Faculdade de Engenharia da Universidade do Porto

Manipulação de Graphs

Trabalho Prático 1

Programação Funcional e em Lógica

Turma 16 - Grupo 1

João Henrique Andrade Proença (up202207835@fe.up.pt)

Ricardo Ascensão Parreira(up202205091@fe.up.pt)

Índice

Shortest Path 3
Objetivo 3
Estruturas de Dados 3
Funções Auxiliares 3
Estratégias Importantes 3
Travel Sales 4
Objetivo 4
Estruturas de Dados 4
Funções Auxiliares 4
findNearestNeighbor: 4
nearestNeighbor: 4
travelSales: 5
Contribuições 5

Shortest Path

Objetivo

A função shortestPath encontra todos os caminhos mínimos entre duas cidades num RoadMap não direcionado e ponderado.

Estruturas de Dados

- **RoadMap**: Representa as arestas do grafo no formato ***(City, City, Distance)***.
- **Lista de Adjacência**: Converte o RoadMap em ***[(City, [(City, Distance)])]*** onde ***[(City, Distance)]*** é uma lista de pares com cada cidade adjacente à primeira e a distância entre as duas para fácil acesso aos vizinhos.
- **Queue**: Usada na BFS, armazena ***(City, Distance, Path)*** para a cidade atual, a distância percorrida e o caminho seguido. Sendo muito útil mais tarde na ***BFS***.

Funções Auxiliares

- **convert**: Transforma ***RoadMap*** numa lista de adjacência previamente explicada.
- **getNeighbors**: Retorna vizinhos de uma cidade.
- **bfs**: Realiza a busca em largura para encontrar caminhos mínimos.
- **updateQueue**: Atualiza a fila apenas com caminhos mais curtos.

Estratégias Importantes

- **BFS**: Garante busca de menor caminho num grafo não direcionado. Aqui, adaptámo-la para considerar distâncias acumuladas e evitar caminhos mais longos do que o mínimo.
- **Filtrar Duplicados*: Data.List.nub*** (função predefinida) elimina caminhos repetidos ao final. Como a BFS pode gerar caminhos redundantes para a mesma cidade com igual distância, esta filtragem garante que a saída contenha apenas caminhos únicos.
- **Otimização**: A transformação para uma lista de adjacência permite que o algoritmo aceda diretamente aos vizinhos de uma cidade, evitando uma pesquisa linear no ***RoadMap*** para cada cidade, o que reduz o tempo de execução, especialmente em mapas

Travel Sales

Objetivo

A função ***travelSales*** foi implementada para resolver oTravel Salesman Problem (TSP), usando uma abordagem baseada no algoritmo nearest neighbor. A ideia é partir de uma cidade inicial e, a cada passo, visitar a cidade mais próxima que ainda não foi visitada. Ao final, tentamos retornar à cidade inicial para completar um ciclo fechado (ou seja, uma rota de ida e volta que passa por todas as cidades uma vez).

Estruturas de Dados

- ***AdjList*** (Lista de Adjacência): Essa estrutura de dados é usada para representar o mapa de estradas como um grafo não-direcionado, onde cada cidade é um nó, e cada estrada (com a distância associada) é uma aresta entre dois nós. Usamos a lista de adjacência porque facilita a busca eficiente dos vizinhos de uma cidade, especialmente para o algoritmo do vizinho mais próximo, onde precisamos de verificar as cidades vizinhas para encontrar a mais próxima.
- ***unvisited*** (Lista de Cidades Não Visitadas): É uma lista que mantém o controle das cidades que ainda não foram visitadas durante a busca. Em cada iteração, essa lista é atualizada removendo a cidade recém-visitada. Isso permite garantir que cada cidade seja visitada apenas uma vez.

Funções Auxiliares

findNearestNeighbor:

Esta função recebe a lista de adjacência (***adjList***), a cidade atual (***currentCity***) e as cidades não visitadas (***unvisited***).

Filtra os vizinhos da cidade atual para obter apenas aqueles que estão na lista de cidades não visitadas, criando a lista ***validNeighbors***.

Caso ***validNeighbors*** esteja vazia, retorna ***Nothing***, indicando que não há cidades acessíveis a partir da cidade atual que ainda não foram visitadas.

Caso contrário, retorna a cidade vizinha mais próxima, com base na menor distância.

nearestNeighbor:

É a função principal recursiva que executa o algoritmo do vizinho mais próximo. A cada chamada:

Verifica se todas as cidades foram visitadas. Se sim, tenta retornar à cidade inicial (***startCity***). Se não houver caminho de volta, retorna uma lista vazia, indicando que um ciclo completo não é possível.

Caso ainda haja cidades a serem visitadas, usa ***findNearestNeighbor*** para determinar a próxima cidade a visitar.

Adiciona essa cidade ao ***path*** e remove-a de ***unvisited***, garantindo que a cidade não será visitada novamente.

travelSales:

Inicializa o grafo como lista de adjacência (*adjList*) e define a cidade de início (***startCity***).

Chama ***nearestNeighbor*** passando ***startCity*** como ponto de partida, a lista de cidades não visitadas (todas exceto a cidade inicial) e o caminho inicial que contém apenas a cidade inicial.

Ao final, verifica se o caminho (***path***) possui o comprimento adequado (número de cidades + 1) e se o último elemento é a cidade inicial. Se essas condições forem atendidas, retorna *path* como a solução do TSP; caso contrário, retorna uma lista vazia, indicando que um ciclo completo não foi possível.

Contribuições

João Proença - 50%, exercícios realizados: 1,3,5,8,9.

Ricardo Parreira – 50%, exercícios realizados: 2,4,6,7,8.