Trabalho Prático III - Algoritmos I

Augusto Guerra de Lima
2022101086

Departamento de Ciência da Computação; Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais; Verão de 2025
augustoguerra@dcc.ufmg.br

1 Introdução

Uma empresa de transportes busca determinar a rota de menor custo, preferencialmente mínima, que percorra todas as cidades apenas uma vez e retorne ao ponto de origem. Esse problema pode ser interpretado como o famoso **Problema do Caixeiro Viajante** (TSP), um clássico da ciência da computação, conhecido por ser NP-Completo.

Neste trabalho, serão utilizadas três abordagens algorítmicas para resolver o problema,a saber: força bruta, programação dinâmica e uma estratégia gulosa.

Na segunda seção, será apresentada a modelagem matemática e computacional do problema; A terceira seção explora as soluções algorítmicas com as três abordagens mencionadas; Na quarta seção, será realizada uma análise de complexidade e uma comparação entre os algoritmos implementados; Por fim, as considerações finais e as referências bibliográficas encerrarão o texto.

2 Modelagem

Como esperado, o objeto matemático que irá representar as cidades e as rotas será um grafo, $G = (V, E)$, não direcionado. Em particular, este trabalho está restrito à classe dos grafos completos K_*.

As cidades serão representadas pelos vértices do grafo, $v \in V$, enquanto as rotas serão modeladas como arestas não direcionadas, $e = (v_i, v_j) \in E$, onde $0 \leq i, j < |V|$ e $i \neq j$. A cada rota será associada uma distância $d \in \mathbb{Z}^+$.

O algoritmo $A : K_* \to \mathbb{Z}^+ \times V^*$ deve receber, como instância de entrada, um grafo restrito à classe supracitada. Ele deverá devolver dois elementos: um inteiro correspondente ao custo mínimo da rota possível e uma sequência de vértices indicando o circuito associado ao custo mínimo.

A modelagem computacional do grafo utilizará uma estrutura de dados baseada em uma **matriz de adjacência**. Essa escolha é óbvia, pois, como lidaremos com um grafo completo, tal representação se mostra bastante natural e eficiente.

Os algoritmos implementados foram de **busca exaustiva**[LA], **Held-Karp**[HK] e **busca local**.

3 Solução

3.1 Força bruta

A estratégia de força bruta consiste em listar recursivamente todas as permutações possíveis dos elementos de V, computando o custo associado a cada uma delas. Fixando um vértice como origem, ainda restam $(|V| - 1)!$ permutações possíveis, essa busca exaustiva é computacionalmente inviável para grafos grandes.

A implementação do algoritmo de geração de permutações é baseada em [LA].

Algorithm 1 TSP Força bruta

```
1: procedure A(K_n)
2:     circuito ← ∅
3:     min ← ∞
4:     busca_exaustiva(K_n)
5:     circuito ← busca_exaustiva.circuito
6:     min ← busca_exaustiva.min
7:     return(min,circuito)
8: end procedure
```

3.2 Programação dinâmica

O algoritmo de programação dinâmica implementado utiliza uma estrutura de dados baseada em uma matriz de memoização com dimensões $|V| \times 2^{|V|}$. A função dessa estrutura de dados é armazenar os estados previamente computados durante as chamadas recursivas, evitando cálculos redundantes. A técnica de memoização é amplamente conhecida e essencial em algoritmos de programação dinâmica.

Juntamente com a estrutura de memoização, foi utilizado o conceito de *bitmasking* [AK]. Essa estratégia consiste em utilizar um inteiro do tipo `int` para codificar um número binário. No caso do TSP, esse número binário representa os vértices já visitados indexados em sua posição de significância na representação binária do inteiro: 0 indica que o vértice não foi visitado, e 1 indica que sim.

A razão para as dimensões da matriz de memoização é que, para cada vértice, são codificados os $2^{|V|}$ subconjuntos possíveis de V, representando todas as combinações de vértices já visitados em diferentes ordens de visitação. A função objetivo a ser minimizada é como em [HK].

Algorithm 2 TSP Programação dinâmica

```
memo ← ∅
circuit ← ∅
procedure A(K_n)
    if todos os vértices visitados then return(K_n[v][0])
    end if
    if estado já computado na memo then return(valor armazenado na memo)
    end if
    for ∀u ∈ V do min ← min(d_uv+ A(v, V − {v}))
      adiciona próximo vértice no circuito
    end for
    return(min,circuito)
end procedure
```

3.3 Algoritmo guloso

O algoritmo guloso utilizado é, na verdade, bastante ingênuo. Ele é caracterizado por uma busca local nas vizinhanças do vértice corrente. Como se trata de um grafo completo, é garantido que existe uma aresta conectando qualquer par de vértices no grafo. Dessa forma, qualquer permutação simples dos vértices é garantidamente um caminho Hamiltoniano.

Baseado nessa propriedade, a estratégia do algoritmo é realizar a melhor escolha local a cada passo: selecionar o vizinho não visitado com o menor custo associado à aresta. Esse procedimento é repetido até que todos os vértices tenham sido visitados.

Algorithm 3 TSP Guloso

```
procedure A(K_n)
    visita v_1
    circuito ← {v_1}
    min ← ∞
    for i → |V| − 1 do
        for vizinhos u de v do toma o u_k que minimiza
            if u_k ∉ circuito then
                circuito ← circuito+{u_k}
                min← min +d_uv
            end if
        end for
        i ← i + 1
    end for
    min← min+d_uv
    return(min,circuito)
end procedure
```

3.4 Comparação

A implementação por força bruta consiste em utilizar recursão para percorrer todas as permutações possíveis. Ela consome um espaço relativamente menor em comparação com a solução por programação dinâmica, que necessita de uma estrutura de memoização que requer muitos recursos computacionais. A complexidade da solução por programação dinâmica é sobrepolinomial, mas ela é mais eficiente que uma busca exaustiva. Por outro lado, a solução gulosa tem uma implementação mais simples, requerendo apenas as variáveis a serem retornadas pelo algoritmo e uma lista de vértices visitados.

4 Análise de complexidade

4.1 Análise assintótica

O algoritmo de **força bruta** busca exaustivamente $(|V|-1)!$ permutações dos vértices. Para cada permutação, o algoritmo percorre as arestas do circuito para calcular o custo associado. Assim, a complexidade de *tempo de execução* pertence ao conjunto $\mathbf{O}(|V|!)$.

As listas `vertices` e `min_circuit` apresentam uma complexidade de *espaço* em $\mathbf{O}(|V|)$.

O algoritmo baseado em busca exaustiva retorna a resposta ótima, uma vez que testa todas as possibilidades de circuito. No entanto, devido à sua complexidade, ele é impraticável para grafos de dimensão grande.

Para o algoritmo de otimização por **programação dinâmica**, um estado do TSP é definido pelo vértice corrente e pelo conjunto dos vértices já explorados. Existem $2^{|V|}$ subconjuntos possíveis de vértices, e para cada estado o algoritmo considera todos os vértices possíveis para avançar. Dessa forma, a complexidade de *tempo de execução* está em $\mathbf{O}(|V| \cdot 2^{|V|})$.

A matriz de memoização, `memo`, armazena todos os estados possíveis. Portanto, a complexidade de *espaço* também está em $\mathbf{O}(|V| \cdot 2^{|V|})$.

Embora o algoritmo forneça uma solução ótima, sua complexidade ainda o torna impraticável para grafos muito grandes. Além disso, o alto custo de armazenamento devido à matriz de memoização é um desafio significativo em termos de recursos computacionais. Ainda assim, a solução é ótima.

O algoritmo de **estratégia gulosa** abre mão da otimalidade garantida da resposta em troca de uma complexidade polinomial. A cada passo, o vértice mais próximo ainda não visitado é escolhido. Assim, ele constrói um caminho válido, mas não garante que a solução global seja ótima, embora seja localmente ótima em cada etapa.

O algoritmo itera por todas as arestas de um grafo completo, onde $|E| = \frac{|V|(|V|-1)}{2}$. Portanto, a complexidade de *tempo de execução* está em $\mathbf{O}(|V|^2)$.

O algoritmo utiliza a lista `circuit` para armazenar o circuito gerado, resultando em uma complexidade de *espaço* em $\mathbf{O}(|V|)$.

4.2 Análise comparativa

4.2.1 Tempo de execução e armazenamento

Para uma análise comparativa, foram utilizados os mesmos casos de teste nas três abordagens implementadas, com um tempo limite de 10 segundos para a execução.

O algoritmo por força bruta, como demonstrado no gráfico, cresce de forma sobre-exponencial e apresentou um desempenho razoável até um número de 10 vértices. O algoritmo de programação dinâmica, por sua vez, não excedeu o tempo limite até 21 vértices, mas revelou sua complexidade exponencial à medida que o número de vértices aumentava.

Por outro lado, o algoritmo guloso suportou todos os casos dentro do tempo estipulado, mantendo uma complexidade polinomial. No entanto, essa abordagem não garante que a resposta obtida seja a solução ótima.

Para a análise de armazenamento, foi utilizada a ferramenta *Valgrind*. O objetivo principal era detectar como a estrutura de memoização impacta no gasto de recursos computacionais, uma das grandes desvantagens do algoritmo de programação dinâmica. Observe na Tabela 1 que o algoritmo aloca uma quantidade de memória exponencial em relação ao número de vértices, justamente devido aos estados dos subconjuntos armazenados na matriz de memoização.

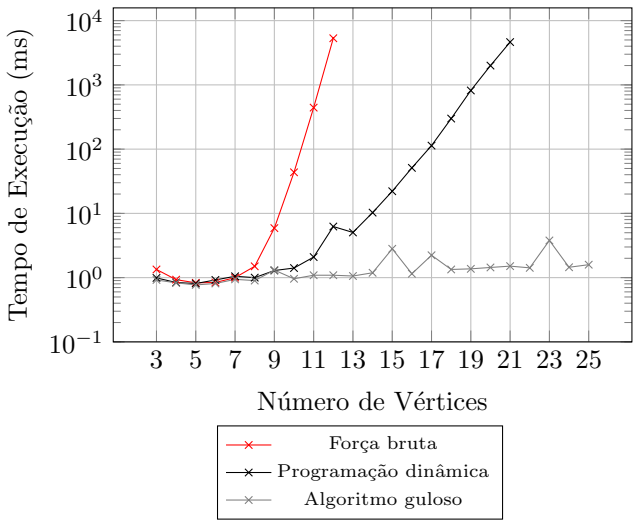

Figura 1: Comparação do tempo de execução dos algoritmos em escala logarítmica.

Vértices	Força bruta	Dinâmica	Guloso
3	1,34	0,99	0,92
4	0,93	0,84	0,83
5	0,83	0,81	0,77
6	0,84	0,92	0,81
7	1,00	1,05	0,94
8	5,90	1,00	1,31
9	43,59	1,29	0,96
10	5318,24	1,41	1,09
11	-	2,09	1,09
12	-	6,24	1,06
13	-	5,05	1,19
14	-	10,28	2,81
15	-	22,12	1,15
16	-	51,10	2,23
17	-	113,46	1,34
18	-	299,08	1,44
19	-	818,21	1,51
20	-	1996,41	1,42
21	-	4635,45	3,79
22	-	-	1,45

Figura 2: Tempo de execução dos algoritmos em milissegundos.

	5 vértices 10 arestas	9 vértices 36 arestas	23 vértices 253 arestas
Força bruta	76.788 bytes allocated	77.588 bytes allocated	-
Programação dinâmica	78.520 bytes allocated	118.888 bytes allocated	1.610.694 bytes allocated

Tabela 1: Gasto de alocação da estrutura de memoização, é notório como essa estrutura armazena exponencialmente no número de vértices, e tal análise deve ser avaliada no *trade-off*.

4.2.2 Qualidade da resposta obtida

Para a análise da qualidade da resposta do algoritmo guloso, foram utilizados os casos de teste disponibilizados. Avaliou-se se o algoritmo encontrava ou não a solução ótima e qual o aumento de custo em relação à resposta ótima devido à solução subótima encontrada.

Caso de teste	Resposta ótima	Guloso	Ótima	Custo adicional
`testCase01`	283	283	Sim	0
`testCase02`	684	755	Não	0,10
`testCase03`	510	650	Não	0,27
`testCase04`	611	645	Não	0,05
`testCase05`	569	644	Não	0,13
`testCase06`	595	595	Sim	0
`testCase07`	636	723	Não	0,13
`testCase08`	682	983	Não	0,44
`testCase09`	743	1179	Não	0,58
`testCase10`	806	1268	Não	0,57

Tabela 2: Qualidade das respostas obtidas pelo algoritmo guloso nos casos de teste disponibilizados.

Como observado, o algoritmo guloso apresenta uma complexidade de *tempo de execução* bem inferior aos outros algoritmos implementados. A melhor escolha local, em alguns casos, leva à melhor escolha global, ou seja, o algoritmo encontra a resposta ótima. No entanto, na maioria dos casos, ele converge para uma resposta subótima, com um aumento de aproximadamente vinte por cento no custo do percurso. Evidentemente, o algoritmo seria capaz de lidar com uma decisão válida em um grafo completo com muitos vértices, mas a resposta provavelmente não seria a melhor possível.

5 Considerações finais

Neste trabalho, foram implementadas estratégias para resolver o Problema do Caixeiro Viajante, amplamente conhecido. Foi possível comparar as diferentes implementações no que diz respeito à complexidade dos algoritmos, ao gasto de recursos computacionais e à qualidade das respostas. Problemas como este, que não possuem uma solução computacionalmente eficiente conhecida, podem, muitas vezes, ser aproximados com outras estratégias que, embora não garantam uma solução ótima, fornecem resultados suficientemente bons.

No caso deste trabalho, a implementação da solução gulosa foi bastante ingênua, mas retornou uma resposta com complexidade polinomial, o que, em muitas situações, pode ser essencial. Esse caso ilustra claramente o *trade-off* entre otimalidade e *tempo de execução*, que deve ser ponderado durante a análise e o projeto de algoritmos.

A parte mais desafiadora do trabalho foi entender o conceito de *bitmasking*, utilizado para codificar os estados da matriz de memoização. Este foi um assunto não abordado em sala de aula, assim como o próprio problema do Caixeiro Viajante, que foi apenas mencionado. Talvez este tema devesse constar na ementa.

Por outro lado, a parte mais simples foi a implementação do algoritmo guloso, já que a estratégia de busca local escolhida foi a mais trivial e ingênua possível. Além disso, foi aproveitada a propriedade do grafo ser completo, o que facilitou o desenvolvimento da solução.

Mais uma vez, pode-se utilizar, talvez, o objeto matemático e computacional mais comum da disciplina, o grafo. Em cima de sua estrutura e representação, foi possível desenvolver algoritmos que respondessem ao problema algorítmico proposto.

Discutir o problema do Caixeiro Viajante também significa refletir sobre a complexidade dos algoritmos e sobre o que podemos ou não resolver até o momento em ciência da computação. É um tema bastante rico.ee

6 Referências

[AK] ARORA, Kartik. *Introduction to Dynamic Programming with Bitmasking.* Codeforces, 14 nov. 2020.
Disponível em: https://codeforces.com/blog/entry/81516. Acesso em: 25 jan. 2025.

[HK] HELD, Michael. KARP, Richard M. *A Dynamic Programming approach to sequencing problems.* New York, IBM, 1961.

[HJ] HROMKOVIC, Juraj. *Theoretical Computer Science, Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography.* 1.ed. Aachen, Alemanha: Springer, 1998.

[LA] LAAKSONEN, Antti. *Guide to Competitive Programming, Learning and Improving Algorithms Through Contests.* 2.ed. Cham, Suíça: Springer, 2020.

[S] SURAJ. *Bitmask - For Beginners.* Codeforces, 30 jul. 2017.
Disponível em: https://codeforces.com/blog/entry/18169. Acesso em: 25 jan. 2025.