Documentação do Trabalho Prático 2

Algoritmos I - 2025/1

Victor Guedes Batista
Matrícula: 2020070817

18 de maio de 2025

Introdução

Este documento apresenta a solução para o **Trabalho Prático 2** da disciplina de Algoritmos I, cujo objetivo é determinar o número mínimo de soldados necessários para proteger a capital de um reino fictício contra invasores.

O problema é modelado como um grafo, onde cada célula do mapa representa um custo de defesa. A solução utiliza algoritmos de fluxo máximo para calcular o corte mínimo que isola a capital do restante do mapa.

Os capítulos seguintes detalham a modelagem, a solução implementada, a análise de complexidade e as considerações finais.

Modelagem

O problema foi modelado como um grafo direcionado, onde cada célula do mapa é representada por dois nós: *vin* (entrada) e *vout* (saída). Essa separação permite definir capacidades específicas para cada célula e suas conexões.

Representação do Grafo

- Cada célula é conectada de *vin* para *vout* com capacidade igual ao custo de defesa.
- Células vizinhas são conectadas de *vout* para *vin* com capacidade infinita.
- Um nó *source* conecta as bordas do mapa (exceto montanhas) com capacidade infinita.
- O nó *vout* da capital é conectado ao *sink* com capacidade infinita.

Estrutura de Dados

O grafo foi representado por uma lista de adjacência, onde cada nó armazena apenas as arestas conectadas a ele e suas respectivas capacidades. Essa abordagem reduz o uso de memória, especialmente para grafos esparsos. Os índices dos nós foram calculados como:

- Cada célula do mapa é representada por um nó único.
- As conexões entre células vizinhas são representadas por arestas com capacidade infinita.

Algoritmos Utilizados

Para resolver o problema, foi utilizado o algoritmo de **Edmonds-Karp**, uma implementação do algoritmo de fluxo máximo de Ford-Fulkerson baseada em busca em largura (BFS). A seguir, descrevemos os principais passos do algoritmo:

1. Inicializar o grafo residual com as capacidades fornecidas.
2. Enquanto houver um caminho aumentador da *source* para o *sink* no grafo residual:
   (a) Encontrar o caminho aumentador utilizando BFS.
   (b) Determinar a capacidade de gargalo do caminho.
   (c) Atualizar as capacidades residuais ao longo do caminho.
3. O fluxo máximo é a soma das capacidades de todos os caminhos aumentadores encontrados.

Tradução do Problema

A tradução do problema para o grafo pode ser resumida nos seguintes passos:

1. Cada célula do mapa é transformada em dois nós (*vin* e *vout*).
2. As conexões entre células vizinhas são representadas por arestas com capacidade infinita.
3. As células nas bordas do mapa são conectadas à *source*.
4. A célula da capital é conectada ao *sink*.

Essa modelagem permite que o problema seja resolvido como um problema de fluxo máximo, onde o corte mínimo no grafo corresponde ao número mínimo de soldados necessários para proteger a capital.

Solução

A solução utiliza o algoritmo de **Edmonds-Karp** para calcular o fluxo máximo no grafo modelado. O fluxo máximo corresponde ao custo do corte mínimo, que é o número mínimo de soldados necessários para proteger a capital.

Passos da Solução

1. **Construção do Grafo**: Cada célula válida (com custo maior que 0) é representada por um nó. As conexões entre células vizinhas são adicionadas com capacidade infinita, e as bordas do mapa são conectadas ao nó *source*. A célula da capital é conectada ao nó *sink*.
2. **Cálculo do Fluxo Máximo**: O algoritmo de Edmonds-Karp encontra caminhos aumentadores no grafo residual e atualiza as capacidades.
3. **Interpretação do Resultado**: O valor do fluxo máximo é o custo do corte mínimo.

A implementação foi realizada em Python, utilizando uma matriz de adjacência para representar o grafo e BFS para encontrar caminhos aumentadores.

Pseudocódigo do Algoritmo

A seguir, apresentamos o pseudocódigo do algoritmo de Edmonds-Karp utilizado na solução:

```
EdmondsKarp(G, source, sink):
    Inicializar fluxo máximo como 0
    Enquanto houver um caminho aumentador no grafo residual:
        Encontrar o caminho aumentador usando BFS
        Determinar a capacidade de gargalo do caminho
        Atualizar as capacidades residuais ao longo do caminho
        Adicionar a capacidade de gargalo ao fluxo máximo
    Retornar o fluxo máximo
```

Detalhes da Implementação

A implementação foi realizada em Python e segue os seguintes passos:

- A função `parse_input_corrected` lê os dados de entrada e converte as coordenadas da capital para índices baseados em zero.
- A matriz de adjacência `capacity_adj_matrix` é construída para representar o grafo, com base na modelagem descrita no capítulo anterior.
- A função `edmonds_karp_max_flow` calcula o fluxo máximo entre o *source* e o *sink*, utilizando BFS para encontrar caminhos aumentadores.
- A função `solve` integra todas as etapas, retornando o número mínimo de soldados necessários para proteger a capital.

Essa abordagem garante que o problema seja resolvido de forma eficiente e correta, utilizando conceitos sólidos de teoria dos grafos.

Análise de Complexidade

Neste capítulo, analisamos a complexidade assintótica de tempo e memória da solução implementada para o problema proposto. A análise considera as três etapas principais da solução: construção do grafo, execução do algoritmo de Edmonds-Karp e interpretação do resultado.

Construção do Grafo

A construção do grafo envolve a criação de uma lista de adjacência para representar as capacidades entre os nós. Cada célula válida (com custo maior que 0) é representada por um nó, e as conexões entre células vizinhas são adicionadas com base nas regras do problema.

- **Tempo**: Para cada célula válida, verificamos seus vizinhos (até 4 vizinhos por célula). Assim, o tempo necessário para construir o grafo é proporcional ao número de células válidas no mapa, ou seja, $O(n \times m)$, onde n é o número de linhas e m é o número de colunas.

- **Memória**: A lista de adjacência armazena apenas as arestas existentes, resultando em um espaço de $O(V + E)$, onde V é o número de nós e E é o número de arestas. Isso é significativamente menor do que o $O(V^2)$ necessário para uma matriz de adjacência.

Execução do Algoritmo de Edmonds-Karp

O algoritmo de Edmonds-Karp utiliza busca em largura (BFS) para encontrar caminhos aumentadores no grafo residual.

- **Tempo**: O algoritmo realiza até $O(E)$ iterações de BFS, onde E é o número de arestas no grafo. Cada BFS percorre no máximo $O(V)$ nós, onde V é o número de nós no grafo. Assim, a complexidade total do algoritmo é $O(V \times E^2)$.
- **Memória**: O grafo residual é armazenado como uma lista de adjacência, ocupando $O(V + E)$ de espaço. Isso é mais eficiente do que o $O(V^2)$ necessário para uma matriz de adjacência.

Interpretação do Resultado

A interpretação do resultado consiste em retornar o valor do fluxo máximo calculado pelo algoritmo de Edmonds-Karp. Essa etapa é constante em termos de tempo e memória.

- **Tempo**: $O(1)$.
- **Memória**: $O(1)$.

Complexidade Total

A complexidade total da solução é dominada pela execução do algoritmo de Edmonds-Karp, que possui:

- **Tempo Total**: $O((n \times m)^3)$, considerando que o número de nós V e o número de arestas E são proporcionais ao tamanho do mapa.
- **Memória Total**: $O(V + E)$, devido ao uso de uma lista de adjacência para representar o grafo. Isso é significativamente mais eficiente do que o $O((n \times m)^2)$ necessário para uma matriz de adjacência.

Discussão

Embora a solução seja eficiente para mapas de tamanho moderado, a complexidade cúbica em relação ao número de células do mapa pode se tornar um gargalo para mapas muito grandes. No entanto, a escolha do algoritmo de Edmonds-Karp foi adequada para o problema, pois garante a correção e permite uma implementação relativamente simples, atendendo aos requisitos do trabalho prático.

Considerações Finais

Este trabalho foi uma experiência desafiadora e enriquecedora, permitindo aplicar conceitos de grafos e algoritmos de fluxo máximo.

Partes Mais Fáceis

A leitura do problema e a implementação da estrutura de dados foram diretas, graças à clareza dos requisitos e à modelagem bem definida.

Partes Mais Difíceis

A modelagem do grafo foi desafiadora, especialmente ao otimizar o uso de memória. A substituição da matriz de adjacência por uma lista de adjacência foi uma decisão importante para reduzir o consumo de memória e melhorar a eficiência, mas exigiu ajustes na implementação.

Aprendizados

A otimização do código para lidar com limitações de memória foi um aprendizado significativo. A transição para uma lista de adjacência demonstrou a importância de escolher estruturas de dados adequadas para problemas específicos.

Referências

As seguintes referências foram utilizadas para o desenvolvimento deste trabalho:

- Slides da disciplina Algoritmos I, disponível no Moodle.
- Documentação oficial do Python (`https://docs.python.org/3/`).

Essas fontes foram fundamentais para compreender os conceitos teóricos e implementar a solução de forma eficiente e correta.