NUMERICAL RECIPES

Webnote No. 14, Rev. 1

Code Implementation for the Traveling Salesman Problem

The following traveling salesman program, using the Metropolis algorithm, illustrates the main aspects of the simulated annealing technique for combinatorial problems.

```
struct Anneal {                                                    anneal.h
```
This algorithm finds the shortest round-trip path for a set of cities using simulated annealing.

```
    Ranq1 myran;
    Anneal() : myran(1234) {}
```
Constructor just initializes random number generator.

```
    void order(VecDoub_I &x, VecDoub_I &y, VecInt_IO &iorder)
```
This routine finds the shortest round-trip path to `ncity` cities whose coordinates are in the arrays `x[0..ncity-1],y[0..ncity-1]`. The array `iorder[0..ncity-1]` specifies the order in which the cities are visited. On input, the elements of `iorder` may be set to any permutation of the numbers `0` to `ncity-1`. This routine will return the best alternative path it can find.

```
    {
        const Doub TFACTR=0.9;          Annealing schedule: reduce t by this factor on each
                                            step.
        Bool ans;
        Int i,i1,i2,nn;
        VecInt n(6);
        Doub de,path=0.0,t=0.5;         0.5 is the initial temperature.
        Int ncity=x.size();
        Int nover=100*ncity;            Maximum number of paths tried at any temperature.
        Int nlimit=10*ncity;
```
Maximum number of successful path changes before continuing.

```
        for (i=0;i<ncity-1;i++) {       Calculate initial path length.
            i1=iorder[i];
            i2=iorder[i+1];
            path += alen(x[i1],x[i2],y[i1],y[i2]);
        }
        i1=iorder[ncity-1];
        i2=iorder[0];
        path += alen(x[i1],x[i2],y[i1],y[i2]);
        cout << fixed << setprecision(6);
        for (Int j=0;j<100;j++) {       Try up to 100 temperature steps.
            Int nsucc=0;
            for (Int k=0;k<nover;k++) {
                do {
                    n[0]=Int(ncity*myran.doub());          Choose beginning of segment
                    n[1]=Int((ncity-1)*myran.doub());      ..and. end of segment.
                    if (n[1] >= n[0]) ++n[1];
                    nn=(n[0]-n[1]+ncity-1) % ncity;        nn is the number of cities
                } while (nn<2);                                not on the segment.
```
Decide whether to do a segment reversal or transport:

```
                if (myran.doub() < 0.5) {          Do a transport.
                    n[2]=n[1]+Int(abs(nn-1)*myran.doub())+1;
```

```
            n[2] %= ncity;                  Transport to location not on the path.
            de=trncst(x,y,iorder,n);        Calculate cost.
            ans=metrop(de,t);               Consult the oracle.
            if (ans) {
                ++nsucc;
                path += de;
                trnspt(iorder,n);           Carry out the transport.
            }
        } else {                            Do a path reversal.
            de=revcst(x,y,iorder,n);        Calculate cost.
            ans=metrop(de,t);               Consult the oracle.
            if (ans) {
                ++nsucc;
                path += de;
                reverse(iorder,n);          Carry out the reversal.
            }
        }
        if (nsucc >= nlimit) break;         Finish early if we have enough successful changes.
    }
    cout << endl << "T = " << setw(12) << t;
    cout << "     Path Length = " << setw(12) << path << endl;
    cout << "Successful Moves: " << nsucc << endl;
    t *= TFACTR;                            Annealing schedule.
    if (nsucc == 0) return;                 If no success, we are done.
  }
}
```

`Doub revcst(VecDoub_I &x, VecDoub_I &y, VecInt_I &iorder, VecInt_IO &n)`
This function returns the value of the cost function for a proposed path reversal. `ncity` is the number of cities, and arrays `x[0..ncity-1],y[0..ncity-1]` give the coordinates of these cities. `iorder[0..ncity-1]` holds the present itinerary. On input, the first two values `n[0]` and `n[1]` of array `n` should be set to the starting and ending cities along the path segment to be reversed. On output, the function value returns the cost of making the reversal. The actual reversal is not performed by this routine.

```
{
    VecDoub xx(4),yy(4);
    Int ncity=x.size();
    n[2]=(n[0]+ncity-1) % ncity;            Find the city before n[0] ..
    n[3]=(n[1]+1) % ncity;                  .. and the city after n[1].
    for (Int j=0;j<4;j++) {
        Int ii=iorder[n[j]];                Find coordinates for the four cities involved.
        xx[j]=x[ii];
        yy[j]=y[ii];
    }
    Doub de = -alen(xx[0],xx[2],yy[0],yy[2]);   Calculate cost of disconnecting the segment at both ends and reconnecting in the opposite order.
    de -= alen(xx[1],xx[3],yy[1],yy[3]);
    de += alen(xx[0],xx[3],yy[0],yy[3]);
    de += alen(xx[1],xx[2],yy[1],yy[2]);
    return de;
}
```

`void reverse(VecInt_IO &iorder, VecInt_I &n)`
This routine performs a path segment reversal. `iorder[0..ncity-1]` is an input array giving the present itinerary. The vector `n` has as its first four elements the first and last cities `n[0],n[1]` of the path segment to be reversed, and the two cities `n[2]` and `n[3]` that immediately precede and follow this segment. `n[2]` and `n[3]` are found by function `revcst`. On output, `iorder[0..ncity-1]` contains the segment from `n[0]` to `n[1]` in reversed order.

```
{
    Int ncity=iorder.size();
    Int nn=(1+((n[1]-n[0]+ncity) % ncity))/2;   This many cities must be swapped to effect the reversal.
    for (Int j=0;j<nn;j++) {
        Int k=(n[0]+j) % ncity;             Start at the ends of the segment and swap pairs of cities, moving toward the center.
        Int l=(n[1]-j+ncity) % ncity;
```

```
        Int itmp=iorder[k];
        iorder[k]=iorder[l];
        iorder[l]=itmp;
    }
}
```

`Doub trncst(VecDoub_I &x, VecDoub_I &y, VecInt_I &iorder, VecInt_IO &n)`
This routine returns the value of the cost function for a proposed path segment transport. `ncity` is the number of cities, and arrays `x[0..ncity-1]` and `y[0..ncity-1]` give the city coordinates. `iorder[0..ncity-1]` is an array giving the present itinerary. On input, the first three elements of array `n` should be set to the starting and ending cities of the path to be transported, and the point among the remaining cities after which it is to be inserted. On output, the function value returns the cost of the change. The actual transport is not performed by this routine.

```
{
    VecDoub xx(6),yy(6);
    Int ncity=x.size();
    n[3]=(n[2]+1) % ncity;                  Find the city following n[2]..
    n[4]=(n[0]+ncity-1) % ncity;            ..and the one preceding n[0]..
    n[5]=(n[1]+1) % ncity;                  ..and the one following n[1].
    for (Int j=0;j<6;j++) {
        Int ii=iorder[n[j]];                Determine coordinates for the six cities
        xx[j]=x[ii];                            involved.
        yy[j]=y[ii];
    }
    Doub de = -alen(xx[1],xx[5],yy[1],yy[5]); Calculate the cost of disconnecting
    de -= alen(xx[0],xx[4],yy[0],yy[4]);        the path segment from n[0] to
    de -= alen(xx[2],xx[3],yy[2],yy[3]);        n[1], opening a space between
    de += alen(xx[0],xx[2],yy[0],yy[2]);        n[2] and n[3], connecting the
    de += alen(xx[1],xx[3],yy[1],yy[3]);        segment in the space, and con-
    de += alen(xx[4],xx[5],yy[4],yy[5]);        necting n[4] to n[5].
    return de;
}
```

`void trnspt(VecInt_IO &iorder, VecInt_I &n)`
This routine does the actual path transport, once `metrop` has approved. `iorder[0..ncity-1]` is an input array giving the present itinerary. The array `n` has as its six elements the beginning `n[0]` and end `n[1]` of the path to be transported, the adjacent cities `n[2]` and `n[3]` between which the path is to be placed, and the cities `n[4]` and `n[5]` that precede and follow the path. `n[3]`, `n[4]`, and `n[5]` are calculated by function `trncst`. On output, `iorder` is modified to reflect the movement of the path segment.

```
{
    Int ncity=iorder.size();
    VecInt jorder(ncity);
    Int m1=(n[1]-n[0]+ncity) % ncity;       Find number of cities from n[0] to n[1]
    Int m2=(n[4]-n[3]+ncity) % ncity;       ...and the number from n[3] to n[4]
    Int m3=(n[2]-n[5]+ncity) % ncity;       ...and the number from n[5] to n[2].
    Int nn=0;
    for (Int j=0;j<=m1;j++) {
        Int jj=(j+n[0]) % ncity;            Copy the chosen segment.
        jorder[nn++]=iorder[jj];
    }
    for (Int j=0;j<=m2;j++) {               Then copy the segment from n[3] to
        Int jj=(j+n[3]) % ncity;                n[4].
        jorder[nn++]=iorder[jj];
    }
    for (Int j=0;j<=m3;j++) {               Finally, the segment from n[5] to n[2].
        Int jj=(j+n[5]) % ncity;
        jorder[nn++]=iorder[jj];
    }
    for (Int j=0;j<ncity;j++)               Copy jorder back into iorder.
        iorder[j]=jorder[j];
}
```

```
    Bool metrop(const Doub de, const Doub t)
    Metropolis algorithm. metrop returns a boolean variable that issues a verdict on whether
    to accept a reconfiguration that leads to a change de in the objective function e. If de<0,
    metrop = true, while if de>0, metrop is only true with probability exp(-de/t), where t
    is a temperature determined by the annealing schedule.
    {
        return de < 0.0 || myran.doub() < exp(-de/t);
    }

    inline Doub alen(const Doub a, const Doub b, const Doub c, const Doub d)
    {
        return sqrt((b-a)*(b-a)+(d-c)*(d-c));
    }
};
```