Problème du voyageur de commerce

(*traveling salesperson problem*)

De l'algorithme de Prim au décroisement des arêtes

Etant données *N* villes, que l'on peut toujours joindre par des lignes droites les unes aux autres, il s'agit de trouver le plus court chemin, à partir d'une ville, qui permet de traverser chaque ville une fois et une seule, pour revenir finalement au point de départ. Cela s'appelle un circuit (ou cycle) hamiltonien sur un graphe complet de *N* sommets, « hamiltonien » signifiant que l'on passe sur chaque sommet une fois et une seule, l'objectif étant de trouver le circuit le plus court. Les poids des arêtes de jonction entre deux sommets quelconques seront pour nous les distances à vol d'oiseau. En faisant cela, nous traitons la version géométrique du problème du voyageur de commerce.

Il n'existe pas d'autre algorithme pour trouver ce chemin le plus court que de prendre tous les chemins possibles, soit N !, correspondant au nombre de façons de permuter les N sommets, pour finalement en déduire le plus court. Plus précisément, il suffit de traiter $(N - 1)$! cas car le point de départ du circuit peut être choisi arbitrairement, et même $(N - 1)$! / 2 cas puisque le circuit peut être parcouru dans un sens ou dans l'autre. Sachant qu'une factorielle devient vite très grande, par exemple 20 ! est de l'ordre de deux milliards de milliards, le temps mis par un tel algorithme est prohibitif dès que le nombre de villes dépasse la douzaine. Aussi doit-on utiliser des algorithmes approchés, qui à défaut du chemin le plus court, nous donnent un chemin proche de celui-ci. Nous allons donner ici une méthode, celle qui utilise l'algorithme de Prim relatif à l'arbre couvrant minimal d'un graphe.

Commençons par enregistrer le graphe complet de *N* sommets, ceux-ci étant pris au hasard avec une distance minimale entre eux. Ces sommets sont numérotés de 0 à $N - 1$. Ce que fait la fonction suivante :

```
void graphecomplet(void)
{ int i,j,k;
/* placement des sommets au hasard avec une distance minimale R, ici 40, entre eux*/
for(i=0;i<N;i++)
  { do  { x[i]=20+rand()%775; y[i]=20+rand()%570; }
    while (getpixel(x[i],y[i])==red);
    filldisc(x[i],y[i],40,red);  /* disque rouge de rayon 40 autour du sommet */
  }
for(i=0;i<N;i++) filldisc(x[i],y[i],5,black);  /* dessin des sommets */
for(i=0;i<800;i++) for(j=0;j<600; j++) if (getpixel(i,j)==red) putpixel(i,j,white); /* suppression du rouge */
for(i=0;i<N;i++) /* listes d'adjacence, avec les voisins de chaque sommet */
  {k=0;
   for(j=0;j<N;j++) if (i!=j)  v[i][k++]=j;  /* voisins v[i][] de chaque sommet i */
  }
for(i=0;i<N;i++) nbv[i]=N-1;  /* nbv[] est le nombre de voisins de chaque sommet 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]],black); /* les arêtes du graphe */
}
```

On en déduit la matrice *P* d'adjacence de ce graphe pondéré, avec des poids qui sont des longueurs, où $P[i][j]$ est la distance entre le sommet i et le sommet j (cette distance est mise à 0 lorsque $i = j$).

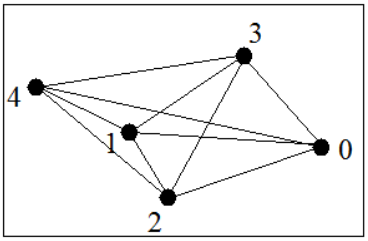
```
void matriceadjacencePrim(void)
{ int i,j;
  for(i=0;i<N;i++) P[i][i]=0;
  for(i=0;i<N;i++)   for (j=0;j<nbv[i];j++)  /* nbv[i] est en fait une constante pour un graphe complet */
  P[i][v[i][j]]=sqrt((x[v[i][j]]-x[i])*(x[v[i][j]]-x[i])+(y[v[i][j]]-y[i])*(y[v[i][j]]-y[i]));
}
```

1. Rappel sur l'algorithme de Prim

Par définition, un arbre couvrant minimal est un arbre qui a comme sommets tous les sommets du graphe et tel que la somme des poids de ses arêtes est la plus petite possible. Sa construction découle de la propriété suivante :

Coupons l'ensemble des *N* sommets du graphe en deux parties *S* et *T*. Alors parmi les arêtes de traverse joignant un sommet de *S* et un sommet de *T*, celle qui possède le poids minimal est une arête de l'arbre couvrant minimal.

De là découle l'algorithme de Prim. On se donne un sommet de départ, qui va être la racine de l'arbre. On met dans *S* ce sommet, et dans *T* tous les autres sommets du graphe. Puis on cherche l'arête de longueur minimale entre *S* et *T*, ce qui donne un sommet extrémité dans *T* que l'on transvase dans *S*. On vient de trouver la première arête de l'arbre couvrant minimal. Puis on recommence entre les sommets de *S* et ceux de *T*, ce qui donne une nouvelle arête de longueur minimale entre les deux, le sommet extrémité dans *T* étant déplacé vers *S*. Et ainsi de suite, jusqu'à ce que l'ensemble *T* soit vide. Prenons un exemple, avec le graphe complet suivant à cinq sommets numérotés de 0 à 4.



- *Etape* 1 : Partons du sommet de départ 0, d'où les tableaux *S* et *T* à l'étape 1 initiale, de longueurs *LS* = 1 et *LT* = 4 respectivement :

S : 0 *T* : 1 2 3 4 , avec *T*[0] = 1 jusqu'à *T*[3] = 4, où les indices sont les positions de chaque sommet dans le tableau.

Par la même occasion, on remplit alors un tableau *poids*[] où l'on met les longueurs des arêtes entre 0 et les points de *T*, soit

1 2 3 4

poids[] : 5,5 4,5 3,5 8, où les indices du tableau sont ici les sommets de *T*.

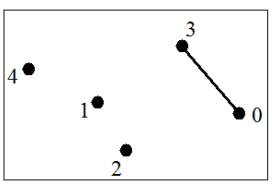
On fait de même avec un tableau *pred*[], où l'on place les extrémités des arêtes aboutissant aux sommets de *T*, et venant de *S*, soit :

1 2 3 4

pred[] : 0 0 0 0 , ici encore les indices du tableau sont les sommets de *T*. On a fini l'étape 1 initiale.

- *Etape* 2 : Grâce au tableau *poids*[], on détermine l'arête de longueur minimale entre *S* et *T*. On trouve dans *T* le sommet *K* = 3, avec l'arête (03), où 0 est enregistré dans *pred*[3] = 0. On déplace le sommet 3 de *T* vers *S*, soit :

S : 0 3 *T* : 1 2 4.



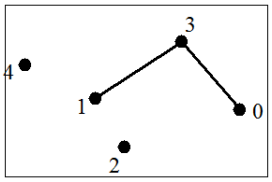
Puis on actualise les tableaux *poids*[] et *pred*[], dans le cas où l'on trouve des arêtes entre le nouveau sommet 3 de *S* et les sommets de *T* plus courtes que celles auparavant trouvées. Cela donne :

sommets numérotés : 1 2 4

poids[] : 4 4,5 6

pred[] : 3 0 3

- *Etape* 3 : On détermine l'arête minimale entre *S* et *T*. On trouve dans *T* le sommet *K* = 1, et l'arête correspondante est (13) avec *pred*[1] = 3. On déplace 1 de *T* vers *S*, soit *S* : 0 3 1 et *T* : 2 4.



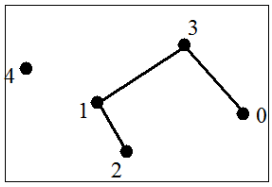
Puis on actualise les tableaux *poids*[] et *pred*[] en tenant compte du nouveau point 1 dans *S* :

2 4

poids[] : 2 2,5

pred[] : 1 1 , ce qui signifie que le sommet le plus proche des sommets 2 et 4 est 1, et non plus ni 3 ni 0.

- *Etape* 4 : On trouve dans *T* le sommet *K* = 2 et l'arête (21), d'où *S* : 0 3 1 2 et *T* : 4.



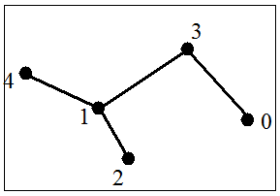
L'actualisation des tableaux *poids*[] et *pred*[] ne produit pas de changement dans le cas présent :

4

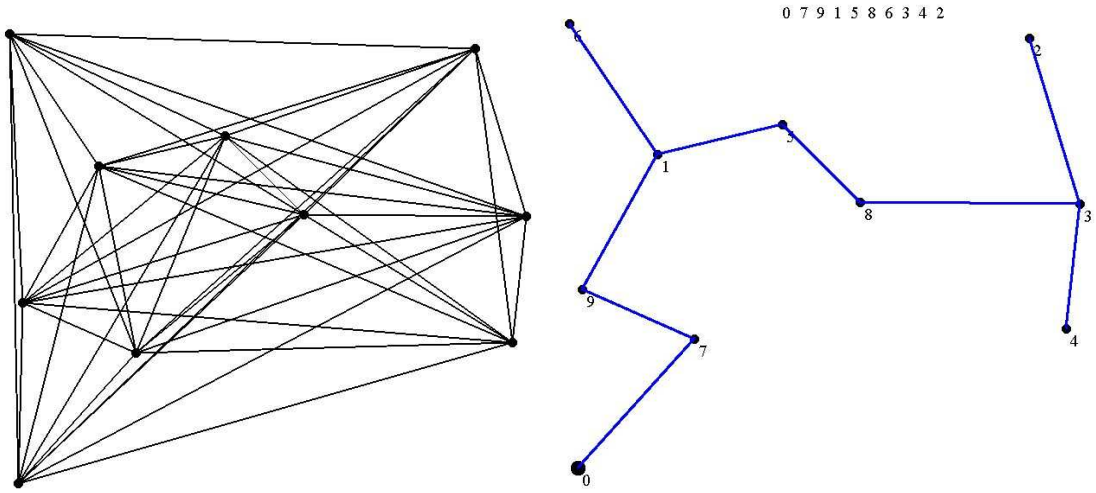
poids[] : 2,5

pred[] : 1

- *Etape* 5 : Le dernier sommet est 4, l'arête minimale est (41), *S* devient 0 3 1 2 4 et *T* est vide. C'est fini. On a trouvé l'arbre couvrant minimal de racine 0. Chaque arête est obtenue en joignant un sommet (autre que 0) à son prédécesseur *pred*[]. Remarquons qu'en partant d'un autre point de départ que 0, on trouverait le même arbre couvrant minimal, sauf dans les cas exceptionnels où il peut en exister plusieurs ayant la même longueur.



Le programme correspondant sera donné ci-dessous, comme première partie de la fonction *Primet cyclepourpointdedepart*(). Voici le résultat obtenu pour un graphe complet à 10 sommets.



A gauche, le graphe complet, *à droite* l'arbre couvrant minimal en partant du sommet 0, avec au-dessus le tableau *S* finalement obtenu.

2. Détermination d'un cycle hamiltonien

La méthode consiste à utiliser le tableau *S* trouvé pour l'arbre couvrant minimal. Reprenons l'exemple précédent, avec *S* = 0 7 9 1 5 8 6 3 4 2, et joignons les points successifs par des traits, en finissant par la jonction (2 0) entre le point final et le point initial. On vient d'obtenir un cycle hamiltonien. Celui-ci contient plusieurs des arêtes de l'arbre couvrant minimal correspondant à des plus courtes distances. Il s'agit d'une approximation très grossière du problème du voyageur de commerce. Une méthode légèrement supérieure, comme on le constate en procédant à des expérimentations, consiste à lire le tableau *S* tout en le transformant en un nouveau tableau *cycle*.

Commençons par mettre l'élément initial de *S* dans *cycle*, et cela deux fois : *cycle*[0] = 0 et *cycle*[1] = 0, selon l'exemple précédent. L'élément *cycle*[0] va rester fixe, seuls les éléments à partir de l'indice 1 vont pouvoir bouger, notamment l'élément 0 de *cycle*[1] va se retrouver en dernier, comme on va voir. Prenons maintenant chaque élément de *S* à tour de rôle et plaçons-le juste devant son prédécesseur dans le tableau *cycle*[]. Rappelons que dans l'algorithme de Prim chaque nouvel élément de *S* (sauf l'élément initial, ici 0) est transvasé depuis *T*, et que son prédécesseur (l'autre extrémité de l'arête de jonction) se trouve déjà dans *S*. Dans notre exemple, cela donne l'évolution suivante :

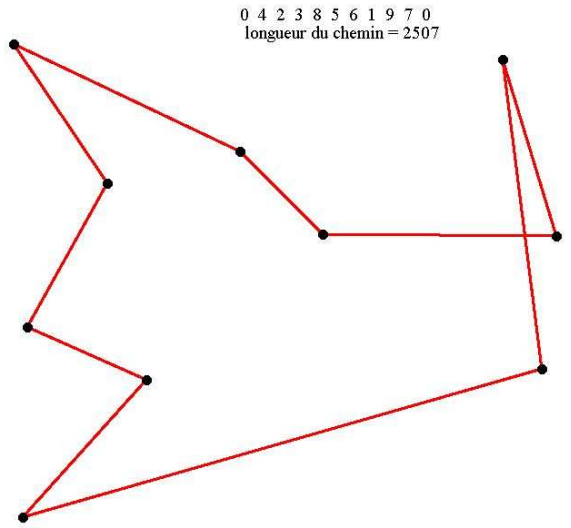
0 0 → (avec 0 prédécesseur de 7) 0 7 0 → 0 9 7 0 → 0 1 9 7 0 → 0 5 1 9 7 0 → 0 8 5 1 9 7 0 → 0 8 5 6 1 9 7 0 → 0 3 8 5 6 1 9 7 0 → 0 4 3 8 5 6 1 9 7 0 → 0 4 2 3 8 5 6 1 9 7 0.

Nous sommes maintenant en mesure de faire le programme de la fonction *Primetcyclepourpointdedepart*(*ptdepart*) qui détermine l'arbre couvant minimal, puis le tableau *cycle*[] et la longueur de ce cycle dans *cumuldistance*[]. Cette fonction prend comme variable le point de départ qui est la racine de l'arbre couvrant minimal. Nous avions pris ici le sommet 0 comme racine, mais la fonction permet de choisir un sommet quelconque pour démarrer.

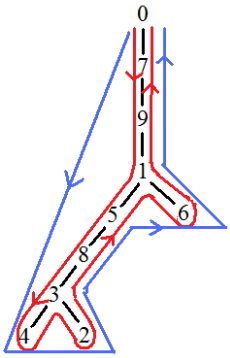
```
void Primetcyclepourpointdedepart(int ptdepart)
{ int i,j,k,imin,pred[N],S[N],T[N],weight[N],LS,LT,step,pmin,K;
  int departmini,ii;
/* Algorithme de Prim donnant l'arbre couvrant minimal à partir de la racine ptdepart */
   S[0]=ptdepart; LS=1;  filldisc(x[ptdepart],y[ptdepart],4,black);
   k=0; for(i=0;i<N;i++) if (i!=ptdepart) T[k++]=i; LT=N-1;
   for(i=0;i<LT;i++) weight[T[i]]=P[ptdepart][T[i]];
   for(i=0;i<LT;i++) pred[T[i]]=ptdepart;
   for(step=2;step<=N;step++)
    {  pmin=10000000;
       for(i=0;i<LT;i++)
       if (weight[T[i]]<pmin)    { pmin=weight[T[i]];      imin=i;}
       K=T[imin];
       S[LS]=K; LS++;
      /* éventuellement faire le dessin : linewithwidth(x[K],y[K],x[pred[K]],y[pred[K]],1,blue);*/
       for(i=imin;i<LT-1;i++) T[i]=T[i+1];
       LT--;
       for(i=0;i<LT;i++)
       if (P[K][T[i]]<weight[T[i]]) { weight[T[i]]=P[K][T[i]]; pred[T[i]]=K;}
    }
  /* Construction du cycle hamiltonien correspondant */
  cycle[0]=ptdepart;
  cycle[1]=ptdepart;
  for(i=1;i<N;i++)
    { ii=i;
       while(cycle[ii]!=pred[S[i]]) { cycle[ii+1]=cycle[ii]; ii--;}
       cycle[ii+1]= cycle[ii]; cycle[ii]=S[i];
    }
     /* éventuellement afficher ici le tableau cycle[] */
   cumuldistances[ptdepart]=0.;
```

```
    for(i=0;i<N;i++)  cumuldistances[ptdepart]+=P[cycle[i]][cycle[i+1]];
}
```

Voici le cycle hamiltonien obtenu en reprenant l'exemple précédent du graphe à dix sommets :



Revenons à la théorie. En joignant les sommets successifs du tableau *cycle*[] qui possède $N + 1$ éléments, on trouve un cycle hamiltonien. Nous allons montrer que la longueur de ce cycle hamiltonien, notée *LCH*, est inférieure ou égale à deux fois la longueur *LCHM* du cycle hamiltonien minimal, celui de notre voyageur de commerce.



Reprenons l'arbre couvrant minimal, de longueur *LACM*, et faisons un parcours de cet arbre, ce qui correspond au chemin rouge sur le dessin. Dans ce parcours, chaque arête est parcourue deux fois, une fois en descente, et une fois en remontée. La longueur du parcours rouge est 2 *LACM*.

Tel que le tableau *cycle*[] a été construit, il correspond, à partir de *cycle*[1], à une lecture « postfixée » de l'arbre, obtenue en notant les sommets dans l'ordre où ils sont obtenus lors des remontées dans le parcours rouge.[1] Par exemple, le premier sommet lu lors d'une remontée est 4, le suivant est 2, etc.

Le cycle hamiltonien correspondant à *cycle*[], et de longueur *LCH*, tracé en bleu sur le dessin, joint les sommets dans le même ordre que la lecture postfixée de l'arbre, mais en ligne directe parfois, et en longeant certaines arêtes de l'arbre une seule fois et jamais deux fois. On peut donc affirmer que

$LCH \leq 2\ LACM$

Prenons enfin le cycle hamiltonien minimal de longueur *LCHM*. Si on lui enlève son arête la plus longue, de longueur *d*, il n'est plus un cycle, et il devient un arbre, qui est l'arbre couvrant minimal. Ainsi : $LCMH - d = LACM$, ou encore $LCMH > LACM$.

Finalement : $LCH \leq 2\ LCMH$

Le fait d'avoir trouvé un cycle qui au pire serait deux fois plus long que le cycle de longueur minimale n'est pas très glorieux, et donne une approximation grossière du problème du voyageur de

[1] Le parcours « préfixé » consisterait, lui, à lire les sommets dans l'ordre où ils sont obtenus au cours des descentes.

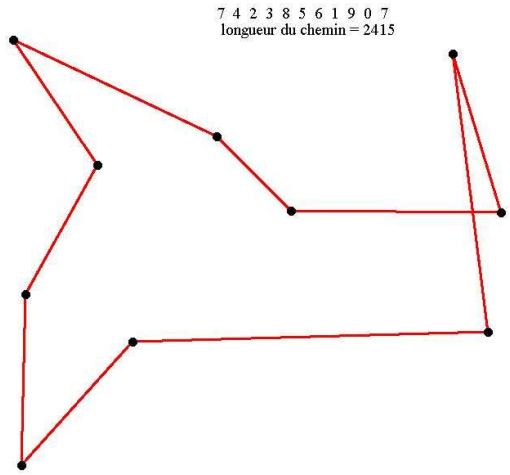
commerce. Mais les expérimentations indiquent que l'on obtient en général une approximation tout à fait acceptable. Et cela d'autant plus que l'on peut procéder à des améliorations.

3. Première amélioration

La fonction précédente permettait de choisir un point de départ quelconque. En prenant chacun des *N* sommets comme point de départ, on trouve le même arbre couvrant minimal mais avec une racine différente, et le cycle hamiltonien est modifié. Il s'agit de choisir le cycle hamiltonien le plus court parmi les *N* résultats obtenus. C'est ce que fait la fonction *cycleminimal*() :

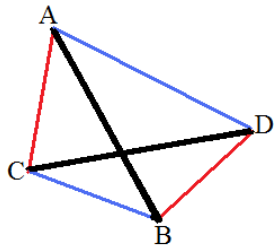
```
void cycleminimal(void)
{  int ptdepart,departmini,distancemini;
   distancemini=10000000;
   for(ptdepart=0;ptdepart<N;ptdepart++)
     { Primetcyclepourpointdedepart(ptdepart);
       if (cumuldistances[ptdepart]< distancemini)
         { distancemini=cumuldistances[ptdepart]; departmini=ptdepart;  }
     }
  Primetcyclepourpointdedepart(departmini);  /* enregistrement dans cycle[] du chemin minimal */
}
```

Dans notre exemple précédent, le cycle hamiltonien le plus court parmi les *N* = 10 cycles, est celui où l'on part du sommet 7 pour construire l'arbre couvrant minimal :



4. Deuxième amélioration : le décroisement des arêtes

On constate que le cycle hamiltonien obtenu a des arêtes qui se croisent, et plus *N* est grand, plus le nombre de croisements augmente. Cela nous invite à modifier le cycle obtenu en supprimant les croisements. Mais lorsqu'on supprime un croisement, cela peut en créer d'autres, et l'on pourrait s'imaginer que l'on s'engage dans un cercle vicieux, sans fin. En fait il n'en est rien. Cela tient à la propriété d'un « décroisement » : il diminue strictement la longueur d'un chemin.



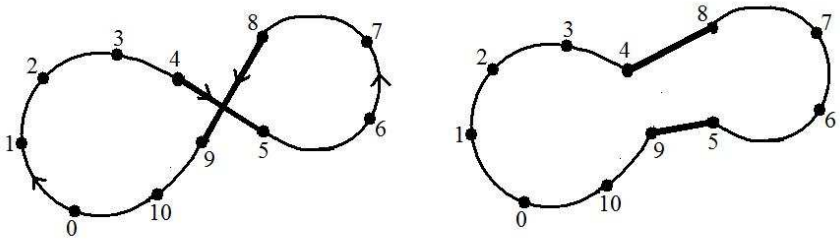
Considérons deux arêtes qui se croisent, soit [*AB*] et [*CD*]. Lorsqu'on les remplace par [*AC*] et [*BD*] (en rouge), ou par [*AD*] et [*CB*] (en bleu), la somme des longueurs des deux segments rouges ainsi que celle des deux segments bleus, sont toutes deux plus petites que la somme des longueurs de diagonales *AB* + *CD*.

Lors du parcours de notre cycle hamiltonien, on va jusqu'au premier croisement d'arêtes, et l'on procède au décroisement, ce qui modifie le cycle hamiltonien et diminue sa longueur. Puis on recommence en cherchant le premier croisement, que l'on décroise, ce qui diminue la longueur du nouveau cycle hamiltonien. Et ainsi de suite, chaque décroisement provoquant une diminution stricte de la longueur du cycle. Mais cette diminution ne peut pas tendre vers 0, car les sommets du graphe ont toujours une distance minimale entre eux. Dans ces conditions, comme tout cycle hamiltonien est minoré par le cycle minimal, celui du voyageur de commerce, le nombre de décroisements est forcément un nombre fini. Le processus des décroisements a une fin.

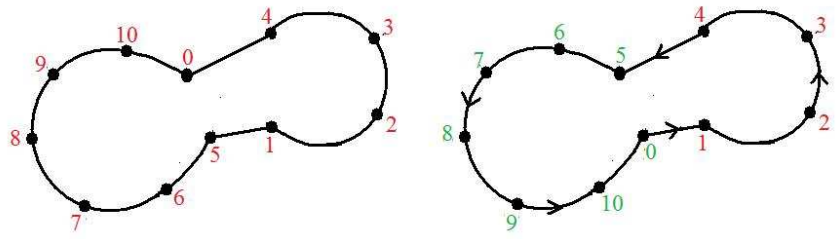
Prenons un cycle hamiltonien que nous avons obtenu, celui-ci est connu par le tableau *cycle*[]. Commençons par chercher son premier croisement. Cela revient à tester chacune des arêtes dont les sommets sont indexés par i et i+1, en commençant par $i = 0$, avec celles qui sont après elle, dont les sommets sont j et $j + 1$, avec $j > i$, jusqu'à trouver un croisement entre les arêtes indexées par (i, i+1) et (j, j+1) dans le tableau *cycle*[]. On utilise pour cela la fonction *intersection*() qui dit si oui ou non deux segments se coupent (voir pour cela dans *enseignements*, le *cours graphisme et géométrie*, chapitre *Angles, orientation, enveloppe convexe*).

Que faire lorsque l'on trouve un croisement entre (i, i+1) et (j, j+1) avec j après i ? On efface ces deux arêtes, puis on les remplace par les arêtes dont les sommets sont pour indices (i, j) et (i+1, j+1). Mais cela ne suffit pas. Encore faut-il que le nouveau cycle soit enregistré avec ses sommets indexés par des numéros successifs de 0 à N, ce qui demande un changement de numérotation.

Prenons un exemple, avec les arêtes qui se croisent pour $i = 4$ et $j = 8$, et $N = 11$, comme sur le dessin suivant, où l'on efface les arêtes (4 5) et (8 9) au profit de (4 8) et (5 9) :



Mais dans le nouveau cycle obtenu, les indices ne sont plus dans l'ordre croissant. On procède alors à un premier réaménagement en décalant des indices de $i = 4$, plus précisément en enlevant 4 modulo N, ce qui donne la numérotation en rouge sur le dessin ci-dessous à gauche. Maintenant la numérotation est bonne sur la partie droite. Il reste à changer celle de la partie gauche, en procédant à des échanges 5 ↔ 0, 6 ↔ 10, 7 ↔ 9, 8 ↔ 8, ces numéros étant en vert sur le dessin de droite. Le nouveau cycle, bien numéroté, est obtenu.

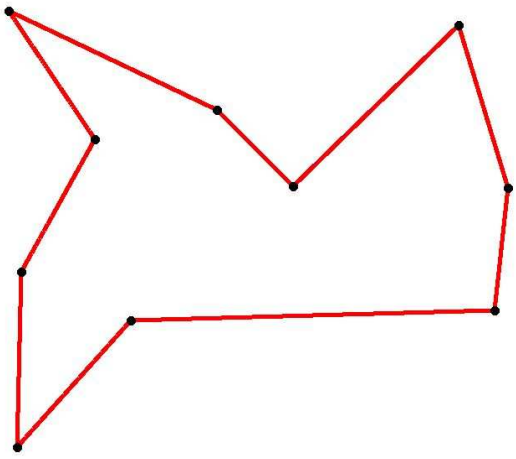


Une fois cela fait, on relance un parcours de ce cycle, à la recherche du premier croisement, et l'on recommence le décroisement, jusqu'à ce qu'il n'y en ait plus. D'où la fonction suivante *cheminminimaldecroise*(), qui traite le cycle hamiltonien enregistré dans la variable globale *cycle*[] à la fin de la fonction précédente *cycleminimal*().

```
void cheminminimaldecroise(void)
{ int i,j,aux,g,d,ee[N+1],ii,flag,cumuldist;
  for(;;)
    { flag=0;
      for(i=0;i<N-1;i++)  for(j=i+1;j<N;j++)
      if (intersection(i,j)==1)
        { flag=1;
          linewithwidth(x[cycle[i]],y[cycle[i]],  x[cycle[i+1]],y[cycle[i+1]],2,white); /* effacement */
          linewithwidth(x[cycle[j]],y[cycle[j]], x[cycle[j+1]],y[cycle[j+1]],2,white);
          for(ii=0;ii<N;ii++) ee[ii]=cycle[(ii+i)%N];
          for(ii=0;ii<N;ii++) cycle[ii]=ee[ii];  /* décalage de i dans cycle[] */
          aux=cycle[(j+1-i)%N];cycle[(j+1-i)%N]=cycle[0];cycle[0]=aux;  /* échanges et mise à l'envers*/
          g=j+2-i; d=N-1;
          while(g<d) { aux=cycle[g]; cycle[g]=cycle[d]; cycle[d]=aux;  g++;d--;   }
          cycle[N]=cycle[0];
          for(i=0;i<N;i++)
          linewithwidth(x[cycle[i]],y[cycle[i]],x[cycle[i+1]],y[cycle[i+1]],2,red); /* dessin du cycle */
          SDL_Flip(screen);SDL_Delay(100); /* affichage des décroisement successifs */
          j=N;i=N-1; /* arrêt des boucles for pour i et pour j, le décroisement ayant été fait */
        }
     if (flag==0) break;  /* test d'arrêt pour la boucle infinie for(;;) */
    }
 cumuldist=0.;  /* calcul de la longueur du cycle décroisé final */
 for(i=0;i<N;i++)  cumuldist+=P[cycle[i]][cycle[i+1]];
 sprintf( chiffre," longueur du chemin decroise = %d",cumuldist);
 texte=TTF_RenderText_Solid(police,chiffre,couleurnoire);
 position.x=250; position.y=-7;
SDL_BlitSurface(texte,NULL,screen,&position);
}

int intersection(int i, int j) /* intersection entre (i, i+1) et (j, j+1) avec i, j indices des sommets dans cycle[]*/
{ int abx,aby,acx,acy,adx,ady,cdx,cdy,cax,cay,cbx,cby;  float det1,det2,det3,det4;
   abx=x[cycle[i+1]]-x[cycle[i]];    aby=y[cycle[i+1]]-y[cycle[i]];
   acx=x[cycle[j]]-x[cycle[i]];       acy=y[cycle[j]]-y[cycle[i]];
   adx=x[cycle[j+1]]-x[cycle[i]];    ady=y[cycle[j+1]]-y[cycle[i]];
   det1=(float) (abx*acy-aby*acx);       det2=(float)(abx*ady-aby*adx);
   cdx=x[cycle[j+1]]-x[cycle[j]];    cdy=y[cycle[j+1]]-y[cycle[j]];
   cax=-acx;    cay=-acy;     cbx=x[cycle[i+1]]-x[cycle[j]];     cby=y[cycle[i+1]]-y[cycle[j]];
   det3=(float)cdx*(float)cay-(float)cdy*(float)cax;       det4=(float)(cdx*cby-cdy*cbx);
   if (det1*det2<0.  && det3*det4<0.) return 1;
   return 0;
}
```

Exemple du graphe précédent avec N = 10 points

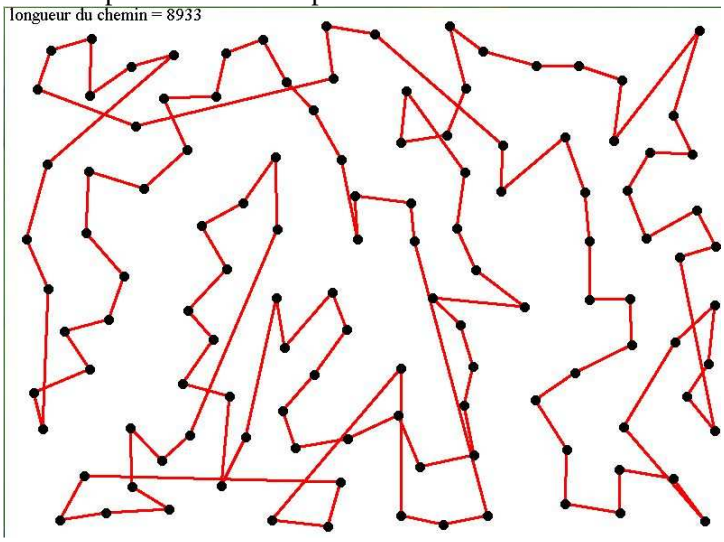


Le programme principal consiste à mettre en succession les onctions précédentes :

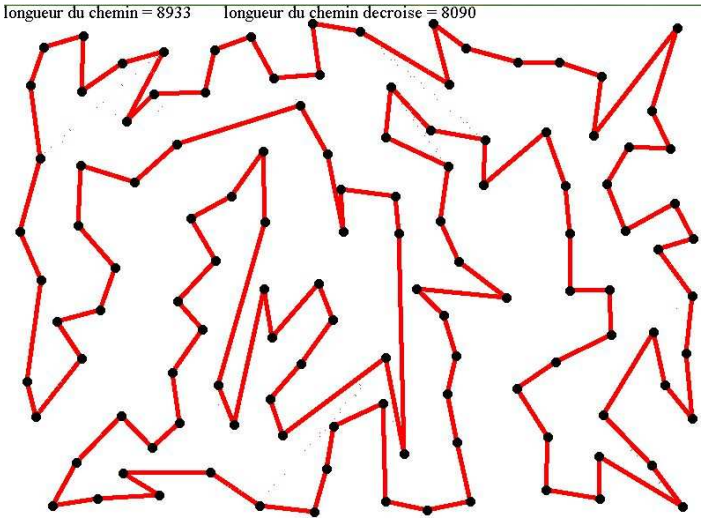
```
int x[N],y[N],v[N][N],nbv[N],P[N][N],cycle[N+1];  /* variables globales */
int cumuldistances[N],departmini;
int main(int argc, char ** argv)
{  SDL_Init(SDL_INIT_VIDEO); srand(time(NULL));
   screen=SDL_SetVideoMode(800,600,32, SDL_HWSURFACE|SDL_DOUBLEBUF);
   white=SDL_MapRGB(screen->format,255,255,255);
   black=SDL_MapRGB(screen->format,0,0,0);
   red=SDL_MapRGB(screen->format,255,0,0);
   blue=SDL_MapRGB(screen->format,0,0,255);
   SDL_FillRect(screen,0,white);   TTF_Init();  police=TTF_OpenFont("times.ttf",20);

 graphecomplet();  SDL_Flip(screen);pause();SDL_FillRect(screen,0,white);
 matriceadjacencePrim();
 cycleminimal();  dessiner le cycle;  SDL_Flip(screen);pause();
 cheminminimaldecroise();
 SDL_Flip(screen);pause(); return 0;
}
```

Un exemple avec N = 120 points



Cycle hamiltonien minimal parmi les 120 cycles déduits de l'arbre couvrant minimal



Cycle hamiltonien après décroisement, solution approchée du problème du voyageur de commerce