



Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Relatório da 3ª Unidade - Grafos

Alexandre Dantas, Andriel Vinicius, Gabriel Carvalho, Maria Paz e
Vinicius de Lima

Professor: Matheus Menezes

11 de dezembro de 2025

Sumário

Lista de Figuras	ii
Lista de Tabelas	iii
Lista de Códigos	iv
1 Introdução	1
1.1 Organização do relatório	1
2 Heurísticas e Buscas Locais	2
3 Algoritmo Genético e Memético	11
3.1 Discussão sobre as decisões	11
3.1.1 Algoritmo memético	13
3.1.2 Resultados	14
4 Conclusão	16
Referências Bibliográficas	17
Appendices	18
A Atividades desenvolvidas por cada integrante	18

Lista de Figuras

Lista de Tabelas

2.1	Execução das heurísticas	10
-----	------------------------------------	----

Lista de Códigos

Capítulo 1

Introdução

O presente trabalho tem por objetivo descrever a implementação realizada para a avaliação prática da 3ª Unidade de DIM0549 Grafos. A avaliação mencionada consiste em implementar computacionalmente as heurísticas do Vizinho Mais Próximo e Inserção Mais Próxima juntamente de Buscas Locais e as metaheurísticas do Algoritmo Genético e Memético. Cada heurística composta com busca local deve ser executada 1 vez em cada um dos 12 problemas e cada metaheurística deve ser executada 20 vezes em cada problema. Cada resultado dessas instâncias deve, então, ser computado e comparado com os resultados do trabalho fornecido pelo professor.

1.1 Organização do relatório

No capítulo 02, iremos comentar acerca das heurísticas desenvolvidas e os resultados da aplicação dos problemas a elas.

No capítulo 03, iremos comentar acerca do Algoritmo Genético e Memético, as decisões tomadas neles e os resultados.

No capítulo 04, refletiremos sobre a implementação feita e traremos conclusões e possíveis pontos de melhoria.

No apêndice A, é possível conferir a lista completa das atividades desenvolvidas por cada integrante.

Capítulo 2

Heurísticas e Buscas Locais

Heurísticas

As heurísticas são algoritmos úteis para encontrar soluções boas em problemas de larga escala, cujos algoritmos tradicionais levariam muito tempo. Neste trabalho, focamos nossa implementação em duas heurísticas para o Problema do Caixeiro Viajante: Nearest Neighbour e Nearest Insertion.

Para representar as soluções encontradas em cada heurística e nas buscas locais, criamos a seguinte estrutura:

```
1 #[derive(PartialEq, Clone)]
2 pub struct Solution<const N: usize> {
3     pub route: Vec<usize>,
4     pub cost: f64,
5 }
```

Essa estrutura possui dois campos: um vetor de naturais que representa a rota formada pelos vértices e um custo associado a essa rota, que é um número de tipo flutuante.

Uma observação é em relação à este tipo genérico N: ele não representa o tipo do nó, pois em nossa implementação escolhemos de forma arbitrária um tipo não-genérico para representar os vértices, o `usize`. Em vez disso, ele é usado para representar de forma dinâmica e durante a etapa de compilação o tamanho do grafo, graças à macro `graph_from_csv` cujas particularidades serão abordadas na seção sobre o algoritmo genético/memético.

Nearest Neighbour (Vizinho Mais Próximo)

A heurística do Nearest Neighbour consiste em um algoritmo guloso para a construção de uma rota otimizada, cujo critério de seleção do próximo nó consiste em escolher a aresta de menor custo até o próximo vizinho, sem se importar com as consequências futuras.

A fase inicial do algoritmo consiste na inicialização de um vetor que armazena todos os vértices já visitados e um vetor para a rota final, como mostrado abaixo. O vetor de visitados difere do vetor de caminho na sua inicialização: como iremos verificar constantemente quais vizinhos já foram visitados, uma operação como `.contains()` tem complexidade $O(n)$, então iniciar um vetor com todas as posições já demarcadas e depois visitar com um simples `visited[i]` tem complexidade $O(1)$, sendo um pouco mais eficiente. Outra otimização foi o uso da variável `visited_count`, que evita precisarmos percorrer todo o vetor de visitados para contar.

```
1 // Armazena para cada nó a informação se ele já foi visitado
```

```

2   let mut visited: Vec<bool> = vec![false; graph.len()];
3   // Guarda o caminho formado ao longo da execução do algoritmo.
4   let mut path: Vec<usize> = Vec::new();
5   path.push(start);
6   // Armazena quantos nós foram visitados.
7   let mut visited_count = 1;
8   // Variáveis auxiliares durante o decorrer das iterações.
9   let mut cost: f64 = 0.0;
10  let mut current_node = start;
11  visited[current_node] = true;
12  let mut better_cost;
13  let mut next_on_path: Option<usize> = None;

```

Em seguida, iniciamos o loop principal, que executará enquanto o caminho não tiver sido completamente preenchido. Neste loop, selecionamos, dentre todos os vizinhos do nó inicial, o vizinho mais próximo (com menor custo) com base no melhor custo calculado até o momento e adicionamos à rota, repetindo o processo até que toda a rota tenha sido preenchida. Quando a rota está completa, calculamos o custo do nó final ao inicial e retornamos uma solução composta pela rota e seu custo total.

```

1   // Loop de construção da rota
2   while visited_count != graph.len() {
3       better_cost = f64::INFINITY;
4       // Loop que busca o vizinho mais próximo (de menor custo) do nó atual.
5       for (i, val) in visited.iter().enumerate() {
6           if *val || current_node == i {
7               continue;
8           }
9           if graph[current_node][i] < better_cost {
10              better_cost = graph[current_node][i];
11              next_on_path = Some(i);
12          }
13      }
14      // Selecionado o vizinho mais próximo, adiciona na rota, adiciona no
15      ↪ custo total, la
16      // marca como visitado e define ele como o nó atual, executando
17      ↪ novamente o loop.
18      if let Some(next_on_path) = next_on_path {
19          let n = next_on_path;
20          path.push(n);
21          cost += better_cost;
22          current_node = n;
23          visited[n] = true;
24          visited_count += 1;
25      }
26  }
27  // Calculamos no custo total a aresta que vai do nó final ao inicial,
28  // formando um ciclo Hamiltoniano
29  cost += graph[current_node][start];
30  Solution { route: path, cost }

```

Note que não adicionamos na rota o nó inicial novamente, pois já sabemos que essa rota é um Ciclo Hamiltoniano, sendo desnecessária essa adição.

Nearest Insertion (Inserção Mais Próxima)

A heurística do Nearest Insertion consiste em um algoritmo que busca selecionar o próximo nó da rota a partir de todos os vértices que já estão na rota, encontrando o nó de menor custo. Além disso, ao inserir é levada em consideração também a posição na rota, buscando sempre inserir onde minimizará o custo total dela.

A fase inicial do algoritmo consiste na inicialização de um vetor para marcar se um vértice já está na rota, um vetor que marca a distância até cada nó e outras variáveis auxiliares. Seguimos uma lógica análoga na criação das variáveis ao Nearest Neighbour, buscando uma certa eficiência.

```

1      let n = graph.len();
2      let mut in_cycle = vec![false; n];
3      let mut min_dist = vec![f64::INFINITY; n];
4      in_cycle[start] = true; // Marca o primeiro nó como pertencente ao ciclo.
5      let mut first: Option<usize> = None;
6      let mut best = f64::INFINITY;
7      // Escolhe o melhor nó no grafo para ser inserido, de acordo com o menor
8      // custo
9      // do inicial até ele.
10     for (i, row) in graph.iter().enumerate().take(n) {
11         if i != start && row[start] < best {
12             best = row[start];
13             first = Some(i);
14         }
15     }
16     let first = first.expect("Invalid graph: no vertex found to start the
17     // Cria um ciclo Hamiltoniano com o inicial e o melhor nó
18     // Calcula a distância até os demais vértices que ainda não estão na rota
19     for v in 0..n {
20         if !in_cycle[v] {
21             min_dist[v] = graph[v][start].min(graph[v][first]);
22         }
23     }
24 
```

Na seção seguinte, é iniciado o loop principal para selecionar e adicionar os nós na rota enquanto ela não estiver completa. Neste loop, faremos a escolha do menor nó alcançável a partir de qualquer um dos vértices que já está no ciclo. Após isso, faremos a escolha de qual posição na rota atual será utilizada para inserir o nó que foi encontrado anteriormente. Tal procedimento é feito calculando para todos os pares de vértices qual local terá a menor distância de u até r e r até v.

```

1      let mut count_in_cycle = 2;
2      while count_in_cycle < n {
3          let mut r_star: Option<usize> = None;
4          let mut best_dist = f64::INFINITY;
5          // Seleciona o menor vértice alcançável a partir do ciclo.
6          for v in 0..n {
7              if !in_cycle[v] && min_dist[v] < best_dist {
8                  best_dist = min_dist[v];
9                  r_star = Some(v);
10             }
11         }
12     }

```

```

11     }
12     let r_star = r_star.expect("No candidate vertex found");
13     let mut best_extra = f64::INFINITY;
14     let mut best_pos = 0;
15     // Para todos os nós do ciclo, calcula a posição que deverá ser
16     ↪ inserido o
17     // vértice, baseado no menor impacto possível no custo
18     for i in 0..cycle.len() - 1 {
19         let u = cycle[i];
20         let v = cycle[i + 1];
21         let du = graph[u][r_star];
22         let dv = graph[r_star][v];
23         let uv = graph[u][v];
24         let extra = du + dv - uv;
25         if extra < best_extra {
26             best_extra = extra;
27             best_pos = i;
28         }
29     }
30     // Insere o menor vértice na melhor posição encontrada e o marca como
31     // pertencente ao ciclo.
32     cycle.insert(best_pos + 1, r_star);
33     in_cycle[r_star] = true;
34     count_in_cycle += 1;
35     // Atualiza as distâncias conhecidas.
36     for v in 0..n {
37         if !in_cycle[v] {
38             let d = graph[v][r_star];
39             if d < min_dist[v] {
40                 min_dist[v] = d;
41             }
42         }
43     }

```

Por fim, calculamos o custo da rota que foi gerada e retornamos a solução. Note que removemos o vértice ao fim do ciclo pois é justamente o vértice inicial, uma informação redundante pois, como comentado anteriormente, sabemos que a rota gerada é um Ciclo Hamiltoniano.

```

1     cycle.pop();
2     let cost = Solution::calculate_cost(&cycle, graph);
3     Solution { route: cycle, cost }

```

Buscas Locais

Heurísticas de melhoramento local são métodos de otimização que buscam aprimorar uma solução inicial através de pequenas modificações iterativas. Essas técnicas são amplamente utilizadas em problemas combinatórios, como o Problema do Caixeiro Viajante (PCV), onde o objetivo é encontrar a rota mais curta que visita um conjunto de cidades exatamente uma vez e retorna à cidade de origem.

A ideia principal das buscas locais é iterar através de soluções candidatas, movendo-se para soluções vizinhas que apresentam uma melhoria em relação à solução atual. Esse processo

continua até que nenhuma melhoria adicional possa ser encontrada. As etapas gerais de uma busca local incluem:

- **Definição da Solução Inicial:** Uma solução inicial é gerada, geralmente por meio de uma heurística construtiva.
- **Geração de Vizinhança:** A vizinhança da solução atual é definida, o que envolve identificar todas as soluções que podem ser alcançadas através de pequenas modificações.
- **Avaliação das Soluções Vizinhas:** Cada solução na vizinhança é avaliada com base em um critério de qualidade (fitness).
- **Movimento para a Melhor Solução Vizinhança:** Se uma solução vizinha melhor for encontrada, a busca se move para essa solução e o processo se repete.
- **Critério de Parada:** A busca termina quando nenhuma melhoria adicional pode ser encontrada ou quando um critério de parada predefinido é atingido (como um número máximo de iterações).

Neste trabalho, implementamos quatro heurísticas de melhoramento local: Swap, Shift, 2-opt e Or-opt. Cada uma dessas heurísticas opera de maneira distinta para explorar o espaço de soluções e melhorar a qualidade da solução inicial gerada pelas heurísticas construtivas (Vizinho Mais Próximo e Inserção Mais Próxima). A implementação define um trait genérico LocalSearch, que abstrai as operações comuns entre as diferentes heurísticas de busca local.

O trait LocalSearch define a interface das buscas locais. Cada função dentro do trait recebe a solução atual, o grafo do problema e possíveis parâmetros adicionais, retornando uma nova solução melhorada.

```
1 pub trait LocalSearch<Graph> {  
2     fn swap(&self, graph: &Graph, start: usize) -> Self;  
3     fn two_opt(&self, graph: &Graph) -> Self;  
4     fn shift(&self, graph: &Graph, start: usize) -> Self;  
5     fn or_opt(&self, graph: &Graph) -> Self;  
6 }
```

Tais funções acima podem ser classificadas de duas formas: **Best Improvement**, aquelas que otimizam ao máximo o custo do cenário inicial que lhe foi dado, ou **First Improvement**, que opera somente sobre a primeira oportunidade de otimização de custo encontrada. Ambas tem seus pontos fortes, esta tem um ganho inferior, mas em contrapartida sua complexidade é muito baixa, já essa gera otimizações melhores por conta de sua complexidade elevada.

Mas todas pertencem à categoria de **Busca Local** por um motivo, elas operam de maneira praticamente idêntica, divergindo apenas na forma em que enxerga e opera sobre sua vizinhança. Dito isso, foram implementadas as heurísticas Swap, Shift e Or-opt da categoria **Best Improvement** e apenas a 2-opt da classe **First Improvement**. Sendo assim, dada a imensa semelhança entre a implementação das buscas, será dado enfoque na forma no qual estas enxergam sua vizinhança.

Shift

Tal abordagem considera como vizinhança apenas os vizinhos diretos, ignorando os que não fazem fronteira com o nó atual da iteração. Ela escolhe um nó a ser removido e reconecta seus vizinhos, e após isso o insere em uma vizinhança que proporcione um melhor custo para a rota. Tudo isso acontece através da função `neighbourhood_by_shift`.

```

1 fn neighbourhood_by_shift(&self, graph: &[[f64; N]; N], start: usize) ->
  ↳ Vec<Self> {
2   let mut solutions: Vec<Solution<N>> = Vec::new();
3   let n = self.route.len();
4
5   if start >= n {
6     return solutions;
7   }
8
9   for target_pos in 0..n {
10    if target_pos == start {
11      continue;
12    }
13
14    let mut new_route = self.route.clone(); // copia a rota atual
15    let elem = new_route.remove(start);    // remove o nó da sua
      ↳ vizinhança
16    new_route.insert(target_pos, elem);    // adiciona ele em outra
17
18    let cost = Self::calculate_cost(&new_route, graph);
19
20    solutions.push(Self {
21      route: new_route,
22      cost,
23    });
24  }
25
26  solutions
27 }

```

Swap

Consiste em gerar uma vizinhança a partir da solução atual por meio da troca entre a cidade localizada em uma posição fixa da rota e cada uma das demais posições possíveis. Para cada troca realizada, uma solução candidata é produzida e avaliada com base no seu custo total. O método percorre todas as soluções vizinhas geradas pela função `neighbourhood_by_swap` e seleciona aquela que apresenta o menor custo. Caso esta solução seja melhor do que a solução corrente, a rota é atualizada e o processo é repetido, caracterizando um procedimento iterativo de melhoria contínua. A busca termina quando nenhuma troca adicional resulta em redução do custo, configurando um processo com estratégia de *best improvement*, no qual a melhor solução vizinha é sempre escolhida como próximo passo de exploração.

```

1 fn neighbourhood_by_swap(&self, graph: &[[f64; N]; N], start: usize) ->
  ↳ Vec<Self> {
2   let mut solutions: Vec<Solution<N>> = Vec::new();
3   let n = self.route.len();
4   if start >= n { return solutions; }
5
6   // para cada posição diferente de `start`, troca as posições e calcula
      ↳ custo
7   for i in 0..n {
8     if i == start { continue; }
9     let mut new_route = self.route.clone();
10    new_route.swap(start, i); // troca os índices start e i

```

```

11         let cost = Self::calculate_cost(&new_route, graph);
12         solutions.push(Self { route: new_route, cost });
13     }
14
15     solutions
16 }
17

```

Or-opt

Tem como objetivo melhorar a solução atual por meio do deslocamento de subsequências contíguas da rota, normalmente de tamanho 1, 2 ou 3, para outras posições do percurso. Essa operação permite explorar movimentos mais amplos do que simples trocas ou deslocamentos unitários, oferecendo maior flexibilidade para escapar de ótimos locais superficiais. A função `neighbourhood_by_or_opt` gera todas as soluções vizinhas possíveis ao remover um bloco da rota e reinseri-lo em outra posição válida. Cada vizinho é avaliado com base no custo total, e a melhor solução encontrada é comparada com a solução atual. Caso represente uma melhoria, a solução corrente é atualizada e o processo é repetido. Assim como em *swap* e *shift*, essa heurística opera segundo a estratégia de *best improvement*, avaliando todas as alternativas da vizinhança antes de escolher a melhor. A busca finaliza quando nenhuma realocação adicional de subsequências resulta em redução do custo, atingindo um ótimo local sob essa vizinhança.

```

1 fn neighbourhood_by_or_opt(&self, graph: &[[f64; N]; N]) -> Vec<Self> {
2     let n = self.route.len();
3     let mut neighbours = Vec::new();
4
5     for seq_len in 1..=3.min(n) {
6         for i in 0..n {
7             if i + seq_len > n {
8                 break;
9             }
10
11             for target in 0..n {
12                 if target >= i && target < i + seq_len {
13                     continue;
14                 }
15
16                 let mut new_route = self.route.clone();
17                 let sequence: Vec<usize> = new_route.drain(i..(i +
18                     ↪ seq_len)).collect();
19
20                 let insert_pos = if target > i { target - seq_len } else {
21                     ↪ target };
22
23                 if insert_pos <= new_route.len() {
24                     new_route.splice(insert_pos..insert_pos, sequence);
25
26                     let cost = Self::calculate_cost(&new_route, graph);
27
28                     neighbours.push(Solution {
29                         route: new_route,
30                         cost,
31                     });
32                 }
33             }
34         }
35     }
36
37     neighbours
38 }
39

```

```

30         }
31     }
32 }
33 }
34
35     neighbours
36 }

```

2-opt

Esta opera removendo duas arestas do ciclo e reconectando os caminhos de forma a reverter a ordem dos vértices entre os cortes. Esse operador tende a eliminar cruzamentos na rota e promove mudanças estruturais mais significativas que uma simples troca de posições. Ao contrário de *swap*, *shift* e *Or-opt* (implementadas aqui com estratégia de *best improvement*), a implementação de 2-opt neste trabalho usa a estratégia de *first improvement*: assim que é encontrada a primeira reversão que reduz o custo, a solução é atualizada e a busca recomeça a partir da nova solução, isso reduz o custo computacional por iteração, embora possa convergir para ótimos locais de qualidade inferior ao método de melhor melhoria.

A vizinhança 2-opt é gerada por `neighbourhood_by_two_opt`, que itera todos os pares (i, j) com $i < j$ e constrói a rota resultante da inversão do subtrecho entre esses índices (implementação padrão: reverter o trecho $(i + 1) .. j$). Exemplo:

```

1  fn neighbourhood_by_two_opt(&self, graph: &[[f64; N]; N]) -> Vec<Self> {
2      let mut neighbours: Vec<Solution<N>> = Vec::new();
3      let n = self.route.len();
4      if n < 2 { return neighbours; }
5
6      for i in 0..(n - 1) {
7          for j in (i + 1)..n {
8              let mut new_route = self.route.clone();
9              // 2-opt padrão: reverter a parte interna entre i+1 e j
10             if i + 1 <= j {
11                 new_route[(i + 1)..j].reverse();
12             }
13
14             let cost = Self::calculate_cost(&new_route, graph);
15             neighbours.push(Solution { route: new_route, cost });
16         }
17     }
18
19     neighbours
20 }

```

Resultados

Para o Problema do Caixeiro Viajante, executamos cada uma das 12 instâncias do problema 1 vez e obtivemos os seguintes resultados, onde NNSWP é uma abreviação para a heurística do Vizinho Mais Próximo com a busca local do Swap e NIOPT para a Inserção Mais Próxima com a busca local do Or-Opt. A execução foi feita em um Acer Aspire A515-57, com CPU Intel i5 12ª geração e 8GB RAM em um SO Debian 12.

Com base nesta tabela, podemos perceber que a heurística NIOPT costuma produzir soluções melhores que as NNSWP. Contudo, nem sempre esse resultado será consistente, como nos resultados da instância 6, onde o NNSWP se sobressaiu. Além disso, em comparação com

Tabela 2.1: Execução das heurísticas

Instância	NNSWP	NIOPT
1	2291.4	2246.4
2	2355	2223
3	1936	1753.2
4	1933	1810
5	1379.3	1352.5
6	1283	1448
7	815.5	672.7
8	609	606
9	477.2	438.3
10	403	364
11	348.3	348.3
12	306	305

o algoritmo genético, que será apresentado no próximo capítulo, nota-se que com instâncias com dados maiores, as soluções diferem bastante, onde o genético se sobressai em relação às heurísticas deste capítulo. Por fim, podemos concluir que as heurísticas nem sempre produzirão uma solução ótima, mas será provavelmente uma solução boa.

Capítulo 3

Algoritmo Genético e Memético

Neste capítulo, vamos abordar as implementações que realizamos para o algoritmo genético e para o algoritmo memético. Optamos por concentrar em um único capítulo dada à sua estreita ligação.

Quanto ao algoritmo genético, como uma implementação tradicional já é entendida, vamos apenas nos ater as principais decisões que tomamos quanto a implementação do nosso algoritmo para o problema. Tais decisões são divididas em 5 pontos principais:

- Como foi gerada a população inicial.
- Como indivíduos foram selecionados para cruzamento.
- Qual a foi a operação de crossover utilizada.
- Como escolhemos os hiper-parâmetros.

Antes começar a abordar esses pontos é interessante ressaltar que uma das principais diferenças na nossa abordagem foi monomorfizar os dados do grafo através de uma procedural macro (proc-macro) que carrega os dados da instância em tempo de compilação:

```
1 use csv_macro::graph_from_csv;  
2  
3 graph_from_csv!("data/006/data.csv");
```

Isso permite que o compilador faça otimizações mais agressivas pois o grafo do problema é agora conhecido durante a compilação e não em execução. Outra vantagem das proc-macros é a capacidade de implementar o algoritmo sem realizar nenhuma alocação de memória na heap, ou seja, usando apenas a stack, o que também é mais eficiente.

3.1 Discussão sobre as decisões

Enfim começando a falar sobre o algoritmo, comecemos pela estratégia utilizada para a geração da população inicial. Para tal a ideia foi gerar sequências aleatórias de 0 até n e popular a coleção de indivíduos com isso, (n é o número de nós do grafo). Nós fizemos isso primeiro definindo um intervalo e criando um arranjo a partir dele, depois, para cada indivíduo na população, embaralhamos esse arranjo e associamos ele a um indivíduo.


```

1 // Inicializa um intervalo [0..NODE_COUNT).
2 let mut rit = 0..NODE_COUNT;
3 // Cada elemento do intervalo é associado a um arranjo.
4 let mut r: [usize; NODE_COUNT] = array::from_fn(|_| unsafe {
5     ↪ rit.next().unwrap_unchecked() });
6 // Para cada indivíduo da população (não estarão inicializados).
7 for i in p {
8     // O arranjo é embaralhado com um gerador de números aleatório.
9     r.shuffle(rng);
10    // O indivíduo é associado ao arranjo embaralhado.
11    *i = r;
12 }

```

Para o cruzamento, apenas dividimos a população em duas metades "iguais" e realizamos o crossover pointwise de cada elemento das duas metades, após isso re-embaralhamos a população para sempre permitir que indivíduos diferentes tenham a oportunidade de sofrer crossover. É importante ressaltar que o crossover não tem a garantia de ser bem sucedido no nosso algoritmo, não no sentido de adicionar um indivíduo com fitness pior na nossa população, mas no de não aceitar a prole se ela não tiver o fitness melhor que pelo menos um dos pais. Entraremos em mais detalhes sobre isso brevemente, mas essa é implementação da seleção.

```

1 // Separa a população em duas metades.
2 let (h1, h2) = p.split_at_mut(p.len() / 2);
3 // Realiza o crossover pointwise em indivíduos das duas metades.
4 for (p1, p2) in h1.iter_mut().zip(h2) {
5     if let Some(i) = cross(p1, p2)
6         && rand::random_bool(mrate)
7     {
8         // Realiza uma mutação somente se o crossover foi bem sucedido e a
9         ↪ prole foi sorteada.
10        mutate([p1, p2][i]);
11    }
12 }
13 // Re-embaralha a população.
14 p.shuffle(rng);

```

Quanto ao crossover, usamos o Sequential Constructive Crossover (SCX) (AHMED, 2010), que é considerado o melhor operador de crossover por alguns autores na literatura (KHAN, 2015). A principal vantagem desse operador é que ele mantém características positivas dos parentes enquanto possivelmente descobre outros bons genes. A ideia geral do algoritmo é iterativamente escolher nós dos pais aonde o próximo nó é um dos primeiros não visitados e o que compõe a menor distância para o nó atual da iteração. Esse primeiro nó não visitado é denominado nó legítimo pelo autor.

```

1 // Inicializa offspring como arranjo zerado.
2 let mut offspring = [0; NODE_COUNT];
3 // Inicializa arranjo de nós visitados.
4 let mut visited = [false; NODE_COUNT];
5 // Escolhe aleatoriamente o primeiro nó da prole.
6 let mut fst = [&p1, &p2][rand::random_range(0..2)][0];
7 offspring[0] = fst;
8 // Marca o primeiro nó como visitado.

```

```

9  visited[fst] = true;
10 // Para cada nó que não é o primeiro, é escolhido os
11 // nós legítimos a e b dos pais e o que tiver a menor
12 // distância para o nó atual da iteração é incorporado na prole.
13 for n in offspring.iter_mut().skip(1) {
14     let a = legitimate(fst, &mut visited, p1);
15     let b = legitimate(fst, &mut visited, p2);
16     *n = if g[fst][a] < g[fst][b] { a } else { b };
17     fst = *n;
18     visited[*n] = true;
19 }
20 // Sobrescreve o pai que tiver menor fitness que a prole (se houver).
21 [p1, p2].iter_mut().enumerate().find_map(|(i, p)| {
22     (fit(&offspring) < fit(p)).then(|| {
23         **p = offspring;
24         i
25     })
26 })

```

Quanto aos hiper-parâmetros, simplesmente realizamos o tuning do irace(LOPEZ-IBANEZ et al., 2016) para as instâncias que tínhamos. Para as instâncias de tempo e distância foram usados os seguintes parâmetros respectivamente:

Tipo	Número de iterações	Tamanho da população	Taxa de mutação
Tempo (min)	2452	197	0.0193
Distância (km)	677	195	0.0152

3.1.1 Algoritmo memético

Quanto ao algoritmo memético, a principal diferença é a existência da busca local na fase de mutação. Esta se dá da seguinte forma:

```

1  // Mesma lógica de seleção.
2  let (h1, h2) = p.split_at_mut(p.len() / 2);
3  for (p1, p2) in h1.iter_mut().zip(h2) {
4      if let Some(i) = cross(p1, p2)
5         && rand::random_bool(mrate)
6      {
7          // Offspring que substitui um dos pais.
8          let offspring = &mut [p1, p2][i];
9          let s = {
10             mutate(offspring);
11             // Desta vez é chamado um construtor de Solution.
12             individual_to_solution(offspring)
13         };
14         // Usamos a instância de Solution para realizar a busca local escolhida
15         ↪ aleatoriamente.
16         let s = match rand::random_range(1..=100) {
17             1..25 => s.shift(&g, s.route[0]),
18             25..50 => s.swap(&g, s.route[0]),
19             50..75 => s.two_opt(&g),
20             _ => s.or_opt(&g),
21         };
22         // Offspring se torna a versão modificada de s.
23         offspring.copy_from_slice(&s.route);

```

```

23     }
24 }
25 // Mesma lógica de embaralhamento.
26 p.shuffle(rng);

```

3.1.2 Resultados

Realizamos os testes do genético e do memético em duas máquinas diferentes, o genético numa com um Ryzen 9 5900X e o memético num Intel Core i7 Ultra 155H.

Estes são os resultados do algoritmo genético:

Instância	Mínimo	μ_{custo}	σ_{custo}	μ_{tempo}	σ_{tempo}	Execuções
1	1952.10	2042.73	25.97	0.16	0.01	11727.00
2	1996.00	2052.01	17.17	0.54	0.03	3497.00
3	1708.00	1777.23	21.29	0.09	0.01	20939.00
4	1663.00	1697.99	13.58	0.31	0.02	5996.00
5	1321.00	1346.34	10.26	0.04	0.00	41693.00
6	1223.00	1240.54	10.32	0.16	0.01	11950.00
7	672.70	673.08	2.29	0.02	0.00	101487.00
8	606.00	606.18	0.75	0.06	0.01	30650.00
9	438.30	438.30	0.15	0.01	0.00	169785.00
10	364.00	364.00	0.08	0.03	0.00	55279.00
11	344.90	344.90	0.00	0.01	0.00	192837.00
12	305.00	305.00	0.00	0.03	0.00	64604.00

Estes são os resultados do algoritmo memético:

Instância	Mínimo	μ_{custo}	σ_{custo}	μ_{tempo}	σ_{tempo}	Execuções
1	1942.30	1960.57	21.64	1.49	0.52	731.00
2	1973.00	1994.96	12.19	1.40	0.45	771.00
3	1695.00	1701.51	10.25	0.68	0.20	1583.00
4	1662.00	1669.38	7.37	0.60	0.18	1761.00
5	1321.00	1324.63	6.48	0.22	0.04	4845.00
6	1223.00	1225.33	4.72	0.22	0.04	4797.00
7	672.70	672.72	0.57	0.06	0.01	17421.00
8	606.00	606.01	0.16	0.06	0.01	16273.00
9	438.30	438.30	0.07	0.03	0.01	29911.00
10	364.00	364.00	0.00	0.03	0.01	29202.00
11	344.90	344.90	0.00	0.03	0.01	31797.00
12	305.00	305.00	0.00	0.03	0.01	30212.00

Um sumário dos resultados dos dois

Instância	Min. Genético	Min. Memético
1	1952.10	1942.30
2	1996.00	1973.00
3	1708.00	1695.00
4	1663.00	1662.00
5	1321.00	1321.00
6	1223.00	1223.00
7	672.70	672.70
8	606.00	606.00
9	438.30	438.30
10	364.00	364.00
11	344.90	344.90
12	305.00	305.00

Capítulo 4

Conclusão

Referências Bibliográficas

AHMED, Z. H. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics & Bioinformatics (IJBB)*, v. 3, n. 6, p. 96, 2010.

KHAN, I. H. Assessing different crossover operators for travelling salesman problem. *International Journal of Intelligent Systems and Applications*, Modern Education and Computer Science Press, v. 7, n. 11, p. 19, 2015.

LOPEZ-IBANEZ, M. et al. *The irace package: Iterated Racing for Automatic Algorithm Configuration*. 2016. 43–58 p.

Apêndice A

Atividades desenvolvidas por cada integrante

- **Alexandre Dantas:** Algoritmo de Kruskal, Algoritmo de Prim, Relatório (Cap. 03, 04), documentação.
- **Andriel Vinicius:** Algoritmo de Dijkstra, Algoritmo de Hierholzer, Relatório (Cap. 01, 02, 03, 04, 05), revisão de código, documentação.
- **Gabriel Carvalho:** Algoritmo de Bellman-Ford, Relatório (Cap. 03, 04), documentação, vídeo demonstrativo.
- **Maria Paz:** Algoritmo de Hierholzer, Relatório (Cap. 02, 03, 04), documentação.
- **Vinicius de Lima:** Algoritmo de Floyd-Warshall e descoberta do caminho mais curto, Relatório (Cap. 02, 03, 04), revisão de código, documentação.