Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Algoritmos I

Trabalho Prático 3

Henrique Gomes dos Santos Medeiros
2021084986

Belo Horizonte
2024

Introdução

O problema computacional abordado neste trabalho é o Problema do Caixeiro Viajante (TSP - Travelling Salesman Problem), que consiste em encontrar o caminho mais curto possível que um caixeiro viajante pode percorrer para visitar todas as cidades de uma região exatamente uma vez e retornar à cidade de origem.

O código implementa três estratégias para resolver o TSP:

- **Força Bruta**: Gera todas as permutações possíveis de caminhos e seleciona o de menor custo.
- **Programação Dinâmica**: Utiliza memorização (ou memoização) e bitmasking para otimizar a busca pelo caminho mínimo.
- **Algoritmo Guloso**: Escolhe, a cada passo, a cidade mais próxima, na esperança de encontrar uma solução ótima global.

Modelagem

O problema foi modelado utilizando um grafo completo, onde cidades são representadas como vértices, estradas são representadas como arestas, com pesos correspondentes às distâncias entre as cidades, e a estrutura de dados principal é uma matriz de adjacência, que armazena as distâncias entre cada par de cidades.

As estratégias implementadas foram:

Força Bruta:

- Gera todas as permutações possíveis de caminhos.
- Calcula o custo total de cada caminho e seleciona o de menor custo.

Programação Dinâmica:

- Utiliza uma tabela de memorização (sub_results) para armazenar resultados parciais.
- Aplica recursão com bitmasking para explorar combinações de cidades visitadas.

Algoritmo Guloso:

- Escolhe, a cada passo, a cidade mais próxima da atual.
- Constrói o caminho iterativamente, na esperança de alcançar optimalidade global.

Solução

Força Bruta

1. Gera todas as permutações possíveis de cidades.
2. Para cada permutação, verifica se o caminho é válido (todas as cidades são conectadas).
3. Calcula o custo total do caminho.
4. Seleciona o caminho com o menor custo.

Pseudocódigo

```
starter_vector = [0, 1, 2, ..., n-1]
permutations = generatePermutations(starter_vector)
smallest_path = []
min_cost = INF

for path in permutations:
    if isValidPath(path, graph):
        cost = calculateCost(path, graph)
        if cost < min_cost:
            min_cost = cost
            smallest_path = path

print(smallest_path, min_cost)
```

Programação Dinâmica

1. Utiliza uma máscara de bits (mask) para representar cidades visitadas.
2. Armazena resultados parciais em uma tabela sub_results.
3. Explora recursivamente todas as combinações de cidades, evitando recomputações.

Pseudocódigo

```
function tsp(mask, pos, dist, sub_results, n):
    if mask == (1 << n) - 1:
        return dist[pos][0]
    if sub_results[mask][pos] != -1:
        return sub_results[mask][pos]

    min_cost = INF
    for city in 0 to n-1:
        if not (mask & (1 << city)):
            cost = dist[pos][city] + tsp(mask | (1 << city),
city, dist, sub_results, n)
            min_cost = min(min_cost, cost)

    sub_results[mask][pos] = min_cost
    return min_cost
```

Algoritmo Guloso

1. Inicia na primeira cidade.
2. A cada passo, escolhe a cidade mais próxima não visitada.
3. Repete até todas as cidades serem visitadas.
4. Retorna ao ponto de partida.

Pseudocódigo

```
current_city = 0
visited = {0}
path = [0]

while len(visited) < n:
    next_city = findNearestNeighbor(current_city, graph,
visited)
    path.append(next_city)
    visited.add(next_city)
    current_city = next_city

path.append(0)  // Retorna ao início
cost = calculateCost(path, graph)
print(path, cost)
```

Comparação

Aspecto	Força Bruta	Programação Dinâmica	Algoritmo Guloso
Complexidade Temporal	O(n!)	O(n² * 2^n)	O(n²)
Complexidade Espacial	O(n!)	O(n² * 2^n)	O(n)
Garantia de Otimalidade	sim	sim	não
Eficiência	Inviável para n > 10	Viável para n < 20	Viável para n grande
Facilidade de implementação	fácil	moderada/difícil	fácil

Análise de Complexidade

Força Bruta

- Tempo: O(n!) devido à geração de permutações de cidades.
- Espaço: O(n!) para armazenar todas as permutações.

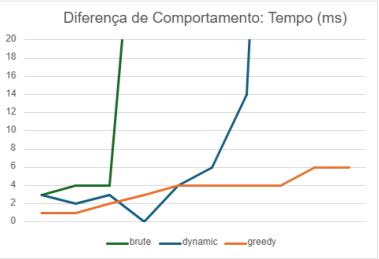
Programação Dinâmica

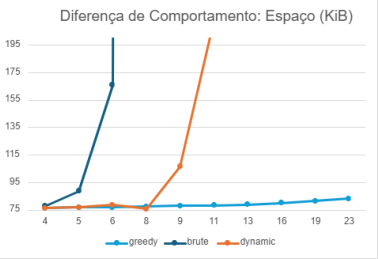
- Tempo: O(n² * 2^n) devido à combinação de bitmasking e recursão.
- Espaço: O(n² * 2^n) para armazenar a tabela sub_results.

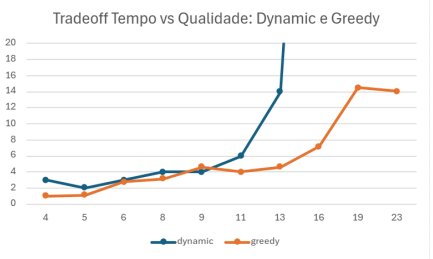
Algoritmo Guloso

- Tempo: O(n²) pois, para cada cidade, busca a cidade mais próxima.
- Espaço: O(n) para armazenar o caminho.

Gráficos e tabelas







O primeiro gráfico mostra a diferença de comportamento em tempo (ms) por tamanho da entrada dos três métodos.

O segundo gráfico mostra a diferença de comportamento em espaço (KiB) por tamanho da entrada dos três métodos.

O terceiro gráfico mostra o tradeoff tempo vs qualidade em tempo/qualidade por tamanho da entrada dos métodos dynamic e greedy.

Para mais detalhes sobre os gráficos e os dados utilizados para gerá-los, acesse: ALG TP3 analysis.xlsx

Conforme demonstrado através da comparação e dos gráficos, o método que utiliza força bruta não é viável a partir de n > 10 pois consome uma quantidade excessiva de memória. O método que utiliza programação dinâmica não é viável a partir de n > 20, aproximadamente, pois leva muito tempo para fornecer a solução, mesmo que esteja correta.

O método que utiliza um algoritmo guloso é viável em casos onde os outros dois métodos não podem ser utilizados, ou seja, quando n cresce demais. Apesar de apresentar taxa de erro variável para a distância fornecida, ainda apresenta uma performance adequada, como mostra o terceiro gráfico.

Considerações Finais

A implementação deste trabalho foi desafiadora, especialmente na parte de Programação Dinâmica, devido à complexidade do uso de bitmasking e recursão. A Força Bruta foi a mais simples de implementar, mas sua ineficiência a torna inviável para instâncias maiores. O Algoritmo Guloso foi o mais rápido, mas sua solução nem sempre é ótima. A parte mais difícil foi garantir a corretude da implementação da Programação Dinâmica, enquanto a mais fácil foi a implementação do Algoritmo Guloso.

A programação dinâmica em particular realmente exigiu muito de mim, me forçando a buscar soluções diferentes, estudar sobre o TSP, rever matérias de outras disciplinas e descobrir conceitos novos. Não somente isso, mas também explorei os conceitos principais da própria disciplina de Algoritmos I para completar esse trabalho, nomeadamente complexidade, DFS, recursão, backtracking, dentre outros.

Ao longo do processo eu pensei, criei e destruí dezenas de soluções diferentes, testei infinitas maneiras de resolver os problemas. Mesmo as coisas mais básicas como a decisão de utilizar uma lista ou matriz de adjacência fazem diferença em um trabalho assim, porque uma vez feito, não vale mais a pena voltar atrás. Como referência, só meu documento auxiliar para modelar as soluções continha cerca de 2000 palavras.

A garantia de receber apenas grafos completos pela entrada ajudou, apesar de eu ter visto tarde demais, após ter implementado as soluções mais complexas e resolver o

problema. Finalmente, foi um trabalho interessante e desafiador, através do qual pude visualizar claramente as diferenças em implementação, complexidade e desempenho de importantes métodos de problem solving.

Referências

https://www.baeldung.com/cs/tsp-exact-solutions-vs-heuristic-vs-approximation-algorithms#:~:text=Traveling Salesman Problem%3A Exact Solutions vs. Heuristic vs.,Applications of TSP ... 6 6. Conclusion

https://en.wikipedia.org/wiki/Travelling_salesman_problem

https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/

[http://www.gpec.ucdb.br/pistori/disciplinas/discreta/aulas/geulham.htm](http://www.gpec.ucdb.br/pistori/disciplinas/discreta/aulas/geulham.htm#:~:text=Um grafo G conexo é Hamiltoniano se existir,grafo acima não satisfaz a condição de Dirac.)

https://cplusplus.com/reference/climits/

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

https://en.wikipedia.org/wiki/Depth-first_search

https://www.geeksforgeeks.org/binary-tree-in-cpp/

https://cppscripts.com/cpp-tree

https://medium.com/the-programming-club-iit-indore/graphs-and-trees-using-c-stl-322e5779eef9

https://cplusplus.com/forum/general/30112/

https://www.geeksforgeeks.org/generic-treesn-array-trees/

https://www.geeksforgeeks.org/what-is-bitmasking/

https://www.geeksforgeeks.org/bitmasking-in-cpp/

https://www.youtube.com/watch?v=6sEFap7hIl4&list=PLb3g_Z8nEv1icFNrtZqByO1CrWVHLlO5g

Como fazer numeração de páginas conforme a ABNT?

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. Aulas e notas fornecidas pelo curso.