**CHAPITRE 9**

# *APPLICATIONS À LA RÉSOLUTION DE PROBLÈME EN TURBO PASCAL VERSION DOS 7.0*

OBJECTIF

- INTÉGRER LES DIFFÉRENTES NOTIONS ET APPROCHES EXPOSÉES DANS LES CHAPITRES PRÉCÉDENTS DANS UNE DÉMARCHE CONCRÈTE DE RÉSOLUTION DE PROBLÈME EN TURBO PASCAL VERSION DOS 7.0.

Après avoir abordé dans les chapitres précédents les éléments de base de l'algorithmique et du langage PASCAL, il apparaît naturel de chercher à intégrer ces différentes notions et approches dans une démarche concrète de résolution de problème. Dans ce chapitre, nous allons traiter deux problèmes de recherche opérationnelle dont les applications pratiques demeurent nombreuses : il s'agit du problème de la recherche d'un arbre sous-tendant minimum et celui de la localisation de concentrateurs ou d'entrepôts. Ces deux problèmes peuvent être résolus, le premier par les algorithmes de Prim et de Kruskal, le deuxième par les algorithmes ADD et DROP. Afin de faciliter la compréhension de l'exposé, nous allons débuter par un rappel de quelques notions de la théorie des graphes.

## 9.1 NOTIONS DE LA THÉORIE DES GRAPHES

On appelle *graphe* un ensemble de nœuds (ou sommets) reliés entre eux par des arcs (ou arêtes). La figure 9.1 en est une illustration.

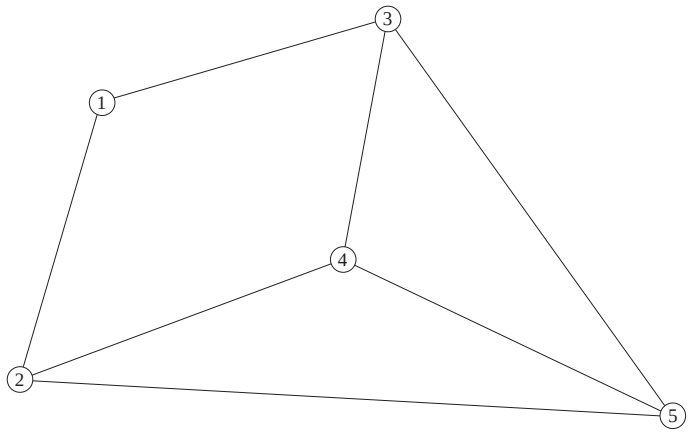


**FIGURE 9.1**
*GRAPHE DE 5 NOEUDS ET DE 7 ARCS.*

Si on désigne par N l'ensemble des nœuds et par A l'ensemble des arcs, on peut noter G = (N, A) le graphe qui en résulte. Pour un ensemble N de *n* nœuds, le nombre maximum d'arcs est désigné par $m_{max} = n(n-1)/2$. Ainsi, dans le cas de la figure 9.1, on a :

N = {1, 2, 3, 4, 5}
*n* = 5
A = {(1, 2), (1, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)}
*m* = 7
$m_{max} = 5(5-1)/2 = 10$.

En effet, le graphe comprend 7 arcs, sur une possibilité de 10, les 3 arcs non considérés étant (1, 4), (1, 5) et (2, 3). Tout graphe construit à partir d'un sous-ensemble des nœuds de G est appelé une *composante* de G.

Tout arc qui possède une origine et une destination non interchangeables est dit *orienté*, la destination étant généralement indiquée par une flèche à l'extrémité correspondante de l'arc. Dans le cas contraire, l'arc est dit *non orienté*.

On dit d'un graphe qu'il est *orienté* lorsque tous ses arcs le sont. Dans le cas de la figure 9.2 qui en est un exemple, l'ensemble des arcs orientés est :

A = {(1, 2), (2, 5), (3, 1), (3, 4), (4, 2), (5, 3), (5, 4)}

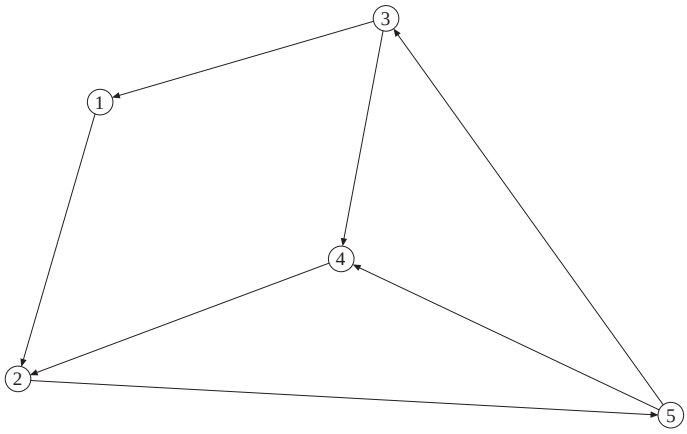


**FIGURE 9.2**
*GRAPHE ORIENTÉ.*

Lorsque tous les arcs d'un graphe sont non orientés, le graphe en question est dit non orienté. C'est le cas du graphe de la figure 9.1.

On dit d'un arc qu'il est *incident à un nœud* si cet arc admet ce nœud comme origine ou destination. Dans le cas de la figure 9.1, l'arc (1, 2) est à la fois incident au nœud 1 et au nœud 2. Ainsi, on appelle *degré d'incidence d'un nœud* le nombre d'arcs incidents à ce nœud; par exemple, les nœud 2, 3, 4 et 5 de la figure 9.1 sont de degré 3, alors que le nœud 1 est de degré 2. On désigne par *degré d'un graphe* le degré d'incidence du nœud le plus faible de ce graphe; à titre d'exemple, le graphe de la figure 9.1 est de degré 2, le plus faible degré d'incidence d'un nœud étant de degré 2.

On appelle *chemin* i-j *entre deux nœud* i *et* j *d'un graphe* la suite d'arcs permettant, à partir du nœud *i*, d'atteindre le nœud *j*. Par exemple, en considérant le graphe de la figure 9.1, pour aller du nœud 1 au nœud 5, on peut utiliser l'une ou l'autre des séquences d'arcs suivantes :

(1, 2)–(2, 5)
(1, 2)–(2, 4)–(4, 5)
(1, 2)–(2, 4)–(4, 3)–(3, 5)
(1, 3)–(3, 5)
(1, 3)–(3, 4)–(4, 5)
(1, 3)–(3, 4)–(4, 2)–(2, 5)

qui sont autant de chemins reliant les deux nœuds 1 et 5.

Chaque arc est muni d'une longueur qui n'est autre que la distance euclidienne entre ses nœuds d'origine et de destination. En conséquence, on désigne par *longueur d'un chemin* la somme des longueurs d'arcs qui composent celui-ci. Un chemin dont l'origine se confond à la destination constitue un *cycle*. Par rapport au graphe de la figure 9.1, le chemin formé par la suite d'arcs (1, 3)–(3, 4)–(4, 2)–(2, 1), par exemple, est un cycle.

On dit d'un graphe qu'il est *connexe* lorsque chacune de ses paires de nœuds est reliée par au moins un chemin. On parle alors de *connexité simple* ou de *1–connexité*. Dans le cas où il existe au moins *k* chemins distincts entre chaque paire de nœuds du graphe ($k > 1$), on dit que le graphe est de degré de connexité *k* ou qu'il est *k–connexe*. Par exemple, le graphe de la figure 9.1 est 2–connexe; dans un graphe connexe, aucun nœud n'est de degré d'incidence nul.

On désigne par *arbre* ou *arborescence* un graphe connexe sans cycle; le degré de connexité d'un tel graphe est donc égal à 1. Un arbre est dit *sous-tendant minimum* lorsqu'il relie tous les nœuds au moyen d'arcs dont la somme totale des longueurs est la plus petite possible. La figure 9.3 en est une illustration.

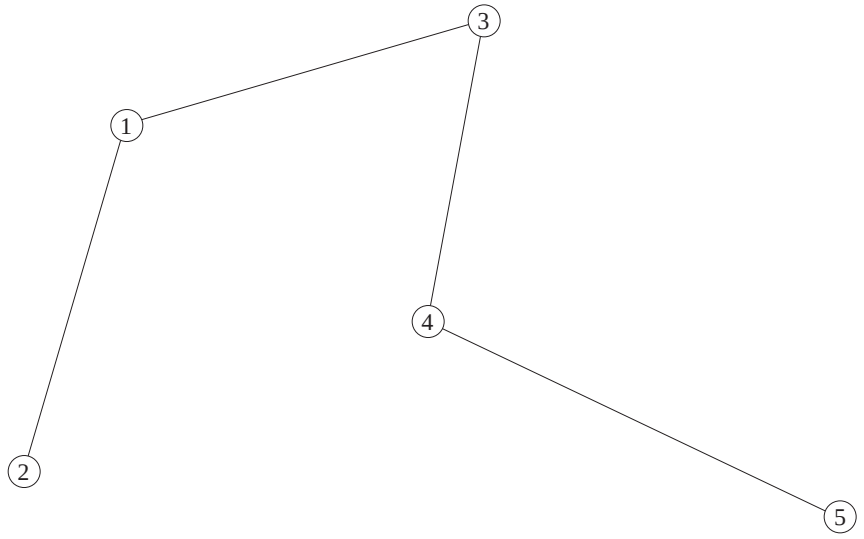


**FIGURE 9.3**
*ARBRE SOUS-TENDANT MINIMUM.*

## 9.2 LA CONSTRUCTION D'UN ARBRE SOUS-TENDANT MINIMUM

Considérons un ensemble de $n$ nœuds numérotés de 1 à $n$ et définissons le coût d'un arbre comme la longueur totale des liaisons qui composent cet arbre. La construction d'un arbre sous-tendant minimum (ASM) pour cet ensemble de nœuds consiste à déterminer un tracé qui permet de relier tous ces nœuds en un arbre de coût minimum. Un tel problème, dont la formulation peut être modifiée pour tenir compte de certaines contraintes, est généralement résolu soit par l'algorithme de Prim ou par celui de Kruskal.

### 9.2.1 Les grandes étapes de l'algorithme de Prim

Considérons un ensemble I de nœuds déjà inclus dans l'arbre en construction et un ensemble NI de nœuds non encore inclus. L'algorithme de Prim se définit par les étapes suivantes :

Étape 1 : Initialiser : I = { }
NI = {1, ..., *n*}.

Étape 2 : Inclure le nœud 1 dans l'arbre, comme premier nœud; et par suite :
I = {1}
NI = {2, 3, ..., *n*}.

Étape 3 : Si l'arbre est connexe, alors aller à l'étape 5.

Étape 4 : Trouver le nœud *i* dont la distance à un nœud quelconque *j* de I est minimale, relier ce nœud *i* au nœud *j*; faire :
I = I + {*i*}
NI = NI – {*i*},
puis retourner à l'étape 3.

Étape 5 : Arrêt.

*Exemple 9.1*

Considérons un ensemble de 15 nœuds de coordonnées cartésiennes respectives :

| | | |
|---|---|---|
| 1. (35, 45) | 6. (35, 25) | 11. (65, 60) |
| 2. (5, 20) | 7. (45, 60) | 12. (65, 20) |
| 3. (10, 60) | 8. (50, 10) | 13. (75, 45) |
| 4. (20, 40) | 9. (55, 40) | 14. (85, 65) |
| 5. (30, 70) | 10. (60, 80) | 15. (90, 15) |

La figure 9.4 montre le résultat de l'application de l'algorithme de Prim sur l'ensemble de nœuds du graphe. Les chiffres entre parenthèses indiquent l'ordre d'insertion des liaisons dans l'ASM.

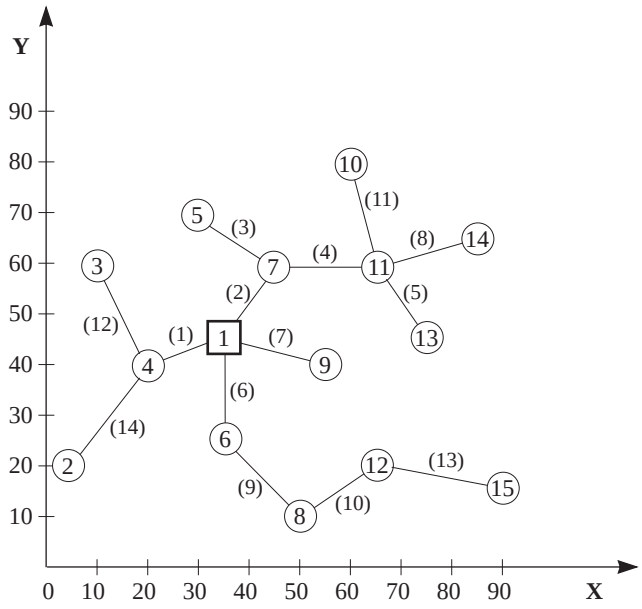


**FIGURE 9.4**
*EXEMPLE DE L'ALGORITHME DE PRIM.*

### 9.2.2 Les notes d'implantation de l'algorithme de Prim

L'algorithme de Prim construit l'arbre sous-tendant minimum en faisant croître une composante connexe à partir d'un nœud donné (ici le nœud 1). L'ensemble I contient les nœuds de cette composante connexe. À chaque étape de l'algorithme, on doit trouver un arc de longueur minimum joignant I à NI. Au lieu de faire cette recherche parmi toutes les paires ($i$, $j$) avec $i \in$ I et $j \in$ NI, on peut maintenir deux tableaux *PlusPresDe* et *DistanceMin* qui auront la propriété invariante suivante :

Pour chaque nœud $i \in$ NI :

– s'il existe au moins un nœud de I adjacent à $i$, alors DistanceMin[$i$] est la valeur de l'arc de coût minimum joignant $i$ à un élément de I et PlusPresDe[$i$] représente un des nœuds de I pour lequel :

ValeurArc($i$, PlusPres[$i$]) = DistanceMin[$i$]

– s'il n'existe pas d'élément de I adjacent à *i*, on l'indique en donnant la valeur infini (∞) à DistanceMin[*i*].

Cela impose toutefois une restriction quant aux valeurs des arcs du graphe : elles devront être strictement inférieures à l'infini, c'est-à-dire, en pratique, une valeur limite qui dépend de la machine. Ainsi, à chaque étape, on recherche le nœud $i \in$ NI qui minimise DistMin[*i*], et on ajoute à l'ASM l'arc joignant *i* et PlusPresDe[*i*]. Puis, on met à jour I, NI, PlusPresDe et DistanceMin.

Toutes les fois que le corps de la boucle principale est exécuté, un arc est ajouté à l'ASM. On devra donc ajouter $(n - 1)$ arcs, car un arbre qui a $n$ nœuds possède $n - 1$ arcs.

Le programme de l'exemple 9.2 est une implantation de l'algorithme de Prim. Ce programme est écrit en Turbo Pascal qui accepte des identificateurs de longueur supérieure à celle que permet le PASCAL (standard).

*Exemple 9.2*

```
PROGRAM Prim(INPUT, OUTPUT);
(*
*  Ce programme fait une mise en oeuvre de l'algorithme de Prim.
*  Une interface par menu permet a un usager de definir les parametres
*  et d'appliquer l'algorithme.
*)

(*  Definition d'un type graphe value non oriente *)
CONST
      Infini = 10.0E20;
      MaxNoeud = 20; (* Nombre maximum de noeuds *)
TYPE
      TypeValeur = REAL; (* Type de valeur des arcs *)
      Noeud = 1..MaxNoeud;
      Graphe =    RECORD
                        nbNoeuds : 0..MaxNoeud;
                        MatAdjacence : ARRAY[Noeud, Noeud] OF BOOLEAN;
                        MatValeur : ARRAY[Noeud, Noeud] OF TypeValeur;
                  END;
```

```
PROCEDURE InitGraphe(VAR g : Graphe; nbNoeuds : INTEGER);
(* But : Initialise g comme un graphe de nbNoeuds sans arc. *)
VAR i, j : Noeud;
BEGIN
        g.nbNoeuds := nbNoeuds;
        FOR i := 1 TO nbNoeuds DO
                FOR j := 1 TO nbNoeuds DO
                        g.MatAdjacence[i, j] := FALSE;
END; (* InitGraphe *)

PROCEDURE AjouterArc(VAR g : Graphe; n1, n2 : Noeud; val : TypeValeur);
(* But : Ajoute entre les noeuds n1 et n2 de g un arc de valeur val. *)
BEGIN
        WITH g DO
        BEGIN
                MatAdjacence[n1, n2] := TRUE;
                MatAdjacence[n2, n1] := TRUE;
                MatValeur[n1, n2] := val;
                MatValeur[n2, n1] := val;
        END;
END;

PROCEDURE RetirerArc(VAR g : Graphe; n1, n2 : Noeud);
(* Enleve de g l'arc joignant n1 et n2. *)
BEGIN
        g.MatAdjacence[n1, n2] := FALSE;
        g.MatAdjacence[n2, n1] := FALSE;
END;

FUNCTION Adjacent(g : Graphe; n1, n2 : Noeud) : BOOLEAN;
(* But :  Donne TRUE si un arc joint n1 et n2;
*          donne FALSE sinon. *)
BEGIN
        Adjacent := g.MatAdjacence[n1, n2];
END;

FUNCTION ValeurArc(VAR g : Graphe; n1, n2 : Noeud) : TypeValeur;
(* But : Donne la valeur de l'arc joignant n1 et n2.
*  Suppose : Adjacent(g, n1, n2) = TRUE. *)
BEGIN
        ValeurArc := g.MatValeur[n1, n2];
END;
```

```
FUNCTION NombreDeNoeuds(VAR g : Graphe) : INTEGER;
(* But : Donne le nombre de noeuds de g. *)
BEGIN
        NombreDeNoeuds := g.NbNoeuds;
END;

FUNCTION SommeValeurArc(VAR g : Graphe) : REAL;
(* But : Donne la somme des valeurs des arcs de g. *)
VAR
        i, j : INTEGER;
        Total : REAL;
BEGIN
        Total := 0.0;
        FOR i := 1 TO g.NbNoeuds DO
                FOR j := i + 1 TO g.NbNoeuds DO
                        IF Adjacent(g, i, j) THEN
                                Total := Total + ValeurArc(g, i, j);
        SommeValeurArc := Total;
END;

(*———————————————— Algorithme de Prim ————————————————*)

PROCEDURE ProcPrim(VAR g : Graphe; VAR ASM : Graphe);
(* But : Donne dans ASM, un arbre recouvrant de cout minimum de g.
*  Suppose : g est connexe. *)
VAR
        PlusPresDe : ARRAY[1..MaxNoeud] OF Noeud;
        DistanceMin : ARRAY[1..MaxNoeud] OF TypeValeur;
        NonInclus : SET OF Noeud;
        i, k, m : Noeud;
        min : TypeValeur;

(*
*  Implicitement Inclus = [1..NombreDeNoeuds(g)] – NonInclus,
*  represente les noeuds de ASM accessibles a partir du noeud 1.
*       Les vecteurs DistanceMin et PlusPresDe possedent la propriete que
*       pour chaque noeud s dans NonInclus, DistanceMin[s] est la valeur
*       de l'arc de valeur minimum joignant ce noeud a un noeud quelconque de Inclus;
*       PlusPresDe[s] represente ce noeud, s'il n'y a aucun noeud de Inclus adjacent a s
*       alors DistanceMin[s] = Infini.
*)
```

```
BEGIN (* ProcPrim *)
    InitGraphe(ASM, NombreDeNoeuds(g));
    NonInclus := [2..NombreDeNoeuds(g)]; (* Implicitement Inclus = [1] *)
    (* Initialisation de DistanceMin et PlusPresDe par rapport a [1] *)
    FOR i := 2 TO NombreDeNoeuds(g) DO
        IF Adjacent(g, i, 1) THEN
        BEGIN
            DistanceMin[i] := ValeurArc(g, 1, i);
            PlusPresDe[i] := 1;
        END
        ELSE
            DistanceMin[i] := Infini;
    FOR m := 2 TO NombreDeNoeuds(g) DO  (* Repeter (NombreDeNoeuds(g) – 1) fois *)
    BEGIN
        (* Recherche de l'arc (k, PlusPresDe[k]) de valeur minimum joignant
        *  un noeud k de NonInclus a un noeud de Inclus. *)
        min := Infini;
        FOR i := 2 TO NombreDeNoeuds(g) DO
            IF (i IN NonInclus) AND (DistanceMin[i] < min) THEN
            BEGIN
                min := DistanceMin[i];
                k := i;
            END;
        AjouterArc(ASM, k, PlusPresDe[k], min);
        NonInclus := NonInclus – [k];
        (* Mise a jour de DistanceMin et PlusPresDe *)
        FOR i := 2 TO NombreDeNoeuds(g) DO
            IF (i IN NonInclus) AND Adjacent(g, k, i) THEN
                IF ValeurArc(g, k, i) < DistanceMin[i] THEN
                BEGIN
                    DistanceMin[i] := ValeurArc(g, k, i);
                    PlusPresDe[i] := k;
                END;
    END; (* NonInclus = [ ] c'est-a-dire Inclus = [1..NombreDeNoeuds(g)] *)
END; (* ProcPrim *)

(*———————— Procedures utilitaires pour l'interface ————————*)

PROCEDURE FaireUnePause;
(* But : Fait une pause jusqu'a ce que l'usager presse sur retour. *)
BEGIN
    WRITE('                                          ');
    WRITE(  '        appuyez sur retour');
    READLN;
END; (* FaireUnePause *)
```

```
PROCEDURE SauterLigne(n : INTEGER);
(* But : Saute n ligne a l'ecran. *)
VAR
        i : INTEGER;
BEGIN
        FOR i := 1 TO n DO
                WRITELN;
END; (* SauterLigne *)

PROCEDURE ViderEcran;
(* But : Fait defiler l'ecran jusqu'a ce qu'il n'y ait plus rien d'affiche. *)
BEGIN
        SauterLigne(25);
END; (* ViderEcran *)

(*———————————————— Test de Prim ————————————————*)

(* Teste l'algorithme de Prim sur un graphe complet genere a partir
*  des coordonnees de points. *)
CONST
        MaxCoordonnees = MaxNoeud;
TYPE
        Coordonnee =    RECORD
                                Abscisse : REAL;
                                Ordonnee : REAL;
                        END;
        VectCoord = ARRAY[1..MaxCoordonnees] OF Coordonnee;
        TypeSpecCoord = RECORD
                                NbCoordonnees : INTEGER;
                                Coordonnees : VectCoord;
                         END;

FUNCTION Distance(x1, x2 : Coordonnee) : REAL;
(* But : Donne la distance euclidienne entre les coordonnees x1 et x2. *)
BEGIN
        Distance := SQRT(SQR(x1.abscisse – x2.abscisse) + SQR(x1.ordonnee – x2.ordonnee));
END; (* Distance *)

FUNCTION CoordPresente(VAR v : VectCoord ; n1, n2 : INTEGER;
                                abs, ord : REAL) : BOOLEAN;
(* But : Teste si  pour une coordonnee de v[n] (n1 <= n <= n2)
*  v[n].Abscisse = abs et v[n].Ordonnee = ord.  *)
VAR
        k : INTEGER;
```

```
BEGIN
        CoordPresente := FALSE;
        FOR k := n1 TO n2 DO
                IF (v[k].Abscisse = abs) AND (v[k].Ordonnee = ord) THEN
                        CoordPresente := TRUE;
END; (* CoordPresente *)

PROCEDURE SaisirCoordonnees(VAR Spec : TypeSpecCoord);
(* But :    – Saisit le nombre de coordonnees;
*            – saisit les coordonnees dans Spec. *)

PROCEDURE SaisiCoordonnee(i : INTEGER; VAR abscisse, ordonnee : REAL;
(* But : Saisit et donne la i-ieme coordonnee. *)
BEGIN
        SauterLigne(2);
        WRITELN('Coordonnee ', i : 3);
        WRITE(  '                abscisse  : '); READLN(abscisse);
        WRITE(  '                ordonnee  : '); READLN(ordonnee);
END;

PROCEDURE SaisiToutesCoordonnees(VAR Spec : TypeSpecCoord; n : INTEGER);
(* But : Saisir dans Spec des coordonnees 1.. n.  *)
VAR
        i : INTEGER;
        abscisse, ordonnee : REAL;
BEGIN
        SauterLigne(3);
        WRITELN('Entrez les coordonnees des noeuds ');
        SauterLigne(2);
        i := 1;
        WHILE i <= n DO
        BEGIN
                SaisiCoordonnee(i, abscisse, ordonnee);
                IF NOT CoordPresente(Spec.Coordonnees, 1, i – 1, abscisse, ordonnee) THEN
                BEGIN
                        Spec.Coordonnees[i].Abscisse := abscisse;
                        Spec.Coordonnees[i].Ordonnee := ordonnee;
                        i := i + 1
                END
                ELSE
                        WRITELN('Coordonnee deja definie, recommencez s.v.p.');
        END;
END;
```

```
BEGIN
        ViderEcran;
        WRITELN('           Specification des coordonnees');
        WRITELN;
        WRITE('Entrez le nombre de noeuds (max. ', MaxNoeud : 2,') : ');
        READLN(Spec.NbCoordonnees);
        SaisiToutesCoordonnees(Spec, Spec.nbCoordonnees);
END; (* SaisirCoordonnees *)

PROCEDURE AfficherCoordonnees(Spec : TypeSpecCoord);
(* But : Affiche les coordonnees de Spec. *)
VAR
        i : INTEGER;
BEGIN
        ViderEcran;
        WITH Spec DO
        BEGIN
                WRITELN;
                WRITELN('                       Coordonnees des noeuds ');
                SauterLigne(2);
                        FOR i := 1 TO NbCoordonnees DO
                        BEGIN
                                IF ((i MOD 16) = 0) THEN
                                        FaireUnePause;
                                WRITELN(' ', i : 2,'    ', '(', Coordonnees[i].abscisse : 12 : 2,
                                                ', ', Coordonnees[i].ordonnee : 12 : 2,')');
                        END;
        END; (* WITH *)
        FaireUnePause;
END; (* AfficherCoordonnees *)

PROCEDURE AfficherGraphe(g : Graphe);
(* But :   –Affiche les arcs de g et leurs valeurs;
*           – affiche le total des valeurs des arcs. *)
VAR
        i, j : INTEGER;
        k : INTEGER;    (* Compteur pour pause *)
```

```
BEGIN
        k := 0;
        FOR i := 1 TO NombreDeNoeuds(g) DO
                FOR j := i + 1 TO NombreDeNoeuds(g) DO
                        IF Adjacent(g, i, j) THEN
                        BEGIN
                                k := k + 1;
                                IF ((k MOD 16) = 0) THEN
                                        FaireUnePause;
                                WRITELN('arc(', i : 2, ', ', j : 2, ') valeur : ', ValeurArc(g, i, j) : 12 : 2);
                        END;
        SauterLigne(1);
        WRITELN('            cout total : ', SommeValeurArc(g) : 12 : 2);
END; (* AfficherGraphe *)

PROCEDURE AppliquerPrim(VAR Spec : TypeSpecCoord);
(* But : – Definit un graphe g complet ou les valeurs des arcs sont egales
*           aux distances entre les coordonnees correspondantes;
*       – applique l'algorithme de Prim sur g -> ASM;
*       – affiche les arcs de ASM et la somme de leurs valeurs. *)
VAR
        i, j : INTEGER;
        g, ASM : Graphe;
BEGIN
        InitGraphe(g, Spec.NbCoordonnees);
        FOR i := 1 TO Spec.NbCoordonnees DO
                FOR j := i + 1 TO Spec.NbCoordonnees DO
                        AjouterArc(g, i, j, Distance(Spec.Coordonnees[i], Spec.Coordonnees[j]));
        ProcPrim(g, ASM);
        ViderEcran;
        WRITELN('Arcs de l''arbre recouvrant de cout minimum suivant l''algorithme de Prim');
        SauterLigne(2);
        AfficherGraphe(ASM);
        SauterLigne(2);
        FaireUnePause;
END; (* AppliquerPrim *)

PROCEDURE TesterPrim;
(* But : Menu permettant a l'usager de :
*       1. definir les coordonnees des noeuds d'un graphe
*       2. afficher les coordonnees du graphe courant
*       3. trouver l'ASM a l'aide de PRIM et de l'afficher *)
```

```
VAR
      Choix : CHAR; (* Choix de l'usager *)
      Spec : TypeSpecCoord; (* Contient les coordonnees. *)
      CoordSontDefinies : BOOLEAN; (* A-t-on defini des coordonnees? *)

PROCEDURE SignifierCoordNonDefinies;
(* Affiche un message indiquant qu'on doit definir des coordonnees. *)
BEGIN
      ViderEcran;
      WRITELN('Vous devez d"abord definir les coordonnees. ');
      FaireUnePause;
END;
BEGIN
      CoordSontDefinies := FALSE;
      REPEAT
            ViderEcran;
            WRITELN('                  Test de Prim');
            SauterLigne(2);
            WRITELN('            1. Definir les coordonnees');
            WRITELN;
            WRITELN('            2. Afficher les coordonnees');
            WRITELN;
            WRITELN('            3. Resultat de Prim');
            WRITELN;
            WRITELN('            0. Terminer');
            SauterLigne(5);
            WRITE('Entrez votre choix : ');
            READLN(Choix);
            CASE Choix OF
                  '1' :  BEGIN
                              SaisirCoordonnees(Spec);
                              CoordSontDefinies := TRUE;
                         END;
                  '2' :  IF CoordSontDefinies THEN
                              AfficherCoordonnees(Spec)
                         ELSE
                              SignifierCoordNonDefinies;
                  '3' :  IF CoordSontDefinies THEN
                              AppliquerPrim(Spec)
                         ELSE
                              SignifierCoordNonDefinies;
                  '0' : ;
            END;
      UNTIL Choix = '0';
END; (* TesterPrim *)
```

```
(*———————— Programme principal ———————— *)
BEGIN
      TesterPrim;
      ViderEcran;
END.
```

### 9.2.3 Les grandes étapes de l'algorithme de Kruskal

Considérons un tableau TabTri destiné à contenir les longueurs des liaisons triées en ordre croissant. L'algorithme de Kruskal comprend les étapes suivantes :

Étape 1 : Calculer les distances entre toutes les paires de noeuds.

Étape 2 : Trier ces distances ou liaisons potentielles dans l'ordre croissant de leur longueur et les placer dans TabTri.

Étape 3 : Inclure dans l'arbre la liaison (*i*, *j*), la plus courte de TabTri.

Étape 4 : Enlever la liaison (*i*, *j*) de TabTri.

Étape 5 : Si l'arbre est connexe, alors aller à l'étape 9.

Étape 6 : Retenir pour insertion la liaison (*i*, *j*), la plus courte de TabTri.

Étape 7 : Si l'insertion de (*i*, *j*) conduit à la formation d'un cycle, alors retourner à l'étape 4.

Étape 8 : Insérer la liaison (*i*, *j*) dans l'arbre et retourner à l'étape 4.

Étape 9 : Arrêt.

La figure 9.5 montre le résultat de l'application de l'algorithme de Kruskal sur l'ensemble de nœuds. Les chiffres entre parenthèses indiquent l'ordre d'insertion des liaisons dans l'ASM dont les nœuds ont été définis à l'exemple 9.1.

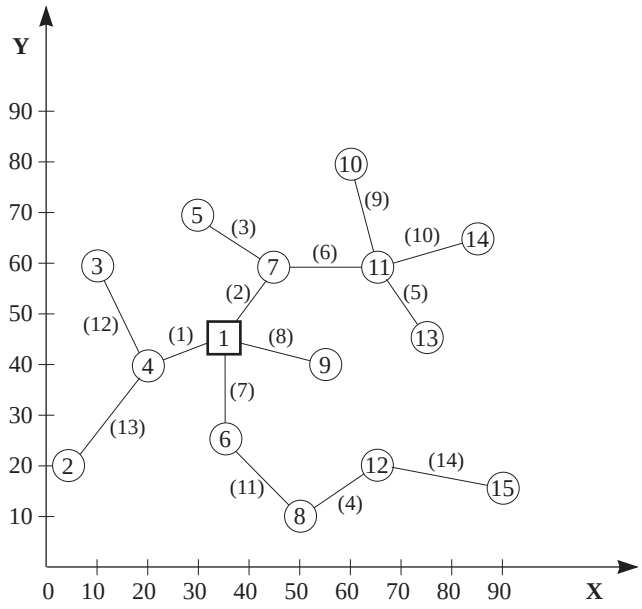


**FIGURE 9.5**
*EXEMPLE D'APPLICATION DE L'ALGORITHME DE KRUSKAL.*

### 9.2.4 Les notes d'implantation de l'algorithme de Kruskal

L'algorithme de Kruskal ajoute des arcs à l'ASM suivant l'ordre croissant de leur longueur, l'ajout d'un arc ne devant pas induire de cycle. On peut montrer que, dans un graphe sans cycle, l'ajout d'un arc ne conduit pas à la formation de cycle si et seulement si cet arc joint deux composantes connexes différentes. Cette propriété nous fournit un moyen simple de tester si un arc peut être ajouté. Il faut cependant pouvoir déterminer si deux nœuds ou sommets appartiennent à des composantes connexes différentes. Ceci se réalise simplement en utilisant un vecteur Composantes[1..*n*] ayant la propriété invariante suivante :

> Deux nœud *i* et *j* de l'ASM font partie de la même composante connexe si et seulement si :
>
> Composantes[*i*] = Composantes[*j*].

Au départ, l'ASM ne possède aucun arc; il contient donc $n$ composantes connexes dégénérées, d'un seul nœud chacune. Nous initialisons alors Composantes de telle sorte que Composantes[$i$] = $i$, avec $i \in \{1, 2, ..., n\}$. Puis, lorsqu'un arc est ajouté, il joint deux nœuds ou sommets $s_1$ et $s_2$ appartenant à des composantes connexes différentes (disons $C_1$ et $C_2$). Il faudra alors joindre les composantes $C_1$ et $C_2$.

Toutes les fois qu'un arc est ajouté, le nombre de composantes connexes diminue de 1. L'exécution de la boucle principale se terminera donc lorsqu'il n'y aura plus qu'une seule composante connexe (un arbre étant un graphe connexe sans cycle).

Le programme 9.3 constitue une implantation de l'algorithme de Kruskal. Ce programme est écrit en Turbo Pascal qui accepte des identificateurs de longueur supérieure à celle que permet le PASCAL standard.

*Exemple 9.3*

```
PROGRAM Kruskal(INPUT, OUTPUT);
(* Ce programme fait une mise en oeuvre des algorithmes Kruskal.
*  Une interface par menu permet a un usager de definir les parametres
*  et d'appliquer l'algorithme. *)

(*————————— Definition d'un type graphe value non oriente —————————*)

CONST
      Infini = 10.0E20;
      MaxNoeud = 20; (* Nombre maximum de noeuds *)
TYPE
      TypeValeur = REAL;       (* Type de valeur des arcs *)
      Noeud = 1..MaxNoeud;
      Graphe =  RECORD
                        nbNoeuds : 0..MaxNoeud;
                        MatAdjacence : ARRAY[Noeud, Noeud] OF BOOLEAN;
                        MatValeur : ARRAY[Noeud, Noeud] OF TypeValeur;
                END;

PROCEDURE InitGraphe(VAR g : Graphe; nbNoeuds : INTEGER);
(* But : Initialise g comme un graphe de nbNoeuds sans arc. *)
VAR i, j : Noeud;
```

```
BEGIN
        g.nbNoeuds := nbNoeuds;
        FOR i := 1 TO nbNoeuds DO
                FOR j := 1 TO nbNoeuds DO
                        g.MatAdjacence[i, j] := FALSE;
END; (* InitGraphe *)

PROCEDURE AjouterArc(VAR g : Graphe; n1, n2 : Noeud; val : TypeValeur);
(* But : Ajoute entre les noeuds n1 et n2 de g un arc de valeur val.  *)
BEGIN
        WITH g DO
        BEGIN
                MatAdjacence[n1, n2] := TRUE;
                MatAdjacence[n2, n1] := TRUE;
                MatValeur[n1, n2] := val;
                MatValeur[n2, n1] := val;
        END;
END;

PROCEDURE RetirerArc(VAR g : Graphe; n1, n2 : Noeud);
(* Enleve de g l'arc joignant n1 et n2. *)
BEGIN
        g.MatAdjacence[n1, n2] := FALSE;
        g.MatAdjacence[n2, n1] := FALSE;
END;

FUNCTION Adjacent(g : Graphe; n1, n2 : Noeud) : BOOLEAN;
(* But :  Donne TRUE si un arc joint n1 et n2;
*          donne FALSE sinon. *)
BEGIN
        Adjacent := g.MatAdjacence[n1, n2];
END;

FUNCTION ValeurArc(VAR g : Graphe; n1, n2 : Noeud) : TypeValeur;
(* But : Donne la valeur de l'arc joignant n1 et n2.
*  Suppose : Adjacent(g, n1, n2) = TRUE *)
BEGIN
        ValeurArc := g.MatValeur[n1, n2];
END;

FUNCTION NombreDeNoeuds(VAR g : Graphe) : INTEGER;
(* But : Donne le nombre de noeuds de g . *)
BEGIN
        NombreDeNoeuds := g.NbNoeuds;
END;
```

```
FUNCTION SommeValeurArc(VAR g : Graphe) : REAL;
(* But : Donne la somme des valeurs des arcs de g.  *)
VAR
      i, j : INTEGER;
      Total : REAL;
BEGIN
      Total := 0.0;
      FOR i := 1 TO g.NbNoeuds DO
            FOR j := i + 1 TO g.NbNoeuds DO
                  IF Adjacent(g, i, j) THEN
                        Total := Total + ValeurArc(g, i, j);
      SommeValeurArc := Total;
END;

(*———————————— Algorithme de Kruskal ————————————*)

PROCEDURE ProcKruskal(VAR g : Graphe; VAR ASM : Graphe);
(* But : Donne dans ASM, un arbre recouvrant de cout minimum de g.
*  Suppose : g est connexe. *)
TYPE
      Arc = RECORD
                  noeud1, noeud2 : Noeud;
                  valeur : TypeValeur;
            END;
      TypeVecteurArc = ARRAY[0..(MaxNoeud * MaxNoeud)] OF Arc;
VAR
      NbComposantes : INTEGER; (* Nombre de composantes connexes *)
      Composantes : ARRAY[Noeud] OF Noeud;
      VectArcs : TypeVecteurArc;
      NbArcsTot : INTEGER;
      i, j, k : INTEGER;
      compTemp : INTEGER;

PROCEDURE TrierArcs(VAR v : TypeVecteurArc; n : INTEGER);
(* Tri par insertion *)
VAR
      i, j : INTEGER;
      temp : Arc;
      ValTemp : TypeValeur;
```

```
BEGIN
        v[0].valeur := – Infini;
        FOR i := 2 TO n DO
        BEGIN
                j := i;
                temp := v[i];
                ValTemp := v[i].valeur;
                WHILE ValTemp < v[j – 1].valeur DO
                BEGIN
                        v[j] := v[j – 1];
                        j := j – 1;
                END;
                v[j] := temp;
        END;
END; (* TrieInsertion *)

(* Le vecteur Composantes sert a contenir les noeuds des composantes connexes.
*  i et j appartiennent a la meme composante connexe
*  si Composantes[i] = Composantes[j]. *)

BEGIN (* ProcKruskal *)
        InitGraphe(ASM, NombreDeNoeuds(g));
        (* Initialisation de VectArcs : vecteur trie des arcs de g.  *)
        NbArcsTot := 0;
        FOR i := 1 TO NombreDeNoeuds(g) DO
                FOR j := i + 1 TO NombreDeNoeuds(g) DO
                        IF Adjacent(g, i, j) THEN
                        BEGIN
                                NbArcsTot := NbArcsTot + 1;
                                VectArcs[NbArcsTot].noeud1 := i;
                                VectArcs[NbArcsTot].noeud2 := j;
                                VectArcs[NbArcsTot].valeur := ValeurArc(g, i, j);
                        END;
        TrierArcs(VectArcs, NbArcsTot);

(* Initialisation de Composantes a NombreDeNoeuds(g) composantes connexes
*  d'un seul noeud chacune. *)
        FOR i := 1 TO NombreDeNoeuds(g) DO
                Composantes[i] := i;
        NbComposantes := NombreDeNoeuds(g);
        k := 0;                         (* Indice de l'arc courant *)
        WHILE NbComposantes <> 1 DO
```

```
        BEGIN
                k := k + 1;
                (* Inclure VectArcs[k] si l'arc joint deux composantes connexes différentes
                *  c'est-a-dire n'introduit pas de cycle dans ASM. *)
                WITH VectArcs[k] DO
                        IF Composantes[noeud1] <> Composantes[noeud2] THEN
                        BEGIN
                                AjouterArc(ASM, noeud1, noeud2, valeur);
                                (* Joint composante de noeud2 a la composante de noeud1. *)
                                compTemp := Composantes[noeud2];
                                FOR i := 1 TO NombreDeNoeuds(g) DO
                                        IF Composantes[i] = compTemp THEN
                                                Composantes[i] := Composantes[noeud1];
                                        NbComposantes := NbComposantes – 1;
                        END;
        END;
END; (* ProcKruskal *)

(*————————————— Procedures utilitaires pour l'interface ——————————————*)

PROCEDURE FaireUnePause;
(* But : Fait une pause jusqu'a ce que l'usager presse sur retour. *)
BEGIN
        WRITE('                                                ');
        WRITE(  '          appuyez sur retour');
        READLN;
END; (* FaireUnePause *)

PROCEDURE SauterLigne(n : INTEGER);
(* But : Saute n ligne a l'ecran. *)
VAR
        i : INTEGER;
BEGIN
        FOR i := 1 TO n DO
                WRITELN;
END; (* SauterLigne *)

PROCEDURE ViderEcran;
(* But : Fait defiler l'ecran jusqu'a ce qu'il n'y ait plus rien d'affiche. *)
BEGIN
        SauterLigne(25);
END; (* ViderEcran *)
```

```
(*———————————— Test de Kruskal ————————————*)

(* Teste l'algorithme de Kruskal sur un graphe complet genere a partir
* des coordonnees de points. *)
CONST
      MaxCoordonnees = MaxNoeud;
TYPE
      Coordonnee =     RECORD
                             Abscisse : REAL;
                             Ordonnee : REAL;
                       END;
      VectCoord = ARRAY[1..MaxCoordonnees] OF Coordonnee;
      TypeSpecCoord = RECORD
                             NbCoordonnees : INTEGER;
                             Coordonnees : VectCoord;
                       END;

FUNCTION Distance(x1, x2 : Coordonnee) : REAL;
(* But : Donne la distance euclidienne entre les coordonnées x1 et x2. *)
BEGIN
      Distance := SQRT(SQR(x1.abscisse – x2.abscisse) + SQR(x1.ordonnee – x2.ordonnee));
END; (* Distance *)

FUNCTION CoordPresente(VAR v : VectCoord; n1, n2 : INTEGER;
                             abs, ord : REAL) : BOOLEAN;
(* But:   Teste si une pour une coordonnee de v[n] (n1 <= n <= n2)
*         v[n].Abscisse = abs et v[n].Ordonnee = ord *)
VAR
      k : INTEGER;
BEGIN
      CoordPresente := FALSE;
      FOR k := n1 TO n2 DO
            IF (v[k].Abscisse = abs) AND (v[k].Ordonnee = ord) THEN
                  CoordPresente := TRUE;
END; (* CoordPresente *)

PROCEDURE SaisirCoordonnees(VAR Spec : TypeSpecCoord);
(* But :  – Saisit le nombre de coordonnees;
*         – saisit les coordonnees dans Spec. *)

PROCEDURE SaisirCoordonnee(i : INTEGER; VAR abscisse, ordonnee : REAL;
(* But : Saisit et donne la i-ieme coordonnee. *)
```

```
BEGIN
    SauterLigne(2);
    WRITELN('Coordonnee ', i : 3);
    WRITE(  '                abscisse : '); READLN(abscisse);
    WRITE(  '                ordonnee : '); READLN(ordonnee);
END;

PROCEDURE SaisirToutesCoordonnees(VAR Spec : TypeSpecCoord; n : INTEGER);
(* But : Saisir dans Spec des coordonnees 1..n.  *)
VAR
    i : INTEGER;
    abscisse, ordonnee : REAL;
BEGIN
    SauterLigne(3);
    WRITELN('Entrez les coordonnees des noeuds ');
    SauterLigne(2);
    i := 1;
    WHILE i <= n DO
    BEGIN
        SaisirCoordonnee(i, abscisse, ordonnee);
        IF NOT CoordPresente(Spec.Coordonnees, 1, i – 1, abscisse, ordonnee) THEN
        BEGIN
            Spec.Coordonnees[i].Abscisse := abscisse;
            Spec.Coordonnees[i].Ordonnee := ordonnee;
            i := i + 1
        END
        ELSE
            WRITELN('Coordonnee deja definie, recommencez s.v.p.');
    END;
END;

BEGIN
    ViderEcran;
    WRITELN('          Specification des coordonnees');
    WRITELN;
    WRITE('Entrez le nombre de noeuds (max. ', MaxNoeud : 2,') : ');
    READLN(Spec.NbCoordonnees);
    SaisirToutesCoordonnees(Spec, Spec.nbCoordonnees);
END; (* SaisirCoordonnees *)

PROCEDURE AfficherCoordonnees(Spec : TypeSpecCoord);
(* But : Affiche les coordonnees de Spec. *)
VAR
    i : INTEGER;
```

```
BEGIN
        ViderEcran;
        WITH Spec DO
        BEGIN
                WRITELN;
                WRITELN('                       Coordonnees des noeuds ');
                SauterLigne(2);
                FOR i := 1 TO NbCoordonnees DO
                BEGIN
                        IF ((i MOD 16) = 0) THEN
                                FaireUnePause;
                        WRITELN(' ', i : 2,'    ', '(', Coordonnees[i].abscisse : 12 : 2,
                                        ', ', Coordonnees[i].ordonnee : 12 : 2, ')');
                END;
        END; (* WITH *)
        FaireUnePause;
END; (* AfficherCoordonnees *)

PROCEDURE AfficherGraphe(g : Graphe);
(* But :  – Affiche les arcs de g et leurs valeurs;
*         – affiche le total des valeurs des arcs.  *)
VAR
        i, j : INTEGER;
        k : INTEGER;    (* Compteur pour pause  *)
BEGIN
        k := 0;
        FOR i := 1 TO NombreDeNoeuds(g) DO
                FOR j := i + 1 TO NombreDeNoeuds(g) DO
                        IF Adjacent(g, i, j) THEN
                        BEGIN
                                k := k + 1;
                                IF ((k MOD 16) = 0) THEN
                                        FaireUnePause;
                                WRITELN('arc (', i : 2, ',' , j : 2,') valeur : ', ValeurArc(g, i, j) : 12 : 2);
                        END;
                        SauterLigne(1);
                        WRITELN('           cout total : ', SommeValeurArc(g) : 12 : 2);
END; (* AfficherGraphe *)

PROCEDURE AppliquerKruskal (VAR Spec : TypeSpecCoord);
(* But : –  Definit un graphe g complet ou les valeurs d'arcs sont egales
*            aux distances entre les coordonnees correspondantes;
*         –  applique l'algorithme de Kruskal sur g -> ASM;
*         –  affiche les arcs de ASM et la somme de leurs valeurs. *)
```

```
VAR
        i, j : INTEGER;
        g, ASM : Graphe;
BEGIN
        InitGraphe(g, Spec.NbCoordonnees);
        FOR i := 1 TO Spec.NbCoordonnees DO
                FOR j := i + 1 TO Spec.NbCoordonnees DO
                        AjouterArc(g, i, j, Distance(Spec.Coordonnees[i], Spec.Coordonnees[j]));
        ProcKruskal(g, ASM);
        ViderEcran;
        WRITELN('Arcs de l''arbre recouvrant de cout minimum suivant l''algorithme de Kruskal');
        SauterLigne(2);
        AfficherGraphe(ASM);
        SauterLigne(2);
        FaireUnePause;
END; (* AppliquerKruskal *)

PROCEDURE TesterKruskal;
(* But : Menu permettant a l'usager de :
*         1. definir les coordonnees des noeuds d'un graphe,
*         2. afficher les coordonnees du graphe courant,
*         3. trouver l'ASM a l'aide de Kruskal et de l'afficher.  *)
VAR
        Choix : CHAR; (* Choix de l'usager *)
        Spec : TypeSpecCoord; (* Contient les coordonnees. *)
        CoordSontDefinies : BOOLEAN; (* A-t-on defini des coordonnees? *)

PROCEDURE SignifierCoordNonDefinies;
(* Affiche un message indiquant qu'on doit definir des coordonnees. *)
BEGIN
        ViderEcran;
        WRITELN('Vous devez d''abord definir les coordonnees. ');
        FaireUnePause;
END;

BEGIN
        CoordSontDefinies := FALSE;
        REPEAT
                ViderEcran;
                WRITELN('                    Test de Kruskal');
                SauterLigne(2);
                WRITELN('                1. Definir les coordonnees');
                WRITELN;
```

```
            WRITELN('              2. Afficher les coordonnees');
            WRITELN;
            WRITELN('              3. Resultat de Kruskal');
            WRITELN;
            WRITELN('              0. Terminer');
            SauterLigne(5);
            WRITE('Entrez votre choix : ');
            READLN(Choix);
            CASE Choix OF
                  '1' :  BEGIN
                              SaisirCoordonnees(Spec);
                              CoordSontDefinies := TRUE;
                         END;
                  '2' :  IF CoordSontDefinies THEN
                              AfficherCoordonnees(Spec)
                         ELSE
                              SignifierCoordNonDefinies;
                  '3':   IF CoordSontDefinies THEN
                              AppliquerKruskal(Spec)
                         ELSE
                              SignifierCoordNonDefinies;
                  '0' : ;
            END
      UNTIL Choix = '0';
END; (* TesterKruskal *)

(* ———————— Programme principal ———————— *)

BEGIN
      TesterKruskal;
      ViderEcran;
END.
```

## 9.3 LA LOCALISATION DE CONCENTRATEURS OU D'ENTREPÔTS

En considérant que chaque groupe de terminaux (ou de points de vente) doit être desservi par un concentrateur (ou un entrepôt), terminaux et concentrateur constituant une partition, le problème consiste à déterminer le nombre et les emplacements réels

de concentrateurs (ou d'entrepôts) devant desservir un certain nombre de terminaux (ou de points de vente), et ce avec une contrainte de coût.

On suppose que chaque terminal est affecté à un et à un seul concentrateur et qu'il n'y a pas de limite quant au nombre de terminaux pouvant être affectés à chaque concentrateur. Si aucun terminal n'est affecté à un emplacement potentiel de concentrateur, cet emplacement ne contiendra aucun concentrateur.

Désignons par $D_{ij}$ le coût d'affectation d'un terminal $i$ à un concentrateur $j$, avec $i \in T = \{1, 2, ..., n\}$ et $j \in C = \{1, 2, ..., m\}$. Le coût total d'affectation est donné par la relation suivante :

$$D = \sum_{i=1}^{n} \sum_{j=1}^{m} t_{ij} D_{ij} + \sum_{j=1}^{m} u_j C_j \qquad (9.1)$$

où $t_{ij}$ vaut 1 si le terminal $i$ est relié au concentrateur $j$, et 0 dans le cas contraire. $C_j$ désigne le coût du concentrateur $j$, $n$ le nombre de terminaux et $m$ le nombre d'emplacements potentiels de terminaux. Quant à $u_j$, il vaut 1 si un terminal au moins est affecté à l'emplacement $j$, et 0 dans le cas contraire. Il s'agit alors de déterminer les emplacements réels des concentrateurs, le nombre réel de ceux-ci et l'affectation des terminaux aux concentrateurs, de manière à minimiser le coût total d'affectation, tel qu'il est exprimé par la relation (9.1).

Ce problème, classé *décidable* mais *difficile*, peut être résolu par l'un ou l'autre des algorithmes ADD et DROP, considérés comme des méthodes heuristiques de résolution en ce sens qu'ils permettent de trouver, non pas des solutions optimales, mais plutôt des solutions proches de l'optimum. Les deux prochaines sections en illustrent le fonctionnement, avec les données de l'exemple 9.4.

*Exemple 9.4*

Considérons 8 terminaux et 5 positions possibles de concentrateur. Le coût d'affectation d'un terminal à un concentrateur est constant et égal à 3 unités. Les concentrateurs sont numérotés de 1 à 5. Il n'y a aucune contrainte sur le nombre de terminaux gérés par un concentrateur. Le tableau 8.1 présente, sous forme matricielle, les coûts $D_{ij}$ d'affectation du terminal *i* au concentrateur *j*, avec *i* = 1, 2, ..., 8 et *j* = 1, 2, ..., 5.

### 9.3.1 Le fonctionnement de l'algorithme ADD

Avec les données de l'exemple 8.4, l'algorithme comprendrait les étapes suivantes :

Étape 1 : On relie tous les terminaux au concentrateur 1, puis on calcule le coût de cette configuration considérée comme une partition de base (tableau 9.1).

**TABLEAU 9.1**
*COÛTS D'AFFECTATION DES TERMINAUX AUX CONCENTRATEURS*

| TERMINAL | CONCENTRATEUR **1** | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 39

Étape 2 : On introduit dans la partition de base le concentrateur 2, puis on relie chaque terminal soit à l'ordinateur central soit au nouveau concentrateur introduit, selon la liaison la plus économique. On obtient ainsi une nouvelle partition, représentée au tableau 9.2, dont le coût total d'affectation D est de 29.

**TABLEAU 9.2**

*SCHÉMA D'AFFECTATION (ÉTAPE 2)*

CONCENTRATEUR

| TERMINAL | **1** | **2** | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 29

$$\text{Coût} = \underbrace{(1 + 5 + 3 + 4 + 2 + 1 + 4 + 3)}_{\text{Coût d'affectation terminaux-concentrateurs}} + \underbrace{(2 \times 3)}_{\text{Coût des concentrateurs}}$$

Étape 3 : On répète l'étape 2 pour les paires de concentrateurs 1-3, 1-4 et 1-5; c'est ce que représentent les tableaux 9.3, 9.4 et 9.5.

**TABLEAU 9.3**

*SCHÉMA D'AFFECTATION 1-3*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | **3** | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 29

**TABLEAU 9.4**

*SCHÉMA D'AFFECTATION 1-4*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | 3 | **4** | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 28

**TABLEAU 9.5**

*SCHÉMA D'AFFECTATION 1-5*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | 3 | 4 | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 35

Étape 4 : On compare la partition la plus économique parmi celles correspondantes aux paires de concentrateurs 1-2, 1-3, 1-4 et 1-5 avec celle de l'étape 1. Celle dont le coût est le plus faible est alors considérée comme nouvelle partition de base. Ainsi, la partition du tableau 9.4, correspondant à la paire 1-4 et de coût 28, devient la partition de base.

Étape 5 : On introduit successivement les concentrateurs 2, 3 et 5 pour former les partitions correspondant aux triplets 1-4-2, 1-4-3 et 1-4-5, en ayant soin de calculer le coût respectif de chacune de ces partitions, ce que montrent les tableaux 9.6, 9.7 et 9.8.

**TABLEAU 9.6**

*SCHÉMA D'AFFECTATION 1-4-2*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | 3 | **4** | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 29

**TABLEAU 9.7**

*SCHÉMA D'AFFECTATION 1-4-3*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | **3** | **4** | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 25

**TABLEAU 9.8**

*SCHÉMA D'AFFECTATION 1-4-5*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | 3 | **4** | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 28

Étape 6 : On compare la partition la plus économique parmi celles qui correspondent aux triplets de concentrateurs 1-4-2, 1-4-3 et 1-4-5 avec celle de la dernière partition de base. Celle dont le coût est le plus faible est alors considérée comme la nouvelle partition de base. Ainsi, la partition du tableau 9.7, relative au triplet 1-4-3 et de coût 25, devient la partition de base.

Étape 7 : On introduit successivement les concentrateurs 2 et 5 pour former les partitions relatives aux quadruplets 1-4-3-2 et 1-4-3-5, en ayant soin de calculer le coût respectif de chacune de ces partitions, ce que montrent les tableaux 9.9 et 9.10.

**TABLEAU 9.9**

*SCHÉMA D'AFFECTATION 1-4-3-2*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | **3** | **4** | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 26

**TABLEAU 9.10**

*SCHÉMA D'AFFECTATION 1-4-3-5*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 25

Étape 8 : On compare la partition la plus économique parmi celles qui correspondent aux quadruplets de concentrateurs 1-4-3-2 et 1-4-3-5 avec la dernière partition de base. Celle dont le coût est le plus faible est alors considérée comme nouvelle partition de base. Ainsi, la partition du tableau 9.10, correspondant au triplet 1-4-3-5 et de coût identique à celui de la dernière partition de base, devient la partition de base.

Étape 9 : On introduit le dernier concentrateur, le numéro 2, à cette nouvelle partition de base, en ayant soin de calculer le coût de cette partition 1-4-3-5-2, ce que montre le tableau 9.11.

**TABLEAU 9.11**

*SCHÉMA D'AFFECTATION 1-4-3-5-2*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 26

Étape 10 : Étant donné que le coût de cette partition, soit D = 26, est supérieur au coût de la dernière partition de base, ce dernier concentrateur (le numéro 2) ne doit pas être introduit dans le réseau. Donc, la partition 1-4-3-5 du tableau 9.10 est la solution du problème.

Le tableau 9.10 peut être interprété de la manière suivante :

- le terminal 1 est affecté au concentrateur 1;
- le terminal 2 est affecté au concentrateur 4;
- le terminal 3 est affecté au concentrateur 1;
- le terminal 4 est affecté au concentrateur 3;
- le terminal 5 est affecté au concentrateur 3;
- le terminal 6 est affecté au concentrateur 4;
- le terminal 7 est affecté au concentrateur 4;
- le terminal 8 est affecté au concentrateur 5.

Ce qui correspond aux partitions suivantes :

– le concentrateur 1 gère les terminaux 1 et 3;
– le concentrateur 2 ne gère aucun terminal;
– le concentrateur 3 gère les terminaux 4 et 5;
– le concentrateur 4 gère les terminaux 2, 6 et 7;
– le concentrateur 5 gère le terminal 8.

Étant donné que le concentrateur 2 ne gère aucun terminal, il n'est donc pas nécessaire, ce qui ramène à quatre le nombre réel de concentrateurs.

L'exemple 9.5 présente les grandes lignes de l'algorithme ADD dont le programme de l'exemple 9.6 constitue une implantation. Ce programme est écrit en Turbo Pascal qui supporte des identificateurs de longueur supérieure à celle que permet le PASCAL standard.

*Exemple 9.5*

```
DEBUT
    S := {1, ..., m}
    REPETER
        trouver le candidat c ∈ S qui minimise CoutTotal(S – {c})
        SI CoutTotal(S-{c}) < CoutTotal(S) ALORS
            S := S – {c}
            EstAméliorée = VRAI
        SINON
            EstAméliorée = FAUX
        FINSI
    JUSQU'A NON(EstAméliorée)
FIN ADD
```

*Exemple 9.6*

```
PROGRAM ADD(INPUT, OUTPUT);
(*  Ce programme fait une mise en oeuvre de l'algorithme ADD.
*  Une interface par menu permet a un usager de definir les parametres
*  et d'appliquer l'algorithme.  *)
```

```
CONST
        Infini = 10.0E20;
        MaxTerm = 20; (* Nombre maximum de terminaux *)
        MaxConc = 10; (* Nombre maximum de concentrateurs *)
TYPE
        Terminal = 1..MaxTerm;
        Concentrateur = 1..MaxConc;
        MatCoutsLiaisons = ARRAY[Terminal, Concentrateur] OF REAL;
                (* Couts des liaisons entre terminaux et concentrateurs *)
        VectCoutsConc = ARRAY[Concentrateur] OF REAL;
                (* Couts d'etablissement des concentrateurs *)
        VectAssignation = ARRAY[Terminal] OF Concentrateur;
                (* Assignation des terminaux aux concentrateurs *)
        EnsConc = SET OF Concentrateur;

PROCEDURE AssignerTerminaux(NbTerm : Terminal; NbConc : Concentrateur;
                                        VAR CoutLiaison : MatCoutsLiaisons;
                                        VAR Conc : EnsConc;
                                        VAR Assignation : VectAssignation);
(* But :  Assigne les terminaux aux concentrateurs.
*  Assignation[i] sera le concentrateur c appartenant a Conc pour lequel
*  CoutLiaison[i, c] est le minimum.
* Suppose : CoutLiaison[i, j] < Infini.  *)
VAR
        t : Terminal;
        c : Concentrateur;
        coutMin : REAL;
BEGIN
(* Assigne chaque terminal au concentrateur de cout minimum. *)
        FOR t := 1 TO NbTerm DO
        BEGIN    (* Assigne t *)
                coutMin := Infini;
                FOR c := 1 TO NbConc DO
                        IF c IN Conc THEN
                                IF CoutLiaison[t, c] < coutMin THEN
                                BEGIN
                                        coutMin := CoutLiaison[t, c];
                                        Assignation[t] := c;
                                END;
        END;
END; (* AssignerTerminaux *)
```

```
FUNCTION CoutAss(NbTerm : Terminal; VAR CoutLiaison : MatCoutsLiaisons;
                        VAR Assignation : VectAssignation) : REAL;
(* But : Donne la somme des NbTerm premiers CoutLiaison[t, Assignation[t]].  *)
VAR
      t : Terminal;
      cout : REAL;
BEGIN
      cout := 0.0;
      FOR t := 1 TO NbTerm DO
            cout := cout + CoutLiaison[t, Assignation[t]];
      CoutAss := cout;
END; (* CoutAss *)

FUNCTION CoutEnsConc( NbConc : Concentrateur; VAR CoutsConc : VectCoutsConc;
                              Conc : EnsConc) : REAL;
(* But : Donne la somme des couts des concentrateurs appartenant a Conc.  *)
VAR
      c : Concentrateur;
      cout : REAL;
BEGIN
      cout := 0.0;
      FOR c := 1 TO NbConc DO
            IF c IN Conc THEN
                  cout := cout + CoutsConc[c];
      CoutEnsConc:= cout;
END; (* CoutEnsConc *)

FUNCTION CoutTotal(NbTerm : Terminal; NbConc: Concentrateur;
                        VAR CoutLiaison : MatCoutsLiaisons;
                        VAR CoutsConc : VectCoutsConc; Conc : EnsConc) : REAL;
(* But : Donne le cout total de Solution
*  c'est-a-dire la somme des couts des concentrateurs appartenant a Conc
*                       +
*  la somme des couts des liaisons de l'assignation des terminaux
*  aux concentrateurs de Conc.
*
*  Si Conc est vide, alors on ne peut assigner les terminaux et
*  alors CoutTotal = Infini.  *)
VAR
      Assignation : VectAssignation;
```

```
BEGIN
    IF Conc = [  ] THEN
        CoutTotal := Infini
    ELSE
        BEGIN
            AssignerTerminaux(NbTerm, NbConc, CoutLiaison, Conc, Assignation);
            CoutTotal := CoutEnsConc(NbConc, CoutsConc, Conc) +
                         CoutAss(NbTerm, CoutLiaison, Assignation);
        END;
END; (* CoutTotal *)

(*———————————— Algorithme ADD ————————————*)

PROCEDURE ProcADD(NbTerm : Terminal; NbConc: Concentrateur;
                  CoutLiaison : MatCoutsLiaisons; CoutsConc : VectCoutsConc;
                  VAR Solution : EnsConc);
(* But : Mise en oeuvre de l'algorithme ADD visant a trouver un sous-ensemble
*  Solution minimisant CoutTotal(Solution).  *)
VAR AmeliorationReussie : BOOLEAN;
    coutOpt, nouveauCout, coutTemp : REAL;
    candidatOpt, c : Concentrateur;
    t : Terminal;
    Candidats : EnsConc;
    Assignation : VectAssignation;
BEGIN
    Solution := [1];   (* On suppose que [1] est une solution faisable. *)
    coutOpt := CoutTotal(NbTerm, NbConc, CoutLiaison, CoutsConc, Solution);
    Candidats := [1..NbConc] – Solution;   (* Candidats a l'ajout *)
    REPEAT
        (* Recherche du candidatOpt dans Candidats dont l'ajout conduit
        *  a la plus grande diminution de cout. *)
        nouveauCout := Infini;
        FOR c := 1 TO NbConc DO
            IF (c IN Candidats) THEN
            BEGIN
                coutTemp := CoutTotal(NbTerm, NbConc, CoutLiaison, CoutsConc,
                            Solution + [c]);
                IF coutTemp < nouveauCout THEN
                BEGIN
                    candidatOpt := c;
                    nouveauCout := coutTemp;
                END;
            END;
```

```
            IF nouveauCout < coutOpt THEN   (* A-t-on trouve un candidat qui ameliore? *)
            BEGIN
                  Solution := Solution + [candidatOpt];
                  Candidats := Candidats – [candidatOpt];
                  coutOpt := nouveauCoet;
                  AmeliorationReussie := TRUE;
            END
            ELSE
                  AmeliorationReussie := FALSE;
      UNTIL NOT AmeliorationReussie;
      (* Ne conserver dans Solution que les concentrateurs utilises dans
      *  l'assignation des terminaux a Solution.  *)
      AssignerTerminaux(NbTerm, NbConc, CoutLiaison, Solution, Assignation);
      Solution := [ ];
      FOR t := 1 TO NbTerm DO
            Solution := Solution + [Assignation[t]];
END; (* ProcADD *)

(*———————— Procedures utilitaires pour l'interface ————————*)

PROCEDURE FaireUnePause;
(* But : Fait une pause jusqu'a ce que l'usager presse sur retour. *)
BEGIN
      WRITE('                                          ');
      WRITE(  '          appuyez sur retour');
      READLN;
END; (* FaireUnePause *)

PROCEDURE SauterLigne(n : INTEGER);   (* But : Saute n ligne a l'ecran.  *)
VAR
      i : INTEGER;
BEGIN
      FOR i := 1 TO n DO
            WRITELN;
END; (* SauterLigne *)

PROCEDURE ViderEcran;
(* But : Fait defiler l'ecran jusqu'a ce qu'il n'y ait plus rien d'affiche. *)
BEGIN
      SauterLigne(25);
END; (* ViderEcran *)
```

```
(*———————————— Test de ADD ————————————*)

TYPE
      Parametres =    RECORD     (* Parametres du probleme de localisation *)
                              NbConc : INTEGER;
                              NbTerm : INTEGER;
                              CoutLiaison : MatCoutsLiaisons;
                              CoutsConc : VectCoutsConc;
                      END;

PROCEDURE SaisirParametres(VAR Param : Parametres);
(* But :   Effectue la saisie au terminal :
*          – du nombre de concentrateurs,
*          – du nombre de terminaux,
*          – des couts d'etablissements des concentrateurs,
*          – des couts des liaisons entre terminaux et concentrateurs. *)
VAR
      i, j : INTEGER;
PROCEDURE SaisirCoutLiaison(i, j : INTEGER; VAR cout : REAL);
(* But : Saisie du cout de la liaison (i, j) *)
BEGIN
      WRITE('term. ', i : 3, ' conc. ', j : 3);
      WRITE('   cout : ');
      READLN(cout);
END; (* SaisirCoutLiaison *)

PROCEDURE SaisirCoutConcentrateur(i : INTEGER; VAR cout : REAL);
(* But : Saisie du cout du concentrateur i *)
BEGIN
      WRITE('concentrateur ', i : 3);
      WRITE('   cout : ');
      READLN(cout);
END; (* SaisirCoutConcentrateur *)

BEGIN
      ViderEcran;
      WITH Param DO
      BEGIN
            WRITELN('               Definition des parametres ');
            SauterLigne(6);
            WRITE('Entrez le nombre de concentrateurs (max. ', MaxConc : 2,') : ');
            READLN(NbConc);
            SauterLigne(3);
```

```
            WRITE('Entrez le nombre de terminaux (max. ', MaxTerm : 2,') : ');
            READLN(NbTerm);
            SauterLigne(3);
            WRITELN('Entrez les couts d''etablissement des concentrateurs ');
            WRITELN;
            FOR i := 1 TO Param.NbConc DO
                  SaisirCoutConcentrateur(i, CoutsConc[i]);
            SauterLigne(6);
            WRITELN('Entrez les couts des liaisons entre terminaux et concentrateurs ');
            FOR i := 1 TO Param.NbTerm DO
            BEGIN
                  WRITELN;
                  FOR j := 1 TO Param.NbConc DO
                        SaisirCoutLiaison(i, j, CoutLiaison[i, j]);
            END;
      END; (* WITH *)
END; (* SaisirParametres *)

PROCEDURE AfficherParametres(VAR Param : Parametres);
(* But :   Affiche :
*           – le nombre de concentrateurs,
*           – le nombre de terminaux,
*           – les couts d'etablissements des concentrateurs,
*           – les couts des liaisons entre terminaux et concentrateurs.   *)
VAR
      i, j : INTEGER;
BEGIN
      ViderEcran;
      WITH Param DO
      BEGIN
            WRITELN('                Parametres  ');
            SauterLigne(3);
            WRITE('nombre de concentrateurs : ', Param.NbConc : 2);
            SauterLigne(2);
            WRITE('nombre de terminaux : ', Param.NbTerm : 2);
            SauterLigne(3);
            FaireUnePause;
            WRITELN('Couts d''etablissement des concentrateurs ');
            SauterLigne(2);
            FOR i := 1 TO Param.NbConc DO
            BEGIN
                  IF i MOD 16 = 0 THEN
                        FaireUnePause;
                  WRITELN(' concentrateur ', i : 2,' : ', Param.CoutsConc[i] : 12 : 2);
            END;
```

```
            FaireUnePause;
            SauterLigne(2);
            WRITELN('Couts des liaisons entre terminaux et concentrateurs ');
            SauterLigne(2);
            FOR i := 1 TO Param.NbTerm DO
            BEGIN
                  FOR j := 1 TO Param.NbConc DO
                        WRITELN('term. ', i : 2,' conc. ', j : 2 ,' : ', ParamCoutLiaison[i, j] : 12 : 2);
                  FaireUnePause;
            END;
      END; (* WITH *)
END; (* AfficherParametres *)

PROCEDURE AfficherResultatLoc(VAR Param : Parametres; Solution : EnsConc);
(* But :    Affiche :
*           – les numeros des concentrateurs inclus dans Solution et leurs couts,
*           – les assignations des terminaux aux concentrateurs et les couts des liaisons,
*           – le total des couts des concentrateurs et des liaisons.   *)
VAR
      Assignation : VectAssignation;

PROCEDURE AfficherConcRetenus VAR Param : Parametres; Solution : EnsConc);
(* But : Affiche les concentrateurs de Solution et leurs couts. *)
VAR
      i : INTEGER;
BEGIN
      WITH Param DO
      BEGIN
            WRITELN('les concentrateurs retenus ');
            WRITELN;
            FOR i := 1 TO NbConc DO
                  IF i IN Solution THEN
                  BEGIN
                        IF (i MOD 10) = 0 THEN
                        FaireUnePause;
                        WRITELN('conc. ', i : 1, '                    cout : ',
                        CoutsConc[i] : 8 : 2);
                  END;
      END;
END;

PROCEDURE    AfficherLiaisonsRetenues(VAR Param : Parametres;
                  Assignation : VectAssignation);
(* But : Affiche les liaisons (i, Assignation[i]) et leurs couts. *)
VAR
      i : INTEGER;
```

```
BEGIN
        WRITELN('les liaisons retenues ');
        WRITELN;
        WITH Param DO
        BEGIN
                FOR i := 1 TO NbTerm DO
                BEGIN
                        IF (i MOD 10) = 0 THEN
                                FaireUnePause;
                        WRITELN('term. ', i : 2,' conc. ', Assignation[i] : 2,'     cout : ',
                                        CoutLiaison[i, Assignation[i]] : 8 : 2);
                END;
        END;
END;
BEGIN
        WITH Param DO
        BEGIN
                SauterLigne(2);
                AfficherConcRetenus(Param, Solution);
                AssignerTerminaux(NbTerm, NbConc, CoutLiaison, Solution, Assignation);
                WRITELN;
                AfficherLiaisonsRetenues(Param, Assignation);
                WRITELN;
                WRITELN('                        cout total : ',
                                CoutTotal(NbTerm, NbConc, CoutLiaison, CoutsConc, Solution) : 8 : 2);
        END;
        FaireUnePause;
END; (* AfficherResultatLoc *)

PROCEDURE AppliquerADD(VAR Param : Parametres);
(* But : Applique l'algorithme ADD sur Param ainsi que les resultats *)
VAR
        Solution : EnsConc;
BEGIN
        ViderEcran;
        WITH Param DO
                ProcADD(NbTerm, NbConc, CoutLiaison, CoutsConc, Solution);
        WRITELN('Resultat de l''application de l''algorithme ADD');
        AfficherResultatLoc(Param, Solution);
END; (* AppliquerADD *)

PROCEDURE TesterADD;
(* But : Menu permettant a l'usager de :
*           1.  definir les parametres,
*           2.  afficher les parametres courants,
*           3.  appliquer l'algorithme ADD sur les parametres courants et d'afficher les resultats. *)
```

```
VAR
      Choix : CHAR; (* Choix de l'usager *)
      Param : Parametres; (* Contient les parametres. *)
      ParamSontDefinis : BOOLEAN; (* A-t-on defini des parametres? *)

PROCEDURE SignifierParamNonDefinis;
(* But : Affiche un message indiquant qu'on doit definir des parametres. *)
BEGIN
      ViderEcran;
      WRITELN('Vous devez d''abord definir les parametres ');
      FaireUnePause;
END;

BEGIN
      ParamSontDefinis := FALSE;
      REPEAT
            ViderEcran;
            WRITELN('                    Test de ADD');
            SauterLigne(2);
            WRITELN('              1. Definir les parametres');
            WRITELN;
            WRITELN('              2. Afficher les parametres');
            WRITELN;
            WRITELN('              3. Resultat de ADD');
            WRITELN;
            WRITELN('              0. Terminer');
            SauterLigne(5);
            WRITE('     Entrez votre choix : ');
            READLN(Choix);
            CASE Choix OF
                  '1' :  BEGIN
                              SaisirParametres(Param);
                              ParamSontDefinis := TRUE;
                         END;
                  '2' :  IF ParamSontDefinis THEN
                              AfficherParametres(Param)
                         ELSE
                              SignifierParamNonDefinis;
                  '3' :  IF ParamSontDefinis THEN
                              AppliquerADD(Param)
                         ELSE
                              SignifierParamNonDefinis;
                  '0' : ;
            END;
      UNTIL Choix = '0';
END; (* TesterADD *)
```

```
(* ———————— Programme principal ———————— *)
BEGIN
      TesterADD;
      ViderEcran;
END.
```

### 9.3.2 Le fonctionnement de l'algorithme DROP

L'application de l'algorithme DROP à l'exemple 9.4 se traduit par les étapes suivantes:

Étape 1: On constitue une partition de base par affectation de chaque terminal *i* à un concentrateur *j* tel que $D_{ij}$ soit la plus faible valeur de la ligne *i* (raccordement par la liaison la plus économique). Puis, on calcule le coût de cette partition. Le tableau 9.12 caractérise cette partition de base qui est de type 1-2-3-4-5.

**TABLEAU 9.12**
*SCHÉMA D'AFFECTATION 1-2-3-4-5*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 26

Étape 2: On enlève successivement de la partition de base un concentrateur pour former les partitions 1-3-4-5, 1-2-4-5, 1-2-3-5 et 1-2-3-4 correspondant aux tableaux 9.13, 9.14, 9.15 et 9.16. Puis, on en calcule les coûts respectifs.

**TABLEAU 9.13**

*SCHÉMA D'AFFECTATION 1-3-4-5*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 25

**TABLEAU 9.14**

*SCHÉMA D'AFFECTATION 1-2-4-5*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | 3 | **4** | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 29

**TABLEAU 9.15**

*SCHÉMA D'AFFECTATION 1-2-3-5*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | **3** | **4** | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 26

**TABLEAU 9.16**

*SCHÉMA D'AFFECTATION 1-2-3-4*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | **3** | **4** | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 26

Étape 3: On compare la partition la plus économique parmi celle que l'on a obtenues à l'étape 2 avec la partition de base. Celles dont le coût est le plus faible sont alors considérées comme nouvelles partitions de base. Ainsi, les partitions des tableaux 9.13 et 9.15 deviennent tour à tour des partitions de base.

Étape 4: À partir de la partition de base 1-3-4-5 du tableau 9.13, on enlève successivement un concentrateur pour obtenir les partitions 1-4-5, 1-3-5 et 1-3-4 respectivement représentées par les tableaux 9.17, 9.18 et 9.19, en ayant soin de calculer le coût de ces partitions.

Étape 5: On compare la partition la plus économique parmi celles que l'on a obtenues à l'étape précédente avec la partition de base 1-3-4-5. Étant donné que la partition 1-3-4 du tableau 9.19 a un coût égal à celui de la partition de base, cette dernière demeure inchangée, ce qui indique la fin du processus d'itération.

Étape 6: À partir de la partition de base 1-2-3-5 du tableau 9.15, on enlève successivement un concentrateur pour obtenir les partitions 1-3-5, 1-2-3 et 1-2-5, respectivement représentées par les tableaux 9.20, 9.21 et 9.22, en ayant soin de calculer le coût de ces partitions.

**TABLEAU 9.17**

*SCHÉMA D'AFFECTATION 1-4-5*

CONCENTRATEUR

| TERMINAL | **1** | 2 | 3 | **4** | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 29

**TABLEAU 9.18**

*SCHÉMA D'AFFECTATION 1-3-5*

CONCENTRATEUR

| TERMINAL | **1** | 2 | **3** | 4 | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 27

**TABLEAU 9.19**

*SCHÉMA D'AFFECTATION 1-3-4*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | **3** | **4** | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 25

**TABLEAU 9.20**

*SCHÉMA D'AFFECTATION 1-3-5*

| TERMINAL \ CONCENTRATEUR | **1** | 2 | **3** | 4 | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 27

**TABLEAU 9.21**

*SCHÉMA D'AFFECTATION 1-2-3*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | **3** | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 26

**TABLEAU 9.22**

*SCHÉMA D'AFFECTATION 1-2-5*

| TERMINAL \ CONCENTRATEUR | **1** | **2** | 3 | 4 | **5** |
|---|---|---|---|---|---|
| 1 | 1 | 8 | 3 | 7 | 4 |
| 2 | 6 | 2 | 6 | 3 | 5 |
| 3 | 2 | 1 | 5 | 5 | 3 |
| 4 | 5 | 5 | 2 | 6 | 7 |
| 5 | 8 | 4 | 1 | 4 | 6 |
| 6 | 7 | 3 | 4 | 2 | 8 |
| 7 | 3 | 6 | 7 | 1 | 2 |
| 8 | 4 | 7 | 8 | 8 | 1 |

Coût = 28

Étape 7: On compare la partition la plus économique parmi celles que l'on a obtenues à l'étape précédente avec la partition de base 1-2-3-5. Étant donné qu'aucune des trois dernières partitions n'a un coût inférieur à celui de la partition de base du tableau 9.15, c'est la fin du processus d'itération.

Le problème admet donc deux solutions représentées par les tableaux 9.13 et 9.15 donnant les emplacements et le nombre réels des concentrateurs, ainsi que l'affectation des terminaux aux concentrateurs.

L'exemple 9.7 présente les grandes lignes de l'algorithme DROP dont le programme de l'exemple 9.8 constitue une implantation. Ce programme est écrit en Turbo Pascal qui accepte des identificateurs de longueur supérieure à celle que permet le PASCAL standard.

*Exemple 9.7*

```
DEBUT
    S := { } (* Une solution faisable *)
    T := {1, 2, ..., m} – S
    REPETER
        trouver le candidat c ∈ T qui minimise CoutTotal(S + {c})
        SI CoutTotal(S + {c}) < CoutTotal(S) ALORS
            S := S + {c}
            T := T – {c}
            EstAméliorée = VRAI
        SINON
            EstAméliorée = FAUX
        FINSI
    JUSQU'A NON(EstAméliorée)
    Retirer de S les éléments qui ne sont affectés à aucun terminal
FIN DROP.
```

*Exemple 9.8*

```
PROGRAM DROP(INPUT, OUTPUT);
(*  Ce programme fait une mise en oeuvre de l'algorithme DROP.
*  Une interface par menu permet a un usager de definir les parametres
*  et d'appliquer l'algorithme.  *)
```

```
CONST
    Infini = 10.0E20;
    MaxTerm = 20; (* Nombre maximum de terminaux *)
    MaxConc = 10; (* Nombre maximum de concentrateurs *)
TYPE
    Terminal = 1..MaxTerm;
    Concentrateur = 1..MaxConc;
    MatCoutsLiaisons = ARRAY[Terminal, Concentrateur] OF REAL;
            (* Couts des liaisons entre terminaux et concentrateurs *)
    VectCoutsConc = ARRAY[Concentrateur] OF REAL;
            (* Couts d'etablissement des concentrateurs *)
    VectAssignation = ARRAY[Terminal] OF Concentrateur;
            (* Assignation des terminaux aux concentrateurs *)
    EnsConc = SET OF Concentrateur;

PROCEDURE AssignerTerminaux(NbTerm : Terminal; NbConc : Concentrateur;
                            VAR CoutLiaison : MatCoutsLiaisons;
                            VAR Conc : EnsConc;
                            VAR Assignation : VectAssignation);
(* But : Assigne les terminaux aux concentrateurs.
*  Assignation[i] sera le concentrateur c appartenant a Conc pour lequel
*  CoutLiaison[i, c] est le minimum.
*  Suppose : CoutLiaison[i, j] < Infini.  *)
VAR
    t : Terminal;
    c : Concentrateur;
    coutMin : REAL;
BEGIN
    (* Assigne chaque terminal au concentrateur de cout minimum. *)
    FOR t := 1 TO NbTerm DO
    BEGIN    (* Assigne t *)
        coutMin := Infini;
        FOR c := 1 TO NbConc DO
            IF c IN Conc THEN
                IF CoutLiaison[t, c] < coutMin THEN
                BEGIN
                    coutMin := CoutLiaison[t, c];
                    Assignation[t] := c;
                END;
    END;
```

```
END; (* AssignerTerminaux *)

FUNCTION CoutAss(NbTerm : Terminal; VAR CoutLiaison : MatCoutsLiaisons;
                        VAR Assignation : VectAssignation) : REAL;
(* But : Donne la somme des NbTerm premiers CoutLiaison[t, Assignation[t]]. *)
VAR
      t : Terminal;
      cout : REAL;
BEGIN
      cout := 0.0;
      FOR t := 1 TO NbTerm DO
            cout := cout + CoutLiaison[t, Assignation[t]];
      CoutAss := cout;
END; (* CoutAss *)

FUNCTION CoutEnsConc(NbConc : Concentrateur; VAR CoutsConc : VectCoutsConc;
                              Conc : EnsConc) : REAL;
(* But : Donne la somme des couts des concentrateurs appartenant a Conc. *)
VAR
      c : Concentrateur;
      cout : REAL;
BEGIN
      cout := 0.0;
      FOR c := 1 TO NbConc DO
            IF c IN Conc THEN
                  cout := cout + CoutsConc[c];
      CoutEnsConc := cout;
END; (* CoutEnsConc *)

FUNCTION CoutTotal(NbTerm : Terminal; NbConc : Concentrateur;
                        VAR CoutLiaison : MatCoutsLiaisons;
                        VAR CoutsConc : VectCoutsConc; Conc : EnsConc) : REAL;
(* But : Donne le cout total de Solution
*  c'est-a-dire la somme des couts des concentrateurs appartenant a Conc
*                              +
*  la somme des couts des liaisons de l'assignation des terminaux
*  aux concentrateurs de Conc.
*
*  Si Conc est vide alors on ne peut assigner les terminaux et
*  alors CoutTotal = Infini.  *)
VAR
      Assignation : VectAssignation;
```

```
BEGIN
    IF Conc = [ ] THEN
        CoutTotal := Infini
    ELSE
        BEGIN
            AssignerTerminaux(NbTerm, NbConc, CoutLiaison, Conc, Assignation);
            CoutTotal := CoutEnsConc(NbConc, CoutsConc, Conc) +
                         CoutAss(NbTerm, CoutLiaison, Assignation);
        END;
END; (* CoutTotal *)

(*———————————— Algorithme DROP ————————————*)

PROCEDURE ProcDROP(NbTerm : Terminal; NbConc : Concentrateur;
                   CoutLiaison : MatCoutsLiaisons; CoutsConc : VectCoutsConc;
                   VAR Solution : EnsConc);
(* But : Mise en oeuvre de l'algorithme DROP visant a trouver un sous-ensemble
*  Solution minimisant CoutTotal(Solution). (Voir annexe.) *)
VAR AmeliorationReussie : BOOLEAN;
    coutOpt, nouveauCout, coutTemp : REAL;
    candidatOpt, c : Concentrateur;
BEGIN
    (* La solution initiale contient tous les concentrateurs. *)
    Solution := [1..NbConc];
    coutOpt := CoutTotal(NbTerm, NbConc, CoutLiaison, CoutsConc, Solution);
    REPEAT
        (* Recherche du candidatOpt dans Solution dont le retrait conduit
        *  a la plus grande diminution de cout. *)
        nouveauCout := Infini;
        FOR c := 1 TO NbConc DO
            IF (c IN Solution) THEN
            BEGIN
                coutTemp := CoutTotal(NbTerm, NbConc, CoutLiaison, CoutsConc,
                                      Solution – [c]);
                IF coutTemp < nouveauCout THEN
                BEGIN
                    candidatOpt := c;
                    nouveauCout := coutTemp;
                END;
            END;
            IF nouveauCout < coutOpt THEN (* A-t-on trouve un candidat qui ameliore? *)
```

```
                BEGIN
                    Solution := Solution – [candidatOpt];
                    coutOpt := nouveauCout;
                    AmeliorationReussie := TRUE;
                END
                ELSE
                    AmeliorationReussie := FALSE;
    UNTIL NOT(AmeliorationReussie);
END; (* ProcDROP *)

(*———————— Procedures utilitaires pour l'interface ————————*)

PROCEDURE FaireUnePause;
(* But : Fait une pause jusqu'a ce que l'usager presse sur retour. *)
BEGIN
    WRITE('                                          ');
    WRITE(  '        appuyez sur retour');
    READLN;
END; (* FaireUnePause *)

PROCEDURE SauterLigne(n : INTEGER);    (* But : Saute n ligne a l'ecran.  *)
VAR
    i : INTEGER;
BEGIN
    FOR i := 1 TO n DO
        WRITELN;
END; (* SauterLigne *)

PROCEDURE ViderEcran;
(* But : Fait defiler l'ecran jusqu'a ce qu'il n'y ait plus rien d'affiche. *)
BEGIN
    SauterLigne(25);
END; (* ViderEcran *)

(*————————————————— Test de DROP —————————————————*)

TYPE
    Parametres =    RECORD      (* Parametres du probleme de localisation *)
                        NbConc : INTEGER;
                        NbTerm : INTEGER;
                        CoutLiaison : MatCoutsLiaisons;
                        CoutsConc : VectCoutsConc;
                    END;
```

```
PROCEDURE SaisirParametres(VAR Param : Parametres);
(* But : Effectue la saisie au terminal :
*         – du nombre de concentrateurs,
*         – du nombre de terminaux,
*         – des couts d'etablissements des concentrateurs,
*         – des couts des liaisons entre terminaux et concentrateur.  *)
VAR
      i, j : INTEGER;

PROCEDURE SaisirCoutLiaison(i, j : INTEGER; VAR cout : REAL);
(* But : Saisie du cout de la liaison (i, j). *)
BEGIN
      WRITE('term. ', i : 3, ' conc. ', j : 3);
      WRITE('     cout : ');
      READLN(cout);
END; (* SaisirCoutLiaison *)

PROCEDURE SaisirCoutConcentrateur(i : INTEGER; VAR cout : REAL);
(* But : Saisie du cout du concentrateur i. *)
BEGIN
      WRITE('concentrateur ', i : 3);
      WRITE('   cout : ');
      READLN(cout);
END; (* SaisirCoutConcentrateur *)

BEGIN
      ViderEcran;
      WITH Param DO
      BEGIN
            WRITELN('                Definition des parametres ');
            SauterLigne(6);
            WRITE('Entrez le nombre de concentrateurs (max. ', MaxConc : 2,') : ');
            READLN(NbConc);
            SauterLigne(3);
            WRITE('Entrez le nombre de terminaux (max. ', MaxTerm : 2,')     : ');
            READLN(NbTerm);
            SauterLigne(3);
            WRITELN('Entrez les couts d''etablissement des concentrateurs');
            WRITELN;
            FOR i := 1 TO Param.NbConc DO
                  SaisirCoutConcentrateur(i, CoutsConc[i]);
            SauterLigne(6);
            WRITELN('Entrez les couts des liaisons entre terminaux et concentrateurs');
            FOR i := 1 TO Param.NbTerm DO
```

```
            BEGIN
                WRITELN;
                FOR j := 1 TO Param.NbConc DO
                    SaisirCoutLiaison(i, j, CoutLiaison[i, j]);
            END;
        END; (* WITH *)
END; (* SaisirParametres *)

PROCEDURE AfficherParametres(VAR Param : Parametres);
(* But : Affiche :
*         – le nombre de concentrateurs,
*         – le nombre de terminaux,
*         – les couts d'etablissements des concentrateurs,
*         – les couts des liaisons entre terminaux et concentrateurs. *)
VAR
    i, j : INTEGER;
BEGIN
    ViderEcran;
    WITH Param DO
    BEGIN
        WRITELN('                Parametres  ');
        SauterLigne(3);
        WRITE('nombre de concentrateurs : ', Param.NbConc : 2);
        SauterLigne(2);
        WRITE('nombre de terminaux      : ', Param.NbTerm : 2);
        SauterLigne(3);
        FaireUnePause;
        WRITELN('Couts d''etablissement des concentrateurs');
        SauterLigne(2);
        FOR i := 1 TO Param.NbConc DO
        BEGIN
            IF i MOD 16 = 0 THEN
                FaireUnePause;
            WRITELN('concentrateur ', i : 2,' : ', Param.CoutsConc[i] : 12 : 2);
        END;
        FaireUnePause;
        SauterLigne(2);
        WRITELN('Couts des liaisons entre terminaux et concentrateurs');
        SauterLigne(2);
        FOR i := 1 TO Param.NbTerm DO
        BEGIN
            FOR j := 1 TO Param.NbConc DO
                WRITELN('term. ', i : 2, 'conc. ', j : 2, ' : ',
                        Param.CoutLiaison[i, j] : 12 : 2);
            FaireUnePause;
        END;
    END; (* WITH *)
END; (* AfficherParametres *)
```

```
PROCEDURE AfficherResultatLoc(VAR Param : Parametres; Solution : EnsConc);
(* But :  affiche :
*          –  les numéros des concentrateurs inclus dans Solution et leurs couts,
*          –  les assignations des terminaux aux concentrateurs et les couts des liaisons,
*          –  le total des couts des concentrateurs et des liaisons.  *)
VAR
      Assignation : VectAssignation;

PROCEDURE AfficherConcRetenus(VAR Param : Parametres; Solution : EnsConc);
(* But : Affiche les concentrateurs de Solution et leurs couts. *)
VAR
      i : INTEGER;
BEGIN
      WITH Param DO
      BEGIN
            WRITELN('les concentrateurs retenus');
            WRITELN;
            FOR i := 1 TO NbConc DO
                  IF i IN Solution THEN
                  BEGIN
                        IF (i MOD 10) = 0 THEN
                              FaireUnePause;
                        WRITELN('conc. ', i : 1,'                    cout : ',
                        CoutsConc[i] : 8 : 2);
                  END;
      END;
END;

PROCEDURE AfficherLiaisonsRetenues(VAR Param : Parametres;
                                   Assignation : VectAssignation);
(* But : Affiche les liaisons (i, Assignation[i]) et leurs couts. *)
VAR
      i : INTEGER;
BEGIN
      WRITELN('les liaisons retenues ');
      WRITELN;
      WITH Param DO
      BEGIN
            FOR i := 1 TO NbTerm DO
            BEGIN
                  IF (i MOD 10) = 0 THEN
                        FaireUnePause;
                  WRITELN('term. ', i : 2,' conc. ', Assignation[i] : 2,'      cout : ',
                              CoutLiaison[i, Assignation[i]] : 8 : 2);
            END;
      END;
END;
```

```
BEGIN
        WITH Param DO
        BEGIN
                SauterLigne(2);
                AfficherConcRetenus(Param, Solution);
                AssignerTerminaux(NbTerm, NbConc, CoutLiaison, Solution, Assignation);
                WRITELN;
                AfficherLiaisonsRetenues(Param, Assignation);
                WRITELN;
                WRITELN('                        cout total : ',
                                CoutTotal(NbTerm, NbConc, CoutLiaison, CoutsConc, Solution) : 8 : 2);
        END;
        FaireUnePause;
END; (* AfficherResultatLoc *)

PROCEDURE AppliquerDROP(VAR Param : Parametres);
(* But : –  Applique l'algorithme DROP sur Param;
*         –  affiche les resultats. *)
VAR
        Solution : EnsConc;
BEGIN
        ViderEcran;
        WITH Param DO
                ProcDROP(NbTerm, NbConc, CoutLiaison, CoutsConc, Solution);
        WRITELN('Resultat de l''application de l''algorithme DROP');
        SauterLigne(2);
        AfficherResultatLoc(Param, Solution);
END; (* AppliquerDROP *)

PROCEDURE TesterDROP;
(* But :  Menu permettant a l'usager de :
*           1.   definir les parametres,
*           2.   afficher les parametres courants,
*           3.   appliquer l'algorithme DROP sur les parametres courants,
*                d'afficher les resultats.  *)
VAR
        Choix : CHAR;  (* Choix de l'usager *)
        Param : Parametres; (* Contient les parametres. *)
        ParamSontDefinis : BOOLEAN; (* A-t-on defini des parametres? *)
PROCEDURE SignifierParamNonDefinis;
(* But : Affiche un message indiquant qu'on doit definir des parametres. *)
BEGIN
        ViderEcran;
        WRITELN('Vous devez d''abord definir les parametres ');
        FaireUnePause;
END;
```

```
BEGIN
      ParamSontDefinis := FALSE;
      REPEAT
            ViderEcran;
            WRITELN('                  Test de DROP');
            SauterLigne(2);
            WRITELN('            1. Definir les parametres ');
            WRITELN;
            WRITELN('            2. Afficher les parametres ');
            WRITELN;
            WRITELN('            3. Resultat de DROP');
            WRITELN;
            WRITELN('            0. Terminer');
            SauterLigne(5);
            WRITE('Entrez votre choix : ');
            READLN(Choix);
            CASE Choix OF
                  '1' :  BEGIN
                              SaisirParametres(Param);
                              ParamSontDefinis := TRUE;
                         END;
                  '2' :  IF ParamSontDefinis THEN
                              AfficherParametres(Param)
                         ELSE
                              SignifierParamNonDefinis;
                  '3' :  IF ParamSontDefinis THEN
                              AppliquerDROP(Param)
                         ELSE
                              SignifierParamNonDefinis;
                  '0' : ;
            END;
      UNTIL Choix = '0';
END; (* TesterDROP *)

(*————————————— Programme principal————————————— *)

BEGIN
      TesterDROP;
      ViderEcran;
END.
```