Algoritmos para o problema do Caixeiro Alugador

Arturo Fonseca de Souza e Társila Samille Santos da Silveira

Instituto Metrópole Digital - IMD / Universidade Federal do Rio Grande do Norte - UFRN

Palavras-chave: algoritmos, caixeiro viajante, heurística, teoria de grafos.

1. Introdução

O Problema do Caixeiro Viajante (PCV) é um dos mais famosos problemas encontrados no estudo da teoria de grafos e otimização combinatória. Ele possui inúmeras aplicações que variam de logística de transporte de materiais e pessoas até na otimização em processos industriais. Pode ser modelado como o problema de encontrar a menor rota passando por um conjunto de cidades, retornando para a cidade inicial.

Uma generalização desse problema seria o problema do Caixeiro Alugador (PCA). Neste, diferentes carros que alteram os custos das rotas podem ser alugados em cada cidade. Se um carro for devolvido em uma cidade diferente da que foi alugado, uma taxa extra deve ser paga [1].

As aplicações do PCA são voltados para indústria de aluguel de carros, onde se somam custos com combustível, pedágio e seguros. Com as opções de aluguel se diversificando cada vez mais com a expansão de empresas no setor no mundo, a busca pela minimização desses custos se faz relevante [2].

Como o PCV é NP-difícil [3] e o PCA possui o PCV como subcaso elementar [2], não há no momento algoritmos exatos que executam em tempo polinomial para a solução do PCA. Logo, dependemos de heurísticas e meta-heurísticas para encontrar uma solução aproximada viável.

Portanto, o objetivo deste artigo é fornecer e comparar implementações de um algoritmo exato e outros 3 baseados nas seguintes meta-heurísticas: *simulated annealing*, busca tabu e algoritmo genético. Essas implementações foram feitas a partir de trabalhos anteriores na literatura e construídas utilizando a linguagem de programação *Python*.

O restante do artigo se organiza da seguinte forma: na seção 2 é definido o problema do PCV. Na seção 3 é feita uma revisão de classes de algoritmos exatos e meta-heurísticas. Na seção 4 se

discute soluções para o problema do PCA presentes na literatura. A seção 5 descreve as três meta-heurísticas utilizadas nesse trabalho, enquanto a seção 6 discorre sobre a complexidade delas. Por fim, na seção 7 se discorre sobre os resultados dos experimentos feitos com os diferentes algoritmos.

2. Descrição do problema

Inicialmente será descrito o Problema do Caixeiro Viajante (PCV), para depois descrever sua generalização na forma do Problema do Caixeiro Alugador (PCA).

2.1. Problema do Caixeiro Viajante

O PCV consiste em, dado um grafo G completo ponderado (pesos não-negativos), encontrar o ciclo hamiltoniano com a menor soma de pesos possível em G, isto é, um ciclo que passe por todas os vértices do grafo.

2.2. Problema do Caixeiro Alugador

Dado um grafo $G = (V, E)$ com V sendo um conjunto de cidades (vértices) e E um conjunto de estradas que ligam duas cidades (arestas) e um conjunto de carros C, temos que d_{ij}^c é o custo de um carro c passar em uma estrada (i, j). Além disso, há o custo adicional f_{ij}^c toda vez que um carro é alugado em uma cidade i e retornado em uma cidade j, com $i \neq j$ [4]. O PCA consiste em encontrar o menor ciclo hamiltoniano considerando os custos de cada carro e aonde eles são retornados.

O PCA possui várias variações [4]. Neste trabalho utilizaremos a seguinte versão: todos carros podem ser alugados e devolvidos em qualquer cidade (*total e irrestrito*); o mesmo carro pode ser alugado mais de uma vez (*com repetição*); o valores de devolução são associados aos vértices em vez do caminho percorrido (*livre*) e o custo associado a uma aresta é independente do sentido (*simétrico*). Levaremos em conta também somente grafos completos.

Como dito anteriormente, o PCV é NP-difícil. E como o PCV é apenas o caso do PCA em que existe somente um carro, o PCV é tão difícil quanto o PCA pois possui todas suas exigências mas com muitas possibilidades a mais.

3. Algoritmos exatos e meta-heurísticas

3.1. Algoritmos exatos

Algoritmos exatos podem encontrar a solução ótima, mas são viáveis apenas para instâncias pequenas do problema. Exemplos de algoritmos exatos incluem:

- **Busca de Força Bruta:** Isso envolve tentar todas as permutações possíveis de caminhos e selecionar o mais curto. No entanto, esse método rapidamente se torna impraticável mesmo para instâncias pequenas do problema devido à sua complexidade fatorial.
- **Programação Dinâmica:** O algoritmo de Held-Karp [5] é um exemplo de abordagem de programação dinâmica para resolver o TSP. Ele possui um tempo de execução de $O(n^2 \cdot 2^n)$.
- **Algoritmo Exato Quântico:** Mais recentemente, Ambainis et al. propuseram um algoritmo quântico exato para o TSP que possui um tempo de execução de $O(1.728^n)$ [6]. Este algoritmo é atualmente o melhor algoritmo conhecido para resolver o TSP exatamente, mas ainda não é prático para resolver instâncias grandes do problema.
- **Algoritmos Branch-and-Bound e Branch-and-Cut:** Esses algoritmos podem lidar com TSPs com 40-60 cidades e instâncias maiores, respectivamente. O Solucionador TSP Concorde, um algoritmo de *branch-and-cut*, pode resolver um TSP com quase 86.000 cidades [7].

3.2. Meta-heurísticas

Algoritmos meta-heurísticos são estratégias ou heurísticas de alto nível projetadas para encontrar, gerar ou selecionar uma heurística que possa fornecer uma solução suficientemente boa para um problema de otimização [8]. Esses algoritmos podem ser classificados em várias categorias com base em seu comportamento ou fonte de inspiração [8].

3.3. Categorias de Algoritmos Meta-heurísticos

O *International Journal of Online and Biomedical Engineering* (iJOE) classificou os algoritmos meta-heurísticos em quatro grupos principais:

1. **Algoritmos baseados em evolução (EAs):** Esses algoritmos são inspirados na seleção natural e no comportamento evolutivo dos organismos. Exemplos incluem Programação Genética, Algoritmo Cultural e Evolução Diferencial.
2. **Algoritmos baseados em inteligência de enxame (SI):** Esses algoritmos modelam o comportamento cooperativo, adaptativo e gregário de bandos naturais ou comunidades. Exemplos incluem Otimização por Colônia de Formigas, Otimização por Enxame de Partículas e Colônia de Abelhas Artificiais.
3. **Algoritmos baseados em ciências naturais (NSAs):** Esses algoritmos imitam princípios químicos específicos ou processos físicos. Exemplos incluem Recozimento Simulado, Busca de Vizinhança Variável e Big Bang-Big Crunch.
4. **Algoritmos baseados em comportamento humano (HBAs):** Esses algoritmos são inspirados no comportamento humano, incluindo atividades não físicas como pensamento e percepções sociais. Exemplos incluem Algoritmo de Civilização da Sociedade, Algoritmo de Otimização de Buscador e Algoritmo Competitivo Imperialista.

Existem também outros algoritmos meta-heurísticos que não se enquadram nessas categorias, como algoritmos inspirados em esportes, algoritmos baseados em música, algoritmos baseados em plantas, algoritmos baseados em matemática e algoritmos baseados no comportamento da água [8].

Outra classificação é de Classificações generalizadas de técnicas de busca de meta-heurísticas [9] [10].

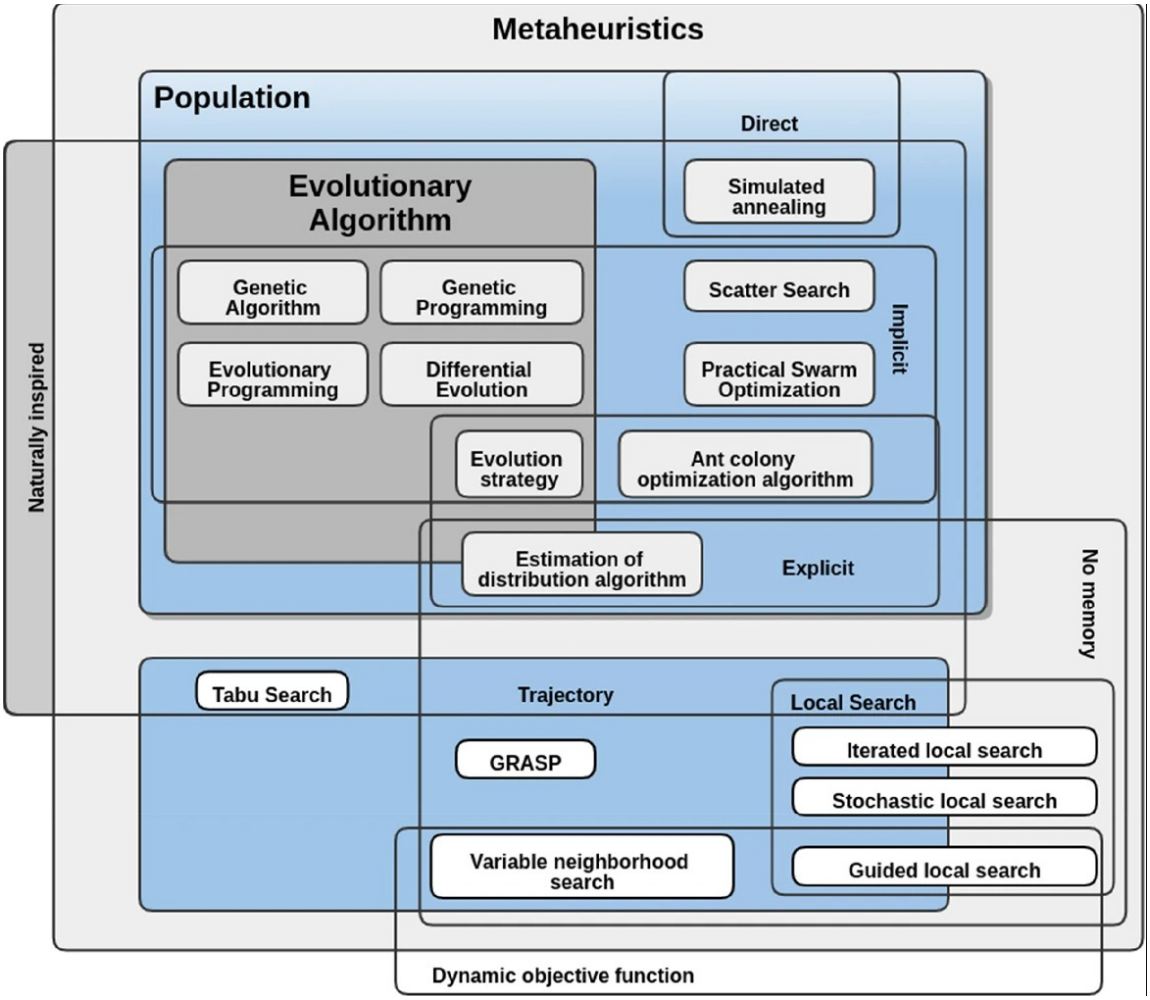

Figure 1: Classificações generalizadas de técnicas de busca de metaheurísticas (De Castro and Von Zuben 2000).

3.4. Meta-heurísticas Recentes

3.4.1. Algoritmo Baseado no Desempenho do Aprendiz (LPB)

O LPB é influenciado pelo processo de admissão à faculdade para graduados do ensino médio. Utiliza a probabilidade de divisão para separar uma parte da população em dois grupos: bom e ruim[8].

3.4.2. Otimizador Giant Trevally (GTO)

O GTO é inspirado nas estratégias do grande *trevally* ao caçar aves marinhas. Possui três etapas principais: busca extensiva, escolha de área e ataque[8].

3.4.3. Busca Especial da Relatividade (SRS)

Baseado no movimento de partículas em um campo eletromagnético. Calcula a distância entre as partículas e determina a carga com base na aptidão[8].

3.4.4. Evolução dos Conselhos da Cidade (CCE)

Modela o desenvolvimento de conselhos da cidade, usando a natureza competitiva da adesão. Utiliza uma versão modificada da recombinação aritmética para aumentar a produtividade[8].

4. Estado da arte

As primeiras formulações e soluções do Problema do Caixeiro Alugador podem ser encontradas no artigo de Goldbarg et al. (2011) [2], fornecendo duas soluções heurísticas baseadas em algoritmos evolucionários: um algoritmo Memético e outro Transgenético. Outra solução dentro dessa área foi desenvolvida por Felipe et al. (2014) [11] e Goldbarg et al. (2017) [12] na forma de algoritmos científicos, uma meta-heurística relativamente recente baseado na pesquisa científica [11].

Algoritmos híbridos baseados em programação linear para solucionar o PCA foram introduzidos por Rios (2018) [4].

Lacroix et al. (2021) também utilizou métodos de programação linear juntamente com um algoritmo de *branch and cut* e resolveu, de forma ótima, a maioria das instâncias utilizadas pelas publicações anteriores, também as resolvendo elas em tempos menores.

5. Algoritmos Meta-heurísticos

Foi testado 3 algoritmos: o Simulated Annealing, Busca Tabu e o Genético (este último com duas versões possuindo parâmetros diferentes).

5.1. Funções Auxiliares

5.1.1. generate_ random_ neighbor

A função chamada `generate_random_neighbor` que recebe dois parâmetros: `route` (rota) e `car_assignments` (atribuições de carros), é uma função auxiliar para alguns dos algoritmos feitos.

O propósito dessa função é gerar um vizinho aleatório para uma rota dada e atribuições de carros.

Primeiramente, a função cria uma cópia das listas `route` e `car_assignments` usando os métodos `copy()` e `[:]`, respectivamente. Isso é feito para evitar a modificação das listas originais.

Em seguida, a função determina o número de cidades na rota usando a função `len()` e atribui esse valor à variável `num_cities`.

Depois, a função gera um número aleatório de trocas a serem realizadas na rota. O número de trocas é determinado ao pegar o mínimo entre um número inteiro aleatório entre 1 e metade do número de cidades e a diferença entre o número de cidades e 2. Isso garante que o número de trocas esteja dentro de uma faixa válida.

A função então entra em um loop que itera `num_swaps` vezes. Em cada iteração, ela gera um intervalo de índices de 1 a `num_cities - 1` e atribui isso à variável `r`.

Em seguida, a função verifica se o índice 2 está dentro do intervalo `r`. Se estiver, ela seleciona duas cidades aleatórias do intervalo usando a função `random.sample()` e as atribui às variáveis `city1` e `city2`.

A função então troca as posições de `city1` e `city2` na lista `new_route` usando a atribuição de tupla.

Da mesma forma, ela troca as atribuições de carros para `city1` e `city2` na lista `new_car_assignments`.

Após a conclusão do loop, a função retorna as listas modificadas `new_route` e `new_car_assignments` como uma tupla.

```
Função generate_random_neighbor(route, car_assignments):
    new_route = route.copy()
    new_car_assignments = car_assignments[:]

    num_cities = len(route)
```

```
num_swaps = min(random_integer(1, num_cities // 2), num_cities - 2)

Para cada swap no intervalo de 1 até num_swaps:
    r = intervalo(1, num_cities - 1)

    Se 2 está dentro de r:
        city1, city2 = random_sample(r, 2)
        Trocar as posições de city1 e city2 em new_route
        Trocar as atribuições de carros para city1 e city2 em new_car_assignments

Retornar (new_route, new_car_assignments)
```

5.1.2. calcular_ distancia_ total_ geral

Esta função calcula a distância total para uma rota, atribuições de carros e carros dados.

A função `calcular_distancia_total_geral` recebe três parâmetros: `rota`, `atribuicoes_carros` e `carros`.

O parâmetro `rota` representa a sequência de locais a serem visitados. É uma lista de inteiros.

O parâmetro `atribuicoes_carros` representa a atribuição de carros para cada local na rota. É uma lista de inteiros.

O parâmetro `carros` representa os carros disponíveis para a rota. É uma lista de objetos de carro.

A função inicializa a variável `distancia_total` como 0.

A lista `rota` é modificada adicionando um 0 no início e no final. Isso é feito para representar o local de início e término da rota.

A variável `carro_atual` é inicializada com a atribuição de carro para o primeiro local na rota.

A variável `local_atual` é inicializada com 0, representando o local de início.

A função então itera sobre cada local na rota usando um loop `for`. A variável de loop `proximo_local` representa o próximo local a ser visitado.

Dentro do loop, há uma declaração `if-else` que verifica se a atribuição de carro atual é diferente da atribuição de carro para o próximo local.

Se as atribuições de carro são diferentes, a função adiciona a distância entre o local atual e o próximo local, bem como a taxa de retorno entre o local atual e o próximo local, à variável `distancia_total`.

Se as atribuições de carro são iguais, a função adiciona apenas a distância entre o local atual e o próximo local à variável `distancia_total`.

Após calcular a distância para o local atual, a variável `local_atual` é atualizada para o próximo local.

Finalmente, a função retorna a variável `distancia_total`, que representa a distância total para a rota, atribuições de carros e carros fornecidos.

```
Função calcular_distancia_total_geral(rota, atribuicoes_carros, carros):
    distancia_total = 0
    rota = AdicionarONoInicioEFim(rota)
    // Adiciona 0 no início e no final da rota
    carro_atual = atribuicoes_carros[rota[0]]
    // Atribuição de carro para o primeiro local
    local_atual = 0  // Início da rota

    Para cada proximo_local na rota:
        Se atribuicoes_carros[proximo_local] diferente de carro_atual:
            distancia_total += DistanciaEntre(local_atual, proximo_local) + TaxaDeRetorno(local_atua
        Senão:
            distancia_total += DistanciaEntre(local_atual, proximo_local)

        local_atual = proximo_local  // Atualiza o local atual
        carro_atual = atribuicoes_carros[local_atual]
        // Atualiza o carro atual

    Retornar distancia_total
```

5.2. Simulated Annealing

O pseudocódigo a seguir é do algoritmo de Simulated Annealing (Recozimento Simulado). O Simulated Annealing é um algoritmo meta-heurístico usado para resolver problemas de otimização. Ele é inspirado no processo de recozimento na metalurgia, onde um material é aquecido e depois resfriado lentamente para reduzir defeitos e melhorar sua estrutura.

```
Função accept_solution(current_cost, neighbor_cost, temperature):
    Se neighbor_cost < current_cost então:
        Retorne Verdadeiro  # Aceitar solução melhor

    Senão:
        probabilidade = exp(-(neighbor_cost - current_cost) / temperature)
        Se random() < probabilidade então:
            Retorne Verdadeiro  # Aceitar solução pior com probabilidade
        Senão:
            Retorne Falso  # Rejeitar solução pior


Função simulated_annealing(num_iterations,
            initial_temperature, cooling_rate,
            route, car_assignments, cars):
    current_cost = calculate_total_distance(route, car_assignments, cars)
    best_route = route
    best_car_assignments = car_assignments
    best_cost = current_cost

    Para cada iteração de 1 até num_iterations faça:
        temperatura = initial_temperature * exp(-cooling_rate * iteração)
        neighbor_route, neighbor_car_assignments =
                generate_random_neighbor(route, car_assignments)
        neighbor_cost = calculate_total_distance(neighbor_route, neighbor_car_assignments, cars)
```

```
        Se accept_solution(current_cost, neighbor_cost, temperatura) então:
            route, car_assignments = neighbor_route, neighbor_car_assignments
            current_cost = neighbor_cost

        Se current_cost < best_cost então:
            best_route, best_car_assignments, best_cost = route, car_assignments, current_cost

    Retorne best_route, best_car_assignments, best_cost
```

A função `simulated_annealing` recebe vários parâmetros: `num_iterations`, `initial_temperature`, `cooling_rate`, `route`, `car_assignments` e `cars`.

- `num_iterations` especifica o número de iterações que o algoritmo será executado.
- `initial_temperature` é a temperatura inicial do sistema, controlando a probabilidade de aceitar soluções piores.
- `cooling_rate` determina quão rapidamente a temperatura diminui ao longo do tempo.
- `route` é uma lista que representa a rota atual.
- `car_assignments` é uma lista que representa as atribuições atuais de carros para clientes.
- `cars` é uma lista de carros disponíveis.

A função inicializa o custo atual chamando a função `calculate_total_distance`, passando a rota atual, atribuições de carros e carros. Ela também inicializa as variáveis `best_route`, `best_car_assignments` e `best_cost` com os valores atuais.

O algoritmo entra em um loop que é executado por `num_iterations` iterações. Em cada iteração, calcula a temperatura atual com base na temperatura inicial e na taxa de resfriamento. Em seguida, gera uma solução vizinha aleatória chamando a função `generate_random_neighbor`, passando a rota atual e as atribuições de carros. A função `generate_random_neighbor` retorna uma nova rota e atribuições de carros.

O algoritmo calcula o custo da solução vizinha chamando a função `calculate_total_distance` com a rota vizinha, atribuições de carros vizinhas e carros. Em seguida, verifica se a solução vizinha

deve ser aceita como a nova solução atual. Isso é determinado pela função `accept_solution`, que compara o custo atual e o custo vizinho com a temperatura. Se a solução vizinha for aceita, a rota atual, as atribuições de carros e o custo são atualizados.

O algoritmo também verifica se o custo atual é melhor do que o melhor custo encontrado até agora. Se for, a melhor rota, as atribuições de carros e o melhor custo são atualizados.

Após o término do loop, a função retorna a melhor rota, atribuições de carros e custo.

Em resumo, o código implementa o algoritmo de Simulated Annealing para encontrar a melhor rota e as atribuições de carros para um problema dado. Ele explora iterativamente soluções vizinhas e aceita soluções piores com uma certa probabilidade baseada na temperatura. O algoritmo diminui gradualmente a temperatura ao longo do tempo, permitindo escapar de ótimos locais e potencialmente encontrar soluções melhores.

5.3. Busc Tabu

Abaixo está descrito duas funções para o algoritmo Busca Tabu.

A função `objective_function` recebe uma rota (uma lista de locais), `car_assignments` (uma lista de atribuições de carros para cada local) e `cars` (uma lista de carros disponíveis). Ela chama a função `calculate_total_distance_geral` para calcular a distância total da rota, levando em consideração as atribuições de carros. A função objetivo retorna esta distância total.

A função `get_neighbors` recebe uma rota e `car_assignments`. Ela inicializa uma lista vazia chamada `neighbors`. Em seguida, itera sobre os locais na rota usando um loop aninhado. No loop interno, ela troca as posições de duas cidades na rota para criar uma nova rota vizinha. Esta nova rota, juntamente com as atribuições de carros originais, é adicionada à lista `neighbors`.

Em seguida, ela itera sobre as atribuições de carros usando outro loop aninhado. No loop interno, ela troca as atribuições de carros de dois locais na lista `car_assignments` para criar uma nova atribuição de carro vizinha. Ela adiciona a rota original e esta nova atribuição de carro à lista `neighbors`.

Finalmente, a função retorna a lista `neighbors`, que contém todas as possíveis soluções vizinhas para a rota e as atribuições de carros fornecidas.

```
Função objective_function(rota, atribuicoes_carros, carros):
    distancia_total = calcular_distancia_total_geral(rota, atribuicoes_carros, carros)
    retornar distancia_total
```

```
Função get_neighbors(rota, atribuicoes_carros):
    vizinhos = []

    Para cada local em rota:
        Para cada outro local em rota:
            trocar_posicoes(rota, local, outro_local)
            adicionar_a(vizinhos, (rota, atribuicoes_carros))
            // Adicionar rota vizinha com atribuições de carros originais

    Para cada local em atribuicoes_carros:
        Para cada outro local em atribuicoes_carros:
            trocar_atribuicoes_carros(atribuicoes_carros, local, outro_local)
            adicionar_a(vizinhos, (rota, atribuicoes_carros))
            // Adicionar rota original com atribuições de carros vizinha

    retornar vizinhos
```

Abaixo o algoritmo Busca Tabu para resolver um problema de atribuição de carros. O objetivo é atribuir um conjunto de carros a um conjunto de cidades de forma a minimizar a distância total percorrida.

O algoritmo começa gerando uma solução inicial de maneira aleatória. As atribuições de carros são representadas por uma lista de inteiros, onde cada inteiro corresponde ao índice do carro atribuído a uma cidade. A rota inicial também é gerada aleatoriamente, representando a ordem em que as cidades são visitadas.

A melhor rota e as atribuições de carros são inicialmente definidas como a solução inicial, e a melhor distância é calculada usando a função objetivo.

O algoritmo então entra em um loop que é executado por um número especificado de iterações. Em cada iteração, o algoritmo gera um conjunto de soluções vizinhas fazendo pequenas modificações na solução atual. Essas modificações podem incluir a troca das atribuições de carros de duas cidades ou a alteração da ordem em que duas cidades são visitadas.

O algoritmo avalia cada solução vizinha usando a função objetivo e seleciona a melhor como

a próxima solução atual. No entanto, o algoritmo também mantém uma lista tabu, que registra soluções visitadas recentemente que devem ser evitadas. Se uma solução vizinha estiver na lista tabu, ela não será considerada como candidata para a próxima solução atual.

Após selecionar a melhor solução vizinha, ela é adicionada à lista tabu. Se a lista tabu exceder um tamanho especificado, a solução mais antiga é removida da lista.

Finalmente, se a distância da solução atual for melhor do que a melhor distância encontrada até agora, a solução atual torna-se a nova melhor solução.

O algoritmo continua até que uma condição de parada seja atendida, que pode ser um número máximo de iterações ou a ausência de soluções melhoradoras.

O algoritmo retorna a melhor rota, atribuições de carros e distância encontrada durante a busca.

Em resumo, o algoritmo Busca Tabu explora iterativamente o espaço de soluções considerando soluções vizinhas e evitando soluções visitadas anteriormente. Isso permite escapar de ótimos locais e encontrar soluções melhores.

```
Função TabuSearch():
    // Inicialização
    melhor_rota, melhores_atribuicoes_carros = GerarSolucaoInicialAleatoria()
    melhor_distancia = CalcularDistanciaTotal(melhor_rota, melhores_atribuicoes_carros)
    lista_tabu = ListaVazia()

    // Parâmetros do algoritmo
    numero_iteracoes = EspecificarNumeroIteracoes()
    tamanho_maximo_lista_tabu = EspecificarTamanhoMaximoListaTabu()

    Para cada iteração de 1 até numero_iteracoes:
        // Geração de soluções vizinhas
        vizinhos = GerarVizinhos(melhor_rota, melhores_atribuicoes_carros, lista_tabu)

        // Avaliação dos vizinhos
        Para cada vizinho em vizinhos:
            distancia_vizinho = CalcularDistanciaTotal(vizinho.rota, vizinho.atribuicoes_carros)
```

```
            Se vizinho não está na lista_tabu e distancia_vizinho < melhor_distancia:
                melhor_rota = vizinho.rota
                melhores_atribuicoes_carros = vizinho.atribuicoes_carros
                melhor_distancia = distancia_vizinho
                AdicionarVizinhoAListaTabu(vizinho, lista_tabu)

                Se tamanho_lista_tabu > tamanho_maximo_lista_tabu:
                    RemoverSolucaoMaisAntigaDaListaTabu(lista_tabu)

    Retornar melhor_rota, melhores_atribuicoes_carros, melhor_distancia
```

5.4. Algoritmo Genético

5.4.1. Funções Auxiliares

A função generate_random_individual gera um indivíduo aleatório para a população. Ela recebe dois parâmetros: num_cities e num_cars. Inicia o indivíduo com um único elemento, 0. Em seguida, adiciona inteiros aleatórios entre 0 e num_cars - 1 ao indivíduo para os elementos restantes num_cities - 1. Por fim, retorna o indivíduo como uma lista de duas listas: a primeira lista representa a ordem das cidades a serem visitadas (excluindo a cidade inicial e final), e a segunda lista representa a atribuição de carros a cada cidade.

A função initialize_population gera uma população de indivíduos aleatórios. Ela recebe três parâmetros: pop_size, num_cities e num_cars. Utiliza uma compreensão de lista para criar uma lista de pop_size indivíduos aleatórios chamando a função generate_random_individual. Por fim, retorna a população como uma lista de indivíduos.

A função evaluate_population avalia o fitness de cada indivíduo na população. Recebe dois parâmetros: population e cars. Inicializa uma lista vazia chamada fitness_scores. Em seguida, itera sobre cada indivíduo na população e calcula a distância total da rota chamando a função calculate_total_distance. O escore de fitness é calculado como o inverso da distância total. O escore de fitness é então adicionado à lista fitness_scores. Por fim, retorna a lista fitness_scores.

A função select_parents seleciona dois pais da população com base em seus escores de fitness.

Recebe dois parâmetros: population e fitness_scores. Calcula o fitness total somando todos os escores de fitness. Em seguida, calcula as probabilidades de selecionar cada indivíduo como pai dividindo cada escore de fitness pelo fitness total. A função random.choices é usada para selecionar dois pais da população com base nas probabilidades calculadas. Por fim, retorna os pais selecionados como uma lista.

A função crossover_route realiza a crossover entre duas rotas. Recebe três parâmetros: list1, list2 e crossover_point. Divide list1 e list2 no crossover_point e cria duas novas listas: left_part1 e left_part2. Em seguida, cria result1 concatenando left_part1 com os elementos de list2 que ainda não estão em left_part1. De maneira semelhante, cria result2 concatenando left_part2 com os elementos de list1 que ainda não estão em left_part2. Por fim, retorna result1 e result2.

A função crossover realiza a crossover entre dois pais. Recebe dois parâmetros: parent1 e parent2. Gera um ponto de crossover aleatório entre 1 e o comprimento de parent1 - 1. Em seguida, chama a função crossover_route para realizar a crossover na primeira lista de cada pai. As listas resultantes são armazenadas em result1 e result2. A segunda lista de cada pai também é cruzada no mesmo ponto de crossover. Por fim, retorna dois filhos como uma lista de dois indivíduos.

A função mutate realiza a mutação em um indivíduo. Recebe três parâmetros: individual, mutation_rate e num_cars. Itera sobre a segunda lista do indivíduo (representando a atribuição de carros a cidades) a partir do índice 1. Para cada elemento, verifica se um número aleatório entre 0 e 1 é menor que a taxa de mutação. Se for, atribui um número inteiro aleatório entre 0 e num_cars - 1 ao elemento. Por fim, retorna o indivíduo mutado.

No geral, essas funções são usadas no algoritmo genético para gerar uma população inicial, avaliar o fitness de cada indivíduo, selecionar pais para reprodução, realizar crossover e mutação, e criar uma nova população para a próxima geração.

```
% Insira o código do Algoritmo Genético aqui
```

5.4.2. Pseudocódigo das Funções Auxiliares

Função ***generate_random_individual***

```
Função: generate_random_individual(num_cities, num_cars)
    individual = [0] + [random.randint(0, num_cars - 1) for _ in range(num_cities - 1)]
    Retornar [individual[:num_cities-1], individual[num_cities-1:]]
```

Função initialize_population

```
Função: initialize_population(pop_size, num_cities, num_cars)
    population = [generate_random_individual(num_cities, num_cars) for _ in range(pop_size)]
    Retornar population
```

Função evaluate_population

```
Função: evaluate_population(population, cars)
    fitness_scores = []
    Para cada indivíduo em population:
        total_distance = calculate_total_distance(indivíduo[0], indivíduo[1], cars)
        fitness = 1 / total_distance
        Adicionar fitness à lista fitness_scores
    Retornar fitness_scores
```

Função select_parents

```
Função: select_parents(population, fitness_scores)
    total_fitness = sum(fitness_scores)
    probabilidades = [fitness / total_fitness for fitness in fitness_scores]
    pais_selecionados = random.choices(population, weights=probabilidades, k=2)
    Retornar pais_selecionados
```

Função crossover_route

```
Função: crossover_route(list1, list2, crossover_point)
    left_part1 = list1[:crossover_point]
    left_part2 = list2[:crossover_point]
    result1 = left_part1 + [elemento for elemento in list2 if elemento not in left_part1]
    result2 = left_part2 + [elemento for elemento in list1 if elemento not in left_part2]
    Retornar result1, result2
```

Função crossover

```
Função: crossover(parent1, parent2)
    ponto_crossover = random.randint(1, len(parent1) - 1)
```

```
    result1_parte1, result2_parte1 = crossover_route(parent1[0], parent2[0], ponto_crossover)
    result1_parte2, result2_parte2 = crossover_route(parent1[1], parent2[1], ponto_crossover)
    Retornar [[result1_parte1, result1_parte2], [result2_parte1, result2_parte2]]
```

Função `mutate`

```
Função: mutate(individual, mutation_rate, num_cars)
    Para cada elemento na segunda lista de individual a partir do índice 1:
        Se um número aleatório entre 0 e 1 for menor que a taxa de mutação:
            Atribuir um número inteiro aleatório entre 0 e num_cars - 1 ao elemento
    Retornar individual
```

5.4.3. Função Principal

Abaixo está a descrição do algoritmo genético para resolver um problema relacionado a carros e cidades.

1. A função `genetic_algorithm` recebe vários parâmetros: `num_generations` (o número de gerações para executar o algoritmo), `pop_size` (o tamanho da população), `mutation_rate` (a probabilidade de mutação), `num_cities` (o número de cidades no problema), `num_cars` (o número de carros disponíveis) e `cars` (uma lista de carros com suas respectivas capacidades).
2. A função começa inicializando a população usando a função `initialize_population`. Essa função cria uma população aleatória de indivíduos, onde cada indivíduo representa uma possível solução para o problema. Cada indivíduo é uma lista de cidades, e cada cidade é representada por um índice.
3. A lista `best_individual_list` é inicializada como uma lista vazia. Esta lista armazenará o melhor indivíduo encontrado em cada geração.
4. O algoritmo então entra em um loop que itera sobre o número especificado de gerações. Em cada geração, os seguintes passos são realizados:
   (a) Os escores de aptidão da população são avaliados usando a função `evaluate_population`. Esta função calcula o escore de aptidão para cada indivíduo na população com base na distância total percorrida.
   (b) Uma nova população é criada selecionando pais da população atual, realizando operações de crossover e mutação, e adicionando os filhos resultantes à nova população. A função

`select_parents` é usada para selecionar dois pais com base em seus escores de aptidão. A função `crossover` é usada para criar dois filhos combinando o material genético dos pais. A função `mutate` é usada para introduzir mudanças aleatórias (mutações) no material genético dos filhos.

(c) A nova população é adicionada à lista `new_population`.

(d) O melhor indivíduo na geração atual é determinado encontrando o indivíduo com o maior escore de aptidão. Isso é feito usando a função `max` com uma função lambda como chave. A função lambda calcula o escore de aptidão para cada indivíduo chamando a função `calculate_total_distance`.

(e) O melhor indivíduo na geração atual é adicionado à `best_individual_list`.

5. Após todas as gerações terem sido processadas, o algoritmo encontra a melhor solução na `best_individual_list`. Isso é feito encontrando o indivíduo com o maior escore de aptidão, novamente usando a função `max` com uma função lambda como chave.
6. A distância total da melhor solução é calculada chamando a função `calculate_total_distance` com as cidades e carros do melhor indivíduo.
7. Finalmente, o melhor indivíduo e sua distância são retornados como resultado da função `genetic_algorithm`.

```
Função genetic_algorithm(num_generations, pop_size, mutation_rate, num_cities, num_cars, cars):
    # Inicialização da população
    population = initialize_population(pop_size, num_cities)

    # Lista para armazenar o melhor indivíduo de cada geração
    best_individual_list = []

    # Loop sobre o número de gerações
    Para cada geração de 1 até num_generations:
        # Avaliação do fitness da população
        fitness_scores = evaluate_population(population, cars)

        # Criação de uma nova população
```

```
    new_population = []
    Para cada indivíduo na população:
        # Seleção de pais
        parent1, parent2 = select_parents(population, fitness_scores)

        # Crossover e mutação
        child1, child2 = crossover(parent1, parent2)
        child1 = mutate(child1, mutation_rate, num_cars)
        child2 = mutate(child2, mutation_rate, num_cars)

        # Adição dos filhos à nova população
        Adicione child1 e child2 à new_population

    # Substituição da população antiga pela nova
    population = new_population

    # Encontrar o melhor indivíduo na geração atual
    best_individual = Encontrar o indivíduo com maior fitness em population

    # Adicionar o melhor indivíduo à lista
    Adicione best_individual à best_individual_list

# Encontrar o melhor indivíduo global
best_global_individual = Encontrar o indivíduo com maior fitness em best_individual_list

# Calcular a distância total da melhor solução
best_distance = calculate_total_distance(best_global_individual, cars)

# Retornar o melhor indivíduo e sua distância
Retorne best_global_individual, best_distance
```

O algoritmo foi rodado 2 vezes, uma vez iniciado com um mutation_rate=0.9 e no outro o mutation rate é 0.1.

6. Complexidade

6.1. Simulated Annealing

A complexidade de tempo do Simulated Annealing é $O(n^2)$. O trecho de código contém um loop que itera num_iterations vezes, o que é proporcional ao número de cidades. Dentro do loop, há uma chamada para a função generate_random_neighbor, que tem uma complexidade de tempo de $O(n)$. Além disso, há uma chamada para a função calculate_total_distance_geral, que também possui uma complexidade de tempo de $O(n)$. Portanto, a complexidade de tempo geral do Simulated Annealing é $O(n^2)$, onde n é o número de cidades.

6.2. Busca Tabu

A complexidade de tempo do `get_neighbors` é $O(n^2)$. O `get_neighbors` contém dois loops aninhados, cada um iterando sobre o número de localidades `num_locations`. Portanto, a complexidade de tempo é $O(n^2)$, onde n é o número de localidades.

A complexidade de tempo da Busca Tabu é $O(n^2)$. A Busca Tabu contém dois loops aninhados. O loop externo itera `num_iterations` vezes, que é um valor constante. O loop interno itera sobre a lista `neighbors`, que tem um comprimento máximo de `num_cities` * `num_cars`. Portanto, a complexidade de tempo geral é $O($ `num_iterations` $*$ `num_cities` $*$ `num_cars` $)$, que pode ser simplificada para $O(n^2)$, onde n representa o valor máximo entre num_iterations, num_cities e num_cars.

6.3. Genético

A complexidade de tempo do `calculate_total_distance` é $O(n)$. O `calculate_total_distance` possui um único loop `for` que itera pela lista `route`. O número de iterações no loop é diretamente proporcional ao tamanho da lista `route`, representado por n. Portanto, a complexidade de tempo do código é $O(n)$.

A complexidade de tempo do generate_random_individual é $O(n)$. Possui um loop `for` que itera `num_cities - 1` vezes. Dentro do loop, a operação de `append` leva tempo constante. Portanto, a complexidade de tempo geral é $O(n)$, onde n é o valor de `num_cities`.

A complexidade de tempo do `initialize_population` é $O($ `pop_size` $*$ `num_cities`$)$. A função utiliza uma compreensão de lista para gerar uma população de tamanho `pop_size`. Para cada indivíduo na população, chama a função `generate_random_individual`, que tem uma complexidade de tempo de $O($`num_cities`$)$. Portanto, a complexidade de tempo geral do trecho de código é $O($`pop_size` $*$ `num_cities`$)$.

A complexidade de tempo do evaluate_population é $O(n)$. O evaluate_population possui um loop `for` que itera sobre cada indivíduo na população. O número de iterações é diretamente proporcional ao tamanho da população, portanto, a complexidade de tempo é $O(n)$, onde n é o tamanho da população.

A complexidade de tempo do select_parents é $O(n)$, onde n é o comprimento da lista de população. O código itera sobre a lista de `fitness_scores` uma vez para calcular o `total_fitness`, o que leva $O(n)$ tempo. Em seguida, calcula a lista de `probabilities` dividindo cada pontuação de `fitness` pelo `total_fitness`, o que também leva $O(n)$ tempo. Finalmente, usa a função `random.choices` para selecionar dois pais da lista de população com base nas probabilidades, o que tem uma complexidade de tempo de $O(1)$ já que é uma operação de tempo constante. Portanto, a complexidade de tempo geral do código é $O(n)$.

A complexidade de tempo do crossover_route é $O(n^2)$. Contém dois loops `for` que iteram sobre as listas `list1` e `list2`. O primeiro loop `for` itera sobre `list2` para verificar se cada elemento não está em `left_part1`, resultando em uma complexidade de tempo de $O(n)$. O segundo loop `for` itera sobre `list1` para verificar se cada elemento não está em `left_part2`, resultando em uma complexidade de tempo de $O(n)$. Portanto, a complexidade de tempo geral do trecho de código é $O(n^2)$, já que os dois loops `for` estão aninhados.

A complexidade de tempo do crossover é $O(n^2)$, pois chama crossover_route.

A complexidade de tempo do mutate é $O(n)$, onde n é o comprimento da lista `individual`. O loop `for` itera pela lista `individual` uma vez, portanto, a complexidade de tempo é diretamente proporcional ao comprimento da lista. As funções `random.random()` e `random.randint()` têm uma complexidade de tempo constante, então elas não afetam a complexidade de tempo geral do trecho de código.

A complexidade de tempo do 'genetic_algorithm' é $O($`num_generations` $*$ `pop_size` $*$ (`num_cities` $*$ `num_cars` $+$ `num_cities` $*$ `num_cars` $+$ `num_cities` $*$ `num_cars`$))$.

O trecho de código consiste em loops aninhados. O loop externo itera `num_generations` vezes,

o loop do meio itera $\frac{pop\_size}{2}$ vezes, e o loop interno itera $num\_cities \times num\_cars$ vezes para cada filho. Dentro do loop interno, existem três operações: crossover, mutate e extend. A operação de crossover têm uma complexidade de tempo de $O(num\_cities \times num\_cars)$, e a operação de extend tem uma complexidade de tempo de $O(1)$. Portanto, a complexidade de tempo geral é $O(num\_generations \times pop\_size \times (num\_cities \times num\_cars + num\_cities \times num\_cars + num\_cities \times num\_cars))$.

7. Resultados

Nesta seção, apresentamos os resultados obtidos a partir da aplicação dos algoritmos para resolver o Problema do Caixeiro Viajante com Carros (CaRS). Incluímos tabelas e gráficos que destacam as principais métricas de desempenho.

7.1. Descrição do Computador

```
Computador:
- Modelo: MacBook Pro
- Processador: 2,6 GHz Intel Core i7 6-Core
- Memória RAM: 16 GB 2667 MHz DDR4
- Sistema Operacional: 14.1.1 (23B81)
- Outras Especificações: AMD Radeon Pro 5300M 4 GB Intel UHD Graphics 630 1536 MB
```

Configuração Experimental

O estudo experimental teve como objetivo avaliar o desempenho de três algoritmos de otimização - Genético(com 0.1 e 0.9 de mutação), Busca Tabu e Simulated Annealing - na resolução do problema do Sistema de Aluguel de Carros (CaRS). Para as instâncias não euclidianas, o conjunto de dados consistiu em 20 instâncias, e as melhores soluções conhecidas foram extraídas de uma fonte confiável, especificamente o artigo intitulado "Formulações eficientes para o problema do viajante alugador de carros e sua variante de cota" de Mathieu Lacroix, Yasmín A. Ríos-Solís e Roberto Wolfler Calvo [13].

Fonte do Conjunto de Dados

O conjunto de dados, a descrição de cada grupo de instâncias e os formatos de arquivo estão disponíveis em http://www.dimap.ufrn.br/lae/en/projects/CaRS.php.

Resultados Experimentais

As instâncias não euclidianas foram escolhidas com base no conjunto de dados utilizado no artigo de 2021. Cada algoritmo (Genético(com 0.1 e 0.9 de mutação) , Busca Tabu e Simulated Annealing) foi executado 50 vezes para cada uma das 20 instâncias. Os resultados, incluindo o nome da instância, o número de cidades, o número de carros disponíveis, a melhor solução encontrada, o tempo de processamento, a pior solução, a solução média e o desvio padrão, foram registrados.

Resumo dos Resultados

A Tabela 1 apresenta um resumo dos resultados, incluindo a frequência de encontrar a melhor solução por cada algoritmo para as instâncias não euclidianas. As melhores soluções conhecidas foram extraídas do artigo de 2021, proporcionando uma referência para avaliar o desempenho dos algoritmos.

7.2. Tabela de Resultados

Instances			GENETIC					EVOLUCIONARY					TABU					SIMULATED ANEALING				
File	Car	#Best	T	std_dev	Avg	Best	Hit	T	std_dev	Avg	Best	Hit	T	std_dev	Avg	Best	Hit	T	std_dev	Avg	Best	Hit
Mauritania10n	2	306	34.071	32.045	1295.08	1214	1	56.045	28.669	1297.14	1224	1	0.01	95.364	1422.28	1200	1	0.001	240.087	1609.02	1253	1
Colombia11n	2	461	1.228	53.984	2420.4	2282	1	1.206	65.228	2402.76	2254	1	0.015	127.452	1215.18	967	1	0.001	662.134	1945.86	1013	1
Libia14n	2	504	1.217	49.851	1660.48	1564	1	1.213	59.033	1653.04	1458	1	0.027	110.995	1679.06	1442	1	0.001	201.668	1975.4	1515	1
BrasilRJ14n	2	101	1.235	9.243	277.12	255	1	1.22	7.611	276.78	261	2	0.027	25.937	289.06	237	1	0.001	150.156	450.06	269	1
Congo15n	2	573	1.28	57.858	3160.84	3049	1	1.262	62.101	3142.96	3023	1	0.035	178.122	2869.7	2491	1	0.002	218.835	3210.52	2614	1
Niger12n	3	607	11.663	86.751	3081.18	2793	1	1.182	79.523	3083.96	2830	1	0.019	144.493	2256.42	1976	1	0.001	410.704	2575.18	2054	1
Mongolia13n	3	551	38.081	51.42	2546.86	2412	1	33.713	59.973	2548.86	2333	1	0.128	157.73	2134.72	1812	1	0.007	317.91	2575.22	1897	1
Indonesia14n	3	522	1.227	79.677	3693.54	3457	1	1.212	87.237	3724.18	3515	1	0.03	171.191	2242.66	1908	1	0.001	654.331	3256.72	2120	1
India16n	3	723	1.253	106.127	4121.24	3813	1	1.245	91.742	4109.52	3756	1	0.044	174.673	3885.6	3382	1	0.002	301.973	4391.3	3758	1
China17n	3	638	1.301	87.717	2126.2	1748	1	1.287	76.814	2139.96	1979	1	0.055	191.751	2052.3	1746	1	0.002	410.252	2633.26	1895	1
BrasilAM26n	3	107	1.444	20.069	921.18	866	1	1.415	16.77	923.62	877	1	0.187	65.752	784.2	680	1	0.005	123.544	962.68	786	1
Egito9n	4	520	1.185	51.328	2673.48	2567	1	1.145	62.249	2662.9	2526	1	0.009	206.034	1624.68	1299	1	0.001	475.504	2437.24	1378	1
Etiopia10n	4	403	1.178	72.117	2374	2099	1	1.167	65.469	2381.26	2151	1	0.012	303.119	1664.62	1181	1	0.001	458.963	2183.08	1335	1
Mali11n	4	494	1.196	64.898	3564.7	3417	1	6.886	63.935	3582.62	3334	1	0.016	326.486	1840.08	1429	1	0.001	870.543	3013.42	1380	1
Chade12n	4	649	1.238	80.052	2263.14	2054	1	1.223	62.913	2265.66	2129	1	0.022	182.607	2328.62	1969	1	0.001	421.195	2934.28	2069	1
Ira13n	4	693	1.212	100.864	2991.18	2619	1	1.193	71.189	2984.12	2703	1	0.025	193.506	3033.54	2653	1	0.001	690.597	3885.52	2896	1
Mexico14n	4	610	36.398	91.769	3644.44	3408	1	38.846	92.129	3638.18	3375	1	0.17	145.843	2301.16	2014	1	0.007	698.525	3476.66	2353	1
Canada17n	4	824	1.302	84.548	5366.42	5086	1	1.281	111.498	5327.78	4971	1	0.058	258.872	4547.56	4093	1	0.002	402.369	5351.42	4367	1
BrasilMG30n	4	160	1.478	20.233	1178.3	1134	2	1.46	24.869	1177.02	1057	1	0.355	45.876	926.22	833	1	0.006	155.651	1106.86	834	1
Cazaquistao15n	5	830	1.283	80.778	4743.6	4533	1	1.264	84.687	4712.86	4438	1	0.043	187.956	3465.78	3095	1	0.002	640.517	4340.84	3492	1

Table 1: Resultados

Apêndice: Códigos

Algoritmos : https://github.com/arturo32/PCA/blob/main/trabalho_heuristica.ipynb

Plots : https://github.com/arturo32/PCA/blob/main/plot.ipynb

Apêndice: Vídeos

Algoritmo Exato: https://drive.google.com/file/d/1R7if7Z9dRCn6RN5ueGdKyJa1cD8WKSan/view?usp=sharing

References

[1] Marco C. Goldbarg, Elizabeth F. G. Goldbarg, Henrique P. L. Luna, Matheus S. Menezes, Lucas Corrales, Integer programming models and linearizations for the traveling car renter problem, Optim Lett 12 (2018) 743–761doi:10.1007/s11590-017-1138-5.

[2] GOLDBARG, Marco Cesar ; Asconavieta, P. ; GOLDBARG, E. F. G., Algoritmos evolucionários na solução do problema do caixeiro alugador, Computação Evolucionária em Problemas de Engenharia (2011) 301–330.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.

[4] B. H. O. Rios, Hibridização de Meta-Heurísticas com Métodos Baseados em Programação Linear para o Problema do Caixeiro Alugador, Dissertação (Mestrado em Sistemas e Computação) - Centro de Ciências Exatas e da Terra, Universidade Federal do Rio Grande do Norte, 2018.

[5] R. Bellman, Dynamic programming treatment of the travelling salesman problem, Journal of Assoc. Computing Mach. 9.

[6] A. Ambainis, K. Balodis, J. Iraids, M. Kokainis, K. Prūsis, J. Vihrovs, Quantum Speedups for Exponential-Time Dynamic Programming Algorithms, pp. 1783–1793. arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611975482.107, doi:10.1137/1.9781611975482.107.
URL https://epubs.siam.org/doi/abs/10.1137/1.9781611975482.107

[7] V. C. . W. J. C. David L. Applegate, Robert E. Bixby, The Traveling Salesman Problem: A Computational Study, Princeton University Press, 2006.

[8] H. T. Sadeeq, A. M. Abdulazeez, Metaheuristics: A review of algorithms, International Journal of Online and Biomedical Engineering 19 (09). doi:10.3991/ijoe.v19i09.39683.
URL https://doi.org/10.3991/ijoe.v19i09.39683

[9] L. N. De Castro, F. J. Von Zuben, The Clonal Selection Algorithm with Engineering Applications, Vol. 2000, 2000.

[10] A. E. Ezugwu, A. K. Shukla, R. Nath, A. A. Akinyelu, J. O. Agushaka, H. Chiroma, P. K. Muhuri, Meta-heuristics: a comprehensive overview and classification along with bibliometric analysis, Artificial Intelligence Review 54 (2021) 4237–4316, published: 16 March 2021. doi:10.1007/s10462-021-09923-3. URL https://link.springer.com/article/10.1007/s10462-021-09923-3

[11] D. Felipe, E. Ferreira Gouvêa Goldbarg, M. C. Goldbarg, Scientific algorithms for the car renter salesman problem (2014) 873–879doi:10.1109/CEC.2014.6900556.

[12] M. C. Goldbarg, P. H. Asconavieta, E. F. G. Goldbarg, Memetic algorithm for the traveling car renter problem: an experimental investigation, Memetic Computing 4 2. doi:10.1007/s12293-011-0070-y.

[13] M. Lacroix, Y. A. Ríos-Solís, R. Wolfler Calvo, Efficient formulations for the traveling car renter problem and its quota variant, Journal of Heuristics 27 (1) (2021) 53–90.