Problemas do Carteiro Chinês e Caixeiro Viajante

Pedro Henrique Oliveira Sousa[1]

[1]Ciência da Computação – Universidade Estadual do Ceará(UECE)
Ceará – CE – Brazil
pedro986@gmail.com

Resumo. *Este relatório explica e soluciona os problemas do Carteiro Chinês e do Caixeiro Viajante. A linguagem utilizada para todos os códigos foi C++.O Caixeiro Viajante precisa encontrar o circuito hamiltoniano, os algoritmos implementados neste problema foram: union-find, dijkstra, dfs, floyd-warshall com paradigmas guloso e programação dinâmica. Alguns algoritmos geram o percurso que deve ser feito, além do seu custo. O Carteiro Chinês precisa encontrar o circuito euleriano, os algoritmos implementados neste problema foram: hierholzer e dijkstra.*

1. Problema do Caixeiro Viajante(PCV)

Este problema busca determinar a menor rota possível saindo de um local (cidade), visitando todos os locais (cidades) que deseja visitar e voltar para o local (cidade) de partida, isso implica que terá o menor custo/esforço para fazer todas as visitas, demonstrando um economia de tempo e custo de transporte/combustível para fazer todas as visitas.

O problema consiste em buscar um Circuito Hamiltoniano de menor custo. Um Circuito Hamiltoniano é um caminho que começa e termina no mesmo vértice. A ideia do algoritmo é iniciar a busca em qualquer local(cidade) e visitar todas os locais(cidades) mapeadas nos dados do sistema e retornar para a cidade inicial, como é um ciclo pode-se começar de qualquer vértice, pois se o inicial não for o correto pode-se apenas locomover o caminho para o ponto inicial desejado.

Este problema é NP-hard o que significa que não existe uma solução polinomial para o problema, ou seja, é muito custoso calcular o resultado exato do problema dependendo da quantidade de locais(cidades) que se deseja visitar, pois a solução irá crescer exponencialmente, tendo como complexidade O(V!) ou $O(V^2*2^V)$, onde 'V' é a quantidade de vértices dados para o programa, a solução O(V!) seria uma busca completa simples, analisando todos os caminhos possíveis e dizendo qual é o melhor, já a solução $O(V^2*2^V)$ é um pouco mais inteligente, ela calcula todas as rotas porém se encontrar um caminho que ela já passou ele simplesmente recupera o valor pré calculado e continua o caminho.

1.1. Algoritmos Implementados

Todos os grafos estão representados na forma de lista de adjacência. Os ciclos estão 0-indexados, ou seja, os vértices vão de 0 a N-1. A função freopen() abre o arquivo sozinho, basta executar o programa.

1.1.1. Programação Dinâmica (PD)

Este algoritmo possui a complexidade citada anteriormente de $O(V^2*2^V)$, a otimização feita comparada com a solução O(V!) é que quando ele faz qualquer calculo de um caminho, ele salva o custo do caminho, para nas próximas vezes que ele precisar daquele trecho ele não irá calcular novamente, economizando tempo de execução, porém gastando mais memória. Essa é a principal ideia da Programação Dinâmica, troca tempo de execução por memória, o paradigma usado para esta solução.

O algoritmo implementado possui um função chamada 'tsp' que vem do nome do problema em inglês '*travelling salesman problem*', possuindo dois dados para calcular os estados e a base da programação dinâmica.

Sendo eles: Uma bitmask (máscara de bits) onde a ideia é dizer se um vértice foi ou não visitado fazendo o teste checando cada bit de um inteiro de 32 bits, dessa forma, podemos alocar até

32 vértices dentro de uma máscara do tipo inteiro. Um id (identificador) que irá indicar em qual vértice está atualmente para pegar a distância dele para o próximo vértice.

A recursão tem como base todos os bits da bitmask setados em 1, ou seja, ter visitado todos os vértices e faltando apenas retornar para o vértice que iniciou a busca, caracterizando uma PD top-down. Onde ela tem como raiz do calculo um vértice e ele como visitado, descendo a árvore da recursão seria os caminhos percorridos pelo caixeiro e as folhas da árvore da recursão é a base da PD seria ter visitado todos os vértices e retornar a distância que fechará o circuito, a resposta ótima final irá subir na árvore fazendo todos os cálculos até a raiz retornando o menor caminho possível desse grafo. A matriz 'memo' serve para guardar os estados da PD, ela vem do nome memorize, que é a função dela, a matriz serve para memorizar tudo que já foi calculado e recuperar os dados quando for necessário.

1.1.2. Kruskal

Este algoritmo tem como complexidade O(E*log V) onde 'E' seria a quantidade de arestas do grafo e 'V' é a quantidade de vértices do grafo, a complexidade é em 'log V' porque está sendo utilizando union-find para agrupar os vértices, essa estrutura deixa a busca do vértice em no máximo log V no total. Este algoritmo é guloso, o que significa que ele não retornar a resposta exata do problema, apenas uma solução aproximada, porém podem ocorrer casos que a solução seja a ótima, mas ele não garante isso.

O algoritmo, como já citado, possui union-find (a função de join e find),para juntar os grupos de vértices, uma dfs(busca em profundidade) para descer no grafo. A organização das arestas nesse algoritmo foi feita através de uma fila de prioridade, ordenando do maior para o menor na fila, para inverter a prioridade apenas recebi o custo da aresta negativa, para o maior valor possível ser 0, dessa forma, posso ver todas as aresta nessa ordem da 'priority_queue' apenas invertendo o custo da aresta, quando dois vértices analisados não estão no mesmo grupo eles iram pertencer a árvore geradora mínima representada pelo grafo 'MST' que significa '*Minimum Spanning Tree*' então junta os dois vértices pela função join, se eles já estiverem no mesmo grupo a aresta será simplesmente ignorada.

Para calcular o ciclo e seu custo aproximados, foi utilizado um 'for()' para partir de todos os vértices e vê qual é o melhor caminho, é feita uma dfs e os vértices postos em pré-ordem pelo vetor 'Pre', se ele possuir o tamanho de um ciclo válido(possuindo todos os vértices) ele é uma possível saída para o circuito de menor custo, no fim, é demonstrado o menor custo encontrado e a rota feita.

1.1.3. Prim

O algoritmo tem complexidade O(E*log V), onde 'E' é o número de arestas e 'V' o número de vértices. Esse algoritmo é muito semelhante ao algoritmo de Dijkstra a única diferença é que Dijkstra pega a soma dos trajetos de um vértice 'u' para um vértice 'v', enquanto Prim pega apenas a última aresta, ou seja, qual é a menor aresta para chegar a um vértice 'v' qualquer.

O algoritmo possui duas funções, dfs(busca em profundidade) e prim que irá indicar qual é a árvore geradora mínima do grafo.

O primeiro passo é converter a matriz de custo da entrada para a forma de um grafo através de lista de adjacência. Em seguida o algoritmo de prim é rodado, ele calcula a menor distância para cada vértices a partir de um vértice inicial,para saber as ligações feitas, acrescentei um vetor chamado 'ponto' para indicar quais ligações foram feitas para futuramente criar o grafo da árvore geradora mínima. Após terminar de rodar prim é gerada a árvore com as arestas sendo bidirecionais. É feita uma aproximação do menor caminho, partindo de todos os vértices e verificando qual seria a escrita do grafo em pré-ordem partindo de qualquer vértice. Calculando o custo desse circuito e escolhendo apenas o melhor deles.

1.1.4. Algoritmo de Christofides

Os passos desse algoritmo são:

- Encontrar a árvore geradora mínima do grafo dado.
- Encontrar o matching mínimo para os vértices de grau impar da árvore.
- Adicionar essas arestas na árvore, gerando um multigrafo.
- Encontrar o circuito Euleriano.
- Retirar os vértices repetidos do circuito para torná-lo em Hamiltoniano.

A busca deste algoritmo é um circuito euleriano para gerar o circuito hamiltoniano, para isso temos que saber que um circuito euleriano sempre existe quando todos os vértices do grafo possuem grau par, existi uma trilha indo de um vértice impar para o outro se o grafo possuir exatamente 2 vértices de grau impar e nunca existe quando possui mais de 2 vértices de grau impar, para solucionar isso, temos que tornar todos os vértices do grafo de grau impar em grau par, para que o circuito sempre exista.

A estrutura das arestas usada nesta solução foi uma struct para possuir as informações de partida, destino, custo e uso, dessa forma, posso facilmente checar se uma aresta já foi passada em qualquer um dos seus sentidos, pois como as arestas são criadas aos pares(nos dois sentidos), o ultimo bit do índice do vetor que as arestas estão diz apenas os dois sentidos daquela aresta, podendo setar os dois sentidos como utilizados com a operação binária (xor 1) para trocar o ultimo bit e setar como já utilizada a mesma aresta, porém no sentido oposto dela.

Para achar a árvore geradora mínima foi utilizado o algoritmo de Kruskal por ser curto e simples, em seguida, recolhe todos os vértices de grau impar em um vetor, para achar o matching mínimo entre eles. No passo seguinte que seria encontrar o caminho e adicionar as arestas no grafo há um problema, a quantidade de vértices impares dessa matriz dada, pois a solução ideal para este algoritmo seria um busca completa com todas as possíveis combinações de vértices e seus caminhos, como a quantidade de vértices é muito grande, 16 vértices impares, é inviável a solução com busca completa, pois a complexidade seria O(16!) apenas para gerar as permutações, sem contar com a complexidade da dijkstra que é O(E* log V). Para solucionar este problema resolvi usar o algoritmo de Floyd-Warshall para calcular a matriz de distância em $O(V^3)$ e usar os caminhos gerados por ela para encontrar uma solução gulosa das ligações que devem ser feitas para ligar aos pares, todos os 16 vértices.

O ideal que seria a busca completa que está comentada no código, podendo apenas tirar seu comentário e comentar a parte demarcada com /*Marcação da alteração*/ para vê o funcionamento do código com um matriz que tenha menos vértices de grau impar.

2. Problema do Carteiro Chinês(PCC)

Esse problema foi desenvolvido pelo matemático chinês Kuan Mei-Ko, em 1962. O problema consiste em percorrer partindo de um local, visitar todas as arestas do grafo, com o menor custo possível, terminando no local de partida, isto é um ciclo Euleriano.

Um circuito Euleriano usa todas as arestas de um grafo. Para ser Euleriano, todos os vértices do grafo devem ser de grau par. Se o grafo tem dois vértices de grau ímpares, então o grafo é dito semi-euleriano. Uma tilha pode ser desenhada a partir de um dos vértices ímpares e terminando no outro vértice ímpar.

Logo, se o grafo não possuir todos só vértices de grau par precisamos adicionar arestas, tornando-o um multigrafo, para passar mais de uma vez pela mesma aresta e concluir o ciclo, para isso, verifica-se apenas os vértices de grau impar, traçando um caminho entre eles, aos pares, pois num grafo conexo, se existirem vértices de grau ímpar eles sempre serão múltiplos de dois. Com isso temos que a solução mais simples desse problema é usar ‘next_permutation()’ para gerar todas as combinações possíveis e dijkstra para calcular a distância entre estes pontos aos pares. Porém se a quantidade de vértices impares for grande, o programa é muito lento, pois gerar todas as permutações tem um O(N!), onde ‘N’ é a quantidade de vértices de grau ímpar.

2.1. Algoritmos implementados

Os algoritmos utilizam a analise dos grafos em forma de lista de adjacência. O ciclo está 0-indexado, ou seja, os vértices vão de 0 a N-1.A função freopen() abre o arquivo sozinho, basta executar o programa.

2.1.1. Dijkstra com memorize

Neste algoritmo a matriz de custo foi transformada em lista de adjacência e distâncias infinitas foram desconsideradas para forma o grafo. Como os cálculos devem ser feitos apenas nos vértices de grau impar, a primeira coisa a ser feita é separa esses vértices em um vetor.

Como todas as arestas serão utilizadas pode-se calcular parte da soma da resposta apenas pela matriz de entrada, somando todos as arestas. Para calcular a segunda parte que seria as arestas adicionadas entre os vértices de grau ímpar, separo todos estes vértices e calculo tudo usando dijkstra e ‘next_permutation()’. Porém o algoritmo de Dijkstra existe algumas pequenas modificações para torná-lo mais eficiente, é posto uma matriz chamada ‘memo’, que vem de memorize, onde ela salva todos os caminhos que estão sendo calculados pela dijkstra, evitando recálculos, torando-a muito mais eficiente. A melhor escolha após testar todas as permutações será a de menor custo, que será somada na resposta final do problema.

2.1.2. Dijkstra com Hierholzer

Neste algoritmo foi montada um estrutura para trabalhar com as arestas e conseguir com facilidade determinar quais foram as arestas que já foram utilizadas para não passar por elas novamente, esta estrutura possui quatro inteiros, de onde ele veio ‘p’(de partida), para onde ela vai ‘d’(de destino), o custo da aresta ‘c’(de custo) e se foi ou não utilizada ‘u’(de utilizada). Dessa forma, o grafo é montado através dos dados dessa aresta, utilizando um vetor para guardar todas as elas e mantendo apenas o índice do vetor dentro dos vetores de adjacência que formam o grafo, fazendo as ligações bidirecionais, posso acessar o sentido oposto da aresta (para indicar que ela já foi utilizada) , apenas com uma operação bit-a-bit (xor 1), dessa forma, basta trocar o primeiro bit que indica que o número é par ou impar para dizer que os dois sentidos da aresta já foram utilizados, ou seja, a aresta não pode mais ser utilizada.

Neste algoritmo a matriz de custo dada é transformada num grafo em forma de lista de adjacência, porém a adição das arestas é feita de forma diferente, os vértices possuem apenas o índice que indica dentro do vetor ‘Arestas’ as informações que a aresta possui. Como o circuito formado é euleriano, guardo todas os vértices de grau ímpar para calcular a menor distância entre eles, dois a dois, usando ‘next_permutation()’ e dijkstra, apenas aumentando o retorno da função, com o caminho que está sendo feito, para acrescentar no grafo. O melhor conjunto de ligações feitas é acrescentado no grafo, tornando-o um multigrafo e permitindo achar um circuito Euleriano nele.

Para determinar o circuito utilizei o algoritmo de Hierholzer com complexidade de O(E+V). Esse algoritmo gera um ciclo qualquer dentro do grafo, como sobram arestas não percorridas, devemos recomeçar a partir de um vértices que já está no ciclo. Assim, o novo ciclo encontrado será colocado na posição em que ele foi começado, gerando o circuito Euleriano após percorrer todas as arestas, com os acréscimos de ciclos encontrados dentro do grafo.

3. Resultados

3.1. Caixeiro Viajante(PCV)

A tabela a seguir possui uma comparação entre todos os algoritmos implementados, Programação Dinâmica(PD), Kruskal, Prim, Christofides. O tempo está em milissegundos e é um valor aproximado de sua duração, dependendo da execução pode variar um pouco este valor, circuito é se o algoritmo calcula a rota feita pelo caixeiro, aproximação é o quanto a resposta é próxima do ideal em porcentagem, onde a resposta ideal representa 100%, ou seja, o limite dessa análise é 200%,

acima disso a solução do algoritmo não é válida para o problema. O quão mais próximo de 100% melhor é a solução do algoritmo.

Algoritmos	Tempo (ms)	Circuito	Aproximação
PD	1960	Não	100
Kruskal	6	Sim	185
Prim	5	Sim	173
Christofides	8	Sim	157

O algoritmo usando Programação Dinâmica é o mais lento, por testar todos os caminhos, porém é o que sempre garante dar a resposta exata ao problema, o restante são soluções aproximadas.

O circuito que o Kruskal encontra é: 5 11 16 19 15 18 17 1 9 7 0 14 10 13 4 6 12 8 3 2 5, começando e terminando o circuito no vértice 5, com os vértices 0-indexados. Com a solução de 61 para o custo do caminho, tendo uma aproximação de 185%.

O circuito que o Prim encontra é: 10 4 15 7 18 17 1 9 0 14 19 6 12 3 8 16 2 11 5 13 10, começando e terminando o circuito no vértice 10, com os vértices 0-indexados. Com a solução de 57 para o custo do caminho, tendo uma aproximação de 173%.

O circuito que o Christofides encontra é: 9 1 17 18 15 19 16 11 5 14 0 10 13 4 3 12 6 2 7 8 9, começando e terminando o circuito no vértice 9, com os vértices 0-indexados. Com a solução de 52 para o custo do caminho, tendo uma aproximação de 157%.

3.2. Carteiro Chinês(PCC)

A tabela a seguir possui uma comparação entre os algoritmos implementados, Dijkstra com memorize e Dijkstra com Hierholzer. O tempo está em milissegundos e é um valor aproximado de sua duração, dependendo da execução pode variar um pouco este valor, circuito é se o algoritmo calcula a rota feita pelo carteiro. Como a resposta é exata, diferente do algoritmo anterior, ambos possuem como resposta 632 de distância como resposta.

Algoritmos	Tempo (ms)	Circuito
Dijkstra	5	Não
Hierholzer	275	Sim

O algoritmo de Dijkstra é extramente rápido por usar a matriz de memorize e evitar cálculos repetidos. O com Hierholzer é bem mais lento por não possuir a matriz memorize em sua dijkstra. O circuito que Hierholzer encontra é: 0 18 19 13 18 17 10 19 9 18 16 17 15 17 14 18 12 16 15 14 16 13 17 10 18 8 17 9 15 11 2 18 7 19 3 18 6 16 11 14 13 12 11 13 10 16 8 14 5 17 7 16 5 15 10 12 9 13 7 12 8 11 10 9 11 7 10 8 9 7 8 6 15 4 1 18 4 17 3 16 2 14 4 13 5 12 6 11 4 10 6 9 4 7 6 5 8 2 12 1 17 0 16 1 15 3 13 0 11 3 10 2 11 1 10 0 8 1 9 2 7 5 4 6 2 5 3 7 1 5 0 4 3 2 4 1 3 0 2 1 0. Como a resposta é exata, o ciclo sempre será esse, a única diferença que pode existir é o ponto inicial do ciclo.

4. Conclusão

Os problemas podem ser solucionados de diversas maneiras, a melhor solução depende da situação que ele está aplicado. O problema do caixeiro viajante, é usado para movimento de pessoas, materiais e veículos, e no turismo(nos pacotes de pesquisa de viagens pessoais), dependendo do número de locais que irá passar, usar a solução exata com Programação Dinâmica pode ser uma solução viável, se a quantidade de cidades for muito grande, uma solução gulosa poderá ser válida, depende de enquanto tempo a resposta deve ser dada. Para algoritmos que devem dar respostas instantâneas, por ser um problema NP-hard, dificilmente será utilizada a solução com programação dinâmica.

O problema do carteiro chinês também possui suas aplicações, sendo alguns exemplos delas: coleta de lixo, limpeza das ruas e patrulhamento policial. Dependendo da quantidade de vértices de grau impares a solução apresentada é válida para saber exatamente qual caminho dever ser precorrido para solucionar este tipo de problema.

Referências

http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/Grafos/EulerHam/euler_ham.html
https://www.geeksforgeeks.org/chinese-postman-route-inspection-set-1-introduction/
https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/
https://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/
http://www.suffolkmaths.co.uk/pages/Maths%20Projects/Projects/Topology%20and%20Graph%20Theory/Chinese%20Postman%20Problem.pdf
https://pt.wikibooks.org/wiki/Log%C3%ADstica/Sistemas_de_distribui%C3%A7%C3%A3o/Problema_do_caixeiro_viajante/Problema_do_caixeiro_viajante:_Aplica%C3%A7%C3%B5es
http://www.ebah.com.br/content/ABAAAhKUkAF/caixeiro-viajante-carteiro-chines