UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS CRATEÚS

DISCIPLINA: ALGORITMOS EM GRAFOS - CRT0390

PROF.: RAFAEL MARTINS BARROS

Francisco Romário Rodrigues Lopes

Diogo Bezerra da Costa

Rodolfo Rodrigues de Araujo

Hércules Bruno Ferreira Norte

SOLUÇÃO APROXIMADA PARA O PROBLEMA DO CAIXEIRO VIAJANTE

CRATEÚS

2025

Francisco Romário Rodrigues Lopes
Diogo Bezerra da Costa
Rodolfo Rodrigues de Araujo
Hércules Bruno Ferreira Norte

SOLUÇÃO APROXIMADA PARA O PROBLEMA DO CAIXEIRO VIAJANTE

Relatório sobre o uso do algoritmo de Christofides para obter uma solução aproximada para o problema do caixeiro viajante, apresentado na disciplina de algoritmos em grafos, ministrada pelo professor Rafael Martins na Universidade Federal do Ceará - campus Crateús. Este relatório servirá de critério avaliativo para aprovação da equipe na disciplina.

CRATEÚS

2025

RELATÓRIO: SOLUÇÃO APROXIMADA PARA O PROBLEMA DO CAIXEIRO VIAJANTE

Introdução

O Problema do Caixeiro Viajante (TSP) é um dos desafios mais estudados e conhecidos na área da ciência da computação e otimização combinatória. Ele busca o ciclo de menor custo que visita cada vértice de um grafo exatamente uma vez. Este trabalho foca em uma variante específica: o Problema do Caixeiro Viajante Métrico (Delta-TSP). Nesta condição, os pesos das arestas satisfazem a desigualdade triangular, garantindo que a distância direta entre dois vértices nunca seja maior que a distância passando por um terceiro.

Devido à sua natureza NP-difícil, mesmo em sua versão métrica, o Delta-TSP não possui um algoritmo de solução ótima em tempo polinomial conhecido. Contudo, algoritmos de aproximação se mostram ferramentas valiosas. Nosso objetivo neste projeto foi implementar o Algoritmo de Christofides, que oferece uma solução aproximada com uma garantia teórica de custo no máximo 1,5 vezes o custo ótimo, tornando-o uma das melhores heurísticas para o problema.

O código-fonte foi desenvolvido em Python, utilizando o VS Code como ambiente de desenvolvimento integrado (IDE) para otimizar a escrita e organização. Conforme as diretrizes do trabalho, evitamos o uso de bibliotecas prontas para operações fundamentais como representação e manipulação de grafos, cálculo da Árvore Geradora Mínima (AGM) e determinação de ciclos eulerianos. A única exceção permitida e utilizada foi a biblioteca `networkx` para a etapa específica de determinação do emparelhamento perfeito de custo mínimo, uma parte complexa que se beneficia enormemente de uma implementação otimizada.

Implementação

A solução foi estruturada de forma modular, com cada etapa do Algoritmo de Christofides encapsulada em uma função Python. Essa abordagem facilitou o desenvolvimento, o teste e a compreensão do fluxo do algoritmo. Abaixo, detalhamos cada uma dessas etapas conforme implementadas:

ETAPA 0: Representação do Grafo e Leitura do Arquivo (`ler_grafo_de_arquivo`)

Esta etapa inicial é responsável por carregar a instância do problema a partir de um arquivo de texto. O arquivo de entrada segue um formato padronizado: a primeira linha contém o número de vértices (N), e as N linhas subsequentes representam a matriz de adjacências do grafo, onde cada valor corresponde ao peso da aresta entre os vértices. A função realiza a leitura, converte os pesos para inteiros e retorna a matriz de adjacências. É fundamental para preparar os dados para as etapas seguintes.

ETAPA I: Árvore Geradora Mínima (AGM) (`algoritmo_prim`)

Nesta fase, construímos a Árvore Geradora Mínima (AGM) do grafo completo utilizando uma implementação do Algoritmo de Prim. A AGM conecta todos os vértices do grafo com o custo total mínimo, sem formar ciclos. Ela serve como a base inicial para a construção do nosso ciclo Hamiltoniano. A implementação mantém o controle dos pais (`parent`), dos menores pesos para alcançar um vértice (`key`) e dos vértices já incluídos na AGM (`mst_set`).

ETAPA II: Vértices de Grau Ímpar (`encontrar_vertices_grau_impar`)

Após a construção da AGM, identificamos os vértices que possuem um número ímpar de arestas incidentes (grau ímpar). É uma propriedade fundamental de qualquer grafo que o número de vértices com grau ímpar deve ser par. Este conjunto de vértices é crucial para a próxima etapa, pois são eles que precisam de "ajustes" para que o grafo final possa conter um ciclo euleriano.

ETAPA III: Emparelhamento Perfeito de Custo Mínimo (`encontrar_emparelhamento_perfeito`)

Esta é uma das etapas mais críticas e onde foi permitida a utilização de uma biblioteca externa. Construímos um subgrafo induzido apenas pelos vértices de grau ímpar encontrados na etapa anterior. Em seguida, utilizamos a função `nx.min_weight_matching` da biblioteca NetworkX para encontrar um emparelhamento perfeito de custo mínimo nesse subgrafo. As arestas desse emparelhamento serão adicionadas à AGM para garantir que todos os vértices no multigrafo resultante tenham grau par.

ETAPA IV: Multigrafo Euleriano (`criar_multigrafo_euleriano`)

Nesta etapa, unimos as arestas da Árvore Geradora Mínima (`agm_arestas`) com as arestas obtidas do emparelhamento perfeito (`arestas_emparelhamento`). O resultado dessa união é um multigrafo H, onde pode haver mais de uma aresta entre dois vértices. A característica principal desse multigrafo é que, devido às adições

das arestas do emparelhamento, todos os seus vértices possuem grau par. Esta é uma condição necessária e suficiente para que o grafo possua um ciclo euleriano.

ETAPA V: Ciclo Euleriano (`encontrar_ciclo_euleriano`)

Com o multigrafo H em mãos, determinamos um ciclo euleriano. Este ciclo é um caminho que percorre todas as arestas do multigrafo exatamente uma vez, retornando ao vértice inicial. A implementação utiliza uma abordagem baseada em pilha para encontrar esse ciclo, removendo as arestas à medida que são visitadas.

ETAPA VI: Atalho (Shortcutting) e Cálculo Final (`criar_ciclo_hamiltoniano`)

A etapa final consiste em transformar o ciclo euleriano (que pode visitar vértices repetidamente) em um ciclo Hamiltoniano aproximado. Isso é feito através do processo de "atalho" (shortcutting), onde os vértices que já foram visitados são "pulados" no ciclo euleriano, garantindo que cada vértice apareça apenas uma vez no ciclo Hamiltoniano final. A função também calcula o custo total desse ciclo Hamiltoniano, que representa a solução aproximada para o TSP.

Testes Realizados para Validação

Para validar a correta implementação das etapas que não utilizaram bibliotecas externas (AGM e Ciclo Euleriano), foram realizados os seguintes testes:

- **Validação da Árvore Geradora Mínima (AGM):**
  - **Grafos Pequenos e Conhecidos:** Testamos a função `algoritmo_prim` com grafos de pequeno porte (e.g., 3 a 5 vértices) cujas AGMs e custos totais eram facilmente calculáveis e verificáveis manualmente. Isso permitiu identificar e corrigir falhas na lógica de seleção de arestas e no controle de vértices visitados.
  - **Verificação de Conectividade:** Após a construção da AGM, verificamos que todos os vértices estavam conectados e que o número de arestas era `N-1`, como esperado para uma árvore com N vértices.
  - **Comparação de Custos:** Confirmamos se o custo total da AGM calculada correspondia ao custo mínimo esperado para os grafos de teste.
- **Validação da Determinação de Ciclos Eulerianos:**
  - Multigrafos Conhecidos com Propriedade Euleriana: Criamos manualmente (ou adaptamos) pequenos multigrafos que sabidamente possuíam um ciclo euleriano (todos os vértices com grau par). A função `encontrar_ciclo_euleriano` foi executada nesses grafos para

verificar se o ciclo retornado era válido (percorrendo todas as arestas e retornando ao início) e se a ordem dos vértices estava correta.
    - **Verificação de Visita de Arestas:** Durante o debugging, confirmamos que cada aresta estava sendo "consumida" e adicionada ao ciclo uma única vez.
    - **Casos Limite:** Testamos com multigrafos muito pequenos (e.g., um triângulo, um quadrado) para garantir que o algoritmo funcionava corretamente em casos mais simples.

Esses testes manuais e de pequena escala foram fundamentais para garantir a robustez das implementações base, antes de integrá-las ao fluxo completo do algoritmo de Christofides, que lida com as complexidades adicionais do emparelhamento e do atalho.

Biblioteca Escolhida para o Emparelhamento

Para a crucial etapa de determinação do emparelhamento perfeito de custo mínimo (Etapa III), optamos pela biblioteca NetworkX do Python.

- **Justificativa da Escolha:**
    - **Eficiência e Robustez:** O NetworkX é uma biblioteca amplamente reconhecida e otimizada para operações em grafos. Sua implementação de algoritmos de emparelhamento é testada e eficiente, o que é vital para um problema de natureza NP-difícil como o TSP.
    - **Facilidade de Uso:** A interface da função `nx.min_weight_matching()` é intuitiva e se integra perfeitamente ao nosso código, minimizando a complexidade de ter que implementar um algoritmo de emparelhamento perfeito de custo mínimo (como o algoritmo de Edmonds Blossom) do zero, o que estaria além do escopo e dos requisitos do trabalho.
    - **Padrão da Indústria:** Utilizar uma biblioteca padrão de mercado em um componente permitido e complexo é uma prática comum em desenvolvimento de software, permitindo focar nos desafios específicos do algoritmo de Christofides como um todo.

Resultados

O programa desenvolvido aceita um arquivo de texto como entrada, processa o grafo conforme as etapas do Algoritmo de Christofides e exibe o ciclo Hamiltoniano aproximado encontrado, juntamente com seu custo total.

Exemplo de Execução e Análise do Resultado

Considere o `grafo_exemplo.txt` fornecido, que representa uma instância do Delta-TSP. Ao executar o programa com este arquivo, observamos a seguinte saída (exemplo):

Grafo lido com sucesso! Número de vértices: X

Iniciando a Etapa I: Construção da Árvore Geradora Mínima... Pensando na conexão mais barata! 🌳 AGM construída! Conectamos tudo da forma mais econômica. 🌲

Iniciando a Etapa II: Identificação dos Vértices de Grau Ímpar... Quem tem conexão 'sobrando'? 🤔 Vértices de grau ímpar (conjunto I): [..., ..., ...]. Vamos resolver isso!

Iniciando a Etapa III: Emparelhamento Perfeito de Custo Mínimo... Conectando os 'solitários' com inteligência! 🤝 Emparelhamento encontrado! Os 'pares' estão conectados. 🎉

Iniciando a Etapa IV: União das Arestas e Criação do Multigrafo H... Juntando as peças do quebra-cabeça! 🧩 Multigrafo H criado com sucesso. Agora temos um mapa 'percorrível' por completo! 🗺️

Iniciando a Etapa V: Encontrando o Ciclo Euleriano... Explorando cada caminho! 🚶 Ciclo Euleriano encontrado: [..., ..., ..., ..., ...]. Percorremos todas as arestas! 🎉

Iniciando a Etapa VI: Criando o Ciclo Hamiltoniano (Shortcutting)... A rota final está surgindo! 🚀 Calculando o custo da rota final... Quase lá! 💰 Custo da aresta (u, v): peso_uv ... (para todas as arestas do ciclo) Custo total calculado.

Chegamos ao fim da jornada! ✅ ========== RESULTADO FINAL DO ALGORITMO ========== O ciclo Hamiltoniano aproximado é: [..., ..., ..., ..., ...] Essa é a nossa melhor rota! 📍 Custo total do ciclo: XXXX. Valor da nossa aventura! 💲 ==========

Análise do Resultado: O ciclo Hamiltoniano resultante e seu custo total representam a solução aproximada fornecida pelo Algoritmo de Christofides. Como esperado, o algoritmo busca um trade-off entre otimalidade e eficiência computacional. A natureza de aproximação do Christofides garante que o custo obtido estará dentro de 1.5 vezes o valor do custo ótimo

real para o Delta-TSP. Na prática, como observado nas figuras do problema (Figura 2 e Figura 3 do enunciado), a diferença entre a solução ótima e a aproximada por Christofides pode ser bastante pequena, ilustrando a eficácia do algoritmo na obtenção de soluções de alta qualidade em tempo polinomial.

Conclusão

A implementação do Algoritmo de Christofides para o Problema do Caixeiro Viajante Métrico demonstrou ser uma abordagem eficiente e robusta para obter soluções aproximadas para este problema NP-difícil. A modularização do código em etapas claras facilitou o desenvolvimento e a compreensão de cada fase do algoritmo. A aderência às restrições de uso de bibliotecas, exceto para o emparelhamento perfeito de custo mínimo com `networkx`, permitiu aprofundar o conhecimento nos algoritmos fundamentais de grafos, como Prim e a busca de ciclos eulerianos.

Este trabalho não apenas cumpriu os requisitos propostos, mas também proporcionou uma valiosa experiência prática na aplicação de conceitos teóricos de Algoritmos em Grafos para resolver um problema de otimização complexo e com grande relevância prática.