Exercices : Cheminements sur un graphe

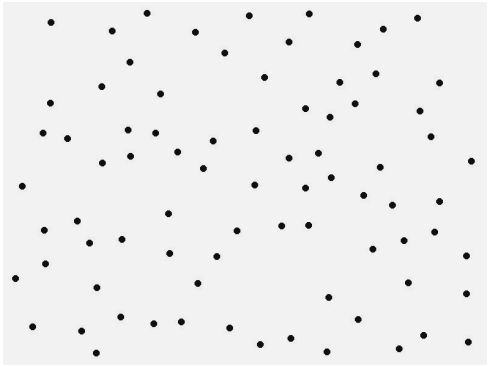
1) Construction d'un graphe orienté au hasard, style réseau ferroviaire

On travaille en coordonnées écran (pixels avec *x* entre 0 et 799 et *y* entre 0 et 599 par exemple).

a) On se donne *N* le nombre de sommets du graphe (par exemple *N* = 80). On va placer ces *N* points sur l'écran (blanc au départ) en faisant en sorte qu'il existe une distance minimale *R* (par exemple *R*=40) entre ces points pris deux à deux. Pour cela on prend le point numéro *i* au hasard en faisant

x[i]=20+rand()%760; y[i]=20+rand()%560;

et l'on remplit en rouge un disque ayant ce point pour centre, et de rayon *R*. Pour chaque valeur de *i*, avec *i* de 0 à *N* – 1, on prend un point au hasard et l'on répète jusqu'à ce que l'on ne tombe pas sur un point rouge. Cela permet d'avoir des points séparés par une distance supérieure à *R*. Une fois cela fait, on dessine les points *x*[*i*],*y*[*i*] sous forme de petits disques noirs, et l'on efface de l'écran tout ce qui était en rouge. Attention, il convient d'éviter de prendre un rayon *R* trop grand, sinon on ne peut pas placer les *N* sommets. Par exemple pour *N* = 300, *R* est de l'ordre de 30, pas plus.



b) Tracé d'arcs de jonction au hasard

Chaque sommet va avoir trois voisins (*nbv*[*i*]=3). On fait cela en trois étapes :

* Pour chaque sommet *i*, on cherche son sommet le plus proche *jmin* et l'on joint *i* à ce sommet *jmin* par un arc (utiliser la fonction *arrow*() qui dessine un trait avec une flèche).Rappelons que la distance au carré entre deux points *i* et *j* est

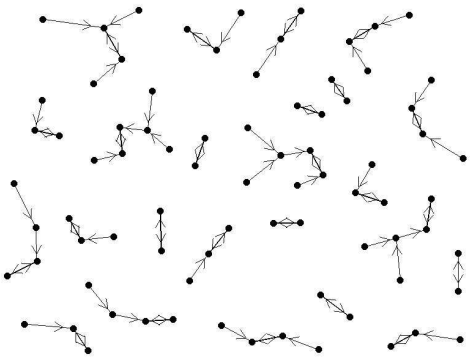
d2=(x[j]-x[i])(x[j]-x[i])+(y[j]-y[i])*(y[j]-y[i]);*

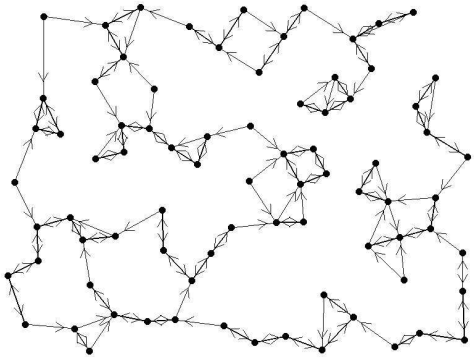
On obtient ainsi tous les voisins *v*[*i*][0], et l'on trace les arcs de jonction correspondants.

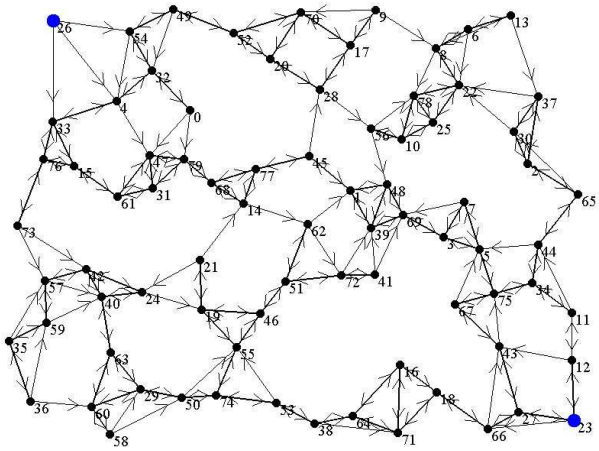
* Pour chaque sommet *i*, on cherche son sommet le plus proche autre que le *v*[*i*][0] déjà trouvé, et l'on trace l'arc de jonction. On obtient maintenant les voisins *v*[*i*][1].

* Pour chaque sommet *i*, on cherche son sommet le plus proche autre que *v*[*i*][0] et *v*[*i*][1], ce qui donne le voisin *v*[*i*][2] et l'on trace le troisième arc de jonction.

On connaît maintenant les sommets du graphe et leurs voisins placés dans les listes d'adjacence. Remarquons que certaines jonctions sont à sens unique, d'autres à double sens. D'autre part la méthode utilisée limite le nombre d'arcs qui se coupent. Si l'on veut éviter que chaque sommet ait toujours trois voisins, on peut faire quelques modifications, en enlevant parfois un arc, ou en rajoutant un arc ici ou là, toujours vers un point proche.







Programme

```
/** placement des sommets */
for(i=0;i<N;i++)
  { do  { x[i]=20+rand()%760; y[i]=20+rand()%560; }
    while (getpixel(x[i],y[i])==red);
    filldisc(x[i],y[i],30,red);
  }
 for(i=0;i<N;i++) filldisc(x[i],y[i],5,black);
 for(i=0;i<800;i++) for(j=0;j<600; j++) if (getpixel(i,j)==red) putpixel(i,j,white);
 SDL_Flip(screen);pause();  /* si l'on veut voir */
 /** premiers arcs */
 for(i=0;i<N;i++)
   { d2min=1000000000;
     for(j=0;j<N;j++) if (i!=j)
      { d2=(x[j]-x[i])*(x[j]-x[i])+(y[j]-y[i])*(y[j]-y[i]);
        if (d2<d2min)  { d2min=d2; jmin=j; }
      }
      v[i][0]=jmin;
   }
for(i=0;i<N;i++)  arrow(x[i],y[i],x[v[i][0]],y[v[i][0]],black);
   /** deuxièmes arcs */
 for(i=0;i<N;i++)
   { d2min=1000000000;
     for(j=0;j<N;j++) if (i!=j && j!=v[i][0])
      { d2=(x[j]-x[i])*(x[j]-x[i])+(y[j]-y[i])*(y[j]-y[i]);
        if (d2<d2min)  { d2min=d2; jmin=j; }
      }
      v[i][1]=jmin;
```

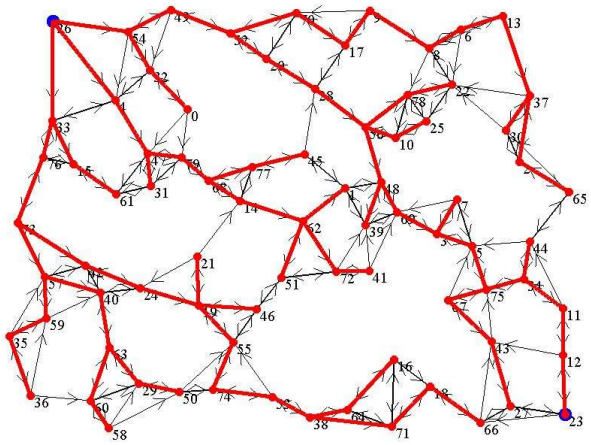
```
    }
for(i=0;i<N;i++)
arrow(x[i],y[i],x[v[i][1]],y[v[i][1]],black);
   /** troisièmes arcs */
  for(i=0;i<N;i++)
   { d2min=1000000000;
     for(j=0;j<N;j++) if (i!=j && j!=v[i][0] && j!=v[i][1])
      { d2=(x[j]-x[i])*(x[j]-x[i])+(y[j]-y[i])*(y[j]-y[i]);
        if (d2<d2min)  { d2min=d2; jmin=j; }
      }
      v[i][2]=jmin;
   }
for(i=0;i<N;i++) nbv[i]=3;
for(i=0;i<N;i++)  arrow(x[i],y[i],x[v[i][2]],y[v[i][2]],black);
SDL_Flip(screen);pause();
```

2) Exploration en largeur du graphe orienté

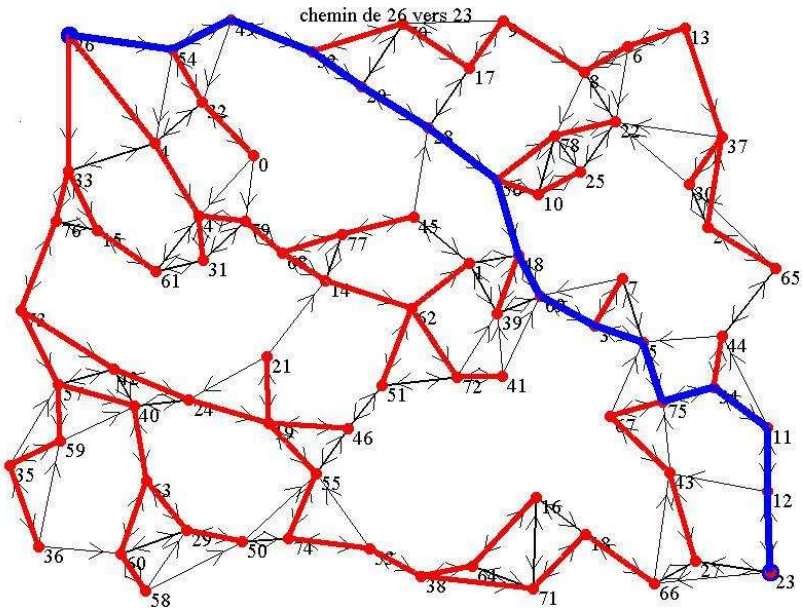
a) Prendre comme point de départ le sommet le plus en haut à gauche. Programmer pour connaître son numéro et colorier ce point en bleu (voir dessin ci-dessus).

Pour les besoins ultérieurs du programme, déterminer aussi un sommet *arrivee* situé le plus en bas à droite et colorier aussi ce point en bleu.

b) A partir du point de départ choisi précédemment, lancer une exploration du graphe **en largeur** en utilisant une file. Il pourra arriver que des sommets ne soient pas atteints. Dessiner les arcs de l'arbre de l'exploration en gros traits rouges.



c) Dessin d'un chemin le plus court (en nombre de stations traversées) entre le point de départ précédent, et le point d'arrivée *arrivee* déterminé précédemment. Pour cela, appeler la fonction *chemin*(*arrivee*), sous réserve que le point d'arrivée soit atteint par l'arbre d'exploration (il est alors en rouge) et programmer cette fonction récursive *void chemin*(int *k*). Dessiner le chemin obtenu en bleu, s'il existe, sinon indiquer qu'il n'y a pas de chemin.



3) Construction d'un graphe non orienté au hasard

On va reprendre intégralement le programme précédent donnant un graphe orienté. Puis on va ajouter des arcs, de façon à avoir des arcs à double sens là où ils étaient à sens unique. Sauf cas très exceptionnel on obtiendra un graphe connexe.

On commence par enregistrer tous les arcs orientés du programme du 1), en déterminant les extrémités *e1*[*i*] et *e2*[*i*] de chaque arc *i*. Par la même occasion, on détermine le nombre *nombrearcs0* de ces arcs. Puis on prend tous ces arcs *i* d'extrémités *e1*[*i*] et *e2*[*i*] et à chaque fois on regarde si l'on a aussi parmi eux un arc à l'envers d'extrémités *e2*[*i*] et *e1*[*i*]. Si l'on n'en trouve pas, on ajoute ce nouvel arc, et le nombre d'arcs augmente de 1. A la fin tous les arcs sont doublés, et le nombre d'arcs est *nombrearcs*. Le fait d'avoir partout des arcs à double sens revient à avoir des arêtes d'un graphe non orienté. Il reste à déterminer les listes des voisins *v*[*i*][[] de chaque sommet, au nombre de *nbv*[*i*].

```
for(i=0;i<N;i++) for(j=0;j<nbv[i];j++)
  { e1[k]=i;e2[k]=v[i][j]; k++;} /* extrémités des arcs */
nombrearcs0=k;
/* on va doubler tous les arcs qui ne le sont pas encore */
nombrearcs=nombrearcs0;  /* condition initiale pour le nombre d'arcs */
for(i=0;i<nombrearcs0;i++)  /* rajout d'arcs pour avoir des doubles arcs partout */
  { flag=0;
    for(j=0;j<nombrearcs0;j++)
    if (e1[j]==e2[i] && e2[j]==e1[i])  { flag=1; break;}
    if (flag==0)  {e1[nombrearcs]=e2[i];e2[nombrearcs]=e1[i];
                   nombrearcs++;
                  }
  }
 for(i=0;i<nombrearcs;i++)
  arrow(x[e1[i]],y[e1[i]],x[e2[i]],y[e2[i]],red);  /* dessin pour voir que tous les arcs ont deux flèches */
 SDL_Flip(screen);pause();
/* liste d'adjacence du graphe non orienté  */
for(i=0;i<N;i++) nbv[i]=0;  /* le nombre des voisisn est mis à 0 au depart */
for(i=0;i<nombrearcs; i++)  /* on parcourt l'ensemble des arcs, pour avoir les voisins de e1[i] */
  { v[e1[i]][nbv[e1[i]]++]=e2[i];
  }
```

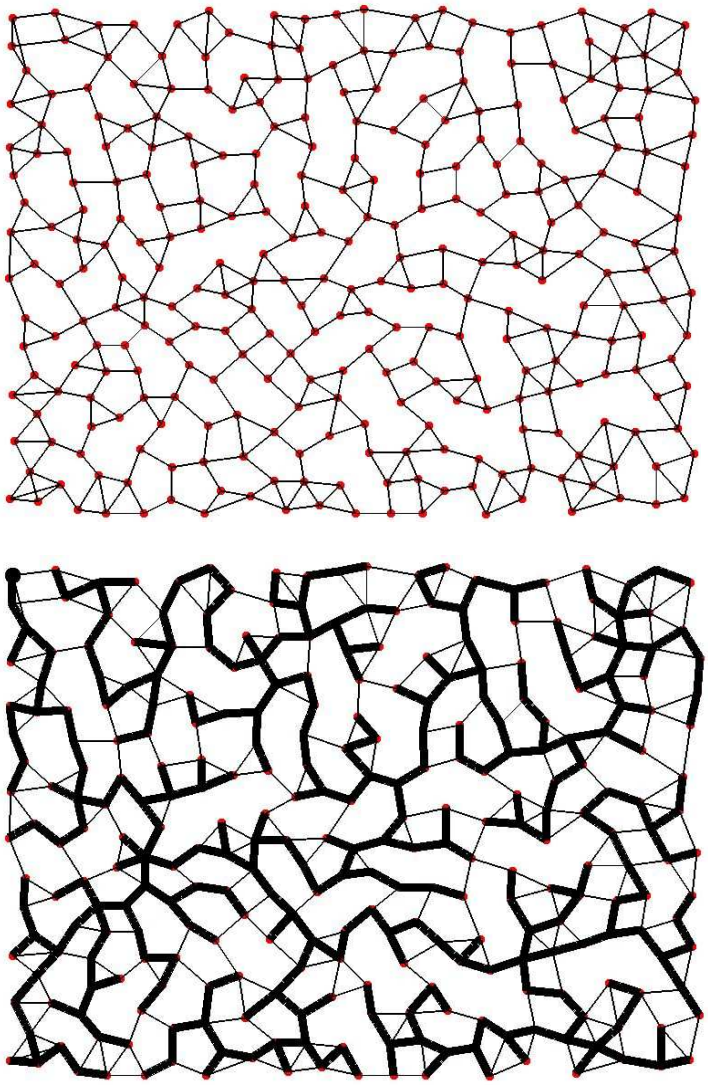
```
for(i=0;i<N;i++) for(j=0;j<nbv[i];j++)
line(x[i],y[i],x[v[i][j]],y[v[i][j]],red);  /* dessin des arêtes, elles sont tracées deux fois chacune */
```

4) Arbre couvrant minimal d'un graphe non orienté

On commence par déterminer la matrice d'adjacence pondérée à partir des listes d'adjacence trouvées au 3°. Les poids sont les longueurs des arêtes.

```
SDL_FillRect(screen,0,white);  /* re-dessin  éventuel*/
for(i=0;i<N;i++)filldisc(x[i],y[i],4,red);
for(i=0;i<N;i++)  for(j=0;j<nbv[i];j++)  line(x[i],y[i],x[v[i][j]],y[v[i][j]],black);
SDL_Flip(screen);pause();
/* matrice d'adjacence pondérée P[][] */
for(i=0;i<N;i++) for(j=0;j<N;j++) P[i][j]=100000000;
for(i=0;i<N;i++) P[i][i]=0;
for(i=0;i<N;i++)   for (j=0;j<nbv[i];j++)
    P[i][v[i][j]]=sqrt((x[v[i][j]]-x[i])*(x[v[i][j]]-x[i])+(y[v[i][j]]-y[i])*(y[v[i][j]]-y[i]));
```

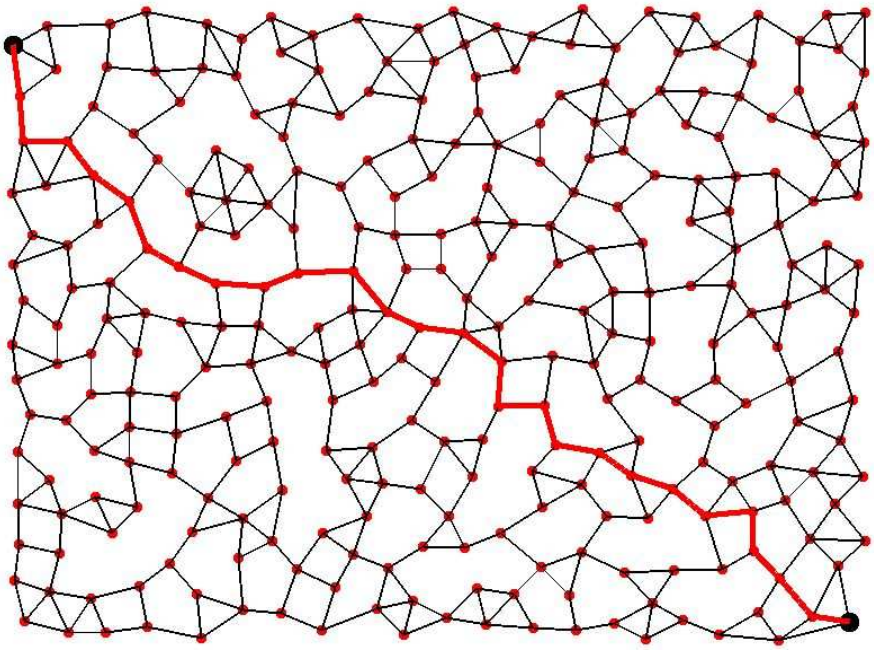
Puis utiliser l'algorithme de Prim pour obtenir l'arbre couvrant minimal, en prenant comme point de départ le sommet le plus en haut et à gauche.



Graphe non orienté de N = 300 sommets, avec son arbre couvrant minimal

5) Chemins les plus courts dans un graphe orienté ou non orienté

En reprenant le graphe non orienté construit au 3°, avec sa matrice d'adjacence pondérée faite au 4°, appliquer l'algorithme de Floyd pour trouver le chemin le plus court entre les deux sommets extrêmes du graphe.



6) Graphe complet, algorithme de Prim et problème du voyageur de commerce

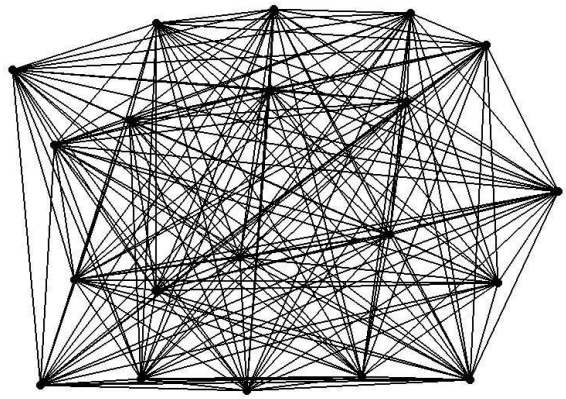
Le problème du voyageur de commerce est le suivant : on a *N* villes et des routes de jonction en ligne droite entre toutes les villes prises deux à deux. Un voyageur de commerce doit faire sa tournée en passant dans chaque ville une fois et une seule, en parcourant la distance minimale, et en revenant finalement à son point de départ.

Le graphe correspondant est un graphe complet, où chaque sommet est relié à tous les autres. Une tournée d'un voyageur de commerce consiste à parcourir des arêtes successives de ce graphe de façon à passer par chaque sommet une fois et une seule, puis à revenir au point de départ. Il s'agit d'un parcours cyclique. Il y a autant de cycles qu'il y a de permutations des sommets du graphe. Par exemple, pour $N = 20$, le nombre de tournées possibles en prenant chaque sommet comme point de départ à tour de rôle est 20 ! soit environ deux milliards de milliards. Le problème du voyageur de commerce consiste à déterminer la tournée la plus courte parmi toutes ces tournées. On peut se contenter de prendre toujours le même point de départ pour les tournées cycliques, ce qui fait 19 ! tournées possibles. Le nombre reste encore énorme et les calculs très longs. On est donc amené à chercher une solution approchée du problème du voyageur de commerce, donnant rapidement un résultat assez proche du résultat idéal. Pour cela on va utiliser l'algorithme de Prim.

1) Prendre *N* sommets au hasard (*N* de l'ordre de 20), comme on l'a fait au 1°, et faire le programme donnant la matrice d'adjacence pondérée *P*[][] du graphe complet en prenant comme poids la distance entre deux sommets.

```
/* listes des voisins */
for(i=0;i<N;i++)
  {k=0;
```

```
   for(j=0;j<N;j++) if (i!=j)    v[i][k++]=j;
   }
for(i=0;i<N;i++) nbv[i]=N-1;
/* dessin */
for(i=0;i<N;i++) for(j=0;j<nbv[i];j++)  line(x[i],y[i],x[v[i][j]],y[v[i][j]],black);
SDL_Flip(screen);pause();
/* Matrice d'adjacence pour Prim  */
for(i=0;i<N;i++) P[i][i]=0;
for(i=0;i<N;i++)   for (j=0;j<nbv[i];j++)
    P[i][v[i][j]]=sqrt((x[v[i][j]]-x[i])*(x[v[i][j]]-x[i])+(y[v[i][j]]-y[i])*(y[v[i][j]]-y[i]));
```



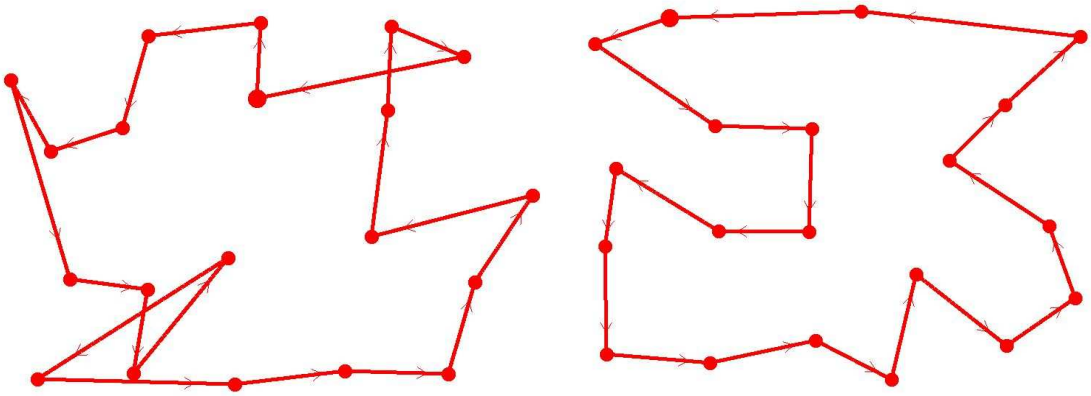
2) En prenant comme point de départ le sommet 0, faire le programme de l'algorithme de Prim donnant l'arbre couvrant minimal.

3) Dans le programme précédent, rajouter l'enregistrement des sommets *K* au fur et à mesure qu'ils sont pris pour l'arbre couvrant minimal. Cela se fait en remplissant un tableau *sommet*[]. Au début, on a *sommet*[0]=*ptdepart* (0 dans notre exemple), puis *chemin*[1] avec le premier sommet obtenu, etc., jusqu'à *sommet*[*N* – 1]. Enfin on ajoute *sommet*[*N*]=*ptdepart*, indiquant que l'on revient au point de départ. Au lieu de dessiner l'arbre couvrant minimal, dessiner le chemin obtenu en joignant les sommets successifs du tableau *sommet*[]. Il s'agit d'une tournée, parmi bien d'autres, du voyageur de commerce. Par la même occasion, déterminer la longueur de ce chemin, en faisant un cumul des distances, du style

```
cumuldistances[ptdepart]=0.;
for(i=0;i<NS;i++)  cumuldistances[ptdepart]+=P[sommet[i]][sommet[i+1]];
```

(on a pris pour *cumuldistances* un tableau en prévision de ce qui suit).

4) Répéter le programme précédent en prenant chaque sommet du graphe comme point de départ. Puis grâce à *cumuldistances*[*depart*], déterminer quel est le point de départ qui donne la longueur minimale du chemin. Dessiner ce chemin. C'est une solution approchée du problème du voyageur de commerce.



Résultats obtenus sur deux exemples de graphe à 20 sommets. Le deuxième dessin est bien meilleur que le premier, mais il n'arrive que rarement.

Pour améliorer les résultats et s'approcher encore plus de la solution idéale, il faudrait éliminer progressivement les croisements, ce qui n'est pas très simple à effectuer par programme.