Algoritmo de Dijkstra

Pedro Lucas Siqueira da Paixão

IF Goiano - Ciência da Computação

Algoritmo de Dijkstra

- Resolve o problema do caminho mais curto em grafos com pesos não-negativos.
- Utiliza uma fila de prioridade para explorar o vértice mais próximo ainda não visitado.
- Mantém um vetor de distâncias mínimas e um vetor para reconstruir o caminho.
- Muito eficiente em redes de roteamento e mapas.

Representação do Grafo

Classe e Estrutura

```
struct Aresta {
    int destino;
    int peso;
};

class Grafo {
    int V;
    vector<vector<Aresta>> adj;
```

- O grafo é representado por lista de adjacência.
- Cada aresta tem um destino e um peso.
- `adj[u]` guarda as arestas saindo de `u`.

Adição de Arestas

Função adicionarAresta

```
void adicionarAresta(int u, int v, int peso) {
    adj[u].push_back({v, peso});
}
```

- Insere uma nova aresta do vértice `u` para `v`.
- O peso representa o custo de ir de `u` a `v`.

Dijkstra - Inicialização

```
vector<int> distancia(V, INF);
vector<int> anterior(V, -1);
distancia[origem] = 0;

priority_queue<pair<int, int>, vector<pair<int, int>
fila.push({0, origem});
```

- Inicializa as distâncias como infinitas.
- Origem tem distância 0.
- Fila de prioridade armazena pares (distância, vértice).

Dijkstra - Laço Principal

```
while (!fila.empty()) {
    int u = fila.top().second;
    int dist_u = fila.top().first;
    fila.pop();

    if (dist_u > distancia[u]) continue;

    for (const auto& aresta : adj[u]) {
        int v = aresta.destino;
        int peso = aresta.peso;

        if (distancia[u] + peso < distancia[v]) {
            distancia[v] = distancia[u] + peso;
            anterior[v] = u;
            fila.push({distancia[v], v});
        }
    }
}
```

Laço principal do Dijkstra (Explicação)

- Enquanto a fila de prioridade não estiver vazia:
  - Pega o vértice com menor distância estimada.
  - Se essa distância for maior que a já registrada, ignora.
  - Para cada vizinho do vértice atual:
    - Verifica se a distância passando pelo vértice atual é menor.
    - Se for, atualiza a menor distância e o caminho anterior.
    - Insere o vizinho com nova distância na fila.

Impressão dos Resultados

```
for (int i = 0; i < V; ++i) {
    cout << "Para " << i << ": ";
    if (distancia[i] == INF) cout << "infinito";
    else cout << distancia[i];
    cout << " | Caminho: ";
    imprimirCaminho(anterior, i);
    cout << "\n";
}
```

- Mostra a menor distância da origem até cada vértice.
- Reconstrói e imprime o caminho mínimo usando o vetor `anterior`.

Função imprimirCaminho

```
void imprimirCaminho(const vector<int>& anterior, in
    if (v == -1) return;
    imprimirCaminho(anterior, anterior[v]);
    if (anterior[v] != -1) cout << " -> ";
    cout << v;
}
```

- Imprime o caminho mínimo de forma recursiva.
- Vai da origem até o destino, seguindo o vetor `anterior`.

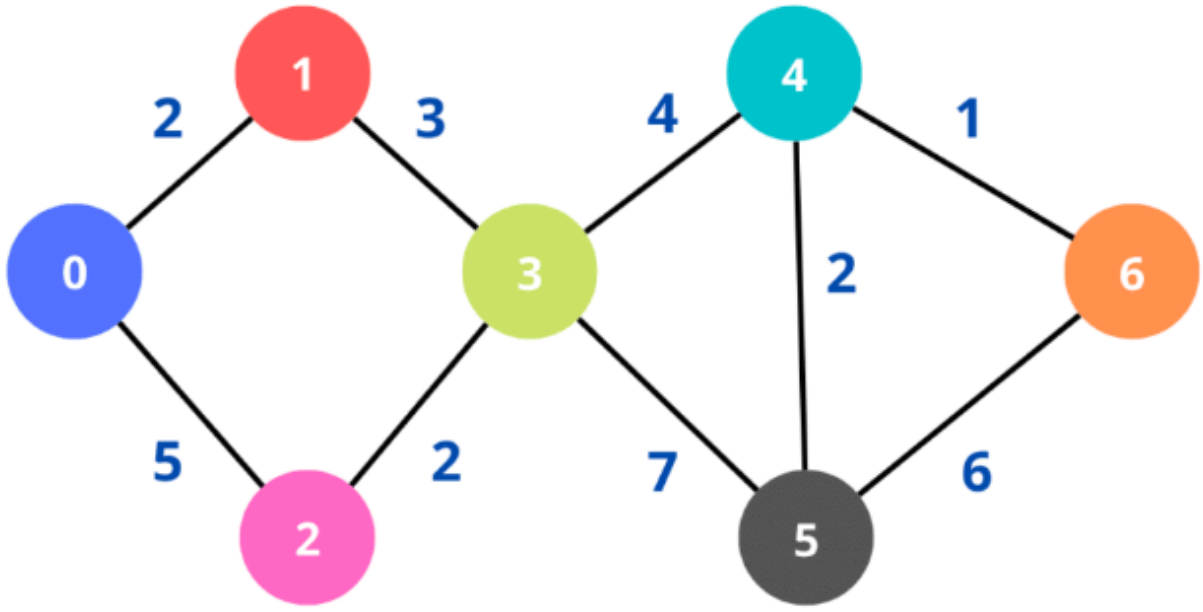
Exemplo de Uso

```
Grafo g(5);
g.adicionarAresta(0, 1, 10);
g.adicionarAresta(0, 4, 5);
g.adicionarAresta(1, 2, 1);
// ...

g.dijkstra(0);
```

- Cria um grafo com 5 vértices.
- Adiciona arestas com seus respectivos pesos.
- Executa o algoritmo a partir do vértice 0.

Imagem do Exemplo



Dijkstra no Grafo da Imagem

- Suponha que queremos encontrar o caminho mais curto do vértice **0** até os demais.
- Inicializamos:
  - Distância de 0 para $0 = 0$
  - Distância para os outros $= \infty$
- Passo a passo:
  1. De 0, temos: $0 \to 1$ (2), $0 \to 2$ (5)
  2. Atualiza: dist[1] $= 2$, dist[2] $= 5$
  3. Próximo menor: vértice 1 (distância 2)
  4. De 1: $1 \to 3$ $(2 + 3 = 5)$, melhor que ∞
  5. Atualiza: dist[3] $= 5$
  6. Continua esse processo até visitar todos os vértices
- O algoritmo termina quando todos os vértices forem visitados.