



Bacharelado em Engenharia de Computação

Lorran Caetano Machado Lopes

Professor: Michel Fornaciali

Projeto Final - Supercomputação

Vehicle Routing Problem

São Paulo

Junho de 2024



Conteúdos

1 - Implementações.....	3
a. Exaustiva (1_brute_force.cpp).....	3
b. Heurística (2_heuristica.cpp).....	6
c. Implementação com OpenMP (3_openMP.cpp).....	9
d. Implementação com MPI + OpenMP (4_mpi.cpp).....	11
2 - Validação das Implementações	15
3 - Avaliação de Resultados.....	17

1 - Implementações

a. Exaustiva (1_brute_force.cpp)

Esta implementação busca exaustivamente todas as rotas possíveis para encontrar a rota de menor custo, respeitando as restrições de capacidade do veículo e o número máximo de paradas.

Passos da Implementação

1. **Leitura do Grafo:**
 - O grafo é lido de um arquivo `grafo.txt` que contém o número de nós, as demandas de cada nó (exceto o depósito), e as arestas com seus respectivos pesos.
2. **Geração de Permutações:**
 - A função `generate_permutations` gera todas as permutações possíveis dos nós (excluindo o depósito), garantindo que cada permutação começa e termina no depósito.
3. **Ajuste das Permutações:**
 - A função `adjust_permutations` ajusta as permutações geradas, inserindo retornos ao depósito sempre que a capacidade do veículo é excedida ou o número máximo de paradas é alcançado. Isso é feito para garantir que as rotas sejam válidas conforme as restrições impostas.
4. **Busca da Rota de Menor Custo:**
 - A função `find_min_cost_route` percorre todas as permutações ajustadas e calcula o custo de cada uma. A permutação com o menor custo é escolhida como a rota de menor custo.

Detalhamento do Código

1. Leitura do Grafo:

```
ifstream infile("grafo.txt");
if (!infile) {
    cerr << "Não foi possível abrir o arquivo grafo.txt" << endl;
    return 1;
}

int num_nodes;
infile >> num_nodes;

unordered_map<int, int> demands;
for (int i = 1; i < num_nodes; ++i) {
    int node, demand;
    infile >> node >> demand;
    demands[node] = demand;
}

vector<vector<int>> adj_matrix(num_nodes, vector<int>(num_nodes, -1));
int num_edges;
infile >> num_edges;
for (int i = 0; i < num_edges; ++i) {
    int from, to, weight;
    infile >> from >> to >> weight;
    adj_matrix[from][to] = weight;
}

infile.close();
```

2. Geração de Permutações:

```
void generate_permutations(vector<int>& nodes, vector<vector<int>>& permutations) {
    sort(nodes.begin(), nodes.end());
    do {
        vector<int> perm = {0};
        perm.insert(perm.end(), nodes.begin(), nodes.end());
        perm.push_back(0);
        permutations.push_back(perm);
    } while (next_permutation(nodes.begin(), nodes.end()));
}
```

3. Ajuste das Permutações:

```
void adjust_permutations(vector<vector<int>>& permutations, const vector<vector<int>>& adj_matrix, const
unordered_map<int, int>& demands, int vehicle_capacity, int num_stops) {
    for (size_t perm_index = 0; perm_index < permutations.size(); ++perm_index) {
        auto& perm = permutations[perm_index];
        int current_capacity = 0;
        int stops = 0;

        for (auto it = perm.begin() + 1; it != perm.end() - 1; ++it) {
            int from = *(it - 1);
            int to = *it;

            if (adj_matrix[from][to] == -1) {
                it = perm.insert(it, 0);
                current_capacity = 0;
                stops = 0;
                continue;
            }

            if (demands.find(to) != demands.end()) {
                current_capacity += demands.at(to);
                stops++;
            }

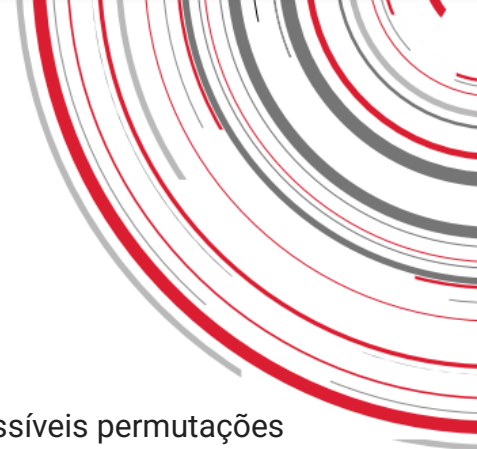
            if (current_capacity > vehicle_capacity || stops > num_stops) {
                if (*(it - 1) != 0) {
                    it = perm.insert(it, 0);
                    current_capacity = demands.at(to);
                    stops = 1;
                }
            }
        }
    }
}
```

4. Busca da Rota de Menor Custo:

```
vector<int> find_min_cost_route(const vector<vector<int>>& permutations, const vector<vector<int>>&
adj_matrix) {
    vector<int> min_cost_route;
    int min_cost = numeric_limits<int>::max();

    for (const auto& perm : permutations) {
        int cost = 0;
        bool valid_route = true;
        for (size_t i = 0; i < perm.size() - 1; ++i) {
            int from = perm[i];
            int to = perm[i + 1];
            if (adj_matrix[from][to] == -1) {
                valid_route = false;
                break;
            }
            cost += adj_matrix[from][to];
        }
        if (valid_route && cost < min_cost) {
            min_cost = cost;
            min_cost_route = perm;
        }
    }

    cout << "Menor custo encontrado: " << min_cost << endl;
    return min_cost_route;
}
```



A implementação faz uma busca exaustiva gerando todas as possíveis permutações dos nós, ajustando essas permutações para respeitar as restrições de capacidade e número de paradas, e finalmente encontrando a permutação com o menor custo. Esta abordagem garante que a solução encontrada seja a ótima (menor custo), mas a um custo computacional elevado, especialmente para grafos maiores, devido ao crescimento exponencial do número de permutações.

b. Heurística (2_heuristica.cpp)

Heurística de Inserção Mais Próxima: começa com uma rota contendo apenas o depósito e, em seguida, iterativamente insere o cliente mais próximo em sua posição de menor custo até que todos os clientes sejam incluídos. Esta abordagem é simples e relativamente rápida, mas pode não encontrar a melhor solução possível.

Passos da Implementação

1. Leitura do Grafo:

- O grafo é lido de um arquivo `grafo.txt` que contém o número de nós, as demandas de cada nó (exceto o depósito), e as arestas com seus respectivos pesos.

2. Função para Encontrar a Melhor Posição de Inserção:

- A função `find_best_insertion` encontra a melhor posição para inserir um novo nó na rota atual, minimizando o aumento do custo da rota.

3. Heurística de Inserção Mais Próxima:

- A função `nearest_insertion` constrói a rota de maneira iterativa, inserindo o nó não visitado mais próximo (em termos de custo de inserção) em cada iteração, e ajustando a rota conforme necessário para respeitar as restrições de capacidade do veículo e número de paradas.

4. Cálculo do Custo da Rota:

- A função `calculate_route_cost` calcula o custo total da rota encontrada.

Detalhamento do Código

1. Função para Encontrar a Melhor Posição de Inserção:

```
pair<int, int> find_best_insertion(const vector<int>& route, int new_node, const vector<vector<int>>& adj_matrix) {
    int min_cost_increase = numeric_limits<int>::max();
    int best_position = -1;

    for (size_t i = 0; i < route.size() - 1; ++i) {
        int from = route[i];
        int to = route[i + 1];

        if (adj_matrix[from][new_node] != -1 && adj_matrix[new_node][to] != -1) {
            int cost_increase = adj_matrix[from][new_node] + adj_matrix[new_node][to] - adj_matrix[from][to];

            if (cost_increase < min_cost_increase) {
                min_cost_increase = cost_increase;
                best_position = i + 1;
            }
        }
    }

    return {best_position, min_cost_increase};
}
```

2. Heurística de Inserção Mais Próxima

```
vector<int> nearest_insertion(const vector<vector<int>>& adj_matrix, int vehicle_capacity, const unordered_map<int, int>& demands, int num_stops) {
    int num_nodes = adj_matrix.size();
    vector<int> route = {0, 0}; // Inicia com o depósito
    vector<bool> visited(num_nodes, false);
    visited[0] = true;
    int current_capacity = 0;
    int current_stops = 0;

    while (true) {
        int best_node = -1;
        int best_position = -1;
        int min_cost_increase = numeric_limits<int>::max();

        for (int i = 1; i < num_nodes; ++i) {
            if (!visited[i]) {
                auto [position, cost_increase] = find_best_insertion(route, i, adj_matrix);
                if (position != -1 && cost_increase < min_cost_increase) {
                    best_node = i;
                    best_position = position;
                    min_cost_increase = cost_increase;
                }
            }
        }

        if (best_node == -1) break; // Todos os nós foram visitados

        route.insert(route.begin() + best_position, best_node);
        visited[best_node] = true;
        current_capacity += demands.at(best_node);
        current_stops++;

        if (current_capacity > vehicle_capacity || current_stops >= num_stops) {
            route.insert(route.begin() + best_position + 1, 0); // Retorna ao depósito
            current_capacity = 0; // Reinicia a capacidade
            current_stops = 0; // Reinicia o número de paradas
        }
    }
}
```

```

// Garante que todos os nós sejam visitados
for (int i = 1; i < num_nodes; ++i) {
    if (!visited[i]) {
        bool inserted = false;
        for (size_t j = 0; j < route.size() - 1; ++j) {
            int from = route[j];
            int to = route[j + 1];
            if (adj_matrix[from][i] != -1 && adj_matrix[i][to] != -1) {
                route.insert(route.begin() + j + 1, i);
                visited[i] = true;
                current_capacity += demands.at(i);
                current_stops++;
                if (current_capacity > vehicle_capacity || current_stops >= num_stops) {
                    route.insert(route.begin() + j + 2, 0); // Retorna ao depósito
                    current_capacity = 0;
                    current_stops = 0;
                }
                inserted = true;
                break;
            }
        }
        if (!inserted) {
            for (size_t j = 0; j < route.size() - 1; ++j) {
                int from = route[j];
                if (adj_matrix[from][i] != -1) {
                    route.insert(route.begin() + j + 1, i);
                    visited[i] = true;
                    route.insert(route.begin() + j + 2, 0); // Retorna ao depósito
                    break;
                }
            }
        }
    }
}

// Remover retornos ao depósito redundantes (caminhos 0 0)
vector<int> clean_route;
clean_route.push_back(0);
for (size_t i = 1; i < route.size(); ++i) {
    if (route[i] != 0 || (route[i] == 0 && route[i - 1] != 0)) {
        clean_route.push_back(route[i]);
    }
}

for (int i = 1; i < num_nodes; ++i) {
    if (!visited[i]) {
        cerr << "Erro: Não foi possível inserir o nó " << i << " na rota." << endl;
    }
}

return clean_route;
}

```


3. Cálculo do Custo da Rota

```
int calculate_route_cost(const vector<int>& route, const vector<vector<int>>& adj_matrix) {
    int cost = 0;
    for (size_t i = 0; i < route.size() - 1; ++i) {
        int from = route[i];
        int to = route[i + 1];
        if (adj_matrix[from][to] == -1) {
            cerr << "Erro: Caminho inválido de " << from << " para " << to << endl;
            return numeric_limits<int>::max(); // Rota inválida
        }
        cost += adj_matrix[from][to];
    }
    return cost;
}
```

A implementação da heurística de inserção mais próxima (Nearest Insertion) constrói a rota iterativamente, inserindo o nó não visitado que resulta no menor aumento de custo na posição mais vantajosa. A heurística verifica e ajusta a rota conforme necessário para garantir que a capacidade do veículo e o número de paradas não sejam excedidos. Embora não garanta a solução ótima, esta abordagem é eficiente em termos de tempo computacional e geralmente fornece soluções razoavelmente boas.

c. Implementação com OpenMP (3_openMP.cpp)

Esta implementação usa OpenMP para paralelizar partes do código, a fim de reduzir o tempo de execução da busca exaustiva (global search) para o problema de roteamento de veículos.

Diferenças Principais em Relação à Implementação Original

1. Uso de OpenMP para Paralelização:

- A principal diferença é a introdução do OpenMP (`#pragma omp`) para paralelizar o ajuste das permutações e a busca da rota de menor custo. Isso permite que múltiplas threads trabalhem simultaneamente, aproveitando os recursos multicore do processador.

2. Ajuste Paralelizado das Permutações:

- Na função `adjust_permutations`, a diretiva `#pragma omp parallel for` permite que o loop sobre as permutações seja executado em paralelo. Cada thread ajusta uma permutação diferente ao mesmo tempo.

3. Busca Paralelizada da Rota de Menor Custo:

- Na função `find_min_cost_route`, duas diretivas OpenMP são usadas:
 - `#pragma omp parallel` cria uma região paralela onde múltiplas threads podem trabalhar simultaneamente.
 - `#pragma omp for nowait` permite que o loop sobre as permutações seja dividido entre as threads sem sincronização no final de cada iteração.
 - `#pragma omp critical` garante que a atualização das variáveis `min_cost` e `min_cost_route` seja feita de maneira segura, evitando condições de corrida.

Detalhamento do Código

1. Geração de Permutações (sem mudanças).

2. Ajuste Paralelizado das Permutações:

```
void adjust_permutations(vector<vector<int>>& permutations, const vector<vector<int>>& adj_matrix, const unordered_map<int, int>& demands, int vehicle_capacity, int num_stops) {  
    #pragma omp parallel for  
    for (size_t perm_index = 0; perm_index < permutations.size(); ++perm_index) {  
        ...  
    }  
}
```

3. Busca Paralelizada da Rota de Menor Custo:

```
vector<int> find_min_cost_route(const vector<vector<int>>& permutations, const vector<vector<int>>& adj_matrix) {
    vector<int> min_cost_route;
    int min_cost = numeric_limits<int>::max();

    #pragma omp parallel
    {
        vector<int> local_min_cost_route;
        int local_min_cost = numeric_limits<int>::max();

        #pragma omp for nowait
        for (size_t i = 0; i < permutations.size(); ++i) {
            const auto& perm = permutations[i];
            int cost = 0;
            bool valid_route = true;
            for (size_t j = 0; j < perm.size() - 1; ++j) {
                int from = perm[j];
                int to = perm[j + 1];
                if (adj_matrix[from][to] == -1) {
                    valid_route = false;
                    break;
                }
                cost += adj_matrix[from][to];
            }
            if (valid_route && cost < local_min_cost) {
                local_min_cost = cost;
                local_min_cost_route = perm;
            }
        }

        #pragma omp critical
        {
            if (local_min_cost < min_cost) {
                min_cost = local_min_cost;
                min_cost_route = local_min_cost_route;
            }
        }
    }

    cout << "Menor custo encontrado: " << min_cost << endl;
    return min_cost_route;
}
```

4. Função Principal (sem mudanças significativas, exceto a inclusão de OpenMP).

d. Implementação com MPI + OpenMP (4_mpi.cpp)

Esta implementação combina MPI (Message Passing Interface) e OpenMP para paralelizar a busca exaustiva (global search) do problema de roteamento de veículos em múltiplas máquinas (ou múltiplos processos), e usa OpenMP para paralelizar a execução local dentro de cada máquina.

Diferenças Principais em Relação às Implementações Anteriores

1. Introdução do MPI:

- MPI é usado para distribuir a carga de trabalho entre múltiplos processos em diferentes máquinas ou núcleos. Isso permite que cada processo trabalhe em uma parte do problema, reduzindo o tempo total de execução.
- A função `MPI_Init` inicializa o ambiente MPI, e `MPI_Finalize` o encerra no final do programa.

2. Divisão do Trabalho entre Processos MPI:

- As permutações geradas são divididas entre os processos MPI. Cada processo MPI trabalha em um subconjunto das permutações.
- A divisão é feita calculando um `chunk_size` que determina o número de permutações para cada processo, e usando o `world_rank` para determinar o intervalo de permutações que cada processo deve processar.

3. Coleta dos Resultados:

- Cada processo MPI calcula a rota de menor custo local e, em seguida, os resultados são combinados para encontrar a rota de menor custo global.
- A função `MPI_Allreduce` é usada para encontrar o menor custo global entre todos os processos.
- `MPI_Bcast` e `MPI_Send/MPI_Recv` são usados para comunicar a rota de menor custo global ao processo principal (rank 0).

4. Uso de OpenMP para Paralelização Local:

- Dentro de cada processo MPI, o OpenMP é usado para paralelizar o ajuste das permutações e a busca da rota de menor custo.
- As diretivas `#pragma omp parallel for` e `#pragma omp parallel` são usadas da mesma forma que na implementação anterior para paralelizar o trabalho dentro de cada processo.

Detalhamento do Código

1. Geração de Permutações (sem mudanças).

2. Ajuste Paralelizado das Permutações (mesmo que na implementação com OpenMP).

3. Busca Paralelizada da Rota de Menor Custo (alteração para suportar MPI):

```
vector<int> find_min_cost_route(const vector<vector<int>>& permutations, const vector<vector<int>>&
adj_matrix, int& local_min_cost) {
    vector<int> local_min_cost_route;
    local_min_cost = numeric_limits<int>::max();

    #pragma omp parallel
    {
        vector<int> thread_min_cost_route;
        int thread_min_cost = numeric_limits<int>::max();

        #pragma omp for nowait
        for (size_t i = 0; i < permutations.size(); ++i) {
            const auto& perm = permutations[i];
            int cost = 0;
            bool valid_route = true;
            for (size_t j = 0; j < perm.size() - 1; ++j) {
                int from = perm[j];
                int to = perm[j + 1];
                if (adj_matrix[from][to] == -1) {
                    valid_route = false;
                    break;
                }
                cost += adj_matrix[from][to];
            }
            if (valid_route && cost < thread_min_cost) {
                thread_min_cost = cost;
                thread_min_cost_route = perm;
            }
        }

        #pragma omp critical
        {
            if (thread_min_cost < local_min_cost) {
                local_min_cost = thread_min_cost;
                local_min_cost_route = thread_min_cost_route;
            }
        }
    }

    return local_min_cost_route;
}
```

4. Função Principal com MPI:

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    auto start = high_resolution_clock::now();

    ifstream infile("grafo.txt");
    if (!infile) {
        cerr << "Cannot open grafo.txt" << endl;
        MPI_Finalize();
        return 1;
    }

    int num_nodes;
    infile >> num_nodes;

    unordered_map<int, int> demands;
    for (int i = 1; i < num_nodes; ++i) {
        int node, demand;
        infile >> node >> demand;
        demands[node] = demand;
    }

    vector<vector<int>> adj_matrix(num_nodes, vector<int>(num_nodes, -1));
    int num_edges;
    infile >> num_edges;
    for (int i = 0; i < num_edges; ++i) {
        int from, to, weight;
        infile >> from >> to >> weight;
        adj_matrix[from][to] = weight;
    }

    infile.close();

    vector<int> nodes;
    for (int i = 1; i < num_nodes; ++i) {
        nodes.push_back(i);
    }

    vector<vector<int>> all_permutations;

    // Generate permutations
    generate_permutations(nodes, all_permutations);

    // Split permutations among MPI processes
    int chunk_size = all_permutations.size() / world_size;
    int start_index = world_rank * chunk_size;
    int end_index = (world_rank == world_size - 1) ? all_permutations.size() : start_index + chunk_size;

    vector<vector<int>> local_permutations(all_permutations.begin() + start_index,
    all_permutations.begin() + end_index);

    int vehicle_capacity = 15;
    int num_stops = 5;

    // Adjust permutations
    adjust_permutations(local_permutations, adj_matrix, demands, vehicle_capacity, num_stops);

    // Find minimum cost route
    int local_min_cost;
    vector<int> local_min_cost_route = find_min_cost_route(local_permutations, adj_matrix,
    local_min_cost);

    // Gather all results
    int global_min_cost;
    MPI_Allreduce(&local_min_cost, &global_min_cost, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    // Determine which process found the global minimum
    int min_cost_rank = (local_min_cost == global_min_cost) ? world_rank : -1;
    int global_min_cost_rank;
    MPI_Allreduce(&min_cost_rank, &global_min_cost_rank, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

    // Communicate the size of the route before sending the actual route
    int local_min_cost_route_size = local_min_cost_route.size();
    MPI_Bcast(&local_min_cost_route_size, 1, MPI_INT, global_min_cost_rank, MPI_COMM_WORLD);

    // Resize global_min_cost_route to the correct size
    vector<int> global_min_cost_route(local_min_cost_route_size, 0);

    // Send the local minimum cost route from the process that found it
    if (world_rank == global_min_cost_rank) {
        MPI_Send(local_min_cost_route.data(), local_min_cost_route_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    // Root process receives the global minimum cost route
    if (world_rank == 0) {
        MPI_Recv(global_min_cost_route.data(), local_min_cost_route_size, MPI_INT, global_min_cost_rank,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        cout << "Menor custo encontrado: " << global_min_cost << endl;
        cout << "Rota de menor custo: ";
        for (int node : global_min_cost_route) {
            cout << node << " ";
        }
        cout << endl;

        auto end = high_resolution_clock::now();
        auto duration = duration_cast<milliseconds>(end - start).count();
        cout << "Tempo total de execução: " << duration << " ms." << endl;
    }

    MPI_Finalize();
    return 0;
}
```



Resumo das Mudanças

- **Paralelização com MPI:**
 - O uso de MPI permite distribuir a carga de trabalho entre múltiplos processos, em diferentes máquinas, para processar subconjuntos das permutações.
 - As funções `MPI_Init`, `MPI_Finalize`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Allreduce`, `MPI_Bcast`, `MPI_Send` e `MPI_Recv` são usadas para inicializar, finalizar e comunicar entre processos MPI.
- **Divisão de Trabalho:**
 - As permutações são divididas entre os processos MPI, e cada processo ajusta suas permutações localmente e busca a rota de menor custo localmente.
- **Coleta de Resultados:**
 - Resultados locais são combinados para encontrar o menor custo global e a rota associada.
 - A comunicação é feita para garantir que o processo principal (rank 0) tenha a rota de menor custo global.
- **Paralelização Local com OpenMP:**
 - Dentro de cada processo MPI, o OpenMP é usado para paralelizar o ajuste das permutações e a busca da rota de menor custo.
 - A implementação de OpenMP é a mesma da versão anterior com OpenMP, mas agora é combinada com a distribuição de trabalho de MPI.

Essa combinação de MPI e OpenMP permite que o algoritmo tire proveito tanto da paralelização entre máquinas quanto da paralelização dentro de cada máquina, oferecendo potencial para reduções significativas no tempo de execução.

2 - Validação das Implementações

Para garantir a correção das implementações dos algoritmos de roteamento de veículos (global search, heuristic, OpenMP e MPI+OpenMP), foi realizado um processo de validação rigoroso utilizando a biblioteca `vrpy`, uma ferramenta amplamente reconhecida para resolver problemas de roteamento de veículos.

Método de Validação

1. Leitura dos Grafos:

- Para cada conjunto de dados (correspondente a diferentes números de nós), foram lidas as demandas dos nós e as arestas do grafo a partir do arquivo `grafo.txt`.

2. Extração de Custos e Rotas:

- De cada arquivo de saída gerado pelas implementações (`.out`), foram extraídos o custo calculado e a rota correspondente. Utilizaram-se expressões regulares para identificar e extrair essas informações do conteúdo dos arquivos.

3. Configuração e Resolução com `vrpy`:

- Construiu-se o grafo usando a biblioteca `networkx` e configurou-se o problema de roteamento de veículos usando o `vrpy`.
- Definiram-se a capacidade do veículo e o número máximo de paradas para corresponder às condições do problema.
- O problema foi resolvido usando o `vrpy` para obter a rota de menor custo e comparar esses resultados com os valores extraídos dos arquivos de saída.

Resultados da Validação

● Comparação dos Resultados:

- Compararam-se os custos calculados pelas implementações com os custos ótimos fornecidos pelo `vrpy`.
- Registraram-se tanto as rotas quanto os custos em um arquivo de texto (`validation_results.txt`) para facilitar a análise e verificação.

● Resultados Consistentes:

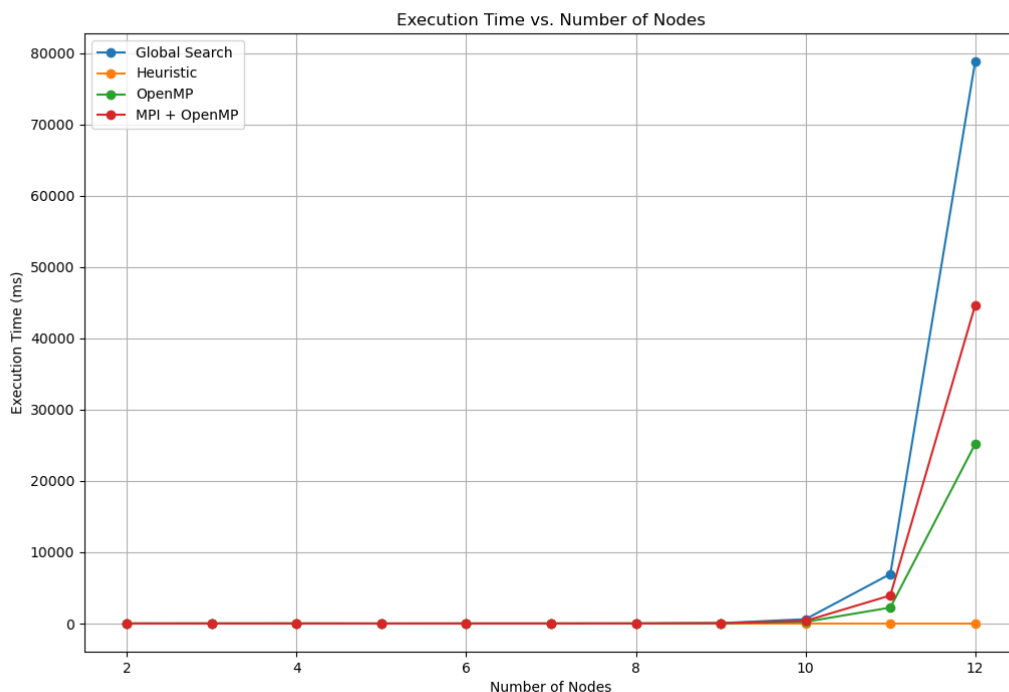
- Em todos os casos, exceto sete, os algoritmos produziram custos que correspondem exatamente aos custos ótimos fornecidos pelo `vrpy`.
- Os sete casos em que não houve correspondência exata foram todos provenientes da implementação heurística. No entanto, mesmo nesses casos, os custos fornecidos pela heurística eram muito próximos dos custos ótimos, indicando que a heurística é eficiente em fornecer soluções de custo similar ao ótimo.

O processo de validação confirma que as implementações estão corretas e produzem resultados precisos para a maioria dos casos. A heurística, embora não sempre exata, fornece custos muito próximos dos ótimos, demonstrando sua eficácia em encontrar soluções rápidas e eficientes. Este processo de validação aumenta a confiança na

precisão e eficiência das implementações, garantindo que elas são adequadas para resolver problemas reais de roteamento de veículos.

Este relatório e os resultados detalhados do processo de validação estão documentados no arquivo [validation_results.txt](#), disponível no diretório do projeto. Os arquivos utilizados estão nas pastas com nomes padrões “NNnos”, onde “NN” são a quantidade de nós usados no grafo de cada teste (02, 03, ..., 12). O código que gera a validação é o [validaCustoRotas.py](#).

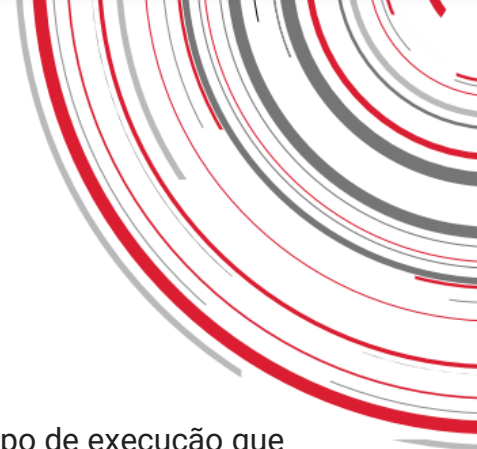
3 - Avaliação de Resultados



Gerado com [analysis.py](#)

Comparações das Abordagens com Tamanhos Diferentes de Grafos

A análise do gráfico mostra o tempo de execução das diferentes abordagens (Global Search, Heuristic, OpenMP, MPI + OpenMP) em função do número de nós no grafo. Aqui destacam-se as observações principais:

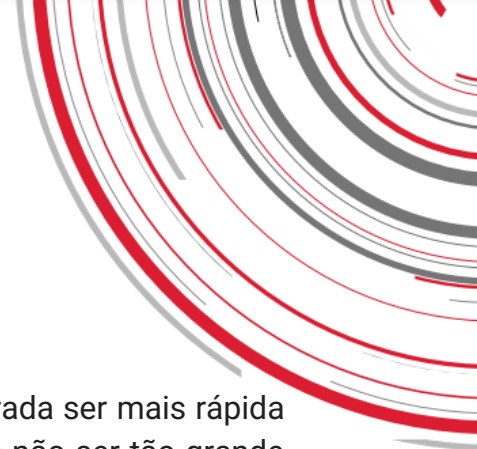
- 
1. **Global Search:** A abordagem de busca global tem um tempo de execução que cresce exponencialmente à medida que o número de nós aumenta. Isso era esperado, pois a busca exaustiva tem uma complexidade computacional muito alta.
 2. **Heuristic (Nearest Insertion):** A heurística utilizada, que é a inserção mais próxima, tem um desempenho significativamente melhor do que a busca global. O tempo de execução cresce muito mais lentamente, o que indica que a heurística é eficiente para tamanhos de grafos moderados. No entanto, a solução encontrada pela heurística nem sempre é a rota mais barata, mas é uma solução aproximada que é computacionalmente mais eficiente.
 3. **OpenMP:** A execução paralela usando OpenMP mostra uma melhoria em relação à busca global, mas não tão significativa quanto a heurística. Isso sugere que o paralelismo pode ajudar, mas não compensa totalmente a complexidade da busca exaustiva.
 4. **MPI + OpenMP:** A combinação de MPI e OpenMP não oferece uma melhoria significativa em relação ao uso apenas de OpenMP. Na verdade, para grafos menores, MPI + OpenMP tem um desempenho pior comparado apenas ao OpenMP. Isso sugere que a sobrecarga de comunicação entre processos MPI pode ser significativa, especialmente para grafos menores.

Clarificação dos Resultados

- O gráfico gerado mostra claramente que a heurística é a abordagem mais eficiente, seguida pelo OpenMP, MPI + OpenMP, e finalmente a busca global.
- O aumento exponencial no tempo de execução da busca global em comparação com as outras abordagens é evidente.
- A heurística mantém um tempo de execução baixo até para grafos com 12 nós, demonstrando sua eficiência.

Justificativas

- **Heurística:** Como esperado, a heurística de inserção mais próxima é mais rápida do que a busca exaustiva, pois evita a necessidade de explorar todas as combinações possíveis. Este resultado é consistente com a teoria de que heurísticas bem escolhidas podem fornecer soluções boas em tempo reduzido, especialmente para problemas de otimização combinatória. No entanto, a solução encontrada pela heurística nem sempre é exata, mas é uma solução aproximada que pode não ser a rota mais barata.

- 
- **OpenMP e MPI + OpenMP:** A execução paralela era esperada ser mais rápida que a execução sequencial. No entanto, a diferença pode não ser tão grande quanto esperado devido à sobrecarga de gerenciar threads e comunicação entre processos MPI. Além disso, para grafos pequenos, o tempo de execução pode ser dominado pela sobrecarga de configuração do paralelismo em vez do cálculo real.
 - **Diferenças do Esperado:**
 - A implementação de MPI + OpenMP mostrou que a sobrecarga de comunicação entre processos MPI pode ser significativa, especialmente para grafos menores. Isso pode explicar por que o tempo de execução não melhora tanto quanto o esperado em comparação com apenas OpenMP.
 - A busca global, mesmo paralelizada, não consegue competir com a heurística devido à sua complexidade intrínseca. A paralelização ajuda, mas não é suficiente para superar a eficiência da heurística.
 - A heurística não apenas fornece soluções mais rapidamente, mas também escala melhor com o tamanho do grafo, o que é uma característica desejável em muitos problemas práticos.

Conclusão

A análise confirma que, embora a paralelização ajude a melhorar os tempos de execução, as heurísticas bem escolhidas como a inserção mais próxima são essenciais para lidar eficientemente com problemas de otimização combinatória como o roteamento de veículos. As abordagens paralelas (OpenMP e MPI + OpenMP) oferecem benefícios adicionais, mas devem ser utilizadas em combinação com heurísticas para obter os melhores resultados. Note-se que a solução encontrada pela heurística é aproximada e pode não ser a rota mais barata, mas é computacionalmente mais eficiente.