Perfeito. Vou te entregar **duas coisas**, bem separadas e organizadas, exatamente para o que você pediu:

1. **GUIA DE APRESENTAÇÃO ORAL / ESTUDO PESSOAL**
   → para você *entender tudo*, explicar com segurança e responder perguntas.

2. **TEXTO DE RELATÓRIO INICIAL (ESTILO ACADÊMICO)**
   → já escrito para **entregar ao professor**, para alguém que **não viu o código**.

Vou ser **didático, completo e fiel ao seu projeto real**, sem inventar nada.

1️⃣ GUIA SUPER DETALHADO DO PROGRAMA (PARA APRESENTAÇÃO)

1. Visão geral do trabalho

O trabalho consiste em **resolver um problema de caminhos mínimos** em mapas bidimensionais, usando **algoritmos clássicos de grafos**, e **comparar desempenho e comportamento** desses algoritmos conforme o tamanho do mapa cresce.

O mapa não é um grafo "pronto":
ele **precisa ser interpretado, convertido em grafo e só então processado**.

2. Interpretação do problema (o que o trabalho pede)

O mapa é um arquivo `.txt` contendo:

Símbolo	Significado	Custo
`I`	Início	0
`F`	Fim	0
`G`	Terreno fácil	1
`S`	Terreno médio	3
`W`	Terreno difícil	5
`#`	Obstáculo	inválido

Objetivo

Encontrar o **menor custo total** para ir de `I` até `F`, movendo-se apenas:

- cima

- baixo
- esquerda
- direita

Sem passar por `#`.

Esse é exatamente um **problema de menor caminho em grafo ponderado**.

3. Como o mapa é lido (`Mapa.py`)

Tudo começa na função:

`carregar_mapa(nome_arquivo)`

Passo a passo:

1. Lê o arquivo linha por linha.
2. Remove espaços e quebras de linha.
3. Cria uma matriz:

`grid[i][j]`

onde `i` é a linha e `j` é a coluna.

4. Conta:

```
linhas = len(grid)
colunas = len(grid[0])
```

5. Varre o grid para encontrar:

- posição de `I`
- posição de `F`

4. Mapeamento célula → vértice (parte-chave)

Cada célula do mapa vira **um vértice do grafo**.

A conversão usada é:

`vertice = i * colunas + j`

Exemplo (mapa 10x10):

Célula	Vértice
0,0	0

(0,1)	1
(1,0)	10
(1,1)	11

Isso garante:

- vértices numerados de 0 até `(linhas * colunas - 1)`
- mapeamento simples e reversível

5. Como o mapa vira um grafo

Ainda em `carregar_mapa()`:

Estrutura usada

```
grafo = ListaAdjacencias(linhas * colunas)
```

👉 **Lista de Adjacências** foi escolhida porque:

- o grafo é esparso
- cada vértice tem no máximo 4 vizinhos
- matriz de adjacência seria inviável em mapas grandes

Arestas

Para cada célula válida `(i,j)`:

1. Calcula `u = i * colunas + j`
2. Testa as 4 direções:

```
direcoes = [(-1,0),(1,0),(0,-1),(0,1)]
```

3. Se o vizinho existe e não é `#`, cria:

```
grafo.addAresta(u, v, peso)
```

Como o peso é definido

```
peso = custo_terreno(grid[ni][nj])
```

🔷 **Peso da aresta = custo de entrar na célula de destino**

Isso significa que o algoritmo minimiza o custo do terreno por onde se pisa.

6. O que o `main.py` faz (pipeline completo)

O `main.py` é o **orquestrador** do projeto.

Fluxo completo:

1. Recebe o mapa via linha de comando:

```
python main.py mapa.txt
```

2. Chama:

```
grid, linhas, colunas, inicio, fim, grafo = carregar_mapa()
```

3. Executa **três algoritmos em sequência**:

- Dijkstra
- Bellman-Ford
- Floyd-Warshall

7. Como cada algoritmo é executado

Tudo passa por:

```
executar_algoritmo(...)
```

Dentro dessa função:

1. Marca tempo inicial (`time.perf_counter()`).
2. Executa o algoritmo.
3. Marca tempo final.
4. Calcula custo:
    - Dijkstra / Bellman: `dist[fim]`
    - Floyd: `dist[inicio][fim]`
5. Reconstrói o caminho via predecessores.
6. Marca o caminho no grid com `*`.
7. Salva em arquivo:
    - `saida_dijkstra.txt`
    - `saida_bellman_ford.txt`
    - `saida_floyd_warshall.txt`
8. Imprime custo e tempo no terminal.

8. Por que Dijkstra faz sentido aqui

- Resolve **menor caminho a partir de uma origem**
- Pesos são **não-negativos**
- Só precisamos de `I → F`
- É o algoritmo **mais adequado conceitualmente**

9. Como Floyd-Warshall entra no problema

Floyd-Warshall calcula:

```
menor caminho entre TODOS os pares (i, j)
```

Ele monta:

```
dist[i][j]
prev[i][j]
```

No projeto, **apenas um valor é usado**:

```
dist[inicio][fim]
```

👉 Ele faz **muito mais trabalho do que o necessário**, mas:

- o trabalho exige comparação
- serve para mostrar o impacto da complexidade `O(V³)`

10. Impacto do tamanho do mapa

Se o mapa tem `L x C`:

- Vértices:
  `V = L * C`
-
- Arestas (aprox.):
  `E ≈ 4V`
-

Complexidade observada

Algoritmo	Complexidade	Impacto
Dijkstra (iniciante)	$O(V^2)$	Muito lento em mapas grandes

Bellman-Ford	$O(V \cdot E) \approx O(V^2)$	Mais rápido que Dijkstra nesse cenário
Floyd-Warshall	$O(V^3)$	Inviável para mapas grandes

Isso explica:

- Floyd estourando tempo ou memória
- Dijkstra chegando a centenas de segundos
- Bellman-Ford sendo surpreendentemente mais rápido

11. Por que lista de adjacência é essencial

Para `300x300`:

- `V = 90.000`
- Matriz de adjacência: `V² = 8,1 bilhões` posições
- Isso **não cabe em memória** em Python

Lista de adjacência:

- Guarda só arestas existentes
- Escala corretamente

12. O que o benchmark faz

O `main_benchmark.py`:

- Executa **10 rodadas por algoritmo e mapa**
- Mede tempo e custo de cada rodada
- Calcula médias
- Gera:
  - tabelas por rodada
  - resumo por mapa
  - Tabela 1 final
- Registra tudo no console **e no `log.txt`**

2️⃣ TEXTO DE RELATÓRIO INICIAL (PARA ENTREGA)

Introdução

Este trabalho prático tem como objetivo implementar e analisar algoritmos clássicos de caminhos mínimos em grafos, aplicados à resolução de trajetos de menor custo em mapas bidimensionais. O problema consiste em encontrar o menor custo total entre um ponto inicial e um ponto final, considerando diferentes tipos de terreno e obstáculos.

Os algoritmos implementados foram **Dijkstra**, **Bellman-Ford** e **Floyd-Warshall**, seguindo fielmente os pseudocódigos apresentados em sala. Além disso, foi desenvolvido um módulo de experimentos computacionais para comparação de desempenho, conforme especificado no enunciado do trabalho.

Modelagem do Problema

O mapa é representado como uma matriz bidimensional de caracteres, onde cada célula corresponde a um tipo de terreno ou obstáculo. Para possibilitar o uso dos algoritmos de grafos, o mapa é convertido em um grafo direcionado e ponderado.

Cada célula válida do mapa é mapeada para um vértice único, utilizando a fórmula `v = i * colunas + j`. Arestas são criadas entre células adjacentes (movimentos em quatro direções), e o peso de cada aresta corresponde ao custo do terreno da célula de destino.

Implementação

O programa é dividido em módulos bem definidos:

- `Mapa.py`: leitura do mapa e conversão para grafo
- `Grafo.py`: estruturas de grafos
- `Algoritmos.py`: implementação dos algoritmos de caminhos mínimos
- `main.py`: execução simples
- `main_benchmark.py`: execução dos experimentos computacionais

A estrutura de **lista de adjacências** foi adotada por ser mais eficiente para grafos esparsos, característica natural do problema tratado.

Algoritmos Utilizados

O algoritmo de **Dijkstra** é utilizado por ser adequado a grafos com pesos não-negativos e resolver o problema a partir de uma única origem. O **Bellman-Ford** também resolve caminhos mínimos a partir de uma origem, sendo utilizado para comparação. Já o **Floyd-Warshall** calcula os caminhos mínimos entre todos os pares de vértices, sendo incluído para análise comparativa, apesar de seu elevado custo computacional.

Experimentos Computacionais

Cada algoritmo foi executado dez vezes para cada mapa fornecido, sendo calculadas as médias do tempo de execução e do custo do caminho mínimo. Nos casos em que a execução não finalizou dentro do limite estipulado, o resultado foi registrado como “TEMPO LIMITE”.

Conclusão

Os resultados obtidos evidenciam o impacto direto do tamanho do grafo na execução dos algoritmos. Enquanto Dijkstra e Bellman-Ford apresentaram desempenho aceitável em mapas de tamanho médio, o Floyd-Warshall mostrou-se inviável para mapas grandes, confirmando a análise teórica de complexidade.

Se você quiser, no próximo passo eu:

- adapto esse texto exatamente ao **padrão do seu professor**, ou
- preparo um **roteiro de apresentação oral de 5–10 minutos**, ou
- escrevo a **seção de análise dos resultados** baseada nos seus logs reais.