```java
import java.util.Random;
import java.util.Arrays;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

class CCM { // chemin de coût minimum (dans un graphe sans "circuit".)
  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("CCM : chemin de coût minimum");
      System.out.println("Usage : CCM nombre_de_sommets_du_graphe nb_runs_validation_statistique");
      System.out.println("Exemple : CCM 10 1000 (graphes à 10 sommets, validation statistique 1000 ``runs'')");
      System.out.println("Exemple : CCM 10 0 (graphes à 10 sommets, pas de validation statistique)");
      return;
    }
    int n = Integer.parseInt(args[0]); // nombre de sommets du graphe
    LA[] g = grapheAleatoire(n);
    System.out.println("graphe G :");
    afficher(g);
    int[][] MA = calculerMA(g);
    int[] M = MA[0], A = MA[1];
    System.out.println("M = " + Arrays.toString(M));
    System.out.println("A = " + Arrays.toString(A));
    System.out.printf("Coût d'un chemin de coût minimum jusqu'en %d : %d\n", n - 1, M[n - 1]);
    descriptionGraphViz(g, "g.graphviz");
    System.out.println(Arrays.toString(A));
    acm(A, g, n - 1); // affichage d'un chemin coût minimum de 0 à n-1
    System.out.println();

    System.out.println("affichage des chemins de coût minimum de 0 à tous les autres sommets :");
    for (int j = 1; j < n; j++) {
      if (A[j] != j) {
        acm(A, g, j); // affichage d'un chemin coût minimum de 0 à n-1
        System.out.printf(" coût = %d\n", M[j]);
      } else
        System.out.printf("Il n'y a pas de chemin de 0 à %d\n", j);
    }
    System.out.println("Description du graphe dans le fichier g.graphviz");
    System.out.printf("Coût par minimisation locale = %d\n", coutParMinimisationLocale(g));

    /* Validation statistique */
    int nruns = Integer.parseInt(args[1]);
    if (nruns > 0) {
      System.out.printf("Validation statistique à %d runs\n", nruns);
      float[] distancesRelatives = validationStatistique(n, nruns);
      System.out.printf("Médiane des distances relatives : %f\n", medianeIterative(distancesRelatives));
      System.out.printf("Max des distances relatives : %f\n", max(distancesRelatives));
    }
  }

  static int[][] calculerMA(LA[] g) {
    /*
     * retourne un tableau MA={M,A} où M[0:n] est de terme général M[j] = m(j) =
     * coût minimum d'un chemin allant de 0 à j, et A = arg M. S'il n'existe pas de
     * chemin de 0 à j on pose m(j) = infini et arg m(j) = j
     */
    int n = g.length;
    int[] M = new int[n];
    int[] A = new int[n];

    /*
     * Initialisation des valeurs de M et A. Init : M[i] = inf, A[i] = i
     */
    for (int i = 1; i < n; i++) {
      M[i] = Integer.MAX_VALUE / 2;
      A[i] = i;
    }

    /*
     * Calcul des valeurs de M et A. m(j) = min_{v \in pred(j)}(m(i)+c(v,j), m(j))
     * tel que pred(j) est l'ensemble des sommets connectés au sommet j et c(v,j) le
     * cout de l'arc connectant le sommet v au sommet j. On procède par relachement
     * d'un contrainte sur l'ensemble des prédécésseurs du sommet j, une technique
     * abordée lors des premiers TDs de programmation dynamique.
     */
    for (int i = 1; i < n; i++) {
      for (LA la = g[i - 1]; !vide(la); la = la.reste()) {
        int j = la.sommet(), c = la.cout();
        int m = M[i - 1] + c;
```

```java
        if (m < M[j]) {
          M[j] = m;
          A[j] = i - 1;
        }
      }
    }

    return new int[][] { M, A };
  }

  static LA[] symetrique(LA[] g) {
    // retourne le graphe g', symétrique du graphe g.
    int n = g.length;
    LA[] gp = new LA[n];

    // Parcours de tous les arcs du graphe g
    for (int i = 0; i < n; i++) {
      for (LA A = g[i]; !vide(A); A = A.reste()) {
        int j = A.sommet();
        // i : (j, cij) -> j : (i, cij)
        gp[j] = new LA(i, A.cout(), gp[j]);
      }
    }
    return gp;
  }

  static void acm(int[] A, LA[] g, int j) {
    // affiche un chemin de coût minimum du sommet 0 au sommet j

    /*
     * Fonction fortement inspiré de celle du TD6. Le tableau A nous permet de
     * retrouver le chemin optimal j -> j-1 ->  ...  -> 0. Or, on souhait afficher ce
     * chemin de 0 jusqu'à j, càd dans le sens inverse. On a alors recours a une
     * recursion.
     */
    if (j == 0) {
      System.out.print("0");
      return;
    }
    int aj = A[j];
    acm(A, g, aj);
    System.out.printf("--(%d)-->%d", coutArc(aj, j, g), j);
  }

  static int coutArc(int i, int j, LA[] g) {
    /* retourne le coût de l'arc i -> j */

    /*
     * On parcours simplement la liste d'arcs g[i] jusqu'à trouver un arc dirigé
     * vers le sommet j. Si il n'en existe pas, on retourne -1.
     */
    int c = -1;
    for (LA A = g[i]; !vide(A); A = A.reste()) {
      if (A.sommet() == j)
        c = A.cout();
    }
    return c;
  }

  /* minimisation locale */
  static int coutParMinimisationLocale(LA[] g) {
    // calcul du coût d'un chemin de coût local minimum. Retourne le coût d'un
    // chemin obtenu par minimisation locale.

    /*
     * On souhaite parcourir le chemin de coût local minimum. Pour cela, on part de
     * g[0] puis on détermine son sommet connecté de cout minimum avec la fonction
     * coutMin_et_argCoutMin(). On ajoute alors le cout retourné au cout final. On
     * répete ce procédé sur g[jstar], tel quel jstar est l'argument retourné par la
     * fonction précédente.
     */
    int c = 0;
    LA las = g[0];
    while (!vide(las)) {
      int[] min = coutMin_et_argCoutMin(las);
      c += min[0];
      las = g[min[1]];
    }
    return c;
  }

  static int[] coutMin_et_argCoutMin(LA las) { // las : liste d'arcs sortant d'un sommet i
```

```java
    /*
     * soit i ⟶ j* l'arc sortant de i, de coût minimum c(i,j*) Cette fonction
     * retourne c(i,j*) et j*.
     */

    /*
     * On cherche l'arc de cout minimum de la liste d'arcs las. La structure de
     * notre calcul sera très similaire au calcul linéaire du minimum d'un tableau.
     * Seulement, on n'itère pas sur un tableau mais sur la liste d'arcs las et pour
     * accèder aux valeurs à comparer, on utilise la méthode cout() de LA.
     */
    int cijstar = las.cout();
    int jstar = las.sommet();
    for (LA la = las; !vide(la); la = la.reste()) {
      int c = la.cout();
      if (c < cijstar) {
        cijstar = c;
        jstar = la.sommet();
      }
    }

    return new int[] { cijstar, jstar };
  }

  static float[] validationStatistique(int n, int nruns) {
    // validation statistique sur des graphes à n sommets
    float[] distancesRelatives = new float[nruns];
    for (int r = 0; r < nruns; r++) {
      if (r % 1000 == 0)
        System.out.print(".");
      LA[] g = grapheAleatoire(n);
      // calcul de la valeur du chemin de coût minimum
      int[][] MA = calculerMA(g);
      int[] M = MA[0];
      int coutMin = M[n - 1];
      int cml = coutParMinimisationLocale(g);
      float distanceRelative = (float) (cml) / (float) coutMin;
      distancesRelatives[r] = distanceRelative;
    }
    System.out.println();
    return distancesRelatives;
  }

  static class LA { // liste d'arcs.
    int j, cij;
    LA r;

    LA(int j, int cij, LA r) {
      this.j = j;
      this.cij = cij;
      this.r = r;
    }

    int sommet() {
      return j;
    }

    int cout() {
      return cij;
    }

    LA reste() {
      return r;
    }
  }

  static boolean vide(LA l) {
    return l == null;
  }

  static LA[] grapheAleatoire(int n) {
    /*
     * Retourne un graphe aléatoire. Chaque sommet i de [0:n-1] envoie un nombre
     * d'arcs quelconque, supérieur ou égal à 1, vers les sommets de numéros plus
     * élevés, donc vers les sommets de [i+1:n]. Le sommet n-1 n'envoie aucun arc.
     * Ce nombre d'arcs est le degré sortant du sommet i, noté ds(i). Le sommet i
     * envoie au moins un arc vers un sommet de [i+1:n] et au plus un arc vers
     * chacun d'eux. Son degré sortant, ds(i), est ≤ à n - (i+1), situation où le
     * sommet i envoie un arc vers chacun des sommets de numéros supérieurs. On
     * rappelle par ailleurs que le sommet i envoie au moins un arc. Donc : 1 ≤
     * ds(i) ≤ n-(i+1), autrement dit 1 ≤ ds(i) < n-i. Le coût de l'arc i → j est
     * aléatoire : nous choisissons la fonction de coût c(i,j) = (j - i) + hasard(0,
```

```java
     * n+1), où hasard(0,n) est un entier au hasard dans l'intervalle [0:n]. Elle
     * "pénalise" en moyenne les arcs reliant des sommets de numéros très distants.
     * Exemples avec un graphe à n=20 sommets : s'il existe un arc 0 -> j=n-1, sa
     * valeur sera (j-i) + hasard(0,n) = n-1 + hasard(0,n) = 19 + hasard(0,20), donc
     * en moyenne 19 + 10 = 29. S'il existe un arc 0 -> 3, sa valeur sera (3-0) +
     * hasard(0,20), donc en moyenne 3 + 10 = 13.
     */
    LA[] g = new LA[n];
    for (int i = 0; i < n - 1; i++) {
      int[] S = permutation(i + 1, n); // S est une permutation de [i+1:n]
      int dsi = hasard(1, n - i); // ds(i) au hasard, 1 <= ds(i) < n-i
      S = Arrays.copyOfRange(S, 0, dsi); // les sommets vers lesquels i envoie un arc
      for (int j : S) { // pour tout arc i->j
        int cij = (j - i) + hasard(0, n); // coût de l'arc
        g[i] = new LA(j, cij, g[i]); // ajout de l'arc i -> j au graphe g.
      }
    }
    return g;
  }

  static int hasard(int i, int j) {
    Random r = new Random();
    return i + r.nextInt(j - i);
  }

  static int[] permutation(int inf, int sup) {
    int n = sup - inf;
    Random r = new Random();
    int[] T = new int[n];
    for (int i = 0; i < n; i++)
      T[i] = inf + i;
    for (int j = n - 1; j > 0; j--) {
      int i = hasard(0, j);
      permuter(T, i, j);
    }
    return T;
  }

  static void permuter(int[] T, int i, int j) {
    int ti = T[i];
    T[i] = T[j];
    T[j] = ti;
  }

  static void afficher(LA[] g) {
    int n = g.length;
    for (int i = 0; i < n; i++) {
      System.out.printf("%d : ", i);
      for (LA las = g[i]; !vide(las); las = las.reste()) {
        int j = las.sommet(), pij = las.cout();
        System.out.printf("(%d,%d) ", j, pij);
      }
      System.out.println();
    }
  }

  static void descriptionGraphViz(LA[] g, String fileName) {
    /*
     * g est un graphe représenté par une table de liste d'arcs. Ecrit dans le
     * fichier fileName la description du graphe du projet P pour le logiciel
     * graphViz. Cette description est à coller dans la fenêtre gauche du site
     * https://dreampuf.github.io/GraphvizOnline Voir aussi l'excellent site
     * GraphViz Pocket Reference, https://graphs.grevian.org
     */
    try {
      int n = g.length;
      PrintWriter ecrivain;
      ecrivain = new PrintWriter(new BufferedWriter(new FileWriter(fileName)));
      ecrivain.println("digraph g{ rankdir=LR;");
      for (int i = 0; i < n; i++) {
        for (LA las = g[i]; (!vide(las)); las = las.reste()) {
          // las : liste des arcs sortants
          int j = las.sommet(), pij = las.cout();
          ecrivain.println(i + "->" + j + "[label=" + pij + "]" + ";");
        }
      }
      ecrivain.println("}");
      ecrivain.println(
          "/* Description à coller dans la fenêtre gauche du site " +
"https://dreampuf.github.io/GraphvizOnline */");
      ecrivain.println("/* Voir aussi l'excellent site GraphViz Pocket Reference, " +
"https://graphs.grevian.org */");
```

```java
      ecrivain.close();
    } catch (IOException e) {
      System.out.println("Erreur écriture");
    }
  }

  /* Calcul de la médiane */
  static float medianeIterative(float[] T) {
    int n = T.length;
    /*
     * Retourne la valeur médiane de T[0:n]. C'est la valeur du tableau telle que T
     * contient autant de valeurs ≤ à la médiane que de valeurs ≥ à m. Exemple :
     * 0,1,2,3 ⟹ médiane = 1 (indice (4-1)/2 = 3/2 = 1) 0,1,2 ⟹ médiane = 1
     * (indice (3-1)/2 = 2/2 = 1 ) De façon générale, avec la convention 0 ≤ p < n,
     * la valeur médiane est la p = (n-1)/2 ème valeur de T.
     */
    return quickSelectIteratif(1 + (n - 1) / 2, T);
    // ou si l'on préfère : qselIteratif((n-1)/2, T)
  }

  static int segmenter(float[] T, int i, int j) {
    // calcule une permutation des valeurs de T[i:j] qui vérifie
    // T[i:k]⩽ T[k:k+1] < T[k+1:j], et retourne l'indice k.
    // I(k,j') : T[i:k] ⩽ T[k:k+1] < T[k+1:j']
    int h = hasard(i, j);
    permuter(T, i, h);
    int k = i, jp = k + 1; // I(k,j') est vraie
    while (jp < j)
      if (T[k] < T[jp]) // I(k,j'+1) est vraie
        jp = jp + 1;
      else {
        permuter(T, jp, k + 1);
        permuter(T, k + 1, k);
        // I(k+1,j'+1) est vraie
        k = k + 1; // I(k,j'+1) est vraie
        jp = jp + 1; // I(k,j') est vraie
      }
    // I(k,j) vraie, i.e. T[i:k] ≤ T[k:k+1] < T[k+1:j]
    return k;
  }

  static void permuter(float[] T, int i, int j) {
    float ti = T[i];
    T[i] = T[j];
    T[j] = ti;
  }

  public static float quickSelectIteratif(int p, float[] T) {
    int n = T.length;
    // 1 ≤ p ≤ n;
    return qselIteratif(p - 1, T);
  }

  static float qselIteratif(int p, float[] T) {
    int n = T.length; // 0 ⩽ p < n
    int pprime = p, i = 0, j = n; // I(p’, i, j)
    while (!(pprime == 0 && j - i == 1)) { // I(p’,i,j) et non arrêt
      int k = segmenter(T, i, j);
      int pppi = pprime + i;
      if (i ⩽ pppi && pppi < k) // I(pprime, i, k)
      {
        j = k;
      } // I(p’, i, j)
      else if (k ⩽ pppi && pppi < k + 1) // I(pprime-(k-i), k, k+1)
      {
        pprime = pprime - (k - i);
        i = k;
        j = k + 1;
      } // I(p’, i, j)
      else // k+1 ⩽ pppi && pppi < j ) // I(pprime - ((k+1) - i) , k+1, j)
      {
        pprime = pprime - ((k + 1) - i);
        i = k + 1;
      } // I(p’, i, j)
    }
    // I(p’, i, j) et arrêt, donc la p-ème valeur de T[ 0 : n ] est T[i];
    return T[i];
  }

  static float max(float[] T) {
    float max = Integer.MIN_VALUE;
    for (float f : T)
```

```
      if (f > max)
        max = f;
    return max;
  }
}
```