Trabalho Prático 3

Sarah Azevedo Pereira

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

sarahazevedo@ufmg.br

1 Introdução

A DeliveryExpress, uma *startup* no ramo de e-commerce, está em busca de uma maneira de alocar os pontos de distribuição centrais das suas vendas online. Isso seria feito com o objetivo de estabelecer o ponto de distribuição central de uma região, e a partir dele, mandar caminhões com as entregas para todas as outras cidades daquela região, seguindo uma rota que passa por cada uma apenas uma vez e volta à cidade de origem.

Assim, dado uma região com cidades e estradas, sendo que uma estrada entre duas cidades é caracterizada por uma distância $d \in \mathbb{N}$, o problema consiste em determinar qual a menor rota que passa por todas as cidades sem repetir nenhuma e retorna a origem. A solução proposta envolveu explorar três formas diferentes de se decidir isso. A primeira delas foi por meio de força bruta, a segunda, por meio de programação dinâmica e a terceira fez o uso de uma estratégia gulosa.

2 Modelagem

O trabalho foi desenvolvido na linguagem C e compilado pelo compilador gcc na versão 9.4.0 (Ubuntu 9.4.0-1ubuntu1 20.04.2) em um processador Intel® Core™ i5-8300H CPU @ 2.30GHz × 8 com 7,6 GB de memória RAM utilizável, sob a distribuição Linux Zorin OS 16.3.

Todas as estratégias utilizadas modelam o problema como um grafo, em que as cidades são os vértices e as estradas são as arestas. O código foi dividido em seus arquivos principais: `brute_force.cpp`, `dinamic.cpp`, `guloso.cpp` e `main.cpp`. Além de seus respectivos arquivos de cabeçalho `brute_force.hpp`, `dinamic.hpp` e `guloso.hpp`.

- `main.cpp`: Contém a função principal do programa, que faz a leitura dos arquivos de entrada e chama os algoritmos de acordo com a primeira linha do arquivo;
- `brute_force.cpp brute_force.hpp`: Implementa a estratégia de força bruta, em que todas as possíveis permutações de caminhos são geradas e a de menor custo é escolhida e retornada;
- `dinamic.cpp dinamic.hpp`: Implementa a solução por meio de programação dinâmica. Nesse sentido, o algoritmo implementado foi o de Held-Karp, que utiliza bitmasks para representar o conjunto de cidades já visitadas e uma tabela de memoização para armazenar os custos mínimos das subsoluções, o que evita cálculos repetitivos;
- `guloso.cpp guloso.hpp`: Implementa a estratégia gulosa. O programa seleciona um vértice de início e, a partir dele, o vértice de menor distância é selecionado. Isso é feito iterativamente

para cada vértice, sempre selecionando o que dista menos do atual. A ideia é retornar uma solução boa o suficiente, mesmo que ela não seja ótima.

3 Solução

Essa seção busca detalhar a ideia e o funcionamento das diferentes abordagens utilizadas.

3.1 Força-Bruta

A abordagem de força bruta testa todas as possíveis rotas e escolhe aquela com o menor custo. Assim, a estratégia garante a solução ótima, mas se torna inviável conforme o número de cidades cresce. O algoritmo é especificado pelo pseudocódigo a seguir:

```
melhor_custo = infinito
melhor_rota = vazio
para cada permutação das cidades:
    custo_atual = calcular_custo_rota(rota)
    se custo_atual < melhor_custo:
        melhor_custo = custo_atual
        melhor_rota = rota
retorna melhor_rota, melhor_custo
```

Assim, o algoritmo gera todas as permutações possíveis das cidades (exceto a cidade inicial). Para cada permutação, ele calcula o custo total da rota. Depois, compara os custos, armazena a menor rota encontrada e, em seguida, retorna a melhor rota e seu custo mínimo.

3.2 Programação Dinâmica (Held-Karp)

Este algoritmo utiliza Programação Dinâmica com Bitmask para evitar recomputações desnecessárias, reduzindo o tempo de execução em comparação à força bruta. Outro ponto importante é que ele armazena subsoluções para melhorar a eficiência. O funcionamento do algoritmo é exemplificado pelo pseudocódigo a seguir:

```
tsp(mask, cidade)
    dp[mask][pos] = infinito
    se todas as cidades foram visitadas:
        retorna custo de volta à cidade inicial
    para cada cidade não visitada:
        novo_custo = custo[pos][cidade] + tsp(mask | cidade, cidade)
    se novo_custo < dp[mask][pos]:
        dp[mask][pos] = novo_custo
        parent[mask][pos] = cidade
    retorna dp[mask][pos]
```

Assim, o algoritmo define uma tabela `dp[mask][pos]`, onde: `mask` representa as cidades já visitadas usando `bitmask` e `pos` é a cidade atual.`dp[mask][pos]` armazena o menor custo para visitar as cidades de `mask`, terminando em `pos`.

Além disso, o algoritmo é recursivo e tem como caso base a condição de que, se todas as cidades

foram visitadas, retorna o custo de voltar à cidade inicial. Assim, para cada cidade ainda não visitada, ele calcula o custo mínimo (recursivamente).

3.3 Algoritmo Guloso (Heurística do Vizinho Mais Próximo)

Este algoritmo escolhe, a cada passo, a cidade mais próxima ainda não visitada. Ele é mais rápido e escalável que os outros dois, mas não garante a solução ótima. O passo a passo do algoritmo pode ser visualizado por meio do pseudocódigo a seguir:

```
rota = [cidade_inicial]
enquanto existirem cidades não visitadas:
    cidade_mais_proxima = encontrar_cidade_mais_proxima(rota_atual)
    adiciona cidade_mais_proxima à rota
retorna à cidade inicial e finaliza rota
```

Assim, a estratégia gulosa é responsável por: iniciar na cidade de origem, repetidamente visitar a cidade mais próxima ainda não visitada e, por fim, retornar à cidade inicial ao final do percurso. Deste modo, ela apresenta melhor escalabilidade que o algoritmo Held-Karp por realizar um menor número de iterações e construir somente uma rota (aplicando a heurística de menor distância).

3.4 Comparação entre as Abordagens

A estratégia de força-bruta usa todas as possibilidades e garante a solução ótima, mas se torna inviável rapidamente devido ao grande número de iterações necessárias. A abordagem de programação dinâmica também emcontra a solução ótima e usa memoização e bitmask para melhorar a eficiência, sendo mais escalável que a estratégia anterior. Já a estratégia gulosa é a mais escalável das três, mas não necessariamente encontra a solução ótima.

4 Análise de Complexidade

1. **Tempo:** A estratégia de **força-bruta** apresenta complexidade $O(n!)$, pois calcula todas as permutações possíveis para os vértices. No caso da estratégia que usa **Held-Karp**, `mask` (bitmask) representa o conjunto de cidades visitadas. Como há n cidades, existem 2^n combinações possíveis de subconjuntos e pos pode variar de 0 a n-1. Assim, a tabela `dp[mask][pos]` tem um total de $O(n \times 2^n)$ estados. Para cada estado, consideramos as n cidades e, assim, a complexidade temporal é $O(n^2 \times 2^n)$. Por fim, na estratégia **gulosa**, escolhemos a cidade inicial em O(1). Depois, precisamos percorrer todas as cidades não visitadas para encontrar a mais próxima. No pior caso, temos que verificar (n-1), (n-2), ..., 1 cidades ao longo do processo e assim, a estratégia é $O(n)$.
2. **Espaço:** Para representar o grafo, todas as três estratégias possuem um gasto de $O(n^2)$, considerando n vértices. **força-bruta** e **estratégia gulosa** apresentam uma lista auxiliar de cidades visitadas, ocupando um espaço adicional de $O(n)$. **Held-Karp** tem uma tabela de memoização que pode ocupar $O(n \times 2^n)$ (`pos` pode variar de 0 a n-1 e `mask` pode ter 2^n possíveis estados).

Teste	Solução Ótima	Guloso	Diferença
1	283	283	0
2	684	684	0
3	510	721	211
4	611	611	0
5	569	582	13
6	595	711	116
7	636	688	52
8	682	1085	403

Tabela 1: Análise comparativa entre os algoritmos

4.1 Qualidade da solução gulosa

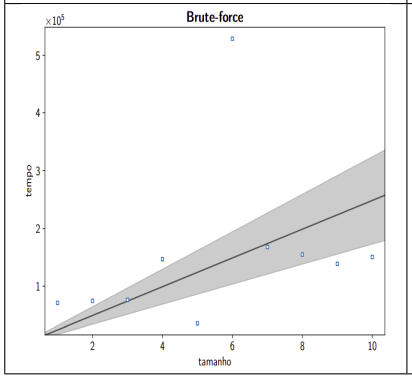
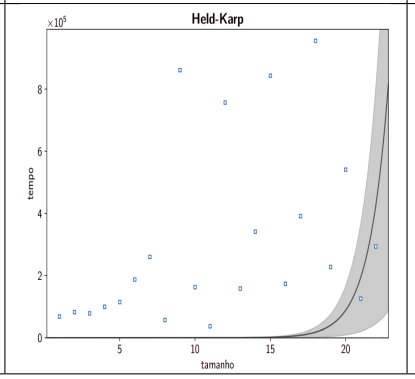
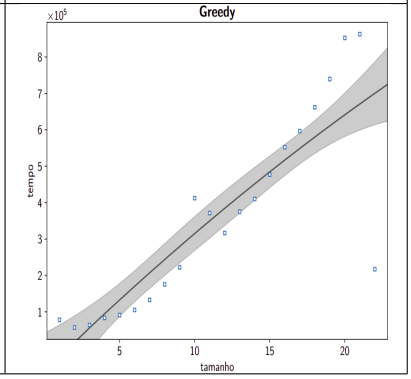
Com o objetivo de comparar as qualidades das soluções, os algoritmos foram testados com 8 testes e os resultados da análise podem ser observados na tabela 1. Em que as colunas contém o custo encontrado pelos algoritmos e a diferença de custo encontrada por eles.

Como pode ser observado, a estratégia gulosa, de fato, não é ótima, mas apresenta vantagem em situações nas quais a precisão não é um fator crítico.

4.2 Análise Experimental

As estratégias foram testadas com grafos que variavam de 2 a 23 vértices (no caso de Held-Karp e estratégia gulosa) e de 2 a 10 vértices para a força bruta. Os resultados das soluções em função do tempo podem ser visualizados na tabela a seguir:

Tabela 2: Complexidade de tempo

Força-bruta	Held-Karp	Guloso
		

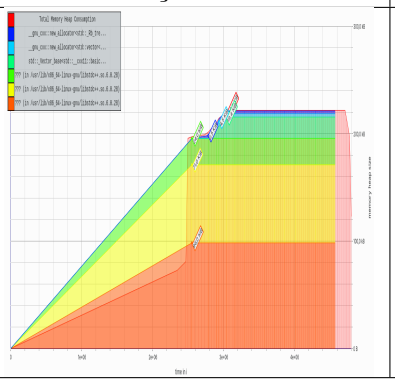
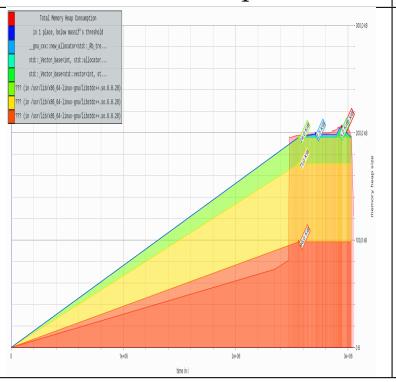
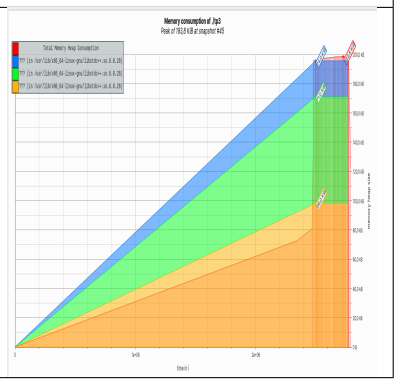
Por meio da tabela 2, é possível observar que, de fato, a estratégia gulosa é a mais rápida em termos de tempo de execução, alcançando o menor tempo em média das soluções. A estratégia de programação dinâmica também teve um bom desempenho, especialmente para grafos com menos de 20 vértices. Já a estratégia de força-bruta foi a que teve o pior desempenho, alcançando mais de $5 * 10^5$ ms tratando um grafo com pouco mais de 6 vértices.

Os algoritmos também foram testados com relação ao seu uso de memória, por meio de uma ferramente chamada `massif`. Os resultados podem ser visualizados na tabela 3.

Observando os gráficos, a abordagem gulosa apresenta um consumo de memória mais linear e menor, o que indica que sua implementação utiliza estruturas mais eficientes em termos de alocação. Já a abordagem dinâmica tem um crescimento mais acentuado da memória, o que é esperado devido à necessidade de armazenar subproblemas intermediários para reutilização, resultando em maior consumo. Por fim, a abordagem força bruta exibe o maior consumo de memória, refletindo sua natureza exploratória de avaliar múltiplas possibilidades sem otimização na alocação de recursos.

Essas diferenças ressaltam o impacto da escolha do algoritmo na eficiência do uso de memória, sendo a abordagem dinâmica um equilíbrio entre eficiência e otimização, enquanto a força bruta é a mais custosa e a gulosa a mais leve, mas potencialmente menos precisa na solução final.

Tabela 3: Uso da memória

Força-bruta	Held-Karp	Guloso
		

5 Considerações Finais

Neste trabalho, foi realizada a implementação de soluções para o problema da empresa DeliveryExpress de três formas diferentes: força-bruta; programação dinâmica e por meio de uma estratégia gulosa. Logo, durente o desenvolvimento do projeto foi possível explorar as vantagens e desvantagens desses três paradigmas, contando com uma análise comparativa que ressaltou a importância de cada um para cenários específicos.

As estratégias gulosas e de força-bruta também se deram de uma forma mais imediata, visto que a seleção de vértices de menor distância iterativamente é característica da estratégia gulosa e a busca exaustiva é a essência da força bruta.

Por fim, a estratégia de programação dinâmica se configurou como o maior desafio. A visualização da solução por meio do algoritmo de Held-Karp não se deu de forma tão imediata e a implementação correta do bitmask juntamente com a tabela de memoização para armazenar os subproblemas foi um desafio a ser superado.

Referências

[1] Wikipédia. *Held–Karp algorithm*. Disponível em: `https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm`. Acesso em: 22 jan. 2025.

[2] GeeksforGeeks. *Bitmasking and Dynamic Programming — Travelling Salesman Problem*. Disponível em: `https://www.geeksforgeeks.org/bitmasking-dynamic-programming-set-2-tsp/`. Acesso em: 22 jan. 2025.