Universidade Federal do Espírito Santo

# Algoritmos de Caminhos Mínimos: Dijkstra e Floyd


Eduarda Coppo - eduardacoppopkm@gmail.com

Pedro Henrique Vieira de Oliveira Azevedo - pedro.hvo.azevedo@gmail.com


Teoria dos Grafos

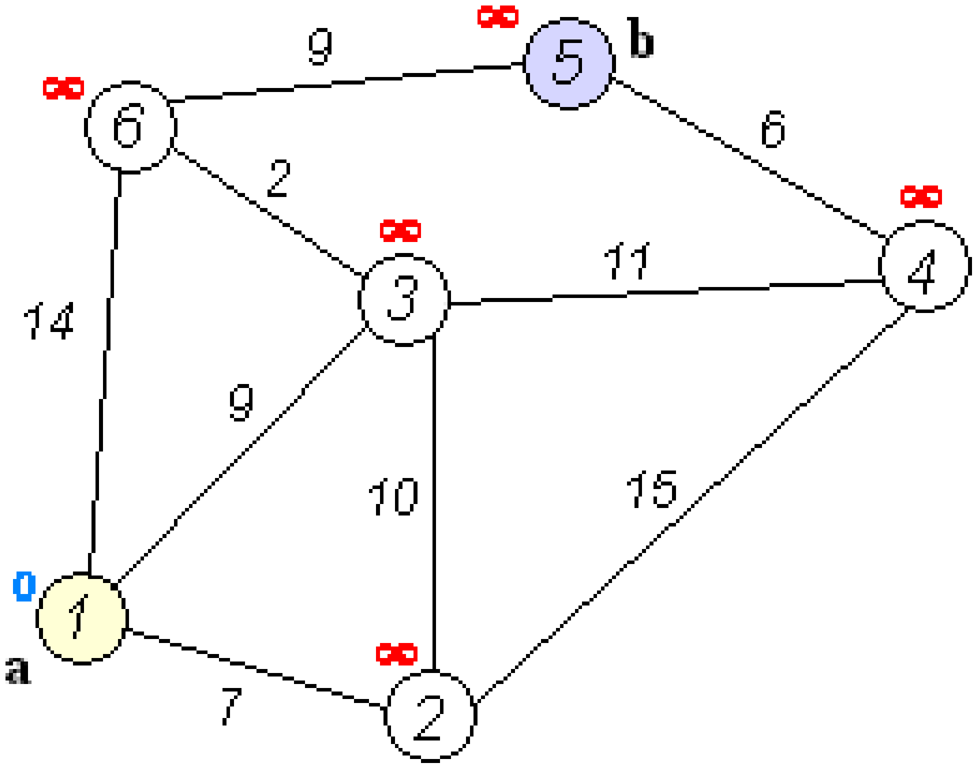
# Introdução

- Minimizar o custo de travessia de um grafo entre dois nós $v$ e $w$.

- Os algoritmos que iremos apresentar possuem características diferentes para a solucionar o problema do caminho mínimo:
  - Algoritmo de Dijkstra:
    - De um vértice $v$ até um vértice $w$;
    - De um vértice $v$ para todos os vértices do grafo;
    - Para todos os pares de vértices.
  - Algoritmo de Floyd:
    - Para todos os pares de vértices.

# Algoritmo de Dijkstra

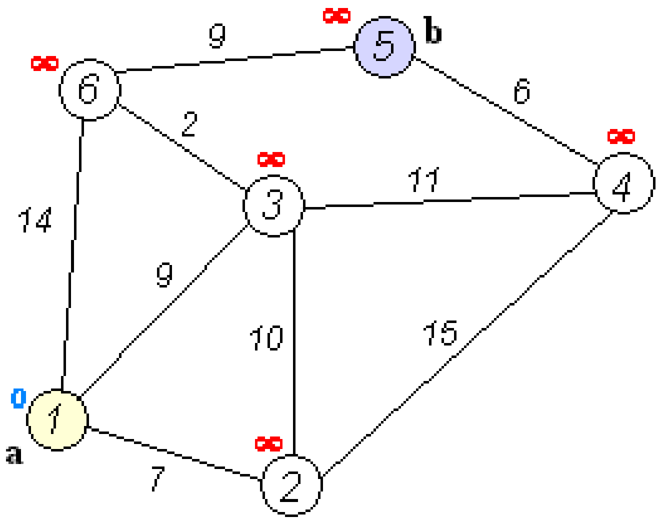
- É hora da revisão!



# Algoritmo de Dijkstra – Representação do Grafo

```
struct
type_graph{
    int vertex;
    int cost;
    struct
type_graph
*prox;
};
```

```
t_graph** add_to_list_undir(t_graph **adjacent_list,
int u, int v, int w){
    t_graph *c,  *p;
    c = new_node;
    c->vertex = v;  c->cost = w; c->prox = NULL;
    p = adjacent_list[u];
    while ( p -> prox != NULL ){
        p = p -> prox;
    }
    p -> prox = c;
    return (adjacent_list);
}
```




| | | | | | |
|---|---|---|---|---|---|
| 1 | -> | 2 \| 7 | 3 \| 9 | 6 \| 14 | |
| 2 | -> | 1 \| 7 | 3 \| 10 | 4 \| 15 | |
| 3 | -> | 1 \| 9 | 2 \| 10 | 4 \|11 | 6 \| 2 |
| 4 | -> | 2 \| 15 | 3 \| 11 | 5 \| 6 | |
| 5 | -> | 4 \| 6 | 6 \| 9 | | |
| 6 | -> | 1 \| 14 | 3 \| 2 | 5 \| 9 | |

# Algoritmo de Dijkstra – Forma Canônica

```
typedef struct{
    int *distancia;
    int *anterior;
    int *fechado;
}t_graph_info;
```

```
t_graph_info dijkstra_array (t_graph** adjacent_list, int graph_size, int vertex_ini, int
vertex_end){
    INICIALIZAÇÕES DE r.distancia, r.anterior, r.fechado
    while (k!=vertex_end){
        for (i=0; i<graph_size; i++){
            if(aberto[i]==1 && r.distancia[i]<menor){
                menor = r.distancia[i];  k=i;
            }
        }
        r.fechado[k] = 1;  t_graph* p;
        for(p = adjacent_list[k]; p!=NULL; p = p->prox){
            if(r.fechado[p->vertex]!=1){
                custo = MIN (r.distancia[p->vertex], (r.distancia[k]+p->cost));
                if(custo < r.distancia[p->vertex]){
                    r.distancia[p->vertex] = custo; r.anterior[p->vertex] = k;
                }
            }
        }   }   }
```

# Algoritmo de Dijkstra – Forma Canônica

```
typedef struct{
   int *distancia;
   int *anterior;
   int *fechado;
}t_graph_info;
```

```
t_graph_info dijkstra_array (t_graph** adjacent_list, int graph_size, int vertex_ini, int
vertex_end){
   INICIALIZAÇÕES DE r.distancia, r.anterior, r.fechado
   while (k!=vertex_end){
      for (i=0; i<graph_size; i++){
         if(aberto[i]==1 && r.distancia[i]<menor){
            menor = r.distancia[i];  k=i;
         }
      }
      r.fechado[k] = 1;  t_graph* p;
      for(p = adjacent_list[k]; p!=NULL; p = p->prox){
         if(r.fechado[p->vertex]!=1){
            custo = MIN (r.distancia[p->vertex], (r.distancia[k]+p->cost));
            if(custo < r.distancia[p->vertex]){
               r.distancia[p->vertex] = custo; r.anterior[p->vertex] = k;
            }
         }
      }   }  }
```

**GARGALO!** $O(n)$

# Algoritmo de Dijkstra – Forma Canônica

```
typedef struct{
   int *distancia;
   int *anterior;
   int *fechado;
}t_graph_info;
```

```
t_graph_info dijkstra_array (t_graph** adjacent_list, int graph_size, int vertex_ini, int
vertex_end){
   INICIALIZAÇÕES DE r.distancia, r.anterior, r.fechado
   while (k!=vertex_end){
      for (i=0; i<graph_size; i++){
         if(aberto[i]==1 && r.distancia[i]<menor){
            menor = r.distancia[i];  k=i;
         }
      }
      r.fechado[k] = 1;  t_graph* p;
      for(p = adjacent_list[k]; p!=NULL; p = p->prox){
         if(r.fechado[p->vertex]!=1){
            custo = MIN (r.distancia[p->vertex], (r.distancia[k]+p->cost));
            if(custo < r.distancia[p->vertex]){
               r.distancia[p->vertex] = custo; r.anterior[p->vertex] = k;
            }
         }
      }   }  }
```

$O(m), m \leq n$

GARGALO! $O(n)$

# Algoritmo de Dijkstra – Forma Canônica

```
typedef struct{
    int *distancia;
    int *anterior;
    int *fechado;
}t_graph_info;
```

```
t_graph_info dijkstra_array (t_graph** adjacent_list, int graph_size, int vertex_ini, int
vertex_end){
    INICIALIZ
    while (k!
        for (i=(
            if(ab
                m
            }
        }
        r.fecha
        for(p = adjacent_list[k]; p!=NULL; p = p->prox){
            if(r.fechado[p->vertex]!=1){
                custo = MIN (r.distancia[p->vertex], (r.distancia[k]+p->cost));
                if(custo < r.distancia[p->vertex]){
                    r.distancia[p->vertex] = custo; r.anterior[p->vertex] = k;
                }
            }
}   }  }
```

$m \leq n$

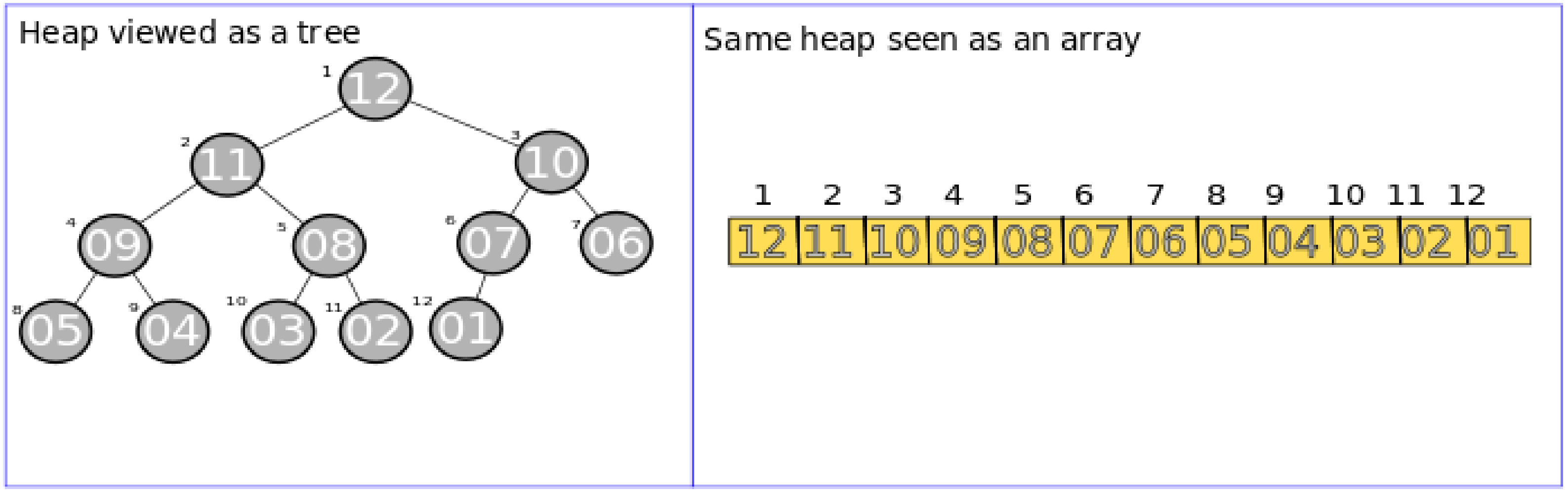
GALO! $O(n)$

**Complexidade Total:**

$$O(m * n)$$

# Algoritmo de Dijkstra – minHeap Binária

- Uma heap é uma árvore binária balanceada




Pai: $\frac{i-1}{2}$

Filho Esquerda: $(2 * i) + 1$

Filho Direita: $(2 * i) + 2$

- No algoritmo de Dijkstra substituímos o loop que busca a menor distância pela heap.
- O vértice com menor distância sempre estará no topo da heap juntamente com o identificador do vértice.

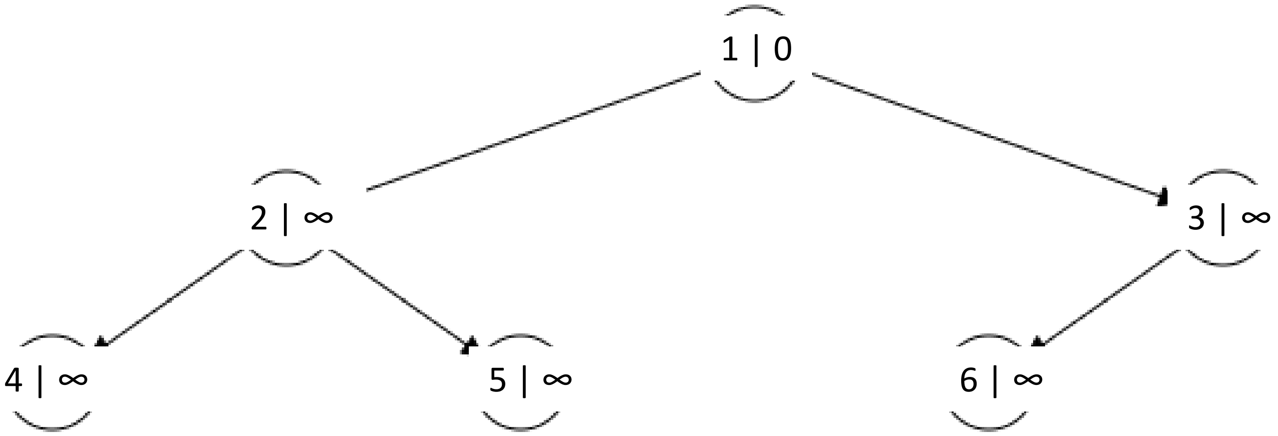
# Algoritmo de Dijkstra – minHeap Binária

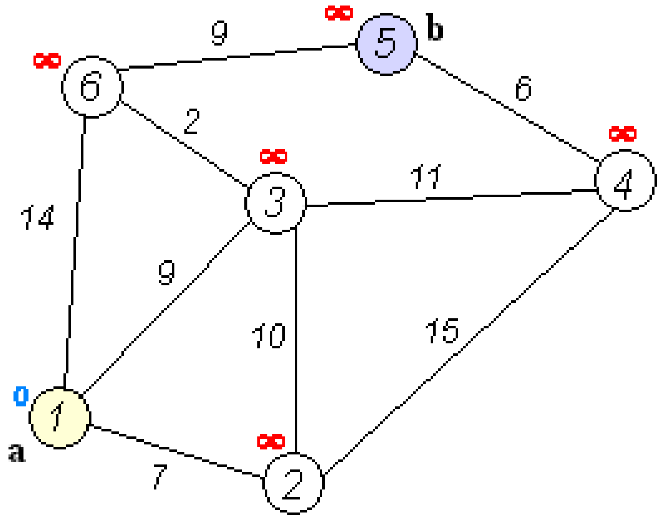
```
typedef struct{
    float key;
    int  dataIndex;
}HeapItem;

typedef struct{
HeapItem *H;
    int *map;
    int n;
    int size;
    int size_map;
}Heap;
```

```
t_graph_info dijkstra_heap (t_graph** adjacent_list, int graph_size, int vertex_ini, int
vertex_end){
    Heap *h = createHeap();
    for(i = 0; i < graph_size; i++){
        if(i == vertex_ini) insert(h,i,0);
        else insert(h,i,inf);
    }
    while (k!=vertex_end){
        k = removeMin(h);
        t_graph* p;
        for(p = adjacent_list[k]; p!=NULL; p = p->prox){
            dist_v1 = h->H[h->map[p->vertex]].key; dist_k = h->H[h->map[k]].key; edge_weight = p->cost;
            if(dist_v1 > (dist_k + edge_weight)){
                dist_v1 = dist_k + edge_weight;
                r.anterior[p->vertex] = k;
                changeKey(h,h->H[h->map[p->vertex]].dataIndex,dist_v1);
            }}}}
```

# Algoritmo de Dijkstra – minHeap Binária
# Inserção




| | | | | | |
|---|---|---|---|---|---|
| 1 \| 0 | 2\| ∞ | 3\| ∞ | 4\| ∞ | 5\| ∞ | 6\| ∞ |

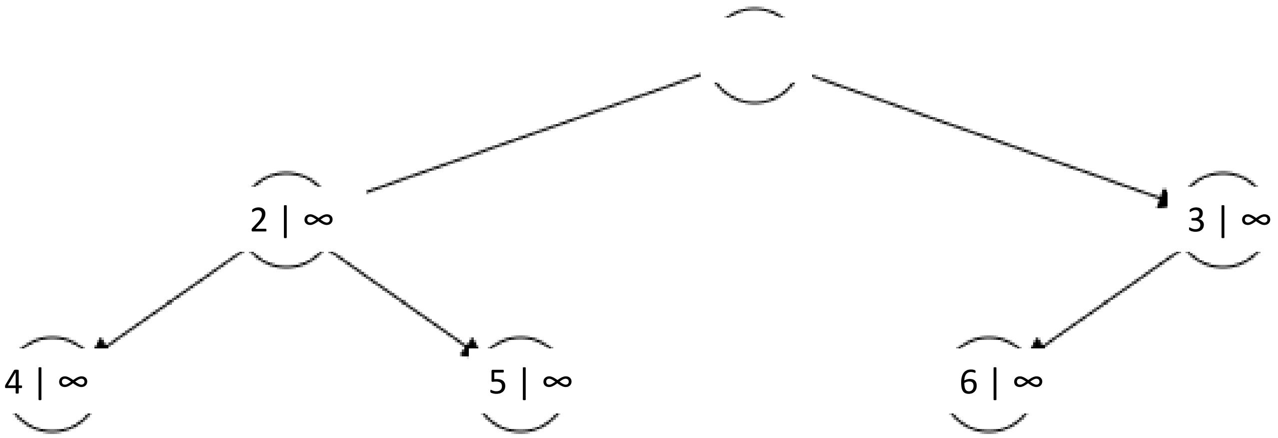
# Algoritmo de Dijkstra – minHeap Binária

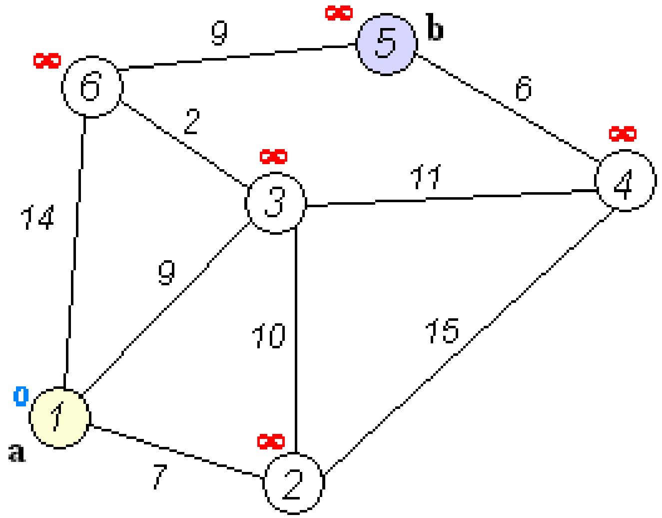
```
typedef struct{
    float key;
    int  dataIndex;
}HeapItem;

typedef struct{
HeapItem *H;
    int *map;
    int n;
    int size;
    int size_map;
}Heap;
```

```
t_graph_info dijkstra_heap (t_graph** adjacent_list, int graph_size, int vertex_ini, int
vertex_end){
    Heap *h = createHeap();
    for(i = 0; i < graph_size; i++){
        if(i == vertex_ini) insert(h,i,0);
        else insert(h,i,inf);
    }
    while (k!=vertex_end){
        k = removeMin(h);
        t_graph* p;
        for(p = adjacent_list[k]; p!=NULL; p = p->prox){
            dist_v1 = h->H[h->map[p->vertex]].key; dist_k = h->H[h->map[k]].key; edge_weight = p->cost;
            if(dist_v1 > (dist_k + edge_weight)){
                dist_v1 = dist_k + edge_weight;
                r.anterior[p->vertex] = k;
                changeKey(h,h->H[h->map[p->vertex]].dataIndex,dist_v1);
            }}}}
```

# Algoritmo de Dijkstra – minHeap Binária Remoção (Escolha do k)

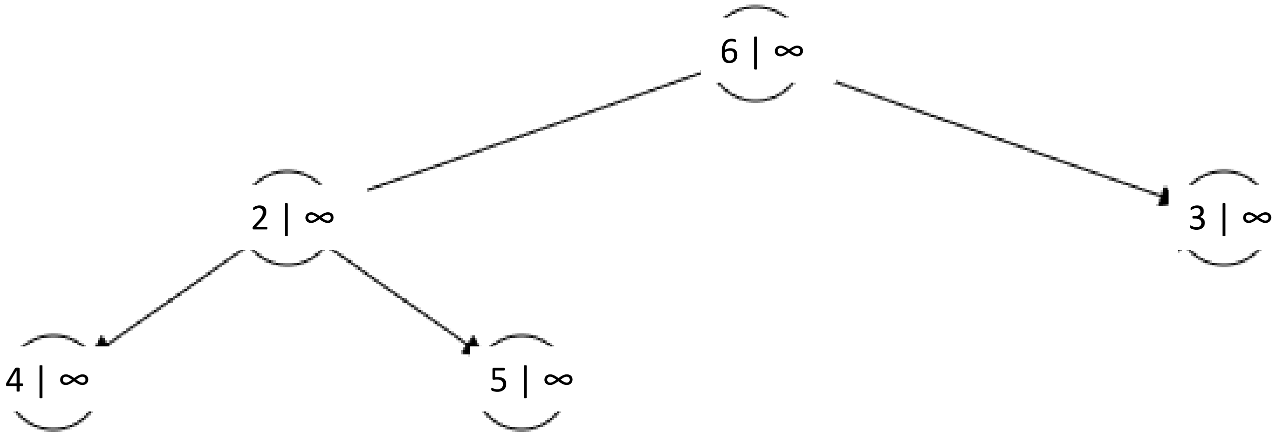


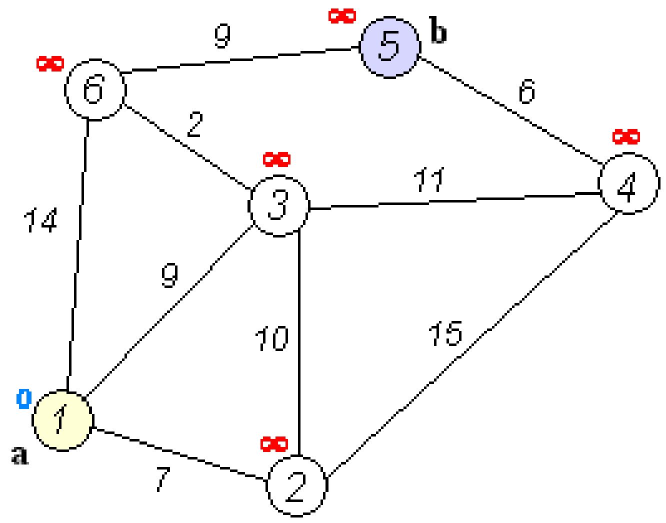




| | | | | | |
|---|---|---|---|---|---|
| | 2\| ∞ | 3\| ∞ | 4\| ∞ | 5\| ∞ | 6\| ∞ |

K = 1

# Algoritmo de Dijkstra – minHeap Binária Remoção (Escolha do k)




| | | | | | |
|---|---|---|---|---|---|
| 6\| ∞ | 2\| ∞ | 3\| ∞ | 4\| ∞ | 5\| ∞ | |

K = 1

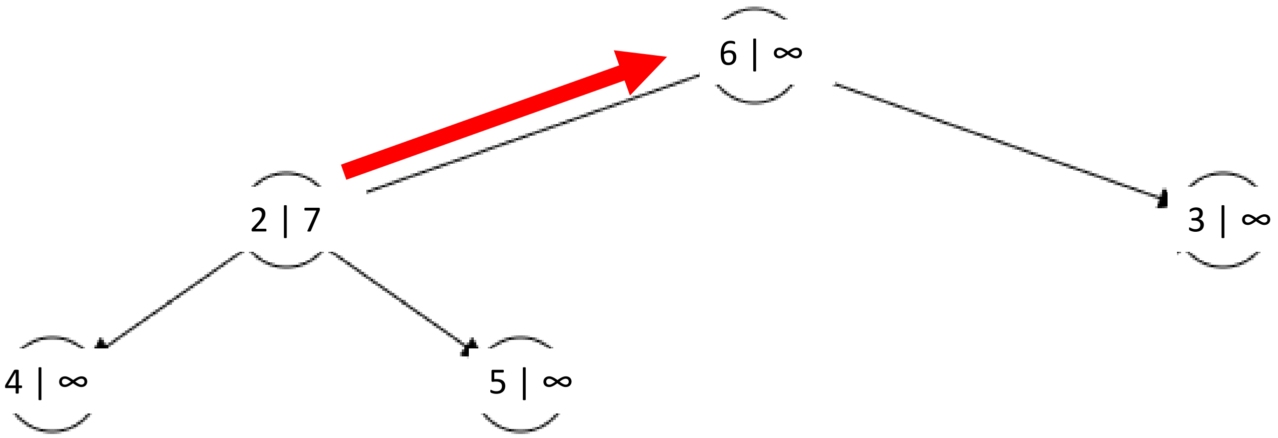
# Algoritmo de Dijkstra – minHeap Binária

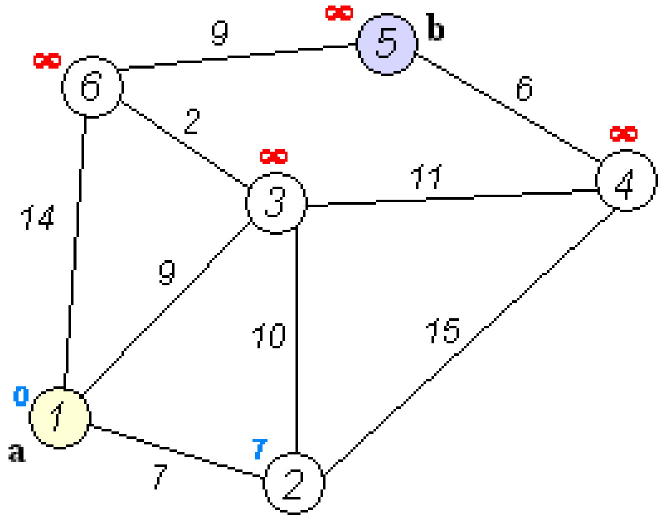
```
typedef struct{
    float key;
    int  dataIndex;
}HeapItem;

typedef struct{
HeapItem *H;
    int *map;
    int n;
    int size;
    int size_map;
}Heap;
```

```
t_graph_info dijkstra_heap (t_graph** adjacent_list, int graph_size, int vertex_ini, int
vertex_end){
    Heap *h = createHeap();
    for(i = 0; i < graph_size; i++){
        if(i == vertex_ini) insert(h,i,0);
        else insert(h,i,inf);
    }
    while (k!=vertex_end){
        k = removeMin(h);
        t_graph* p;
        for(p = adjacent_list[k]; p!=NULL; p = p->prox){
            dist_v1 = h->H[h->map[p->vertex]].key; dist_k = h->H[h->map[k]].key; edge_weight = p->cost;
            if(dist_v1 > (dist_k + edge_weight)){
                dist_v1 = dist_k + edge_weight;
                r.anterior[p->vertex] = k;
                changeKey(h,h->H[h->map[p->vertex]].dataIndex,dist_v1);
            }}}}
```

# Algoritmo de Dijkstra – minHeap Binária changeKey

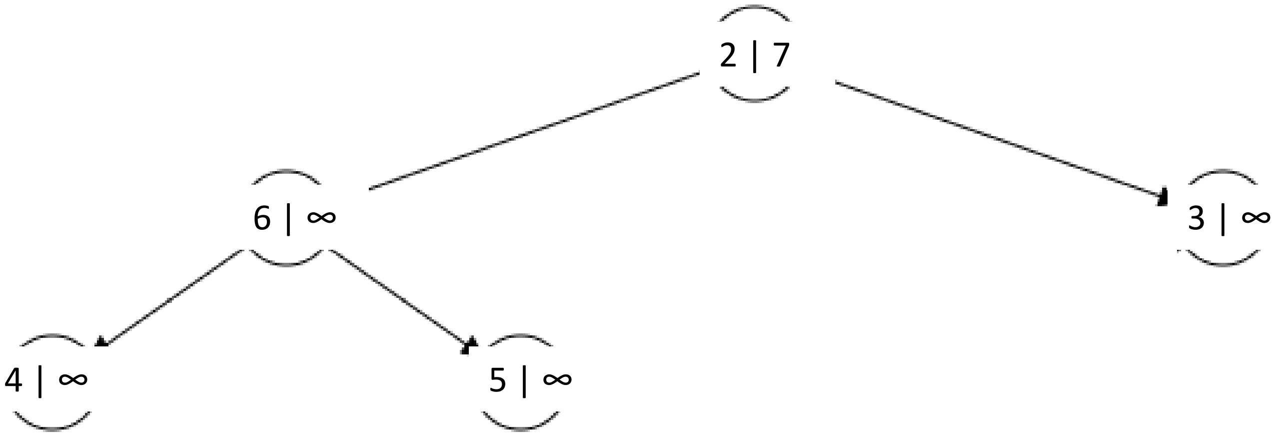


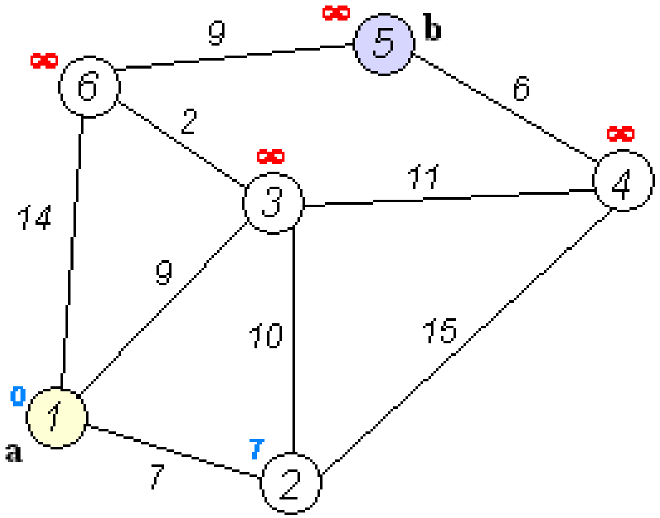

| | | | | | |
|---|---|---|---|---|---|
| 6\| ∞ | 2\| 7 | 3\| ∞ | 4\| ∞ | 5\| ∞ | |

K = 1

# Algoritmo de Dijkstra – minHeap Binária changeKey

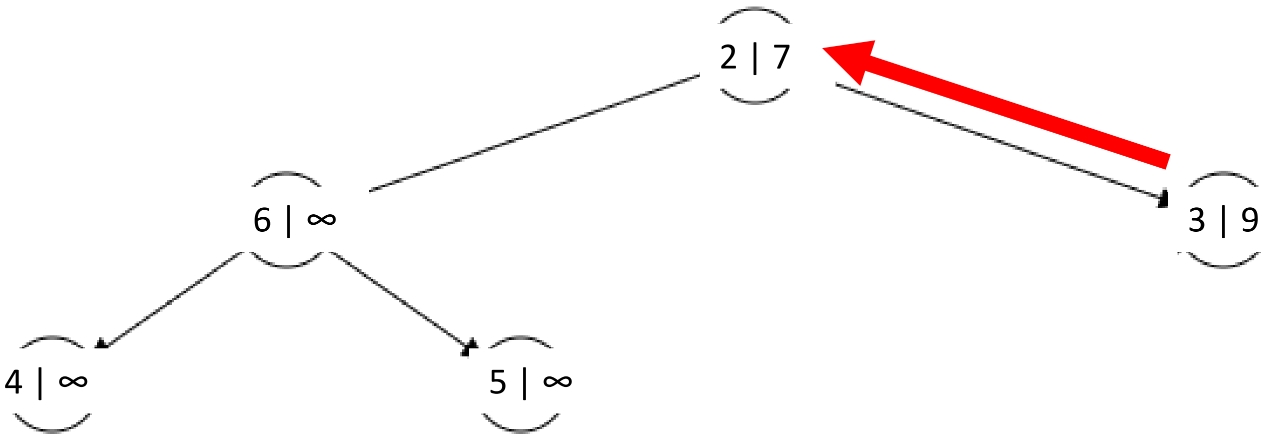


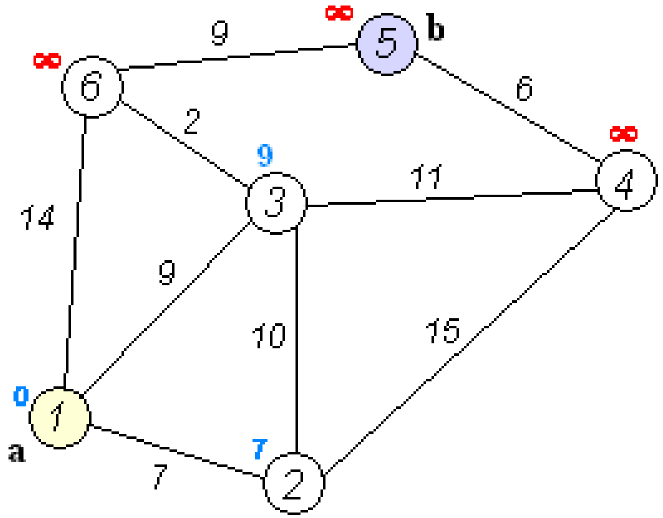

| | | | | | |
|---|---|---|---|---|---|
| **2 \| 7** | **6\| ∞** | **3\| ∞** | **4\| ∞** | **5\| ∞** | |




K = 1

# Algoritmo de Dijkstra – minHeap Binária changeKey

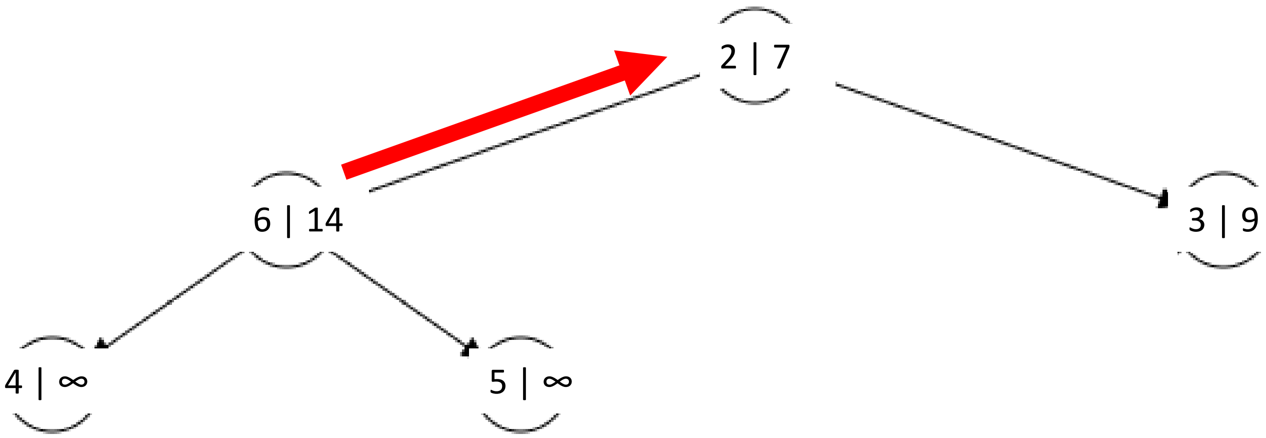


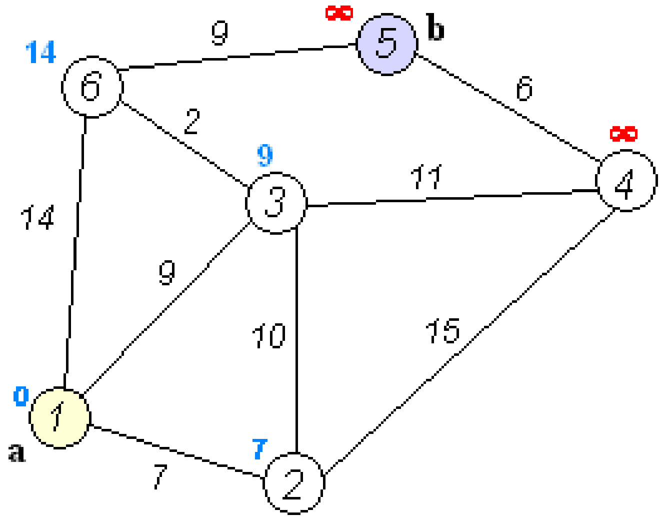

| | | | | | |
|---|---|---|---|---|---|
| 2 \| 7 | 6\| ∞ | 3\| 9 | 4\| ∞ | 5\| ∞ | |

K = 1

# Algoritmo de Dijkstra – minHeap Binária changeKey

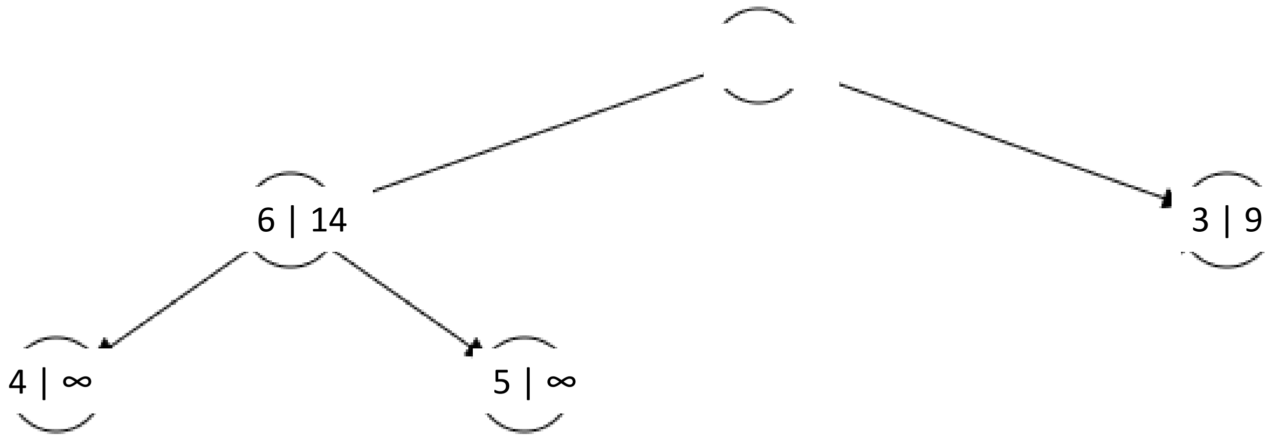



<table><tr><td>2 | 7</td><td>6| 14</td><td>3| 9</td><td>4| ∞</td><td>5| ∞</td><td></td></tr></table>

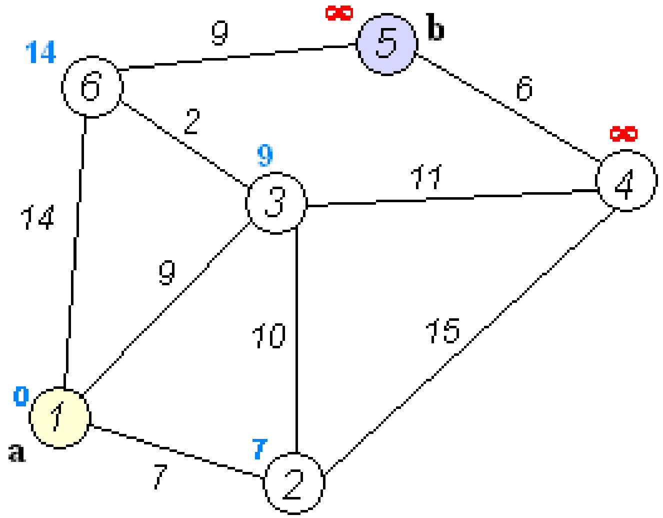



K = 1

# Algoritmo de Dijkstra – minHeap Binária
# Remoção (Escolha do k)



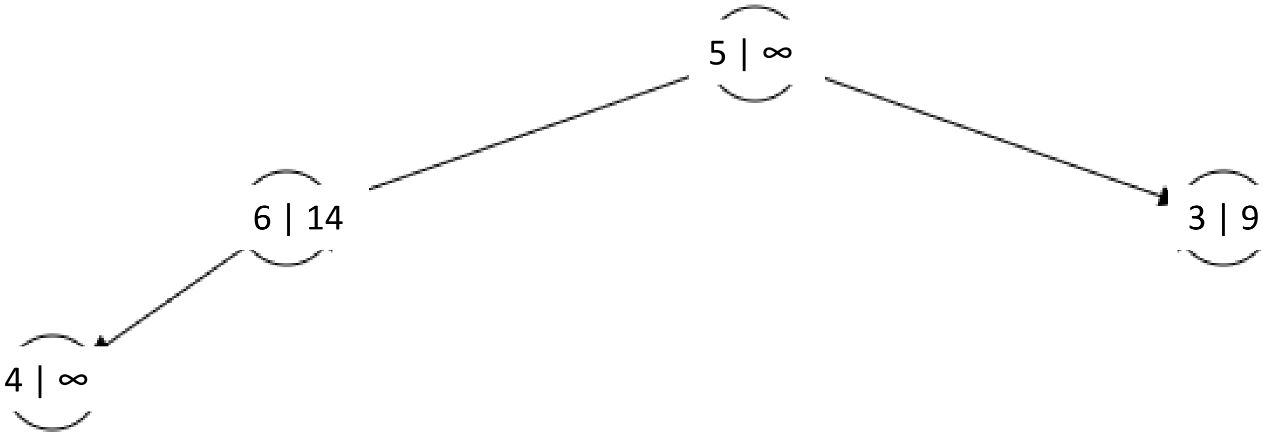

| | | | | | |
|---|---|---|---|---|---|
| 2 \| 7 | 6\| 14 | 3\| 9 | 4\| ∞ | 5\| ∞ | |



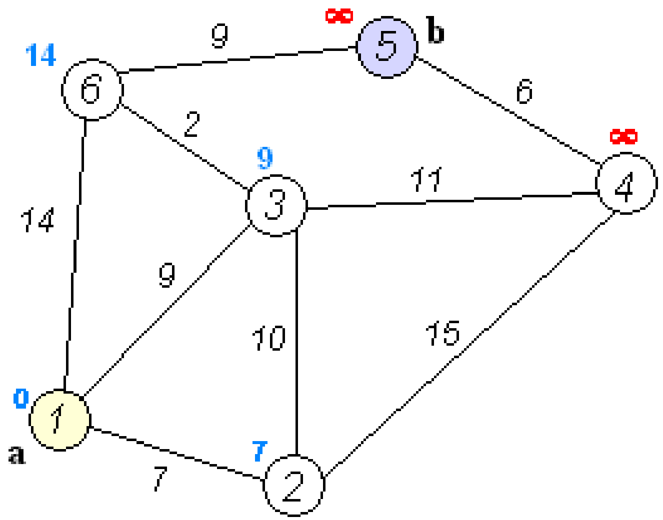

K = 1

# Algoritmo de Dijkstra – minHeap Binária
# Remoção (Escolha do k)

Qual dos filhos do topo é o menor?



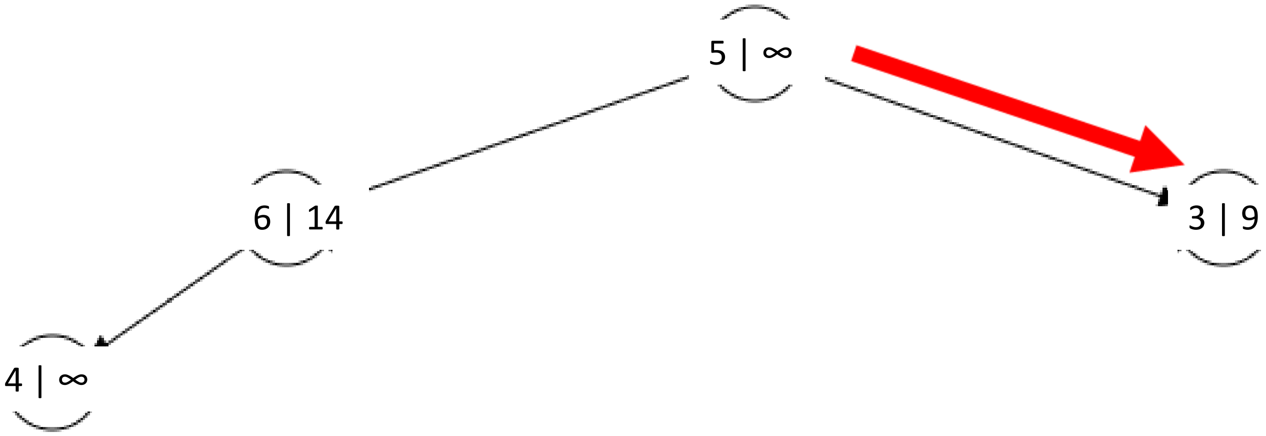

| | | | | | |
|---|---|---|---|---|---|
| 5\| ∞ | 6\| 14 | 3\| 9 | 4\| ∞ | | |



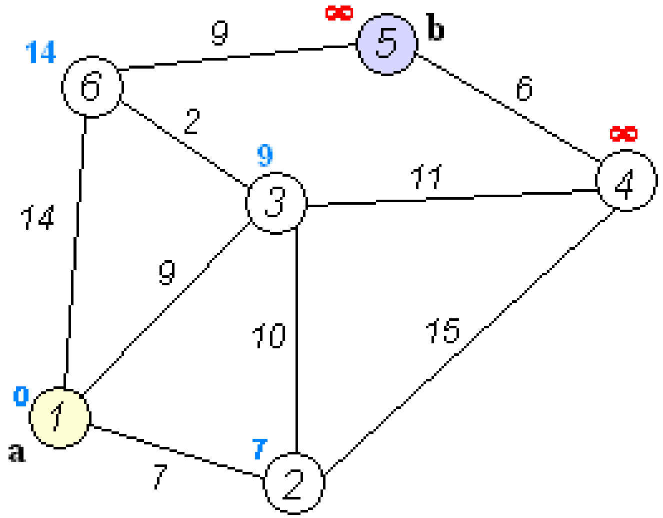

K = 2

# Algoritmo de Dijkstra – minHeap Binária
# Remoção (Escolha do k)

Qual dos filhos do topo é o menor?



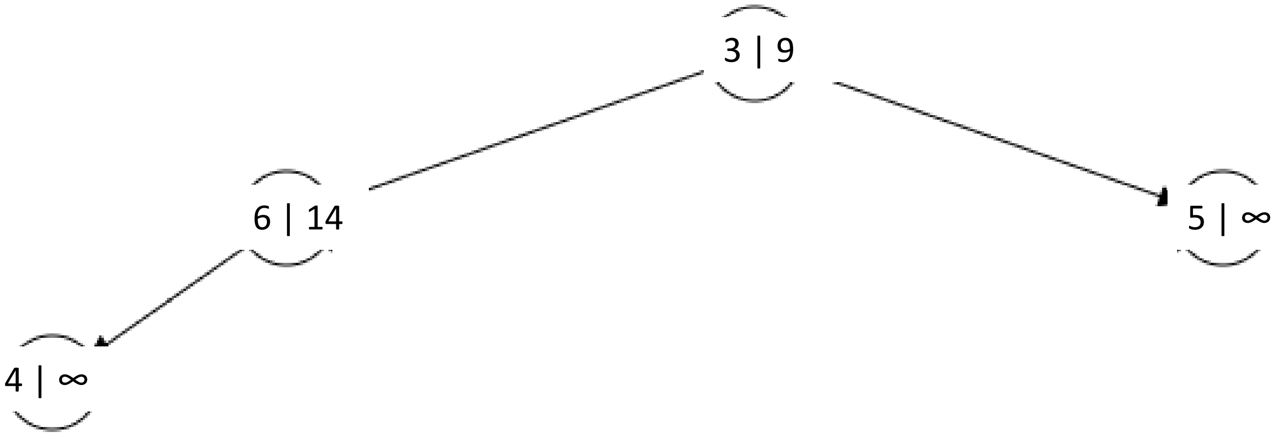

| | | | | | |
|---|---|---|---|---|---|
| 5\| ∞ | 6\| 14 | 3\| 9 | 4\| ∞ | | |

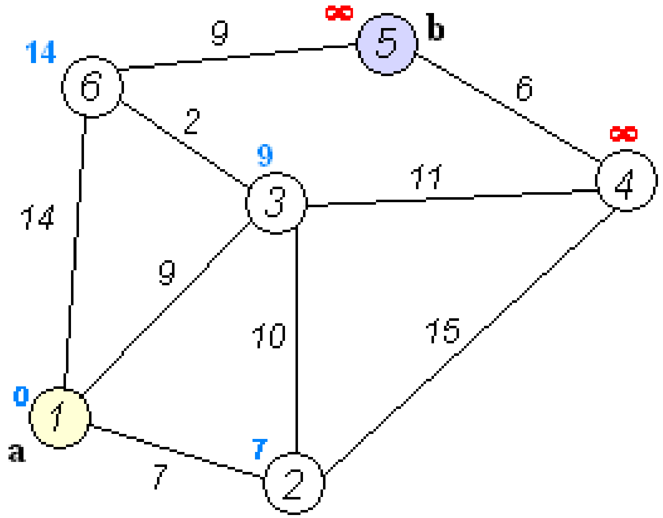



K = 2

# Algoritmo de Dijkstra – minHeap Binária
# Remoção (Escolha do k)

Qual dos filhos do topo é o menor?




| | | | | | |
|---|---|---|---|---|---|
| 3 \| 9 | 6\| 14 | 5\| ∞ | 4\| ∞ | | |

K = 2

# Algoritmo de Dijkstra – minHeap Binária

- O máximo de operações a serem feitas seriam no máximo a altura da árvore o que faz com que o custo computacional das operações da heap, tanto na remoção quanto na changeKey sejam $O(\log n)$.

# Algoritmo de Dijkstra – minHeap Binária

```
typedef struct{
    float key;
    int  dataIndex;
}HeapItem;

typedef struct{
HeapItem *H;
    int *map;
    int n;
    int size;
    int size_map;
}Heap;
```

```
t_graph_info dijkstra_heap (t_graph** adjacent_list, int graph_size, int vertex_ini, int
vertex_end){
    Heap *h
    for(i = 0;
        if(i ==
        else in
    }
    while (k!
        k = rer
        t_grap
        for(p = adjacent_list[k]; p!=NULL; p = p->prox){
            dist_v1 = h->H[h->map[p->vertex]].key; dist_k = h->H[h->map[k]].key; edge_weight = p->cost;
            if(dist_v1 > (dist_k + edge_weight)){
                dist_v1 = dist_k + edge_weight;
                r.anterior[p->vertex] = k;
                changeKey(h,h->H[h->map[p->vertex]].dataIndex,dist_v1);
            }}}}
```

**Complexidade Total:**

$$O(m\ log\ n)$$

# Resultados Computacionais

- Cada instância foi dividida em um caminho fácil, médio e difícil.
- Vértice 1 como ponto de partida para todos as instâncias.
  - Difícil (df) = primeiro vértice periférico encontrado
  - Médio (md) = $\frac{df}{2} - 100 < \frac{df}{2} < \frac{df}{2} + 100$
  - Fácil (fa) = $\frac{md}{2} - 100 < \frac{md}{2} < md + 100$

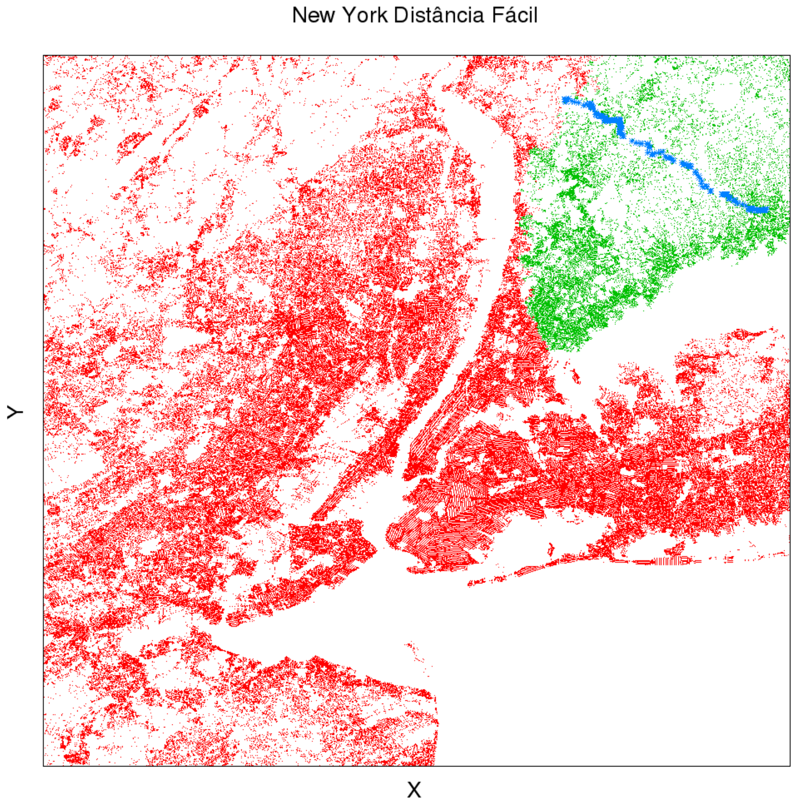
# Resultados Computacionais – NY

## Vértices: 264346 | Arestas: 733846

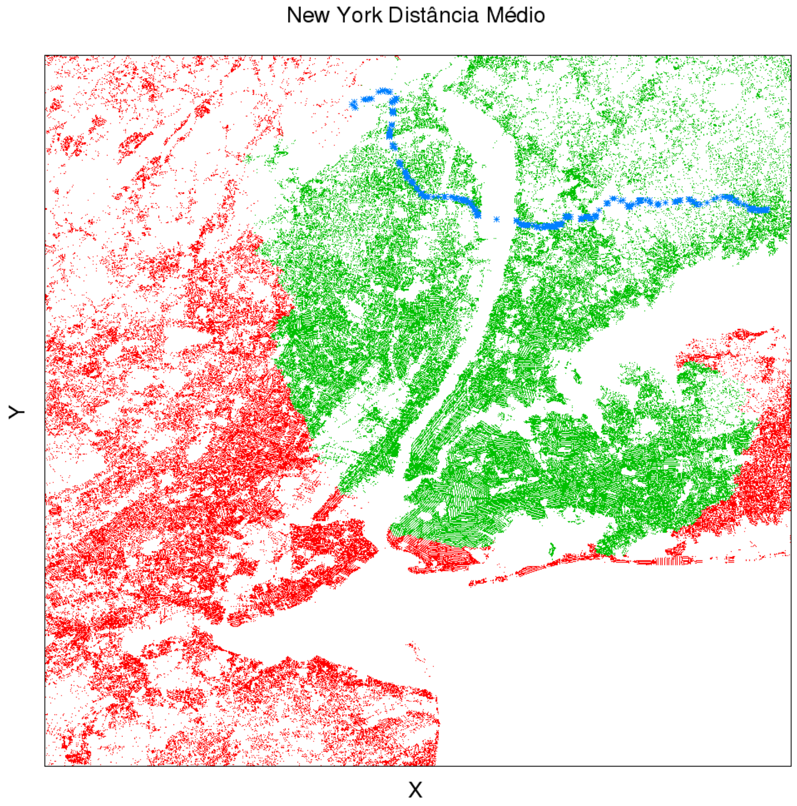
| **NY.d** | **TEMPO DE EXECUÇÃO (Vetor \| Heap)** | | **CUSTO** |
|---|---|---|---|
| Fácil | 13.848363 secs | 0.016686 secs | 359425 |
| Médio | 1 min 7 secs | 0.067572 secs | 719659 |
| Difícil | 1 min 52 secs | 0.111725 secs | 1440081 |

| **NY.t** | **TEMPO DE EXECUÇÃO (Vetor \| Heap)** | | **CUSTO** |
|---|---|---|---|
| Fácil | 9.890253 secs | 0.014851 secs | 477526 |
| Médio | 1 min 14 secs | 0.080786 secs | 957198 |
| Difícil | 1 min 57 secs | 0.114438 secs | 1916421 |

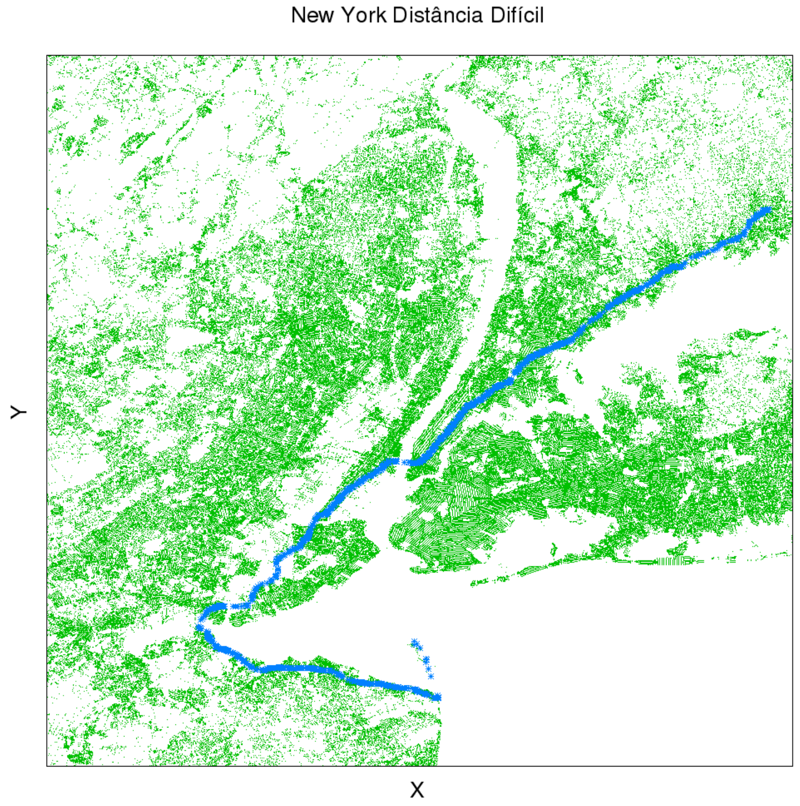
# Resultados Computacionais – NY.d




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 27448 | 193 |

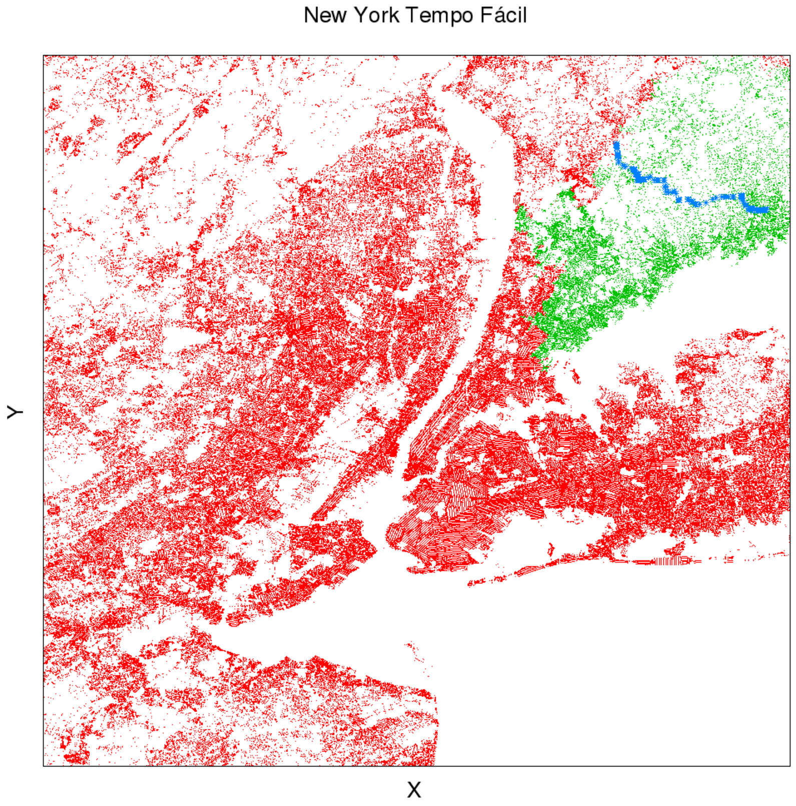



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 147106 | 296 |

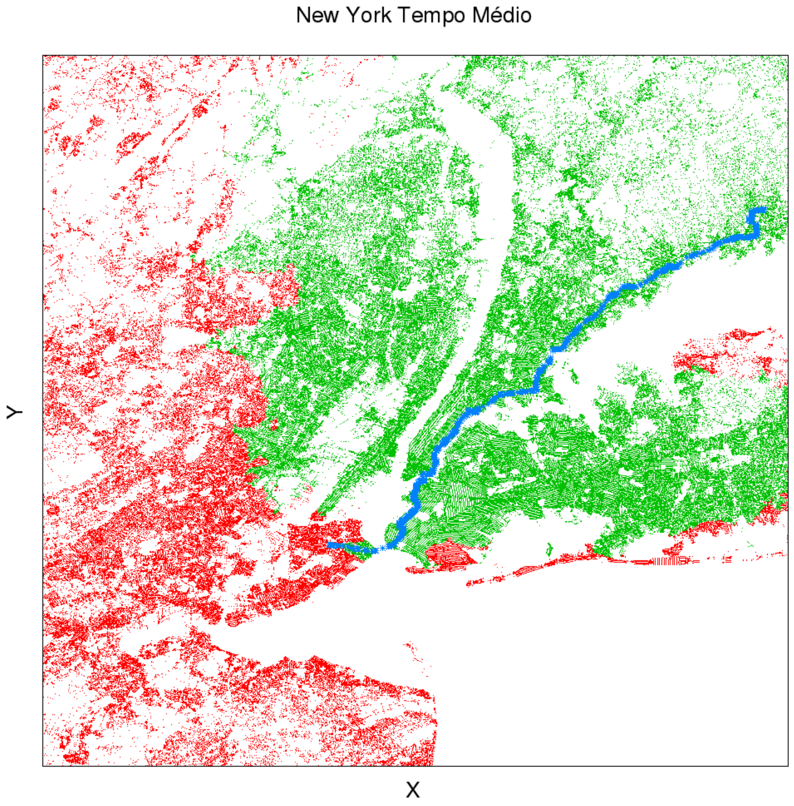



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 264346 | 1177 |

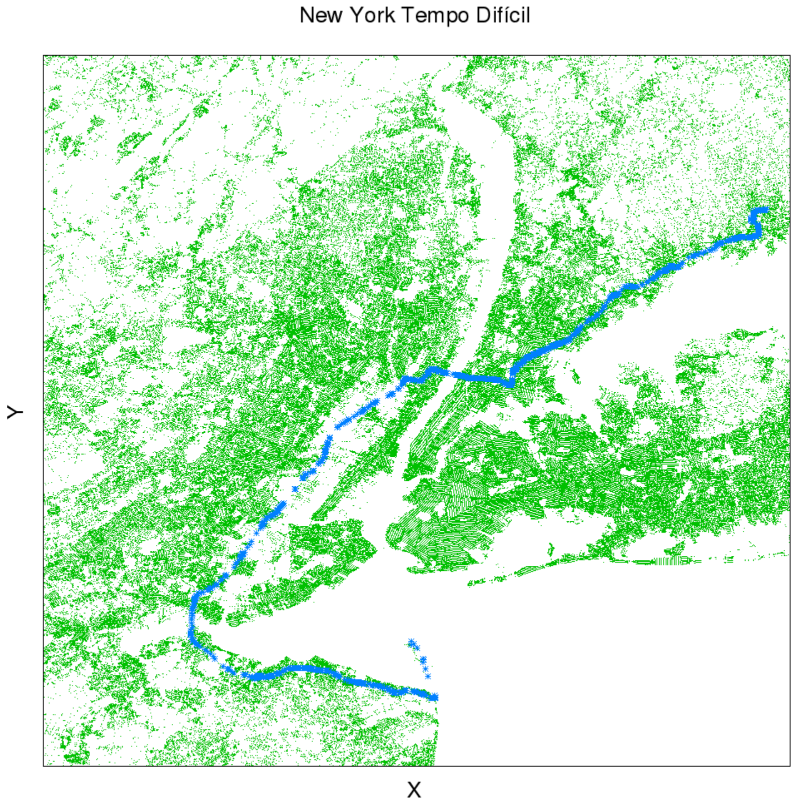
# Resultados Computacionais – NY.t




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 22105 | 120 |




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 166863 | 601 |




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 264346 | 868 |

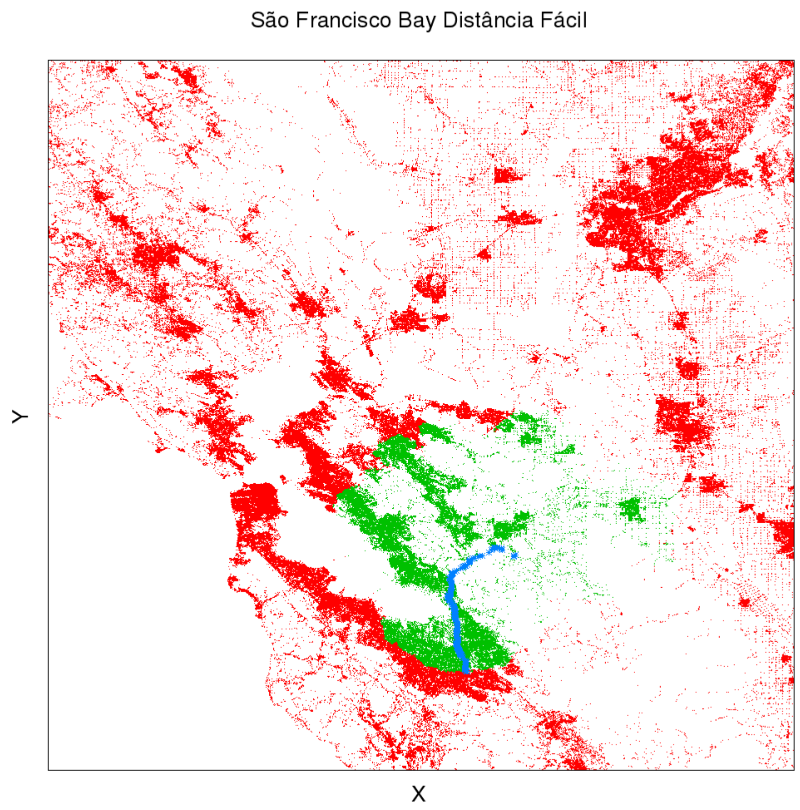
# Resultados Computacionais – BAY

## Vértices: 321270 | Arestas: 800172

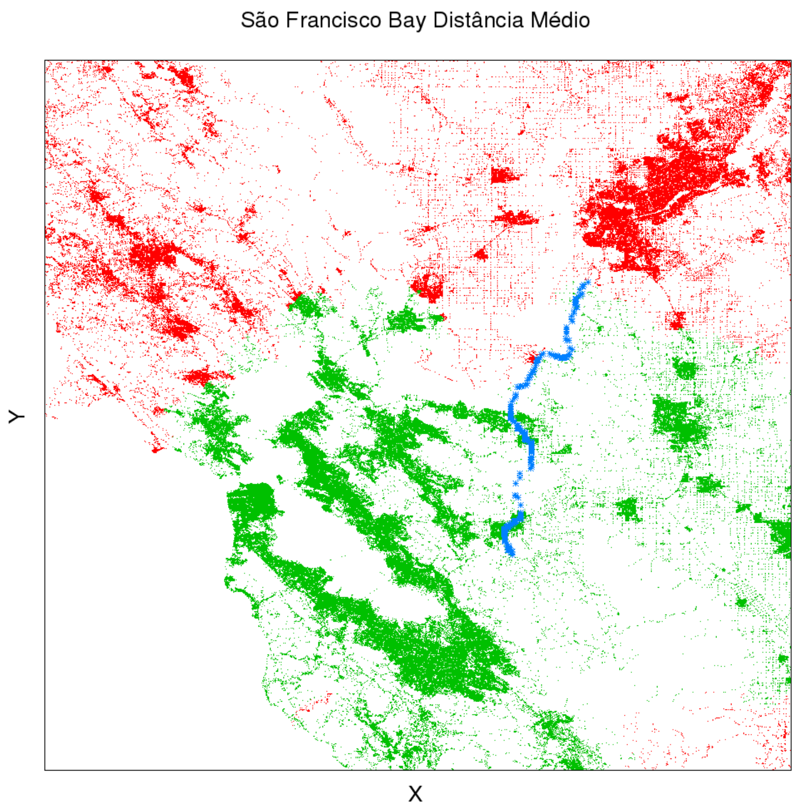
| BAY.d | TEMPO DE EXECUÇÃO (Vetor \| Heap) | | CUSTO |
|---|---|---|---|
| Fácil | 40.983618 secs | 0.039093 secs | 539988 |
| Médio | 1 min 54 secs | 0.096090 secs | 1082034 |
| Difícil | 2 mins 45 secs | 0.138710 secs | 2165941 |

| BAY.t | TEMPO DE EXECUÇÃO (Vetor \| Heap) | | CUSTO |
|---|---|---|---|
| Fácil | 57.966595 secs | 0.053536 secs | 866301 |
| Médio | 2 mins 13 secs | 0.117376 secs | 1737617 |
| Difícil | 2 mins 45 secs | 0.145794 secs | 3480192 |

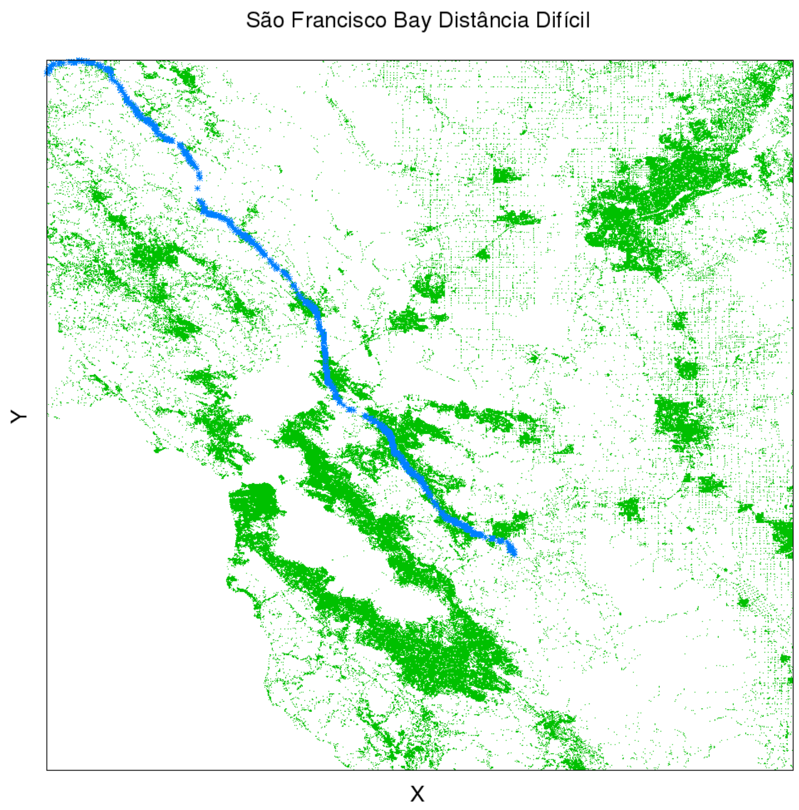
# Resultados Computacionais – BAY.d




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 73920 | 266 |

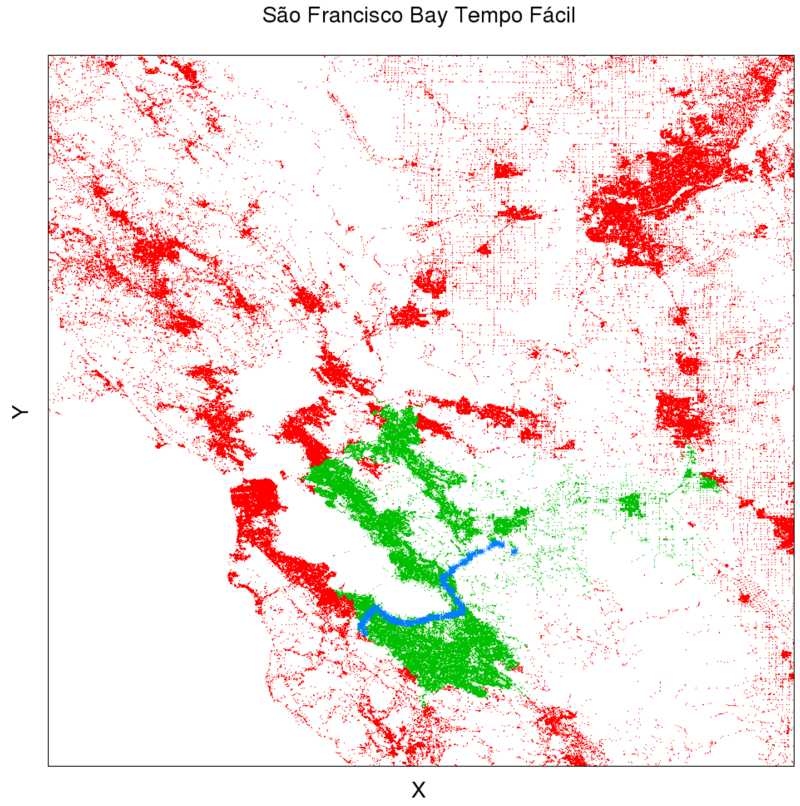



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 212411 | 301 |

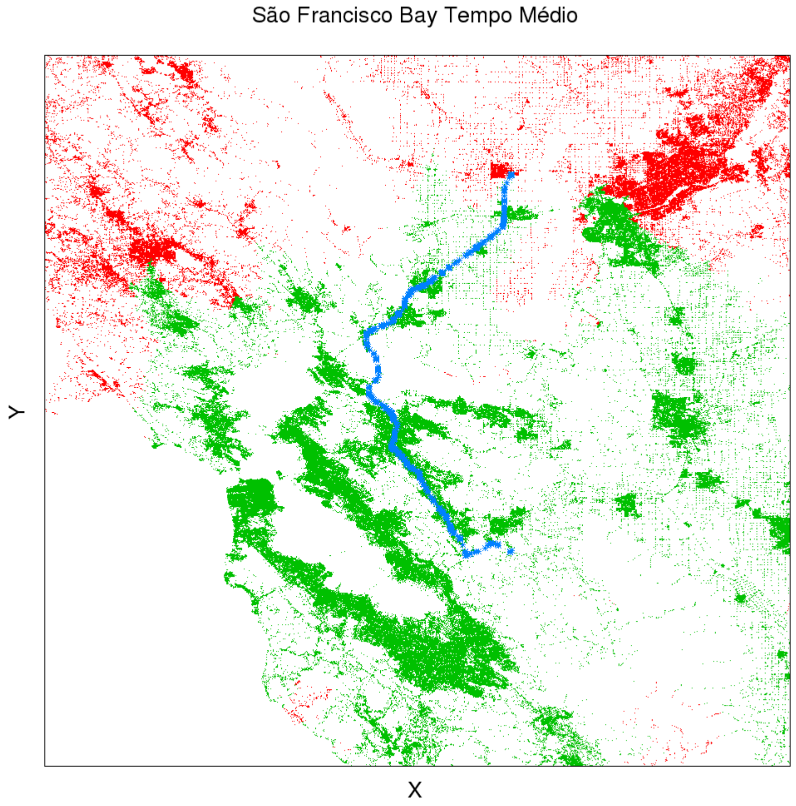



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 321270 | 942 |

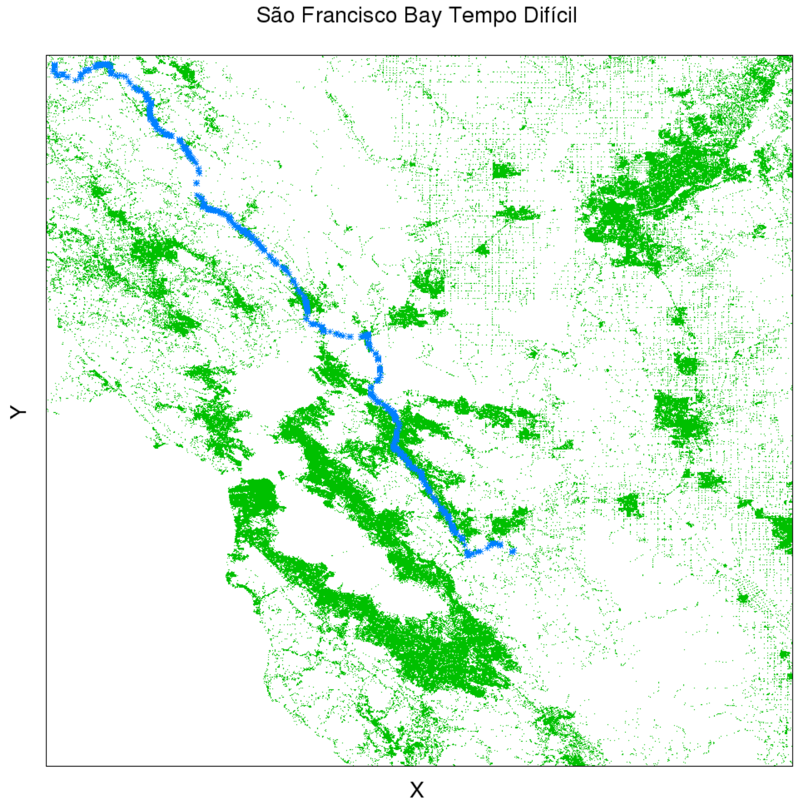
# Resultados Computacionais – BAY.t




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 103233 | 303 |




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 248040 | 512 |




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 321270 | 748 |

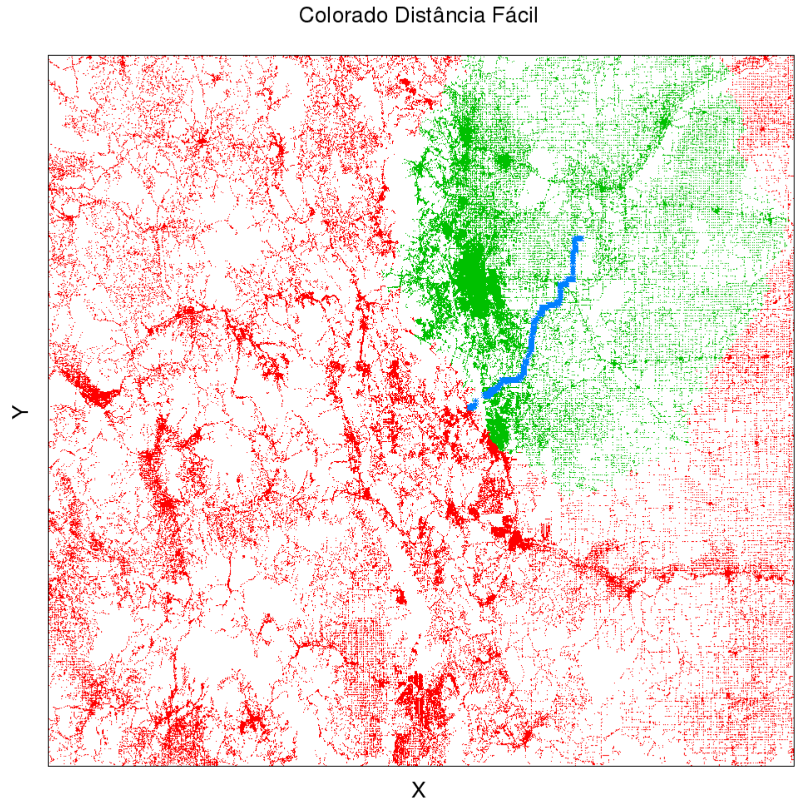
# Resultados Computacionais – COL

## Vértices: 435666 | Arestas: 1057066

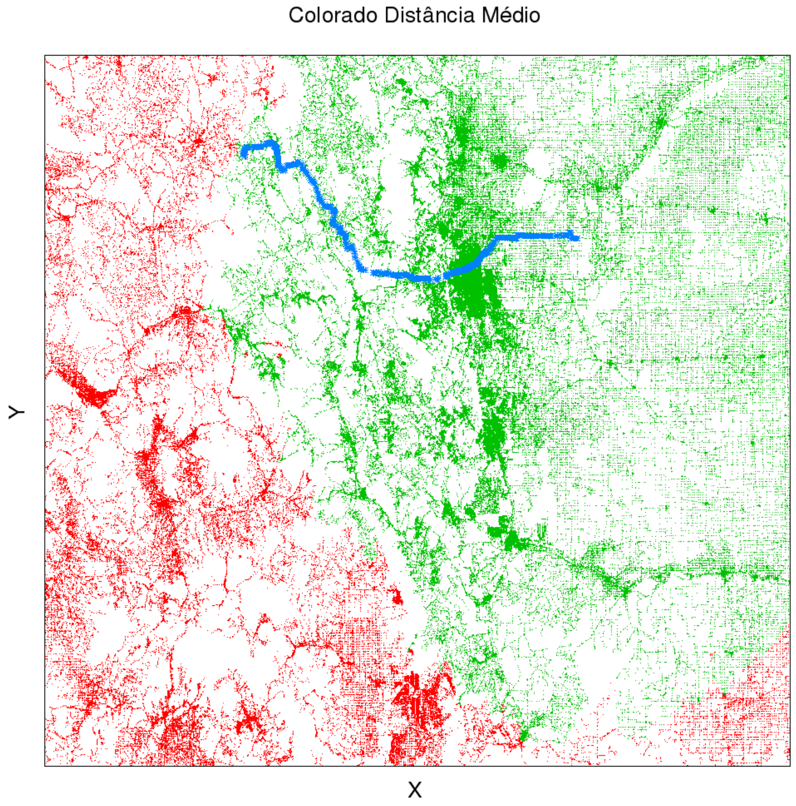
| COL.d | TEMPO DE EXECUÇÃO (Vetor \| Heap) | | CUSTO |
|---|---|---|---|
| Fácil | 3 mins 28 secs | 0.098990 secs | 1804428 |
| Médio | 5 mins 26 secs | 0.150582 secs | 3616936 |
| Difícil | 7 mins 6 secs | 0.195049 secs | 7246198 |

| COL.t | TEMPO DE EXECUÇÃO (Vetor \| Heap) | | CUSTO |
|---|---|---|---|
| Fácil | 3 mins 21 secs | 0.098132 secs | 2694987 |
| Médio | 5 mins 45 secs | 0.164361 secs | 5483070 |
| Difícil | 7 mins 23 secs | 0.196074 secs | 10935293 |

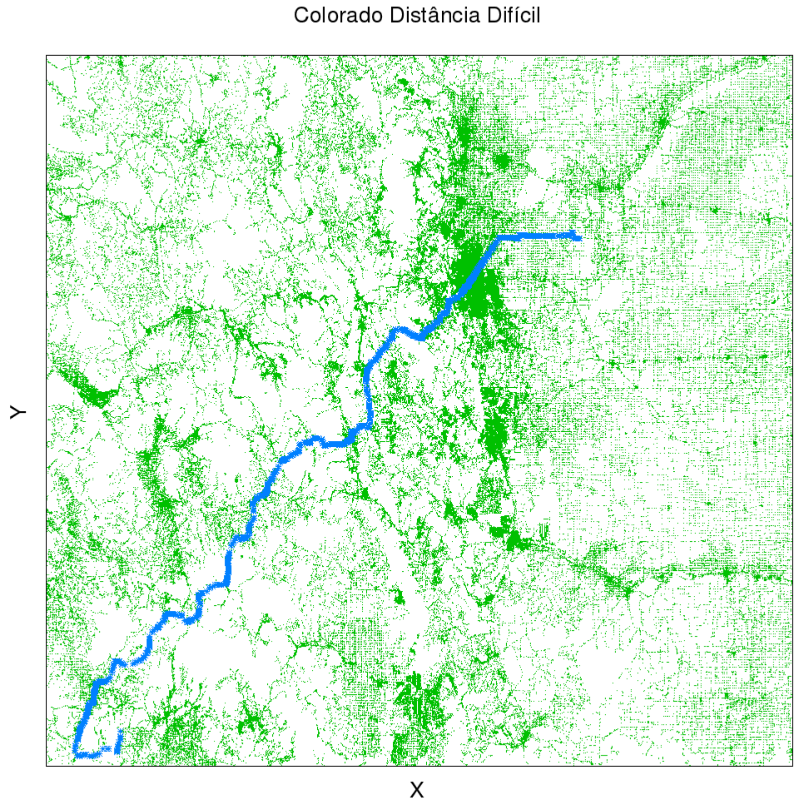
# Resultados Computacionais – COL.d




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 199934 | 337 |

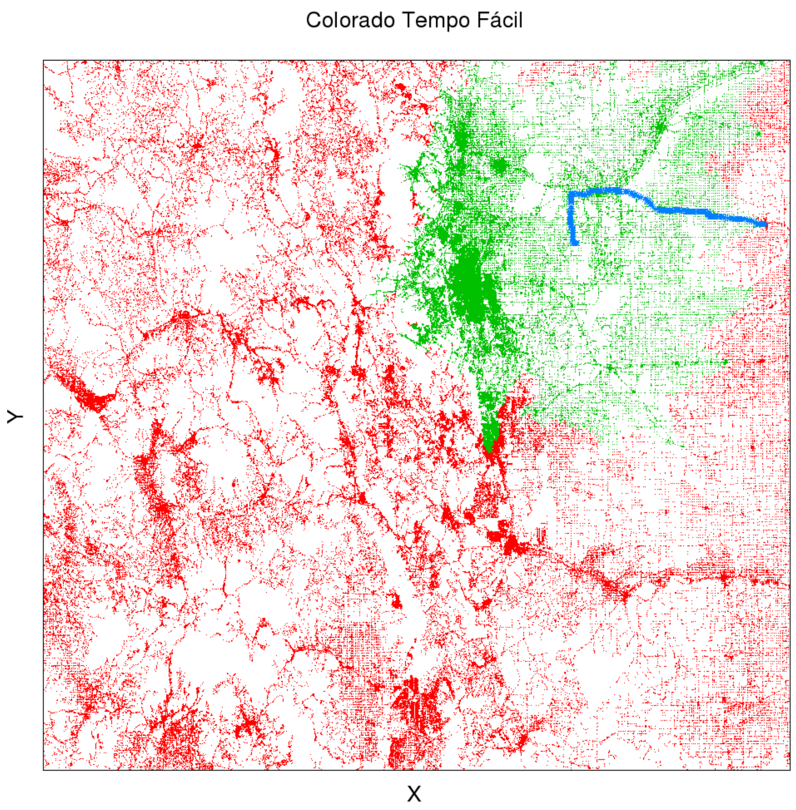



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 321101 | 813 |

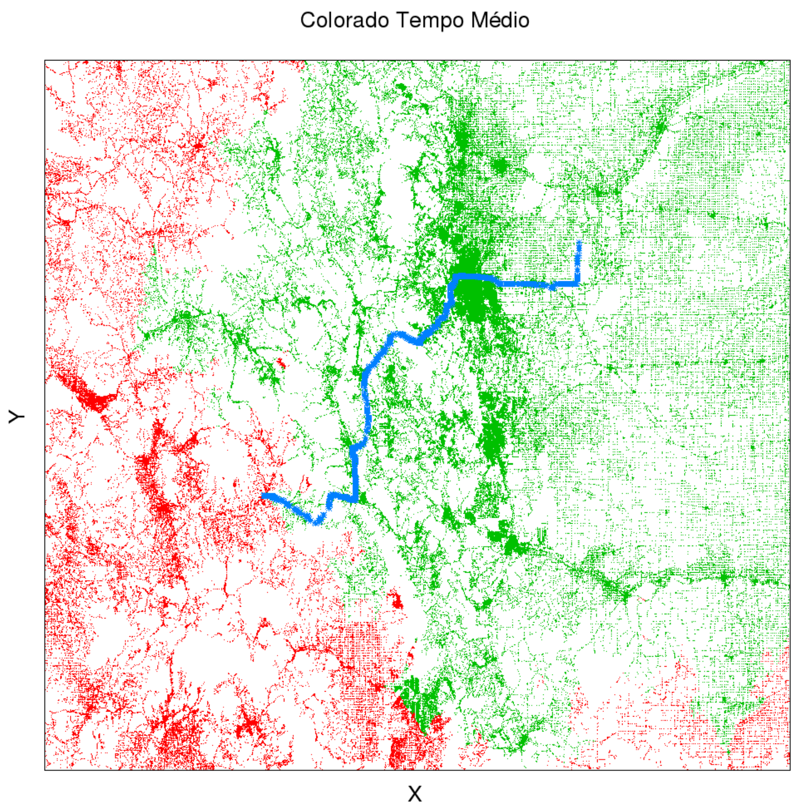



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 435666 | 1750 |

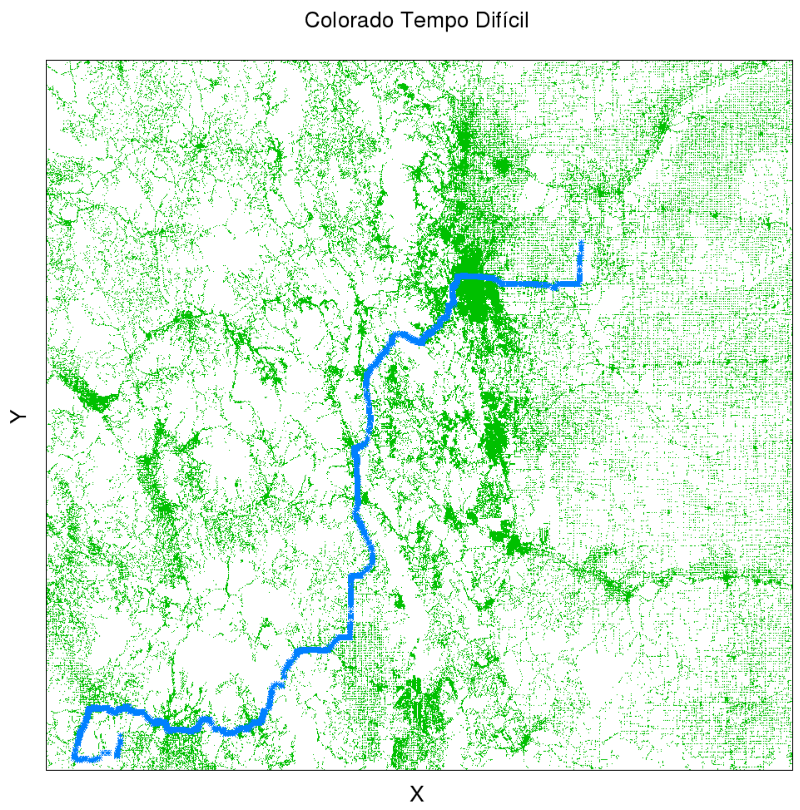
# Resultados Computacionais – COL.t




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 190177 | 331 |




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 336250 | 1091 |




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 435666 | 2020 |

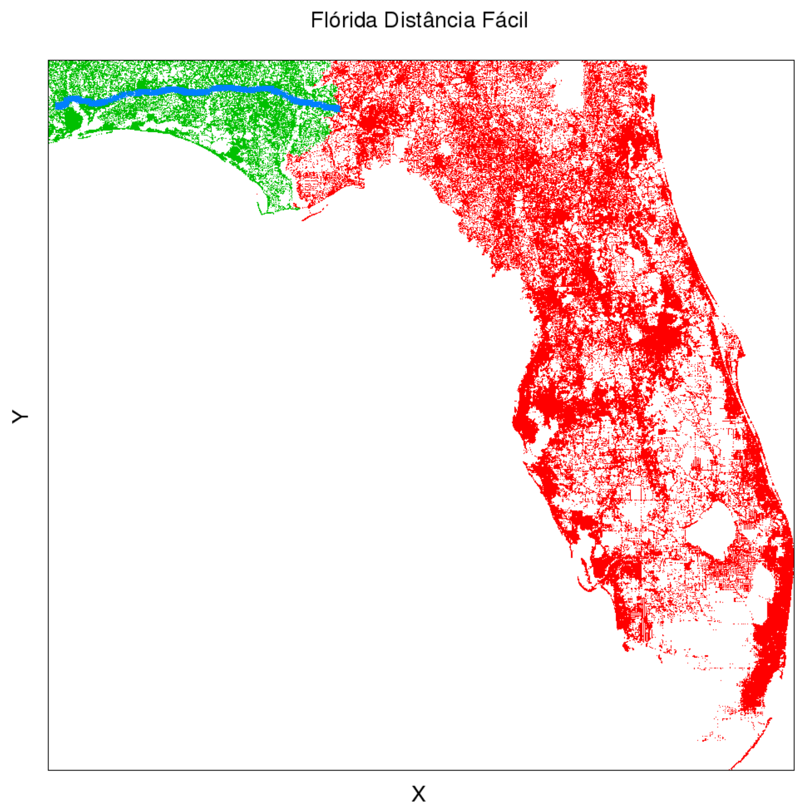
# Resultados Computacionais – FLA

## Vértices: 1070376 | Arestas: 2712798

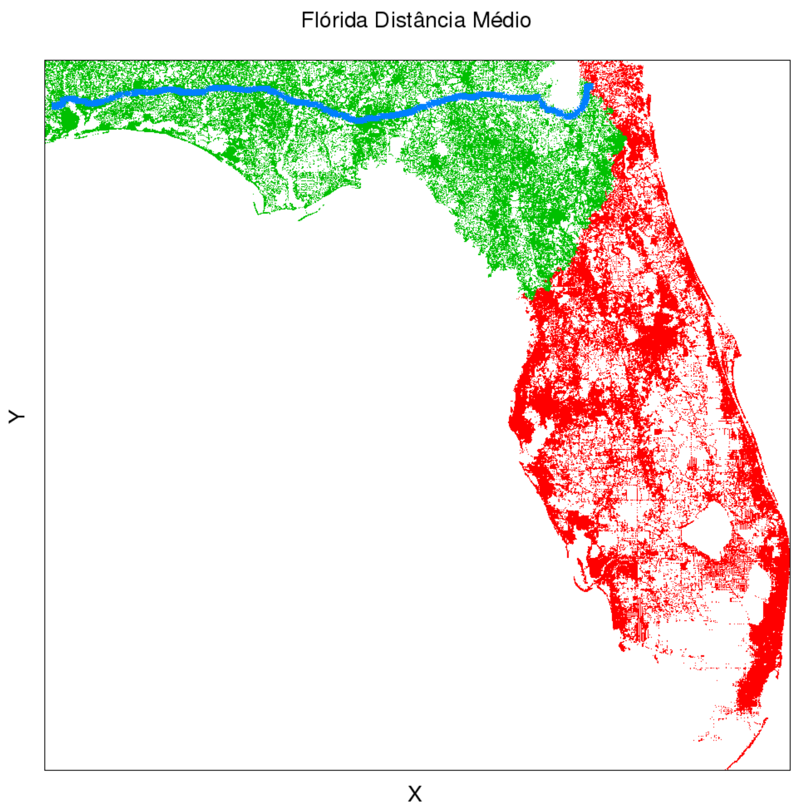
| FLA.d | TEMPO DE EXECUÇÃO (Vetor \| Heap) | | CUSTO |
|---|---|---|---|
| Fácil | 3 mins 13 secs | 0.074377 secs | 2907930 |
| Médio | 8 mins 17 secs | 0.149866 secs | 5829987 |
| Difícil | 30 mins 20 secs | 0.491751 secs | 11674195 |

| FLA.t | TEMPO DE EXECUÇÃO (Vetor \| Heap) | | CUSTO |
|---|---|---|---|
| Fácil | 3 mins 36 secs | 0.077648 secs | 3417632 |
| Médio | 11 mins 45 secs | 0.206668 secs | 6870773 |
| Difícil | 30 mins 3 secs | 0.508526 secs | 13776909 |

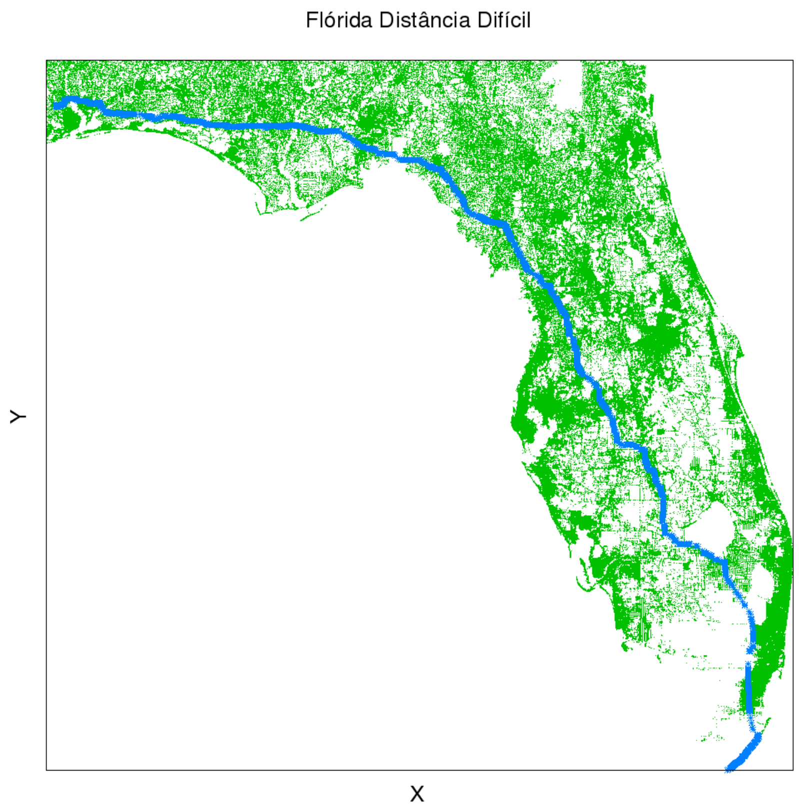
# Resultados Computacionais – FLA.d




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 110581 | 768 |

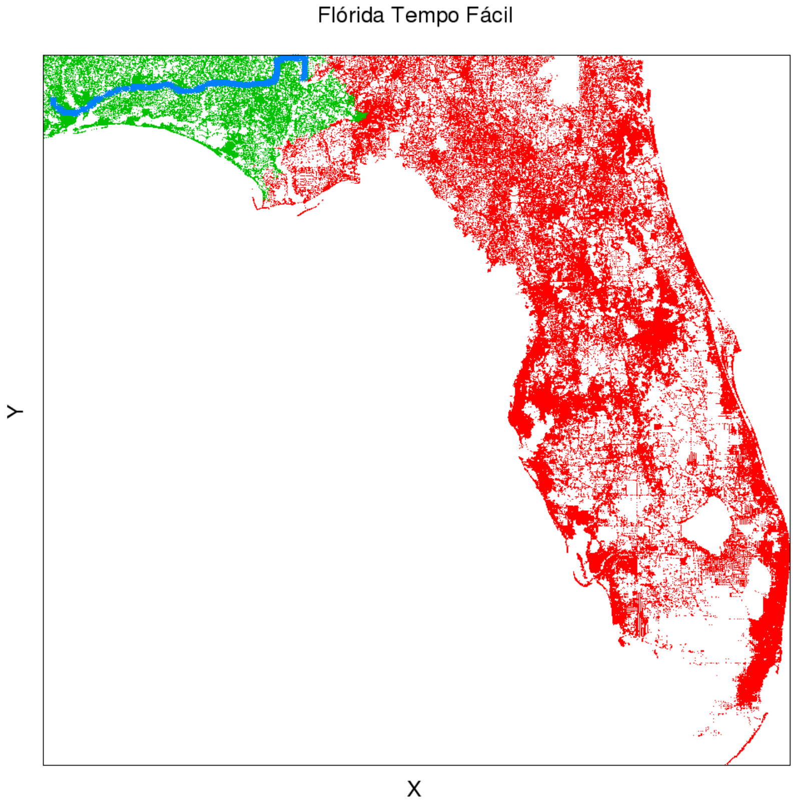



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 281082 | 1363 |

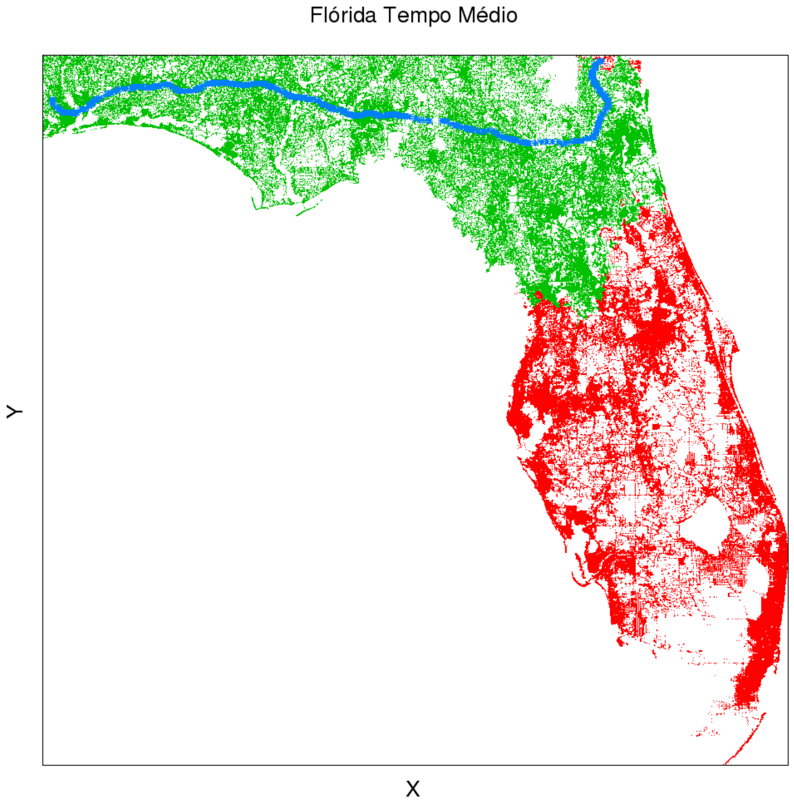



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 1070376 | 3529 |

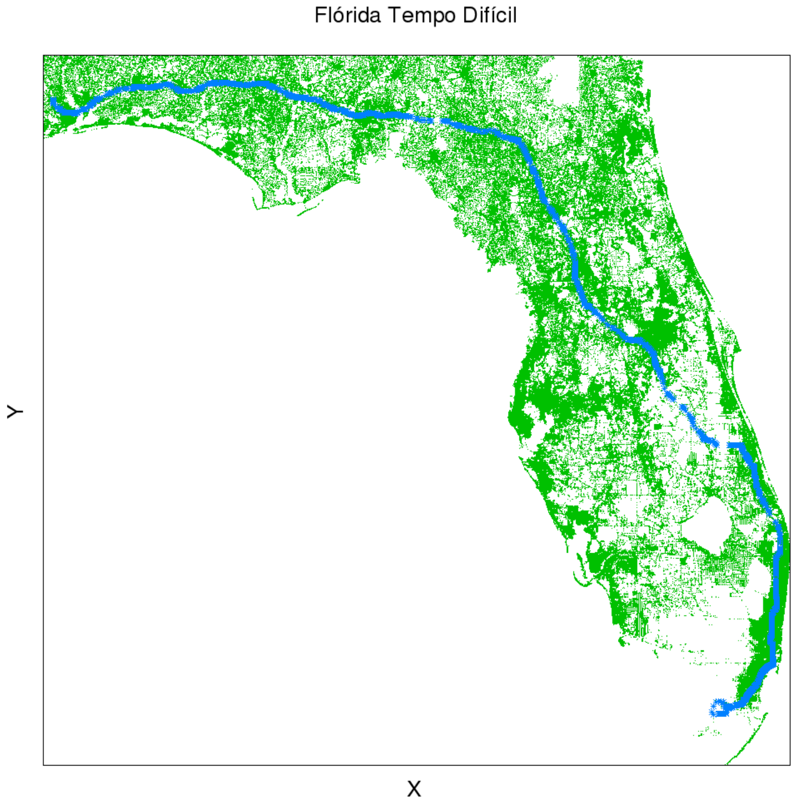
# Resultados Computacionais – FLA.t




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 114490 | 608 |




| Pontos Visitados | Tamanho Caminho |
|---|---|
| 385384 | 1198 |

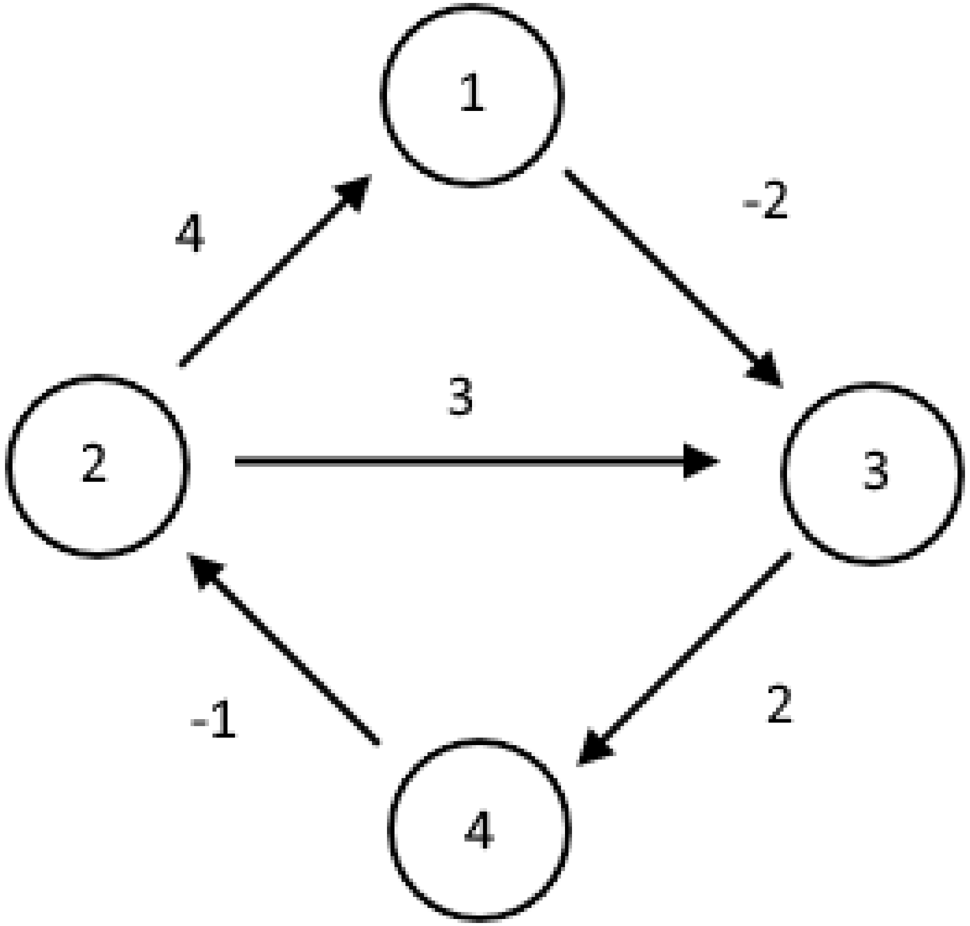



| Pontos Visitados | Tamanho Caminho |
|---|---|
| 1070376 | 2332 |

# Algoritmo de Floyd

- Caminho mais curto entre **todos** os pares de vértices.
- Grafo orientado e valorado.
- Ciclos negativos não são permitidos.
- Arestas negativas são permitidas.

# Algoritmo de Floyd



# Algoritmo de Floyd

```
let V be the number of vertices in a graph
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u,v)
    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for each vertex v
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

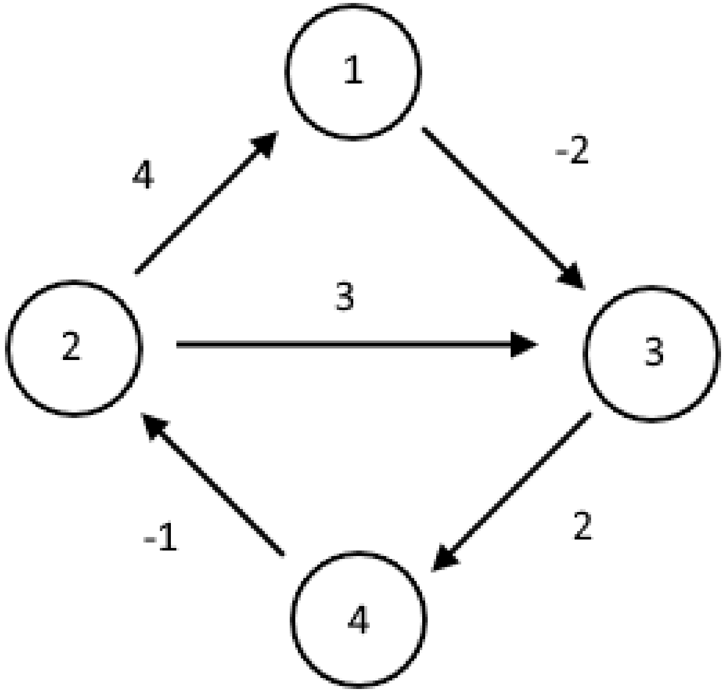
# Algoritmo de Floyd

2  let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | ∞ | ∞ | ∞ | ∞ |
| 2 | ∞ | ∞ | ∞ | ∞ |
| 3 | ∞ | ∞ | ∞ | ∞ |
| 4 | ∞ | ∞ | ∞ | ∞ |

# Algoritmo de Floyd

```
for each edge (u,v)
    dist[u][v] ← w(u,v)
for each vertex v
    dist[v][v] ← 0
```




| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | ∞ | -2 | ∞ |
| 2 | 4 | 0 | 3 | ∞ |
| 3 | ∞ | ∞ | 0 | 2 |
| 4 | ∞ | -1 | ∞ | 0 |

# Algoritmo de Floyd

```
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

# Algoritmo de Floyd

```
if dist[i][j] > dist[i][k] + dist[k][j]
    dist[i][j] ← dist[i][k] + dist[k][j]
```

| k = 1 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | ∞ | -2 | ∞ |
| 2 | 4 | 0 | **2** | ∞ |
| 3 | ∞ | ∞ | 0 | 2 |
| 4 | ∞ | -1 | ∞ | 0 |

| k = 2 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | ∞ | -2 | ∞ |
| 2 | 4 | 0 | 3 | ∞ |
| 3 | ∞ | ∞ | 0 | 2 |
| 4 | **3** | -1 | **1** | 0 |

# Algoritmo de Floyd

```
if dist[i][j] > dist[i][k] + dist[k][j]
    dist[i][j] ← dist[i][k] + dist[k][j]
```

| k = 3 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | ∞ | -2 | **0** |
| 2 | 4 | 0 | 2 | **4** |
| 3 | ∞ | ∞ | 0 | 2 |
| 4 | ∞ | -1 | ∞ | 0 |

| k = 4 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | **-1** | -2 | ∞ |
| 2 | 4 | 0 | 3 | ∞ |
| 3 | **5** | **1** | 0 | 2 |
| 4 | 3 | -1 | 1 | 0 |

# Algoritmo de Floyd

```
let V be the number of vertices in a graph
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
let next be a |V| × |V| array of adjacent vertices initialized to -1
for each edge (u,v)
    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
    next[u][v] ← v
for each vertex v
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
                next[i][j] ← next[i][k]
            end if
```

# Algoritmo de Floyd

- Devido aos 3 loops for aninhados, a complexidade do algoritmo de Floyd para todos os casos de teste é $O(n^3)$.

# Resultados Computacionais – RO

Vértices: 3353 | Arestas: 8870

| TEMPO DE EXECUÇÃO | | | |
|---|---|---|---|
| **Dijkstra (Vetor \| Heap)** | | **Floyd (Matriz \| Vetor)** | |
| 1 min 3 secs | 0.000486 secs | 1 min 52 secs | 5 mins 32 secs |

As matrizes de distância e de recuperação de caminhos geradas pelos algoritmos de Floyd e Dijkstra são exatamente iguais.

# DÚVIDAS???