Trabalho prático de Grafos

Análise de grafos utilizando os algoritmos de Dijkstra, Bellman-Ford e Floyd-Warshall

Nome: Vinícius Alochio Santos

Matrícula: 18.1.8145

Repositório: (https://github.com/Alochio/trabalho-grafos)

Introdução

O código desenvolvido é uma aplicação Python para análise de grafos, utilizando três algoritmos clássicos de caminho mínimo: Dijkstra, Bellman-Ford e Floyd-Warshall. A aplicação permite a leitura de um arquivo de entrada no formato DIMACS, executa os algoritmos com um timeout definido e gerencia a memória durante a execução. O objetivo é comparar o desempenho e a aplicabilidade dos algoritmos para diferentes tipos de grafos.

Estrutura do Código

O código é organizado nas seguintes funções, cada uma com responsabilidades específicas:

1. **Leitura do Arquivo DIMACS**
2. **Execução com Timeout**
3. **Algoritmos de Caminho Mínimo**
4. **Gerenciamento de Memória**
5. **Função Principal (main)**

Detalhamento das Funções

1. **read_dimacs_file(file_path)**
    - **Descrição**: Lê um arquivo de grafo no formato DIMACS e constrói um grafo direcionado (DiGraph) usando a biblioteca NetworkX.
    - **Entrada**: file_path - Caminho para o arquivo DIMACS.
    - **Saída**: Um grafo (DiGraph) com os nós e arestas especificados no arquivo.
    - **Tratamento de Erro**: Captura MemoryError e levanta uma exceção com a mensagem "MEMORIA EXCEDIDA".
2. **run_with_timeout(func, *args, timeout=600)**
    - **Descrição**: Executa uma função com um timeout definido.
    - **Entrada**: func - Função a ser executada; *args - Argumentos da função; timeout - Tempo máximo de execução (padrão de 600 segundos).
    - **Saída**: Retorna o resultado da função, um erro (se houver), e o tempo gasto.
    - **Tratamento de Erro**: Captura MemoryError e outros erros, retornando a mensagem apropriada.
3. **dijkstra(graph, start, end)**
    - **Descrição**: Implementa o algoritmo de Dijkstra para encontrar o caminho mínimo entre dois nós em um grafo.
    - **Entrada**: graph - O grafo direcionado; start - Nó inicial; end - Nó final.
    - **Saída**: Caminho mais curto e custo total.
    - **Detalhes**:
        - Utiliza uma fila de prioridade para otimizar a escolha do próximo nó.
        - Atualiza as distâncias e predecessores até encontrar o caminho mais curto.
        - Reconstrói o caminho a partir dos predecessores.
4. **bellman_ford(graph, start, end)**
    - **Descrição**: Implementa o algoritmo de Bellman-Ford para encontrar o caminho mínimo em um grafo, com detecção de ciclos negativos.
    - **Entrada**: graph - O grafo direcionado; start - Nó inicial; end - Nó final.
    - **Saída**: Caminho mais curto e custo total, ou uma mensagem de erro se houver um ciclo negativo.
    - **Detalhes**:
        - Relaxamento das arestas repetidamente.
        - Verificação adicional para ciclos negativos.
        - Reconstrução do caminho a partir dos predecessores.
5. **floyd_warshall(graph)**
    - **Descrição**: Reimplementa o algoritmo de Floyd-Warshall usando listas para encontrar caminhos mínimos entre todos os pares de nós.

- **Entrada**: graph - O grafo direcionado.
- **Saída**: Função reconstruct_path(start, end) que retorna o caminho mínimo e custo entre dois nós.
- **Detalhes**:
  - Inicializa uma matriz de distâncias e uma matriz de próximos nós.
  - Aplica o algoritmo de Floyd-Warshall para atualizar a matriz de distâncias.
  - Reconstrói o caminho mais curto entre dois nós.

6. **clear_memory()**
   - **Descrição**: Força a coleta de lixo e limpa variáveis globais para liberar memória.
   - **Entrada**: Nenhuma.
   - **Saída**: Nenhuma.
   - **Detalhes**:
     - Utiliza a coleta de lixo (gc.collect()) para limpar a memória.
     - Remove variáveis globais não essenciais.
7. **main()**
   - **Descrição**: Função principal que orquestra a leitura do arquivo, execução dos algoritmos e exibição dos resultados.
   - **Entrada**: Argumentos da linha de comando (sys.argv).
   - **Saída**: Resultados dos algoritmos e tempos de execução.
   - **Detalhes**:
     - Lê o grafo do arquivo DIMACS.
     - Verifica se os nós de origem e destino existem no grafo.
     - Executa os algoritmos de Dijkstra, Bellman-Ford e Floyd-Warshall, exibindo os resultados e tempos de execução.
     - Limpa a memória entre as execuções dos algoritmos para evitar consumo excessivo.

Os resultados da execução do código estão documentados no arquivo log.txt, incluído no repositório junto com o código. Este arquivo contém os resultados de todas as 10 execuções realizadas para cada grafo, incluindo os valores gerados pelos três algoritmos (Dijkstra, Bellman-Ford e Floyd-Warshall). O log.txt deve ser consultado para revisar o desempenho e os resultados de cada execução.

Resumo Geral dos Algoritmos de Caminho Mínimo

1. Algoritmo de Dijkstra

- **Descrição**: Encontra o caminho mais curto a partir de um vértice fonte para todos os outros vértices em grafos com arestas com pesos não-negativos.
- **Desempenho**:
  - **Tempo de Execução**: Consistentemente rápido, variando de 0.076s a 0.251s em grafos grandes e densos.
  - **Custo**: Precisão alta com resultados corretos e confiáveis.
- **Aplicabilidade**: Ideal para grafos grandes e densos com pesos de arestas não-negativos. Efetivo para encontrar o caminho mínimo de um único ponto a todos os outros pontos.

2. Algoritmo de Bellman-Ford

- **Descrição**: Encontra o caminho mais curto a partir de um vértice fonte para todos os outros vértices e pode lidar com arestas com pesos negativos. Também detecta ciclos negativos.
- **Desempenho**:
  - **Tempo de Execução**: Muito mais lento, variando de 222.311s a 450.150s para grafos grandes.

- **Custo**: Resultados corretos, mas o alto tempo de execução torna-o menos prático para grafos grandes.
- **Aplicabilidade**: Útil para grafos com pesos negativos. Menos eficiente para grafos grandes devido ao seu alto tempo de execução.

3. Algoritmo de Floyd-Warshall

- **Descrição**: Calcula todos os pares de caminhos mínimos em um grafo, útil para grafos onde se deseja saber o caminho mínimo entre todos os pares de vértices.
- **Desempenho**:
  - **Tempo de Execução**: Excedeu o limite de tempo para todos os grafos testados, indicando que não é prático para grafos grandes.
- **Aplicabilidade**: Adequado para grafos pequenos ou para casos onde é necessário saber o caminho mínimo entre todos os pares de vértices. Não é viável para grafos grandes devido ao elevado tempo de execução.

Conclusão Geral

- **Dijkstra** é o algoritmo preferido para grafos grandes com pesos não-negativos devido ao seu desempenho rápido e preciso.
- **Bellman-Ford** é uma boa opção para grafos com pesos negativos, mas seu uso em grafos grandes é limitado devido ao tempo de execução elevado.
- **Floyd-Warshall** é mais adequado para grafos menores ou para situações em que todos os pares de caminhos mínimos são necessários, mas não é prático para grafos grandes devido ao seu alto custo computacional.

Tabela de Resultados

Grafo	Dijkstra		Bellman-Ford		Floyd-Warshall	
	T. médio(s)	Custo médio	T. médio(s)	Custo médio	T. médio(s)	Custo médio
Toy	0s	3,2	0s	3,2	0s	3,2
facebook combined	0,083s	2,9	249,395s	2,9	TEMPO_LIMITE	2,9
rg300 4730	0,010s	12,2	1,672s	12,2	10,567s	12,2
rome99c	0,017s	25791,5	2,172s	25791,5	TEMPO_LIMITE	25791,5
USA-road-dt.DC	0,083s	6845,9	249,404s	6845,9	TEMPO_LIMITE	6845,9

Referências:

Slide Aula 07 – Busca em Grafos:
https://moodlepresencial.ufop.br/pluginfile.php/1664782/mod_resource/content/1/07-busca.pdf

Slide Aula 08 – Problema do caminho mínimo Algoritmo de Dijkstra:
https://moodlepresencial.ufop.br/pluginfile.php/1670382/mod_resource/content/1/08-dijkstra.pdf

Slide Aula 09 – Problema do caminho mínimo Algoritmo de Belman-Ford:
https://moodlepresencial.ufop.br/pluginfile.php/1670383/mod_resource/content/1/09-bellman-ford.pdf

Slide Aula 10 – Problema do caminho mínimo Algoritmo de Floyd-Warshall:
https://moodlepresencial.ufop.br/pluginfile.php/1674045/mod_resource/content/1/10-floyd-warshall.pdf