Nomes: Iara Campos - 202111255

Gabriel Camargos Alves - 201720471

Introdução

O Problema do Caixeiro Viajante (TSP) é um dos problemas mais emblemáticos da otimização combinatória, desafiando pesquisadores e profissionais devido à sua complexidade. Esse problema consiste em determinar o menor caminho possível para um caixeiro viajante, que deve visitar um conjunto de cidades, passando por todas uma única vez e retornando à cidade de origem. A dificuldade principal do TSP está em encontrar uma solução eficiente à medida que o número de cidades aumenta, uma vez que o espaço de soluções cresce exponencialmente com a quantidade de cidades.

Este relatório apresenta as melhorias implementadas em um algoritmo de resolução do TSP, utilizando uma abordagem híbrida que combina uma estratégia gulosa estocástica com técnicas de otimização local, como a busca 2-opt. Além disso, são realizados ajustes finos nos parâmetros do algoritmo, com o objetivo de melhorar sua eficiência e qualidade das soluções encontradas, especialmente em instâncias de maior dimensão. O foco dessas melhorias é reduzir o custo máximo do percurso, promovendo um balanceamento entre precisão e tempo de execução.

1. Abordagem Gulosa Estocástica

O algoritmo guloso original foi reformulado para incorporar um processo estocástico na seleção dos candidatos durante a fase de *lookahead*. Em vez de seguir a estratégia convencional de escolher sempre o próximo nó com a menor distância, a modificação permite que o próximo nó seja selecionado aleatoriamente entre os *k* melhores candidatos, ou seja, aqueles que apresentam as menores distâncias em relação ao nó atual, mas sem ser necessariamente o melhor entre todos. Essa alteração visa aumentar a diversidade das soluções iniciais geradas, permitindo uma exploração mais ampla do espaço de soluções e evitando que o algoritmo fique preso em soluções locais subótimas.

Essa estratégia é inspirada em métodos como o GRASP (Greedy Randomized Adaptive Search Procedure), que utiliza uma combinação de escolhas gulosa e aleatória para gerar soluções iniciais de maneira mais diversificada. O GRASP é eficaz em problemas de otimização combinatória, pois não depende de uma única sequência determinística de escolhas, mas sim de uma série de escolhas que equilibram exploração e exploração do espaço de soluções.

Com a introdução dessa aleatoriedade, o algoritmo tem uma maior chance de explorar diferentes trajetórias para encontrar soluções de qualidade superior. Isso é especialmente útil em problemas como o TSP, onde a busca por uma solução ótima pode ser limitada por uma exploração insuficiente do espaço de soluções. A modificação visa não apenas melhorar a solução final, mas também oferecer uma robustez adicional contra a queda em ótimos locais.

Essa abordagem tem um impacto positivo em instâncias de TSP mais complexas, onde as soluções puramente determinísticas poderiam ser limitadas. A aleatoriedade ajuda a evitar

que o algoritmo se prenda em um ciclo de escolha de cidades que leva a soluções subótimas, aumentando as chances de encontrar um caminho mais eficiente ao longo das iterações do algoritmo.

```python
def optimized_greedy(matrix, start=None, lookahead=50, k_best=5):
    """Algoritmo Guloso otimizado que escolhe aleatoriamente entre os melhores k candidatos"""
    n = matrix.shape[0]  # Número de nós
    start = random.randint(0, n-1) if start is None else start  # Se não especificado, escolhe aleatoriamente o nó de partida
    tour = [start]  # Lista para armazenar o tour
    candidates = set(range(n)) - {start}  # Conjunto de nós candidatos
    current_max = 0  # Distância máxima no tour atual

    while candidates:
        current = tour[-1]  # Último nó no tour
        neighbors = np.argsort(matrix[current])[:lookahead]  # Olha os vizinhos mais próximos (lookahead)
        valid_neighbors = [n for n in neighbors if n in candidates]  # Filtra os vizinhos válidos

        if not valid_neighbors:
            valid_neighbors = list(candidates)  # Se não houver vizinhos válidos, pega todos os candidatos

        # Ordena os vizinhos com base na distância máxima até o nó atual
        sorted_neighbors = sorted(valid_neighbors,
                                  key=lambda x: max(current_max, matrix[current, x]))
        top_candidates = sorted_neighbors[:k_best]  # Seleciona os top k candidatos
        best_next = random.choice(top_candidates) if top_candidates else valid_neighbors[0]  # Escolhe aleatoriamente o melhor

        tour.append(best_next)
        candidates.remove(best_next)  # Remove o nó escolhido dos candidatos
        current_max = max(current_max, matrix[current, best_next])  # Atualiza a distância máxima

    tour.append(start)  # Retorna ao nó inicial
    final_max = max(current_max, matrix[tour[-2], start])  # Distância máxima final
    return tour, final_max
```

O algoritmo optimized_greedy implementa uma versão do problema do Caixeiro Viajante (TSP) que utiliza uma abordagem gulosa estocástica, combinando seleção aleatória com uma busca por candidatos próximos.

1. **Inicialização:**
    - O algoritmo começa escolhendo um nó de partida, que pode ser especificado ou escolhido aleatoriamente.
    - A lista tour armazena a sequência de cidades visitadas, e o conjunto candidates contém todas as cidades que ainda precisam ser visitadas.
    - A variável current_max mantém o controle da distância máxima já percorrida no tour.
2. **Busca pelo próximo nó:**
    - A cada iteração, o algoritmo examina os nós vizinhos mais próximos ao nó atual (baseado na matriz de distâncias) até um número definido de lookahead vizinhos.
    - Os vizinhos válidos são aqueles que ainda não foram visitados (ainda estão em candidates).
    - O algoritmo ordena esses vizinhos pela distância máxima até o nó atual, para priorizar a escolha de cidades com maior "distância acumulada", evitando ciclos curtos.
    - Em seguida, escolhe aleatoriamente entre os k_best candidatos mais próximos (ou, se não houver, entre todos os vizinhos válidos).
3. **Atualização do tour:**
    - O nó escolhido é adicionado ao tour e removido do conjunto de candidatos.
    - A variável current_max é atualizada com a distância do novo nó, sempre mantendo o valor da maior distância já percorrida.
4. **Conclusão:**
    - O algoritmo retorna ao nó inicial para completar o tour.

- O valor final_max representa a maior distância do último trecho do caminho, completando a medida do "máximo" ao longo do tour.

2. **Busca Local Intensiva (2-opt)**

A técnica 2-opt é uma estratégia de otimização local utilizada para resolver o problema do Caixeiro Viajante. O conceito básico por trás dessa técnica é substituir dois segmentos do caminho por uma nova configuração, com o objetivo de reduzir a distância total percorrida. Isso é feito trocando a ordem de dois pontos no percurso, o que pode resultar em um caminho mais curto. Por exemplo, se você tem um caminho A-B-C-D, o algoritmo 2-opt tentaria trocar os segmentos (A-B) e (C-D) por (A-C) e (B-D), e assim por diante, procurando sempre uma melhoria na distância.

O aprimoramento feito no algoritmo envolve aumentar o número de tentativas de troca de segmentos de 100 para 500. Isso significa que o algoritmo agora pode realizar muito mais tentativas de troca e explorar um número maior de combinações possíveis, o que aumenta as chances de encontrar uma configuração mais eficiente. Ao não interromper a busca na primeira melhoria encontrada e continuar tentando, o algoritmo refina ainda mais o caminho, buscando uma solução ótima ou quase ótima. Isso é especialmente útil em problemas complexos, com um número grande de cidades, pois quanto maior o número de tentativas e passes de otimização, maior a probabilidade de o algoritmo encontrar um caminho muito mais curto.

Essa abordagem permite que o algoritmo evite soluções subótimas, aquelas que não são as melhores possíveis, mas que podem parecer boas em uma busca superficial. Ao realizar uma série de passes de otimização, o algoritmo se aprofunda no espaço de soluções, explorando configurações que, de outra forma, poderiam ser negligenciadas. O resultado é um caminho mais refinado e mais próximo da solução ótima, melhorando significativamente a qualidade da solução final.

```python
def fast_2opt(tour, matrix, max_attempts=500):
    """Refinamento do tour com a técnica 2-opt para melhorar a solução"""
    best_tour = tour.copy()  # Copia do tour atual
    best_max = max(matrix[tour[i], tour[i+1]] for i in range(len(tour)-1))  # Distância máxima do tour
    improved = True  # Flag para indicar se houve melhoria
    n = len(tour)

    while improved:
        improved = False  # Assume-se que não houve melhoria inicialmente
        attempts = 0

        while attempts < max_attempts:
            # Escolhe aleatoriamente dois índices para tentar reverter o caminho entre eles
            i, j = random.sample(range(1, n-1), 2)
            if i > j: i, j = j, i  # Garante que i < j

            old_max = max(
                matrix[best_tour[i-1], best_tour[i]],
                matrix[best_tour[j], best_tour[j+1]]
            )  # Distância máxima antes da troca
            new_max = max(
                matrix[best_tour[i-1], best_tour[j]],
                matrix[best_tour[i], best_tour[j+1]]
            )  # Distância máxima após a troca

            if new_max < old_max:  # Se a troca melhorar o tour
                best_tour[i:j+1] = best_tour[i:j+1][::-1]  # Reverte a parte do tour entre os índices i e j
                best_max = max(best_max, new_max)  # Atualiza a distância máxima
                improved = True  # Indica que houve melhoria
                break
            attempts += 1

    return best_tour, best_max
```

1. **Inicialização:**
    - best_tour = tour.copy(): Cria uma cópia do caminho atual (tour) para preservar a solução inicial e realizar as modificações nela.
    - best_max = max(matrix[tour[i], tour[i+1]] for i in range(len(tour)-1)): Calcula a distância máxima do caminho inicial (tour) considerando todas as arestas entre as cidades. Essa distância máxima é armazenada para monitorar as melhorias.
    - improved = True: Inicializa uma flag indicando que melhorias podem ser feitas.
    - n = len(tour): Obtém o número de cidades no caminho.
2. **Loop Principal de Otimização:**
    - O algoritmo entra em um loop enquanto a flag improved for True, ou seja, enquanto melhorias estiverem sendo feitas.
    - Dentro do loop principal, a flag improved é definida como False inicialmente, pois assume-se que não houve melhorias ainda.
3. **Tentativas de Melhoria:**
    - Dentro de outro loop (com número de tentativas controlado por max_attempts), o algoritmo escolhe aleatoriamente dois índices, i e j, do caminho (exceto os primeiros e últimos elementos) e tenta inverter a parte do caminho entre esses índices.
    - i, j = random.sample(range(1, n-1), 2): Escolhe aleatoriamente dois índices, que são garantidos de ser diferentes e dentro dos limites do caminho, para serem considerados para a troca.
    - O código assegura que i < j para garantir que a troca seja válida e não cause erros de índice.

4. **Cálculo da Distância Antes e Depois da Troca:**
    - Antes de realizar a troca, calcula-se a distância máxima do caminho entre as cidades i-1 a i e de j a j+1, que são os dois segmentos que serão invertidos.
    - Depois de inverter o segmento entre i e j, calcula-se a nova distância máxima entre i-1 a j e de i a j+1.
5. **Verificação de Melhoria:**
    - Se a nova distância máxima (após a troca) for menor do que a distância máxima anterior, então a troca é considerada uma melhoria.
    - best_tour[i:j+1] = best_tour[i:j+1][::-1]: Se a troca for vantajosa, o segmento entre os índices i e j é invertido (revertido) para melhorar o caminho.
    - best_max = max(best_max, new_max): Atualiza a distância máxima do caminho se necessário.
    - improved = True: Define a flag improved como True para continuar tentando mais melhorias.
6. **Limitação de Tentativas:**
    - O loop de tentativas continua até atingir o limite de tentativas (max_attempts) ou até que não sejam mais encontradas melhorias.
7. **Retorno:**
    - Depois de realizar todas as tentativas de melhoria, a função retorna o melhor caminho encontrado (best_tour) e a distância máxima associada a esse caminho (best_max).

Funções de pré processamento de dados:

1. vectorized_distance_matrix

Esta função calcula a matriz de distâncias entre os nós com base em dois tipos de peso possíveis: **EUC_2D** (distância euclidiana) ou **GEO** (distância geográfica usando a fórmula de Haversine). Ela utiliza operações vetorizadas para melhorar a performance na criação da matriz de distâncias entre os pontos.

2. data_transformation

A função data_transformation processa o conteúdo da instância de entrada (um arquivo de texto), extraindo as coordenadas dos nós e calculando a matriz de distâncias correspondente. Ela identifica o tipo de peso (EUC_2D ou GEO), extrai as coordenadas dos nós e chama a função vectorized_distance_matrix para calcular a matriz de distâncias.

3. solve_instance

Essa função resolve uma instância do TSP. Ela executa o algoritmo **Guloso Otimizado** para gerar uma solução inicial e, em seguida, aplica a técnica **2-opt** para refinar o caminho. O processo é repetido várias vezes para tentar melhorar a solução. O número de iterações e o parâmetro lookahead são ajustados dinamicamente com base no tamanho da instância. O melhor caminho encontrado e a distância máxima associada são retornados.

4. process_files

A função process_files é responsável por processar os arquivos de instâncias TSP em um diretório de entrada, resolvendo cada instância com a função solve_instance. Ela gera uma solução para cada

instância e grava os resultados em arquivos de saída. Também exibe a distância máxima encontrada, o tempo de execução e a economia em relação ao valor alvo (distância ideal).

Soluções

Execução 1:

Instância	Semente	Resultado Obtido	Resultado Esperado	Diferença em %	Tempo de execução (em segundos)
01	515	5710.7	3986	43.3	2.271
02	17	1323.7	1289	2.7	3.299
03	1203	1547	1476	4.8	3.05s
04	1539	1400.1	1133	23.6	3.594
05	363	678	546	24.2	4.027
06	1164	568.8	431	32.0	4.217
07	417	488.1	219	122.9	3.007
08	49	615.4	266	131.3	2.654
09	456	67.9	52	30.7	1.937
10	1379	647.6	237	173.2	3.075

Execução 2:

Instância	Semente	Resultado Obtido	Resultado Esperado	Diferença em %	Tempo de execução (em segundos)
01	466	5624.6	3986	41.1	2.951
02	523	1341.4	1289	4.1	3.328
03	535	1494.5	1476	1.3	3.432
04	329	1399.8	1133	23.6	3.799
05	816	653.4	546	19.7	4.532
06	107	569.4	431	32.1	4.016
07	521	401.8	219	83.5	3.212
08	272	529.7	266	99.1	3.526

09	324	77.4	52	48.8	2.232
10	136	711.6	237	200.3	3.160

Execução 3:

Instância	Semente	Resultado Obtido	Resultado Esperado	Diferença em %	Tempo de execução (em segundos)
01	239	4427.7	3986	11.1	2.251
02	960	1323.7	1289	2.7	3.204
03	472	1607.8	1476	8.9	3.022
04	1597	1371.7	1133	21.1	3.685
05	217	697.1	546	27.7	4.510
06	1257	585.5	431	35.9	3.636
07	742	579.7	219	164.7	2.905
08	926	484.8	266	82.3	2.878
09	98	72.4	52	39.3	1.866
10	818	757.8	237	219.7	3.068

Execução 4:

Instância	Semente	Resultado Obtido	Resultado Esperado	Diferença em %	Tempo de execução (em segundos)
01	248	5572.5	3986	39.8	2.338
02	485	1306.3	1289	-1.3	3.299
03	19	1594.7	1476	8	3.299
04	2013	142.8	1133	26.5	3.685
05	912	714.4	546	30.8	3.954
06	28	539.0	431	25.1	3.607
07	36	477.3	219	118.0	2.696
08	1023	492.1	266	85.0	3.073

09	316	72.4	52	58.9	1.832
10	35	704.4	237	197.2	2.964

Execução 5:

Instância	Semente	Resultado Obtido	Resultado Esperado	Diferença em %	Tempo de execução (em segundos)
01	56	5572.5	3986	39.8	3.694
02	532	1390.8	1289	7.9	3.438
03	1324	1548.9	1476	4.9	3.414
04	200	1413.9	1133	24.8	3.714
05	369	691.3	546	26.6	4.594
06	1360	507.9	431	17.8	3.863
07	1123	581.4	219	165.5	2.993
08	750	450.2	266	69.3	3.137
09	688	77.4	52	48.9	2.099
10	1660	575.9	237	143.0	3.104

Com os dados das cinco execuções, podemos agora realizar uma análise mais robusta do desempenho do algoritmo TSP, calculando médias e desvios padrão.
Análise dos Resultados:

Confirmação da variabilidade: Os desvios padrão, especialmente para a diferença percentual, confirmam a alta variabilidade dos resultados devido à influência da semente.

Desempenho por instância: A tabela de médias permite comparar o desempenho médio do algoritmo em diferentes instâncias. Por exemplo, a instância 7 continua apresentando a maior diferença percentual média (138.34%), indicando que é uma instância mais desafiadora para o algoritmo. Da mesma forma as instâncias 9 e 10 se destacam com maior dificuldade para o algoritmo.

Tempo de execução: A análise do tempo de execução médio e desvio padrão confirma a menor variabilidade em comparação com a qualidade da solução. Isso sugere que o tempo de execução é menos afetado pela semente.