

Primeiro Projeto de PFL

1 Implementação de `shortestPath` com algoritmo de Dijkstra

Na função `shortestPath`, utilizámos o algoritmo de Dijkstra para encontrar todos os caminhos mais curtos entre duas cidades num `RoadMap`. Desta forma, explora de maneira eficiente os caminhos entre duas cidades, dando prioridade a paths mais curtos através de uma priority queue.

1.1 Estruturas de Dados

1. Lista de Adjacência (`AdjList`):

- O `roadMap` é convertido numa `AdjList` com `roadMapToAdjList`, permitindo acessos rápidos aos vizinhos de cada cidade.
- **Justificação:** A lista de adjacência é eficiente para grafos esparsos, poupando espaço e permitindo acesso rápido aos nodes vizinhos.

2. Fila de Prioridade (`QueueEntry`):

- Cada entrada na fila é um tuplo (`Distance`, `City`, `Path`), ordenado pela distância acumulada.
- **Justificação:** A fila dá prioridade a caminhos mais curtos, garantindo que as cidades são exploradas por ordem de distância.

1.2 Passos do Algoritmo

1. Inicialização:

- Se `start == end`, dá return a `[start]`. Caso contrário, inicializa a fila de prioridade com `(0, start, [start])` e começa a explorar os caminhos.

2. Ciclo Principal (Algoritmo de Dijkstra):

- O algoritmo remove da fila a entrada com a menor distância acumulada.
- Se a cidade atual corresponde a `end`, o caminho é adicionado ao resultado se tiver a distância mínima.

- Os vizinhos são adicionados à fila com distâncias atualizadas, mantendo a ordem dos caminhos com `insertQueue`.
- **Seleção de Caminhos:** Apenas os caminhos com a distância mínima total até `end` são mantidos.

1.3 Complexidade

O algoritmo atinge uma complexidade de tempo de $O((V + E) \cdot V)$, onde V é o número de cidades e E o número de estradas, devido ao uso eficiente de uma lista de adjacência e de uma fila de prioridade.

2 Implementação do TSP com Programação Dinâmica

2.1 Recursão Base

1. **Caso base:** Se o conjunto S contém apenas dois nós, ou seja, $S = \{1, i\}$:

$$C(S, i) = \text{dist}(1, i)$$

2. **Caso recursivo:** Se o tamanho de S é maior que 2:

$$C(S, i) = \min\{C(S - \{i\}, j) + \text{dist}(j, i)\}$$

onde j pertence a S , $j \neq i$ e $j \neq 1$.

Aqui, $C(S, i)$ representa o custo da viagem para o nó i no estado S de nós por visitar.

2.2 Estruturas de Dados

Utilizamos as seguintes definições de tipo:

- **TspCoord:** Representa um estado no TSP, contendo um nó e o conjunto de nós restantes.
- **TspEntry:** Representa uma entrada na tabela, contendo o custo total e o caminho correspondente.

A utilização de uma bitmask para representar o estado da tour é mais eficiente em termos de tempo de pesquisa e memória do que listas, pois permite operações rápidas de verificação e manipulação de estados com menor uso de memória.

2.3 Representação do Grafo

A representação do grafo é feita através de uma matriz de adjacência:

- **Matriz de Adjacência:** Esta estrutura é eficiente para buscas rápidas de pesos de arestas. A desvantagem é o uso maior de memória, mas a escolha é justificada para grafos menores como normalmente é o caso dos grafos onde se aplica TSP dado ser um problema NP.

2.4 Complexidade

A complexidade do algoritmo pode ser analisada da seguinte forma:

- **Tamanho da Tabela:** Para cada n vai haver 2^n valores possíveis para S (máscara) por isso o tamanho da tabela vai ser $n \cdot (2^n)$.
- **Cálculo de Entradas:** Cada entrada na tabela pode exigir até n etapas para ser computada, resultando em uma eficiência total do algoritmo de $O(n^2 \cdot 2^n)$ para o cálculo.

Consequentemente, a eficiência do algoritmo é $O(n^2 \cdot 2^n)$. Embora essa complexidade pareça considerável, não existe outro algoritmo eficiente conhecido para resolver o problema.

A abordagem de programação dinâmica proporciona um desempenho razoavelmente bom para grafos pequenos, tornando-a uma escolha prática, apesar da complexidade exponencial.

2.5 Funções Principais

1. **travelSales:** Converte o grafo em uma matriz de adjacência, chama a função `tsp` para calcular o custo mínimo e depois converte o resultado de volta para os dados iniciais.
2. **tsp:** Inicializa a tabela dinâmica e começa a busca a partir do nó n . Armazena os resultados de subproblemas calculados pelo `compTsp`, evitando cálculos redundantes.
3. **compTsp:** Verifica na tabela o custo de ir em direção a cada nó vizinho, calculando o custo do vizinho no estado S mais o peso da aresta, e escolhe o menor custo, concatenando-o com o caminho atual.

2.6 Considerações Finais

- A abordagem de programação dinâmica para o TSP oferece uma forma eficiente de resolver o problema, ao armazenar resultados intermediários.
- O uso de uma matriz de adjacência garante buscas rápidas, compensando o maior uso de memória, o que é aceitável no contexto do TSP.

3 Divisão de trabalho

José Sousa: 50%

funções implementadas: 1, 3, 5, 7, 9

João Mendes: 50%

funções implementadas: 2, 4, 6, 8

Referência

Fethi Rabhi e Guy Lapalme. *Algorithms: A Functional Programming Approach*. Addison-Wesley, 2ª edição, 1999.