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 3
3.1 Discussão sobre as decisões 3
3.1.1 Algoritmo memético 5
3.1.2 Resultados 6

4 Conclusão 8

Referências Bibliográficas 9

Appendices 10

A Atividades desenvolvidas por cada integrante 10

Lista de Figuras

Lista de Tabelas

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: Nearest Neighbour e Nearest Insertion.

Nearest Neighbour (Vizinho Mais Próximo)

A heurística do Nearest Neighbour consiste em um algoritmo guloso que, dado um nó arbitrário, sempre escolhe a aresta de menor custo até o próximo vizinho sem se importar com as consequências futuras.

Buscas Locais

Resultados

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 dividas 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.

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

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.

```rust
// Separa a população em duas metades.
let (h1, h2) = p.split_at_mut(p.len() / 2);
// Realiza o crossover pointwise em indivíduos das duas metades.
for (p1, p2) in h1.iter_mut().zip(h2) {
    if let Some(i) = cross(p1, p2)
        && rand::random_bool(mrate)
    {
        // Realiza uma mutação somente se o crossover foi bem sucedido e a
        ↪  prole foi sorteada.
        mutate([p1, p2][i]);
    }
}
// Re-embaralha a população.
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 possívelmente descobre outros bons genes. A ideia geral do algorimo 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.

```rust
// Inicializa offspring como arranjo zerado.
let mut offspring = [0; NODE_COUNT];
// Inicializa arranjo de nós visitados.
let mut visited = [false; NODE_COUNT];
// Escolhe aleatóriamente o primeiro nó da prole.
let mut fst = [&p1, &p2][rand::random_range(0..2)][0];
offspring[0] = fst;
// Marca o primeiro nó como visitado.
```

```
visited[fst] = true;
// Para cada nó que não é o primeiro, é escolhido os
// nós legítimos a e b dos pais e o que tiver a menor
// distância para o nó atual da iteração é incorporado na prole.
for n in offspring.iter_mut().skip(1) {
    let a = legitimate(fst, &mut visited, p1);
    let b = legitimate(fst, &mut visited, p2);
    *n = if g[fst][a] < g[fst][b] { a } else { b };
    fst = *n;
    visited[*n] = true;
}
// Sobrescreve o pai que tiver menor fitness que a prole (se houver).
[p1, p2].iter_mut().enumerate().find_map(|(i, p)| {
    (fit(&offspring) < fit(p)).then(|| {
        **p = offspring;
        i
    })
})
```

Quanto aos hiper-parâmetros, simplemesmente realizamos o tuning do irace(LOPEZ-IBANEZ et al., 2016) para as instâncias que tinhamos. 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:

```
// Mesma lógica de seleção.
let (h1, h2) = p.split_at_mut(p.len() / 2);
for (p1, p2) in h1.iter_mut().zip(h2) {
    if let Some(i) = cross(p1, p2)
        && rand::random_bool(mrate)
    {
        // Offspring que substitui um dos pais.
        let offspring = &mut [p1, p2][i];
        let s = {
            mutate(offspring);
            // Desta vez é chamado um construtor de Solution.
            individual_to_solution(offspring)
        };
        // Usamos a instância de Solution para realizar a busca local escolhida
        ↪ aleatóriamente.
        let s = match rand::random_range(1..=100) {
            1..25 => s.shift(&g, s.route[0]),
            25..50 => s.swap(&g, s.route[0]),
            50..75 => s.two_opt(&g),
            _ => s.or_opt(&g),
        };
        // Offspring se torna a versão modificada de s.
        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.