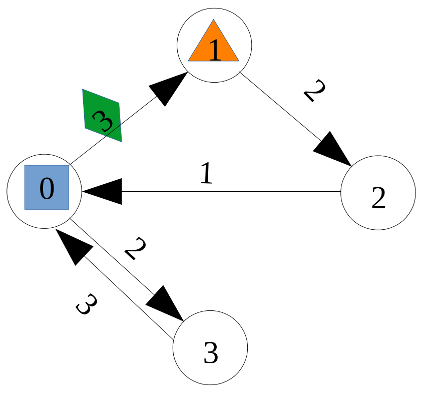
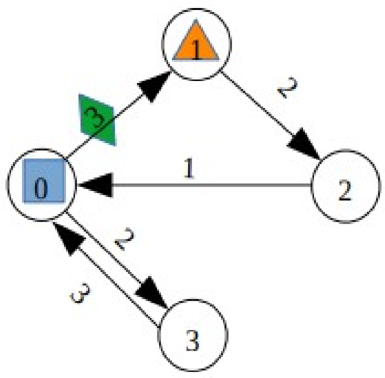
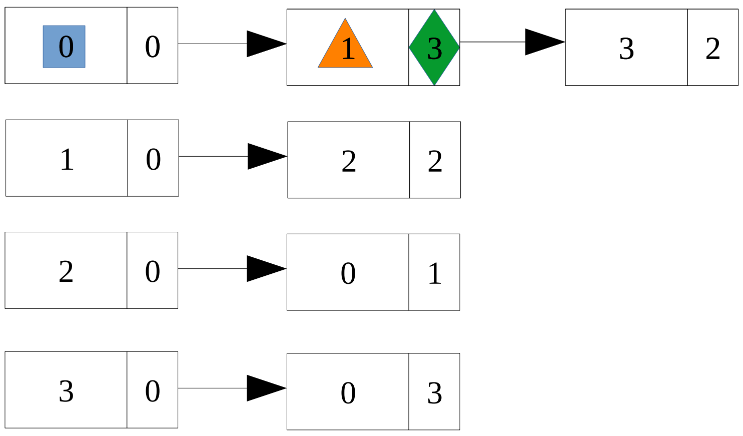
## Grafos – Matriz de Adjacência



| Vertices | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 3 | 0 | 2 |
| 1 | 0 | 0 | 2 | 0 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 3 | 0 | 0 | 0 |

## Grafos – Lista Ligada





## Algoritmo de Dijkstra

### Pseudocódigo

```
Function Dijkstra(G,source, target):
    for each vertex v in G
        dist[v] = infinity
    dist[source] = 0
    Q has the set of all nodes in G
    while Q is not empty:
        u = vertex in Q with smallest dist
        remove u from Q
        if u = target
            break
        for each arc (v,u) in G
            if dist[v] > dist[u] + dist_between(v, u)
                dist[v] = dist[u] + dist_between(v, u)
    return dist
```

### Código

```c
void dijkstra(int graph[][MAX_NODES], int num_nodes, int source, int target){
    for (int i=0; i<num_nodes; i++) graph[i][i] = __INT_MAX__;
    graph[source][source] = 0;


    int node_set[num_nodes];
    for (int i=0; i<num_nodes; i++) node_set[i] = 1;

    while (set_is_empty(node_set, num_nodes) != 1){
        int smallest = smallest_dist(graph, node_set, num_nodes);

        node_set[smallest] = -1;

        if (smallest == target) break;

        for (int i=0; i<num_nodes; i++){
            if (graph[smallest][i] != 0 && node_set[i] != -1){
                if(graph[i][i] > graph[smallest][smallest]+graph[smallest][i]
                    && graph[smallest][smallest] != __INT_MAX__
                    && graph[smallest][i] != __INT_MAX__){
                    graph[i][i]=graph[smallest][smallest] + graph[smallest][i];
                }
            }
        }
    }
}

int smallest_dist(int graph[][MAX_NODES], int node_set[], int num_nodes){
    int min = __INT_MAX__, node=-1;
    for (int i=0; i<num_nodes; i++){
        if (node_set[i] != -1){
            if (graph[i][i] < min){
                min = graph[i][i];
                node = i;
            }
        }
    }
    return node;
}

int set_is_empty(int node_set[], int num_nodes){
    int is_empty = 1;
    for (int i=0; i<num_nodes; i++){
        if (node_set[i] != -1){
            return -1;
        }
    }
    return is_empty;
}
```

## Algoritmo de Bellman-Ford

### Pseudocódigo

```
Function BelmannFord(G,source):
    for each vertex v in G
        dist[v] = infinity
    d[source]=0;

    for(i=0; i<|V|-1; i++)
        for each arc (u,v) in G
            if dist[v] > dist[u]+ dist_between(u, v)
                d[v] = d[u] + dist_between(u, v)

    // Verificação de ciclos negativos
    for each arc (u,v) in G
        if dist[v] > dist[u]+ dist_between(u, v)
            return false   // Ciclo negativo!!
    return true
```

### Código

```
int bellman_ford(int graph[][MAX_NODES], int num_nodes, int source, int target){
    for (int i=0; i<num_nodes; i++) graph[i][i] = __INT_MAX__;

    graph[source][source] = 0;

    /*Iterate |V| - 1, i.e, number of nodes - 1 */
    for (int i=0; i<num_nodes-1; i++){
        for (int j=0; j<num_nodes; j++){
            for (int k=0; k<num_nodes; k++){
                if (k==j || graph[j][k] == 0) continue;
                if (graph[k][k] > graph[j][j] + graph[j][k]
                    && graph[j][j] !=__INT_MAX__ && graph[j][k] != __INT_MAX__){
                    graph[k][k] = graph[j][j] + graph[j][k];
                }
            }
        }
    }

    /*Iteration number |V| serves to detect any negative cycles*/
    for (int j=0; j<num_nodes; j++){
        for (int k=0; k<num_nodes; k++){
            if (k==j || graph[j][k] == 0) continue;
            if (graph[k][k] > graph[j][j] + graph[j][k]
                && graph[j][j] !=__INT_MAX__ && graph[j][k] != __INT_MAX__){
                return -1;
            }
        }
    }

    return 1;
}
```

## Algoritmo de Floyd-Warshall

### Pseudocódigo

```
Function Floyd-Warshall(G)
    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]
```

### Código

```
void floyd_warshall(int graph[][MAX_NODES], int num_nodes){
    /*Set the diagonal to 0*/
    for (int i=0; i<num_nodes; i++) graph[i][i] = 0;

    /*Set 0 values to infinity (INT_MAX)*/
    for(int i=0; i<num_nodes; i++){
        for(int j=0; j<num_nodes; j++){
            if (i!=j && graph[i][j] == 0) graph[i][j] = __INT_MAX__;
        }
    }

    for (int k=0; k<num_nodes; k++){
        for (int i=0; i<num_nodes; i++){
            for (int j=0; j<num_nodes; j++){
                /*Don't calculate if right side values are INT_MAX, overflow*/
                if (graph[i][j] > graph[i][k] + graph[k][j]
                   && graph[i][k]!=__INT_MAX__ && graph[k][j] != __INT_MAX__)
                    graph[i][j] = graph[i][k] + graph[k][j];
            }
        }
    }
}
```

## Algoritmo BFS (Breadth-first Search)

### Pseudocódigo

```
Function BFS(G, start_node):
    let S be a queue
    S.enqueue(start_node)
    while S is not empty
        v = S.dequeue()
        if v is the goal:
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered:
                label w as discovered
                w.parent = v
                S.enqueue(w)
```

### Código

```
void bfs(int graph[][MAX_NODES], int num_nodes, int current, int target, int
visited[], int previous[]){
    /*Queue that holds neighbor nodes of current that haven't yet been visited*/
    queue <int> bfs_queue;
    /*Push the first node (the source node)*/
    bfs_queue.push(current);

    previous[0] = -1;

    /*While stack has nodes to visit*/
    while(!bfs_queue.empty()){
        /*Update current node and pop*/
        current = bfs_queue.front();
        bfs_queue.pop();

        /*We have visited this function node*/
        visited[current] = 1;

        /*For each edge that goes out of current node*/
        for (int j=0; j<num_nodes; j++){
            if (graph[current][j]!=0){
                /*If one of these nodes is our target*/
                if (j==target){
                    previous[j] = current;
                    return;
                }
                /*If we haven't visited node yet*/
                else if(visited[j]==0){
                    previous[j] = current;
                    bfs_queue.push(j);
                }
            }
        }
    }
}
```

## Algoritmo DFS (Depth-first Search)

### Pseudocódigo

```
Function DFS(G, v):
    if v is the goal:
        exit
    label v as visited
    for each neighbor u of v:
        if u is not labeled as discovered:
            u.parent = v
            DFS(G, u)
```

### Código

```c
int dfs(int graph[][MAX_NODES], int num_nodes, int current, int target, int visited[],
int previous[]){
    previous[0] = -1;

    /*We have visited this node*/
    visited[current]=1;

    if (current==target){
        return 1;
    }
    else{
        /*For each edge*/
        for (int j=0; j<num_nodes; j++){
            /*If an edge exists*/
            if (graph[current][j]!=0)
            {
                /*Unvisited node and in recursion we found target,update previous*/
                if (dfs(graph, num_nodes, j, target, visited, previous)==1
                     && visited[j]!=1){
                    previous[j] = current;
                    return 1;
                }
            }
        }
    }

    /*Haven't found target in this recursive step*/
    return 0;
}
```

## Algoritmo de Kruskal - Estrutura União-Busca (Disjoint-set)

### Pseudocódigo – MakeSet

```
function MakeSet(x)
   if x is not already present:
     add x to the disjoint-set tree
     x.parent = x
     x.rank   = 0
     x.size   = 1
```

### Pseudocódigo – Find

**Path compression**

```
function Find(x)
  if x.parent!= x
    x.parent:= Find(x.parent)
  return x.parent
```

**Path halving**

```
function Find(x)
  while x.parent!= x
    x.parent:= x.parent.parent
    x:= x.parent
  return x
```

**Path splitting**

```
function Find(x)
  while x.parent!= x
    next :=  x.parent
    x.parent := next.parent
    x := next
  return x
```

### Pseudocódigo – Union

**Union by rank**

```
function Union(x, y)
  xRoot:= Find(x)
  yRoot:= Find(y)

  //x and y are already in the same set
  if xRoot == yRoot
      return

  //x and y are not in same set,so merge them
  if xRoot.rank < yRoot.rank
    // swap xRoot and yRoot
    temp := xRoot
    xRoot := yRoot
    yRoot := temp

  // merge yRoot into xRoot
  yRoot.parent:= xRoot
  if xRoot.rank == yRoot.rank:
    xRoot.rank:= xRoot.rank + 1
```

**Union by size**

```
function Union(x, y)
  xRoot:= Find(x)
  yRoot:= Find(y)

  //x and y are already in the same set
  if xRoot == yRoot
      return

  //x and y are not in same set,so merge them
  if xRoot.size < yRoot.size
    // swap xRoot and yRoot
    temp := xRoot
    xRoot := yRoot
    yRoot := temp

  // merge yRoot into xRoot
  yRoot.parent:= xRoot
  xRoot.size:= xRoot.size + yRoot.size
```

# Algoritmo de Kruskal - Estrutura União-Busca (Disjoint-set)

## Código

```
void make_set(int parent_set[], int rank_set[], int size_set[], int num_nodes){
    /*Create a new set with num_nodes nodes, all of them are their own parents*/
    for (int i=0; i<num_nodes; i++){
        parent_set[i] = i;
        rank_set[i] = 0;
        size_set[i] = 1;
    }
}
```

```
int find_compress(int parent_set[], int x){
    if (parent_set[x] != x){
        parent_set[x] = find_compress(parent_set, parent_set[x]);
    }

    return parent_set[x];
}
```

```
int find_halve(int parent_set[], int x){
    while (parent_set[x] != x){
        parent_set[x] = parent_set[parent_set[x]];
        x = parent_set[x];
    }

    return x;
}
```

```
int find_split(int parent_set[], int x){
    while (parent_set[x] != x){
        int next = parent_set[x];
        parent_set[x] = parent_set[next];
        x = next;
    }

    return x;
}
```

# Algoritmo de Kruskal - Estrutura União-Busca (Disjoint-set)

## Código (cont.)

```c
void union_rank(int parent_set[], int rank_set[], int x, int y){
    int xRoot = find_compress(parent_set, x);
    int yRoot = find_compress(parent_set, y);

    /* x and y are already in the same set */
    if (xRoot == yRoot) return;

    /* x and y are not in same set, so we merge them */
    if (rank_set[xRoot] < rank_set[yRoot]){
        int temp = xRoot;
        xRoot = yRoot;
        yRoot = temp;
    }

    /*Merge yRoot into xRoot*/
    parent_set[yRoot] = xRoot;
    if (rank_set[xRoot]==rank_set[yRoot])
       rank_set[xRoot] = rank_set[xRoot] + 1;
}



void union_size(int parent_set[], int size_set[], int x, int y){

    int xRoot = find_compress(parent_set, x);
    int yRoot = find_compress(parent_set, y);

    /* x and y are already in the same set */
    if (xRoot == yRoot) return;

    /* x and y are not in same set, so we merge them */
    if (size_set[xRoot] < size_set[yRoot]){
        int temp = xRoot;
        xRoot = yRoot;
        yRoot = temp;
    }

    /*Merge yRoot into xRoot*/
    parent_set[yRoot] = xRoot;
    size_set[xRoot] = size_set[xRoot] + size_set[yRoot];
}



/*******************************************
 * Structure that represents a graph edge.
 * u: node u;
 * v: node v;
 *
 *******************************************/
typedef struct Edge{
    int u, v;
    int weight;
} Edge;
```

## Algoritmo de Kruskal – Algoritmo

### Pseudocódigo

```
Function Kruskal(G):
    A = Ø
    foreach v ∈ G.V:
        MAKE-SET(v)

    foreach (u, v) in G.E ordered by weight(u, v), increasing:
        if FIND-SET(u) ≠ FIND-SET(v):
            A = A ∪ {(u, v)}
            UNION(u, v)

    return A
```

### Código

```
void kruskal(int graph[][MAX_NODES], int num_nodes){
    /*Empty set of edges, will hold result*/
    Edge *spanning_tree = (Edge*) malloc(MAX_NODES*(MAX_NODES)-1 * sizeof(int));
    int spanning_tree_size = 0;

    /*Set that will hold all edges in graph sorted by weight*/
    Edge *sorted_edges = (Edge*)  malloc(MAX_NODES*(MAX_NODES)-1 * sizeof(int));
    int sorted_edges_size = 0;

    /*Create new set*/
    int parent_set[num_nodes]; int rank_set[num_nodes]; int size_set[num_nodes];
    make_set(parent_set, rank_set, size_set, num_nodes);

    /*Add the edges and sort them*/
    for (int i=0; i<num_nodes; i++){
        for (int j=0; j<num_nodes; j++){
            if (graph[i][j] != 0){
                sorted_edges[sorted_edges_size].u = i;
                sorted_edges[sorted_edges_size].v = j;
                sorted_edges[sorted_edges_size].weight = graph[i][j];
                sorted_edges_size+=1;
            }
        }
    }

    qsort(sorted_edges, sorted_edges_size, sizeof(struct Edge), comparator);

    /*For each sorted edge*/
    for (int i=0; i<sorted_edges_size; i++){
        int u = sorted_edges[i].u; int v = sorted_edges[i].v;
        if (find_compress(parent_set, u) != find_compress(parent_set, v)){
            /*Add the edge to the spanning tree*/
            spanning_tree[spanning_tree_size] = sorted_edges[i];
            spanning_tree_size+=1;
            union_rank(parent_set, rank_set, u, v);
        }
    }

    /*Print results*/
    printf("Spanning tree: \n");
    for (int i=0; i<spanning_tree_size; i++){
        printf("Edge (%d, %d): %d\n", spanning_tree[i].u, spanning_tree[i].v,
spanning_tree[i].weight);
    }

    /*Free allocated memory*/
    free(spanning_tree);
    free(sorted_edges);
}
```

**Algoritmo para calcular pontos de articulação de um grafo**