Universidade Federal do Rio Grande do Norte – UFRN
Centro de Ensino Superior do Seridó – CERES
Departamento de Computação e Tecnologia – DCT
Bacharelado em Sistemas de Informação – BSI

Relatório sobre soluções heurística para o problema do Caixeiro Viajante

Alec Can Yalçin

Relatório Técnico apresentado ao Curso de Bacharelado em Sistemas de Informação como parte da avaliação da primeira unidade do componente Inteligência Artificial.

Laboratório de Elementos do Processamento da Informação – LabEPI
Caicó, RN, 11 de novembro de 2025

Resumo

Este trabalho apresenta um estudo comparativo entre algoritmos destinados a encontrar o melhor caminho possível presente no problema do Caixeiro Viajante. O método escolhido para descobrir a melhor combinação possível de cidades apresentados no problema foi através da distância euclidiana em um ponto, foram empregadas soluções exatas e aproximadas para o problema. Na abordagem exata, a criação de um algoritmo capaz de devolver todas as permutações possíveis de uma lista permitiu a análise da melhor combinação possível em uma série de pontos com relação à distância euclidiana, já nas abordagens aproximadas soluções heurísticas foram empregadas para chegar o mais próximo possível da solução exata, mas com menor tempo de execução, pois as permutações crescem em tempo fatorial.

Palavras-chave: Caixeirio Viajante; Heurísticas; Permutações;

Sumário

1. Introdução

Esse capítulo está organizado em seções como Motivação, Objetivos e Objeto de Estudo. Em Motivação é explicado a origem desse documento, em Objetivos é comentado sobre o que se busca deduzir ao fim desse estudo, por fim, em Objeto de Estudo é apresentado sobre todas as características do estudo. É possível encontrar todo o conteúdo desse relatório em: https://github.com/AlecYalcin/CaixeiroViajante.

1.1 Motivação

O estudo busca entender a dificuldade associada a construção de soluções aproximadas para problemas *Nondeterministic Polynomial time* dentro do estudo de algoritmos. Associando tanto a dificuldade da produção de solução de um problema ($NP - Completo$) quanto a facilidade da verificação dessa solução (NP).

1.2 Objetivos

Busca-se entender as diferentes implementações heurísticas para a resolução do problema do Caixeiro Viajante (sem solução exata encontrada) em comparação com o valor encontrado mediante um algoritmo de permutação.

1.3 Objeto de Estudo

Criação de algoritmos com saída aproximada em distância euclidiana em comparação com o melhor resultado possível do problema do Caixeiro Viajante.

2. Metodologia

Para seguir com os objetivos propostos e a motivação desejada, a metodologia desse trabalho foi dividida em duas partes diferentes. Primeiramente, é comentado sobre o ambiente do problema do caixeiro viajante e como ele foi configurado, após isso são apresentadas as métricas de comparação utilizadas no estudo e por fim é apresentado o método de comparação ideal entre o resultado ideal e os obtidos através dos algoritmos heurísticos produzidos.

2.1 Ambiente do Caixeiro Viajante

Os algoritmos utilizaram o padrão de Longitude e Latitude como referência para as distâncias x e y em um plano cartesiano representado por uma matriz de 360 por 180 – O que condiz com valores de 180 até -180 da Longitude e 90 até -90 da Latitude.

O objeto responsável por guardar essas informações era capaz de gerar aleatoriamente pontos em um plano como também importar tuplas de informação de longitude e latitude.

2.2 Métricas de Comparação

Todos os algoritmos utilizam a distância euclidiana, calculada através da conta matemática $d = \sqrt{x^2 + y^2}$, como parâmetro de identificação da menor distância acumulada. Dada uma lista de pontos, com longitude e latitude definidos, percorre-se a lista e compara o ponto atual com o próximo através da distância euclidiana, conforme a Figura 3.5.

Esse valor é somado em uma variável d que, ao final, adiciona a distância do último ponto com o primeiro, realizando a volta completada conforme o problema do Caixeiro Viajante.

```
def euclidean_distance(p1: Point, p2: Point) -> float:
    x_squared = (p2.latitude - p1.latitude) ** 2
    y_squared = (p2.longitude - p1.longitude) ** 2
    return math.sqrt((x_squared+y_squared))

def total_distance(points: list[Point]) -> float:
    d = 0
    for i in range(len(points)-1):
        d += euclidean_distance(points[i], points[i+1])
    return d + euclidean_distance(points[-1], points[0])
```

Figura 2.1: Função de cálculo de distância total

Para produzir valores comparativos das distâncias, foram criados *benchmarks* com valores de n (quantidade de pontos) com valores de 4 a 9 e 10 a 70. Esses *benchmarks* são arquivos que possuem tuplas de longitude e latitude salvas em listas que representam um grafo de cidades. Foram gerados aleatoriamente 10 exemplares para cada quantidade de n.

Os algoritmos foram então testados em todos os exemplares para cada *benchmark* de n, e realizada a média de distância encontrada nesses percursos. Esse valor é a principal métrica de avaliação apresentada no trabalho, seguida após ele pelo tempo médio de execução.

2.3 Resultado Ideal

Os valores de distância média foram comparados entre todos os algoritmos testados. A principal referência é o algoritmo de permutação, que para os valores de 4 a 9 encontrou a melhor combinação possível que gera a menor distância média conforme os parâmetros definidos anteriormente.

A limitação de 4 a 9 se deve pelo tempo de resolução do algoritmo de permutação, dado em tempo fatorial, apresentar longos períodos de resolução acima disso. No entanto, os *benchmarks* com valores de n variando de 10 a 70 foram testados em todos os algoritmos, exceto o de permutação, para melhor amostragem da progressão de distância e tempo neles.

Os resultados de 4 a 9 podem ter grande disparidade com os de 10 a 70, pois alguns algoritmos apresentam médias melhores com mais listas de pontos para testar. O caso do algoritmo genético é bem explícito.

3. Desenvolvimento

Nesse capítulo são apresentados os algoritmos criados para solucionar o problema do caixeiro viajante e o resultado da análise conforme as métricas definidas na metodologia. Na primeira parte são introduzidas as *benchmarks*, na segunda os algoritmos, a distância média e o tempo de execução para cada *benchmark*. E na terceira parte são mostrados gráficos comparativos entre os resultados.

3.1 Benchmarks

Os *benchmarks* estão no repositório. Os arquivos *.txt* contêm, em cada linha, listas com tuplas de latitude e longitude, conforme o exemplo apresentado na Figura 3.1. Há valores de *benchmark* para n de 4 até 9 e 10 até 70.

```
[(-33.5, 130.04), (0.17, -34.38), (-40.88, 27.53), (89.86, -76.23)]
[(17.31, 135.13), (-5.8, -71.69), (-27.72, 163.16), (-71.66, -96.6)]
[(-36.56, 9.15), (-19.59, 40.48), (1.89, -176.84), (19.9, 106.76)]
[(69.72, -67.08), (22.03, 164.61), (-42.5, 128.73), (-38.85, -18.77)]
[(4.46, -174.46), (16.93, 84.56), (-65.03, 163.5), (-43.7, 118.56)]
[(21.68, 107.95), (-9.29, 61.18), (58.99, 18.39), (66.89, -90.29)]
[(17.31, 10.56), (59.27, -41.94), (48.37, 98.28), (-61.31, -117.1)]
[(-77.69, 161.26), (-71.8, 151.85), (-3.92, 124.37), (80.02, 54.71)]
[(72.38, 138.01), (77.33, 2.77), (-15.59, -153.63), (46.56, -124.06)]
[(-78.28, -32.08), (85.24, 28.14), (0.52, -10.82), (89.8, 8.34)]
```

Figura 3.1: benchmark-n-4.txt

3.2 Algoritmos

Nessa seção, são abordados os algoritmos utilizados para resolver o problema do Caixeiro Viajante, são apresentados os códigos e resultados de distância. Os algoritmos presentes para testar foram: Permutação, Vizinhos mais Próximos e Interseção entre Grupos. Os dois últimos também tiveram uma testagem considerando todos os pontos de partida possíveis.

3.2.1 Algoritmo de Permutação

As permutações são todos os arranjos possíveis passado uma lista de elementos. No cenário do Caixeiro Viajante, ele é única maneira de encontrar a menor distância possível dada uma malha de pontos, mas seu custo em tempo e memória podem ser gigantescos. A Figura 3.2 mostra como esse algoritmo foi programado em python.

```
def _permutation(
    symbols: list[any],
    index: int = 0,
    memory: list = ...,
    results: list = ...,
) -> list[list[any]]:
    if memory == ...:
        memory = [-1] * len(symbols)
    if results == ...:
        results = []
    if index < len(memory):
        for i in range(len(symbols)):
            memory[index] = symbols[i]
            new_symbols = symbols[:i] + symbols[i+1:]
            results = _permutation(
                symbols=new_symbols,
                memory=memory,
                index=index + 1,
                results=results
            )
    else:
        results.append(memory.copy())
    return results

def traveler_permutations(points: list[Point]) -> list[Point]:
    permutations: list[list[Point]] = _permutation(points)
    best_distance = total_distance(permutations[0])
    best_permutation = permutations[0]
    for permutation in permutations[1:]:
        current_distance = total_distance(permutation)
        if current_distance < best_distance:
            best_permutation, best_distance = permutation, current_distance
    best_permutation.append(best_permutation[0])
    return best_permutation
```

Figura 3.2: Algoritmo de permutação com Caixeiro Viajante

3.2.2 Algoritmo de Vizinhos mais Próximos

A heurística dos vizinhos mais próximos foi desenvolvida com base em um conceito simples: seguir os pontos mais próximos uns dos outros até formar o caminho completo. Essa heurística, apesar de simples, demonstrou bons resultados. A Figura 3.3 mostra como ela foi implementada.

```
def nearest_neighbor_heuristic(
    points: list[Point],
    initial_index: int | None = None
) -> list[Point]:
    choosen_idx = initial_index or random.randint(0, len(points)-1)
    first_point = points.pop(choosen_idx)
    solution = [first_point]
    current_point = first_point
    while points:
        d_lowest, p_lowest = None, None
        for p in points:
            d_current = euclidean_distance(current_point, p)
            if not d_lowest or d_current < d_lowest:
                d_lowest, p_lowest = d_current, p
        current_point = p_lowest
        points.remove(p_lowest)
        solution.append(p_lowest)
    solution.append(first_point)
    return solution
```

Figura 3.3: Heurística "Vizinho mais Próximo"com Caixeiro Viajante

3.2.3 Algoritmo de Interseção entre Agrupamentos

A heurística da interseção entre grupos foi desenvolvida com inspiração no algoritmo de agrupamento não-supervisionado *k-means*. Ela agrupa todos os pontos da malha em um grupo e seleciona o *best starter*, o ponto mais longe do centroide desse grupo, como iniciador. O algoritmo então calcula todos os elementos restantes com a distância do ponto selecionado atual, para cada elemento em seu respectivo grupo, o compara com o centroide do grupo. Se esse centroide estiver mais longe do elemento do que ele em comparação com o ponto atual, esse elemento é realocado em um novo grupo. Esse feito é realizado até que não sobre mais nenhum elemento restante. A Figura 3.10 mostra como foi implementada essa lógica.

O algoritmo demonstrou-se eficiente em casos específicos, mas não obteve resultados melhores do que a heurística de vizinhos mais próximos na maioria das instâncias. Ele pode ser bom para elementos que estão muito separados em um conjunto de pontos, mas falha em encontrar um caminho melhor quando estão muito próximos.

3.2.4 Algoritmo Genético

A meta-heurística de algoritmos genéticos utiliza do princípio de criar uma população aleatória de exemplo e a partir dela extrair um "filho"que seja capaz de satisfazer o problema. No caso do caixeiro viajante, o objetivo é criar listas de pontos aleatórias (inicialmente) e cruzá-las a um nível que retorne filhos. São selecionados as listas com melhores distâncias totais calculadas, e sempre essa população é atualizada de acordo com esse parâmetro para a seleção da próxima base de pais para reprodução. A Figura 3.11 demonstra como foi implementada essa lógica.

3.2.5 Algoritmo de Vizinhos mais Próximos em todos os Pontos

A heurística de vizinhos mais próximos foi testada em todos os pontos e então escolhida conforme a menor distância resultante entre eles. Com essa seleção, o resultado ideal ficou mais próximo do que com a seleção aleatória.

```
def n_nearest_neighbor_heuristic(points: list[Point]) -> list[Point]:
    best = nearest_neighbor_heuristic(points.copy(), initial_index=0)
    best_distance = total_distance(best)
    for i in range(1, len(points)):
        solution = nearest_neighbor_heuristic(points.copy(), initial_index=i)
        distance = total_distance(solution)
        if distance < best_distance:
            best, best_distance = solution, distance
    return best
```

Figura 3.4: Heurística "Vizinhos mais Próximos"implementada em todo o conjunto de pontos

3.2.6 Algoritmos de Interseção de Agrupamentos em todos os Pontos

A heurística de interseção entre grupos foi testada em todos os pontos e então escolhida conforme a menor distância resultante entre eles. Com essa seleção, o resultado ideal ficou melhor do que selecionando o ponto mais distante, mas pior do que a heurística de vizinhos mais próximos quando testada nas mesmas proporções.

```
def n_group_intersection_heuristic(points: list[Point]) -> list[Point]:
    best = group_intersection_heuristic(points.copy(), initial_index=0)
    best_distance = total_distance(best)
    for i in range(1, len(points)):
        solution = group_intersection_heuristic(points.copy(), initial_index=i)
        distance = total_distance(solution)
        if distance < best_distance:
            best, best_distance = solution, distance
    return best
```

Figura 3.5: Heurística "Interseção entre Grupos"implementada em todo o conjunto de pontos

3.3 Comparações

Os resultados, quando comparados, mostram grande divergência nas métricas conforme de aumenta a quantidade de pontos a percorrer. No entanto, nenhum deles chega próximo o suficiente do valor ideal (encontrado nas permutações). Embora haja uma pequena variância nos valores menores (4 a 9), conforme o valor de n cresce, a disparidade aumenta.

3.3.1 Distância

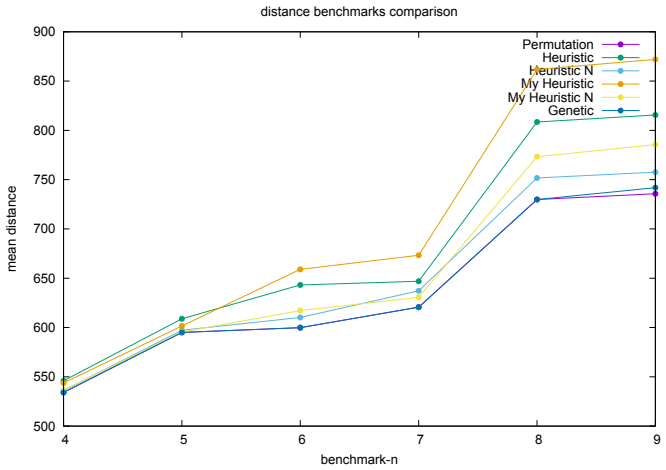

Figura 3.6: Média de distância nos benchmarks de 4 a 9

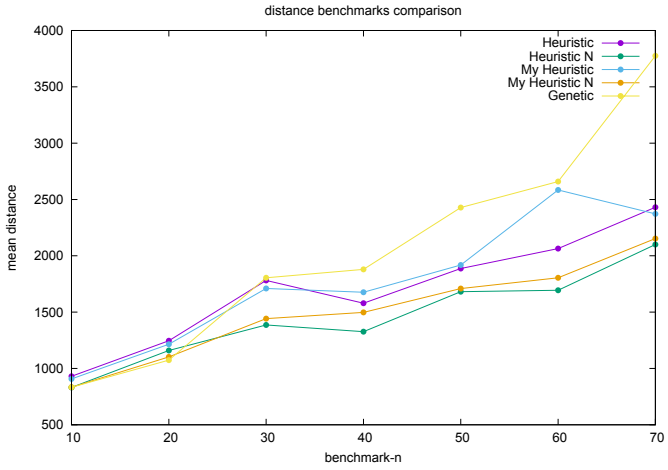

Figura 3.7: Média de distância nos benchmarks de 10 a 70

3.3.2 Tempo

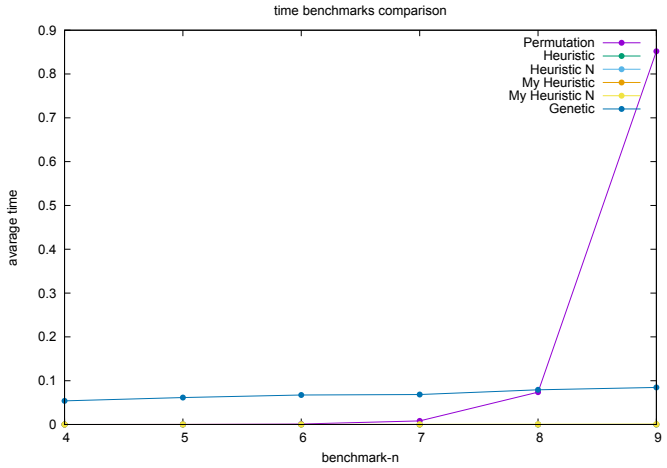

Figura 3.8: Média de tempo nos benchmarks de 4 a 9

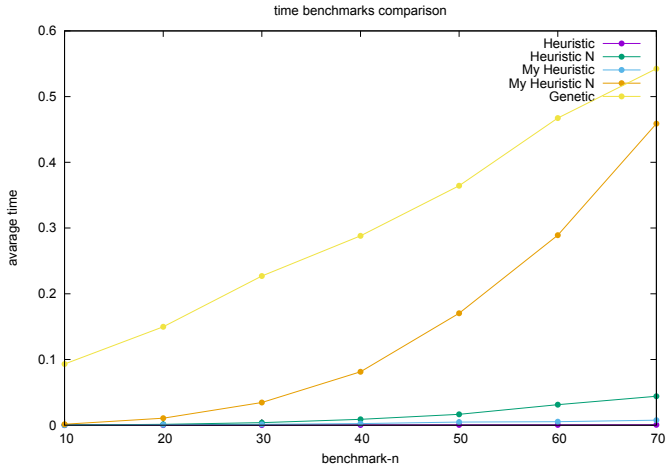

Figura 3.9: Média de tempo nos benchmarks de 10 a 70

```
def analyze_current_point(
    current_point: Point,
    groups: list[Group],
    threshold: float = 1.0
):
    for i in range(len(groups)):
        groups[i]._calculate_centroid()

    recently_groups = []
    for g in groups:
        new_group = Group(centroid=Point(0, 0, f"C{random.randint(0, 999)}"))
        current_d_from_centroid = euclidean_distance(current_point, g.centroid)
        for e in g.elements:
            current_d_from_element = euclidean_distance(current_point, e)
            if current_d_from_element < (current_d_from_centroid * threshold):
                new_group.elements.append(e)
                g.elements.remove(e)
        if not new_group.is_empty():
            recently_groups.append(new_group)
    groups.extend(recently_groups)

    closest_group, closest_distance = None, None
    for i in range(len(groups)):
        groups[i]._calculate_centroid()
        distance = euclidean_distance(current_point, groups[i].centroid)
        if closest_distance == None or distance < closest_distance:
            closest_group, closest_distance = i, distance

    return groups[closest_group].elements.pop(0)

def group_intersection_heuristic(
    points: list[Point],
    threshold: float = 1.0,
    initial_index: int | None = None
) -> list[Point]:
    choosen_idx = initial_index or _select_the_best_starter(points)
    first_point = points.pop(choosen_idx)
    solution = [first_point]

    first_group = Group(
        elements=points,
        centroid=Point(0, 0, f"C{random.randint(0, 999)}")
    )

    groups = [first_group]
    next_point = first_point
    while groups:
        next_point = analyze_current_point(next_point, groups, threshold)
        solution.append(next_point)
        groups_to_remove = []
        for i in range(len(groups)):
            if groups[i].is_empty():
                groups_to_remove.append(i)
        for i in groups_to_remove:
            groups.pop(i)
    solution.append(first_point)
    return solution
```

Figura 3.10: Heurística "Interseção entre Grupos"com Caixeiro Viajante

```
EPOCHS = 100
MAX_PARENTS = int(EPOCHS * 0.20)
MAX_CHILDREN = int(MAX_PARENTS * (MAX_PARENTS-1))
MUTATION_CHANCE = 0.25

def mutate(child: list[Point]) -> list[Point]:
    first_idx = random.randint(0, len(child)-1)
    second_idx = random.randint(0, len(child)-1)
    while second_index == first_index:
        second_index = random.randint(0, len(child)-1)
    child[first_idx], child[second_idx] = child[second_idx], child[first_idx]
    return child

def mate(
    first: list[Point], second: list[Point]
) -> list[Point]:
    size = len(first)
    child = [None]*size
    position = 0
    while first[position] not in child:
        child[position] = first[position]
        position = second.index(first[position])
    child = [
        child[i] if child[i] is not None else second[i]
        for i in range(len(second))
    ]
    if random.random() <= MUTATION_CHANCE:
        child = mutate(child)

    return child

def mating_season(
    parents: list[list[Point]], kids_len: int = MAX_CHILDREN
) -> list[list[Point]]:
    children = []
    for i, p1 in enumerate(parents):
        for j, p2 in enumerate(parents):
            if i != j:
                child = mate(p1, p2)
                children.append(child)
            if len(children) >= kids_len:
                return children[:kids_len]
    return children

def genetic_heuristic(points: list[Point]) -> list[Point]:
    population = [random.sample(points, len(points)) for _ in range(100)]
    population = sorted(population, key=lambda k: total_distance(k))
    epochs = EPOCHS
    max_parents = MAX_PARENTS
    while epochs:
        parents = population[:max_parents]
        children = mating_season(parents)
        population = parents + children
        population = sorted(population, key=lambda k: total_distance(k))
        epochs = epochs - 1
    return population[0]
```

Figura 3.11: Meta-heurística Genética com Caixeiro Viajante

4. Conclusões

Neste trabalho, foi analisado diferentes formas de solucionar o problema do Caixeiro Viajantes através de heurísticas. Como resultado dessa análise, notou-se que a criação de heurísticas eficientes pode ser um processo demorado, e nem sempre recompensador. Os problemas NP-Completos apresentam uma dificuldade associada em suas resoluções que foram demonstrados através dos algoritmos apresentados, mostrando que a resolução exata de certos problemas permanece um mistério e que as aproximações são a forma mais próximas de se chegar em algum resultado satisfatório.

Neste trabalho foi analisado as diferentes formas de busca em computação. Analisando de uma perspectiva de mapas com matrizes, foram implementados quatro algoritmos de busca visando entender a diferença de complexidade, tempo de execução e quantidade de passos necessários para chegar a um destino.