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 entre 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 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 cade 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 uma implementação em python de um algoritmo exata e algumas metaheurística para o problema do PCA, se baseando em algoritmos presentes na literatura.

2. Descrição do problema

Começaremos descrevendo mais formalmente o PCV, para depois descrever o 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 soma de pesos possível em G, isto é, um ciclo que passe por todas as vértices do grafo.

2.2. Problemas 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*), 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. Estado da arte algoritmos exatos para o problema do PCV

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].

4. Estado da arte algoritmos Heurísticos

Algoritmos metaheurísticos 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].

4.1. Categorias de Algoritmos Metaheurísticos

O International Journal of Online and Biomedical Engineering (iJOE) classificou os algoritmos metaheurísticos 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 metaheurísticos 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 metaheurísticas [9] [10].

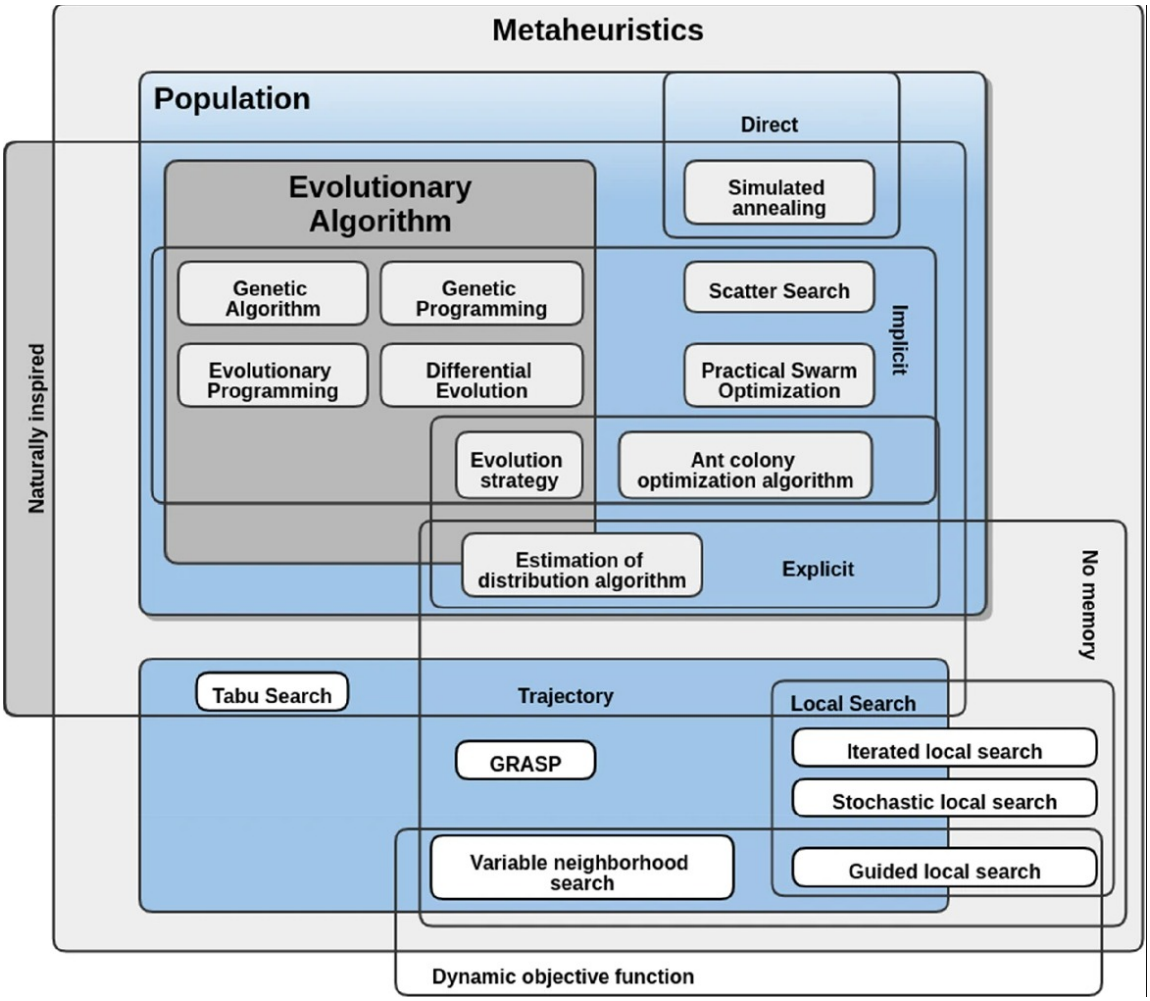

Figure 1: Classificações generalizadas de técnicas de busca de metaheurísticas (De Castro and Von Zuben 2000).

4.2. Metaheurísticas Recentes

4.2.1. 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].

4.2.2. 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].

4.2.3. 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].

4.2.4. 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].

5. Problema do Caixeiro Viajante com Aluguel de Carros

5.1. Gerador de Instância de Problema

O codigo do script em Python que gera um arquivo contendo instâncias para o problema CaRS (Roteamento e Agendamento de Carros) esta em Apêndice: Códigos, Listing 1: Instância de Problema. O arquivo gerado segue um formato específico e inclui informações sobre o número de cidades, número de carros, pesos de arestas e taxas de retorno.

O script começa importando o módulo `random`, que é utilizado para gerar números aleatórios posteriormente no código.

A função `generate_symmetric_matrix` recebe um parâmetro `size` e retorna uma matriz simétrica de tamanho `size x size`. A matriz é inicializada com todos os elementos configurados como 0. Em seguida, um loop aninhado é usado para iterar sobre a matriz e atribuir valores aleatórios entre 1 e 100 à parte triangular superior da matriz. A parte triangular inferior é preenchida com os mesmos valores para tornar a matriz simétrica. Finalmente, a função retorna a matriz gerada.

A função `generate_cars_instance_file` recebe três parâmetros: `citys`, `cars` e `file_name`. Ela abre um arquivo com o `file_name` fornecido no modo de escrita usando uma instrução `with`, que garante que o arquivo seja fechado corretamente após a escrita. A função então escreve várias informações sobre a instância no arquivo, incluindo o nome, tipo, comentário, dimensão, número de carros, tipo de peso de aresta e formato de peso de aresta.

Em seguida, a função escreve a seção `EDGE_WEIGHT_SECTION`, que contém os pesos de arestas entre cidades. Ela itera sobre o intervalo de `cars` e escreve o índice do carro no arquivo. Em seguida, gera uma matriz simétrica usando a função `generate_symmetric_matrix` e escreve os elementos da matriz no arquivo, separados por espaços. Após escrever todos os pesos de arestas para um carro, é escrita uma quebra de linha para separar os carros.

Após escrever a seção `EDGE_WEIGHT_SECTION`, a função escreve a seção `RETURN_RATE_SECTION`, que contém as taxas de retorno para cada carro. Ela segue um processo semelhante à `EDGE_WEIGHT_SECTION`, gerando uma matriz simétrica e escrevendo os elementos da matriz no arquivo.

Finalmente, a função escreve o marcador `EOF` (Fim do Arquivo) para indicar o final do arquivo.

O exemplo de uso no final do código chama a função `generate_cars_instance_file` com os parâmetros `6` (número de cidades), `2` (número de carros) e `"teste.car"` (nome do arquivo). Isso irá gerar um arquivo chamado "teste.car" com instâncias para o problema CaRS.

Um pseudo código do algoritmo descrito acima pode ser encontrado abaixo:

```
 gerar_matriz_simetrica(tamanho):
iz = inicializar_matriz(tamanho)

 cada linha i de 0 até tamanho-1 faça:
para cada coluna j de i até tamanho-1 faça:
    se i != j:
        valor = gerar_valor_aleatorio(1, 100)
        matriz[i][j] = valor
        matriz[j][i] = valor  // Tornar a matriz simétrica

rnar matriz

 gerar_instancia_cars(cidades, carros, nome_arquivo):
r_arquivo(nome_arquivo, modo='w') como arquivo:
escrever_linha(arquivo, "NAME : Test{n}\n".format(cidades))
escrever_linha(arquivo, "TYPE : CaRS\n")
escrever_linha(arquivo, "COMMENT : Instâncias para o Problema CaRS\n")
escrever_linha(arquivo, "DIMENSION : {n}\n".format(cidades))
escrever_linha(arquivo, "CARS_NUMBER : {n}\n".format(carros))
escrever_linha(arquivo, "EDGE_WEIGHT_TYPE : EXPLICIT\n")
escrever_linha(arquivo, "EDGE_WEIGHT_FORMAT : FULL_MATRIX\n")

escrever_linha(arquivo, "EDGE_WEIGHT_SECTION\n")
para cada carro de 0 até carros-1 faça:
    escrever_linha(arquivo, "{carro}\n".format(carro))
    matriz = gerar_matriz_simetrica(cidades)
    para cada linha na matriz faça:
        para cada valor na linha faça:
            escrever_linha(arquivo, "{valor} ".format(valor))
        escrever_linha(arquivo, "\n")

escrever_linha(arquivo, "RETURN_RATE_SECTION\n")
para cada carro de 0 até carros-1 faça:
    escrever_linha(arquivo, "{carro}\n".format(carro))
    matriz = gerar_matriz_simetrica(cidades)
    para cada linha na matriz faça:
        para cada valor na linha faça:
            escrever_linha(arquivo, "{valor} ".format(valor))
        escrever_linha(arquivo, "\n")
```

6. Algoritmo exato - Brute Force

Um algoritmo para resolver o Problema do Caixeiro Viajante (TSP) com a adição de carros é brute force. O objetivo é encontrar a rota mais curta que visita todas as localizações e retorna à localização inicial, atribuindo cada localização a um carro.

O codigo esta em Apêndice: Códigos, Listing 3: Algoritmo de Brute-Force. O código começa importando os módulos necessários: `itertools` para gerar permutações e combinações, e `time` para medir o tempo de execução.

6.1. Função `calculate_total_distance`

A função `calculate_total_distance` recebe a rota atual, as atribuições de carros, os carros e a melhor distância encontrada até agora. Esta função calcula a distância total percorrida para uma determinada rota e atribuições de carros. Itera sobre cada localização na rota e verifica se a atribuição de carro atual é diferente da atribuição de carro da próxima localização. Se forem diferentes, adiciona a distância da localização atual para a próxima localização e a taxa de retorno da próxima localização para a localização atual. Se forem iguais, apenas adiciona a distância entre as localizações atual e próxima. Após cada iteração, verifica se a distância total atual é maior que a melhor distância encontrada até agora. Se for, retorna imediatamente a distância total atual. Finalmente, atualiza a localização atual para ser a próxima localização.

6.2. Função `brute_force_tsp_with_cars`

A principal função é chamada `brute_force_tsp_with_cars` e recebe uma instância do problema e uma lista de carros. Ela inicializa algumas variáveis: `num_cars` para armazenar o número de carros, `num_locations` para armazenar o número de localizações, `best_route` para armazenar a melhor rota encontrada até agora, e `best_distance` para armazenar a melhor distância encontrada até agora (inicializada como infinito).

A função então entra em um loop aninhado usando `itertools.permutations` e `itertools.product` para gerar todas as rotas e atribuições de carros possíveis. Garante que a rota comece e termine na localização 0 adicionando-a no início e no final da rota. Para cada combinação de rota e atribuições de carros, ela chama a função `calculate_total_distance` para calcular a distância total percorrida. Se a distância calculada for menor que a melhor distância encontrada até agora, ela atualiza a melhor distância, melhor rota e melhores atribuições de carros.

Após o loop aninhado, a função retorna a melhor rota, melhor distância e melhores atribuições de carros.

Um pseudo código do algoritmo descrito acima pode ser encontrado abaixo:

```
Função calcularDistanciaTotal(rota, atribuicoesCarros, carros, melhorDistancia):
    totalDistancia = 0
    carroAtual = atribuicoesCarros[0]
    localizacaoAtual = 0

    Para cada proximaLocalizacao em rota:
        Se carroAtual não é igual a atribuicoesCarros[proximaLocalizacao]:
            totalDistancia += carros[carroAtual].matrizDistancia[localizacaoAtual][proximaLocaliz
            totalDistancia += carros[carroAtual].matrizTaxaRetorno[localizacaoAtual][proximaLocal
        Senão:
            totalDistancia += carros[carroAtual].matrizDistancia[localizacaoAtual][proximaLocaliz

        Se melhorDistancia < totalDistancia:
            Retorne totalDistancia

        localizacaoAtual = proximaLocalizacao

    Retorne totalDistancia

Função forcaBrutaTSPComCarros(instancia, carros):
    numCarros = instancia.numeroCarros
    numLocalizacoes = instancia.numeroLocalizacoes

    melhorRota = Nulo
    melhorDistancia = Infinito

    Para cada rota em permutações(range(1, numLocalizacoes)):
        Para cada atribuicoesCarros em produto(range(numCarros), repeat=numLocalizacoes):
            RotaCompleta = (0,) + rota + (0,)
            distancia = calcularDistanciaTotal(RotaCompleta, atribuicoesCarros, carros, melhorDis

            Se distancia < melhorDistancia:
                melhorDistancia = distancia
                melhorRota = RotaCompleta

    Retorne melhorRota, melhorDistancia
```

7. Algoritmos Metaheuristicos

Testamos 4 algoritmos: o Simulated Annealing, Tabu Search, Genetico e Evolutivo.

7.1. Funções Auxiliares

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 algorimos 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)
```

7.2. Simulated Annealing

O pseudocigo a seguir é do algoritmo de Simulated Annealing (Recozimento Simulado). O Simulated Annealing é um algoritmo metaheurístico 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.

7.3. Tabu Search

Abaixo está descrito o algoritmo Tabu Search para resolver um problema de roteamento. O objetivo é encontrar a melhor rota e as atribuições de carros que minimizam a distância total percorrida.

A função `generate_neighbors` recebe uma rota e atribuições de carros como entrada e gera uma lista de soluções vizinhas. Isso é feito chamando a função `generate_random_neighbor`, que gera um vizinho aleatório fazendo uma pequena modificação na solução atual. A nova rota e atribuições de carros são adicionadas à lista de vizinhos.

A função `tabu_search` é a função principal que executa o algoritmo Tabu Search. Ela recebe vários parâmetros: `num_iterations` especifica o número de iterações para executar o algoritmo, `tabu_list_size` especifica o tamanho da lista tabu, `initial_route` e `initial_car_assignments` especificam a solução inicial, e `cars` é uma lista de carros disponíveis.

A função inicializa a solução atual, a melhor solução e as variáveis de custo com a solução inicial. Também inicializa uma lista tabu vazia.

O algoritmo então entra em um loop que é executado pelo número especificado de iterações. Em cada iteração, ele gera uma lista de vizinhos chamando a função `generate_neighbors` com a solução atual e atribuições de carros.

Em seguida, itera sobre cada vizinho e calcula o custo da solução vizinha usando a função `calculate_total_distance`. Verifica se o custo do vizinho é melhor que o custo do melhor vizinho atual e se a solução vizinha não está na lista tabu. Se ambas as condições forem atendidas, a solução vizinha torna-se o novo melhor vizinho.

Após iterar sobre todos os vizinhos, o algoritmo verifica se um melhor vizinho válido foi encontrado. Se não, sai do loop.

Se um melhor vizinho válido foi encontrado, ele adiciona o melhor vizinho à lista tabu. Se a lista tabu exceder o tamanho especificado, remove a entrada mais antiga.

A solução atual e o custo são atualizados com a melhor solução e o custo do melhor vizinho.

Finalmente, se o custo atual for melhor que o melhor custo, a melhor solução, atribuições de carros e custo são atualizados com a solução atual, atribuições de carros e custo.

O algoritmo continua a iterar até o número especificado de iterações ser alcançado ou um melhor vizinho válido não puder ser encontrado.

No final do algoritmo, a melhor solução, atribuições de carros e custo são retornados. Estes representam a solução encontrada pelo algoritmo Tabu Search.

```
Função: generate_neighbors(route, car_assignments)
    vizinhos = []
    Para i de 1 até num_vizinhos:
```

```
        novo_vizinho = generate_random_neighbor(route, car_assignments)
        Adicionar novo_vizinho à lista de vizinhos
    Retornar vizinhos

Função: tabu_search(num_iterations, tabu_list_size, initial_route, initial_car_assignments, cars)
    melhor_rota = rota_atual = initial_route
    melhor_atribuicoes = atribuicoes_atuais = initial_car_assignments
    melhor_custo = custo_atual = calculate_total_distance(rota_atual, atribuicoes_atuais, cars)
    lista_tabu = []

    Para iteração de 1 até num_iterations:
        vizinhos = generate_neighbors(rota_atual, atribuicoes_atuais)

        melhor_vizinho = None
        melhor_custo_vizinho = infinito

        Para cada vizinho em vizinhos:
            custo_vizinho = calculate_total_distance(vizinho.rota, vizinho.atribuicoes, cars)
            Se custo_vizinho < melhor_custo_vizinho e vizinho não está em lista_tabu:
                melhor_vizinho = vizinho
                melhor_custo_vizinho = custo_vizinho

        Se melhor_vizinho for None:
            Interromper o loop

        Adicionar melhor_vizinho à lista_tabu
        Se o tamanho de lista_tabu exceder tabu_list_size:
            Remover o elemento mais antigo de lista_tabu

        rota_atual = melhor_vizinho.rota
        atribuicoes_atuais = melhor_vizinho.atribuicoes
```

```
        custo_atual = melhor_custo_vizinho

        Se custo_atual < melhor_custo:
            melhor_rota = rota_atual
            melhor_atribuicoes = atribuicoes_atuais
            melhor_custo = custo_atual

    Retornar melhor_rota, melhor_atribuicoes, melhor_custo
```

7.4. Algoritmo Genético e Evolutivo

7.4.1. 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
```

7.4.2. 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
```

7.4.3. 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
```

7.4.4. Diferença entre os aalgoritmos

A diferença entre o algoritmos genetico e evolutivo é que o evolutivo é iniciado com uma mutation_rate=0.9 e no genetico o mutation rate é 0.1.

8. Resultados

9. 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.

9.1. 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
```

10. Geração de Resultados

A seguir, descrevemos o processo de geração de resultados para o nosso experimento utilizando o código Python abaixo.

10.1. Configuração do Experimento

Para explorar diferentes cenários, realizamos uma varredura nos parâmetros, considerando diferentes números de cidades (`citys`) e números de carros (`cars_number`). O código foi executado para valores de `citys` variando de 4 a 9 e `cars_number` variando de 2 a 4.

10.2. Execução do Algoritmo

O algoritmo foi executado três vezes para cada combinação de parâmetros para garantir resultados robustos. Durante cada execução, um arquivo de instância de carros (`teste.car`) foi gerado para a configuração específica. Em seguida, o algoritmo de *brute-force* para o Problema do Caixeiro Viajante com Carros foi aplicado à instância gerada.

A seguir está um trecho do código Python usado para executar o experimento:

Listing 1: Execução do Experimento

```
for citys in range(3,15):
    for cars_number in range(2,5):
        for i in range(0  ,3):
            generate_cars_instance_file(citys,cars_number, "teste.car")

            file_path = "teste.car"
            cars_instance, cars = read_cars_instance(file_path)

            start_time1 = time.time()
            best_route, best_distance, best_car_assignments =
            brute_force_tsp_with_cars(cars_instance, cars)
            end_time1 = time.time()

            #get_data

            start_time2 = time.time()
            best_solution, best_distance =
            genetic_algorithm(num_generations=100, pop_size=500,
            mutation_rate=0.9, num_cities=cars_instance.dimension,
            num_cars=cars_instance.cars_number, cars=cars)
            end_time2 = time.time()

            #get_data


            start_time3 = time.time()
            best_solution, best_distance =
            genetic_algorithm(num_generations=100, pop_size=500,
            mutation_rate=0.1, num_cities=cars_instance.dimension,
            num_cars=cars_instance.cars_number, cars=cars)
            end_time3 = time.time()
```

```
#get_data



start_time4 = time.time()
best_route, best_car_assignments, best_cost =
tabu_search(cars_instance, cars, num_iterations= 100,
tabu_list_size= 10)
end_time4 = time.time()

#get_data



start_time5 = time.time()
best_route, best_car_assignments, best_cost =
simulated_annealing(cars_instance, cars, initial_temperature=
100.0, cooling_rate= 0.03)
end_time5 = time.time()

#get_data
```

10.3. Tabela de Resultados

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
brute_force	3 - 2	0	5.412101745605469e-05	99
genetic_algorithm	3 - 2	0	1.3291656970977783	186
evolucionary_algorithm	3 - 2	0	1.3020470142364502	186
tabu_search	3 - 2	0	3.695487976074219e-05	222
simulated_annealing	3 - 2	0	8.487701416015625e-05	162
brute_force	3 - 2	1	2.7894973754882812e-05	102

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
genetic_algorithm	3 - 2	1	1.359969139099121	221
evolucionary_algorithm	3 - 2	1	2.261741876602173	221
tabu_search	3 - 2	1	4.601478576660156e-05	288
simulated_annealing	3 - 2	1	9.322166442871094e-05	172
brute_force	3 - 2	2	2.4080276489257812e-05	160
genetic_algorithm	3 - 2	2	1.3442959785461426	197
evolucionary_algorithm	3 - 2	2	1.1794869899749756	197
tabu_search	3 - 2	2	3.5762786865234375e-05	321
simulated_annealing	3 - 2	2	7.128715515136719e-05	220
brute_force	3 - 3	0	4.601478576660156e-05	112
genetic_algorithm	3 - 3	0	1.2812762260437012	188
evolucionary_algorithm	3 - 3	0	1.146139144897461	188
tabu_search	3 - 3	0	3.5762786865234375e-05	254
simulated_annealing	3 - 3	0	6.914138793945312e-05	137
brute_force	3 - 3	1	3.886222839355469e-05	139
genetic_algorithm	3 - 3	1	1.3963558673858643	162
evolucionary_algorithm	3 - 3	1	1.4214756488800049	162
tabu_search	3 - 3	1	5.698204040527344e-05	313
simulated_annealing	3 - 3	1	8.606910705566406e-05	237
brute_force	3 - 3	2	4.7206878662109375e-05	126
genetic_algorithm	3 - 3	2	1.4493107795715332	141
evolucionary_algorithm	3 - 3	2	1.3794348239898682	141
tabu_search	3 - 3	2	6.29425048828125e-05	258
simulated_annealing	3 - 3	2	0.00016999244689941406	159
brute_force	3 - 4	0	8.511543273925781e-05	156
genetic_algorithm	3 - 4	0	1.3464219570159912	156

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
evolucionary_algorithm	3 - 4	0	1.1907191276550293	156
tabu_search	3 - 4	0	3.886222839355469e-05	221
simulated_annealing	3 - 4	0	8.606910705566406e-05	341
brute_force	3 - 4	1	8.606910705566406e-05	136
genetic_algorithm	3 - 4	1	2.1262059211730957	165
evolucionary_algorithm	3 - 4	1	1.2283000946044922	165
tabu_search	3 - 4	1	3.1948089599609375e-05	278
simulated_annealing	3 - 4	1	6.604194641113281e-05	222
brute_force	3 - 4	2	7.605552673339844e-05	108
genetic_algorithm	3 - 4	2	1.219635009765625	223
evolucionary_algorithm	3 - 4	2	1.2238948345184326	223
tabu_search	3 - 4	2	3.409385681152344e-05	161
simulated_annealing	3 - 4	2	7.200241088867188e-05	234
brute_force	4 - 2	0	7.009506225585938e-05	149
genetic_algorithm	4 - 2	0	1.3045709133148193	149
evolucionary_algorithm	4 - 2	0	1.2670071125030518	149
tabu_search	4 - 2	0	4.601478576660156e-05	431
simulated_annealing	4 - 2	0	0.00011801719665527344	305
brute_force	4 - 2	1	8.320808410644531e-05	183
genetic_algorithm	4 - 2	1	1.3613030910491943	183
evolucionary_algorithm	4 - 2	1	1.3086907863616943	183
tabu_search	4 - 2	1	7.700920104980469e-05	243
simulated_annealing	4 - 2	1	0.00012302398681640625	300
brute_force	4 - 2	2	7.700920104980469e-05	131
genetic_algorithm	4 - 2	2	1.791175127029419	279
evolucionary_algorithm	4 - 2	2	1.6275041103363037	279

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
tabu_search	4 - 2	2	4.9114227294921875e-05	353
simulated_annealing	4 - 2	2	0.00012302398681640625	327
brute_force	4 - 3	0	0.00032806396484375	100
genetic_algorithm	4 - 3	0	1.3135488033294678	100
evolucionary_algorithm	4 - 3	0	1.3024680614471436	100
tabu_search	4 - 3	0	4.792213439941406e-05	385
simulated_annealing	4 - 3	0	0.00015997886657714844	348
brute_force	4 - 3	1	0.0002849102020263672	60
genetic_algorithm	4 - 3	1	1.3998408317565918	60
evolucionary_algorithm	4 - 3	1	1.2522621154785156	60
tabu_search	4 - 3	1	4.291534423828125e-05	147
simulated_annealing	4 - 3	1	0.00011706352233886719	218
brute_force	4 - 3	2	0.00030803680419921875	138
genetic_algorithm	4 - 3	2	1.228579044342041	239
evolucionary_algorithm	4 - 3	2	1.2351150512695312	239
tabu_search	4 - 3	2	4.100799560546875e-05	416
simulated_annealing	4 - 3	2	0.00010395050048828125	338
brute_force	4 - 4	0	0.0006902217864990234	81
genetic_algorithm	4 - 4	0	1.2472829818725586	131
evolucionary_algorithm	4 - 4	0	1.1823251247406006	131
tabu_search	4 - 4	0	3.910064697265625e-05	255
simulated_annealing	4 - 4	0	0.0001049041748046875	395
brute_force	4 - 4	1	0.0007469654083251953	126
genetic_algorithm	4 - 4	1	1.3500657081604004	160
evolucionary_algorithm	4 - 4	1	1.2631330490112305	160
tabu_search	4 - 4	1	4.1961669921875e-05	320

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	4 - 4	1	0.00010395050048828125	332
brute_force	4 - 4	2	0.0007810592651367188	161
genetic_algorithm	4 - 4	2	1.1916170120239258	208
evolucionary_algorithm	4 - 4	2	1.1966791152954102	205
tabu_search	4 - 4	2	3.814697265625e-05	327
simulated_annealing	4 - 4	2	0.00010514259338378906	510
brute_force	5 - 2	0	0.00048089027404785156	152
genetic_algorithm	5 - 2	0	1.237684965133667	180
evolucionary_algorithm	5 - 2	0	1.1754088401794434	180
tabu_search	5 - 2	0	9.202957153320312e-05	345
simulated_annealing	5 - 2	0	0.00026488304138183594	333
brute_force	5 - 2	1	0.0005178451538085938	227
genetic_algorithm	5 - 2	1	1.2312159538269043	237
evolucionary_algorithm	5 - 2	1	1.2149271965026855	245
tabu_search	5 - 2	1	0.00010704994201660156	404
simulated_annealing	5 - 2	1	0.0003230571746826172	377
brute_force	5 - 2	2	0.0005428791046142578	142
genetic_algorithm	5 - 2	2	1.204615831375122	178
evolucionary_algorithm	5 - 2	2	1.1899051666259766	178
tabu_search	5 - 2	2	9.107589721679688e-05	348
simulated_annealing	5 - 2	2	0.0002880096435546875	342
brute_force	5 - 3	0	0.0031921863555908203	146
genetic_algorithm	5 - 3	0	1.236175775527954	176
evolucionary_algorithm	5 - 3	0	1.173262119293213	217
tabu_search	5 - 3	0	9.298324584960938e-05	368

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	5 - 3	0	0.00027108192443847656	357
brute_force	5 - 3	1	0.0028328895568847656	84
genetic_algorithm	5 - 3	1	1.2395341396331787	194
evolucionary_algorithm	5 - 3	1	1.2212908267974854	194
tabu_search	5 - 3	1	0.00010609626770019531	310
simulated_annealing	5 - 3	1	0.0003249645233154297	406
brute_force	5 - 3	2	0.0028939247131347656	104
genetic_algorithm	5 - 3	2	1.2386937141418457	263
evolucionary_algorithm	5 - 3	2	1.225696086883545	256
tabu_search	5 - 3	2	9.799003601074219e-05	274
simulated_annealing	5 - 3	2	0.0002999305725097656	359
brute_force	5 - 4	0	0.012367963790893555	95
genetic_algorithm	5 - 4	0	1.2841770648956299	116
evolucionary_algorithm	5 - 4	0	1.203874111175537	141
tabu_search	5 - 4	0	0.0001049041748046875	426
simulated_annealing	5 - 4	0	0.0003190040588378906	453
brute_force	5 - 4	1	0.014150142669677734	137
genetic_algorithm	5 - 4	1	1.2563540935516357	151
evolucionary_algorithm	5 - 4	1	1.2019259929656982	168
tabu_search	5 - 4	1	9.703636169433594e-05	460
simulated_annealing	5 - 4	1	0.00027823448181152344	329
brute_force	5 - 4	2	0.013682126998901367	158
genetic_algorithm	5 - 4	2	1.2282240390777588	253
evolucionary_algorithm	5 - 4	2	1.2135989665985107	271
tabu_search	5 - 4	2	9.012222290039062e-05	378

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	5 - 4	2	0.0002796649932861328	395
brute_force	6 - 2	0	0.004912853240966797	214
genetic_algorithm	6 - 2	0	1.247527837753296	279
evolucionary_algorithm	6 - 2	0	1.1897180080413818	281
tabu_search	6 - 2	0	0.00016689300537109375	446
simulated_annealing	6 - 2	0	0.00041794776916503906	412
brute_force	6 - 2	1	0.0048520565032958984	137
genetic_algorithm	6 - 2	1	1.2620849609375	189
evolucionary_algorithm	6 - 2	1	1.3030638694763184	164
tabu_search	6 - 2	1	0.00016999244689941406	387
simulated_annealing	6 - 2	1	0.00035190582275390625	198
brute_force	6 - 2	2	0.004622936248779297	120
genetic_algorithm	6 - 2	2	1.2149529457092285	192
evolucionary_algorithm	6 - 2	2	1.2335178852081299	213
tabu_search	6 - 2	2	0.00025272369384765625	373
simulated_annealing	6 - 2	2	0.0003898143768310547	479
brute_force	6 - 3	0	0.0524749755859375	127
genetic_algorithm	6 - 3	0	1.314605951309204	203
evolucionary_algorithm	6 - 3	0	1.1907997131347656	183
tabu_search	6 - 3	0	0.0002200603485107422	351
simulated_annealing	6 - 3	0	0.0004298686981201172	338
brute_force	6 - 3	1	0.04930615425109863	135
genetic_algorithm	6 - 3	1	1.2439260482788086	202
evolucionary_algorithm	6 - 3	1	1.2249510288238525	173
tabu_search	6 - 3	1	0.0002429485321044922	311

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	6 - 3	1	0.00039505958557128906	428
brute_force	6 - 3	2	0.056941986083984375	213
genetic_algorithm	6 - 3	2	1.241912841796875	226
evolucionary_algorithm	6 - 3	2	1.247114896774292	245
tabu_search	6 - 3	2	0.00018930435180664062	492
simulated_annealing	6 - 3	2	0.00033736228942871094	388
brute_force	6 - 4	0	0.21653223037719727	60
genetic_algorithm	6 - 4	0	1.2753710746765137	125
evolucionary_algorithm	6 - 4	0	1.1949892044067383	146
tabu_search	6 - 4	0	0.00021576881408691406	451
simulated_annealing	6 - 4	0	0.0003287792205810547	403
brute_force	6 - 4	1	0.26843905448913574	148
genetic_algorithm	6 - 4	1	1.2800321578979492	308
evolucionary_algorithm	6 - 4	1	1.232832908630371	253
tabu_search	6 - 4	1	0.00022792816162109375	448
simulated_annealing	6 - 4	1	0.0003590583801269531	510
brute_force	6 - 4	2	0.25243592262268066	128
genetic_algorithm	6 - 4	2	1.2590582370758057	164
evolucionary_algorithm	6 - 4	2	1.2450487613677979	207
tabu_search	6 - 4	2	0.00022339820861816406	378
simulated_annealing	6 - 4	2	0.0003483295440673828	351
brute_force	7 - 2	0	0.06450104713439941	226
genetic_algorithm	7 - 2	0	1.2595741748809814	257
evolucionary_algorithm	7 - 2	0	1.1848289966583252	266
tabu_search	7 - 2	0	0.0044498443603515625	540

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	7 - 2	0	0.0005292892456054688	371
brute_force	7 - 2	1	0.05397486686706543	132
genetic_algorithm	7 - 2	1	1.2765982151031494	233
evolucionary_algorithm	7 - 2	1	1.238044261932373	195
tabu_search	7 - 2	1	0.0009670257568359375	328
simulated_annealing	7 - 2	1	0.00046181678771972656	329
brute_force	7 - 2	2	0.06332063674926758	209
genetic_algorithm	7 - 2	2	1.239548921585083	242
evolucionary_algorithm	7 - 2	2	1.2437303066253662	256
tabu_search	7 - 2	2	0.0009179115295410156	498
simulated_annealing	7 - 2	2	0.00048279762268066406	455
brute_force	7 - 3	0	1.0563719272613525	239
genetic_algorithm	7 - 3	0	1.2794311046600342	329
evolucionary_algorithm	7 - 3	0	1.2080271244049072	288
tabu_search	7 - 3	0	0.004442930221557617	627
simulated_annealing	7 - 3	0	0.00047779083251953125	476
brute_force	7 - 3	1	0.9363620281219482	189
genetic_algorithm	7 - 3	1	1.3509540557861328	288
evolucionary_algorithm	7 - 3	1	1.2707631587982178	283
tabu_search	7 - 3	1	0.004734992980957031	370
simulated_annealing	7 - 3	1	0.0008540153503417969	425
brute_force	7 - 3	2	0.9031639099121094	158
genetic_algorithm	7 - 3	2	1.2685198783874512	262
evolucionary_algorithm	7 - 3	2	1.2746450901031494	235
tabu_search	7 - 3	2	0.0011470317840576172	486

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	7 - 3	2	0.0005612373352050781	559
brute_force	7 - 4	0	5.6363818645477295	76
genetic_algorithm	7 - 4	0	1.2711381912231445	277
evolucionary_algorithm	7 - 4	0	1.2257647514343262	318
tabu_search	7 - 4	0	0.003943920135498047	282
simulated_annealing	7 - 4	0	0.0004730224609375	375
brute_force	7 - 4	1	6.089732885360718	133
genetic_algorithm	7 - 4	1	1.2753081321716309	311
evolucionary_algorithm	7 - 4	1	1.2583999633789062	315
tabu_search	7 - 4	1	0.0006170272827148438	551
simulated_annealing	7 - 4	1	0.0004749298095703125	424
brute_force	7 - 4	2	6.24392294883728	136
genetic_algorithm	7 - 4	2	1.2218718528747559	265
evolucionary_algorithm	7 - 4	2	1.3442471027374268	282
tabu_search	7 - 4	2	0.003902912139892578	417
simulated_annealing	7 - 4	2	0.00049591064453125	463
brute_force	8 - 2	0	0.7374131679534912	130
genetic_algorithm	8 - 2	0	1.271299123764038	376
evolucionary_algorithm	8 - 2	0	1.200143814086914	380
tabu_search	8 - 2	0	0.004458904266357422	382
simulated_annealing	8 - 2	0	0.0005357265472412109	390
brute_force	8 - 2	1	0.7859938144683838	176
genetic_algorithm	8 - 2	1	1.2569029331207275	312
evolucionary_algorithm	8 - 2	1	1.270751953125	250
tabu_search	8 - 2	1	0.00579380989074707	491

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	8 - 2	1	0.0009632110595703125	468
brute_force	8 - 2	2	0.7240760326385498	143
genetic_algorithm	8 - 2	2	1.2298710346221924	332
evolucionary_algorithm	8 - 2	2	1.246183156967163	364
tabu_search	8 - 2	2	0.0045146942138671875	412
simulated_annealing	8 - 2	2	0.0005931854248046875	505
brute_force	8 - 3	0	17.8694109916687	119
genetic_algorithm	8 - 3	0	1.2603631019592285	356
evolucionary_algorithm	8 - 3	0	1.3086628913879395	312
tabu_search	8 - 3	0	0.0060269832611083984	504
simulated_annealing	8 - 3	0	0.0006458759307861328	457
brute_force	8 - 3	1	19.42444896697998	167
genetic_algorithm	8 - 3	1	1.2108674049377441	304
evolucionary_algorithm	8 - 3	1	1.2044520378112793	333
tabu_search	8 - 3	1	0.00474095344543457	480
simulated_annealing	8 - 3	1	0.0005488395690917969	372
brute_force	8 - 3	2	19.941742181777954	189
genetic_algorithm	8 - 3	2	1.167201042175293	415
evolucionary_algorithm	8 - 3	2	1.1962831020355225	303
tabu_search	8 - 3	2	0.004658937454223633	596
simulated_annealing	8 - 3	2	0.0007991790771484375	475
brute_force	8 - 4	0	174.44684600830078	144
genetic_algorithm	8 - 4	0	1.2159011363983154	365
evolucionary_algorithm	8 - 4	0	1.142552137374878	326
tabu_search	8 - 4	0	0.00462794303894043	539

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	8 - 4	0	0.0005438327789306641	454
brute_force	8 - 4	1	180.8446102142334	167
genetic_algorithm	8 - 4	1	1.1978492736816406	325
evolucionary_algorithm	8 - 4	1	1.2063961029052734	330
tabu_search	8 - 4	1	0.004392147064208984	450
simulated_annealing	8 - 4	1	0.0006537437438964844	589
brute_force	8 - 4	2	178.81642603874207	156
genetic_algorithm	8 - 4	2	1.1855831146240234	413
evolucionary_algorithm	8 - 4	2	1.1913142204284668	330
tabu_search	8 - 4	2	0.004471778869628906	530
simulated_annealing	8 - 4	2	0.0005419254302978516	477
brute_force	9 - 2	0	13.216864109039307	204
genetic_algorithm	9 - 2	0	1.2006120681762695	346
evolucionary_algorithm	9 - 2	0	1.1480262279510498	322
tabu_search	9 - 2	0	0.006228923797607422	476
simulated_annealing	9 - 2	0	0.0007469654083251953	534
brute_force	9 - 2	1	13.926792860031128	231
genetic_algorithm	9 - 2	1	1.2013359069824219	360
evolucionary_algorithm	9 - 2	1	1.2093257904052734	360
tabu_search	9 - 2	1	0.005855083465576172	578
simulated_annealing	9 - 2	1	0.000698089599609375	471
brute_force	9 - 2	2	11.245200872421265	153
genetic_algorithm	9 - 2	2	1.1795811653137207	362
evolucionary_algorithm	9 - 2	2	1.1945128440856934	361
tabu_search	9 - 2	2	0.0068819522857666016	328

Continua na Próxima Página

Table 1 – Continuação da Página Anterior

Algoritmo	**Configuração (cidades -carros)**	**Execução**	**Tempo de Execução (s)**	**Melhor Distancia**
simulated_annealing	9 - 2	2	0.0006978511810302734	416
brute_force	9 - 3	0	493.90284991264343	139
genetic_algorithm	9 - 3	0	1.2399382591247559	454
evolucionary_algorithm	9 - 3	0	1.2220170497894287	423
tabu_search	9 - 3	0	0.006768226623535156	537
simulated_annealing	9 - 3	0	0.0007891654968261719	471
brute_force	9 - 3	1	445.74506282806396	177
genetic_algorithm	9 - 3	1	1.2052030563354492	454
evolucionary_algorithm	9 - 3	1	1.1604909896850586	420
tabu_search	9 - 3	1	0.005966901779174805	549
simulated_annealing	9 - 3	1	0.0007119178771972656	543
brute_force	9 - 3	2	466.5500748157501	190
genetic_algorithm	9 - 3	2	1.2446162700653076	508
evolucionary_algorithm	9 - 3	2	1.285329818725586	515
tabu_search	9 - 3	2	0.006006002426147461	443
simulated_annealing	9 - 3	2	0.0007112026214599609	581
brute_force	9 - 4	0	6374.771116018295	171
genetic_algorithm	9 - 4	0	1.425189733505249	457
evolucionary_algorithm	9 - 4	0	1.4842188358306885	444
tabu_search	9 - 4	0	0.007068157196044922	456
simulated_annealing	9 - 4	0	0.0007290840148925781	567

Table 1: Tempos de execução para diferentes configurações.

10.4. Gráfico de Desempenho (instancias cidades de 3 a 9 e carros de 2 a 4) - para cada configuração houve 3 execuções

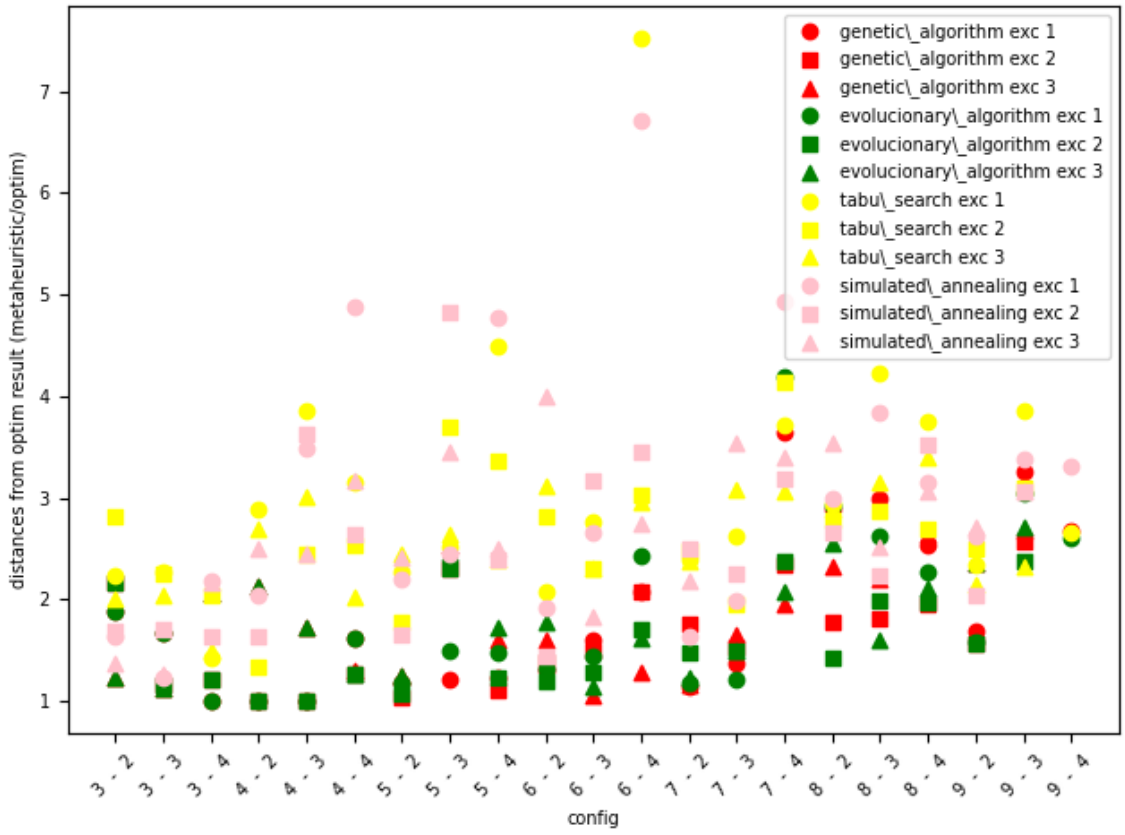

Figure 2: Gráfico de Desempenho dos Algoritmos

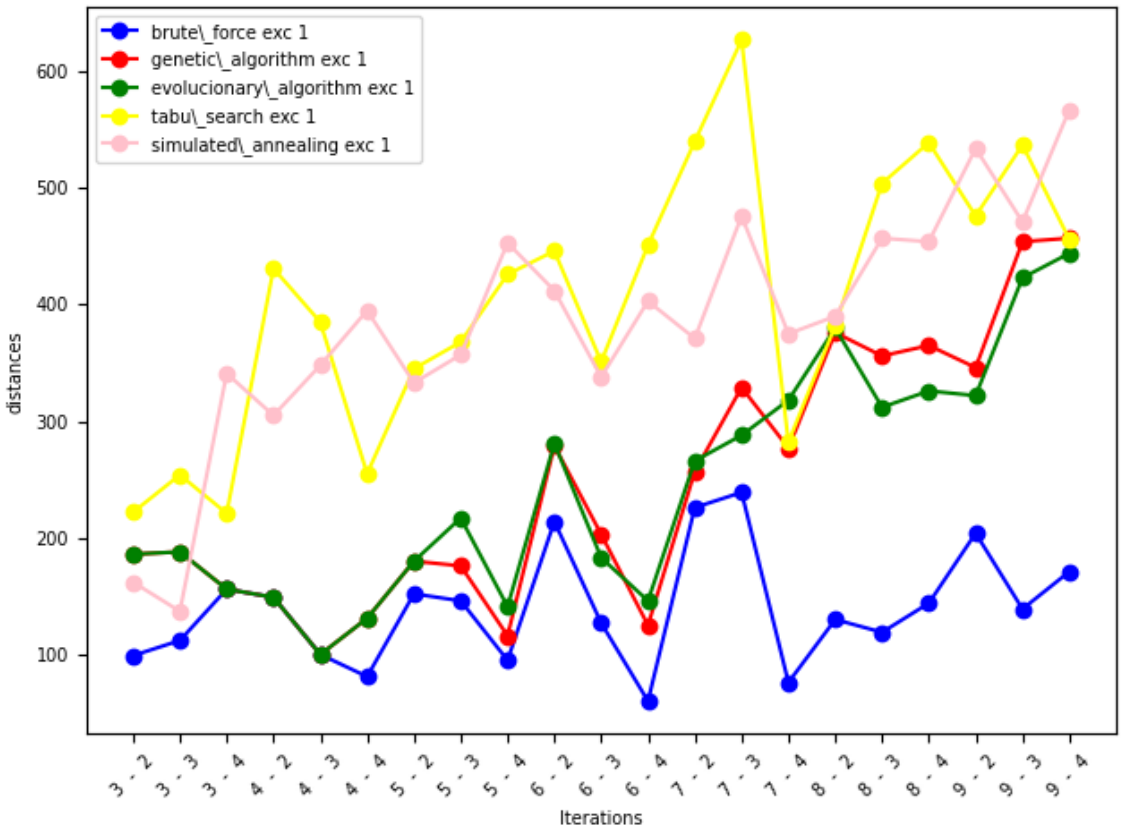

Figure 3: Distancia do resultados metaheuristicos em relação do optimo (brute_force) - execução 1

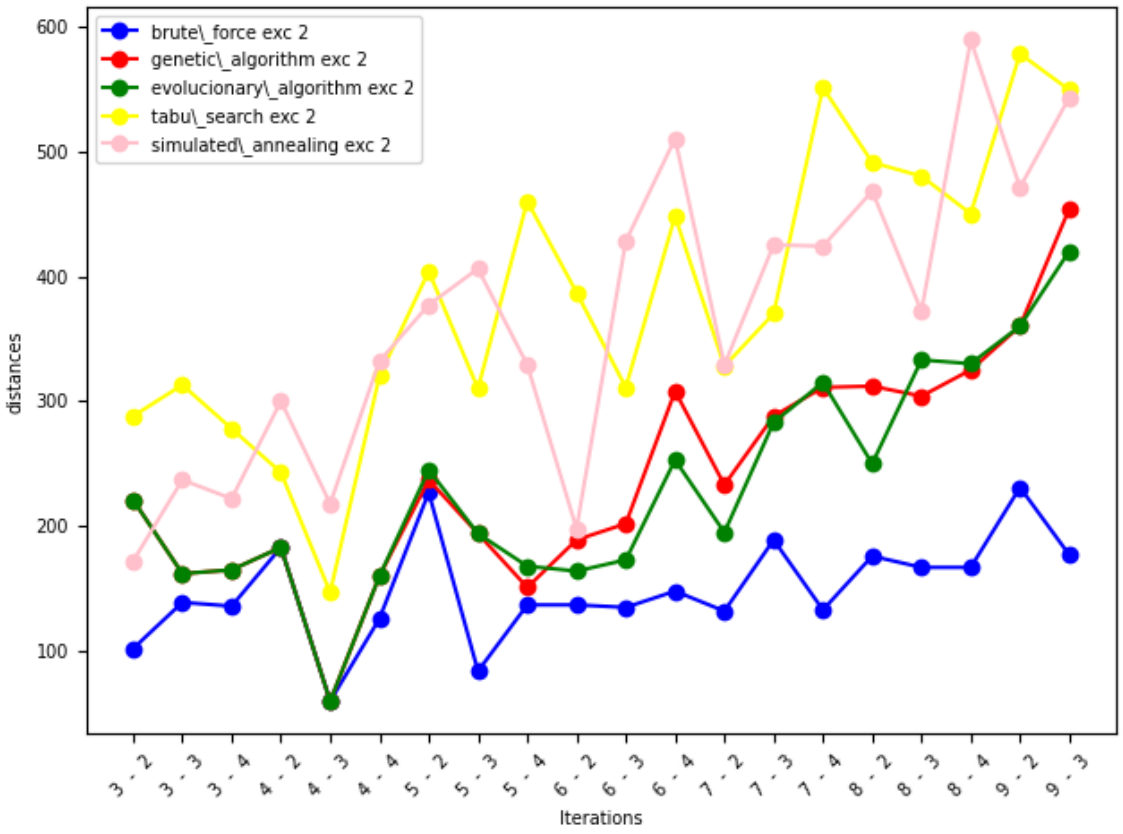

Figure 4: Distancia do resultados metaheuristicos em relação do optimo (brute_force) - execução 2

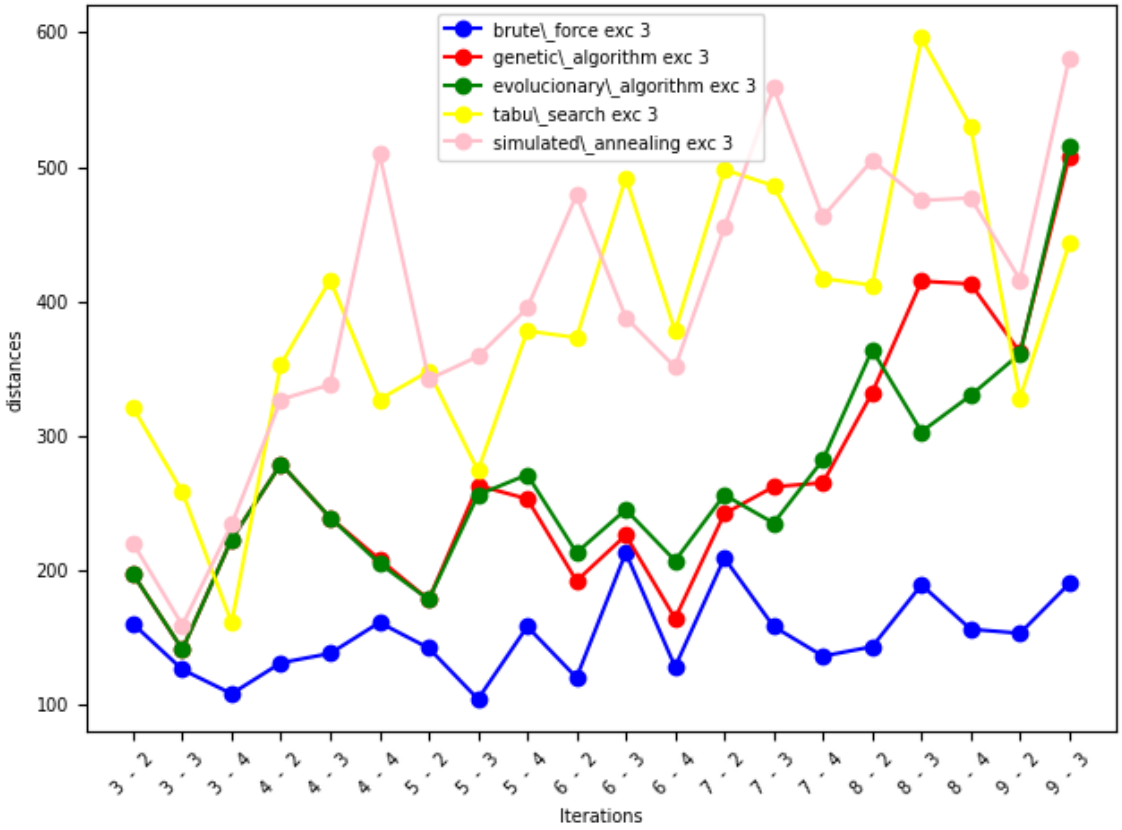

Figure 5: Distancia do resultados metaheuristicos em relação do optimo (brute_force) - execução 3

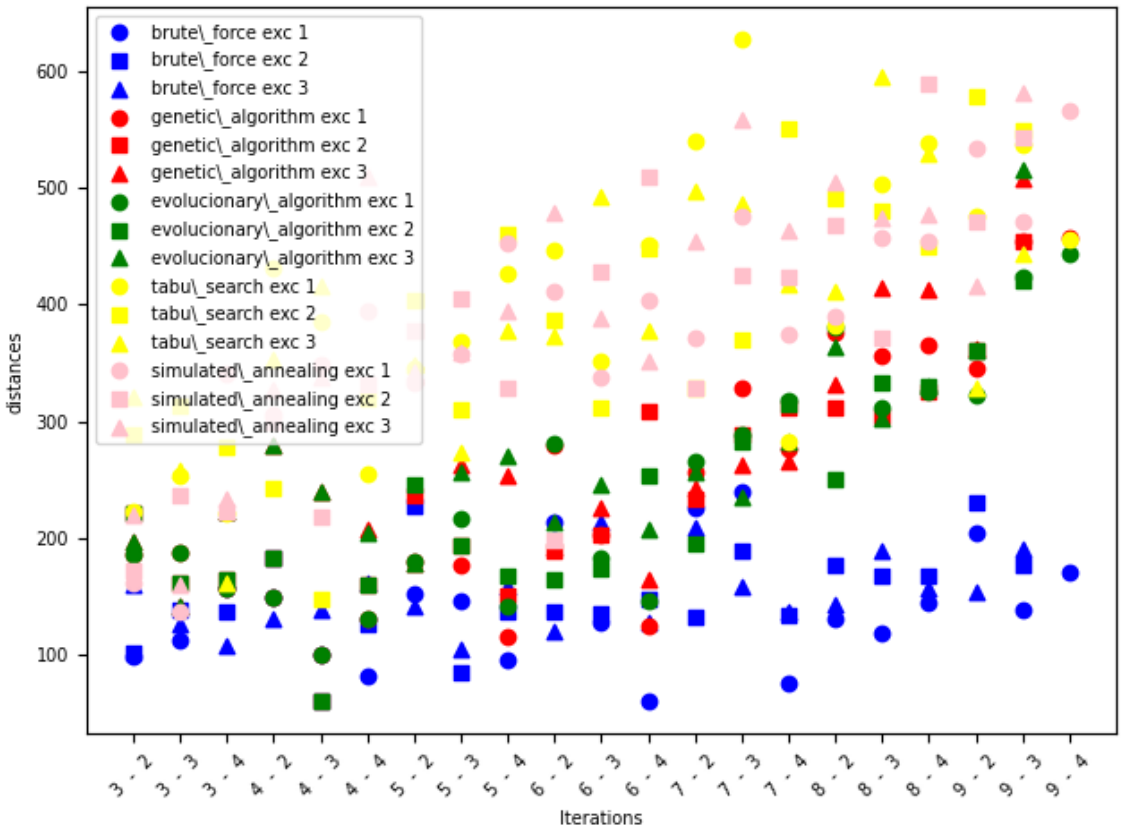

Figure 6: Distancia do resultados metaheuristicos em relação do optimo (brute_force) - todas as execuções

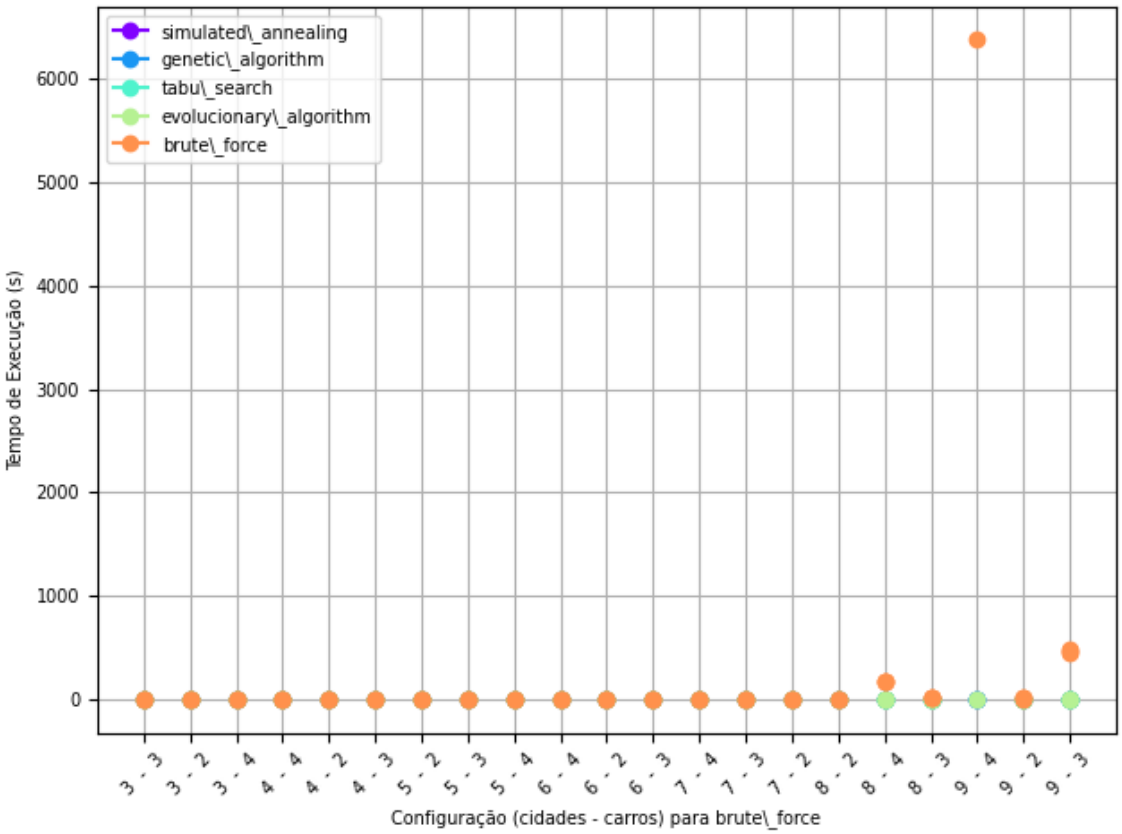

Figure 7: Tempo de execução dos algoritmos

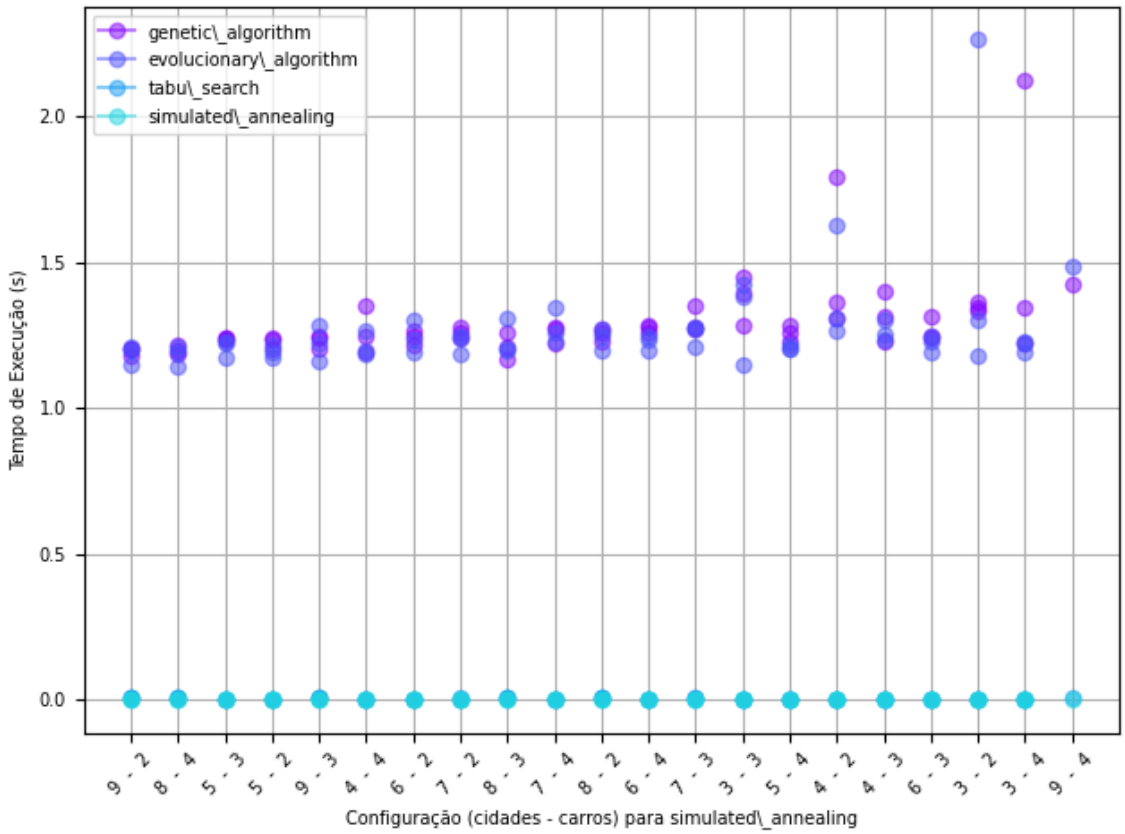

Figure 8: Tempo de execução dos algoritmos heuristicos

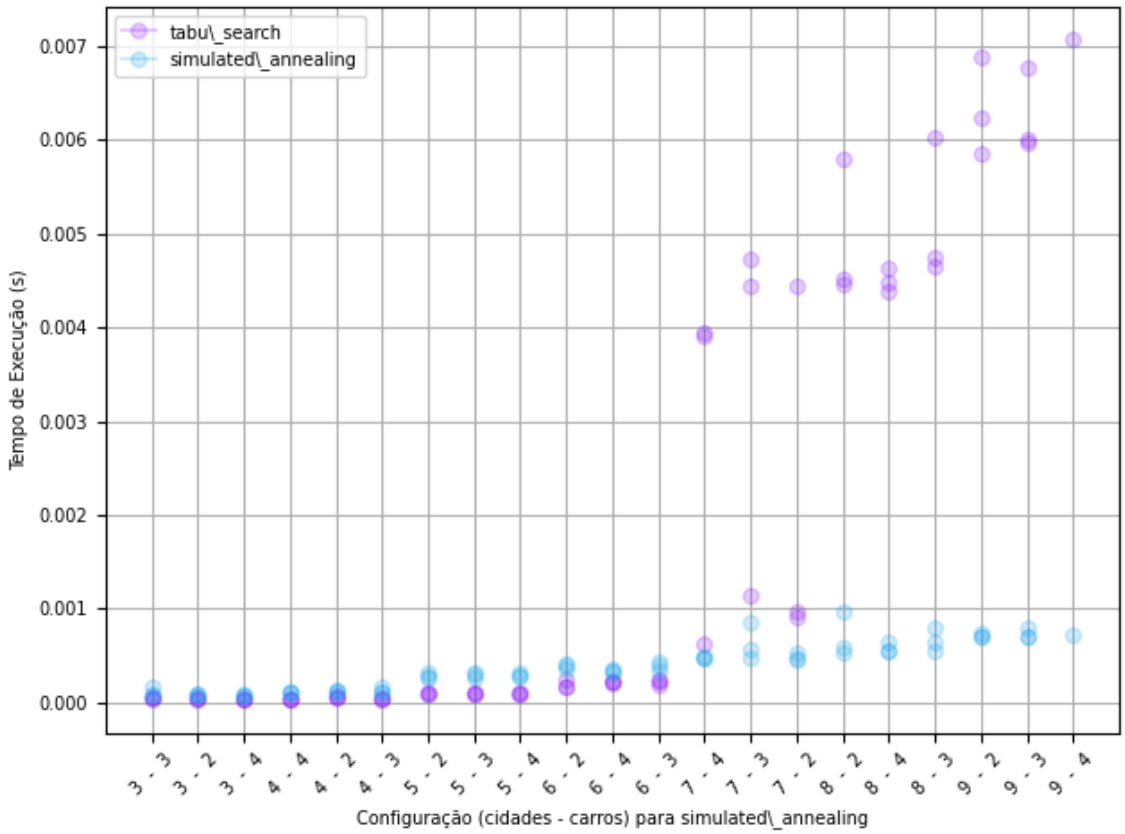

Figure 9: Tempo de execução dos algoritmos - tabu e simulated annealing

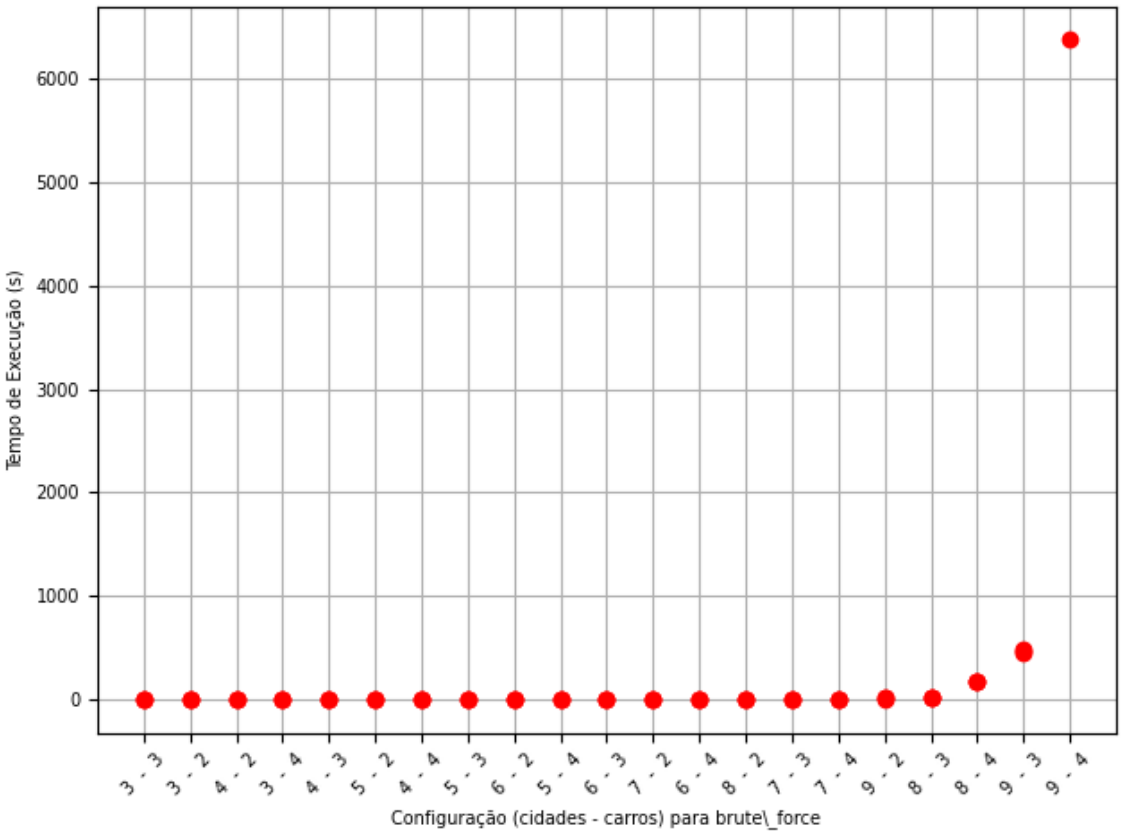

Figure 10: Tempo de execução dos algoritmos - brute force

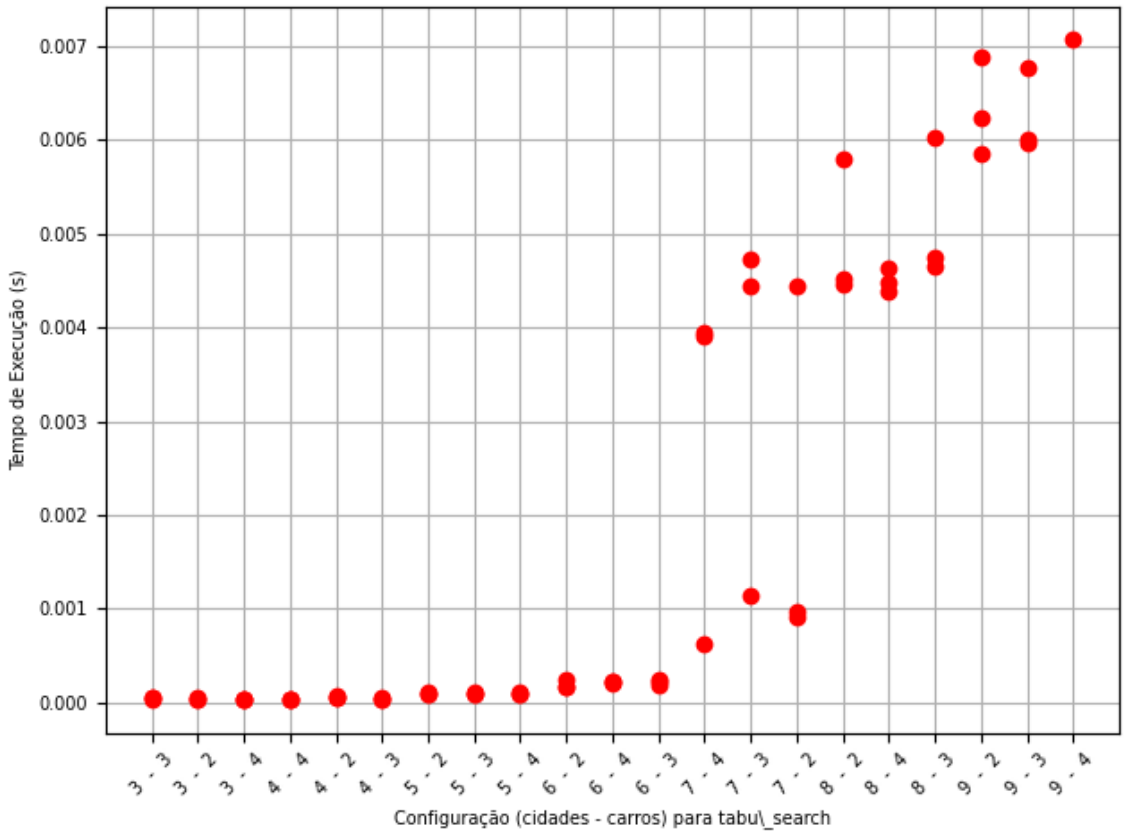

Figure 11: Tempo de execução dos algoritmos - tabu search

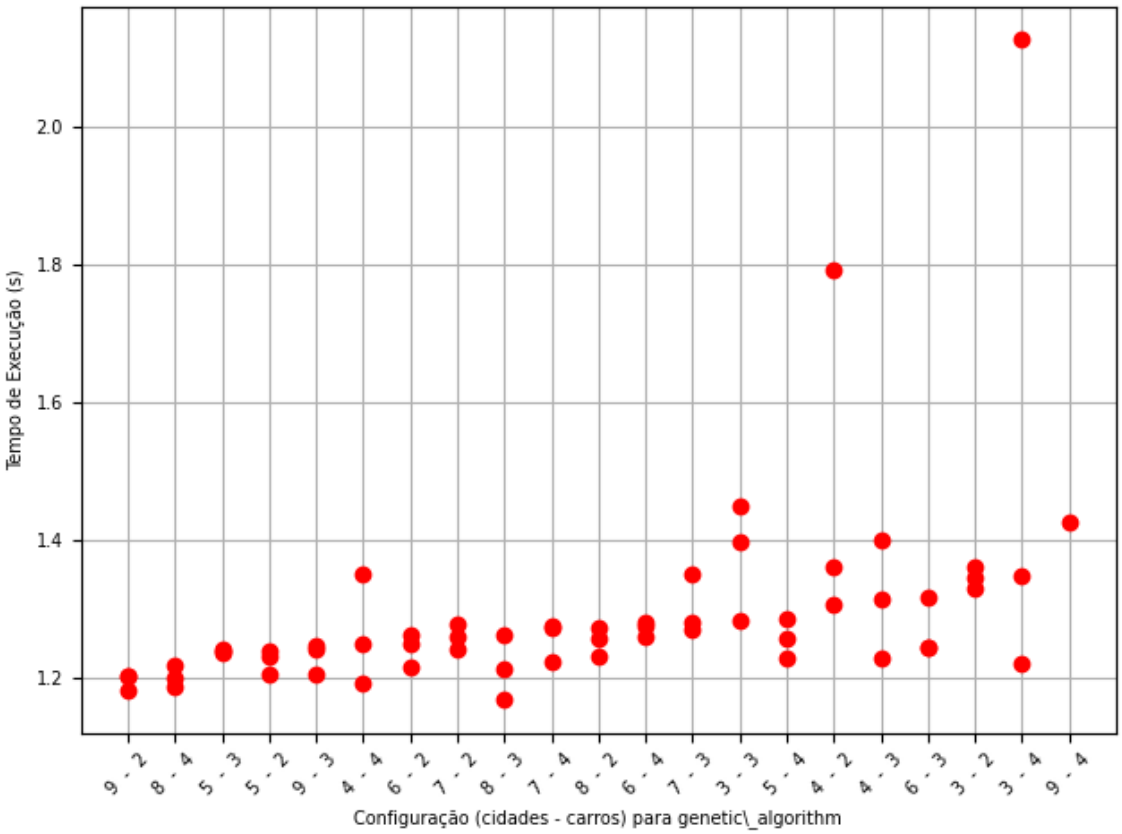

Figure 12: Tempo de execução dos algoritmos - genetic algorithm

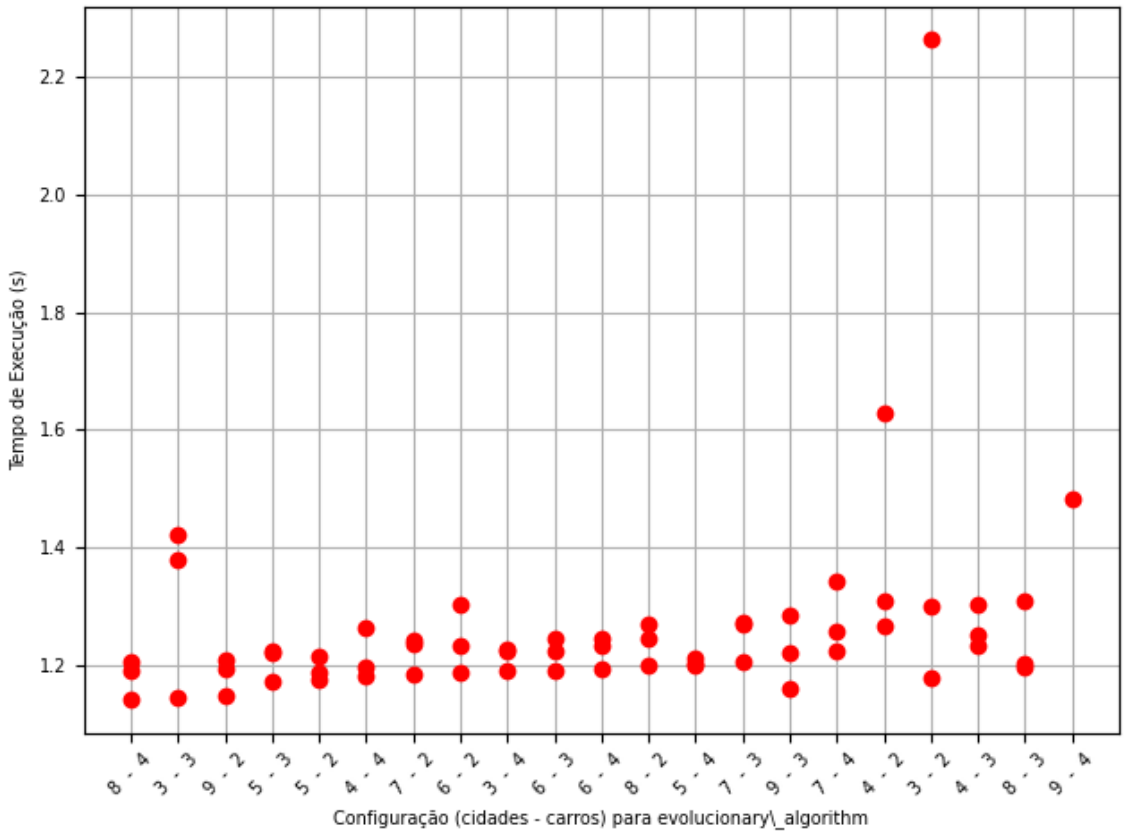

Figure 13: Tempo de execução dos algoritmos - evolucionary algorithm

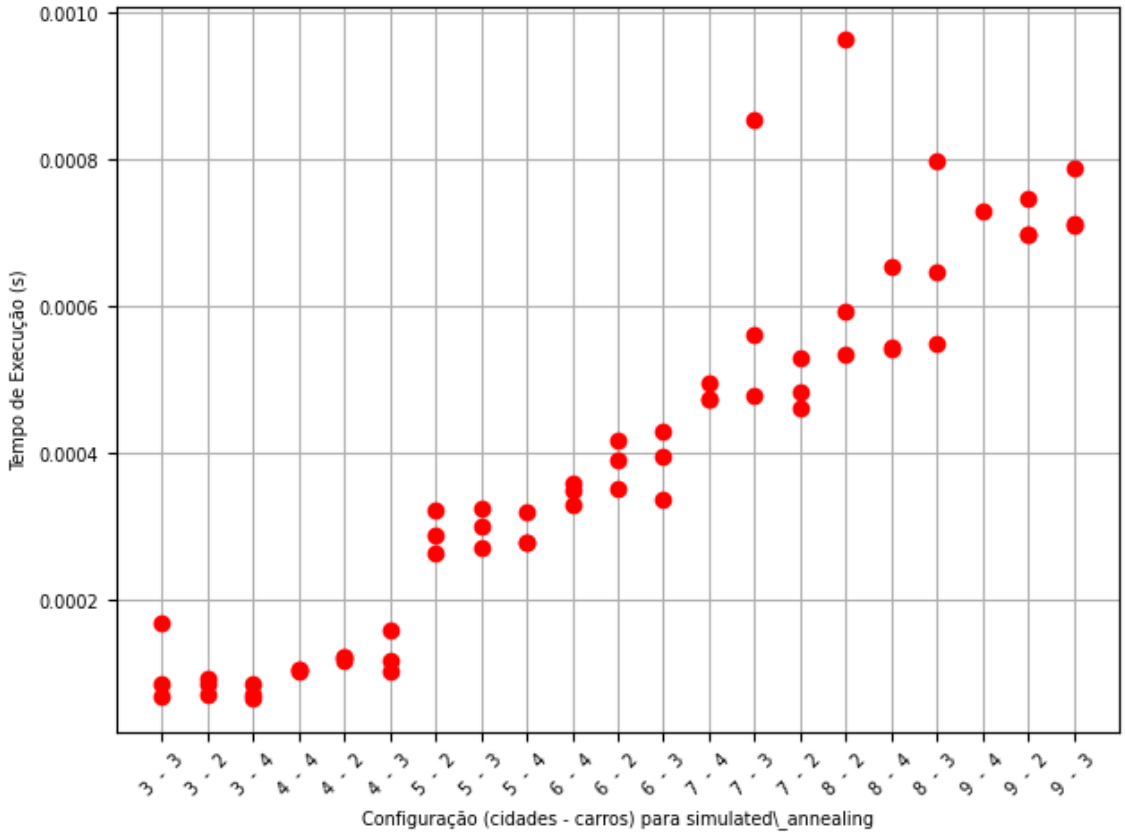

Figure 14: Tempo de execução dos algoritmos - simulated annealing

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, Metaheuristics: 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