Algoritmo de Dijkstra **(um para todos ; arestas de peso não negativo ; guloso)**

- **1º passo:** iniciam-se os valores:

```
para todo v ∈ V[G]
     d[v]← ∞
     π[v] ← -1
d[s] ← 0
```

V[G] é o conjunto de vértices(v) que formam o Grafo G. d[v] é o vetor de distâncias de s até cada v. Admitindo-se a pior estimativa possível, o caminho infinito. π[v] identifica o vértice de onde se origina uma conexão até v de maneira a formar um caminho mínimo.

- **2º passo:** temos que usar o conjunto *Q*, cujos vértices ainda não contém o custo do menor caminho d[v] determinado.

```
Q ← V[G]
```

- **3º passo:** realizamos uma série de **relaxamentos** das arestas, de acordo com o código:

```
enquanto Q ≠ ø
         u ← extrair-mín(Q)                    //Q ← Q - {u}
         para cada v adjacente a u
              se d[v] > d[u] + w(u, v)         //relaxe (u, v)
                 então d[v] ← d[u] + w(u, v)
                       π[v] ← u
                       Q ← Q ∪ {v}
```

```java
   static void dijkstra(long A[][], int n){
      boolean visitados[] = new boolean[n];
      for (int i=1;i<n;i++){
         int min = -1;
         long minVal = Integer.MAX_VALUE;
         for (int j=1;j<n;j++){
            if ((!visitados[j]) && (A[j][0] < minVal)){
                  minVal = A[j][0];
                  min = j;
            }
         }
         visitados[min] = true;
         for (int j=1;j<n;j++){
            if (A[min][0] + A[min][j] < A[j][0]){
               A[j][0] = A[min][0] + A[min][j];
            }
         }
      }
   }
```

Bellman–Ford algorithm **(um para todos ; arestas podem ter peso negativo)**

```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then distance[v] := 0
       else distance[v] := inf
       predecessor[v] := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"
   return distance[], predecessor[]
```

Algoritmo de Floyd-Warshall (todos para todos)

```
ROTINA fw(Inteiro[1..n,1..n] grafo)
    # Inicialização
    VAR Inteiro[1..n,1..n] dist := grafo
    VAR Inteiro[1..n,1..n] pred
    PARA i DE 1 A n
        PARA j DE 1 A n
            SE dist[i,j] < Infinito ENTÃO
                pred[i,j] := i
    # Laço principal do algoritmo
    PARA k DE 1 A n
        PARA i DE 1 A n
            PARA j DE 1 A n
                SE dist[i,j] > dist[i,k] + dist[k,j] ENTÃO
                    dist[i,j] = dist[i,k] + dist[k,j]
                    pred[i,j] = pred[k,j]
    RETORNE dist
```

```
static void Floyd(long A[][], int n){
   for(int x=0;x<n;x++){
      for(int i=0;i<n;i++){
         for(int j=0;j<n;j++){
            if (A[i][x] != Integer.MAX_VALUE && A[x][j] != Integer.MAX_VALUE
                  && A[i][j] > ( A[i][x] + A[x][j])){
               A[i][j]=A[i][x] + A[x][j];
            }
         }
      }
   }
}
```

```java
class No{
  int id;
  int numVizinhos;
  No[] vizinhos;
  boolean visitado;
  int cor;
  int distancia;
  No(){}
  No(int pid){
    id = pid;
  }
}



import java.util.LinkedList;
import java.util.Random;
public class Grafo {
  static final int INFINITO = Integer.MAX_VALUE/4;

  static int numNos;
  static boolean[][] adjacencias;
  static int[][] matrizDistancia;
  static boolean[] visitados;
  static boolean[] cores;
  static No nos[];
  static int numCores = 5;

  static No[] solucao;

  static int[] tempSolucao;
  static int[] melhorSolucao;
  static int valorMelhorSolucao;
  static int valorSolucaoAtual;


  public Grafo(int n, int m){
    numNos = n;
    adjacencias = new boolean[n][n];
    visitados = new boolean[n];
    nos = new No[n];
    matrizDistancia = new int[n][n];
```

```java
// Calcula o numero de componentes conexos de maneira recursiva utilizando a matriz de adjacencias
  private static int calcularComponentesConexos4() {
    for (int i=0;i<numNos;i++) visitados[i] = false;
    int componentes = 0;
    for (int atual=0;atual<numNos;atual++){
      if (!visitados[atual]){
        componentes++;
        visitados[atual] = true;
        visitarVizinhos(atual);
      }
    }
    return componentes;
  }

  private static void visitarVizinhos(int atual) {
    for (int v=0;v<numNos;v++){
      if (!visitados[v] && adjacencias[atual][v]){
        visitados[v] = true;
        visitarVizinhos(v);
      }
    }
  }
```

```java
// Calcula o numero de componentes conexos de maneira iterativa utilizando a matriz de adjacencias
  private static int calcularComponentesConexos3() {
    int componentes = 0;
    LinkedList<Integer> lista = new LinkedList<Integer>();
    int atual = 0;
    for (int i=0;i<numNos;i++){
      atual = i;
      if (!visitados[atual]){
        componentes++;
        visitados[atual] = true;
        lista.add(atual);
        while (!lista.isEmpty()){
          atual = lista.remove();
          for (int v=0;v<numNos;v++){
            if (!visitados[v] && adjacencias[atual][v]){
              visitados[v] = true;
              lista.add(v);
            }
          }
        }
      }
    }
    return componentes;
  }

// Calcula o numero de componentes conexos de maneira recursiva utilizando o arranjo de vizinhos
  private static int calcularComponentesConexos1() {
    int componentes = 0;
    for (int i=0;i<numNos;i++){
      if (!nos[i].visitado){
        componentes++;
        nos[i].visitado = true;
        visitarVizinhos(nos[i]);
      }
    }
    return componentes;
  }

  private static void visitarVizinhos(No atual) {
    for (int v=0;v<atual.numVizinhos;v++){
      if (!atual.vizinhos[v].visitado){
        atual.vizinhos[v].visitado = true;
        visitarVizinhos(atual.vizinhos[v]);
      }
    }
  }
}
```

```java
// Calcula o numero de componentes conexos de maneira iterativa utilizando o arranjo de vizinhos
  private static int calcularComponentesConexos2() {
    for (int i=0;i<numNos;i++){
      nos[i].visitado = false;
    }
    int componentes = 0;
    LinkedList<No> lista = new LinkedList<No>();
    No atual;
    for (int i=0;i<numNos;i++){
      atual = nos[i];
      if (!atual.visitado){
        componentes++;
        atual.visitado = true;
        lista.add(atual);
        while (!lista.isEmpty()){
          atual = lista.remove();

          for (int v=0;v<atual.numVizinhos;v++){
            if (!atual.vizinhos[v].visitado){
              atual.vizinhos[v].visitado = true;
              lista.add(atual.vizinhos[v]);
            }
          }
        }
      }
    }
    return componentes;
  }
```

Busca em Largura e em Profundidade

```java
private static int buscaEmLargura(No inicio, No fim){
  if (inicio == fim) return 0;
  LinkedList<No> ll = new LinkedList<No>();
  inicio.distancia = 0;
  inicio.visitado = true;
  ll.add(inicio);
  No atual;
  while (!ll.isEmpty()){
    atual = ll.remove();
    for (No temp: atual.vizinhos){
      if (!temp.visitado){
        if (temp == fim) return atual.distancia + 1;
        temp.visitado = true;
        temp.distancia = atual.distancia + 1;
        ll.add(temp);
      }
    }
  }
  return -1;
}

private static int buscaEmProfundidade(No atual, No fim, int visitados){
  atual.visitado = true;
  if (atual == fim) return visitados;
  for (No temp: atual.vizinhos){
    if (!temp.visitado){
      int res = buscaEmProfundidade(temp, fim, visitados+1);
      if (res > 0) return res;
    }
  }
  return -1;
}

private static int buscaEmProfundidade(No atual, No fim){
  atual.visitado = true;
  if (atual == fim) return 0;
  for (No temp: atual.vizinhos){
    if (!temp.visitado){
      int res = buscaEmProfundidade(temp, fim) + 1;
      if (res > 0) return res;
    }
  }
  return -1;
}
```

Ciclo Hamiltoniano

```
private static boolean cicloHamiltoniano() {
  solucao = new No[nos.length];
  nos[0].visitado = true;
  solucao[0] = nos[0];
  return cicloHamiltonianoAux(1);
}

private static boolean cicloHamiltonianoAux(int i) {
  if (i==nos.length){
    for (No t: solucao[i-1].vizinhos) if (t == solucao[0]) return true;
    return false;
  }
  for (No t: solucao[i-1].vizinhos){
    if (!t.visitado){
      t.visitado = true;
      solucao[i] = t;
      if (cicloHamiltonianoAux(i+1)) return true;
      t.visitado = false;
    }
  }
  return false;
}
```

Caixeiro Viajante

```
   private static int caixeiroViajante(int inicial ) {
      tempSolucao = new int[numNos];
      melhorSolucao = new int[numNos];
      valorMelhorSolucao = INFINITO;
      valorSolucaoAtual = 0;
      visitados[inicial] = true;
      tempSolucao[0] = inicial;
      caixeiroViajanteAux(1);
      if (valorMelhorSolucao < INFINITO) return valorMelhorSolucao;
      return -1;
   }


   private static void caixeiroViajanteAux(int i) {
      if (valorSolucaoAtual > valorMelhorSolucao) return;
      if (i==numNos){
         int dist = matrizDistancia[tempSolucao[i-1]][tempSolucao[0]] ;
         if (dist < INFINITO && valorSolucaoAtual + dist < valorMelhorSolucao) {
            System.out.println("\t" + valorMelhorSolucao + " -> " +
(valorSolucaoAtual + dist));
            valorMelhorSolucao = valorSolucaoAtual + dist;
            melhorSolucao = tempSolucao.clone();
         }
         return;
      }
      int ultimo = tempSolucao[i-1];
      for (int t=0;t<numNos;t++){
         if (!visitados[t] && matrizDistancia[ultimo][t] < INFINITO){
            visitados[t] = true;
            tempSolucao[i] = t;
            valorSolucaoAtual += matrizDistancia[ultimo][t];
            caixeiroViajanteAux(i+1);
            valorSolucaoAtual -= matrizDistancia[ultimo][t];
            visitados[t] = false;
         }
      }
```