



UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

Relatórios Técnicos
do Departamento de Informática Aplicada
da UNIRIO
n° SN/2021

**Complexidade empírica do algoritmo
de Dijkstra para o Problema do
Caminho Mínimo**

Caio Souza
Emmanuel Gomes
Fabício Janssen
Marcelo Simas

Departamento de Informática Aplicada

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
Av. Pasteur, 458, Urca - CEP 22290-240
RIO DE JANEIRO – BRASIL

Complexidade empírica do algoritmo de Dijkstra para o Problema do Caminho Mínimo *

Caio Souza¹, Emmanuel Gomes¹, Fabrício Janssen¹, Marcelo Simas¹

¹Depto de Informática Aplicada – Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

caio.souza@edu.unirio.br, emmanuel.souza@edu.unirio.br,
fabricao.janssen@uniriotec.br, marcelo.mattos@edu.unirio.br

Abstract. This technical report presents an implementation in C of the Dijkstra algorithm to solve the shortest path tree problem. Two versions of the algorithm were implemented, in C language, using an *array* and a *binary heap* to store the lowest costs and it's presented a comparison between those. The empirical complexity analysis was performed using the five available classes' instances. It was concluded that in classes that in instances with sparse graphs, the array version presented a processing time close to the theoretical complexity n^2 and that the heap version presented a processing time close to the theoretical complexity $m \log n$. The classes of symmetric complete graphs and asymmetric complete graphs also presented processing time close to the theoretical complexity n^2 for the array version, and for the heap version presented processing time close to the theoretical complexity $m \log n$.

Keywords: *Dijkstra Algorithm, Shortest path, Time complexity.*

Resumo. Este relatório técnico apresenta a implementação na linguagem C do algoritmo de Dijkstra para resolver o problema da árvore de caminho mínimo. É feita a implementação de duas versões do algoritmo, em linguagem C, usando *array* e *heap* binário para o armazenamento dos menores custos e apresentado um comparativo entre elas. Essa análise de complexidade empírica foi realizada utilizando-se as cinco classes de instâncias disponibilizadas. Concluiu-se que nas classes que possuem grafos esparsos, o algoritmo em versão *array* apresentou tempo de processamento próximo à complexidade teórica n^2 e que a versão em *heap* apresentou tempo de processamento próximo à complexidade teórica $m \log n$. As classes de grafos completos simétricos e grafos completos assimétricos também apresentaram para o algoritmo em versão *array* tempo de processamento próximo à complexidade teórica n^2 e para a versão em *heap* apresentou tempo de processamento próximo à complexidade teórica $m \log n$.

Palavras-chave: *Algoritmo Dijkstra, Caminho mais curto, Complexidade de tempo.*

Sumário

1. Introdução	4
2. Conceitos Fundamentais	5
2.1 Problema do Caminho Mínimo	5
2.2 O algoritmo de Dijkstra	6
3. Implementação e Análise do algoritmo	6
3.1 Algoritmo de Dijkstra com armazenamento em array	7
3.2 Algoritmo de Dijkstra com armazenamento em heap	8
4. O ambiente computacional utilizado	10
5. Rotina para medição do tempo	10
6. Discussão sobre a implementação e análise do algoritmo	12
6.1 Instâncias ALUE	12
6.2 Instâncias ALUT	15
6.3 Instâncias DMXA	18
6.4 Instâncias test_set1	21
6.5 Instâncias test_set2	24
7. Conclusão	26
Referências	27

1. Introdução

Os algoritmos são cada vez mais importantes na seleção das informações consideradas de maior relevância para a sociedade (Gillespie, 2014). Mas o que são **algoritmos**? Para Cormen (2017), trata-se de um conjunto de etapas para se executar uma tarefa. “Você tem um algoritmo para escovar os dentes: abrir o tubo da pasta dental, pegar a escova de dentes, apertar o tubo da pasta dental sobre a escova colocando a quantidade necessária, fechar o tubo”, explica o autor.

No contexto dos Sistemas de Informação, Ziviani (2004) define que “um algoritmo pode ser visto como uma sequência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema”. Para ele, cada passo “deve ser bem definido, sem ambiguidades, e executáveis computacionalmente”. Ainda segundo o autor, **estruturas de dados** e algoritmos estão intimamente ligados, pois não se pode estudar estruturas de dados sem considerar os algoritmos ligados a elas, assim como a escolha dos algoritmos em geral depende da estrutura dos dados: “**Programar** é basicamente estruturar dados e construir algoritmos”.

É possível desenvolver vários algoritmos para solucionar o mesmo problema. Na área de **análise de algoritmos**, existem dois caminhos bem distintos, segundo Knuth (1971): (i) Qual é o custo de usar um algoritmo para resolver um problema específico? (ii) Qual é o algoritmo de menor custo possível para resolver o problema? De acordo com Duch (2007), a característica básica que um algoritmo deve ter é que seja correto, ou seja, que produza o resultado desejado em tempo finito. Mais distante, pode interessar que seja claro, bem estruturado, fácil de usar, fácil de implementar e eficiente. Entende-se pela eficiência de um algoritmo a quantidade de recursos de computação que requer; ou seja, qual é o seu tempo de execução e quanto usa de memória. A quantidade de tempo necessária para a execução de um determinado algoritmo é chamado de **custo de tempo**, enquanto a quantidade de memória que é necessária é frequentemente chamado de **custo de espaço** (Duch, 2007).

O problema de caminho mínimo é um dos problemas genéricos mais estudados e utilizados em diversas áreas como Ciência da Computação e Inteligência Artificial. Dada uma rede, composta de nós (ou vértices) e arcos (ou ligações) entre esses nós, o problema pode ser enunciado como o de encontrar o menor caminho entre dois nós dessa rede (Von Atzingen *et. al*, 2012). O algoritmo *Dijkstra*, que consiste em verificar se há a possibilidade de melhorar o caminho obtido, é eficiente para resolver esse problema.

Neste trabalho, foram implementadas duas versões desse algoritmo, na linguagem C. Na primeira, os valores são armazenados em um vetor (*array*); e, na segunda, os valores são armazenados em uma fila de prioridades implementada com um *heap binário*. Em seguida, foi realizada uma análise de complexidade empírica utilizando as cinco classes de instâncias disponibilizadas.

Conforme esperado, nas classes que possuem instâncias de grafos esparsos, o algoritmo em versão *array* mostrou-se mais lento que a versão em *heap*, por sua complexidade ser maior. Porém, observou-se que o tempo de processamento foi equivalente nas classes de grafos completos.

Além desta Introdução, a Seção 2 apresenta o problema do caminho mínimo, e o algoritmo Dijkstra; a Seção 3 explica a implementação e a análise do algoritmo; a Seção 4 apresenta o ambiente computacional que foi utilizado; a Seção 5 relata a rotina para medição do tempo de execução; a Seção 6 traz uma discussão sobre a implementação do algoritmo; na Seção 7 é realizado um comparativo dos tempos de execução com a complexidade esperada; e, finalmente, a Seção 8 apresenta a conclusão.

2. Conceitos Fundamentais

2.1 Problema do Caminho Mínimo

O problema do caminho mínimo ou caminho mais curto, consiste em encontrar o melhor caminho entre dois nós. Assim, resolver este problema pode significar determinar o caminho entre dois nós com o custo mínimo, ou com o menor tempo de viagem.

Numa rede qualquer, dependendo das suas características, pode existir vários caminhos entre um par de nós, definidos como origem e destino. Entre os vários caminhos, aquele que possui o menor “peso” é chamado de caminho mínimo. Este peso representa a soma total dos valores dos arcos que compõem o caminho e estes valores podem ser: o tempo de viagem, a distância percorrida ou um custo qualquer do arco.

1) Modelagem matemática

O modelo matemático para o problema de caminho mais curto do nó 1 ao nó n de um grafo $G=(N,E)$, $N = \{1, 2, \dots, n\}$.

- Variáveis:
 $x_{ij} \in \{0,1\}$ ativação, ou não do arco (i, j)
- Parâmetros:
 c_{ij} custo unitário do fluxo em (i, j)
 $S(j)$ é o conjunto dos nós sucessores de j
 $P(j)$ é o conjunto dos nós predecessores de j
- Função objetivo:

$$\min Z = \sum_{i=1}^n \sum_{j \in S(i)} c_{ij} x_{ij}$$

- Restrições:

$$\sum_{j \in S(1)} x_{1j} = 1$$

$$\sum_{i \in P(n)} x_{in} = 1$$

$$\sum_{i \in P(j)} x_{ij} = \sum_{k \in S(j)} x_{jk}, j = 2, \dots, n-1$$

Para resolução do problema de caminho mínimo, existem algoritmos alternativos mais simples e mais eficientes, como por exemplo o *Dijkstra*.

2.2 O algoritmo de *Dijkstra*

O algoritmo de *Dijkstra* é uma solução para o problema do caminho mínimo de origem única. Funciona em grafos orientados e não orientados, no entanto, todas as arestas devem ter custos não negativos. Se houver custos negativos, usa-se o algoritmo de *Bellman-Ford*.

Entrada: Grafo ponderado $G=(N,E)$ e nó origem $O \in N$, de modo que todos os custos das arestas sejam não negativos.

Saída: Comprimentos de caminhos mais curtos (ou os caminhos mais curtos em si) de um determinado nó origem $O \in N$ para todos os outros nós.

3. Implementação e Análise do algoritmo

O algoritmo de *Dijkstra* identifica, a partir do nó O , qual é o custo mínimo entre esse nó e todos os outros do grafo. No início, o conjunto S contém somente esse nó, chamado origem. A cada passo, selecionamos no conjunto de nós sobrando, o que está mais perto da origem. Depois, para cada nó que está sobrando, é atualizada a sua distância em relação à origem. Se passando pelo novo nó acrescentado, a distância fica menor, é essa nova distância que será memorizada.

Escolhido um nó como origem da busca, este algoritmo calcula, então, o custo mínimo deste nó para todos os demais nós do grafo. O procedimento é iterativo, determinando, na iteração 1, o nó mais próximo do nó O , na segunda iteração, o segundo nó mais próximo do nó O , e assim sucessivamente, até que em alguma iteração todos os n nós sejam atingidos.

Seja $G=(N,E)$ um grafo orientado e s um nó de G :

- Atribua valor zero à estimativa do custo mínimo do nó O (a origem da busca) e infinito às demais estimativas;
- Atribua um valor qualquer aos precedentes (o precedente de um nó t é o nó que precede t no caminho de custo mínimo de s para t);

- Enquanto houver nó aberto:
 - Seja k um nó ainda aberto cuja estimativa seja a menor dentre todos os nós abertos;
 - Feche o nó k ;
- Para todo nó j ainda aberto que seja sucessor de k faça:
 - Some a estimativa do nó k com o custo do arco que une k a j ;
 - Caso esta soma seja melhor que a estimativa anterior para o nó j , substitua-a e anote k como precedente de j .

3.1 Algoritmo de *Dijkstra* com armazenamento em *array*

O algoritmo de *Dijkstra*, em alto nível, está descrito nas figuras a seguir, seguido de suas implementações.

A figura 01 apresenta o algoritmo em sua versão com armazenamento em *array*. Na página seguinte, é apresentada a implementação do mesmo em C, na figura 02.

```

algorithm Dijkstra;
begin
   $S := \emptyset; \bar{S} := N;$ 
   $d(i) := \infty$  for each node  $i \in N$ ;
   $d(s) := 0$  and  $\text{pred}(s) := 0$ ;
  while  $|S| < n$  do
    begin
      let  $i \in \bar{S}$  be a node for which  $d(i) = \min\{d(j) : j \in \bar{S}\}$ ;
       $S := S \cup \{i\}$ ;
       $\bar{S} := \bar{S} - \{i\}$ ;
      for each  $(i, j) \in A(i)$  do
        if  $d(j) > d(i) + c_{ij}$  then  $d(j) := d(i) + c_{ij}$  and  $\text{pred}(j) := i$ ;
    end;
  end;

```

Figura 01. Algoritmo de *Dijkstra* em alto nível com armazenamento em *array*. [Ahuja, pág 109]

```

void dijkstra_array(Graph graph, size_t costs[]) {
    LinkedList unexplored = LL_new(graph->vertices);

    // Initialize the unexplored to every vertex and the costs to max
    for (size_t i = 0; i < graph->vertices; i++) {
        costs[i] = SIZE_MAX;
        LL_append(unexplored, i, SIZE_MAX);
    }

    // Starts with the first on the graph as explored
    costs[0] = 0;

    while (unexplored->len) {
        size_t smallest_vertex = LL_remove_smallest_by_cost(unexplored, costs);

        LinkedList adjacency = graph->adjacency[smallest_vertex];
        size_t current_cost = costs[smallest_vertex];

        for (LL_foreach(adjacency, neighbor)) {
            size_t n_vertex = neighbor->vertex;
            size_t new_cost = current_cost + neighbor->cost;

            if (new_cost < costs[n_vertex]) {
                costs[n_vertex] = new_cost;
            }
        }
    }

    LL_delete(&unexplored);
}

```

Figura 02. Implementação do algoritmo de *Dijkstra* armazenamento em *array*.

3.2 Algoritmo de *Dijkstra* com armazenamento em *heap*

A figura 03 apresenta o algoritmo em sua versão com armazenamento em *heap*, em alto nível. Logo abaixo, é apresentada a implementação do mesmo em C, na figura 04.


```

algorithm heap-Dijkstra;
begin
    create-heap(H);
     $d(j) := \infty$  for all  $j \in N$ ;
     $d(s) := 0$  and  $\text{pred}(s) := 0$ ;
    insert(s, H);
    while  $H \neq \emptyset$  do
        begin
            find-min(i, H);
            delete-min(i, H);
            for each  $(i, j) \in A(i)$  do
                begin
                     $\text{value} := d(i) + c_{ij}$ ;
                    if  $d(j) > \text{value}$  then
                        if  $d(j) = \infty$  then  $d(j) := \text{value}$ ,  $\text{pred}(j) := i$ , and insert(j, H)
                        else set  $d(j) := \text{value}$ ,  $\text{pred}(j) := i$ , and decrease-key( $\text{value}$ , i, H);
                end;
            end;
        end;
    end;

```

Figura 03. Algoritmo de *Dijkstra* em alto nível com armazenamento em *heap*. [Ahuja, pág 115]

```

void dijkstra_heap(Graph graph, size_t costs[]) {
    Heap explored = HEAP_new(graph->vertices);

    // Initialize the unexplored to every vertex and the costs to max
    for (size_t i = 0; i < graph->vertices; i++)
        costs[i] = SIZE_MAX;

    // Starts with the first on the graph as explored
    costs[0] = 0;
    HEAP_add(explored, edge: 0, cost: 0);

    while (explored->size) {
        Edge smallest = HEAP_pop(explored);
        size_t smallest_vertex = smallest.id;

        LinkedList adjacency = graph->adjacency[smallest_vertex];
        size_t current_cost = costs[smallest_vertex];

        for (LL_foreach(adjacency, neighbor)) {
            size_t n_vertex = neighbor->vertex;
            size_t new_cost = current_cost + neighbor->cost;

            if (new_cost < costs[n_vertex]) {
                if (costs[n_vertex] == SIZE_MAX)
                    HEAP_add(explored, n_vertex, new_cost);
                else
                    HEAP_decrease_to(explored, n_vertex, new_cost);
                costs[n_vertex] = new_cost;
            }
        }
    }

    HEAP_delete(&explored);
}

```

Figura 04. Implementação do algoritmo de *Dijkstra* armazenamento em *heap*.

4. O ambiente computacional utilizado

Para a realização da simulação do algoritmo de *Dijkstra* e avaliação empírica foi desenvolvido a programação seguindo projeto modular de implementação dos algoritmos necessários, para isso, foi utilizado o Ambiente de Desenvolvimento Integrado (IDE) CLion. Neste IDE, foi realizada a codificação usando a Linguagem de programação C. O código fonte foi compilado usando o GNU Compiler Collection (gcc) na versão 9.3.0, sendo que foi usado com flag de otimização "-O3". Um destaque importante em relação a performance é à codificação de medição do tempo no qual foi usada a função da Linguagem C *clock_gettime* com o parâmetro *CLOCK_MONOTONIC*, este parâmetro permite a precisão de décimos de milésimos de segundos. A seguir são apresentados os recursos computacionais usados para a implementação do algoritmo:

- Processador: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz. Arquitetura x86_64. 6 cores (12 processadores lógicos). Programa single thread.
- Cache: L1 384KB. L2 1,5MB. L3 12,0MB.
- Memória RAM: 32GB.
- Linguagem: C. Versão C11.
- Sistema operacional: Linux Ubuntu/WSL 5.10.16.3-microsoft-standard-WSL2.
- Compilação: GNU Compiler Collection (gcc). Versão 9.3.0. Compilado com flag de otimização "-O3".

5. Rotina para medição do tempo

```
Graph graph = GRAPH_parse_test( filename: "../instances/test_set1/check_v5_s1.dat");

benchmark(
    name: "array",
    graph,
    dijkstra_array,
    print_result: false,
    warmup_amount: 10,
    measure_amount: 20);

benchmark(
    name: " heap",
    graph,
    dijkstra_heap,
    print_result: false,
    warmup_amount: 10,
    measure_amount: 20);
```

Figura 05. Exemplo de invocação do *benchmark*.

```

double get_time() {
    struct timespec t;
    clock_gettime(CLOCK_MONOTONIC, &t);
    return (double) t.tv_sec + 1.0e-9 * t.tv_nsec;
}

void benchmark(
    char *name,
    Graph graph,
    void (*dijkstra)(Graph, size_t[]),
    bool print_result,
    size_t warmup_amount,
    size_t measure_amount
) {
    size_t costs[graph->vertices];

    // Warmup
    for (int i = 0; i < warmup_amount; i++)
        dijkstra(graph, costs);

    double measures[measure_amount];

    // Measure
    printf( fmt: "Benchmark %s (%zu, %zu):", name, graph->vertices, graph->edges);
    for (int i = 0; i < measure_amount; i++) {
        double start_time = get_time();
        dijkstra(graph, costs);
        measures[i] = get_time() - start_time;

        // Print the last result
        if (print_result && i == measure_amount - 1) {
            printf( fmt: "Results:\n");
            for (size_t j = 0; j < graph->vertices; j++)
                printf( fmt: "%zu %zu\n", j, costs[j]);
            printf( fmt: "End results");
        }
    }

    for (int i = 0; i < measure_amount; i++)
        printf( fmt: " %2.8f", measures[i]);
    printf( fmt: "\n");
}

```

Figura 06. Algoritmo para medição do tempo de execução.

A medição do tempo se fez da seguinte forma:

- O algoritmo é executado 50 vezes. O tempo de cada uma desta execução foi armazenado.
- Em cada execução, foi armazenado o tempo preciso (em nanosegundos) em que ela iniciou e finalizou, ignorando a leitura de dados.
- Nas tabelas são apresentados os valores mínimos destas execuções.

Conforme Chen, o valor mínimo é o melhor para representar esta medida, se comparado com média ou mediana por exemplo, pois qualquer ruído que ocorrer na execução somente é capaz de adicionar e nunca remover tempo da execução. Por isso o valor mínimo é o mais fiel.

6. Discussão sobre a implementação e análise do algoritmo

Para o estudo do comportamento empírico do algoritmo de *Dijkstra* realizou-se a implementação de duas versões do algoritmo: a versão em que os valores $d(i)$ são armazenados em um vetor (*array*), e temos $i = 1, \dots, n$; e a versão em que os valores $d(i)$ são armazenados em uma fila de prioridades implementada com um *heap binário*, com $i = 1, \dots, n$. Ambas as versões foram codificadas pelo grupo especificamente para este trabalho.

A análise de complexidade empírica foi realizada utilizando-se as cinco classes de instâncias disponibilizadas para o trabalho, a saber:

- Classe ALUE - conjunto de 14 instâncias simétricas de grafos esparsos.
- Classe ALUT - conjunto de 9 instâncias simétricas de grafos esparsos.
- Classe DMXA - conjunto de 15 instâncias simétricas de grafos esparsos.
- test_set1 - classe de grafos completos e simétricos composta de dez instâncias. Sendo uma instância para cada valor de $n \in \{100; 200; 300; 400; 500; 600; 700; 800; 900; 1000\}$.
- test_set2 - classe de grafos completos e assimétricos composta de dez instâncias. Sendo uma instância para cada valor de $n \in \{100; 200; 300; 400; 500; 600; 700; 800; 900; 1000\}$.

Para cada uma das instâncias, o algoritmo foi executado 20 vezes em cada uma das versões (*array* e *heap*). Os resultados do tempo de execução gerados de cada instância foram imprimidos para se realizar a análise de complexidade. A programação foi codificada de modo que estes resultados ficassem dispostos de forma tabulada, facilitando assim a geração de gráficos. Tanto as tabelas geradas, quanto os gráficos são exibidos nas próximas seções, assim como as devidas análises.

6.1 Instâncias ALUE

As instâncias ALUE são simétricas de grafos esparsos, quanto à característica de serem grafos esparsos isso pode ser verificado na Tabela 1 pois o número de arestas é da mesma ordem n de nós em cada uma das instâncias.

Tabela 1 – Características de nós e arestas das instâncias ALUE

Instâncias	Nós	Arestas
alue7229	940	1.474
alue2105	1.220	1.858
alue2087	1.244	1.971
alue6951	2.818	4.419
alue6179	3.372	5.213
alue5067	3.524	5.560
alue3146	3.620	5.869
alue6457	3.932	6.137
alue6735	4.119	6.696
alue5623	4.472	6.938
alue5345	5.179	8.165
alue7066	6.405	10.454
alue5901	11.543	18.429
alue7065	34.046	54.841
alue7080	34.479	55.494

A Tabela 2 traz o mínimo dos valores para o tempo de execução de cada instância ALUE em nanossegundos (ns). Para fazer o estudo do comportamento empírico e analisar a complexidade e o crescimento do tempo de processamento, foram gerados gráficos a partir dos valores dos mínimos dos tempos para as versões do algoritmo *array* e *heap* obtidos da Tabela 2. Os gráficos são apresentados a seguir.

Tabela 2 – Tempo mínimo de execução das instâncias ALUE

Instâncias	Array - tempo mínimo (ns)	Heap - tempo mínimo (ns)
alue7229	734.000	37.700
alue2105	1.260.100	53.800
alue2087	1.300.000	55.300
alue6951	6.509.900	148.800
alue6179	9.310.500	182.900
alue5067	9.934.500	195.400

alue3146	10.796.000	214.600
alue6457	12.619.900	215.500
alue6735	14.091.600	251.099
alue5623	16.575.800	276.800
alue5345	22.210.400	321.600
alue7066	34.210.700	403.100
alue5901	120.564.300	803.700
alue7065	1.055.607.000	2.740.700
alue7080	1.080.797.000	2.582.200

Com os resultados da versão *array* obtidos da Tabela 2, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias ALUE para a versão *array* o resultado empírico de tempo de execução é próximo a complexidade teórica n^2 , o que pode ser verificado no gráfico exibido na Figura 7, que faz a comparação do tempo real de execução com a complexidade teórica.

**Comparação tempo real e complexidade teórica
ALUE com array**

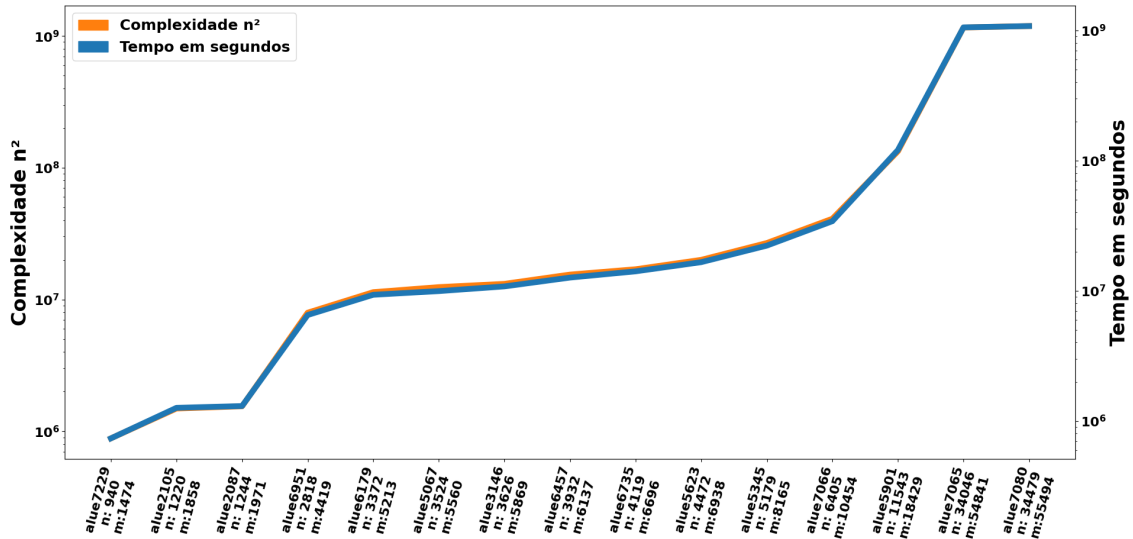


Figura 07. Gráfico de instâncias ALUE com comparação tempo real com a versão array e complexidade teórica.

Com os resultados da versão *heap* obtidos da Tabela 2, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias ALUE para a versão *heap* o resultado empírico de tempo de execução é próximo a complexidade teórica $m \log m$, o que pode ser verificado no gráfico exibido na Figura 8, que faz a comparação do tempo real de execução com a complexidade teórica.

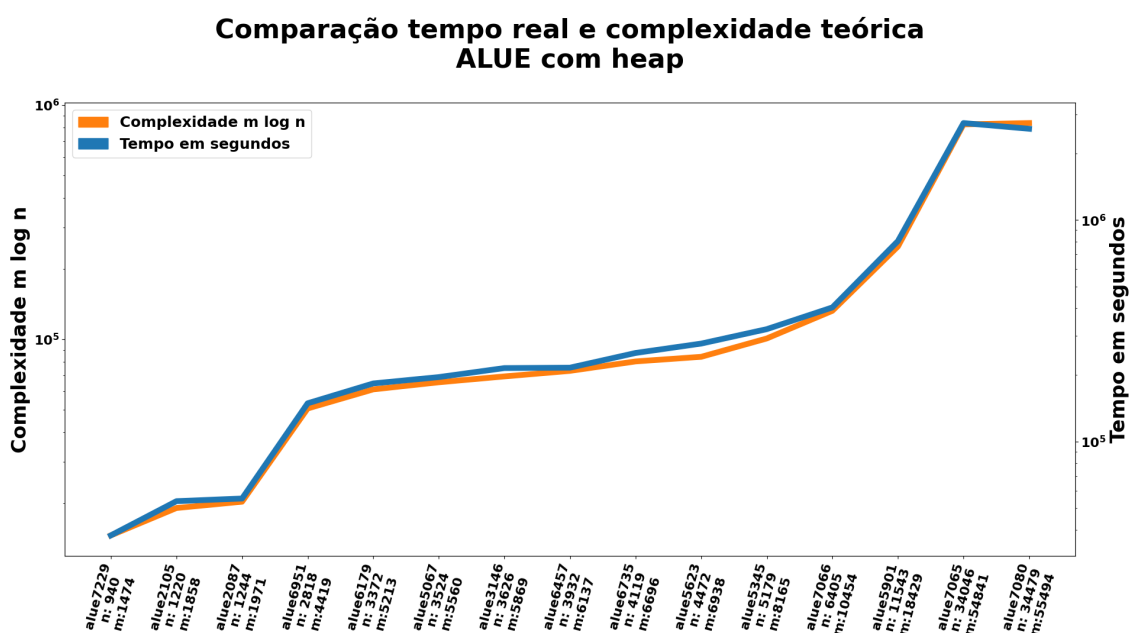


Figura 08. Gráfico de instâncias ALUE com comparação tempo real com a versão *heap* e complexidade teórica.

6.2 Instâncias ALUT

As instâncias ALUT, assim como as instâncias ALUE, também são simétricas de grafos esparsos. O número de nós e arestas de cada instância podem ser verificados na Tabela 3.

Tabela 3 – Características de nós e arestas das instâncias ALUT

Instâncias	Nós	Arestas
alut2764	387	626
alut0805	966	1.666
alut0787	1.160	2.089
alut1181	3.041	5.693
alut2566	5.021	9.055
alut2010	6.104	11.011
alut2288	9.070	16.595
alut2610	33.901	62.816
alut2625	36.711	68.117

A Tabela 4 traz os valores mínimos para o tempo de execução de cada instância ALUT em nanossegundos (ns). Para fazer o estudo do comportamento empírico e analisar a complexidade e o crescimento do tempo de processamento, foram gerados

gráficos a partir dos valores mínimos dos tempos para as versões do algoritmo *array* e *heap* obtidos da Tabela 4. Os gráficos são apresentados a seguir.

Tabela 4 – Tempo mínimo de execução das instâncias ALUT

Instâncias	Array - tempo mínimo (ns)	Heap - tempo mínimo (ns)
alut2764	131.200	8.500
alut0805	776.100	36.100
alut0787	1.106.000	43.200
alut1181	7.697.400	152.500
alut2566	20.835.200	280.000
alut2010	31.061.900	342.600
alut2288	72.560.600	568.400
alut2610	1.044.071.800	3.072.200
alut2625	1.230.391.300	2.419.100

Com os resultados da versão *array* obtidos da Tabela 4, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias ALUT para a versão *array* o resultado empírico de tempo de execução é próximo a complexidade teórica n^2 , o que pode ser verificado no gráfico exibido na Figura 9, que faz a comparação do tempo real de execução com a complexidade teórica.

**Comparação tempo real e complexidade teórica
ALUT com array**

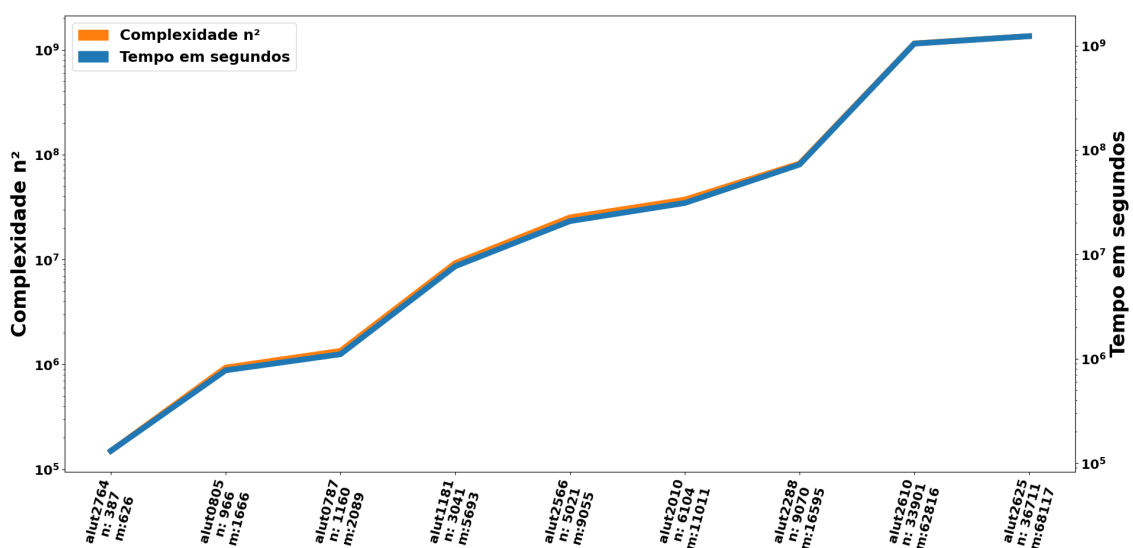


Figura 09. Gráfico de instâncias ALUT com comparação tempo real com a versão *array* e complexidade teórica.

Com os resultados da versão *heap* obtidos da Tabela 4, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias ALUT para a versão *heap* o resultado empírico de tempo de execução é próximo a complexidade teórica $m \log m$, o que pode ser verificado no gráfico exibido na Figura 10, que faz a comparação do tempo real de execução com a complexidade teórica.

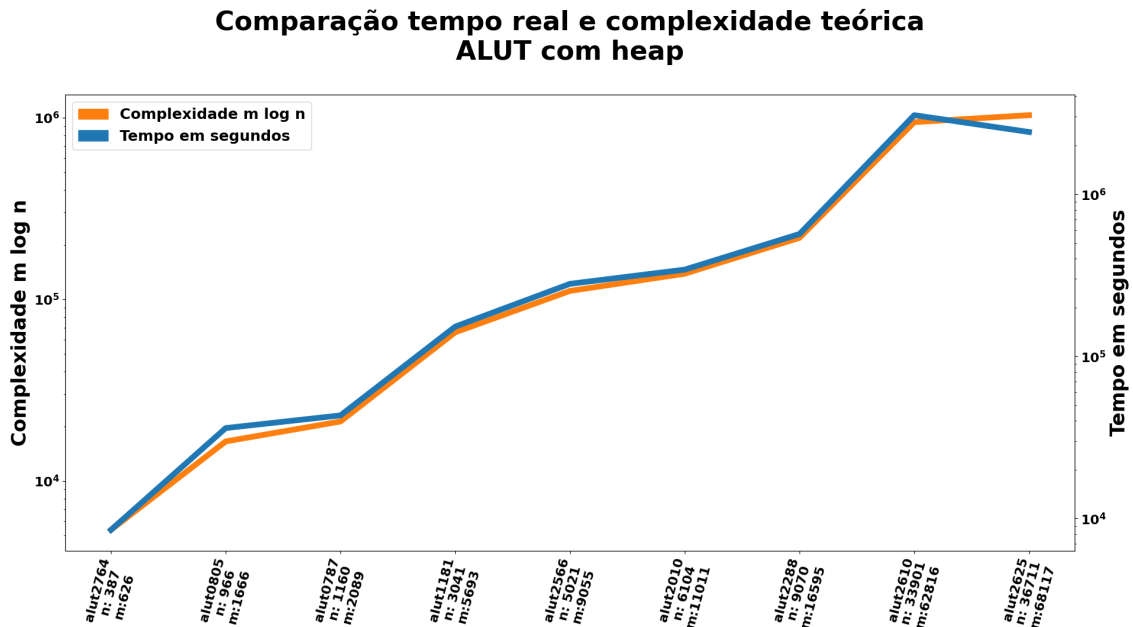


Figura 10. Gráfico de instâncias ALUT com comparação tempo real com a versão *heap* e complexidade teórica.

6.3 Instâncias DMXA

As características sobre nós e arestas das instâncias DMXA são mostradas na Tabela 5.

Tabela 5 - Características de nós e arestas das instâncias DMXA

Instâncias	Nós	Arestas
dmxa0628	169	280
dmxa0296	233	386
dmxa1304	298	503
dmxa1109	343	559
dmxa0848	499	861
dmxa0903	632	1.087
dmxa0734	663	1.154
dmxa1516	720	1.269

dmxa1200	770	1.383
dmxa1721	1.005	1.731
dmxa0454	1.848	3.226
dmxa0368	2.050	5.765
dmxa1801	2.333	4.137
dmxa1010	3.083	7.108

A Tabela 6 exibe os mínimos para tempo de execução das versões *array* e *heap* em nanosegundos (ns) do algoritmo de estudo para as instâncias DMXA. Para fazer o estudo do comportamento empírico e analisar a complexidade e o crescimento do tempo de processamento, foram gerados gráficos a partir dos valores dos mínimos dos tempos para as versões do algoritmo *array* e *heap* obtidos da Tabela 6. Os gráficos são apresentados a seguir.

Tabela 6 – Tempo mínimo de execução das instâncias DMXA

Instâncias	Array - tempo mínimo (ns)	Heap - tempo mínimo (ns)
dmxa0628	28.500	3.400
dmxa0296	51.800	4.600
dmxa1304	81.400	6.500
dmxa1109	106.100	8.400
dmxa0848	214.900	14.900
dmxa0903	337.800	20.400
dmxa0734	369.200	22.100
dmxa1516	432.900	24.700
dmxa1200	496.200	27.400
dmxa1721	844.900	38.400
dmxa0454	2.849.200	88.200
dmxa0368	3.470.100	92.200
dmxa1801	4.524.000	116.300
dmxa1010	13.222.400	204.300

Com os resultados da versão *array* obtidos da Tabela 6, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias DMXA para a versão *array* o resultado empírico de tempo de execução é próximo a complexidade teórica n^2 , o que pode ser verificado no gráfico exibido na Figura 11, que faz a comparação do tempo real de execução com a complexidade teórica.

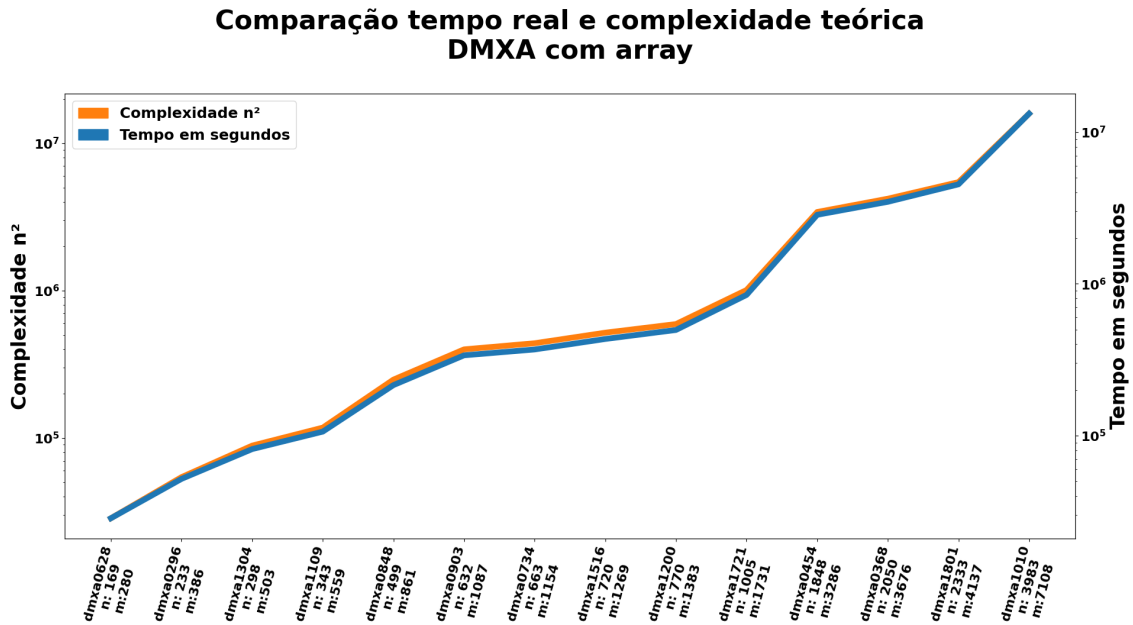


Figura 11. Gráfico de instâncias DMXA com comparação tempo real com a versão *array* e complexidade teórica.

Com os resultados da versão *heap* obtidos da Tabela 6, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias DMXA para a versão *heap* o resultado empírico de tempo de execução é próximo a complexidade teórica $m \log m$, o que pode ser verificado no gráfico exibido na Figura 12, que faz a comparação do tempo real de execução com a complexidade teórica.

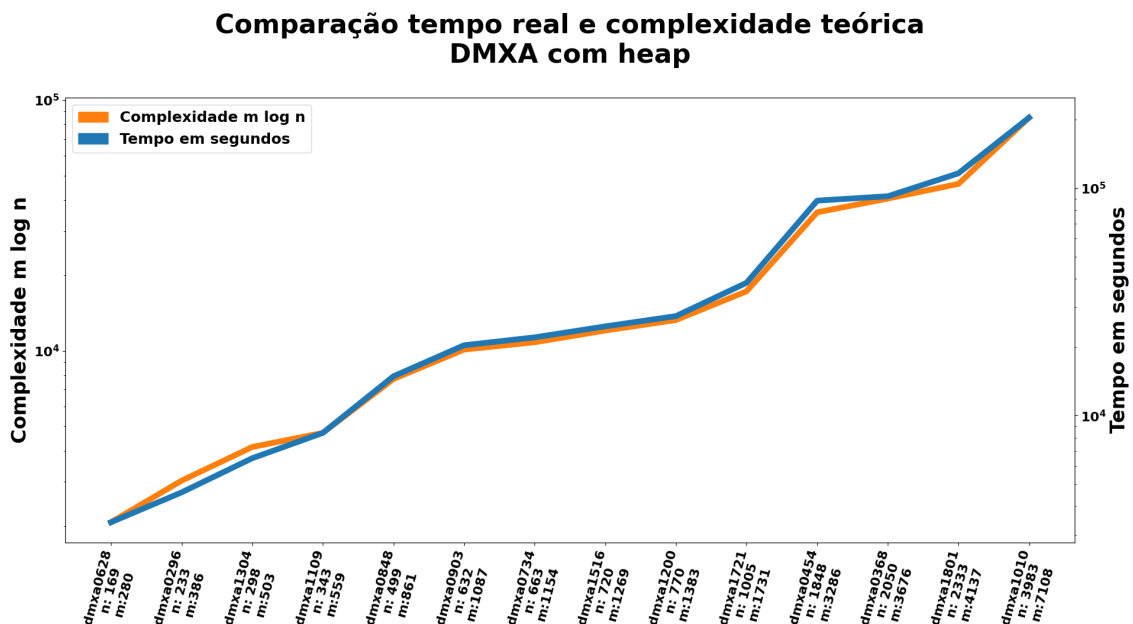


Figura 12. Gráfico de instâncias DMXA com comparação tempo real com a versão *heap* e complexidade teórica.

6.4 Instâncias test_set1

A tabela 7 mostra o número de nós e arcos para cada uma das instâncias de test_set1. No caso destas instâncias elas são grafos completos e simétricos. Não aparece nesta tabela a instância check_v5_s1, pois ela é destinada para validar a corretude do algoritmo. Realizamos uma representação da instância check_v5_s1 que é mostrada na Figura 13 e pode ser testada através do link:

<https://graphonline.ru/pt/?graph=IhbkgxGURktXZVtO>.

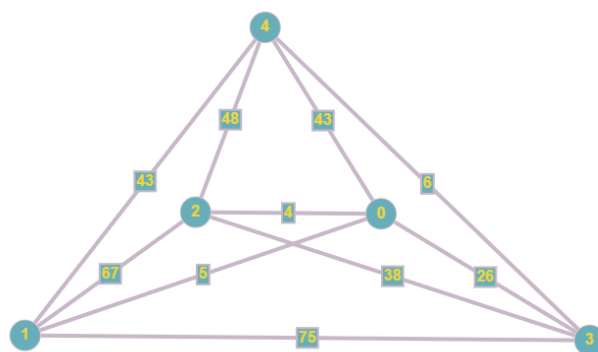


Figura 13. Representação da instância check_v5_s1.

Tabela 7 - Características de nós e arcos das instâncias test_set1

Instâncias	Nós	Arcos
inst_v100_s1	100	9.900
inst_v200_s1	200	39.800
inst_v300_s1	300	89.700
inst_v400_s1	400	159.600
inst_v500_s1	500	249.500
inst_v600_s1	600	359.400
inst_v700_s1	700	489.300
inst_v800_s1	800	639.200
inst_v900_s1	900	809.100
inst_v1000_s1	1.000	999.000

A Tabela 8 exibe os mínimos para tempo de execução das versões *array* e *heap* em nanosegundos (ns) do algoritmo de estudo para instâncias *test_set1*.

Tabela 8 - Tempo mínimo de execução das instâncias *test_set1*.

Instâncias	Array - tempo mínimo (ns)	Heap - tempo mínimo (ns)
inst_v100_s1	52.400	4.600
inst_v200_s1	301.300	285.100
inst_v300_s1	762.500	723.900
inst_v400_s1	1.403.600	1.276.800
inst_v500_s1	2.269.900	2.625.300
inst_v600_s1	4.596.800	4.613.000
inst_v700_s1	13.756.000	13.271.000
inst_v800_s1	19.484.200	18.588.300
inst_v900_s1	21.077.200	24.991.100
inst_v1000_s1	25.933.100	32.364.300

Diferente das outras classes de instâncias apresentadas anteriormente, para o caso de instâncias de *test_set1* ambas as versões do algoritmo, *array* e *heap*, apresentam tempo de processamento próximos entre as duas versões, *array* e *heap* (Tabela 8).

Com os resultados da versão *array* obtidos da Tabela 8, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias *test_set1* para a versão *array* o resultado empírico de tempo de execução é próximo a complexidade teórica n^2 , o que pode ser verificado no gráfico exibido na Figura 14, que faz a comparação do tempo real de execução com a complexidade teórica.

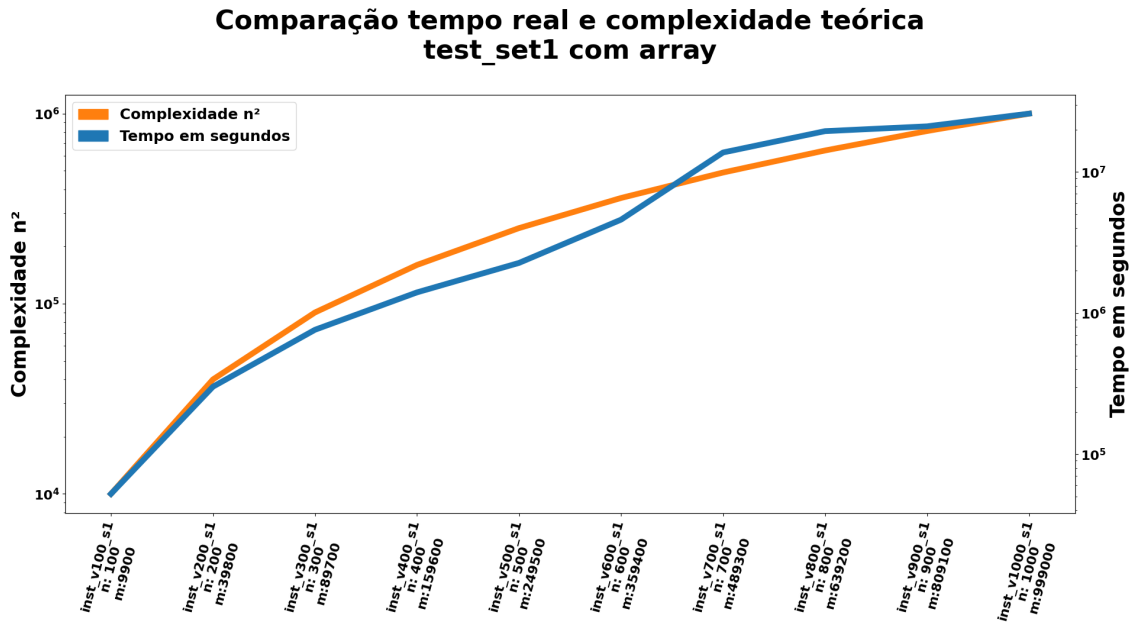


Figura 14. Gráfico de instâncias test_set1 com comparação tempo real com a versão *array* e complexidade teórica.

Com os resultados da versão *heap* obtidos da Tabela 8, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias test_set1 para a versão *heap* o resultado empírico de tempo de execução é próximo a complexidade teórica $m \log m$, o que pode ser verificado no gráfico exibido na Figura 15, que faz a comparação do tempo real de execução com a complexidade teórica.

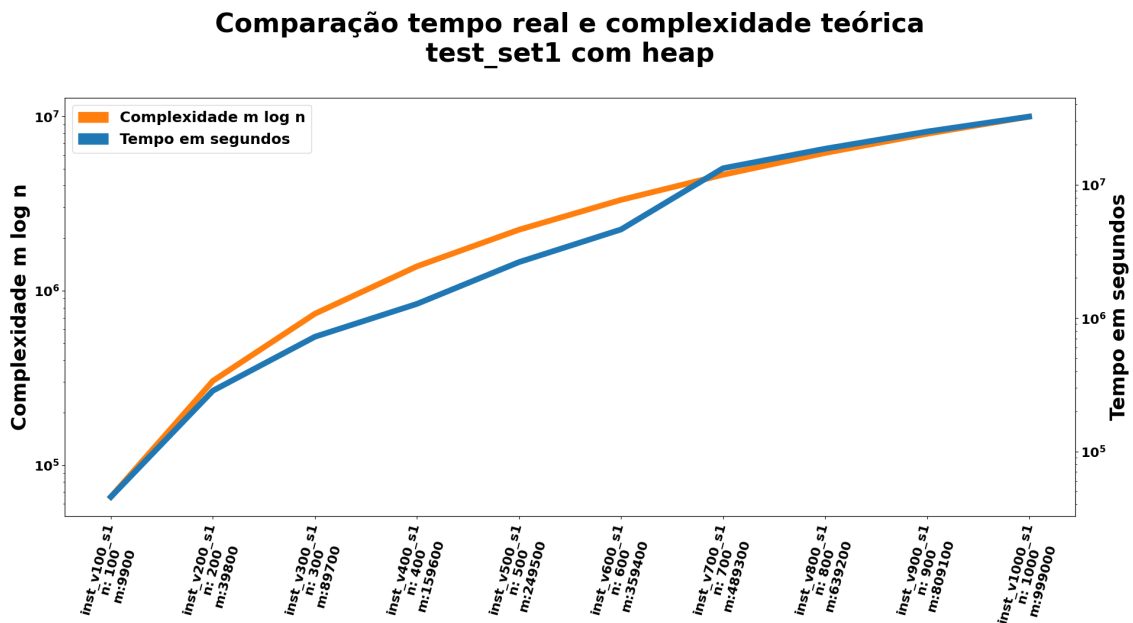


Figura 15. Gráfico de instâncias test_set1 com comparação tempo real com a versão *heap* e complexidade teórica.

6.5 Instâncias test_set2

A tabela 9 mostra o número de nós e arcos para cada uma das instâncias de test_set2. No caso destas instâncias elas são grafos completos e assimétricos. Não aparece nesta tabela a instância check_v5_s2, pois ela é destinada para validar a corretude do algoritmo.

Tabela 9 – Características de nós e arcos das instâncias test_set2.

Instâncias	Nós	Arcos
inst_v100_s2	100	9.900
inst_v200_s2	200	39.800
inst_v300_s2	300	89.700
inst_v400_s2	400	159.600
inst_v500_s2	500	249.500
inst_v600_s2	600	359.400
inst_v700_s2	700	489.300
inst_v800_s2	800	639.200
inst_v900_s2	900	809.100
inst_v1000_s2	1000	999.000

A Tabela 10 exibe os mínimos para tempo de execução das versões *array* e *heap* em nanosegundos (ns) do algoritmo de estudo para instâncias test_set2.

Tabela 10 – Tempo mínimo de execução das instâncias test_set2.

Instâncias	Array - tempo mínimo (ns)	Heap - tempo mínimo (ns)
inst_v100_s2	54.900	49.700
inst_v200_s2	323.300	301.900
inst_v300_s2	709.600	688.000
inst_v400_s2	1.242.000	1.229.500
inst_v500_s2	2.440.500	2.376.700
inst_v600_s2	6.600.800	7.714.200
inst_v700_s2	9.575.800	11.294.400

inst_v800_s2	13.241.600	17.059.600
inst_v900_s2	18.903.800	26.806.200
inst_v1000_s2	24.024.800	34.306.200

Com os resultados da versão *array* obtidos da Tabela 10, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias test_set2 para a versão *array*, o resultado empírico de tempo de execução é próximo a complexidade teórica n^2 , o que pode ser verificado no gráfico exibido na Figura 16, que faz a comparação do tempo real de execução com a complexidade teórica.

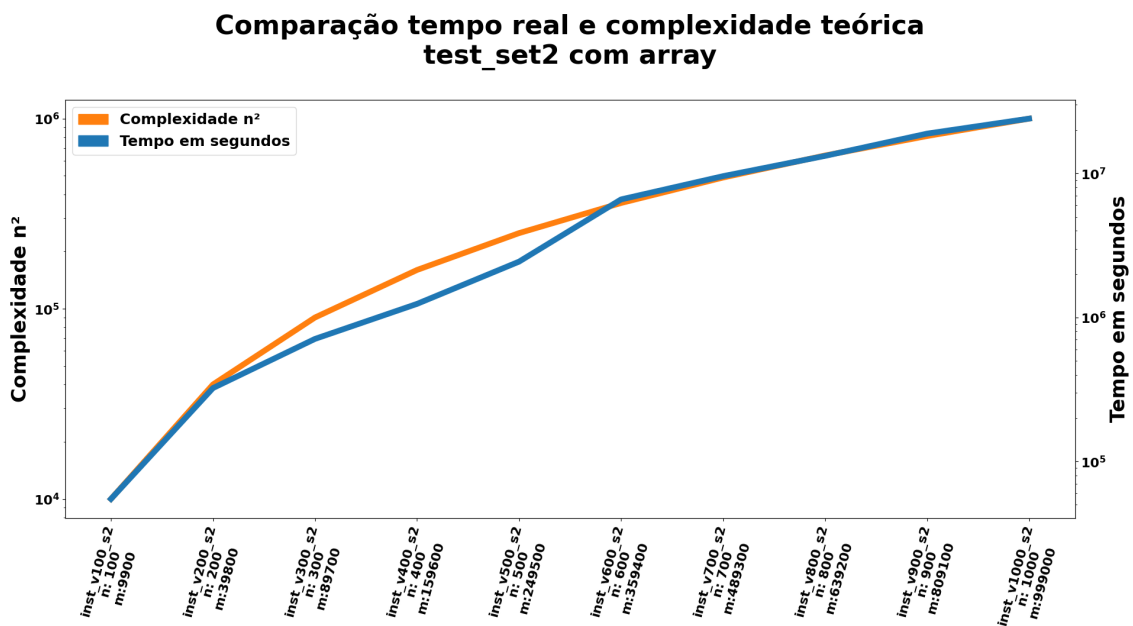


Figura 16. Gráfico de instâncias test_set2 com comparação tempo real com a versão *array* e complexidade teórica.

Com os resultados da versão *heap* obtidos da Tabela 10, foi realizada a análise de complexidade para essa versão. Chegou-se a conclusão que para instâncias test_set2 para a versão *heap* o resultado empírico de tempo de execução é próximo a complexidade teórica $m \log m$, o que pode ser verificado no gráfico exibido na Figura 17, que faz a comparação do tempo real de execução com a complexidade teórica.

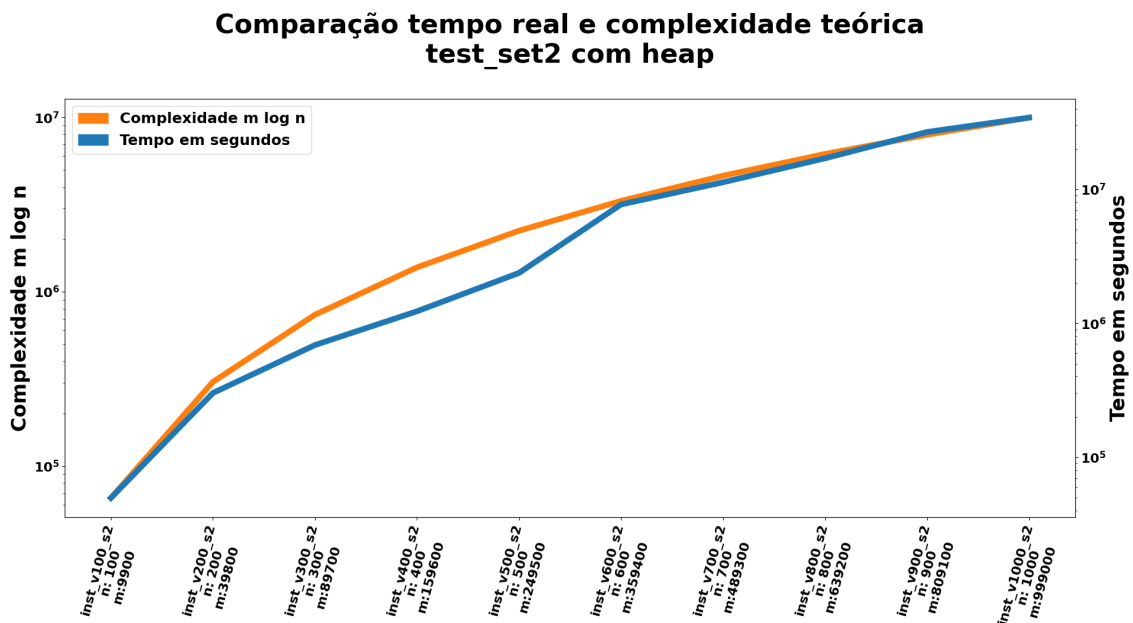


Figura 17. Gráfico de instâncias test_set1 com comparação tempo real com a versão *heap* e complexidade teórica.

7. Conclusão

Com os resultados experimentais obtidos a partir da implementação e com os recursos computacionais apresentados, há o indicativo de que, para as classes de instâncias ALUE, ALUT e DMXA, que possuem instâncias de grafos esparsos, a versão *array* do algoritmo de *Dijkstra* apresentou tempo de processamento próximo à complexidade teórica n^2 e a versão *heap* do algoritmo de *Dijkstra* apresentou tempo de processamento próximo à complexidade teórica $m \log n$, conforme esperado.

Para as classes de teste TEST_SET1 (grafos completos e simétricos) e TEST_SET2 (grafos completos e assimétricos), a versão *array* do algoritmo de *Dijkstra* apresentou tempo de processamento próximo à complexidade teórica n^2 e a versão *heap* do algoritmo de *Dijkstra* apresentou tempo de processamento próximo à complexidade teórica $m \log n$.

Referências

Ahuja, R., et al. *"Network Flows. Theory, Algorithms and Applications"*. Prentice hall, 1993.

Chen, J., Revels, J. "Robust benchmarking in noisy environments".
<https://arxiv.org/abs/1608.04295>

Cormen, T. "Desmistificando algoritmos. Vol. 1". Elsevier Brasil, 2017.

Duch, A. "Análisis de Algoritmos" Barcelona, Universidad Politécnica de Barcelona, 2007.

Gillespie, T.; *"The relevance of algorithms"*, livro: *"Media Technologies: Essays on Communication, Materiality, and Society"*. MIT Press, 2014.

Knuth, Donald E. *"An empirical study of Fortran programs Software: Practice and experience 1.2. 1971.*

Von Atzingen, J., et al. "Análise comparativa de algoritmos eficientes para o problema de caminho mínimo." Universidade de São Paulo (USP). São Paulo, Escola Politécnica, 2012.

Ziviani, N. "Projeto de Algoritmos Com Implementações Em Pascal e C". Editora: Pioneira Thomson Learning, 2004.