

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

Alunos

Caio Souza
Emmanuel Gomes
Fabrício Janssen
Marcelo Simas

Disciplina

Análise e Projeto de Algoritmos

Professora

Adriana Cesário de Faria Alvim

Introdução, Problema e Solução

- Um **algoritmo** é uma sequência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema (*Ziviani, 2004*)
- 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. (*Duch, 2007*)
- O problema de **caminho mínimo** é um dos problemas genéricos mais estudados. Dada uma rede, composta de nós (ou vértices) e arcos (ou ligações) entre esses nós, o problema é 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.
- Foram implementadas **duas** versões: vetor (*array*) e fila de prioridades (*heap* binário).
- A análise de complexidade empírica realizada com **cinco** classes de instâncias.

Implementação - Armazenamento em Array



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;

```
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);
}
```

Implementação - Armazenamento em Heap



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*(*value*, *i*, *H*);

end;

end;

end;

```
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);
}
```

O ambiente computacional utilizado

- Benchmark: Função com previsão de nanosegundos e monotônica.
- Processador: Intel i7 3.20GHz. Arquitetura 64 bits.
- Programa: rodando em um único core
- Cache: L1 384KB. L2 1,5MB. L3 12,0MB.
- Memória RAM: 32GB.
- Sistema operacional: Linux Ubuntu/WSL 5.10.16.3-microsoft-standard-WSL2.
- Programação e compilação: C com gcc. Versão 9.3.0. Compilado com flag de otimização "-O3".

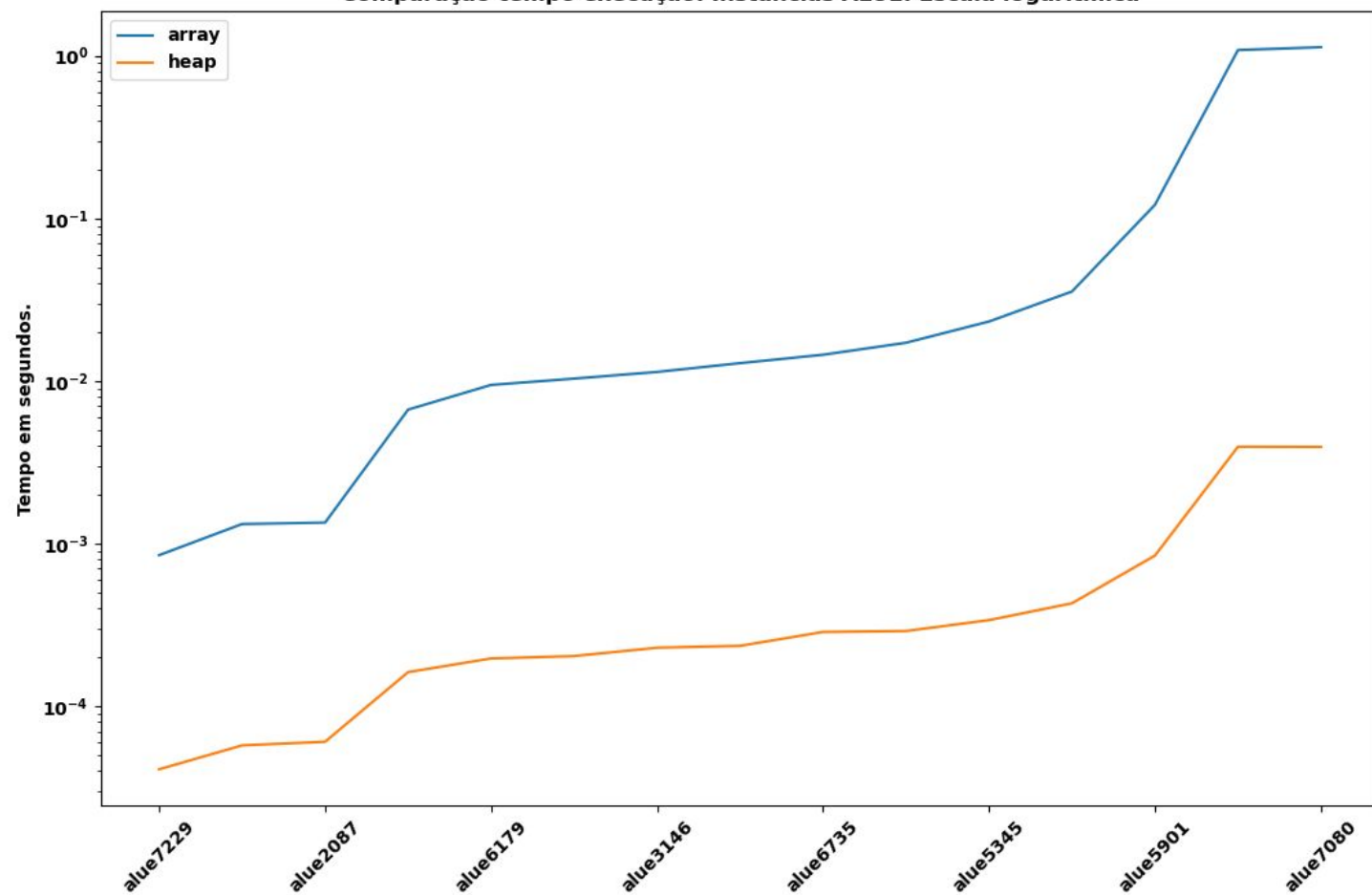
Rotina para medição do tempo

- 
- A medição do tempo se fez da seguinte forma:
 - Inicialmente o algoritmo é executado 10 vezes para fazer o “warmup” da CPU conforme o algoritmo. Esse warmup é necessário para a CPU melhor prever o caminho da execução do algoritmo, preencher cache, dentre outros.
 - Após este “warmup”, o algoritmo foi executado 20 vezes.
 - Em cada execução, foi armazenado o tempo preciso (em nanosegundos) em que ela iniciou e finalizou, ignorando a leitura de dados.

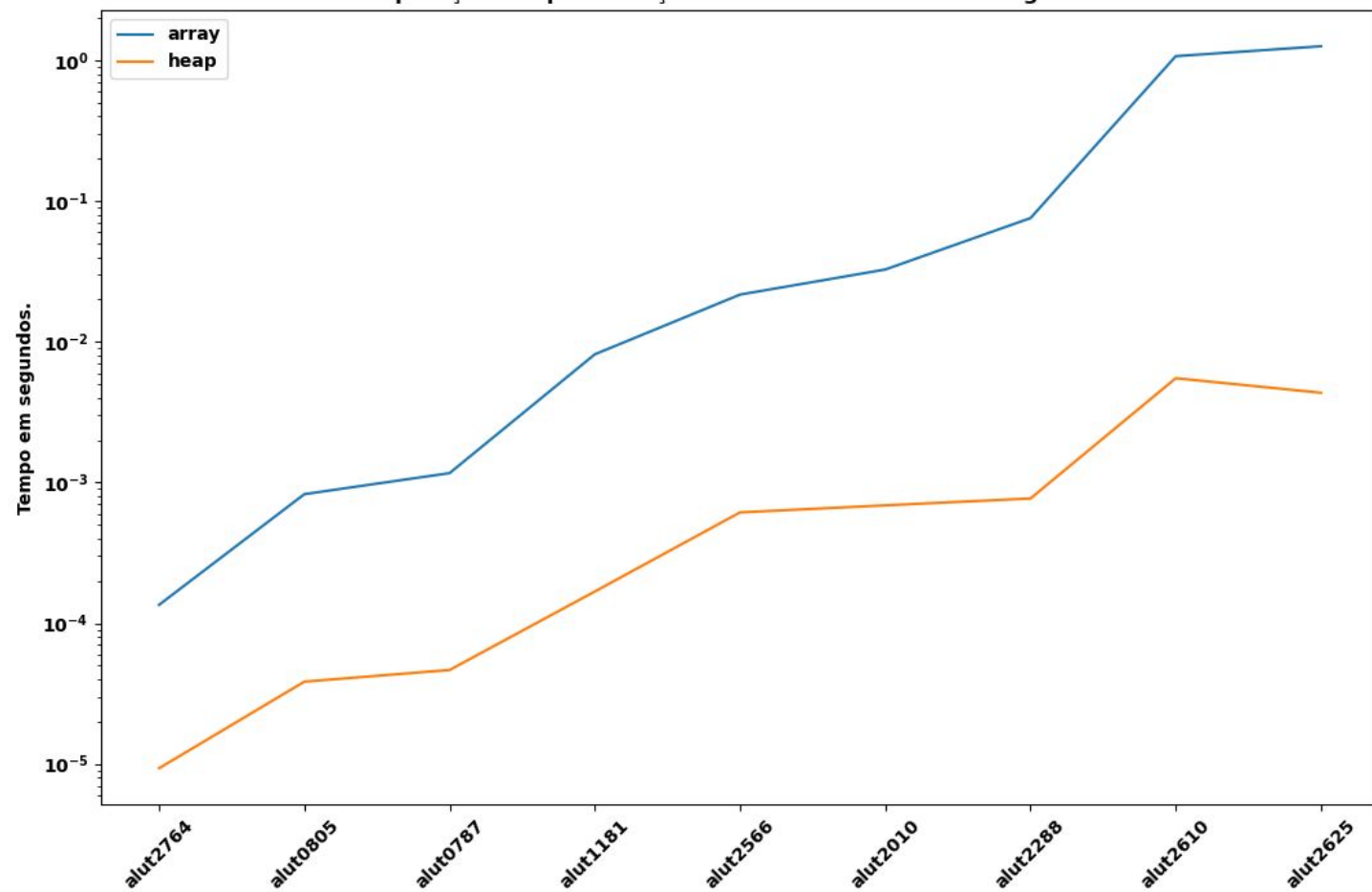
Complexidade “indeterminada” do free

```
LinkedListNode LL_remove_node(LinkedList self, LinkedListNode prev_node, LinkedListNode node) {  
    if (node) {  
        LinkedListNode next = node->next;  
  
        if (prev_node) prev_node->next = node->next;  
        else self->first = node->next;  
  
        #ifndef NO_FREE  
            free(node);  
        #endif  
  
        self->len--;  
  
        return next;  
    }  
    return NULL;  
}
```

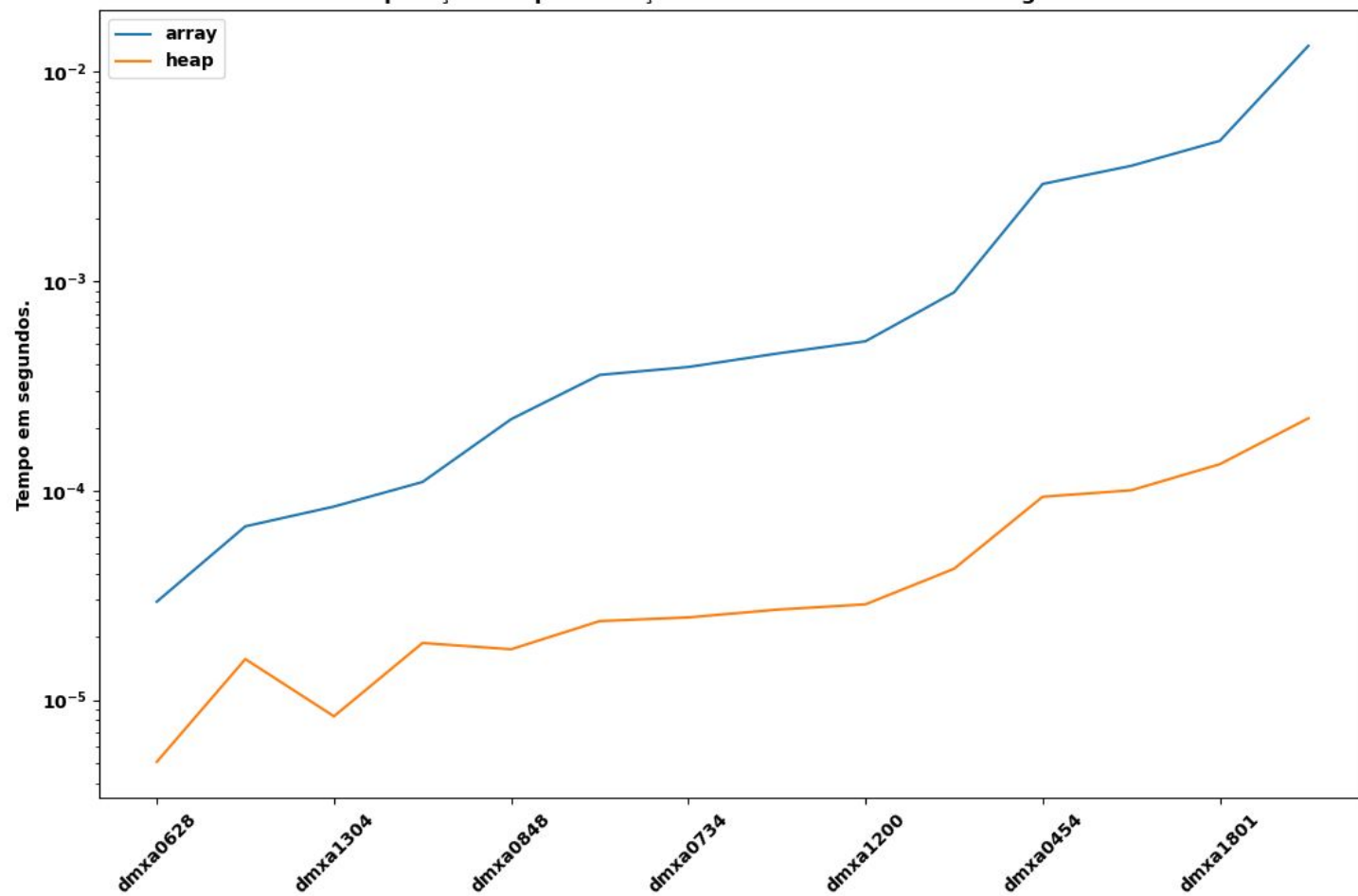
Comparação tempo execução. Instâncias ALUE. Escala logarítmica



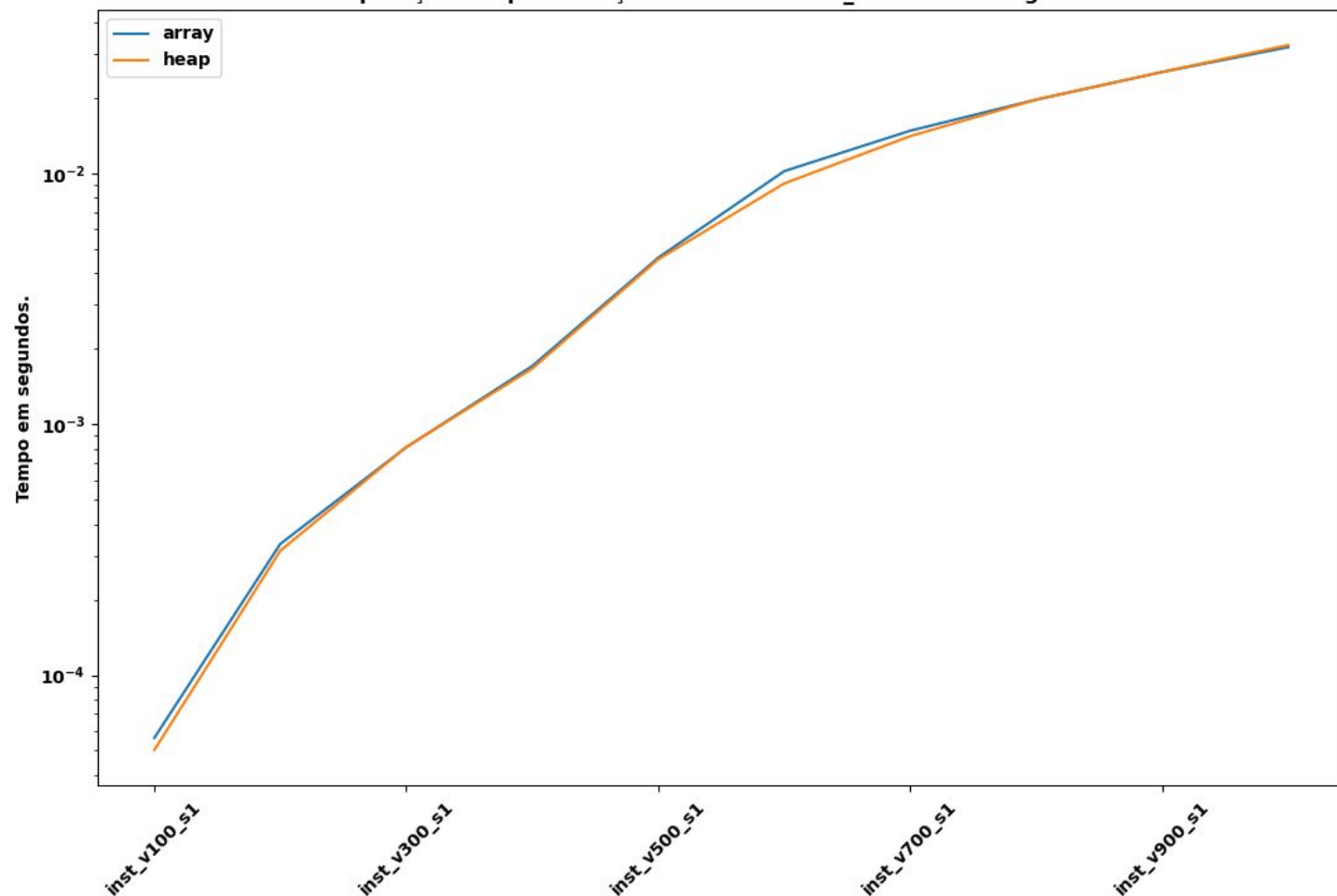
Comparação tempo execução. Instâncias ALUT. Escala logarítmica



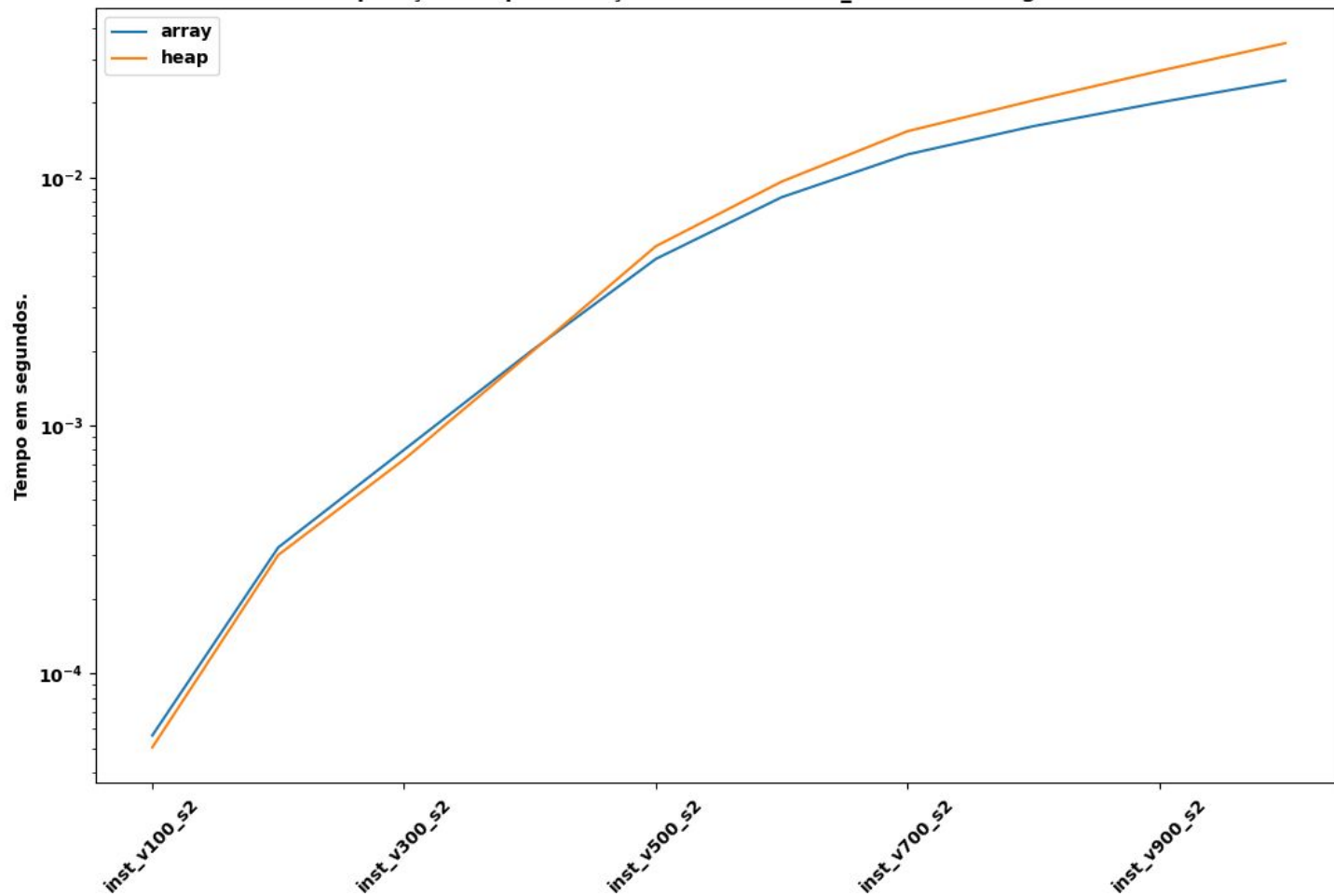
Comparação tempo execução. Instâncias DMXA. Escala logarítmica

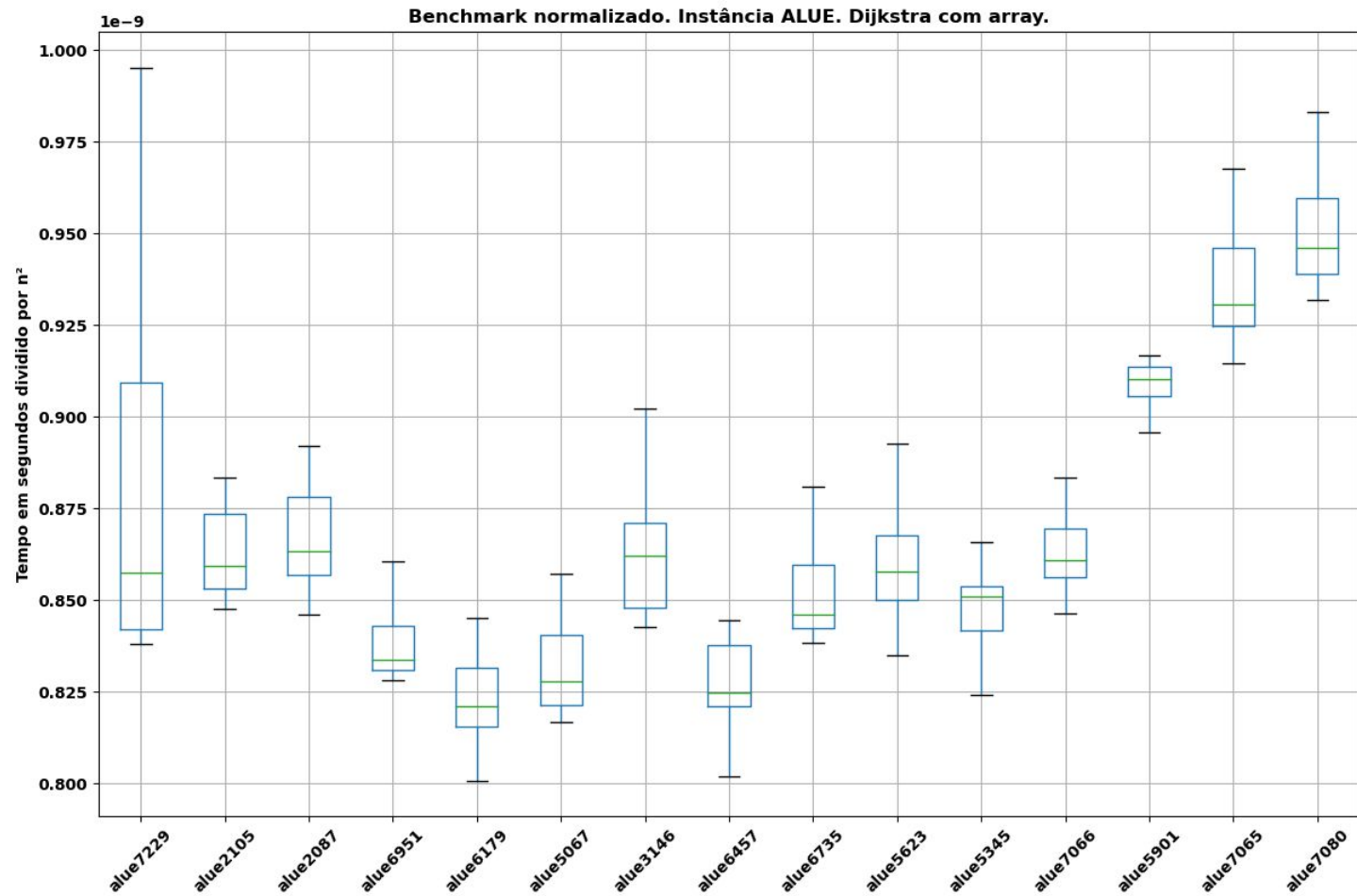


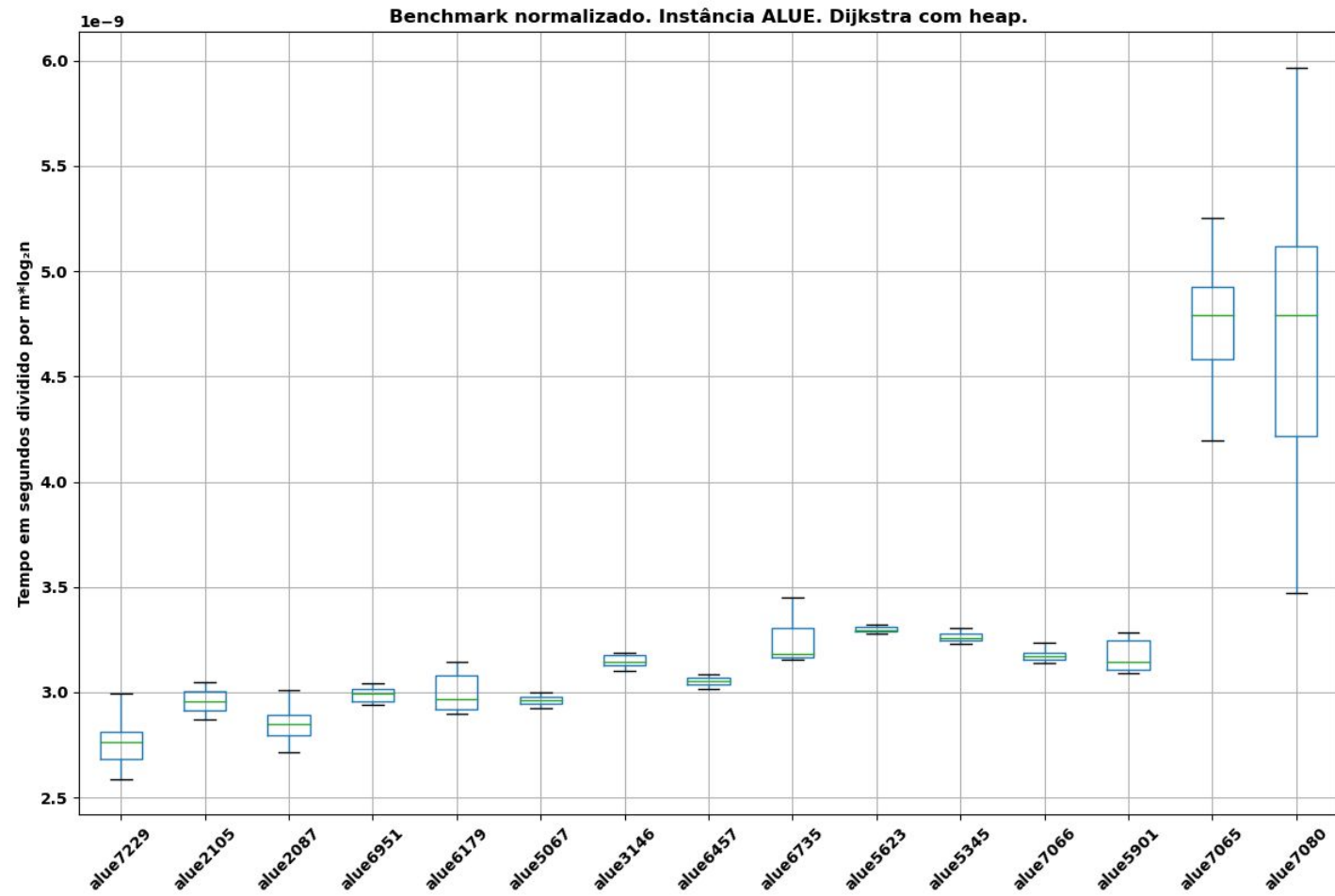
Comparação tempo execução. Instâncias test_set1. Escala logarítmica



Comparação tempo execução. Instâncias test_set2. Escala logarítmica







Conclusão

A partir dos resultados experimentais de benchmark concluímos:

- As classes de instâncias ALUE, ALUT e DMXA tiveram um tempo de processamento melhor com a implementação com heap.
- As classes de instâncias TEST_SET1 e TEST_SET2, tiveram tempo de processamento semelhante entre as implementações, apesar da com heap ter melhor complexidade.
- Recomenda-se entender o tipo de grafo a ser processado para recomendar o algoritmo.

Bibliografia



- Ahuja, R., et al. "Network Flows. Theory, Algorithms and Applications". Prentice hall, (1993).
- 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.
- Implementação do algoritmo: <https://github.com/caiopsouza/dijkstra>.
- 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.