PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Trabalho Prático 02/2025
Algoritmos em Grafos

Professora: Edwaldo Soares Rodrigues

Alunos:
Alan Torres de Sá
Arthur Machado Ferreira
Vitor Ribeiro Lacerda

Belo Horizonte
Dezembro de 2025

4) II – RESOLUÇÃO DOS PROBLEMAS

I. Roteamento com Menor Custo

Algoritmo Utilizado: Dijkstra

Pois deseja-se determinar a rota de menor custo entre dois pontos da rede logística, considerando o peso das arestas. Ou seja, no grafo direcionado sem pesos negativos, deve-se encontrar um caminho mínimo entre dois vértices no grafo.

Implementação:
Inicialização das distâncias com infinito (exceto o vértice de origem), uso de fila de prioridade, relaxamento de arestas e reconstrução do caminho pelo dicionário de predecessores. Utilizando este algoritmo, chegamos no resultado desejado.

II. Capacidade Máxima de Escoamento

Algoritmo Utilizado: Edmonds-Karp

Para calcular o fluxo máximo da origem ao destino, considerando a capacidade de cada aresta. Grafo de fluxo com capacidades, necessidade de obter fluxo máximo e identificar gargalos. Foi utilizado o algoritmo Edmonds-Karp para evitar ciclos infinitos

Implementação:
Construção do grafo residual, busca por caminhos aumentantes utilizando a busca em largura, definição do gargalo, atualização das capacidades e iteração até não existir mais caminho possível.

O resultado esperado (cálculo do fluxo máximo e identificação das arestas do corte mínimo) foi alcançado.

III. Expansão da Rede de Comunicação (Árvore Geradora Mínima)

Algoritmos Utilizados: Kruskal e Prim

Justificativa:
Como a expansão da rede exige ligar todos os pontos de forma mais econômica possível, adotou-se o conceito de árvore geradora mínima. Tanto Kruskal quanto Prim atendem esse objetivo, e por isso foram considerados no processo, sendo utilizados grafos mais esparsos e densos, nesta ordem.

Implementação – Kruskal:

Seleciona-se as arestas em ordem crescente de custo, adicionando apenas aquelas que conectam componentes diferentes. O processo continua até que todas as conexões necessárias sejam formadas sem ciclo. (Utilizado em grafos mais esparsos).

Implementação – Prim:
Parte-se de um vértice inicial e, a cada passo, escolhe-se a aresta de menor custo que expande a árvore para um novo vértice ainda não conectado. (Utilizado em grafos mais densos).

Resultado:
Com o uso desses algoritmos, obtém-se uma rede totalmente conectada com exatamente V-1 arestas e custo mínimo garantido (resultado igual para os dois algorítmos).

IV. Agendamento de Manutenções sem Conflito

Algoritmo Utilizado: Welsh-Powell

Justificativa:
O problema foi modelado como um grafo de conflitos, em que cada cor representa um turno possível. O método Welsh-Powell foi escolhido por ser uma abordagem prática e eficiente para separar atividades que não podem ocorrer simultaneamente.

Implementação:
Primeiro, ordenam-se as rotas pelo número de conflitos. Depois, atribui-se uma cor (turno) para cada rota, sempre escolhendo o menor turno possível e evitando cores que já foram usadas por rotas conflitantes. Em seguida, tenta-se aplicar a mesma cor a outras rotas que não tenham conflito entre si.

Resultado:
Com esse processo, obtém-se uma distribuição organizada das manutenções, utilizando o menor número de turnos possível e garantindo que rotas incompatíveis não sejam agendadas juntas. O resultado atendeu as expectativas.

V. Rota Única de Inspeção

Algoritmo Utilizado: Circuito Euleriano (Hierholzer)

Objetivo: percorrer todas as arestas exatamente uma vez e retornar ao ponto de origem.

Justificativa:
Para que o circuito exista, todos os vértices precisam ter grau par. Quando essa condição é atendida, o algoritmo de Hierholzer garante a construção do percurso de forma simples e eficiente.

Implementação:
Primeiro verifica-se o grau dos vértices. Em seguida, aplica-se o algoritmo começando por um vértice qualquer, removendo arestas à medida que são percorridas e retornando por caminhos anteriores quando necessário.

Resultado:
Uma sequência de vértices que utiliza cada aresta uma única vez e forma um circuito fechado.

Algoritmo Utilizado: Circuito Hamiltoniano via Busca em profundidade

Justificativa:
Como não há regras simples que garantam a existência desse tipo de circuito, utiliza-se uma busca. A abordagem com DFS e backtracking permite explorar possibilidades e encontrar uma solução sempre que ela existir.

Implementação:
A busca começa por um vértice e tenta construir o caminho passo a passo, retrocedendo sempre que encontra um impasse (backtracking). O método funciona bem para grafos menores e pode ser otimizado com podas.

Resultado:
Um caminho que passa por todos os vértices exatamente uma vez ou a confirmação de que o circuito não existe.

IV) Relatório técnico

1. Descrição da Rede Logística

A rede logística foi modelada como um grafo direcionado e ponderado, no qual os vértices representam hubs logísticos e as arestas correspondem às rotas disponíveis. Os pesos associados às arestas indicam custo, distância ou tempo, enquanto as capacidades determinam o limite de fluxo. Essa estrutura possibilita a aplicação de diferentes algoritmos para apoiar decisões operacionais e estratégicas.

2. Roteamento de Menor Custo (Dijkstra)

O algoritmo de Dijkstra foi aplicado para identificar o menor custo entre dois hubs. A escolha se justifica pela necessidade de trabalhar com pesos não negativos e pela eficiência computacional apresentada. Esse procedimento permite selecionar rotas de custo reduzido, contribuindo para otimização operacional. Entre suas limitações, destaca-se a impossibilidade de lidar com pesos negativos e a dependência de dados estáticos.

3. Capacidade Máxima (Edmonds-Karp)

Para determinar o fluxo máximo entre dois vértices, foi empregado o algoritmo Edmonds-Karp, uma implementação do método de Ford-Fulkerson com uso de BFS. A técnica permite identificar o valor máximo de escoamento e, adicionalmente, o corte mínimo da rede, apontando gargalos estruturais. O método apresenta custo computacional superior, porém fornece resultados robustos para o cenário analisado.

4. Expansão da Rede (Árvore Geradora Mínima)

A expansão da rede foi modelada como o problema da Árvore Geradora Mínima, resolvido por meio dos algoritmos de Kruskal e Prim. A seleção entre um e outro depende da densidade do grafo. A solução obtida conecta todos os hubs com o menor custo total, evitando ciclos e reduzindo redundâncias estruturais. Após a construção da árvore mínima, recomenda-se a inclusão de rotas adicionais para aumentar a resiliência da rede.

5. Agendamento Sem Conflitos (Coloração – Welsh-Powell)

O problema de agendamento foi representado por um grafo de conflitos. Utilizou-se a heurística Welsh-Powell para realizar a coloração dos vértices, classificando cada manutenção em um turno distinto. A abordagem reduz o tempo total de execução das atividades, embora não garanta solução ótima em todos os casos. Ainda assim, apresenta eficiência adequada para grafos de tamanho moderado.

6. Inspeção da Rede
6.1. Percurso das Rotas (Euleriano)

A verificação das condições eulerianas permitiu determinar se era possível inspecionar todas as rotas percorrendo cada aresta uma única vez. Quando os graus dos vértices atendem aos requisitos, o algoritmo de Hierholzer gera um circuito euleriano de forma eficiente.

6.2. Percurso dos Hubs (Hamiltoniano)

Para a visita única a cada hub, aplicou-se uma busca em profundidade com retrocesso. Este método garante a identificação de um circuito, caso exista, porém apresenta alto custo computacional e escalabilidade limitada.

7. Conclusão

Os algoritmos aplicados permitiram analisar diferentes aspectos da rede logística: custo de rotas, capacidade máxima, expansão com menor custo, agendamento sem conflitos e inspeção completa. A combinação dessas técnicas oferece suporte consistente para decisões operacionais e estruturais. Recomenda-se a atualização periódica dos dados e a integração dos métodos em um sistema automatizado para ampliar sua eficácia.

Por conta da complexidade dos algorítmos e para evitar a criação de diversos arquivos, o grupo optou por salvar os Logs em um único arquivo, sobrescrevendo os logs a cada execução.

Resolução - Beecrowd Problema do Sapateiro Viajante

1. Descrição do Problema
O desafio consiste em ajudar Poly, o Sapateiro, a visitar todas as cidades de Nlogônia exatamente uma vez. Cada cidade pertence a uma ou duas confederações. Para entrar em uma cidade, Poly precisa apresentar um tíquete de uma confederação da qual ela faz parte; ao sair, recebe um tíquete da outra confederação (ou da mesma, se a cidade tiver apenas uma).

O problema é verificar se existe uma cidade inicial e um tíquete inicial que permitam visitar todas as cidades sem repetir nenhuma.

2. Modelagem da Solução
Para resolver, transformamos o problema em um grafo:

Vértices → representam as confederações.

Arestas → representam as cidades, conectando as confederações às quais pertencem.
Se a cidade pertence a apenas uma confederação, ela é modelada como um laço (aresta que começa e termina no mesmo vértice).

Se pertence a duas, é uma aresta normal entre dois vértices.

Assim, visitar todas as cidades uma vez equivale a percorrer todas as arestas uma única vez. Esse é exatamente o conceito de um caminho euleriano.

3. Condições de Existência
Segundo a teoria dos grafos:

Um grafo tem circuito euleriano se todos os vértices têm grau par.

Um grafo tem caminho euleriano se exatamente dois vértices têm grau ímpar.

Se houver mais que dois vértices ímpares, não existe solução.

Nosso código verifica essas condições antes de tentar construir o caminho.

4. Algoritmo Utilizado – Fleury Otimizado
Escolhemos o Algoritmo de Fleury, que constrói o caminho euleriano passo a passo:

Escolha do vértice inicial

Se houver dois vértices ímpares, começa em um deles.

Se todos forem pares, pode começar em qualquer vértice.

Percurso das arestas

Em cada passo, escolhe uma aresta não usada.

Evita usar pontes (arestas críticas que desconectam o grafo) até que seja inevitável.

Marca a aresta como usada e avança para o próximo vértice.

Finalização

Continua até usar todas as arestas (cidades).

Se o caminho cobre todas as cidades e começa na cidade inicial escolhida, essa cidade é a resposta.

Caso contrário, imprime -1.

5. Lógica do Código

De forma simples, o código faz:

Leitura da entrada: número de cidades e confederações, e quais confederações cada cidade pertence.

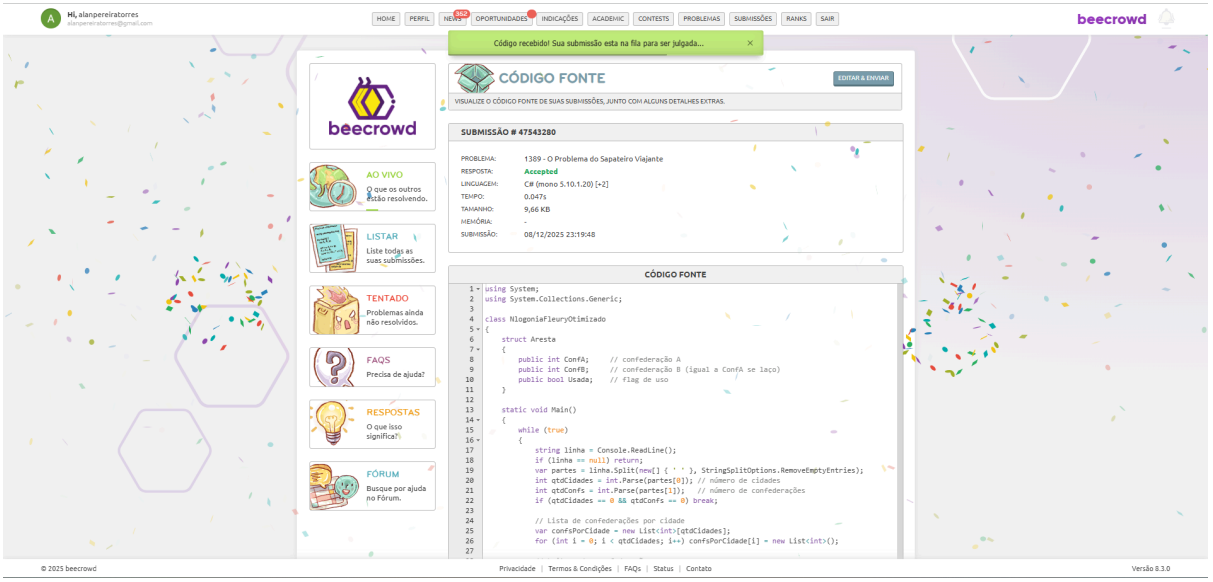
Construção do grafo: cria listas de adjacência ligando confederações pelas cidades.

Verificação de graus e conectividade: garante que o grafo atende às condições de Euler.

Teste de cidades iniciais: simula começar em cada cidade com cada tíquete possível.

Execução do Fleury: percorre todas as arestas sem repetir, evitando pontes.

Validação final: se todas as cidades foram usadas, imprime a cidade inicial; caso contrário, imprime -1.


Hi, alanpereiratorres
HOME
PERFIL
NEWS
OPORTUNIDADES
INDICAÇÕES
ACADEMIC
CONTESTS
PROBLEMAS
SUBMISSÕES
RANKS
SAIR
beecrowd
Código recebido! Sua submissão esta na fila para ser julgada...
beecrowd
AO VIVO
O que os outros estão resolvendo.
LISTAR
Liste todas as suas submissões.
TENTADO
Problemas ainda não resolvidos.
FAQS
Precisa de ajuda?
RESPOSTAS
O que isso significa?
FÓRUM
Busque por ajuda no Fórum.
CÓDIGO FONTE
EDITAR & ENVIAR
VISUALIZE O CÓDIGO FONTE DE SUAS SUBMISSÕES, JUNTO COM ALGUNS DETALHES EXTRAS.
SUBMISSÃO # 47543280
PROBLEMA: 1389 - O Problema do Sapateiro Viajante
RESPOSTA: Accepted
LINGUAGEM: C# (mono 5.10.1.20) [+2]
TEMPO: 0,047s
TAMANHO: 9,66 KB
MEMÓRIA: -
SUBMISSÃO: 08/12/2025 23:19:48
CÓDIGO FONTE
using System;
using System.Collections.Generic;
class NlogoniaFleuryOtimizado
{
struct Aresta
{
public int ConfA; // confederação A
public int ConfB; // confederação B (igual a ConfA se laço)
public bool Usada; // flag de uso
}
static void Main()
{
while (true)
{
string linha = Console.ReadLine();
if (linha == null) return;
var partes = linha.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
int qtdCidades = int.Parse(partes[0]); // número de cidades
int qtdConfs = int.Parse(partes[1]); // número de confederações
if (qtdCidades == 0 && qtdConfs == 0) break;
// Lista de confederações por cidade
var confsPorCidade = new List<int>[qtdCidades];
for (int i = 0; i < qtdCidades; i++) confsPorCidade[i] = new List<int>();
© 2025 beecrowd
Privacidade | Termos & Condições | FAQs | Status | Contato
Versão 8.3.0