Relatório: Problema do Caminho Mínimo em Grafos

Introdução

O problema do caminho mínimo em grafos consiste em encontrar a rota mais curta entre dois vértices de um grafo, considerando que as arestas possuem pesos que podem representar distância, custo ou tempo. Este problema possui ampla aplicação em diversas áreas, como redes de transporte, comunicação e logística.

Existem várias variantes para problemas de caminho mínimo, cada uma adequada a um conjunto de problemas diferente:

- Problema de único destino: consiste em determinar o menor caminho entre cada um dos nós do grafo e um nó de destino dado.
- Problema de única origem: determinar o menor caminho entre um nó dado e todos os demais nós do grafo.
- Problema de origem-destino: determinar o menor caminho entre nós dados.
- Problemas de todos os pares: determinar o menor caminho entre cada par de nós presentes no grafo.

Principais Algoritmos

Algoritmo de Dijkstra

O algoritmo de Dijkstra é um método guloso que encontra o caminho mais curto de um vértice fonte para todos os outros vértices de um grafo com pesos não negativos. O método original utiliza o método de programação dinâmica, com uma matriz de pesos, onde o valor registrado é infinito quando não há arestas conectando dois vértices diretamente, e caso contrário, é o peso da dita aresta. No caso implementado, são usados dois vetores, um para manter registro dos vértices já visitados do grafo, enquanto o outro corresponde ao peso efetivo (total) para ir de um vértice ao outro.

Etapas principais:

1. Inicialização das distâncias como infinito, exceto a distância do vértice fonte, que é zero.
2. A cada iteração, seleciona-se o vértice de menor distância ainda não visitado.
3. Atualiza-se as distâncias dos vértices vizinhos, caso encontre um caminho mais curto.
4. O algoritmo termina quando todas os vértices foram visitados.

Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford é uma estratégia gulosa que resolve o problema do caminho mínimo em grafos que podem conter arestas com pesos negativos. O algoritmo também é capaz de identificar ciclos negativos, que são úteis em alguns casos.

Etapas principais:

1. Inicialização das distâncias como infinito, exceto a distância do vértice fonte, que é zero.
2. Relaxação de todas as arestas do grafo por |V| - 1 iterações. Relaxar uma aresta significa tentar melhorar a distância conhecida de um vértice a partir de outro.
3. Verificação de ciclos negativos: se, após |V| - 1 iterações, alguma aresta puder ser relaxada, o grafo possui um ciclo negativo.

Implementações

Foram implementados os algoritmos de Dijkstra e Bellman-Ford em C, utilizando estruturas dinâmicas para representar o grafo. O código utiliza listas simples para armazenar as arestas de cada vértice, além de um vetor para guardar os vértices, implementado como um ponteiro. Cada endereço do ponteiro de vértices guarda uma lista de arestas.

Trechos Importantes do Código:

- Estruturas de Dados

```
// Estrutura para representar uma aresta
typedef struct Aresta {
	int indice_destino;
	int peso_aresta;
	struct Aresta* proxima_aresta;
} Aresta;

// Estrutura para representar um nó no grafo
typedef struct No {
	Aresta* lista_arestas;
} No;

// Estrutura para representar o grafo
typedef struct Grafo {
	int quantidade_nos;
	No* lista_nos;
```

```
} Grafo;
```

- Criação do Grafo

```
Grafo* criar_grafo(int quantidade_nos) {
	Grafo* grafo = (Grafo*)malloc(sizeof(Grafo));
	grafo->quantidade_nos = quantidade_nos;
	grafo->lista_nos = (No*)malloc(quantidade_nos * sizeof(No));
	for (int indice = 0; indice < quantidade_nos; índice++) {
		grafo->lista_nos[indice].lista_arestas = NULL;
	}
	return grafo;
}
```

- Adição de Arestas

```
void adicionar_aresta(Grafo* grafo, int indice_origem, int indice_destino,
int peso_aresta) {
	Aresta* nova_aresta = (Aresta*)malloc(sizeof(Aresta));
	nova_aresta->indice_destino = indice_destino;
	nova_aresta->peso_aresta = peso_aresta;
	nova_aresta->proxima_aresta = grafo-
>lista_nos[indice_origem].lista_arestas;
	grafo->lista_nos[indice_origem].lista_arestas = nova_aresta;
}
```

Algoritmo de Dijkstra

O algoritmo de Dijkstra foi implementado conforme as etapas descritas acima, utilizando um vetor para armazenar as distâncias e uma lista para os vértices visitados.

Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford segue a mesma lógica teórica descrita, garantindo a verificação de ciclos negativos.

Execução

Foram criados grafos com pelo menos 10 vértices e 20 arestas. Ambos os algoritmos foram executados com sucesso, fornecendo os resultados esperados para o caminho mínimo entre os vértices. Como não foram realizados testes com arestas pesos negativos, não há mostra do funcionamento nesse caso.

Saída no terminal

```
Resultado do Algoritmo de Dijkstra:
Distância do nó 0 ao nó 0: 0
Distância do nó 0 ao nó 1: 10
Distância do nó 0 ao nó 2: 15
Distância do nó 0 ao nó 3: 22
Distância do nó 0 ao nó 4: 24
Distância do nó 0 ao nó 5: 25
Distância do nó 0 ao nó 6: 30
Distância do nó 0 ao nó 7: 33
Distância do nó 0 ao nó 8: 38
Distância do nó 0 ao nó 9: 37

Resultado do Algoritmo de Bellman-Ford:
Distância do nó 0 ao nó 0: 0
Distância do nó 0 ao nó 1: 10
Distância do nó 0 ao nó 2: 15
Distância do nó 0 ao nó 3: 22
Distância do nó 0 ao nó 4: 24
Distância do nó 0 ao nó 5: 25
Distância do nó 0 ao nó 6: 30
Distância do nó 0 ao nó 7: 33
Distância do nó 0 ao nó 8: 38
Distância do nó 0 ao nó 9: 37
```

Ou seja, é possível ver que ambos os algoritmos chegaram ao mesmo resultado.

Conclusão

Os algoritmos de Dijkstra e Bellman-Ford são ferramentas poderosas para resolver o problema do caminho mínimo em grafos, a partir de um único vértice de fonte. Cada um possui vantagens e limitações dependendo do tipo de grafo utilizado, sendo que no geral, o algoritmo de Dijkistra é mais rápido, com complexidade O(E logV), enquanto o de Bellman-Ford tem complexidade de tempo O(EV). No entanto, o algoritmo de Dijkstra não funciona quando o grafo tem arestas com peso negativo, enquanto o de Bellman-Ford não somente resolve esses casos, como também é capaz de mostrar a presença de ciclos negativos, sendo, portanto, o melhor para essa situação.

Quanto à aplicabilidade do grafo, por conta do uso de listas dinâmicas, que flexibilizam a representação e não alocam memória desnecessária (como seria o caso de um grafo por matriz de adjacência), ele pode ser usado em diferentes contextos.

Referências

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
2. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.
3. Sites e materiais complementares sobre implementação de algoritmos em C.