Reversible-Segment List

László Hárs*

September 13, 1989

Keywords. Data structures, Design of algorithms

1. Introduction

In the simulated annealing heuristic of the traveling salesman problem we choose randomly 2 edges of an already constructed tour i.e., of a Hamiltonian circuit of a given graph with associated length to the edges, and try to exchange them with the edges connecting the first endpoints and connecting the last endpoints of the chosen edges. If this reduces the total length of the tour, we perform this exchange, otherwise we perform this only with probability p. This p depends on the increase of the tour cost and the "temperature", which decreases in the course of the algorithm, according to the applied annealing schedule.

Experiments show, (see [6]) that a reasonable good implementation of this simulated annealing algorithm performs $O(n)$ choosings of edge pairs and $O(n)$ rearrangements of the tour at any temperature value. (Then everything is repeated with smaller and smaller temperatures, i.e., accepting length-increasing edge exchanges with smaller and smaller probabilities.) With suitable implementation this algorithm finds very good tours in acceptable running time. However, if the graph has thousands of nodes the running time gets too long. With our new data structure this can be reduced.

If the tour is stored in the simplest data structures, in an array, or in a linked list, a tour rearrangement takes $O(n)$ time in the average, so the simulated annealing procedure uses $O(n^2)$ time in each selected temperature.

If we use the well known *reversible list* data structure (see e.g., [7]), it takes constant time to reverse a segment of the list, but to find the next element of a randomly chosen element u, we may have to follow the whole list from a fixed element until u. (We know only the two neighbors of an element, but do not know which is the next. If we pick up the wrong one, the tour will fall apart into two circles.) In the average following the list from an element to another takes $O(n)$ time, so with this the simulated annealing

* University Bonn, Institute for Operations Research and
Eötvös Loránd University, Budapest, Department of Computer Science.

procedure takes $O(n^2)$ time in each selected temperature, too. This can be reduced to $O(n \log n)$ with the data structure presented here.

Our reversible-segment list data structure is a kind of balanced tree. We are free to choose the height of this tree, by that we can control the trade off between the cost of finding the next element of the list and the cost of reversing a segment of it. For example, with height 2, the next element of the list can be obtained in constant time (about four times as much as in the simple linked list) and reversing a segment takes $O(\sqrt{n})$ time in the worst case. This is the most promising special case in the practice, since it is very simple, and the algorithms tend to use much more next-element queries than reversions, so the next-element time gives usually the largest part of the total running time. (See [3].)

Another interesting special case is of height $\log n$, when both the next-element query and the segment reversion take $O(\log n)$ time. If we have much more next-element queries than reversions, a simple modification of this data structure reduces the average time for the next-element queries to $O(1)$.

We present the details of the data structures and the algorithms operating on them in PASCAL programing language. All the codes in the Appendices are parts of fully functional programs which were tested and run on several test problems.

2. The "Boss-list"

n elements are given to store. They can be coded by the integers $1 \ldots n$. We store them in an ordinary double-linked list, with links $nbor_0$, $nbor_1$, together with the information related to the node, if there is any. (See e.g., [7].)

Let us break the list into *sections* of length between d and $2d-1$, with some fixed constant d. The list elements in a section get a *boss*, an element of another double linked list. The boss has a field *O1* telling which links of all of its *subordinates* point to the next, and which links point to the previous (subordinate) elements of the first list. (Changing the value of this field results in changing the direction of the whole section.)

It is convenient to store in a boss some more fields: *len* the actual length of the corresponding section of the subordinates, *first, last* the first and last subordinate of the section. (See Figure 1.)

The next-element (and the previous-element) operations are easy at any subordinate: we ask the boss which link points to the next subordinate.

In a 2-opt type improving step (see e.g., [2] or [5]) we have to reverse a segment. It should be first cut off from the list, and after reversing it should be joined back to the rest of the list.

Cutting the list after the element u means to do nothing if u is the last element of its section. Otherwise we split the boss. The new one has the subordinates preceding u and u, making the element u to be the last of the new section. (We may make two sections shorter than d.)

The reversion requires now only to change the $O1$ values at the bosses of the cut off segment.

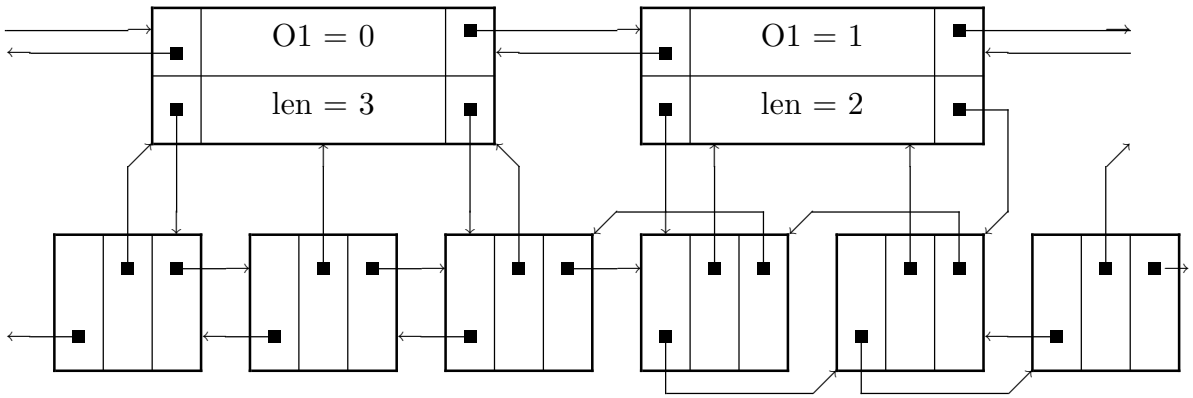

Figure 1.

After reversion the list must be restored, i.e., the pieces joined together. This requires only to change the pointers at both sides of the reversed segment.

If a boss now has too few subordinates, we dispose it, joining its subordinates to the next boss. The resulting new section may become too large, when we split it into two. The parts then are of correct size.

In a 2-opt type edge exchange procedure (*XCON*) for the traveling salesman problem we do not need anything else: next-elements to find the edges in correct direction and reversion of a segment, with rebalancing.

Theorem 1. *Finding the next (or previous) element in a "boss-list" takes constant time.*

Proof. Immediate. ■

Theorem 2. *Reversion of a segment of a "boss-list" takes $O(\sqrt{n})$ worst case computation time, if we take $d = \sqrt{n/2}$.*

Proof. To cut off the segment from the list can take $O(d)$ time, since we may have to change the *boss* pointer at all but one element in a section (of length less than $2d$), and to count them. The number of other operations is constant.

The actual reversion requires to follow a segment of the list of bosses. Its length is at most $O(n/d) = O(\sqrt{n})$. At each boss we do constant number of operations.

To join back to the list the cut off segment takes constant number of steps, since we have to modify only the ends of the segments.

Rebalancing the boss-list requires at most 4 *Meld* and *Split* operations on one or two bosses each. Creating, disposing, inserting, deleting bosses requires only constant number of operations each. The total number of subordinates involved is $O(d)$, and changing the boss pointer, exchanging the links if necessary at each subordinate takes $O(1)$ steps. ■

3. The "Boss-tree"

In an h level tree we can keep the idea to store the direction information at "boss nodes", like in the boss-list. Our tree is balanced, and similar to the *2–3 trees* (see e.g., [1]). Again, with the fixed constant d, we divide the list of elements into sections of length between d and $2d-1$. In the boss-tree the bosses are also joined into sections of the same size limits. The bosses of bosses form sections, too, up to level h. Now on the same level the consecutive vertices of the tree are not linked together, instead their bosses have pointers to them stored in arrays *Sub* of length *len*. At the subordinates we store the index *indx*, i.e., the location in the boss' array of the pointers which point to this subordinate.

There is a topmost boss, the *God*, where the pointers in the array *Sub* are thought to form a circular list i.e., Sub_1 follows Sub_{len}. On the other hand, the bottommost elements of the tree have no subordinates, and their *len* fields contain the subscripts of the original elements (tour-nodes) stored in the array *Node*. In that we store pointers to the corresponding tree leaves. Here we can also store information related to the nodes, e.g., their coordinates in the plane. (See Figure 2.)

We call the original elements given to store *nodes*, the vertices of the boss-tree *vertices*.

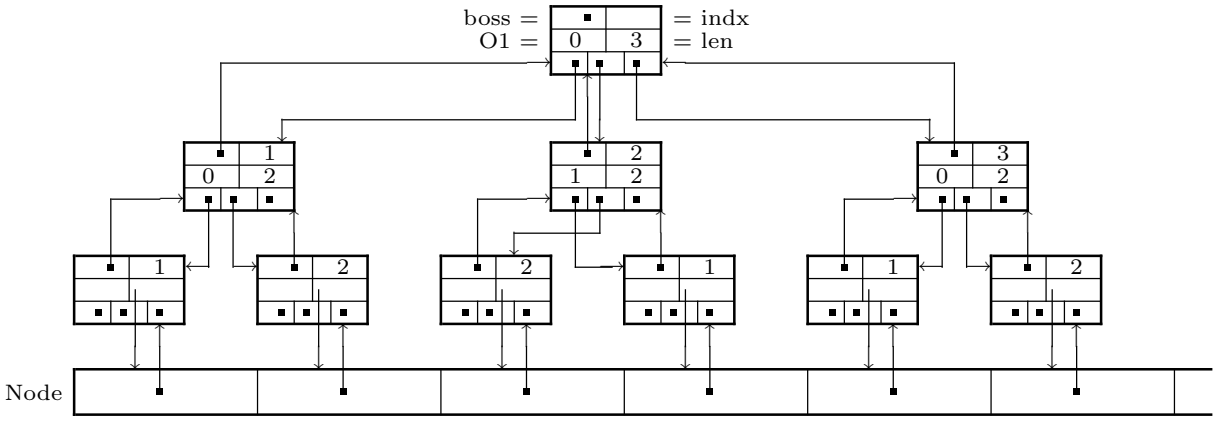

Figure 2.

(For test, as in Appendix 2, we can put, e.g., $n = 24$, $Height = 4$ and $d = 2$, allowing sections of length 2 and 3 only, except if in a level there are only a few (at most 4) vertices. They can have length 1, too.)

To find the next element of a given one, first the real directions, i.e., the order of the pointers in the arrays *Sub* have to be determined at each boss on the path up to the *God*. For this, we follow the *boss* pointers up to the topmost level, then turn back. The value *O1* or its reverse gives the real direction of the vertices, which have boss of real direction 0 or 1, respectively. The result is stored in an array. (Its element i is 1, iff the subordinate pointers of the corresponding level i boss are in reverse order in the array *Sub*).

To find the next vertex in any level, we look at the boss. If the index of our vertex is the last (it means the physical first one if the subordinate pointers are in reverse

order), we have to follow the *boss* pointers until a boss is found where the index is not the last. (There is no last subordinate of the *God*.) Then from the next subordinate we go down following the leftmost pointers, until we reach the starting level. This procedure actually finds the next vertices of other bosses, too, on the beginning of the upward path from our starting vertex. We can call the same procedure again at the first boss on the upward path which has got no next vertex determined. It results in finding the next vertices of all the bosses on the path going up from a starting vertex. These next vertices can be stored in an array. With little extra work the real directions of these next vertices are also calculated and stored.

To reverse a segment, again, first we cut off at each level the subtree of this segment from the rest of the tree i.e., if vertex u and the next one v on this level have a common boss, split it into two. The first one has the subordinates of the original boss that precede the vertex u, and u itself, while the second one has v and the vertices that follow v. Then repeat these on upper levels, until the top of the tree.

After we cut off the subtree of the node-segment to be reversed, we change the *O1* values of the topmost bosses of the subtree, and reverse their (circular) sequence.

We have to rebalance the tree, since on both sides of the two cut-paths there can be short sections of vertices. If vertex u has fewer then d subordinates, and if the section of the subordinates of the next vertex v is long enough, we move the first vertices of v to the end of the subordinates of u. Otherwise we join the subordinates of u to the beginning of the section of subordinates of v and delete u. (The deletion affects the length of the boss of u, therefore the balancing should be done from bottom up. The boss of v would be balanced later anyway.)

The vertices next to the bosses on the second paths of the two cuts and the vertices of the 2–2 paths of the cuts are the only ones involved in the balancing. We have to be careful, since some of these six vertices in the actual level can coincide. We perform that balance first, which does not delete a vertex of a later balance. It is not possible, if this precedence form a cycle, but if this happens, the number of the vertices at this level is at most 4. In this case we do nothing, accept deleting the bosses with no subordinates. On this level there can remain at most 4 vertices with too few subordinates, what does not affect the running time nor the validity of our algorithms. (Later this level may get more vertices, but if their number is larger then 4, no new short sections appear.)

Lemma. *Finding the next (or previous) vertices of all the vertices of an upward path in a boss-tree takes $O(h)$ steps.*

Proof. To determine the real directions on the upward path we follow this path up to the topmost level, and come back. This is done only once.

Then we go up to a vertex which is not the last subordinate of its boss, take the next subordinate, and go down to the starting level always on the leftmost subordinates. At each step we determine the real direction of the last vertex (which is next to some vertex on the upward path) by looking at only its boss' real direction and its *O1* value. The work is proportional to the length of the path we use coming down, and the number of the found next vertices is equal to this length. ■

Theorem 3. *Finding the next (or previous) node of a given node of the tour takes*

$O(h)$ computation time, using the boss-tree.

Proof. Consequence of the Lemma. ■

Theorem 4. *Reversion of a segment of nodes in a boss-tree takes $O(dh + n/d^{h-1})$ worst case computation time.*

Proof. To cut off the segment from the list can take $O(d)$ time at each level, (going from bottom up), all together $O(dh)$.

On the bottommost level we have all the n elements. Then, on the second level at most $O(n/d), \ldots, O(n/d^{h-1})$ on the topmost level. Changing the *O1* values and reversing a segment of the subordinates here takes also $O(n/d^{h-1})$ steps in the worst case.

We have to rebalance the split sections. All together there can be $6h$ of them involved in local balances. Each section is shorter than $2d$, and each of their vertices is touched by the algorithm constant times, therefore the total number of steps is at most $O(dh)$. ■

Corollary. *With constant d and $h = \log_d n$ the boss-tree supports the* next-element *and* segment-reversion *operations in $O(\log n)$ steps each.*

4. Speed up the next-element queries

We can store at each node the subscript (relative to the array *Node*) of its lastly found next-node together with the version number of the tree. This version number is incremented after each segment reversion, showing that next-subscripts are no longer valid. Also, when the real direction of a boss is determined, we can store it at that boss together with the version number of the tree.

Theorem 5. *With these modifications of the boss-tree m consecutive next-element queries can be answered in $O(m + n)$ time.*

It means, that the average cost of $O(n)$ consecutive next-element queries is $O(1)$.

Proof. We employ a simplified version of the procedure *Next*: Remove the repeated call of itself which would determine the next vertices of all the ones of the upward path. This procedure is used to answer a next-element query of a node at most once. Each call takes time proportional to the visited tree edges. Each tree edge is used in both directions at most once by calls of procedure *Next*, and once when determining the real directions. (If we reach a boss with already determined real direction, i.e., if the stored version number agrees with the version number of the tree, we can turn back there.) The total time is therefore proportional to the number of tree edges, to $O(n)$. ■

5. Remarks

We did not mention how to build up initially the boss-list or the boss-tree. It is trivial, if we join elements, bosses to sections of size d from an arbitrary start. The last section may have size up to $2d-1$.

These data structures also support in the obvious way insertion and deletion, which may be useful in tour building procedures for the traveling salesman problem.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ulman "The design and analysis of computer algorithms," *Addison-Wesley, Reading, MA* **1974**.

[2] B. L. Golden and W. R. Stewart "Empirical analysis of heuristics," in The Travelling Salesman Problem *edited by* E. L. Lawler, J. K. Lenstra, A. H. G. Rinnoy Kan and D. B. Shmoys *John Wiley & Sons, Chichester*, **1986**, pp. 207–249.

[3] L. Hárs "Random search in the traveling salesman problem," *(Manuscript)* **1989**.

[4] S. Lin and B. Kernighan "An effective heuristic algorithm for the traveling salesman problem," *Operations Research* **21**(1973), pp. 498–516.

[5] C. H. Papadimitriou, K. Steiglitz "Combinatorial Optimization: Algorithms and Complexity," *Prentice-Hall* **1982**.

[6] Y. Rossier, M. Troyon, Th. M. Liebling "Probabilistic exchange algorithms and Euclidean traveling salesman problems," *OR Spectrum* **8**(1986), pp. 151–164.

[7] R. E. Tarjan "Data structures and network algorithms," *CBMS-NSF Regional Conference Series in Applied Mathematics* **44**(1983).

Appendix 1. (Boss-list)

```
CONST
      MaxN          = 10000;
TYPE
      bit           = 0..1;
      NodeRange     = 1..MaxN;
      Nbors         = Array[bit] of NodeRange;
      BossPointer   = ->BossType;
      BossType      = RECORD  first, last : NodeRange;
                              O1: bit;
                              len: Integer;
                              next, prev : BossPointer;
                      END;
      NodeType      = RECORD  nbor : Nbors;
                              x, y : Integer;           {Or other info}
                              boss : BossPointer;
                      END;
VAR
      Node          : Array[NodeRange] of NodeType;
      n,                                    {Number of nodes in list   }
      MaxSec,MinSec: Integer;               {SectionSize limits: d,2d-1}
```

```
Function NXT( u: NodeRange): NodeRange;
Begin  With Node[u] Do  NXT := nbor[boss->.O1]  end;

Function PRV( u: NodeRange): NodeRange;
Begin  With Node[u] Do  PRV := nbor[1 - boss->.O1]  end;

Procedure Cut( u,v: NodeRange);            {Cut between u and v=NXT(u)}
Var  BossU, BossV: BossPointer;
     l : Integer;
Begin
  BossU := Node[u].boss;
  If BossU = Node[v].boss Then Begin
    New(BossV);
    BossV->.first := v;
    BossV->.last  := BossU->.last;
    BossU->.last  := u;
    BossV->.O1    := BossU->.O1;
    BossV->.next  := BossU->.next;
    BossV->.prev  := BossU;
    BossU->.next  := BossV;
    BossV->.next->.prev := BossV;

    l := 1;
    Node[v].boss  := BossV;
    While v <> BossV->.last Do Begin
      v := NXT(v);  l := l + 1;
      Node[v].boss := BossV end;
    BossV->.len := l;
    BossU->.len := BossU->.len - l;
  end; {If}
end;

Procedure Join( u,v: NodeRange);           {u=last, v=first of segment}
Begin
  Node[u].boss->.next := Node[v].boss;
  Node[v].boss->.prev := Node[u].boss;
  With Node[u] Do  nbor[boss->.O1] := v;
  With Node[v] Do  nbor[1 - boss->.O1] := u;
end;

Procedure Reverse( u,v: NodeRange);        {u,v: first,last of segment}
Var  B, BossP, BossN, BossU, BossV: BossPointer;
Begin
  BossU := Node[u].boss;
  BossV := Node[v].boss;
  BossP := BossU->.prev;  BossP->.next := BossV;
  BossN := BossV->.next;  BossN->.prev := BossU;

  B := BossU;
  Repeat
   With B-> Do Begin
    B := next;
    next := prev;  prev := B;
    O1 := 1 - O1;
    Swap(first,last) end;
  Until B = BossN;

  BossU->.next := BossN;
  BossV->.prev := BossP;
end;
```

```
Procedure Split( B: BossPointer);
Var  i, l, u: Integer;
     A: BossPointer;
Begin
  l := B->.len div 2;
  u := B->.first;
  New(A); With A-> Do Begin
    first := u;
    O1    := B->.O1;
    len   := l;
    prev  := B->.prev;
    next  := B;
    prev->.next := A end;

  For i := 1 To l Do Begin
    Node[u].boss := A; u := NXT(u) end;
  A->.last  := PRV(u);

  With B-> Do Begin
    first := u;
    len   := B->.len - l;
    prev  := A end;
end;

Procedure Meld( A, B: BossPointer);        {A discarded, B gets larger}
Var  u: NodeRange;
Begin
  u := A->.first;
  If A->.O1 = B->.O1 Then
    While u <> B->.first Do Begin
      Node[u].boss := B;
      u := NXT(u)  end
  Else
    While u <> B->.first Do With Node[u] Do Begin
      u := NXT(u);
      Swap(nbor[0],nbor[1]);
      boss := B  end;

  B->.prev := A->.prev;
  B->.first:= A->.first;
  B->.len  := B->.len + A->.len;
  A->.prev->.next := B;
  Dispose(A);
end;

Procedure Balance( u: NodeRange);
Var  k, l: Integer;
     A, B: BossPointer;
Begin
  B := Node[u].boss; l := B->.len;
  If l < MinSec Then Begin
    A := Node[u].boss->.prev;
    k := A->.len;
    Meld(B,A);
    If k + l > MaxSec Then Split(A);
  end;
end;
```

```
Procedure XCon( u1,v1, u2,v2: NodeRange);
Begin
  Cut(u1,v1);  Cut(u2,v2);
  Reverse(v1,u2);
  Join(u1,u2); Join(v1,v2);
  Balance(u1); Balance(u2);
  Balance(v1); Balance(v2);
end;
```

Appendix 2. (Boss-tree)

```
CONST
      N      = 24;                          {Number of the nodes       }
      Height = 4;                           {Height of the boss-tree   }
      MinSec = 2;                           {Shortest allowed section  }
      MaxSec = 2 * MinSec - 1;              {Longest  allowed  section }
      MxSc   = 2 + Max( 4, MaxSec);         {Max len of sections, w.cut}
      Inv    = TRUE;   NoInv  = FALSE;

TYPE
      NodeRange    = 1..N;
      bit          = 0..1;
      Pointer      = ->NodeType;
      Subords      = Array[1..MxSc] of Pointer;
      NodeType     = Record
                       Sub : Subords;
                       O1  : bit;
                       len : Integer;
                       indx: Integer;
                       boss: Pointer;
                     End;
      DArray       = Array[1..Height+1] of bit;       {Real directions}
      NArray       = Array[1..Height] of Pointer;     {Nbors of a path}

VAR
      Node         : Array[NodeRange] of Pointer;
      God          : Pointer;              {->Boss of highest bosses  }

Procedure Inc( Var i : Integer); Begin  i := i + 1  end;
Procedure Dec( Var i : Integer); Begin  i := i - 1  end;
Procedure Rev( Var i : Integer); Begin  i := 1 - i  end;

Procedure Swap( Var u,v : Integer);
Var  i : Integer;
Begin  i := u;  u := v;  v := i  end;

Procedure Dir( A: Pointer; Var dr: DArray);
Var  i: Integer;
Begin
  For i := 2 To Height+1 Do Begin
    A := A->.boss; dr[i] := A->.O1  end;
  For i := Height Downto 2 Do
    If dr[i+1] = 1 Then Rev(dr[i]);
end;
```

```
Procedure Next( A: Pointer;                              {Starting vertex}
            Var Nx: NArray;                              { Next vertices }
            Var du, dv: DArray;                          {Real directions}
                lev: Integer);                           { Starting level}
Var  i, l, d: Integer;
Begin
  l := lev+1;                                          {one level above A}
  While ( ( (du[l] = 0) & (A->.indx = A->.boss->.len) )
       or ( (du[l] = 1) & (A->.indx = 1) ) ) & (l <= Height) Do
  Begin  Inc(l);  A := A->.boss  end;
  If l > Height Then With God-> Do A := Sub[1 + A->.indx mod len]
  Else With A-> Do Begin
    Next( boss, Nx, du,dv, l);
    If du[l] = 0 Then A := boss->.Sub[indx+1]
                 Else A := boss->.Sub[indx-1];
  end;
  d := du[l];
  For i := l-1 Downto lev+1 Do Begin
    If d = A->.O1 Then d := 0 Else d := 1;
    Nx[i] := A;  dv[i] := d;
    With A-> Do If d = 0 Then A := Sub[1] Else A := Sub[len];
  end;
  If d = A->.O1 Then dv[lev] := 0 Else dv[lev] := 1;
  Nx[lev] := A;
end;

Function NXT( u: NodeRange): NodeRange;
Var  du, dv: DArray;
     Nx: NArray;
Begin
  Dir( Node[u], du);
  Next( Node[u], Nx, du, dv, 1);
  NXT := Nx[1]->.len;
end;
Procedure Move ( A: Pointer;                        { Old boss }
                 i1, l: Integer;                    {Segment start/length}
                 B: Pointer;                        { New boss }
                 jb: Integer;                       { Destination index }
                 Inv: Boolean);                     { Reverse order? }
Var  i, j: Integer;
Begin
  With B-> Do Begin
    For j := len DownTo jb Do Begin
      Sub[j+l] := Sub[j];
      Sub[j]->.indx := Sub[j]->.indx + l  end;
    len := len + l;
    If Inv Then
      For i := i1 + l - 1 DownTo i1 Do Begin
        Sub[jb] := A->.Sub[i];
        Rev( Sub[jb]->.O1);
        Sub[jb]->.boss := B;
        Sub[jb]->.indx := jb;
        jb := jb + 1 end
    Else
      For i := i1 To i1 + l - 1 Do Begin
        Sub[jb] := A->.Sub[i];
        Sub[jb]->.boss := B;
        Sub[jb]->.indx := jb;
        jb := jb + 1 end;
  end;
```

```
  With A-> Do Begin
    len := len - l;
    For i := i1 To len Do Begin
      Sub[i] := Sub[i+l];
      Sub[i]->.indx := Sub[i]->.indx - l end;
  end;
end;

Procedure Cut( u,v: NodeRange);
Var  A, B, P, Q: Pointer;
     dr: DArray;
     i, j, l: Integer;
Begin
  Dir( Node[u], dr);
  A := Node[u]; B := Node[v];
  For l := 2 To Height Do Begin
    P := A;  A := A->.boss;
    Q := B;  B := B->.boss;

    If A = B Then With A-> Do Begin
      New(B); B->.O1 := O1; B->.boss := boss; B->.len := 0;
      If dr[l] = 1 Then Move( A, 1, Q->.indx, B, 1, NoInv)
        Else Move( A, Q->.indx, len - P->.indx, B, 1, NoInv);

      If dr[l+1]=0 Then j := 1+A->.indx Else j := A->.indx;
      With A->.boss-> Do Begin
        For i := len DownTo j Do Begin        {Make room for the new B}
          Inc( Sub[i]->.indx);
          Sub[i+1] := Sub[i] end;
        Sub[j] := B;                             {Insert B to A->.boss}
        B->.indx := j;
        Inc(len) end;
    end; {A=B}
  end; {For l}
end;

Procedure Reverse( u,v : NodeRange);
Var  A, B: Pointer;
     S: Subords;
     i, j, k: Integer;
Begin
  A := Node[u]; B := Node[v];
  For i := 2 To Height Do Begin
    A := A->.boss; B := B->.boss end;
  With God-> Do Begin
    k := 1 + B->.indx mod len;
    i := A->.indx;
    Repeat
      Rev( Sub[i]->.O1);
      S[i] := Sub[i];
      Inc(i); If i > len Then i := 1;
    Until i = k;

    i := A->.indx;
    j := B->.indx;
    Repeat
      Sub[i] := S[j];
      Sub[i]->.indx := i;
      Inc(i); If i > len Then i := 1;
      Dec(j); If j = 0 Then j := len;
    Until i = k;
  end;
end;
```

```
Procedure Delete( A: Pointer);
Var  i: Integer;
Begin
  With A->.boss-> Do Begin
    For i := A->.indx+1 To len Do Begin
      Dec(Sub[i]->.indx);
      Sub[i-1] := Sub[i] end;
    Dec(len) end;
  Dispose(A);
end;

Procedure Balance( A,B: Pointer;                        {B is next to A}
                   da,db: bit);                      {The real directions}
Var  d, t: Integer;
Begin
 If A->.len = 0 Then Delete(A) Else
  If A->.len < MinSec Then Begin
    d := MinSec - A->.len;                           {Need}
    t := 1 + B->.len - MinSec + A->.len;             {Tail of B}
    If A->.len + B->.len > MaxSec Then                       {Meld/Split}
      If da=0 Then
        If db=0 Then Move( B, 1, d, A, A->.len+1, NoInv)
                Else Move( B, t, d, A, A->.len+1, Inv)
      Else
        If db=0 Then Move( B, 1, d, A, 1, Inv)
                Else Move( B, t, d, A, 1, NoInv)
    Else Begin                                                 {Meld}
      If da=0 Then
        If db=0 Then Move( A, 1, A->.len, B, 1, NoInv)
                Else Move( A, 1, A->.len, B, B->.len+1, Inv)
      Else
        If db=0 Then Move( A, 1, A->.len, B, 1, Inv)
                Else Move( A, 1, A->.len, B, B->.len+1, NoInv);
      Delete(A);
    end;
  end;
end;

Procedure XCon( u1,v1, u2,v2: NodeRange);         { vi = NXT(ui)      }
Var  Pu, Pv: NArray;                              { The next vertices }
     du, dv, da, db, dc, dd: Darray;              {The real directions}
     A, B, C, D, A0, B0, C0, D0: Pointer;
     i,l: Integer;
Begin
  Cut(u1,v1);  Cut(u2,v2);
  Reverse( v1,u2);

  Dir( Node[u1], da); Dir( Node[u2], db); Next( Node[u2],Pu, db,du, 1);
  Dir( Node[v1], dc); Dir( Node[v2], dd); Next( Node[v2],Pv, dd,dv, 1);

  A0 := Node[u1]->.boss;  B0 := Node[u2]->.boss;
  C0 := Node[v1]->.boss;  D0 := Node[v2]->.boss;

  For l := 2 To Height Do Begin
    A := A0; A0 := A->.boss;
    B := B0; B0 := B->.boss;
    C := C0; C0 := C->.boss;
    D := D0; D0 := D->.boss;
```

```
    If (A = Pv[l]) or (A = D) Then
      If (C = Pu[l]) or (C = B) Then Begin          {Cyclic precedence}
        If A->.len = 0 Then Delete(A);
        If B->.len = 0 Then Delete(B);
        If (C->.len = 0) & (C <> B) Then Delete(C);
        If (D->.len = 0) & (D <> A) Then Delete(D) end
      Else Begin                                              {C,D first}
        Balance( C,D, dc[l],dd[l]);
        Balance( D,Pv[l], dd[l],dv[l]);
        If A <> D Then Balance( A,B, da[l],db[l]);
        Balance( B,Pu[l], db[l],du[l]) end
    Else Begin                                                {A,B first}
      Balance( A,B, da[l],db[l]);
      Balance( B,Pu[l], db[l],du[l]);
      If C <> B Then Balance( C,D, dc[l],dd[l]);
      Balance( D,Pv[l], dd[l],dv[l]) end;
  end;
end;
```