

Problema Comis-Voiajorului

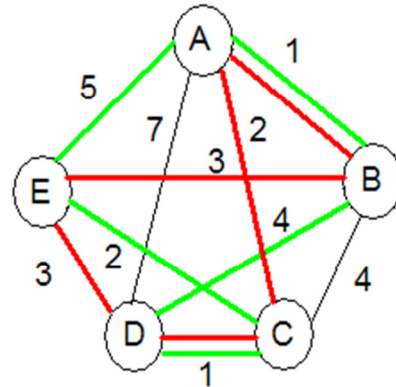
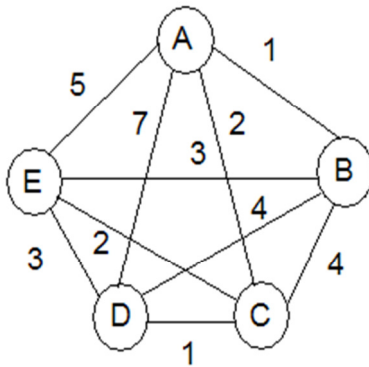
Problema Comis-Voiajorului este definită astfel:

Fie $G = (V, E)$ este un graf neorientat în care oricare două vârfuri diferite ale grafului sunt unite printr-o latură căreia îi este asociat un cost strict pozitiv. Cerința este de a determina un ciclu care începe de la un nod aleatorie a grafului, care trece exact o dată prin toate celelalte noduri și care se întoarce la nodul inițial, cu condiția ca ciclul să aibă un cost minim. Costul unui ciclu este definit ca suma tuturor costurilor atașate laturilor ciclului.

Numele problemei provine din analogia cu un vânzător ambulant care pleacă dintr-un oraș, care trebuie să viziteze un număr de orașe dat și care apoi trebuie să se întoarcă la punctul de plecare, cu un efort minim (de exemplu timpul minim, caz în care costul fiecărei laturi este egal cu timpul necesar parcurgerii drumului).

Prin urmare, fie $G = (V, E)$ un graf conectat neordonat, definit de un set de noduri V și de un set de laturi E .

Dacă vom marca cu $V' = \{v_i \mid i = 1, \dots, n-1\}$ set de noduri asociate orașelor care trebuie să fie vizitate și dacă vom marca cu v_0 nodul care este asociat cu orașul din care vânzător ambulant pleacă (de bază), vom avea $V = V' \cup \{v_0\}$. Am marca, de obicei, lungimea laturii dintre nodurile v_i și v_j cu l_{ij} și costul asociat parcurgerii laturii dintre v_i și v_j cu c_{ij} .



Marginile colorate în verde, ne dau un ciclu care porneste de la nodul A, având costul următor:
 $1 + 4 + 1 + 2 + 5 = 13$ pe calea A-B-D-C-E-A

Marginile colorate în roșu ne dau un ciclu care porneste de la nodul B, având costul următor:
 $1 + 2 + 1 + 3 + 3 = 10$ pe calea B-A-C-D-E-B

Prin urmare, al doilea ciclu este mai bun ca primul, având în vedere costul redus.

Metode de soluționare

Căile principale de abordare ale problemelor de tipul « NP » sunt următoarele :

- Algoritmi pentru găsirea soluțiilor exacte (ex. **Backtracking** - aceștia vor funcționa rezonabil doar pentru problemele de dimensiuni relativ mici)
 - Prin metoda *backtracking*, orice vector soluție este construit *progresiv*, începând cu prima componentă și mergând către ultima, cu eventuale *reveniri* asupra valorilor atribuite anterior.
 - Metoda *backtracking* urmărește să evite generarea tuturor soluțiilor posibile, scurtându-se astfel timpul de calcul.
- Născocirea metodelor **euristice** de rezolvare aproximativă a TSP (metode bune, dar care nu pot fi considerate optime)
 - În situațiile în care pentru anumite probleme complexe, pentru a caror rezolvare nu se cunosc algoritmi, sau aceștia sunt ineficienți (timp, memorie, cost), se preferă utilizarea unor algoritmi care rezolvă problema dată mult mai rapid, cu efort mic, însă nu ne va furniza întotdeauna cea mai bună soluție, ci doar soluții acceptabile, adică soluții corecte care pot fi eventual îmbunătățite.
 - Prin algoritmi euristici vom înțelege un algoritm care furnizează soluții bune, nu neapărat optime, care poate fi implementat rapid și furnizează rezultate în timp util.
 - O idee frecvent utilizată constă în descompunerea procesului de determinare a soluției în mai multe etape succesive pentru care se poate determina optimul local.
- Găsirea unor cazuri speciale pentru problemă (**Greedy**)
 - În strategia *backtracking* căutarea soluției, adică vizitarea secvențială a nodurilor grafului soluțiilor cu revenire pe urma lăsată, se face oarecum “orbește” sau rigid, după o regulă simplă care să poată fi rapid aplicată în momentul “părăsirii” unui nod vizitat. În cazul metodei (strategiei) *greedy* apare suplimentar ideea de a efectua în acel moment o alegere. Dintre toate nodurile următoare posibile de a fi vizitate sau dintre toți pașii următori posibili, se alege acel nod sau pas care asigură un maximum de “câștig”, de unde și numele metodei: *greedy = lacom*.
 - Aparent această metodă de căutare a soluției este mai eficientă, din moment ce la fiecare pas se trece dintr-un optim (parțial) într-altul. Totuși, ea nu poate fi aplicată în general ci doar în cazul în care există certitudinea alegerii optime la fiecare pas, certitudine rezultată în urma etapei anterioare de analiză a problemei.

Variante de rezolvare

1. Metoda backtracking

1.1 Considerente teoretice

Metoda backtracking urmareste sa evite generarea tuturor solutiilor posibile, scurtandu-se astfel timpul de calcul.

Componentele vectorului x primesc valori in ordinea crescatoare a indicilor. Aceasta inseamna ca lui x_k nu i se atribuie valori decat dupa ce $x_1, \dots, x_{k-1}$ au primit valori care nu contrazic conditiile interne. Mai mult, valoarea lui x_k trebuie aleasa astfel incat $x_1, \dots, x_k$ sa indeplineasca si ele anumite conditii, numite *conditii de continuare*, care sunt strans legate de conditiile interne.

Astfel, daca in exemplul dat componentele x_1 si x_2 au primit amandoua valoarea X , atunci lui x_3 nu i se mai poate atribui aceasta valoare (pentru a nu incalca restrictia ca numarul maxim de pronosticuri X sa fie cel mult 2).

Neindeplinirea conditiilor de continuare exprima faptul ca oricum am alege $x_{k+1}, \dots, x_n$, nu vom obtine o solutie (deci conditiile de continuare sunt *strict necesare* pentru obtinerea unei solutii). Ca urmare se va trece la atribuirea unei valori lui x_k doar cand conditiile de continuare sunt indeplinite. In cazul neindeplinirii conditiilor de continuare, se alege o noua valoare pentru x_k sau, in cazul cand multimea finita de valori V_k a fost epuizata, se incearca sa se faca o noua alegere pentru componenta precedenta x_{k-1} a vectorului, micsorand pe k cu o unitate etc. Aceasta revenire da numele metodei, exprimand faptul ca atunci cand nu putem avansa, urmarim inapoi secventa curenta din solutie.

Trebuie observat ca in anumite cazuri, faptul ca $v_1, v_2, \dots, v_{k-1}$ satisfac conditiile de continuare *nu este suficient* pentru a garanta ca se va obtine o solutie alei care prime $k-1$ componente coincid cu aceste valori. De pilda, chiar daca x_1 si x_2 sunt X , iar x_3 este 1 (deci aceste valori indeplinesc conditiile de continuare curente), totusi $(X, X, 1, 1)$ nu este solutie.

Alegerea conditiilor de continuare este foarte importanta, o alegere buna ducand la o reducere substantiala a numarului de calcule. In cazul ideal, conditiile de continuare ar trebui sa fie nu numai *necesare*, dar si *suficiente* pentru obtinerea unei solutii. De obicei insa, acestea reprezinta restrictia conditiilor interne la primele k componente ale vectorului. Evident, conditiile de continuare in cazul $k=n$ sunt chiar conditiile interne.

Prin metoda *backtracking*, orice vector solutie este construit *progresiv*, incepand cu prima componenta si mergand catre ultima, cu eventuale *reveniri* asupra valorilor atribuite anterior. Reamintim ca $x_1, x_2, \dots, x_n$ primesc valori in multimile $v_1, \dots, v_n$.

```

<2> Procedure back()
|   pentru i=1,n
|       x[i] ← 0
|   k ← 2
|   x[1] ← 1
|   cat timp k>1
|       |       daca (k=n+1)
|       |           |       pentru i=1,n
|       |           |           |       scrie x[i]
|       |           |           |       ____#
|       |           |           |       k ← k-1
|       |           |       ____#
|       |           altfel
|       |               |       daca x[k]<n
|       |               |           |       x[k] ← x[k]+1
|       |               |       |       daca (cond(k))
|       |               |       |           |       k ← k++
|       |               |       |       altfel
|       |               |       |           |       |x[k] ← 0
|       |               |       |           |       |       k ← k-1
|       |               |       |           |       |       ____#
|       |               |       |       ____#
|       |               |       ____#
|       |       ____#
|       ____#

```

```

<3> Void main()
|   scrie "n="
|   citește n
|   pentru i=1,n
|       pentru j=1,n
|           a[i][j]←0
|   scrie "m="
|   citește m
|   pentru j=1,m
|       citește b
|       citește y
|       a[b][y]←a[y][b]←1
|   _____#
|   x[n+1]=1
|   back()
|   _____#

```

<3> In functia main() se citesc datele principale, n este numarul de noduri ale grafului, m este numarul de muchii ce se vor forma. Pentru fiecare muchie formam matricea de adiacenta inscriind 1 la intersectia coordonatelor b si y. In fapt definim daca exista drum de la b la y. Matricea astfel formata este una simetrica. Initializam x[n+1] cu 1 pentru ca drumul comis voiajorului va pleca din 1 si se va intoarce tot aici.. Apelam functia care foloseste metoda backtracking ->back().

<2> In functia back() prin apeluri recursive incercam sa formam drumurile posibile. Tot aici se apeleaza si functia cond care retine conditiile de continuare. Ea returneaza o variabila ok care daca este 0 k se decrementeaza si se cauta alta solutie pentru satisfacerea cerintei.

Intai se initializeaza intr-un for vectorul x cu 0 pentru a nu avea surprize de la solutiile precedente, k primeste valoarea 2 pentru ca vom cauta un al doilea element iar x[1] devine 1, el fiind si punctul de plecare. Retinem insa ca din functia main avem si al n+1-elea element din vector tot pe 1 initializat.

Cat timp k>1 testam prima data daca am ajuns la final(k=n), caz in care se vor afisa valorile vectorului, valori care reprezinta una din solutiile grafului. In caz contrar verificam daca mai avem noduri nevizitate, daca da, incrementam pe x[k] si intram in functia cond. Daca valoarea returnata de functia cond este una pozitiva k va fi la randul lui incrementat si trecem pe alta pozitie a vectorului x[k] ce trebuie completat, altfel x[k] primeste valoarea 0 si k este decrementat.

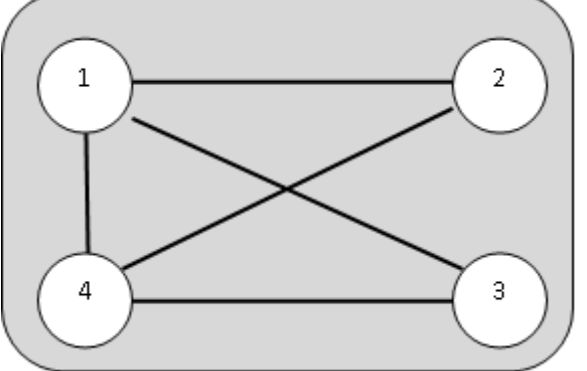
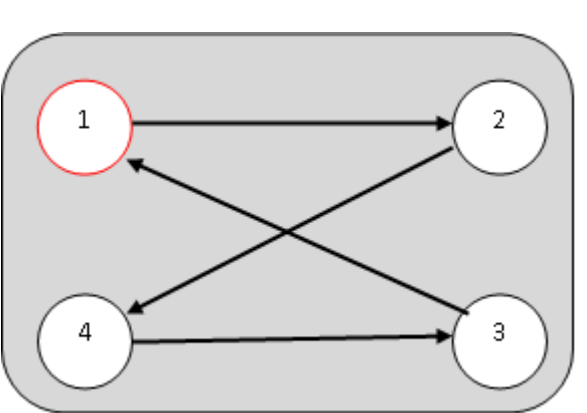
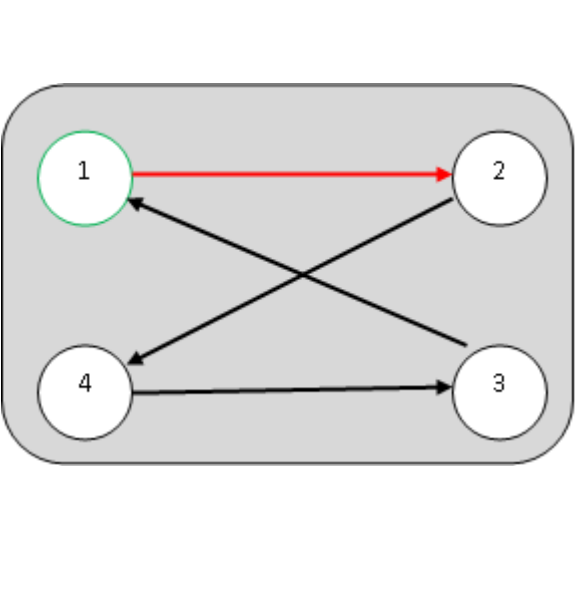
<1> Functia cond() este compusa din 3 „if”-uri si reprezinta conditiile de continuare. In prima faza variabila care va fi returnata este pusa pe 1, ea putand lua valoarea 0 in cazul in care una din cele 3 conditii este indeplinita. In cazul in care returneaza 0 nu avem conditii de continuare.

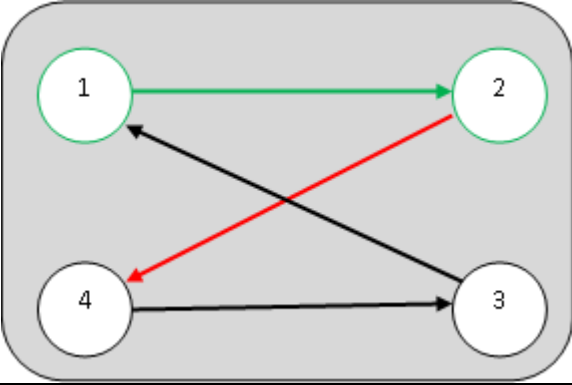
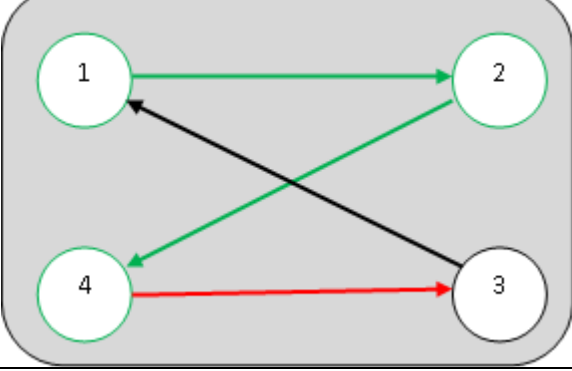
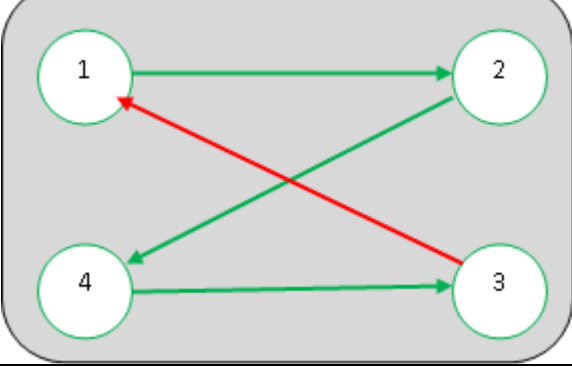
Prima conditie face referire la testarea vecinatatii oraselor selectate existand posibilitatea de a nu avea muchie intre ele.

Urmatoarea conditie verifica daca primul nod este diferit de ultimul.

Ultima conditie din functie testeaza daca nu a mai fost vizitat orasul selectat.

1.3 Exemplu detaliat

Explicatii, date	Felul cum se completeaza graful.
Input: $n=4$ $m=5$ $b=1,2,4,3,1;$ $y=2,4,3,1,4;$ Dupa care se intra in fct back Graful creat cu datele de mai sus arata in felul urmator:	
Pas 1: se initializeaza X la 0	Nu avem graf
Pas 2. $X=0,1,0,0,0,0,0,0$ Vectorul arata ca se pleaca din 1, se va ajung la sfarsit in 1 si ne positionam pe primul element cautand un succesor.	
Pas 3: $x=0,1,2,0,0,0,0,0$ Marcam primul nod ca fiind vizitat si ne indreptam spre nodul 2 pe care il adaugam in vector.	

Pas 4: $x=0,1,2,4,0,0,0,0$ Singurul vecin a lui 2 spre care avem legatura este cel pe care il adaugam in vector la acest moment. Avem nodurile 1,2 vizitate.	
Pas 5: $x=0,1,2,4,3,1,0,0$ Marcam nodul 4 ca vizitat si adaugam in vector nodul 3	
Pas 6: $x=0,1,2,4,3,1,0,0$ Ne intoarcel in 1, incheiem ciclul din care am plecat, nodul 3 devine vizitat iar graful a fost complet parcurs.	
Pas 7: se tipareste vectorul x cu valorile existente	Valorile lui x sunt 1,2,4,3,1.

1.4 Complexitate

Toti algoritmi de calcul pentru problema comis voiajorului sunt exponenciali.

Complexitatea de calcul al algoritmului este patratica. Totusi pentru un volum mare de date rezolvarea prin backtracking devine intructabila. Solutiile obtinute depasesc cu cel mult 7-25% lungimea drumului optim dupa diversi autori.

Algoritmul backtracking pentru problema comis-voiajorului are o punere în aplicare mai complexa, deoarece calculeaza practic toate posibilitățile de recurență

2. Metoda Greedy

2.1 Considerente teoretice

În cazul metodei (strategiei) greedy apare suplimentar ideea de a efectua în acel moment o alegere. Dintre toate nodurile următoare posibile de a fi vizitate sau dintre toți pașii următori posibili, se alege acel nod sau pas care asigură un maximum de “câștig”, de unde și numele metodei: *greedy* = *lacom*. Evident că în acest fel poate să scadă viteza de vizitare a nodurilor – adică a soluțiilor ipotetice sau a soluțiilor parțiale – prin adăugarea duratei de execuție a subalgoritmului de alegere a următorului nod după fiecare vizitare a unui nod. Există însă numeroși algoritmi de tip greedy veritabili care nu conțin subalgoritmi de alegere. Asta nu înseamnă că au renunțat la alegerea greedy ci, datorită “scurtăturii” descoperite în timpul etapei de analiză a problemei, acei algoritmi efectuează la fiecare pas o alegere fără efort și în mod optim a pasului (nodului) următor. Această alegere, dedusă în etapa de analiză, conduce la maximum de “profit” pentru fiecare pas și scurtează la maximum drumul către soluția căutată.

Dezavantajul acestei metode este că, la majoritatea problemelor dificile, etapa de analiză nu poate oferi o astfel de “pistă optimă” către soluție. Un alt dezavantaj al acestei strategii este că nu poate să conducă către toate soluțiile posibile ci doar către soluția optimă (din punct de vedere al alegerii efectuate în timpul căutării soluției), dar poate oferi toate soluțiile optime echivalente.

Conform strategiei greedy, vom construi ciclul pas cu pas, adaugând la fiecare iteratie cea mai scurta muchie disponibila cu urmatoarele proprietati:

- ❖ nu formeaza un ciclu cu muchiile deja selectate (exceptand pentru ultima muchie aleasa, care completeaza ciclul)
- ❖ nu exista inca doua muchii deja selectate, astfel incat cele trei muchii sa fie incidente in acelasi varf

La:	2	3	4	5	6
De la:					
1	3	10	11	7	25
2		6	12	8	26
3			9	4	20
4				5	15
5					18

Tabelul 6.4 Matricea distantelor pentru problema comis-voiajorului.

De exemplu, pentru sase orase a caror matrice a distantelor este data in Tabelul 6.4, muchiile se aleg in ordinea: {1, 2}, {3, 5}, {4, 5}, {2, 3}, {4, 6}, {1, 6} si se obtine ciclul (1, 2, 3, 5, 4, 6, 1) de lungime 58. Algoritmul greedy nu a gasit ciclul optim, deoarece ciclul (1, 2, 3, 6, 4, 5, 1) are lungimea 56.

2.2 Pseudocod

Algoritmul greedy construiește o singură cale începând de la un nod dat.

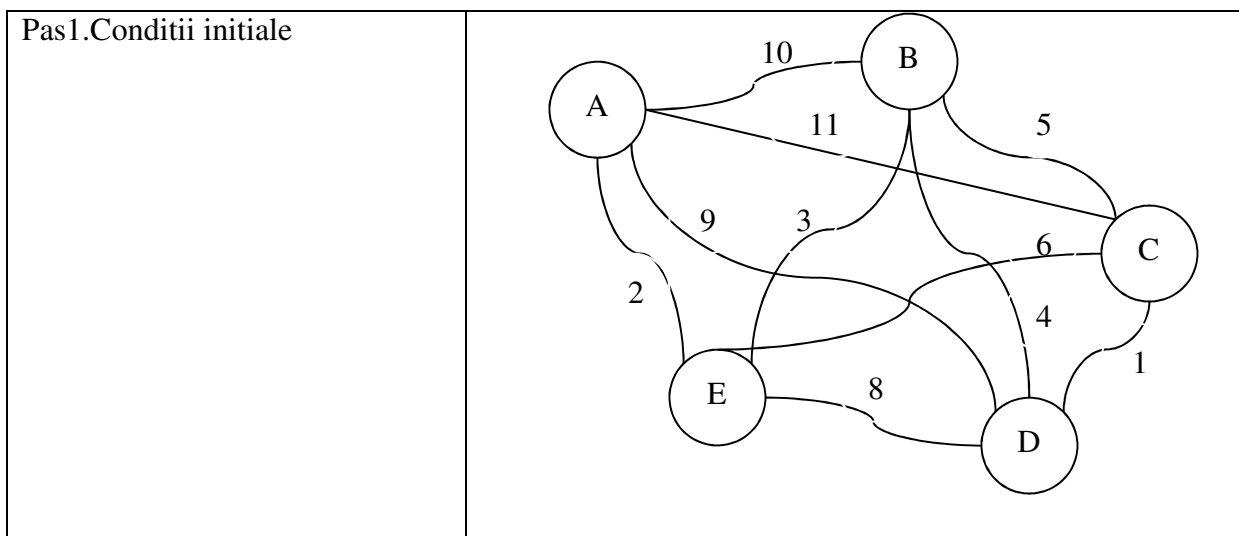
În ambele abordări lacomă și euristică noi trebuie să utilizăm o procedură esențială care alege ceea ce adăugăm la calea curentă. Este nevoie de trei parametri:

- ultima - întreg, care ne informează în cazul în care începem de la marginea
- min - pointer întreg - puncte de la costul de marginea adăugată prin procedură
- j_min - pointer întreg - puncte la indicele de nod sfârșitul marginei adăugate

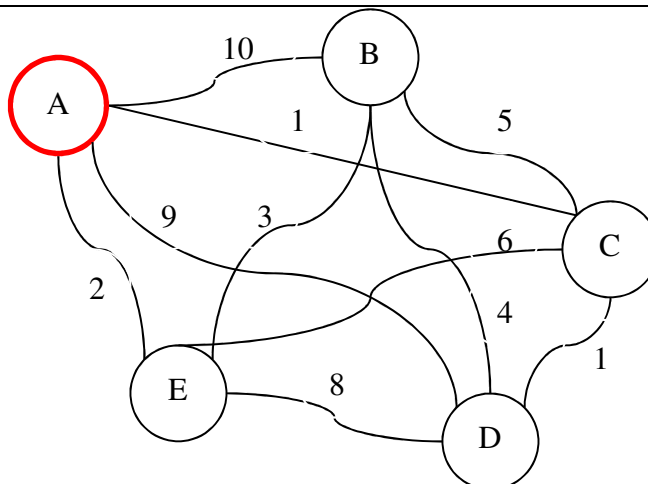
```
procedure choose(integer last, integer ptr min, integer ptr j_min )
  for j = 0 to n
    if node j is not visited
      if distance[last,j]<*min
        *min=distance[last,j]
        *j_min=j
  #
#
```

Această procedură este utilizată pentru a genera calea necesară. La fiecare pas se adaugă un alt nod pe nod vizitat ultima dată, și dacă sau nu un nod este deja în această cale.

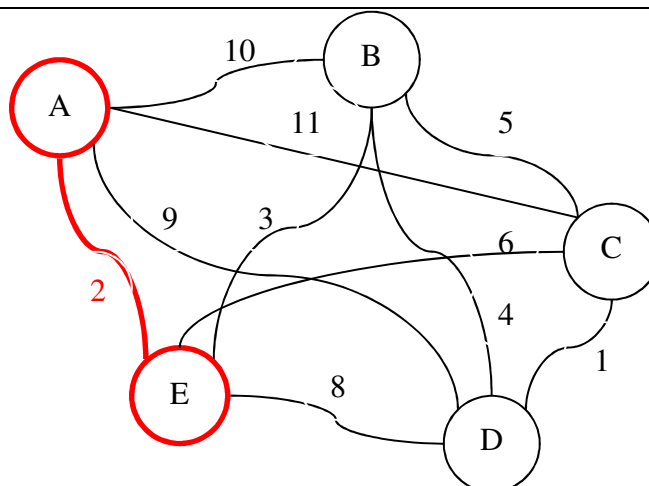
2.3 Exemplu detaliat



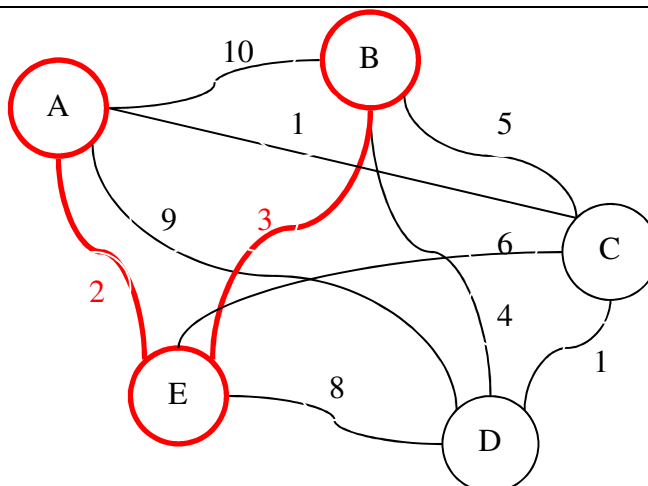
Pas2.Am ales pentru prima nodul A (am putea alege orice alt nod dacă nusuntem în mod special instruit altfel)



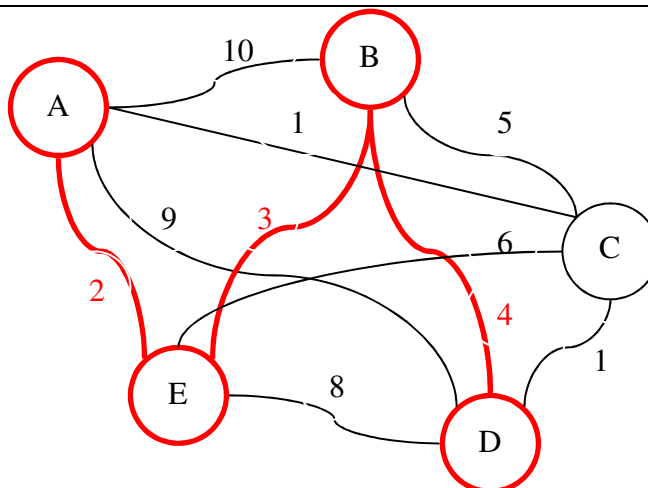
Pas3.Cea mai scurtă distanță de la A este de 2 (marginea A-E)



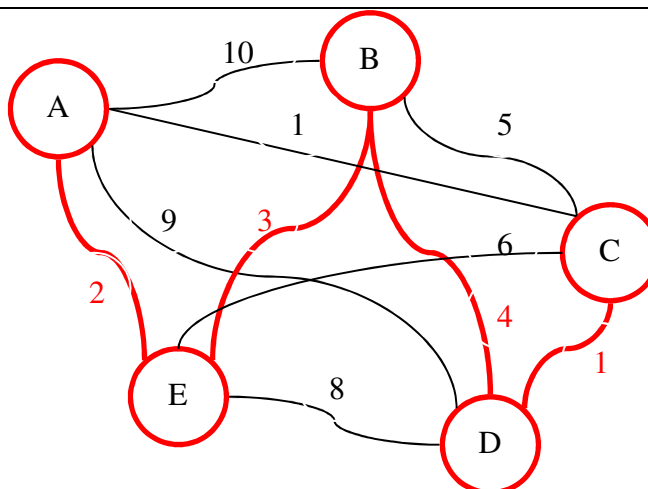
Pas4.Cea mai scurtă distanță de la E este 3 (marginea E-B)



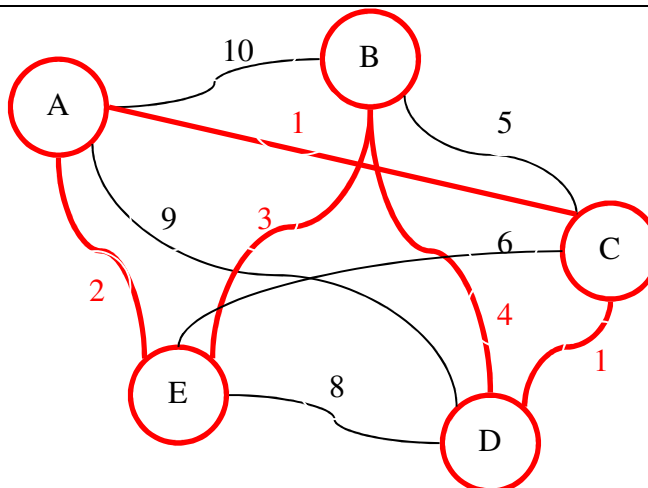
Pas5.Cea mai scurtă distanță de la B este 4 (marginea B-D)



Pas6.Cea mai scurtă distanță de la D este de 1 (marginea D-C)



Pas7.Adauga restul de marginea C-O lungime de 11



2.4 Complexitate

În teoria complexității, versiunea finală de PCV aparține clasei de probleme NP-complete. Astfel, se presupune că nu există nici un algoritm eficient pentru rezolvarea PCV. Cu alte cuvinte, este posibil ca în cel mai rău caz de rulare în ceea ce privește timpul pentru orice algoritm, pentru PCV crește exponențial cu numărul de orașe.

Soluția cea mai directă ar fi să încercați toate permutări (combinatii ordonate) și să vedeți care este mai eficient (utilizând funcția de căutare brute force). Timpul de rulare pentru această abordare se află într-un factor polinomial $O(n!)$, factorialul numărului de orașe, astfel încât această soluție devine imposibilă chiar și pentru numai 20 de orașe. Una dintre primele aplicații de programare dinamică este un algoritm care rezolvă problema în timp $O(n^2 2^n)$.

3. Metoda Euristica

3.1 Algoritmi euristici

În situațiile în care pentru anumite probleme complexe, pentru a căror rezolvare nu se cunosc algoritmi, sau aceștia sunt ineficienți (timp, memorie, cost), se preferă utilizarea unor algoritmi care rezolvă problema data mult mai rapid, cu efort mic, însă nu ne va furniza întotdeauna cea mai bună soluție, ci doar soluții acceptabile, adică soluții corecte care pot fi eventual îmbunătățite.

Prin *algoritmuristic* vom înțelege un algoritm care furnizează soluții *bune*, nu neapărat optime, care poate fi implementat rapid și furnizează rezultate în timp util.

O idee frecvent utilizată constă în descompunerea procesului de determinare a soluției în mai multe etape successive pentru care se poate determina *optimumul local*.

Optimumul global nu poate fi obținut întotdeauna prin determinări successive ale optimumului local, deci nu se poate aplica metoda Greedy, doar o strategie de tipuristic.

În practică se găsesc de multe ori soluții aproximative, mai ales dacă algoritmul se folosește de puține ori și efortul de determinare al soluției optime este mai mare decât beneficiul obținut.

O metodauristica poate să împartă rezolvarea în mai multe etape, se determină optimal din fiecare etapă (până în acel moment) urmărind prin aceasta ca rezultatul final să fie cât mai bun. Dacă este posibil să determinăm rapid mai multe soluții, atunci rezultatul dat va fi cel mai bun dintre acestea.

La scrierea unui algoritmururistic vom aplica doar condițiile simple dar necesare pentru o soluție corectă, care eventual va mai fi îmbunătățită.

•**Strategia:** se alege întotdeauna cea mai apropiată localitate (dintre cele nevizitate).

•**Îmbunătățire:** deoarece un traseu este un circuit închis, putem considera ca punct de plecare oricare dintre localități (traseul optim nu depinde de localitatea de start). Pentru fiecare localitate considerată ca localitate de plecare, se **determină traseul preferat**, apoi dintre aceste **k trasee corecte** se determină traseul cel mai bun (*de lungime minimă*) dacă rezultat final (care nu este neapărat **traseul optim**). Evident că traseul final va fi refăcut astfel încât să aibă ca localitate de

start *localitatea de domiciliu* a comis voiajorului (în exemplul dat am considerat toate cele n variante, deci $k = n$).

3.2 Pseudocod

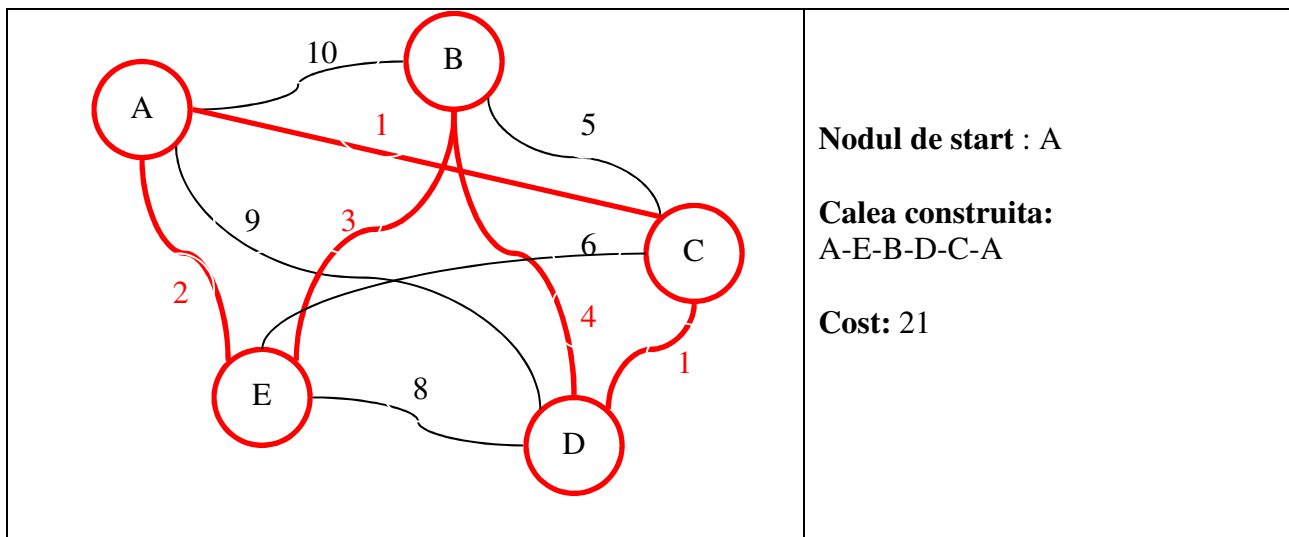
Algoritmul euristic construiește n trasee (n = numărul de noduri), și îl păstrează pe cel care oferă rezultatul optim, dar cu un cost de complexitate.

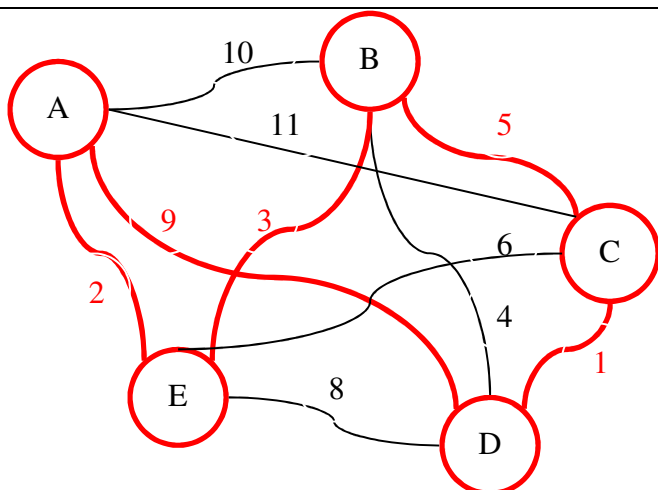
```

for nodes = 1 to n
|   call greedy subroutine
|   if current_cost > best_cost
|       |   best_cost = current_cost
|       |   save current_path
|       |   _____#
|   _____#

```

3.3 Exemplu detaliat

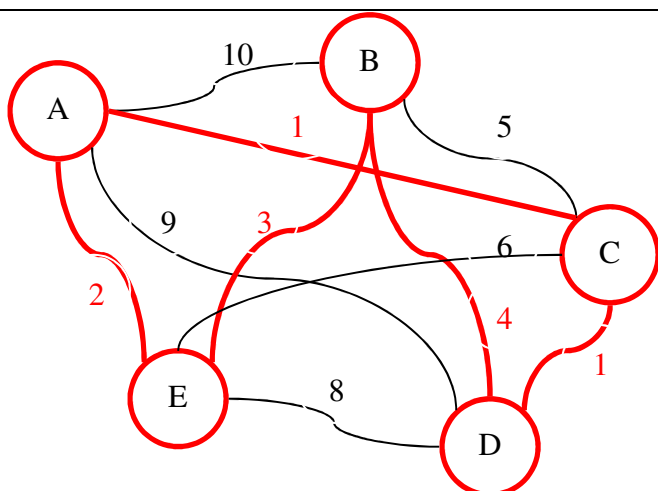




Nodul de start : B

Calea construita:
B-E-A-D-C-B

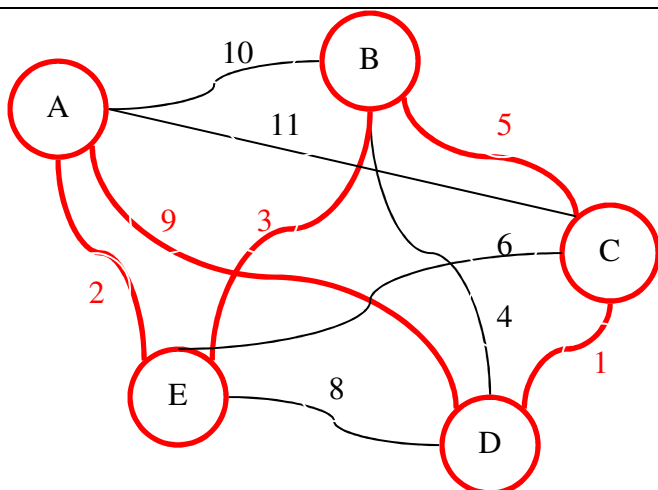
Cost: 20



Nodul de start : C

Calea construita:
C-D-B-E-A-C

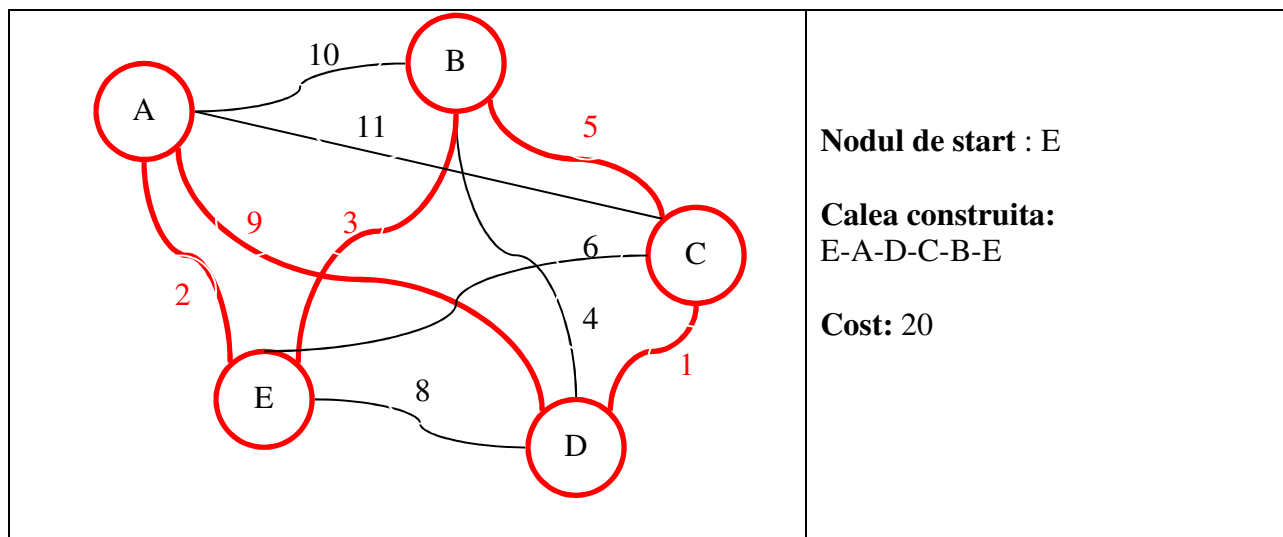
Cost: 21



Nodul de start : D

Calea construita:
D-C-B-E-A-D

Cost: 20



Cel mai bun rezultat obținut pornind de la nodul B (de asemenea, de la nodurile D și E, dar primul nod care conduce la un rezultat optim este stocat).

3.4 Complexitate

Metoda euristica are același principiu ca și metoda Greedy dar în loc să pornească de la un nod dat și să construiască calea, începe de la fiecare dintre noduri și preia în cele din urmă cea mai bună soluție.

//IMPLEMENTARE BACKTRACKING

```
#include<conio.h>
#include<stdio.h>
int x[10],a[10][10],n,m,i,j,k,ok,b,y;
int cond(int k) //Functia testeaza conditia de oprire
{ ok=1; //Consideram initial ok=1 si daca este cazul il modificam ulterior
  if(a[x[k]][x[k-1]]==0) //testeaza daca 2 orase selectate sunt vecine
ok=0;
  if((k==n)&&(a[x[k]][x[1]]==0)) //Testez daca primul oras este diferit de ultimul
    ok=0;
  for(i=1;i<k;i++)
    if(x[k]==x[i]) //testez sa nu mai fi vizitat orasul
      ok=0;
  return (ok); //returnez pe k pentru a putea fi testat in fct back
}
void back()
{ for(i=1;i<=n;i++)
  x[i]=0;
  k=2;
  x[1]=1;
  while(k>1)
    if(k==n+1) // daca am vizitat toate nodurile afisez vectorul
    {
for(i=1;i<=n+1;i++) // parcurg vectorul solutie si il afisez
  printf("%d", x[i]);
  printf("\n");
  k--;
    }
    else
if(x[k]<n) //daca x[k]<n inseamna ca mai sunt noduri nevizitate un
  //merita testate conditiile de continuare
  {
    x[k]=x[k]+1;
    if(cond(k)) //interoghez functia cond pentru a vedea daca mai am
      //solutii
      k++; //daca am solutii il incrementez pe k pentru a cauta
// mai eparte
  }
  else //altfel decrementez pe k
  {
    x[k]=0;
    k--;
  }
}
void main()
{
  printf("n="); // citesc numarul de puncte din graf
```



```

scanf("%d",&n);
//initializam matricea
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
a[i][j]=0;
printf("m="); //citesc numarul de muchii din graf
scanf("%d",&m);
for(j=1;j<=m;j++) // marcheaz in matrice arcele grafului
{
printf("x=");
scanf("%d",&b);
printf("y=");
scanf("%d",&y);
a[b][y]=1;
a[y][b]=1;
}
x[n+1]=1; //initializez ultimul element cu 1
back();
getch();
}

```

//IMPLEMENTARE GREEDY

```

#include <stdio.h>
#define MAXNUM 30 /* Maximum number of points */
#define MINIMUM 10000 /* Initial value when searching for minimum */
int n; /* Number of points */
int d[MAXNUM][MAXNUM]; /* Distance matrix */
int path[MAXNUM]; /* Path of the traveling salesman; contains the indexes of the visited points*/
int visited[MAXNUM]; /* Array that contains information about what points have been visited*/

/*Function that chooses which point will be visited next */
void choose(int last, int *min, int *j_min) {
    int j;
    /* The minimum distance to a point not yet visited is searched */
    *min = MINIMUM; *j_min = -1;
    for (j=0; j<n; j++)
        if (!visited[j]) {
            if (d[ last ][ j ] < *min) {
                *min = d[ last ][ j ];
                *j_min = j;} }
}

int main(void) {

    FILE *fin;

```

```

int i, j, index, count, cost, min, j_min, save_cost=MINIMUM, save_path[MAXNUM];
fin = fopen("euristic.txt", "rt"); if (!fin) {
    printf("ERROR: cannot open file.\n");
    return -1; }
fscanf(fin, "%d", &n); /* Read input from file */
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        fscanf( fin, "%d", &(d[ i ][ j ]));
printf("%d points.\n", n);
printf("Distances between points:\n");
for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
        printf("%d ", d[ i ][ j ]);
    printf("\n"); }
printf("\n");

/* First point visited is of index 0 */
for (index=0; index<n; index++){
    printf("We start at point %d\n", index+1);
    min=MINIMUM;
    j_min=MINIMUM;
    for (i=0; i<n; i++){
        visited[i] = 0; /* Initially no point is visited */
        path[i]=0;
    }
    path[0] = index; visited[index] = 1;
    count = 1; cost = 0;
    /* The rest of the points are visited */
    for (i=0; i<n-1; i++) {
        /* We choose next point to be visited */
        choose(path[count-1], &min, &j_min);
        printf("Connected (%d, %d) of cost %d.\n", path[count-1]+1, j_min+1,
min);

        path[count] = j_min;
        visited [j_min ] = 1;
        count++;
        cost += min;
    }
    /* We go through the path from the last visited point
    to the first one and we update the total cost*/
    cost += d[path[n-1]][index];
    /* Display chosen path */
    printf("\nPath has cost %d and is:\n", cost);
    for (i=0; i<n; i++)
        printf("%d ", path[i]+1);
    printf("%d\n\n\n", index+1);
    if (cost<save_cost){

```

```

        for(i=0;i<n;i++)
            save_path[i]=path[i];
        save_cost=cost;
    }
}
/* Display chosen path */
printf("\nShortest path starts at point %d has cost %d and is:\n", save_path[0],save_cost);
for (i=0; i<n; i++)
    printf("%d ", save_path[i]+1);
printf("%d\n\n\n\n",save_path[0]);
return 0;
}

```