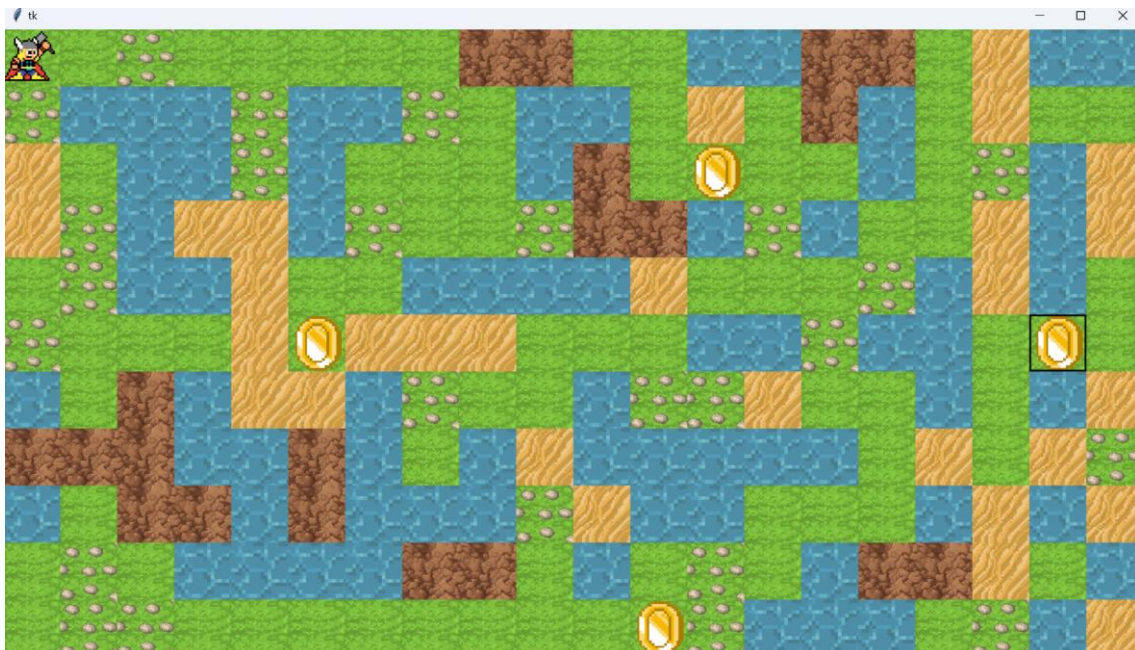
Uma classe que roda todos os algoritmos implementados e extrai as métricas de comparação.

## interface.py

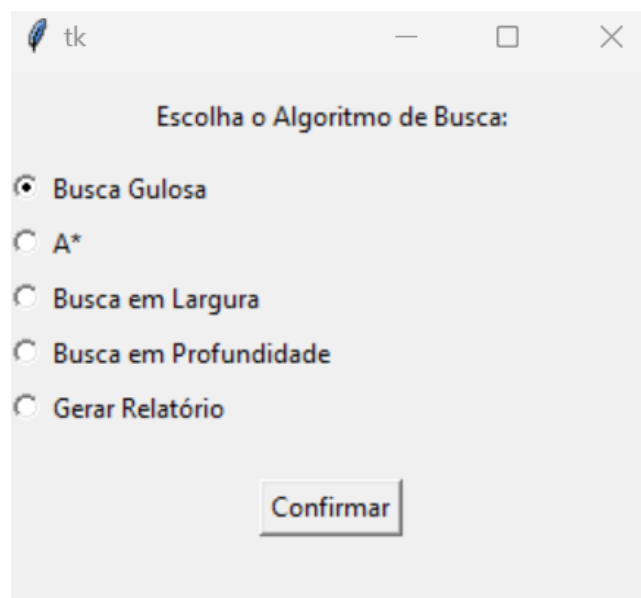
Classe responsável por gerar a interface inicial do mapa, onde o usuário poderá clicar em que posição ele quer ir e escolher o algoritmo que quer executar, ou caso ele queria pode gerar o relatório com as comparações dos algoritmos.

Cada bloco da interface é uma posição da matriz do txt, então é criado uma janela com essa quantidade de blocos e para cada bloco é aplicado uma textura de /imgs com base no valor da posição.

Também foi adicionado eventos de mouse para realçar o bloco selecionado pelo usuário e capturar o objetivo escolhido pelo usuário



Quando o usuário selecionado o objetivo é solicitado qual dos algoritmos ele deseja rodar, ou pode gerar o relatório



## graph\_animation.py

Classe responsável por gerar a animação de busca do grafo, usando a biblioteca networkx juntamente com a matplotlib é gerado uma animação de busca.

A biblioteca networkx tem funções prontas para desenhar os grafos, a seguinte linha de código seta as posições dos nós para se manterem igual o da matriz (isso para facilitar a visualização)

```
rows, cols = self.matrix.shape # Pega o tamanho da matriz
pos = {(i, j): (j, -i) for i in range(rows) for j in range(cols) if self.matrix[i, j] != 0} # Posição dos nós
node_labels = {node: f"{node}\nH:{self.G.nodes[node]['heuristic']:.2f}" for node in self.G.nodes()} # Labels dos nós
```

E as seguintes funções desenharam os grafos no plot

```
nx.draw(self.G, pos, ax=self.ax, with_labels=False, node_color="lightblue", edge_color="gray", node_size=500)
nx.draw_networkx_labels(self.G, pos, labels=node_labels, font_size=6, font_color="black")
nx.draw_networkx_nodes(self.G, pos, ax=self.ax, nodelist=coin_node, node_color="yellow", node_size=500)
nx.draw_networkx_nodes(self.G, pos, ax=self.ax, nodelist=[goal], node_color="purple", node_size=400)
```

Usando o matplotlib é gerado uma animação, onde será gerado vários gráficos de nós para cada passo do algoritmo.

```
ani = animation.FuncAnimation(self.fig, update, frames=len(self.path) + 1, interval=1, repeat=False)
```

Para cada frame, extraímos a posição equivalente nos caminhos. Por exemplo se o frame atual é 50 vamos extrair o nó guardado na posição 50 do caminho retornando pelo algoritmo.

```
current_node = self.path[frame] # Nó atual
```

E então pintamos os nós já visitados e o atual para facilitar o entendimento da busca.

```
# Atualizar nós visitados (verde)
nx.draw_networkx_nodes(self.G, pos, ax=self.ax, nodelist=self.visited_nodes, node_color="green")
# Atualizar o nó atual (vermelho)
nx.draw_networkx_nodes(self.G, pos, ax=self.ax, nodelist=[current_node], node_color="red")
```

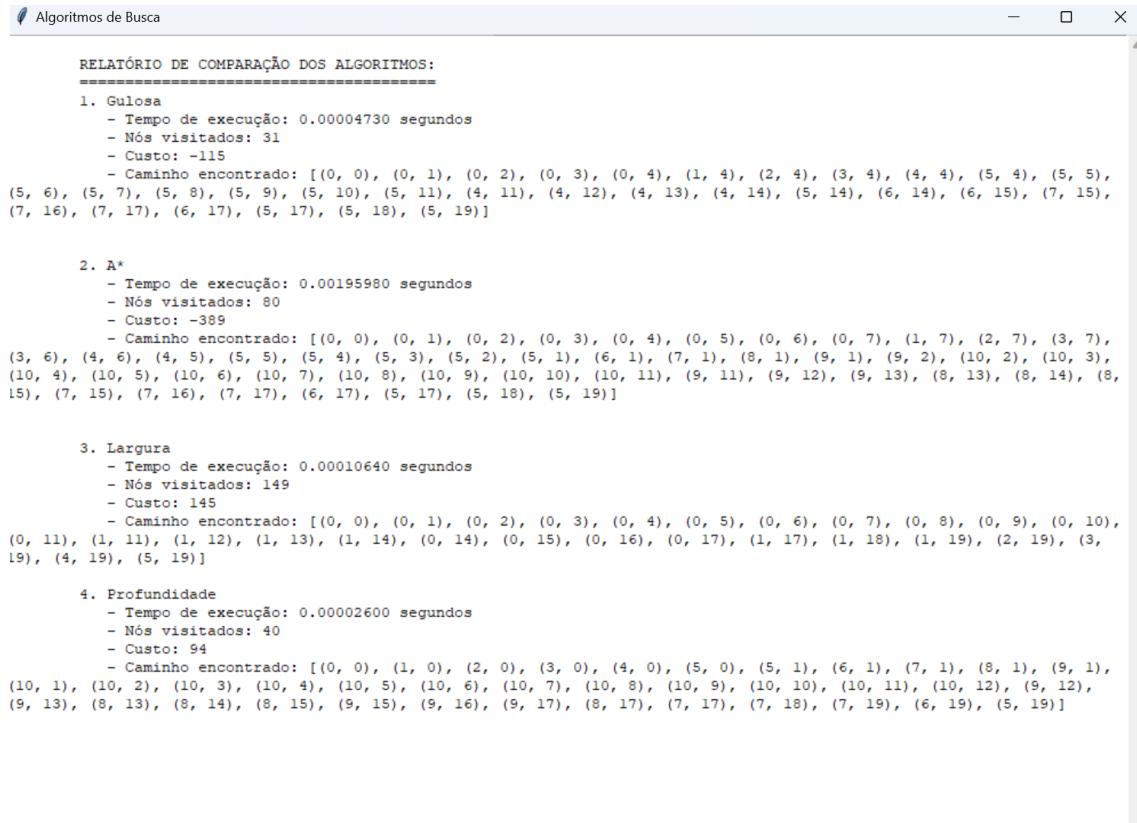
Ao chegar no final, verificamos se o nó objetivo existe no caminho e pintamos o caminho mínimo de vermelho, anteriormente estava sendo pintando todos os nós expandidos pelo algoritmo, agora será pintado o caminho mínimo e por fim paramos a animação.

```
# Verificar se a animação chegou ao fim
if frame >= len(self.path):
    # Verifica se o objetivo foi alcançado
    message = "Objetivo alcançado!" if goal in self.best_path else "Objetivo não alcançado."

    # Pinta os nós do caminho final de vermelho
    nx.draw_networkx_nodes(self.G, pos, ax=self.ax, nodelist=self.best_path, node_color="red")
    self.ax.set_title(message)
    ani.event_source.stop() # Parar a animação
    return
```

## interface\_relatorio.py

Classe que printa o relatório na tela numa janela, ela utilizando o retorno da classe relatorio.py para criar uma janela com as métricas.



```
Algoritmos de Busca

RELATÓRIO DE COMPARAÇÃO DOS ALGORITMOS:
=====
1. Gulosa
- Tempo de execução: 0.00004730 segundos
- Nós visitados: 31
- Custo: -115
- Caminho encontrado: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (5, 5),
(5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (5, 11), (4, 11), (4, 12), (4, 13), (4, 14), (5, 14), (6, 14), (6, 15), (7, 15),
(7, 16), (7, 17), (6, 17), (5, 17), (5, 18), (5, 19)]

2. A*
- Tempo de execução: 0.00195980 segundos
- Nós visitados: 80
- Custo: -389
- Caminho encontrado: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (1, 7), (2, 7), (3, 7),
(3, 6), (4, 6), (4, 5), (5, 5), (5, 4), (5, 3), (5, 2), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (9, 2), (10, 2), (10, 3),
(10, 4), (10, 5), (10, 6), (10, 7), (10, 8), (10, 9), (10, 10), (10, 11), (9, 11), (9, 12), (9, 13), (8, 13), (8, 14), (8,
15), (7, 15), (7, 16), (7, 17), (6, 17), (5, 17), (5, 18), (5, 19)]

3. Largura
- Tempo de execução: 0.00010640 segundos
- Nós visitados: 149
- Custo: 145
- Caminho encontrado: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10),
(0, 11), (1, 11), (1, 12), (1, 13), (1, 14), (0, 14), (0, 15), (0, 16), (0, 17), (1, 17), (1, 18), (1, 19), (2, 19), (3,
19), (4, 19), (5, 19)]

4. Profundidade
- Tempo de execução: 0.00002600 segundos
- Nós visitados: 40
- Custo: 94
- Caminho encontrado: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1),
(10, 1), (10, 2), (10, 3), (10, 4), (10, 5), (10, 6), (10, 7), (10, 8), (10, 9), (10, 10), (10, 11), (10, 12), (9, 12),
(9, 13), (8, 13), (8, 14), (8, 15), (9, 15), (9, 16), (9, 17), (8, 17), (7, 17), (7, 18), (7, 19), (6, 19), (5, 19)]
```

## main.py

arquivo main do projeto, centraliza as operações das classes e as executa na ordem.