Tutorial on Reverse Search with C Implemenatations

David AVIS

School of Computer Science
McGill University
3480 University, Montreal, Quebec, Canada H3A 2A7
avis@cs.mcgill.ca
13 July 2000

1. Introduction

This paper is a tutorial on reverse search, a technique developed by Komei Fukuda and the author for the generation of large sets of discrete objects [1]. Although reverse search was originally used for generating all vertices of a convex polyhedron, it has been applied to many other geometric and combinatorial problems. The purpose of this tutorial is to illustrate the basic technique on some simple examples and go through some basic implementations in C. Various refinements are studied, along with their effect on the time complexity and running time of the implementation. This is an informal introduction, and the reader is referred to [1] for a rigorous approach and other more complex examples.

2. Basic Components: Adjacency and Local Search

Reverse search is a technique for generating large, relatively unstructured, sets of discrete objects. In its most basic form, it can be viewed as the traversal of a spanning tree, called the *reverse search tree*, of a graph G whose nodes are the objects to be generated. Edges in the graph are specified by an *adjacency oracle*, and the subset of edges of the reverse search tree are determined by an auxiliary function, which can be thought of as a *local search function* f for an optimization problem defined on the set of objects to be generated. One vertex, v^*, is designated as the *optimum vertex*. For every other vertex v of G, repeated application of f must generate a path in G from v to v^*. The set of these paths defines the reverse search tree, which has root v^*. A reverse search is initiated at v^*, and only edges of the reverse search tree are traversed. When a node is visited the corresponding object is output. Since there is no possibility of visiting a node by different paths, the nodes are not stored. Backtracking can be performed in the standard way using a stack, but this is not required as the local search function can be used for this purpose. This means that it is not necessary to keep more than one node of the tree at any given time, and this memoryless property is the main feature of reverse search.

For a given problem, there may be many choices of adjacency oracle and local search function. However, in the basic setting described here, a few properties are required. Firstly, the underlying graph G must be connected and an upper bound on the maximum vertex degree, *maxdeg*, must be known. As we will see the performance of the method depends on G having *maxdeg* as low as possible. The adjacency oracle must be capable of generating the adjacent vertices of some given vertex v sequentially and without repetition. This is done by specifying a function $Adj(v, i)$, where v is a vertex of G and $i = 1, 2, \ldots,$ *maxdeg*. Each value of $Adj(v, i)$ is either a vertex adjacent to v or *null*. Each vertex adjacent to v appears precisely once as i ranges over its possible values. Finally, the local search function f should be easy to compute, so that it is easy to determine edges of the reverse search tree that will be traversed. These ideas can be illustrated on a simple example: generating all permutations of a set of integers.

Example 1: Permutations

Given an integer n we are asked to generate all permutations of the integers $1, 2, \cdots, n$. For example, if $n = 3$ the required output is:

$$123, \quad 231, \quad 312, \quad 213, \quad 132, \quad 321.$$

From the discussion in the previous section we require:

(i) a graph G defined on the set of permutations given by an adjacency oracle;

(ii) a starting vertex v^*; and

(iii) a local search function f which defines a spanning tree of the graph G with root v^*.

For given n, we represent a permutation of $1, 2, \cdots, n$ by $x_1 x_2 \cdots x_n$ There are many choices for an adjacency oracle to define G. One is as follows: for every $i = 1, \ldots, n-1$,

$$Adj(x_1 x_2 \cdots x_n, i) = x_1 \cdots x_{i-1} x_{i+1} x_i \cdots x_n.$$

In other words, we can swap any two adjacent integers in a given permutation to get one of its neighbours in G. For example

$$Adj(213, 1) = 123, \quad Adj(213, 2) = 231$$

and

$$Adj(3124, 1) = 1324, \quad Adj(3124, 2) = 3214, \quad Adj(3124, 3) = 3142.$$

From the definition, we see that G is a regular graph of degree $n-1$ and so we have $maxdeg = n-1$. In fact, when $n = 3$, G is the cycle of length 6, see Figure 2.1(a).

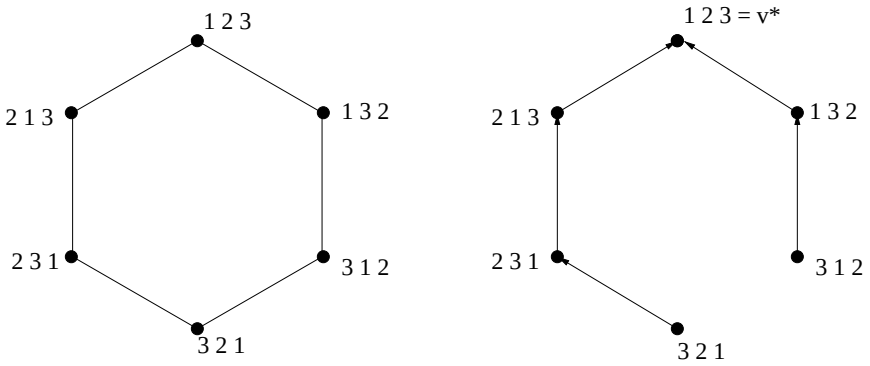

Figure 2.1(a)Graph of permutations with $n = 3$ (b) Spanning tree defined by f

Next we define a spanning tree on G by means of a local search function f. For the starting vertex we choose the identity permutation, so that

$$v^* = 12 \cdots n.$$

To define the spanning tree, we define a function f on the vertices of G whose repeated application yields a path in G to v^*. Here is one choice for f:

$$f(x_1 x_2 \cdots x_n) = x_1 \cdots x_{i-1} x_{i+1} x_i \cdots x_n$$

where i is the smallest integer such that $x_i > x_{i+1}$. To complete the definition, set $f(v^*) = v^*$. For example, with $n = 5$ we have:

$$f(31245) = 13245, \quad f(13245) = 12345 \quad f(12345) = 12345.$$

It is easy to see f has the required properties. Clearly for any vertex of G that is not v^* it gives an adjacent vertex in G. In fact it behaves like bubble sort, in that large integers move to the right and small integers move to the left. It is an easy exercise to show that starting at any permutation, repeated application of f leads to v^*, ie. the permutation is "sorted". f therefore defines a unique path from any vertex of G to v^*, and the set of these paths is the desired spanning tree of G. Figure 2.1(b) gives the spanning tree when $n = 3$, where the edges are directed towards the root, as induced by the function f.

Exercises.

2.1. Draw the graph of permutations for $n = 4$ and corresponding reverse search tree.

2.2. Let D be a directed graph on n nodes, so that for each edge uv of D, $u < v$. A permutation of 1 2 ... n is *compatible* with D if for each edge uv, u occurs before v in the permutation. For example if D has the single edge 12, the compatible permutations are 1 2 3, 1 3 2, 3 1 2. Generalize the results of this section to the problem of generating all permutations compatible with D (note that 1 2 ... n is always compatible).

3. Reverse Search Algorithm

The reverse search algorithm can be stated generally in terms of the basic components described in the previous section. We use these components to describe four functions: *root*, *reverse*, *backtrack* and *reversesearch*. In this section we give *C*-like pseudo code for the simplest generic implementation of these functions. Depending on the application, specifically tailored implementations can greatly improve overall performance.

The boolean function *root*(v) returns *true* if v is v^*, the root of the reverse search tree, otherwise it returns *false*. The boolean function *reverse*(v, i) is used to determine if the neighbour $w = Adj(v, i)$ of v is a child of v in the reverse search tree. It returns *true* in this case, otherwise it returns *false*. Note that $Adj(v, i)$ may be *null* for some values of i.

```
reverse(v, i)
  w = Adj(v, i);
  if (w! = null)
     return ( v = = f(w)) ;
  else
     return FALSE;
```

The integer function *backtrack*(v) is used to return from a node v to its parent $w = f(v)$ in the reverse search tree. It also finds the value i such that $v = Adj(w, i)$, and this is returned to the calling function.

```
backtrack(v)
  i = 0;
  child = v;      /* keep a copy of v */
  v = f(v);
  do i++;
       while ( child ! = Adj(v, i));
  return i;
```

With these three functions we can specify the function *reversesearch*(v, *maxdeg*) which traverses the reverse search tree rooted at v.

```
reversesearch(v, maxdeg)
  i = 0;                          /* adjacency counter       */
  count = 1;
  output(v);
  while  ( ! root(v)  ||  i < maxdeg )      /* main loop */
   {
    do  i++;
      while ( i ≤ maxdeg  &&  ! reverse(v, i) ) ;

    if ( i ≤ maxdeg )         /* go deeper in reverse search tree */
          {
             v = Adj(v, i);
           count++;
             output(v);
             i = 0;
          }
    else                        /* backtrack and restore v, i*/
          i = backtrack(v);
   }
    return count;
```

In the code the variable v is the tree node currently being processed, and is initially the root of the tree. In the main loop each neighbour $Adj(v, i)$ of v is examined to see if it is a child of v in the tree. This is the function of the **do .. while** loop, and uses the function *reverse*. If this loop terminates with a value $i \leq maxdeg$ a child has been found, and v is updated to be this node. Otherwise no further children exist, and a backtrack step is performed, replacing v by its parent, and restoring i to the value it had previously. This is performed by the function *backtrack*. The program terminates when the last potential child of the root has been examined, that is when v is the root and $i = maxdeg$, and returns the number of objects generated.

We now consider the time and space complexity of *reversesearch*. Suppose the graph G has $|V|$ vertices and $|E|$ edges. Let respectively $t_{reverse}$, t_{Adj}, t_f, $t_{backtrack}$ $t_{reversesearch}$ be the time required to execute *reverse*, *Adj*, f, *backtrack* and *reversesearch*. In each execution of the main loop, the current node v is either replaced by a child or by its parent in the tree. Therefore this loop is executed at most $2\,|V|$ times. For each vertex the **do** statement is executed for $i = 1, \ldots, maxdeg$, hence *reverse* is executed a total of $maxdeg\,|V|$ times. In *reversesearch*, the **if** statement and the **else** statement are both executed once for each vertex v, for a total of $|V|$ times each. Adding up we see that

$$t_{reversesearch} = O(\ |V|\ (\ maxdeg\ t_{reverse} + t_{Adj} + t_{backtrack}\ )\ ) \tag{3.1}$$

In the implementation of *reverse* above the function *Adj* is evaluated each time, but the function f is only evaluated when *Adj* returns a neighbour. Therefore in *reverse*, *Adj* is executed a total of *maxdeg* $|V|$ times, whereas f is executed a total of $|E|$ times. For *backtrack* we have

$$t_{backtrack} = O(\ t_f + maxdeg\ t_{Adj}).$$

If we substitute into (3.1) and simplify, using the fact that $|E| \geq |V| - 1$, we get

$$t_{reversesearch} = O(\ maxdeg\ |V|\ t_{Adj} + |E|\ t_f). \tag{3.2}$$

From this equation we see the importance of a small value of *maxdeg* and an efficient implementation of both *Adj* and f.

For the space complexity, we observe that no data structures are used in the search apart from that required to hold v and perhaps its parent and one of its children. The space complexity is therefore proportional to the space required to hold a single node of G.

Exercise

3.1 Trace the application of *reversesearch* on the example of Figure 2.1.

4. A C Implementation for Permutations

In this section we examine a possible C implementation of reverse search for the problem of generating permutations. The first step is to decide the data type *perm* to hold a permutation. For simplicity we choose a static array and use the global variable n to keep the size of the permutations to be generated:

```
typedef int perm[100];
int n;
```

Next we define the functions $f(v)$ and $Adj(v, i)$. Since *perm* is a static array, these functions cannot return a permutation. Instead we update the permutation supplied as the argument v. The implementations are straightforward from the definitions in Section 2.

```
void swap (perm v, int i)
{
  int t;
  t = v[i];
  v[i] = v[i + 1];
  v[i + 1] = t;
}

void Adj (perm v, int i)        /* adjacency oracle */
{
  swap (v, i);
}

void f (perm v)                 /* local search function */
{
  int i=1;
  while ( i < n && v[i] <= v[i + 1] ) i++;
  if (i < n) swap (v, i);
}
```

Now we can implement the three basic functions for reverse search: $root(v)$, $reverse(v, i)$ and $backtrack(v)$. The first functions is simply:

```
int root (perm v)
{
  int i;
  for (i = 1; i <= n; i++)
    if (v[i] ! = i) return FALSE;
  return TRUE;
}
```

To implement $reverse(v, i)$ we need to make a copy, w, of v before making the required test, which is implemented by the function *equal*. Note that both *Adj* and f update the value of w:

```
int equal (perm w, perm v)
{
 int i;
 for (i = 1; i <= n; i++)
   if (w[i] ! = v[i]) return FALSE;
 return TRUE;
}

int reverse (perm v, int i)
{
 perm w;
 copy (w, v);
 Adj (w, i);
 f (w);
 return equal (v, w);
}
```

In this implementation of *backtrack*(*v*) we again need a copy *child*, of *v*. This is used in the **do..while** loop that restores the value of *i*. Note that in this loop it is necessary to restore *v* at each iteration, as it gets updated in the call *Adj*(*v*, *i*). Here we use the fact that in this application, two consecutive calls *Adj*(*v*, *i*) will result in *v* being unchanged.

```
int backtrack (perm v)
{
 perm child;
 int i = 0;
 int found = FALSE;
 copy (child, v);
 f (v);                    /* v is parent of child */
 do{
    i++;
    Adj (v, i);             /* v replaced by jth child */
    found = equal (child, v);
    Adj (v, i);            /* restore v */
   } while (!found);
 return i;
}
```

Having assembled all the pieces, we complete the program by implementing *reversesearch*(*v*, *maxdeg*) which controls the search itself. This is virtually identical to the pseudocode given in the last section:

```
int reversesearch (perm v, int maxdeg)
{
 int i=0, count=1;
 output (v);
 while (!root (v) | | i < maxdeg)
  {
   do i++;
      while (i <= maxdeg && !reverse (v, i));
   if (i <= maxdeg)
     {
      Adj (v, i);
      output (v); count++;
      i = 0;
     }
   else i = backtrack (v);
  }
 return count;
}
```

The program can be tested by the simple driver:

```
int main ()
{
  perm v;
  int i, count;
  scanf ("%d", &n);   /* enter size of permutation */
  for (i = 1; i <= n; i++)  v[i] = i; /* set root */
  count = reversesearch (v, n - 1);
  printf ("\nnumber of permutations=%d\n", count);
  return 0;
}
```

Let us analyze the time and space complexity of this implementation. It is easy to see that $|V| = n!$, $|E| = (n-1)\,|V|$, $maxdeg = n-1$, $t_{Adj} = O(1)$ and $t_{reverse} = O(n)$. Therefore, using equation (3.2) we see that

$$t_{reversesearch} = O(n(n-1)\,n!),$$

or quadratic time per permutation.

Exercise

4.1 Implement the algorithm for listing compatible permutations described in Exercise 2.2.

5. An Improved Implementation

The implementation given in the previous sections is very basic. It can be improved in two obvious places: restoring the index i in *backtrack* and performing the test in *reverse*. Analysis of equation (3.2) show that these are the bottlenecks in the program. To improve the function *backtrack* we rewrite $f(v)$ so that it is an integer function that returns the value of the index i such that $v = Adj(f(v), i)$. Then *backtrack* becomes a function call to f, with running time $O(n)$:

```
int f (perm v)              /* local search function */
{
  int i=1;
  while ( i < n && v[i] <= v[i + 1] ) i++;
  if (i < n) swap (v, i);
  return i;
}

int backtrack (perm v)
{
  return f(v);
}
```

Improving $reverse(v, i)$ is more subtle. The basic idea is to determine if $v = f(Adj(v, i))$ without directly computing $Adj(v, i)$ and f. Consider the permutation $v = 12354786$ and suppose we know that the smallest index s for which $v[s] > v[s+1]$ is $s = 4$.

Case 1. $i < s = 4$**.**

Since $v[i] < v[i+1]$, these integers are out of order in $Adj(v, i)$. Since $s > i$, they will be restored by f, and so $reverse(v, i)$ is true.

Case 2. $i = s = 4$.

Since $v[i] > v[i+1]$ we can immediately conclude that $reverse(v, i)$ is false. This is because $Adj(v, i)$ will place these two entries in order, and this cannot be undone by f, which is a sorting function.

Case 3. $i > s + 1 = 5$

In this case f will swap $v[s]$ and $v[s+1]$, so $reverse(v, i)$ is false.

Case 4. $i = s + 1 = 5$**,** $v[s] < v[s+2]$**.**

In this case, the first out of order pair in $Adj(v, i)$ are $v[s+1]$ and $v[s+2]$. These get put in order by f, so $reverse(v, i)$ is true. Note that if $v[s] > v[s+2]$ (eg. $v = 12384756$) then s is not changed by $Adj(v, i)$ and $reverse(v, i)$ is false.

These cases are exhaustive, and can be summarized by the following proposition whose proof is left as an exercise.

Proposition. Let v be a permutation of length n and let s be the smallest index for which $v[s] > v[s+1]$. Then *reverse*(v, i) is true if and only if
(a) $i < s$, or
(b) $i = s + 1 < n - 1$ and $v[s] < v[s+2]$. □

We can use the proposition to implement *reverse* in $O(1)$ time, providing we know the value of s. This can be calculated once each time v is updated by a function *findindex*(v) and stored in the unused position $v[0]$ of the array storing v.

```
findindex(perm v)  /* find first out of order position in v */
{
 int i;
 for(i=1; i < n && v[i] < v[i+1]; i++);
 v[0]=i;
}
```

The statement *findindex*(v) is inserted as the first statement in the main loop of *reversesearch*, just before the **do** statement. Finally we can replace *reverse* by:

```
int reverse (perm v, int i)
{
 perm w;
 int s = v[0];    /*smallest index s.t.v[s]>v[s+1]*/
 if ( i <= s-1 ) return TRUE;
 if ( i == s+1 && s <= n-2 && v[s] < v[s+2] ) return TRUE;
 return FALSE;
}
```

We can measure the improvements analytically by using equation (3.1) of Section 2. As noted above $t_{backtrack} = t_f$ and $t_{reverse} = O(1)$. However for each execution of the main loop we must execute *findindex*, at a cost of $O(n)$ time. Making the substitutions in (3.1) we obtain

$$t_{reversesearch} = O(n\ n!),$$

or linear time per permutation. We can test this empirically by running the two programs. For example, on a Sun Solaris with $n = 10$ there are 3,628,800 permutations. The original version took 28 seconds and the improved version took 5 seconds, 5.6 times faster.

Exercise

5.1 Generalize the proposition to the problem of listing compatible permutations described IN Exercise 2.2.

6. Example 2: Trees

In this section we show how to develop a reverse search algorithm for generating all labeled trees on $n + 1$ points. We label the vertices 0,1, ..., n and consider the tree rooted at vertex 0. It is convenient to direct the edges in a tree toward the root. Each vertex i is the endpoint of one directed edge ij, and we define $tree(i) = j$. It is well known that there are $(n + 1)^{n-1}$ such trees. Figure 6.1 shows the three labeled trees on three points.

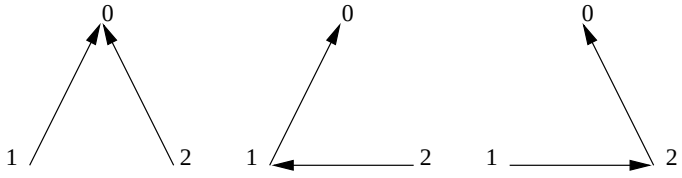

Figure 6.1 Labeled trees on 3 points

The first step in designing a reverse search algorithm for generating trees is to define the adjacency relationship. Given a tree T, consider a vertex $i \geq 1$. There is a unique directed edge ik in T. For any vertex $j \neq k$ we construct T' from T by deleting edge ik and inserting edge ij. T' is a tree if and only if the path from j to the root of T does not include vertex i. For example, in Figure 6.2, edge 51 may replace edge 52, but edge 15 may not replace 10, since 1 is on the path from 5 to 0.

Using this idea we can construct the adjacency oracle, for $1 \leq i \leq n$, $0 \leq j \leq n$, $j \neq i$:

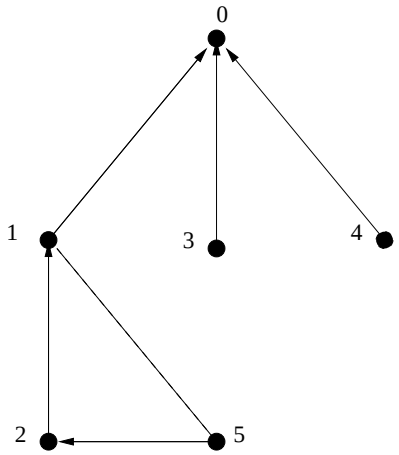

Figure 6.2 Adjacency for trees

$$Adj(T, i, j) = \begin{cases} T - ik + ij & \text{where } k = tree(i) \text{ and } i \text{ is not an ancestor of } j \\ nil & \text{otherwise} \end{cases}$$

In this definition we have used two parameters i and j for clarity. They could of course be replaced by a single parameter to conform to the definition in Section 2. The essential point is that all neighbours of a vertex (which is a tree in this case) can be found by trying all possible indices i and j. Notice also that in this example, some values of i and j do not give a neighbour of T.

For the starting tree T^*, we choose the star with edges $i0$, $i = 1, \ldots, n$. A local search function is easy to define. For any tree $T \neq T^*$ choose any node i not adjacent to the root and set $tree(i) = 0$. In other words:

$$f(T) = T - ik + i0 \qquad \text{where } i \text{ is the smallest index such that } k = tree(i) \neq 0.$$

Clearly for any tree $T \neq T^*$ at most $n - 1$ applications of f leads to the star T^*. We are now ready to describe the implementation. We define the data type *tree* as a static array:

```
typedef int tree[100];
int n;
```

The star T^* is represented by setting the array to zero, and the procedure $root(v)$ simply tests this condition. The adjacency oracle can be implemented:

```
int Adj (tree v, int i, int j)        /*   adjacency oracle */
{
 int p=v[j];
 while ( p != 0 && p != i ) p=v[p];
 if (p == i) return FALSE;          /* i is ancestor of j    */
 v[i]=j;                        /* j becomes parent of i */
 return TRUE;
}
```

For the local search function $f(v)$, we implement the improved version described in Section 5 that returns the indices i and j such that $v = f(Adj(v, i, j))$. Since C allows only one value in a return statement, we pass parameters i and j explicitly as pointers:

```
void f (tree v, int* i, int* j)              /* local search function */
{                    /* set i,j s.t v=Adj(f(v),i,j)  then set v=f(v) */
  int k=1;
  while ( k <= n && v[k] ==  0 ) k++;
  if (k <= n)                              /* v is not root of tree */
   { (*i)=k; (*j)=v[k];
     v[k]=0;
   }
}
```

Using these functions we have a very simple procedure $reverse(v, i, j)$:

```
int reverse (tree v, int i, int j)
{
 tree w;
 copy (w, v);
 Adj (w,i,j);
 f (w,&i,&j);     /* i, j get changed, but are not returned to reversesearch */
 return equal (v, w);
}
```

As we saw in Section 5, the procedure *backtrack* becomes redundant, and we can write procedure *reversesearch*:

```
int reversesearch (tree v)
{
 int i=1, j=1, count=1, depth=0;
 output (v,depth);
 while ( i <= n )
  {
   do next(v,&i,&j);
      while (i <= n && !reverse (v,i,j));
   if (i <= n)
    {
     Adj (v,i,j);
     count++; depth++;
     output(v,depth);
     i=1; j=1;
    }
   else if (!root(v))
         {
          f(v,&i,&j);
          depth--;
         }
  }
 return count;
}
```

In the code we use the function *next*(v, i, j) to cycle through the possible values of i and j:

```
next(int* i, int* j)
{
 (*j)++;  if ( (*i) == (*j)) (*j)++;
 if ((*j) > n )
   { (*i)++; (*j)=1; }
}
```

This concludes the implementation. A sample output for $n = 3$ is given in Figure 6.3. Each line lists the parent of node 1,2 and 3 of the tree, respectively, and the depth of the vertex in the reverse search tree. As noted above, the maximum depth is $n - 1 = 2$.

For the complexity analysis, the definition of *Adj* implies that *maxdeg* $= n^2$. The implementations of *Adj* and f both have time complexity $O(n)$. Therefore from equation 3.2 we see that the time complexity is $O(n^3)$ per tree generated.

Exercises

6.1. Show that the procedure *next* can be improved by adding the statement

```
if( v[*i] != 0 ) (*i)=n+1;
```

just before the final parenthesis.
Hint: Show that *reverse* will be false for all remaining values of i and j.

6.2. Observe that if the array *tree* is read in reverse order, the output in Figure 6.3 comes in lexicographically increasing order. Prove this fact and use it develop a simple iterative algorithm for listing lableled trees.

6.3. Generalize the method to find all trees whose edges come from the edges of a given graph on $n + 1$ points.

0 0 0 depth=0
2 0 0 depth=1
3 0 0 depth=1
0 1 0 depth=1
3 1 0 depth=2
0 3 0 depth=1
2 3 0 depth=2
3 3 0 depth=2
0 0 1 depth=1
2 0 1 depth=2
0 1 1 depth=2
0 3 1 depth=2
0 0 2 depth=1
2 0 2 depth=2
3 0 2 depth=2
0 1 2 depth=2

Figure 6.3 All 16 labelled trees on 4 nodes

7. Euclidean Trees

In this section we describe a geometrical version of the spanning tree problem. The input is a set of n points in the plane, no three of which are collinear. A *Euclidean tree* is a spanning tree drawn with the points as vertices, straight line segments as edges, and such that no two edges intersect at an interior point. The number of such trees depends on the configuration of points. In this section we show how a very small modification of the method described in the last section can be used to list all Euclidean trees for a given input point set.

The modification consists of simply adding the new condition on non crossing edges. The revised adjacency oracle is, for $1 \le i \le n$, $0 \le j \le n$, $j \ne i$:

$$Adj(T, i, j) = \begin{cases} T - ik + ij & k = tree(i),\ i \text{ is not an ancestor of } j \text{ and } ij \text{ does not cross an edge in } T \\ nil & \text{otherwise} \end{cases}$$

In Figure 7.1, the tree in the centre is adjacent to the four outer trees.

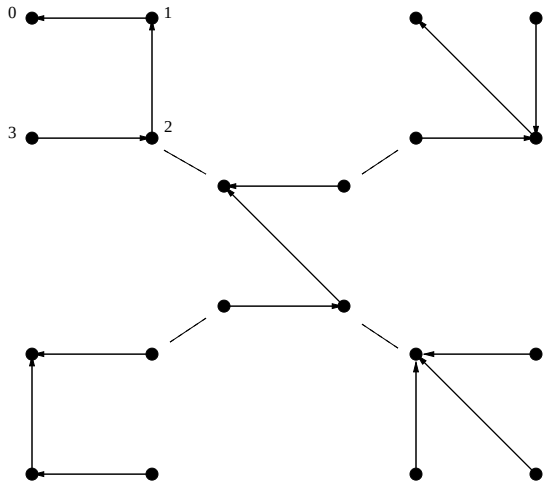

Figure 7.1 Adjacency for Euclidean trees

We choose the same target tree as before: the star with vertex 0 as the centre. Finally the local search function becomes: choose the vertex with smallest index i that is not adjacent to the root and for which the segment $i0$ does not cross any edge of T, then set $tree(i) = 0$. In other words:

$$f(T) = T - ik + i0 \quad \text{where } i \text{ is the least index } s.t.\ k = tree(i) \ne 0 \text{ and } i0 \text{ does not cross any edge of } T.$$

It is now necessary to prove that such a vertex i exists in every Euclidean tree that is not the target T *. A proof of this can be found in [1]. As before, for any tree $T \ne T$ * at most $n - 1$ applications of f leads to the star T *.

For the implementation we require a function *crossedge*(*tree v*, *int i*, *int j*) which returns *true* if the segment from i to j crosses any edge in the tree v.This is a standard function in computational geometry which we do not discuss here. With this function a few small changes to the code in the last section

suffice. Line 5 of *Adj* is replaced by:

```
if ( (p==i) || crossedge(v,i,j) ) return FALSE;
```

Line 4 of function f is replaced by:

```
while ( k <= n && ( v[k] ==  0 || crossedge(v,0,k)) ) k++;
```

Line 5 of function *reverse* becomes:

```
if( !Adj (w,i,j)) return FALSE;
```

The main program reads the set of points and performs a reverse search from *T* *. A sample output for $n = 3$ is given in Figure 7.3. The points are listed with their euclidean coordinates, and a list of crossing segments is computed. Each following line lists a Euclidean tree, by listing the parent of node 1,2 and 3 in the tree, respectively, and the depth of the vertex in the reverse search tree. As noted above, the maximum depth is $n - 1 = 2$.

```
points: 0 5, 5 5, 5 0, 0 0
crossings (if any):
0 2   1 3
euclidean spanning trees:
 0 0 0   depth=0
 2 0 0   depth=1
 0 1 0   depth=1
 3 1 0   depth=2
 0 1 1   depth=2
 0 3 0   depth=1
 2 3 0   depth=2
 3 3 0   depth=2
 0 3 1   depth=2
 0 0 2   depth=1
 2 0 2   depth=2
 0 1 2   depth=2
number of trees=12
```

Figure 7.3 Euclidean trees on the point set of Figure 7.1

The complexity analysis is similar to that of the last section. The definition of *Adj* implies that $maxdeg = n^2$. The function *crossedge* requires $O(n)$ time. It is called once from *Adj*, so $t_{Adj} = O(n)$. However f may require up to n calls to *crossedge*, so $t_f = O(n^2)$. Therefore from equation 3.2 we see that the time complexity is $O(n^4)$ per tree generated. We can reduce this back to $O(n^3)$ per tree by precomputing all values of *crossedge* for each tree, and storing them in a table. This table can be built in $O(n^3)$ time and accessed in $O(1)$ time, so $t_f = O(n)$.

Exercises

7.1. Prove that the function f is well defined.

7.2. Generalize the method to find all Euclidean trees whose edges come from the edges of a graph drawn on the given point set.

References

1. D. Avis and K. Fukuda, “Reverse Search for Enumeration,” *Discrete Applied Math*, 6, pp. 21-46 (1996).