Relatório de Algoritmos para o Problema do Caixeiro Viajante (TSP Euclidiano)

Antônio Araújo de Brum*

Gustavo Cunha Kneip†

02/06/2025

*aadbrum@inf.ufpel.edu.br
†gckneip@inf.ufpel.edu.br

Sumário

1 Introdução

Este relatório tem como objetivo descrever a implementação e análise de desempenho de três algoritmos para solução do Problema do Caixeiro Viajante (TSP): um algoritmo exato de força bruta (permutações), um algoritmo exato utilizando programação dinâmica (Held-Karp) e um algoritmo aproximativo (Christofides). Os testes foram realizados sobre instâncias com matrizes de adjacência, cujos nomes dos arquivos incluem o custo ótimo esperado, na linguagem **C++**.

Arquivo	Número de nodos	Menor custo
tsp1_253.txt	11	253
tsp2_1248.txt	6	1248
tsp3_1194.txt	15	1194
tsp4_7013.txt	44	7013
tsp5_27603.txt	29	27603

Tabela 1: Características principais dos TSPs

Todas as instâncias trabalhadas respeitam a desigualdade triangular, com exceção do tsp1, ou seja, ir diretamente de um vértice A até um vértice C nunca é mais lento que ir de A a C passando por B. Além disso, com exceção do 'tsp4', todos os grafos são completos. Isso quer dizer que é possível ir de qualquer vértice a qualquer outro vértice diretamente.

Era esperado que não fosse possível obter os dados das maiores instâncias (44 e 29 vértices) por força bruta, afinal é um algoritmo extremamente ineficiente, mas que talvez o held-karp fosse capaz de entregar, ao menos o 'tsp5'. E, é claro, o algoritmo de Christofides não encontraria o caminho ótimo, mas uma boa aproximação.

O código fonte está implementado em **C++** e disponível neste repositório do github. Mais detalhes da implementação e de como foi contornado o problema de incompletude do 'tsp4' a seguir.

2 Implementação

As especificações da máquina utilizada para executar o programa são:

- Processador: AMD Ryzen™ 7 5700U with Radeon™ Graphics × 16
- Memória: 12,0 GiB
- Disco: 1,5 TB

Em geral, todos os códigos utilizam bibliotecas de estruturas de dados básicas (vetores, filas, pilhas, etc.) e `chrono` para medição do tempo de execução.

O algoritmo de Christofides utiliza uma biblioteca `blossom V` para encontrar os pares mínimos para vértices em uma das etapas de execução.

Tanto para o código aproximativo quanto para o código exato por programação dinâmica foi necessário alterar a matriz do 'tsp4' utilizando o algoritmo de Floyd-Warshall (Subseção 2.2), pois estes algoritmos levam em conta que é possível ir de qualquer vértice até qualquer outro, o que não é o caso desta instância.

2.1 Força Bruta (Permutações)

O algoritmo de força bruta avalia exaustivamente todas as permutações possíveis dos vértices, calculando o custo total de cada ciclo Hamiltoniano possível. Ele garante encontrar a solução ótima, mas seu custo computacional torna-o proibitivo para instâncias com mais de 15 vértices.

A implementação utiliza a função `std::next_permutation` para gerar todas as permutações possíveis dos vértices, mantendo o primeiro vértice fixo para evitar ciclos redundantes.

Definição do ciclo A cada permutação dos vértices, é calculado o custo do caminho que passa por todos os vértices na ordem especificada e retorna ao vértice inicial, formando um ciclo fechado.

Tratamento de arestas inexistentes Durante o cálculo de cada ciclo, verifica-se se há arestas ausentes (custo igual a zero, exceto na diagonal). Se alguma aresta for inexistente, a permutação é descartada e o caminho não é considerado válido.

Limite de tempo Como o número de permutações cresce de forma fatorial ($(n-1)!$), um limite de tempo de 10 horas foi imposto à execução. O algoritmo é interrompido quando esse tempo é ultrapassado, sendo impressos o número total de permutações possíveis e quantas foram de fato executadas até então.

Exemplo de saída Ao final da execução, o algoritmo exibe:

- O custo mínimo encontrado;
- O caminho correspondente (incluindo o retorno ao vértice inicial);
- O tempo de execução;
- O número total de permutações possíveis ($(n-1)!$);
- O número de permutações realmente avaliadas dentro do tempo limite;

Aspectos práticos Mesmo sendo uma abordagem simples de implementar, o algoritmo de força bruta é viável apenas para grafos muito pequenos ($n \lesssim 15$). Para o 'tsp4' e 'tsp5' o cálculo das permutações totais ultrapassa a representação de inteiros de 64 bits. Então o *número total de permutações possíveis*, apresentado na saída do código, não representa a realidade.

Crescimento assintótico

- **Complexidade temporal**: Para percorrer todos os caminhos possíveis, começando sempre pelo mesmo vértice, o algoritmo deve calcular $(n-1)!$ caminhos. Levando um tempo fixo para cada permutação o algoritmo leva $(n-1)!$ unidades de tempo para a realização da tarefa. Portanto $O((n-1)!)$.
- **Complexidade espacial**: É preciso espaço para armazenar a matriz de adjacência e a permutação atual. $n^2 + n$, portanto $O(n^2)$.

2.2 Floyd-Warshall

Como foi mencionado anteriormente, o grafo em 'tsp4' não é um grafo completo e o 'tsp1' não respeita a desigualdade triangular (Existe caminho mais curto entre 0 e 3 passando por 4).

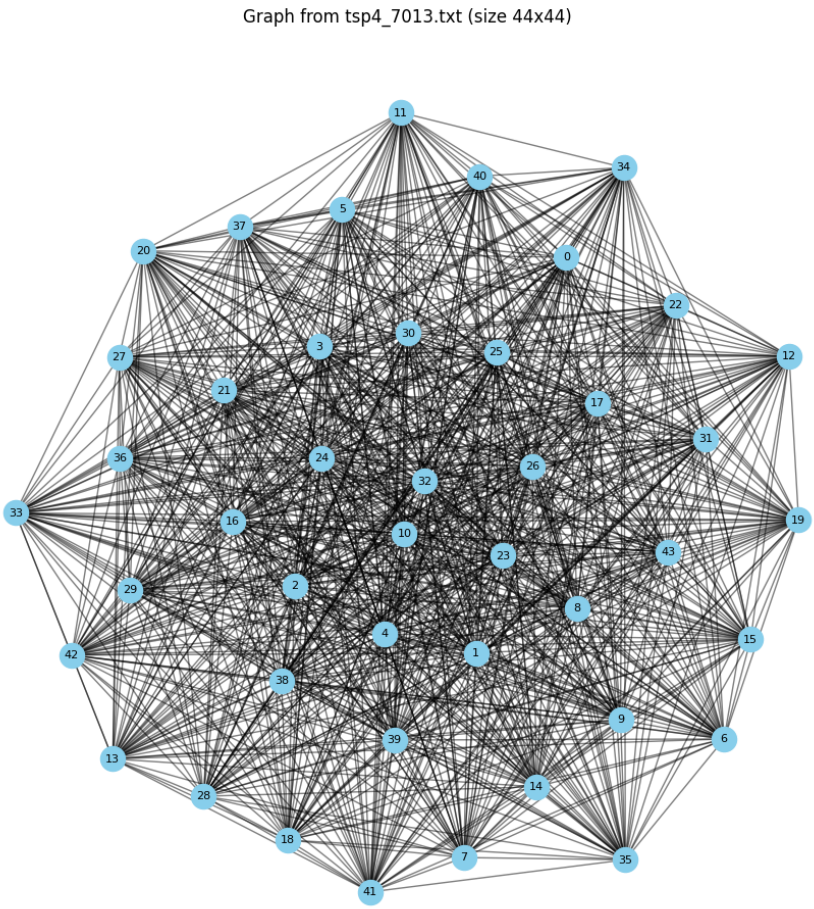

Figura 1: TSP4_7013.txt representado como um grafo utilizando matplotlib em python

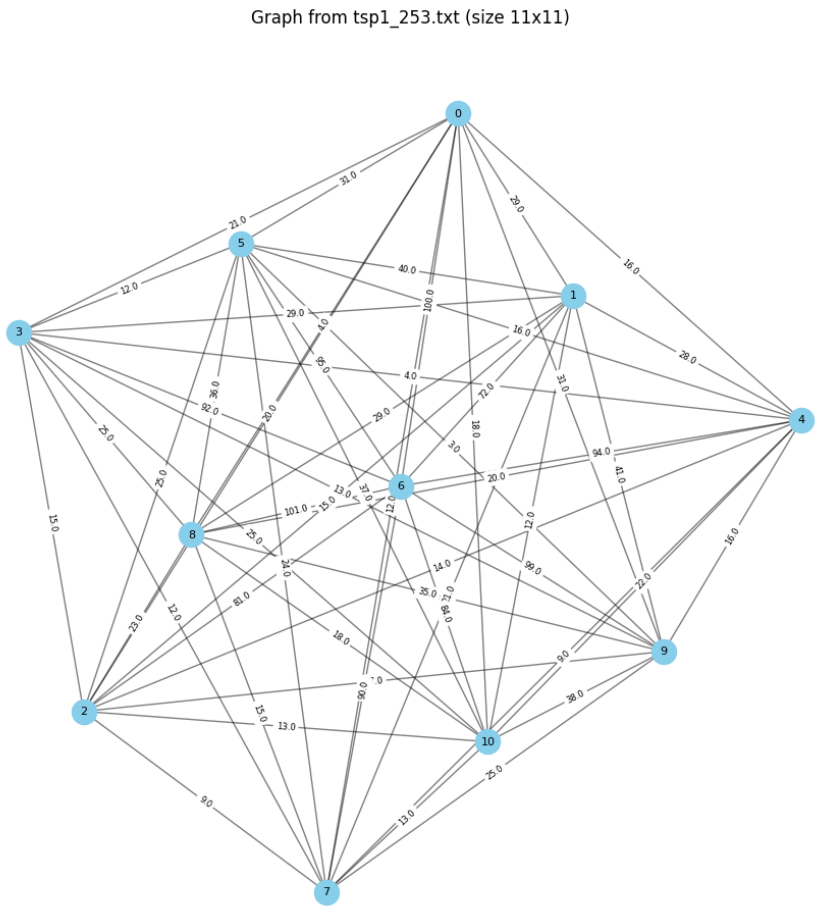

Figura 2: TSP1_253.txt representado como um grafo utilizando matplotlib em python

No 'tsp4' existem desconexões entre alguns nodos, o que atrapalha a confiabilidade do resultado nos algoritmos de Held-Karp e de Christofides. Ao aplicar estes programas à instância os resultados não foram o esperado para o algoritmo aproximativo, enquanto o exato não funcionou.

Isto acontece pois estes algoritmos pressupõe que é possível ir de qualquer vértice para qualquer outro vértice (nas explicações destes algoritmos, a seguir, isto ficará mais claro). E pode ser que seja necessário que o caminho envolva uma aresta que não existe.

Com relação ao 'tsp1', por ele não respeitar a desigualdade triangular, não é possível garantir a qualidade da resposta do algoritmo aproximativo.

$$d(0,3) = 21 > d(0,4) + d(4,3) = 16 + 4 = 20$$

Para contornar este problema, foi implementado, como passo intermediário, o algoritmo **Floyd-Warshall**. Este algoritmo utiliza 3 iterações aninhadas para encontrar a menor distância entre todos os vértices. Se a instância tratar-se de um grafo euclidiano, a menor distância de um nó a outro é o caminho direto, contudo se não há caminho direto (que é o caso de alguns nós no 'tsp4') ou existe um caminho mais curto passando por um outro nodo, este algoritmo cria o menor caminho utilizando algum intermediário e adiciona-o à matriz de adjacência nos índices das distâncias faltantes.

2.3 Held-Karp (Algoritmo Exato e dinâmico)

O algoritmo Held-Karp resolve o Problema do Caixeiro Viajante (TSP) de forma exata utilizando Programação Dinâmica com *bitmasking*, representando subconjuntos de vértices por inteiros. Ele possui complexidade temporal de $O(n^2 \cdot 2^n)$ e espacial de $O(n \cdot 2^n)$, o que o torna computacionalmente viável apenas para instâncias pequenas ou médias ($n \lessapprox 22$).

Completamento do grafo Antes da execução principal, o grafo original (possivelmente esparso) é completado usando o algoritmo de Floyd-Warshall (Subseção 2.2), garantindo conectividade total entre os vértices. Isso é necessário pois o algoritmo pressupõe que qualquer vértice pode ser alcançado a partir de qualquer outro. Um exemplo de uso é o tsp4 que, por não ser completo, era imcompatível com o algoritmo.

Definição de Estados A estrutura principal do algoritmo utiliza uma tabela `dp[mask][u]`, onde:

- `mask` representa um subconjunto de vértices visitados, codificado como inteiro.
- `u` representa o último vértice visitado no caminho parcial correspondente ao subconjunto `mask`.
- O valor armazenado é o menor custo para atingir o vértice `u` após visitar todos os vértices em `mask`.

Transição de Estados Para cada estado atual (`mask, u`), o algoritmo considera todos os vértices `v` ainda não visitados, e tenta estender o caminho:

$$\texttt{dp[mask | (1 << v)][v]} = \min(\texttt{dp[mask | (1 << v)][v]}, \texttt{dp[mask][u]} + \texttt{graph[u][v]})$$

Reconstrução do caminho Uma matriz auxiliar `parent[mask][u]` é usada para armazenar o vértice anterior em cada transição de estado. Com isso, é possível reconstruir o caminho ótimo ao final da execução da DP, de trás para frente.

Fechamento do ciclo Após visitar todos os vértices, o algoritmo verifica qual é o menor custo de retornar ao vértice inicial 0, fechando o ciclo Hamiltoniano:

$$\texttt{dp[FULL\_MASK - 1][i]} + \texttt{graph[i][0]}$$

Limitações Apesar de ser exato, o algoritmo tem limitações práticas:

- Diferentemente do algoritmo de força bruta, o Held-Karp apresenta um crescimento exponencial no uso de memória, o que geralmente limita sua aplicação a grafos com, no máximo, 20 a 22 vértices. Embora medidas como o uso de arquivos de swap possam permitir a execução em instâncias maiores, isso costuma resultar em uma queda significativa de desempenho.
- Para `n > 63`, mesmo com `uint64_t`, não é possível representar corretamente todos os subconjuntos, pois `1ULL << n` extrapola o tipo.

- O tempo de execução cresce rapidamente, tornando o algoritmo teoricamente inviável para instâncias maiores. No entanto, na prática, o consumo de memória é frequentemente o principal impeditivo: para valores altos de nn, o algoritmo pode falhar já na etapa de alocação, impedindo a execução antes mesmo que o tempo de processamento se torne um fator.
- Como armazenamos o caminho em uma matriz `parent[mask][u]` o custo de memória é duplicado, ou seja, passa de $O(n \cdot 2^n)$ para $O(2 \cdot n \cdot 2^n)$ No entanto, essa duplicação não impacta significativamente a complexidade geral, já que o termo dominante continua sendo $O(n \cdot 2^n)$;
- O algoritmo supõem que o grafo é completo e simétrico, necessitando de algoritmos como Floyd-Warshall caso não atenda essas especificações;

Resultados Ao final da execução, são exibidos o custo mínimo do ciclo, o caminho correspondente, o tempo total de execução e o consumo de memória estimado.

2.4 Christofides (Algoritmo Aproximativo)

O algoritmo de Christofides fornece uma solução com fator de aproximação garantido de 1.5-aproximado em grafos. Este resultado é obtido a partir da união de diferentes algoritmos. O passo a passo é:

Árvore Geradora Mínima (MST) Utiliza algoritmo de Prim para encontrar a árvore geradora mínima T.

Combinação perfeita mínima Retira do grafo original os vértices que em T possuem grau par, deixando apenas os vértices que em T possuem grau ímpar, formando um subgrafo M. Utiliza o **algoritmo de Blossom** para encontrar o conjunto de arestas que conecte os pares destes vértices de M com o menor custo possível.

Multigrafo Junta as arestas de T com as de M formando um multigrafo G.

Passeio Euleriano e Ciclo Hamiltoniano Encontra um passeio Euleriano E em G utilizando o **Algoritmo de Hierholzer** e então encontra o ciclo Hamiltoniano pulando os nós repetidos em E.

Saída A saída retorna o passeio aproximado e o seu custo, juntamente com o tempo de execução.

Crescimento assintótico

- **Complexidade temporal**: Mesmo que não seja necessário utilizar **Floyd-Warshall**, o **algoritmo de blossom** possui complexidade temporal contida em $O(n^3)$. Todos os outros algoritmos usados possuem crescimento na ordem de $O(n^3)$ ou menor.
- **Complexidade espacial**: O que mais consome espaço de armazenamento é a matriz de adjacência, portanto o crescimento de complexidade espacial está contido em $O(n^2)$.

3 Resultados

3.1 Força bruta

Instância	Custo Obtido	Tempo (s)	Dif. p/ Ótimo
tsp1_253.txt	253	12.9678	0
tsp2_1248.txt	1.248	0.000706583	0
tsp3_1194.txt	1287	7200	1,078
tsp4_7013.txt	22.304	36000	3,18
tsp5_27603.txt	42.065	36000	1,52

Tabela 2: Resultados do força bruta para as diferentes instâncias do TSP

Caminhos

- tsp1_253.txt: $0 \to 8 \to 10 \to 1 \to 6 \to 2 \to 5 \to 9 \to 3 \to 4 \to 7 \to 0$
- tsp2_1258.txt: $0 \to 5 \to 4 \to 3 \to 2 \to 1 \to 0$
- tsp3_1194.txt: $0 \to 2 \to 3 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12 \to 1 \to 4 \to 13 \to 14 \to 0$
- tsp4_7013.txt: $0 \to 37 \to 43 \to 38 \to 39 \to 36 \to 35 \to 34 \to 33 \to 42 \to 41 \to 40 \to 31 \to 32 \to 30 \to 29 \to 28 \to 27 \to 26 \to 25 \to 24 \to 23 \to 22 \to 21 \to 20 \to 19 \to 18 \to 17 \to 16 \to 15 \to 14 \to 13 \to 12 \to 11 \to 10 \to 9 \to 8 \to 7 \to 6 \to 5 \to 4 \to 3 \to 2 \to 1 \to 0$
- tsp5_27603.txt: $0 \to 18 \to 17 \to 16 \to 20 \to 21 \to 22 \to 28 \to 27 \to 25 \to 19 \to 24 \to 26 \to 23 \to 15 \to 14 \to 13 \to 12 \to 11 \to 10 \to 9 \to 8 \to 7 \to 6 \to 5 \to 4 \to 3 \to 2 \to 1 \to 0$

Conclusão Os resultados dos TSP's 'tsp4' e 'tsp5' foram obtidos após 10 horas de execução. São caminhos sub-ótimos. Levando em conta o número de permutações executadas e o número total de permutações existentes, é possível realizar uma estimativa do tempo de execução necessário para obtermos o menor caminho na máquina utilizada:

- Para o 'tsp5', foram executadas 5.805.537.839 permutações de um total de $3,048883446 \times 10^{29}$, logo seriam necessárias $5,251681292 \times 10^{20}$ horas ou $5,995069968 \times 10^{16}$ anos. Isto é o equivalente a $4.611.592x$ a idade atual do universo.

- Para o 'tsp4', foram executadas 5.995.501.228 permutações de um total de $6,041526306 \times 10^{52}$, logo seriam necessárias $1,007676602 \times 10^{44}$ horas ou $1,150315756 \times 10^{40}$ anos. O que é um número de anos tão absurdo que nenhuma comparação é capaz de dar a dimensão correta desta grandeza.

O resultado do TSP 'tsp3' foi obtido após 2 horas de execução. A estimativa para o final da execução:

- foram executadas 1.856.814.026 permutações de um total de 87.178.291.200, logo seriam necessárias $93,9$ horas para o fim da execução.

3.2 Held-Karp

Instância	Custo Obtido	Tempo (s)
tsp1_253.txt	253	0.00282435
tsp2_1248.txt	1248	6.9771e-05
tsp3_1194.txt	1194	0.0537157
tsp4_7013.txt	–	–
tsp5_27603.txt	–	–

Tabela 3: Resultados do Held-Karp para as diferentes instâncias do TSP

- tsp1_253.txt: $0 \to 8 \to 10 \to 1 \to 1 \to 6 \to 9 \to 5 \to 3 \to 4 \to 8 \to 0$
- tsp2_1248.txt: $0 \to 1 \to 5 \to 4 \to 3 \to 2 \to 0$
- tsp3_1194.txt: $0 \to 1 \to 9 \to 10 \to 11 \to 12 \to 13 \to 14 \to 8 \to 6 \to 5 \to 7 \to 4 \to 3 \to 2 \to 0$
- tsp4_7013.txt: –
- tsp5_27603.txt: –

Conclusão O algoritmo de Held-Karp se mostrou eficiente para grafos pequenos e médios, obtendo o resultado exato de forma rápida para os testes tsp1 (11 nós), tsp2 (6 nós) e tsp3 (15 nós). No entanto, para grafos grandes ($n \gtrsim 22$) seu alto consumo de memória o torna inviável, levando à falha na alocação de memória (bad_alloc) antes mesmo da execução ser iniciada. Para que fosse possível executar o algoritmo de Held-Karp para o tsp5 (29 nós), seria necessário um sistema com uma grande quantidade de memória RAM disponível.

No caso específico de $n = 29$, teríamos:

$$n \times 2^n = 29 \times 229 = 29 \times 536870912 \approx 15569256448$$

entradas. Supondo que cada entrada da tabela (um número inteiro representando o custo mínimo) ocupe 4 bytes, o consumo total de memória seria aproximadamente:

$$15569256448 \times 4B \approx 62277025792B \approx 58GB.$$

Já no caso do tsp4, que possui $n = 44$ nós:

$$n \times 2^n = 44 \times 2^{44} = 44 \times 17592186044416 \approx 774056185953984$$

entradas, o que resulta em:

$$774056185953984 \times 4B \approx 3096224743815936B \approx 2812TB.$$

Levando em consideração que também armazenamos o caminho, esse valor dobra de tamanho. Logo, se torna inviável rodá-lo em grafos grandes sem uma outra lógica para lidar com o problema de memória (swap file);

3.3 Chirstofides

Instância	Custo Obtido	Tempo (s)	Dif. p/ Ótimo
tsp1_253.txt	259	0,0016	1,02
tsp2_1248.txt	1.272	0,000779	1,019
tsp3_1194.txt	1.411	0,001159	1,18
tsp4_7013.txt	13.512	0,01	1,92
tsp5_27603.txt	30.652	0,005	1,11

Tabela 4: Resultados do Christofides para as diferentes instâncias do TSP

Caminhos

- tsp1_253.txt: $0 \to 7 \to 2 \to 10 \to 1 \to 6 \to 9 \to 5 \to 3 \to 4 \to 8 \to 0$
- tsp2_1248.txt: $0 \to 1 \to 5 \to 4 \to 3 \to 2 \to 0$
- tsp3_1194.txt: $0 \to 1 \to 9 \to 10 \to 11 \to 12 \to 13 \to 14 \to 8 \to 6 \to 5 \to 7 \to 4 \to 3 \to 2 \to 0$
- tsp4_7013.txt: $0 \to 7 \to 22 \to 21 \to 16 \to 1 \to 2 \to 23 \to 24 \to 38 \to 43 \to 3 \to 39 \to 17 \to 25 \to 29 \to 15 \to 12 \to 11 \to 6 \to 5 \to 28 \to 27 \to 9 \to 8 \to 10 \to 32 \to 30 \to 18 \to 19 \to 31 \to 40 \to 20 \to 41 \to 42 \to 13 \to 14 \to 4 \to 26 \to 36 \to 34 \to 35 \to 33 \to 37 \to 0$
- tsp5_27603.txt: $0 \to 1 \to 5 \to 4 \to 3 \to 2 \to 6 \to 8 \to 7 \to 11 \to 12 \to 13 \to 16 \to 17 \to 21 \to 22 \to 20 \to 28 \to 27 \to 25 \to 19 \to 15 \to 24 \to 26 \to 23 \to 18 \to 14 \to 9 \to 10 \to 0$

Conclusão Em geral, o algoritmo entregou resultados bastante abaixo de 1,5-aproximado em um tempo muito baixo. Isto justifica o seu uso e utilidade conforme as instâncias do problema crescem.

O resultado do 'tsp4' (1,9x pior que o resultado ótimo esperado) sugere que o custo ótimo para esta instância não é o custo passado no nome do arquivo. Mesmo que **Christofides** tenha entregue o pior resultado (1,5x o caminho ótimo), o que é bastante difícil, o custo do caminho ótimo seria 9008. Ainda assim, a confirmação desta hipótese não pode ser feita em computadores casuais antes do fim do universo (Utilizando força bruta para a memória não ser um problema).

Observações

- Instâncias maiores (por exemplo, $n > 25$) causaram erro de alocação de memória com Held-Karp. - O algoritmo de força bruta foi interrompido por tempo em casos com $n >= 29$. - Christofides manteve tempos baixos mesmo em instâncias maiores.

4 Conclusão

Nesse documento, foram apresentadas diferentes soluções para o problema do caixeiro viajante (TSP), incluindo algoritmos lineares, dinâmicos e aproximativos. Foi possível observar a utilidade de cada um para os diferentes casos de uso (lineares para precisão em grafos pequenos e aproximativos para resultados sub-ótimos em grafos grandes) e o porquê é importante para o profissional saber avaliar o problema e escolher o algoritmo certo para a situação;

5 Executando o código

Após fazer o download do repositório do github, faça a extração, entre na pasta com os arquivos e então, em um terminal siga o seguinte passo-a-passo:

1. $make clean
2. $make
3. $./TSP_algorithms tsp/{nome_do_arquivo.txt} 0

Obs - O número ao final do comando indica qual algoritmo será usado para a resolução: 0 para força bruta, 1 para Held-Karp e 2 para Christofides.

Referências

[1] Gutin, G. and Punnen, A. *The Traveling Salesman Problem and Its Variations*. Springer, 2002.

[2] Held, M. and Karp, R. M. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 1962.

[3] Floyd, R. W. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962.

[4] Christofides, N. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Carnegie Mellon University, 1976.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., e Stein, C. *Algoritmos: Tradução da 2ª Edição Americana*. [s.l.]: Elsevier, 2002.

[6] Kolmogorov, V. *Blossom V: A New Implementation of a Minimum Cost Perfect Matching Algorithm*. Mathematical Programming Computation, v. 1, n. 1, p. 43–67, 2009.