1-1-1993

A Report On Planar Point Location: Some New Techniques

Sumeet Rajput

INSTITUTE FOR SIMULATION AND TRAINING

COMPUTATIONAL GEOMETRY CDA 6938

A REPORT ON

PLANAR POINT LOCATION

SOME NEW TECHNIQUES

BY

SUMEET RAJPUT

IST

Computational Geometry CDA 6938

A Report on

Planar Point Location

Some New Techniques

By

Sumeet Rajput

Introduction

Point location, often known in graphics as **"hit detection"**, is one of the fundamental problems of Computational Geometry. In a point location query we want to identify which of a given collection of geometric objects ie. polygons, edges or points contains a particular point. The point location query is normally carried out on a **Planar Straight Line Graph (PSLG)**. This is often referred to as a subdivision. We judge the performance of a particular point location method by measuring three key factors:

1. The **preprocessing time, P**: This is the time it takes for the PSLG to be processed and arranged in a suitable data structure which can then be searched. This is expressed as a function of the input size (the number of points in the subdivision).

2. The **space complexity, S**: This determines the amount of storage that the data structure requires and is again expressed as a function of the data size.

3. The **query time, Q**: This is the time required to search the data structure for the query point. Again, it is expressed as a function of the input data size.

Here is a brief summary of some of the algorithms already known to us:

Slab Method

This method consists of splitting up the plane into vertical slabs. The line segments of the subdivision intersecting a slab are totally ordered, from the bottom to the top of the slab. Associate with each line segment the polygon just above it. Now it is possible to locate a query point with two binary searches: the first on the x-coordinate, locates the slab containing the point; the second, on the line segments intersecting the slab, locates the nearest line segment below the point, and hence the polygon containing the point. It has complexity bounds $Q = O(\log n)$, $S = O(n^2)$, and $P = O(n^2)$.

Chain Method

In this method the plane is subdivided by a number of monotone "chains". The chains can be ordered and easily arranged in a data structure used for searching. Essentially, the search procedure involves discriminatimg the point against a particular chain. Another binary search within two chains allows the point to be discriminated to within an edge of the chain. The region containing the point can then be ascertained. Therefore, we have $Q = O(\log^2 n)$, $S = O(n)$ and $P = O(n\log n)$.

Kirkpatrick's Triangulation Refinement Method

Here we assume the PSLG to be **triangulated** (if not it can be transformed into one in O(n log n) time. A search tree is then created whose nodes represent triangles. Each enclosing triangle can be split up into regions which are themselves triangles. Point location proceeds by testing the inclusion of a point within a triangle and its children recursively until we come to the point where the triangle enclosing the point is a leaf node of the tree (ie. an actual triangle of the subdivision). Thus, Q = O(log n), S = O(n) and P = O(n log n).

Some New Techniques

In addition to the techniques already discussed above some new techniques are presented below:

1. Planar point location using persistent search trees

Introduction

This technique uses the concept of persistent data structures. A persistent data structure differs from an ordinary one in the sense that the current version of the structure can be modified and all versions of the structure, past and present, can be accessed.
This technique developed from **Cole's observation**. Cole observed that the point location problem reduces to the problem of storing a sorted set subject to insertions and deletions so that all past versions of the set, as well as the current version, can be accessed efficiently.

We start by dividing the PSLG into vertical slabs which are created by drawing a vertical line through each vertex. The line segments of the subdivision intersecting a slab are totally oredered, from the bottom to the top of the slab. We notice, as Cole did, that the sets of line segments intersecting contiguous slabs are similar. Considering the x-coordinate as time we can see how these sets change as time increases from -infinity to +infinity. As the boundary from one slab to the next is crossed, certain segment are deleted from the set and other segments are inserted.

A data structure is then created which is capable of storing the different sets of line segments ocurring in different slabs but which allows access to any set in logarithmic time. Such a data structure is a persistent form of a balanced binary tree.

Persistent sorted sets and search trees

The problem is to maintain a set of items that changes over time.

Three operations on the set are allowed:

1. **Access(x, s, t):** Find and return the item in set s at time t with greatest key less than or equal to x. If there is no such item, return a special null item.

2. **Insert(i, s, t):** At time t, insert item i into set s.

3. **Delete(i, s, t):** At time t, delete item i from set s.

We start with an empty set, and we wish to perform on-line a sequence of operations, including m updates(insertions and deletions) with one stipulation that any update occurs at a time no earlier than any previous operation in the sequence.

An approach is to start with an **ephemeral data structure** (one that is not persistent) for sorted sets or lists and make it persistent. This was a technique pursued by many other authors and is called **path copying.**

Review of binary search trees and how they can be made persistent using path copying

A binary search tree is a binary tree containing the items of the set in its nodes, one item per node, with the items arranged in symmetric order: if x is any node, the key of the item in x is greater than the keys of all items in its left subtree and less than the keys of all items in its right subtree. A binary search tree can be balanced by storing certain balance information in each node. Some types of balanced binary trees are **AVL trees, weight balanced trees, and red-black trees.** The authors use the red-black tree for storing data because updating them is specially efficient.

In red-black trees each node has a color, either red or black, subject to the following constraints:

(i) all missing (external) nodes are regarded as black.
(ii) all paths from the root to a missing node contain the same number of black node.
(iii) any red node, if it has a parent, has a black parent.

To make a tree balanced after an insertion or deletion rotations are performed. Rebalancing red-black trees require O(1) rotations and O(log n) color changes. Please see figure 1 for insertion in a red-black tree.

Making red-black trees persistent (Path copying)

Please refer figure 2(a).

Let us now consider how to make red-black trees persistent. We need a way to retain the old version of the tree when a new version is created by an update. We can of course copy the entire tree each time an update occurs, but this takes O(n) time and space per update. Another idea is to copy only the nodes in which changes are made. **Any node that contains a pointer to a node that is copied must itself be copied.** Assuming that every node contains pointers only to its children, this means that copying one node requires copying the entire path to the node from the root of the tree. Thus this method is called **path copying.**

The effect of this method is to create a set of search trees, one per update, having different roots but sharing common subtrees. The time and space per update in a red-black tree is O(log n) since such an operation changes only nodes along a single path in the tree. If the update times are the integers 1 through m, we can create an array of pointers to the roots ordered by the time of creation. Thus we can have direct access into the root array to provide O(1)-time access to the appropriate root, and the total time for an access operation is only O(log n).

Making red-black trees persistent (No node copying)

Please refer figure 2(b).

A major **drawback of the path copying method is its non linear space usage** bacause we always copy the entire access path each time an update occurs. Here red-black trees are implemented by allowing nodes to become arbitrarily **"fat"**: each time we want to change a pointer, we store the new pointer in the node, along with a time stamp indicating when the change occurred.

With this approach an update takes only O(1) space, since an insertion creates only one new node and either kind of update causes only O(1) pointer changes. However, there is a time penalty: since a node can contain an arbitrary number of left or right pointers, deciding which one to follow during a search is not a constant time operation. If we use binary search by time stamp to decide which pointer to follow, choosing the correct pointer takes O(log m) time (m is the number of updates), and the time for an access, insertion or deletion is O((log n)(log m)).

Making red-black trees persistent (Limited node copying)

Please refer figure 2(c).

This method removes the time penalty of the previous method. We allow each node to hold k pointers in addition to its original two. We choose k to be a small positive constant say k = 1. When attempting to add a pointer to a node, if there is no empty slot

for a new pointer, we copy the node, setting the initial left and right pointers of the copy to their latest values. (Thus the new node has k empty slots). We must also store a pointer to the copy in the latest parent of the copied node. If the parent has no free slot, it, too, is copied. Thus copying proliferates through successive ancestors until the root is copied or a node with a free slot is reached.

Searching the resulting data structure is quite easy: when arriving at a node, we determine which pointer to follow by examining the key to decide whether to branch left or right and examining the time stamps of the extra pointers to select among multiple left or multiple right pointers. (We follow the pointer with latest time stamp no greater than the search time if there is one, or else the initial pointer). As with path copying, a single update operation can result in O(log n) nodes. However, on the average there are only O(1) nodes copied per update, implying an O(n) space bound for the data structure. An update (insertion or deletion) requires O(log n) time. Similarly, an access requires O(log n) time if a root array is maintained which provides O(1)-time access to the roots.

Applications and extensions

The authors have proposed a new data structure for representing persistent sorted sets. The structure has O(log m) access time (where m is the total number of update), O(log n) update time, and needs O(1) amortized space per update starting from an empty set. The preprocessing time necessary to build the data structure is O(n log n) and the query time is O(log n).

The structure supports a generalization of the planar point location problem in which the queries are of the following form: given a vertical line segment, report all polygons the segment intersects. Such a query is equivalent to an access range operation on the corressponding persistent sorted set and thus takes O(log n + k) time where k is the number of reported polygons.

2. Optimal point location in a monotone subdivision

Introduction

This technique shows an elegant modification to the separating chain method of Lee and Preparata. The algorithm is based on a new data structure called the **layered dag.** In this new data structure the separating chains built into a binary tree by Lee and Preparata are refined so that (i) Once a point has been discriminated against a chain, it can be discriminated against a child of that chain with **constant** extra effort and (ii) Overall storage only doubles. The

layered dag simultaneously attains S = O(n) and Q = O(log n). An additional insight allows the dag to built in O(n) time.

Monotone polygons, subdivisions and vertical ordering

A subset of the plane is said to be monotone if its intersection with any line parallel to the y axis is a single interval (possibly empty). A subdivision is said to be monotone if all its regions (polygons) are monotone and it has no vertical edges.

A subdivision that is not monotone can be converted into one by the process of **regularization.** This esentially consists of sweeping the plane and inserting edges between two vertices - one that did not have any right going edges and the other that did not have any left going edges or vice versa. The regularization proces takes O(n log n) time. For more information on the process of regularization please refer to the original paper.

A region A is said to be above B if for every pair of vertically aligned points (x, y_a) of A and (x, y_b) of B we have $y_a >= y_b$, with strict equality holding atleast once. It is written as A >> B.

Separators

A separator for a subdivision is a polygonal line s, consisting of vertices and edges of the plane, with the property that it meets every vertical line at exactly one point. Since s extends from x=-infinity to x=+infinity, any element of the subdivision that is not part of s is either above or below it. The elements of s have pairwise disjoint projections on the x-axis, and so can be ordered from left to right; the first and last elements are infinite edges.

A **complete family of separators** for a monotone subdivision with n regions is a sequence of n-1 separators $s_1 << s_2 << \ldots << s_{n-1}$. If the plane admits a complete family of separators, its regions can be enumerated as $R_0, R_1, \ldots, R_{n-1}$ in such a way that $R_i << s_j$ if and only if i < j; for example:

$$R_0 << s_1 << R_1 << s_2 << \ldots << s_{n-1} << R_{n-1}.$$

Given a complete family of separators and an enumeration of the regions we denote by **index(R)** the index of a region R in the enumeration. Then $s_{index(R)} << R << s_{index(R)+1}$.

We can now prove that every monotone subdivision admits a complete family of separators.

Proof: Let $R_0, R_1, \ldots, R_{n-1}$ be a linear ordering of the regions of the plane that is compatible with the << relation, ie. $R_i << R_j$ only if i < j. For i = 1, 2, ... , n-1, let s_i be the collection of all edges and vertices that are on the frontier between regions with indices < i and regions with indices >=i

Now if a verticle line, l, is drawn, it is easily seen that there is exactly one point on l that is on the frontier between a region with index < i and a region with index >= i, that is, on s_i.

Clearly, the elements of s_i have disjoint x-projections, and therefor can be ordered form left to right; they must be alternately edges and verticesss, the first and last being infinite edgesss. To prove that s_i is a separator, it remains only to show that s_i is connected; if it were not the case, we would have some vertex v of s_i that is distinct from, but has the same x-coordinate as, one endpoint of an adjacent edge e of s_i. But then we would have e << R << v (where R is the region between the edge e and v), which contradicts the construction of s_i. Therefore, each s_i is a separator, and s_1, s_2, ... , s_{n-1} is a complete family of them.

Gaps and chains

Suppose an edge is common to a number of separators s_{i+1} to s_j. While creating a binary search tree containing the separators as its nodes, it suffices to store the edge as part of separator s_k where k is the least common ancestor of i and j. This is the highest node in the tree whose separator contains the edge e. Only edges assigned to s_k are actually stored in such a structure. In general these will form a proper subset of all the original edges of s_k so between successive stored edges of s_k there may be gaps. Actually, it may happen that all the edges of s_k are stored higher up in the tree, so that s_k is reduced to a single gap, extending from x=-infinity to x=+infinity. The ordered list of stored edges and gaps corresponding to separator sk will be termed the chain c_k.

A faster point location method

We refine the chains so that we produce for each chain c_k a list L_k of x-values, defining a partitioning of the x-axis into x-intervals. Each such interval of L_k overlaps the x-projection of exactly one edge or gap of c_k and at most two x-intervals of the lists $L_{l(k)}$ and $L_{r(k)}$.

The lists L_k and their interconnections can be conveniently represented by a linked data structure that is called the **layered dag**. This is a directed acyclic graph whose nodes correspond to tests of three kinds: **x-tests, edge tests, and gap tests.**

An x-test node t contains the corresponding x-value of L_k, denoted by $x_{val(t)}$, and two pointers left(t) and right(t) to the adjacent edge

or gap nodes of L_k. An edge or gap test node t contains two links down(t) and up(t) to appropriate nodes of $L_{l(k)}$ and $L_{r(k)}$. In addition, an edge test contains a reference edge(t) to the edge of c_k whose projection covers the x-interval represented by t. A gap test node contains instead the chain number chain(t) = k. (Please see figure 3(a) and 3(b) for a visual description of the different kinds of nodes).

The properties of the refined lists ensure that the x-interval of L_k corresponding to an edge or gap test t covers either one x-interval I of $L_{l(k)}$, or two such intervals I1, I2 separated by some element x_k of $L_{l(k)}$. In the first case, down(t) points to the edge or gap test of $L_{l(k)}$ corresponding to the interval I; in the second case, down(t) points to the x-test corresponding to the separating abscissa x_k. Similarly, the link up(t) points to a node of $L_{r(k)}$ defined in an analogous manner. Again please refer figure 3(a) and 3(b).

The layered dag contains a distinguished node, root, where the point location search begins. This node is the root of a balanced tree of x-tests whose leaves are the edge tests corresponding to the list for the root node of T (T being the binary search tree of separators).

The point location algorithm is as follows:

Algorithm **FAST_POINT_LOCATION_IN_A_MONOTONE_SUBDIVISION**

This algorithm takes as input the root node of the layered dag and the number n of regions. Its output is placed in the variable loc.

1. Set i = 0, j = n-1, t = root.
2. While i < j do:
   {At this point we know p is above the separator s_i and below one of the separators $s_{i+1}, \ldots, s_j$ (or in some edge case, p is on a vertex between two of these regions). The variable t points to a test node in the layered dag, which together with its descendants will allow us to locate the point p among those regions.}
   3. If t is an edge test then let e = edge(t) and do:
      {At this point we know the projection of point p on the x-axis(denoted as px) lies within the projection of the edge e on the x-axis.}
      4. If p is on e, set loc = e and terminate the algorithm.
      5. If p is above e, set t = up(t) and i = index(above(e)). Else set t = down(t) and j = index(below(e)).
   6. Else if t is an x-test then do:
      {The following x-test routes us to the appropriate edge of the next chain we need to test against.}
      7. If px <= xval(t) then t = left(t) else t = right(t).
   8. Else t is a gap test; do

{We have already compared p against the appropriate edge of the chain of the gap test. We just need to reconstruct how that comparison went.}

9. If j < chain(t) then t = down(t) else t = up(t).

10. Set loc = R_i and terminate the search.

Analysis

At each iteration, beginning at step 2, we descend one level down the tree, so we have O(log n) iterations. Once a point has been discriminated against a chain it takes only O(1) time to discriminate the point against an edge of that chain. Therefore, total time is O(log n).

Conclusions and applications

This technique introduces a new data structure, the layered dag, which solves the point location problem for a monotone subdivision in the plane in optimal time and space. The layered dag can be built from standard representations in linear time. The advantage of the layered dag is that:

* it admits a simple, practical implementation, and
* it can be extended to subdivisions with curved edges.

3. A new approach to planar point location

Introduction

This paper presents a **practical** point location algorithm. The query time = O(log n), preprocessing = O(n log n) and the storage complexity = O(n log n). This technique could be viewed as an evolution of the slab method of Dobkin and Lipton. However, the method rests crucially on the observation that each edge of the graph can be decomposed uniquely into O(log n) fragments (as opposed to partitioning an edge into O(n) fragments in the slab method approach.

Preliminaries

This section deals with one of the basic data structures of comutational geometry namely, the **doubly connected edge list (DCEL)**. We are quite familiar with this data structure and so we will not explain it further.

The next and primary objective is to obtain from G a partial ordering relation, <, on the set of edges, E, as follows: for e_1, e_2 that belong to E, $e_1 < e_2$ means that there is a horizontal line l intersecting both e_1 and e_2 such that the intersection of l with

e_1 is to the left of that with e_2. The relation < can be obtained by a procedure analogous to "regularization" used for creating a monotone subdivision (For more information please refer to the text book for regularizing a monotone subdivision). This takes O(n log n) time.

A **topological sorting**, *P*, of < will be called a **consistent ordering** of the edges of set E. (For any horizontal line l, the left-to-right sequence of the edges intersected by l is a subsequence of *P*). This can be obtained by using a standard topological sorting techniques as described by Knuth (The Art of Computer Programming, p. 262). This requires O(n) time. Please refer figure 4(a) and 4(b).

Definition and construction of the search structure

The search data structure is a tree, *K*, which can be produced for a graph G. In the construction of *K* use is made of the list *P* previously obtained, and of an auxiliary structure, a **"segment tree"**, T(1, n). Again, we are quite familiar with the construction of a search tree and therefore no further detail will be given regarding this.

For any node *v* of T(1, n), we let M[*v*] = floor((B[*v*]+E[*v*])/2), where B[*v*] is the beginning and E[*v*] is the end of the interval associated with a node *v* of the segment tree, and call slab(*v*) the plane strip comprised between y = B[*v*] and y = E[*v*]; a segment *e*, with extreme ordinate r and s (r < s), is said to **span** slab(*v*) if r = B[*v*] and s = E[*v*]; slab(v_1) and slab(v_2) are said to be companion if v_1 and v_2 are siblings in T(1, n).

In *K* we have two types of nodes, with different graphical representations: *V*, a *V*-node or "horizontal node", is associated with a horizontal line and has an ordinate Y[.] as discriminator; *O*, an *O*-node or "segment node", is associated with a straight line segment *e* and has as a discriminator a linear function f[*e*] of x and y such that f[*e*] = 0 is the equation of the line containing *e*.

Each call of the following algorithm which constructs the tree *K* processes one slab. Specifically, for some node *v* in T(1, n), it accepts the left-to-right sequences S of the segments which either span or are contained in slab(*v*) and organizes them in a search tree. Thus *K* is built by TREE(*P*, root(T(1, n)) where *P* is the previously defined consistent ordering of the edges of G, structured as a queue, and TREE(S, v) is the following recursive procedure (where S, S1, S2 and U are queues).

```
procedure TREE(S, v)
begin
  if (S = {}) then U <- {}
              else S1 <- S2 <- U <- {}
```

```
  while S <> {} do
    begin
      e <= S     /* remove from set S and assign to e */
      if ((B[v] < B[e]) or (E[e] < E[v]))
      then
        /* e does not span slab(v) */
        begin
          if B[e] < M[v]
          then
            S1 <= e       /* Add to set S1 the edge e */
          if M[v] < E[e]
          then
            S2 <= e       /* Add to set S2 the edge e */
        end               /* queues S1 and S2 are being built */


      if (B[e] <= B[v]) and (E[v] <= E[e]) or (S = {})
      then
        /* e either spans slab(v) or is last term in S */
        begin
          if (S1 U S2 <> {})
          then
            /* the trapeze is nonempty */
            begin
              w <- new horizontal node of K
              Y[w] <- M[v]
              LTREE[w] <- TREE(S1, LSON[v])
              RTREE[w] <- TREE(S2, RSON[v])
              /*
                 The segments in a trapeze are organized by joining
                 together th structures corresponding to companion
                 slab
              */
              U <= w  /* Add to set U horizontal node w */
            end
    end  /* end of while */
    U* <- BALANCE(U)
    return
end
```

The procedure **BALANCE** takes a sequence of terms, which are either trees or segments, and arranges them in a conviniently balanced tree.

Analysis of procedure TREE:

It is convenient to charge the work to the individual edges of G. Since there are O(n) edges in G (By Euler's theorem on planar graphs) and each edge is charged O(log n) work. the generation of all "segment nodes" of T(1 , n) uses work O(n log n) globally.

Each *V*-node is produced and processed in steps 9 and 13. It can be shown that the number of *V*-nodes is O(n log n). Hence the total work used by procedure TREE to produce tree *K*, except for the work attributable to subroutine BALANCE, is O(n log n). An analysis yields the running time for procedure BALANCE to be O(n log n). Thus the total time for the procedure TREE is O(n log n).

Point Location

To locate a point $P_0 = (x_0, y_0)$ in the planar subdivision induced by G, we use *K* as a binary search tree. With each *O*-node of *K* which has one or no descendant we append one or two leaves, refer figure 5, respectively, and with each such leaf we associate the identifer of a plane region (bordering with the edge associated with the parent *O*-node). **The point location proceeds as follows:** at each node V of *K*, we choose a branch; if V is a *V*-node, by comparing y_0 with Y[V]; if V is an *O*-node, by testing the sign of $f(x_0, y_0)$, where f(x, y) is the discriminant function of V. Thus we trace a unique path from the root to a leaf, at which stage the point location is completed. By the preceding discussion this process uses a number of comparisons bounded by the depth of *K*, ie., O(log n).

Comments and applications

Thus we see that the point location is simply done in time O(log n) using a search structure which can be stored in O(n log n) space. It is conceivable that the simple approach presented in this paper could be further refined to achieve O(n) storage while maintaining O(log n) search time.

We now mention **two applications** of the given method:

1. **Fixed-radius near neighbor searching** which involves finding all points of a set F in the plane which are within some fixed radius r of "query point" .

2. **Maxima testing in three dimensions** which involves deciding whether a target point p in the set dominates all the other points in that set.

4. Fully Dynamic Technique for Point Location and Transitive Closure in Planar Structures.

Introduction

This technique differs from the previous three techniques in that the planar subdivision can be modified by insertions and deletions of points and segments. In this paper monotone subdivisions are

considered. The topological underpinning of a monotone subdivision is a planar *st-graph*.

A planar *st-graph* admits two total orders (referred to as **leftist** and **rightist**) on the set V U E U F, where V, E, F are respectively the sets of vertices, edges and faces. It is shown that by dynamically maintaining these two orders on the planar *st-graph* we are able to dynamically maintain the monotone subdivision whose underlying topology is represented by this graph.

Planar st-graphs

A planar *st-graph* is a planar acyclic graph digraph *G* with exactly one source (vertex without incoming arcs), *s*, and exaclty one sink (vertex without outgoing arcs), *t*, which is embedded in the plane so that *s* and *t* are on the boundary of the external face.

Some important properties of planar *st-graphs* are expressed by the following lemmas:

Lemma 1. [2] For every vertex *v* of *G*, the incoming (outgoing) edges appear consecutively around *v*.

Lemma 2. [2] For every face *f* of *G*, the boundary of *f* consists of two directed paths with common origin and destination.

Lemma 3. Let *G* be a planar st-graph with *n* vertices. There exists two total orders on the vertices of *G*, denoted $<_L$ and $<_R$, such that there is a directed path from *u* to *v* if and only if $u <_L v$ and $u <_R v$. Furthermore, orders $<_L$ and $<_R$ can be computed in O(n) time.

Please refer figure 6(a) and 6(b) for some basic definitions regarding planar *st-graph*.

The authors then define some primitive operations on the elements of the planar *st-graph* (The elements being the vertices, edges and faces of the graph). These primitives are *HIGH*(x), *LOW*(x), *LEFT*(x) and *RIGHT*(x). These primitives define the respective orientations of the elements with respect to each other. In particular we say that *x is below y*, denoted x|y, if there is a path in *G* from *HIGH*(x) to *LOW*(y). Also, we say that *x is to the left of y*, denoted x->y, if there is a path in the *dual of G* (refer figure 6(c)) from *RIGHT*(x) to *LEFT*(y). We define the *left-sequence (right-sequence)* of *G* as the sequence of elements of *G* sorted according to $<_L$ ($<_R$) order. For example, the left-sequence of the graph of figure 6(a) can be represented by $f_0v_0e_1Av_3e_6v_5e_9f_2B$, where $A=f_1e_2v_1e_5$ and $B= e_7f_3e_4f_4e_3v_2e_8v_4e_{10}v_6f_5$.

On-line Maintainance of a Planar *st-graph*

The update operations on a planar *st-graph* are *INSERT*, *DELETE*,

EXPAND, and *CONTRACT*. An example of the transformation of the leftist order $<_L$ as a consequence of an INSERT operation is defined as: $v <_L f <_L u$: A *v* B *f* C *u* D -> A f_1 C *u e v* B f_2 D and is shown in figure 7.

Dynamic Planar Point Location

The planar *st-graph* framework is specialized to the classical problem of planar point location in a **monotone** subdivision. A monotone subdivision *R* is a partition of the entire plane into monotone polygons, called the *regions* of *R*. Also, we know from an earlier technique that a monotone subdivision admits a complete family of separators. The following primitive operations are defined for the update of a monotone subdivision: *INSERTPOINT*, *REMOVEPOINT*, *INSERTCHAIN*, *REMOVECHAIN* and *MOVEPOINT*.

Two regions r_1 and r_2 with $r_1 <_L r_2$ are vertically consecutive if $r_1 | r_2$ and there is no region $_r$ such that $r_1 <_L r <_L r_2$. **Lemma:** If r_1 and r_2 are two vertically consecutive regions of a monotone subdivision *R*, then the monotone chain from *HIGH*(r_1) to *LOW*(r_2) is unique and is called a ***channel***. We can merge in this fashion any sequence of vertically consecutive pairs and we have:

Clusters are recursively defined as follows: (1) An individual region *r* is a cluster; (2) Given two vertically consecutive clusters X_1 and X_2; with $X_1 | X_2$, there union is a cluster *X*. A **maximal cluster** X is one which is not properly contained in any other cluster. The unique subdivision resulting by forming all maximal clusters of *R* is denoted *R'* and this is found to be regular.

Data Structures

The search data structures is very complicated and we present below just a brief overview of it:

It consists of a main component, called the *augmented separator-tree* ***T***, and an **auxiliary component.** The augmented separator-tree has a primary and a secondary structure. The *primary structure* is a separator tree for *R'*, ie., each of its leaves is associated with a region of *R'* (a maximal cluster of *R*, ie. *X*), and each of its internal nodes is associated with a separator of *R'*. The *secondary structure* is a collection of lists, each realized as a search tree. The auxiliary component consists of balanced search trees TL and TR respectively associated with the orders $<_L$ and $<_R$ on V U E U F, and of a *dictionary*, which contains the lists of the vertices, edges, and regions of *R*, each sorted according to the lexicographic order of their names.

Query

The point location search for a query point q consists of tracing a path from the root to a leaf *X* of T. At each internal node we discriminate *q* against a separator, and proceed to the left or right child depending upon whether *q* lies to the left or right of that separator. We then discriminate the point to within an edge of the separator. **Thus we see that the algorithm that we use is the conventional old technique of Lee and Preparata.** The time complexity of the query operation is $O(\log^2 n)$.

Update

The analysis requires consideration of the subsequence of the left-sequence of *G* formed by the regions of *R*, referred to as the *region-sequence*.

We show below how a chain can be inserted into the subdivision. The region sequence before the update is given by the string:

$L_1 l_1 - l_2 L_2 r R_1 g_1 - g_2 R_2$ where - represents a channel.

If we now apply the transformation as described above, we have the obvious correspondence:

A <-> $L_1 l_1$, B <-> $l_2 L_2$, C <-> $R_1 g_1$, D <-> $g_2 R_2$.

Please refer figure 8.

Conclusions:

This technique presents a fully dynamic data structure for point location queries in a monotone subdivision. This technique could be specially useful for solving such queries in a dynamically changing terrain.

Figures

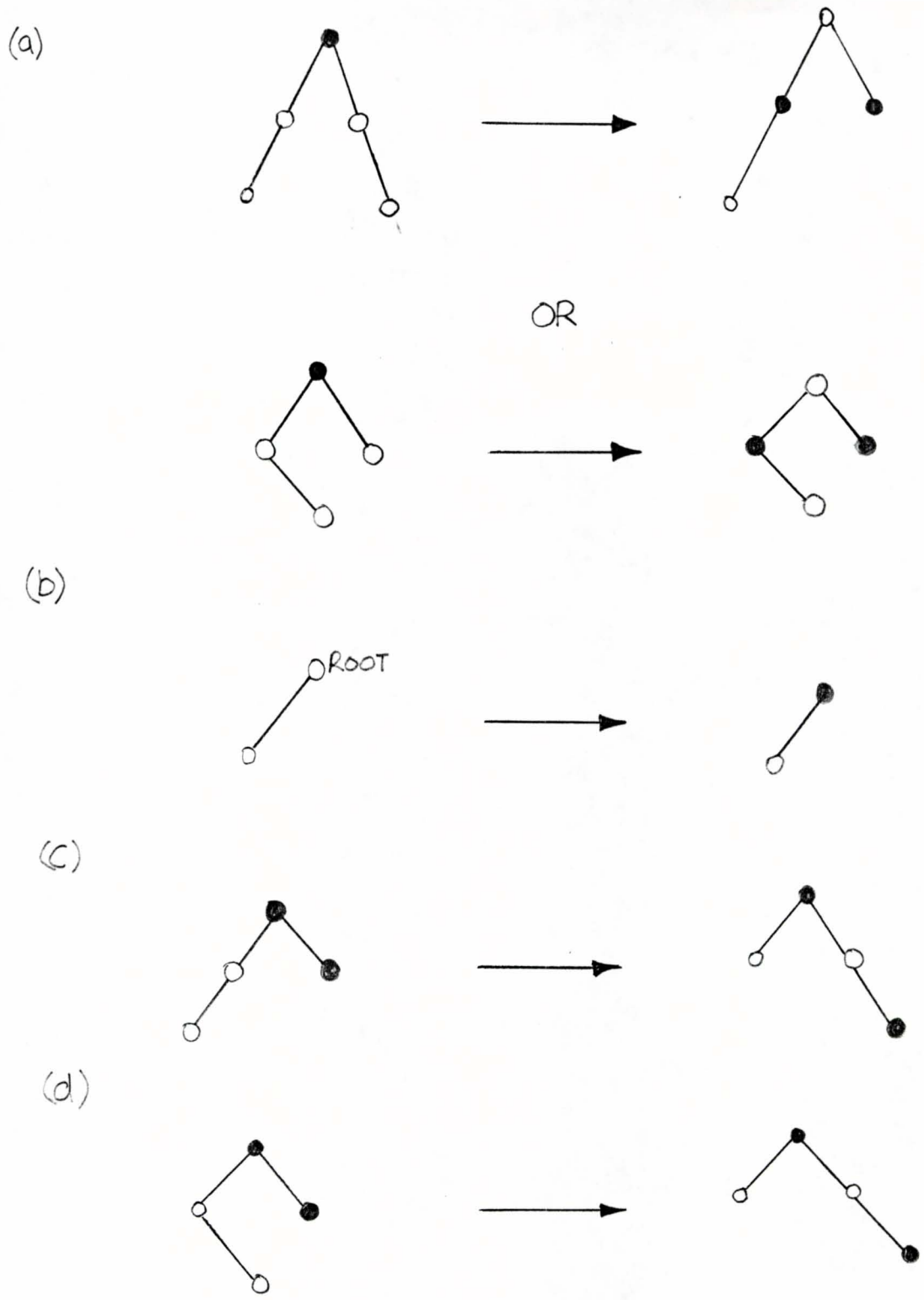


Figure 1. The Rebalancing Transformations in Red-Black Tree Insertion. Symmetric cases are omitted. Solid nodes are black; hollow nodes are red. All unshown children of red nodes are black.

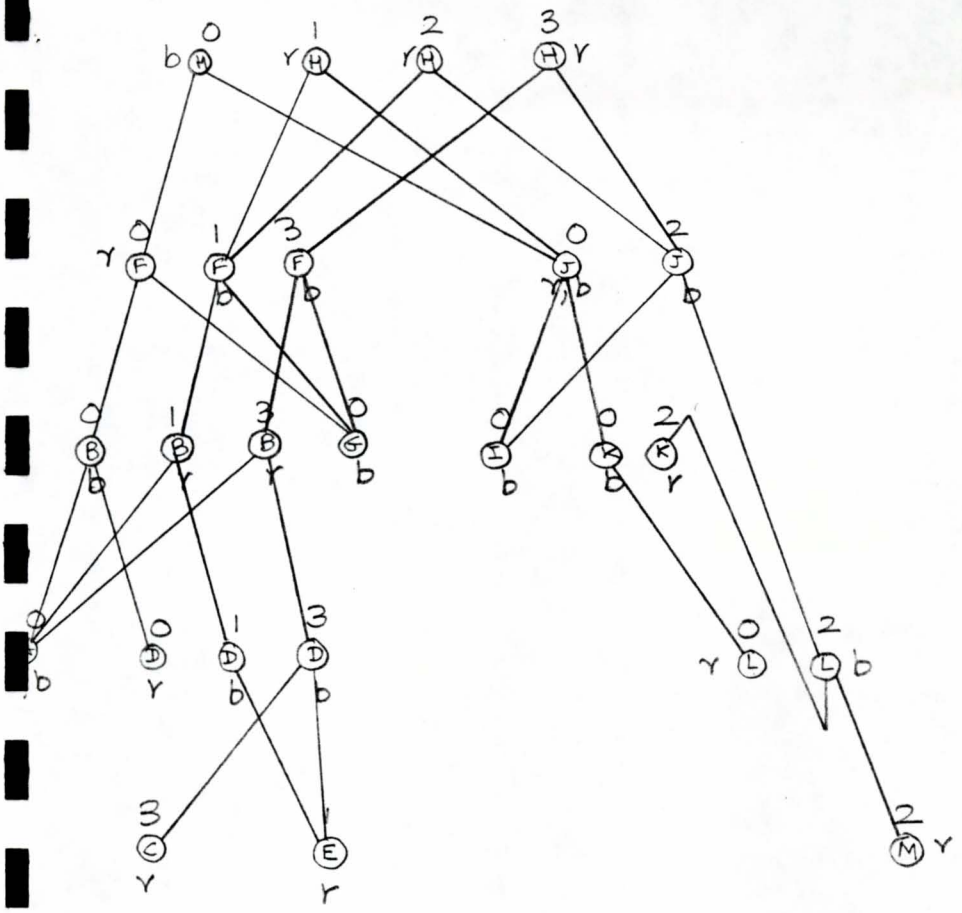


Figure 2(a) A persistent Red Black tree with Path Copying. The initial tree, existing at time 0, contains A, B, D, F, G, H, I, J, K. Item E is inserted at time 1, M at time 2, and C at time 3. The nodes are labeled by their colors, r for red, b for black. The nodes are also labeled by their time of creation. All edges exit the bottom of nodes and enter the tops.

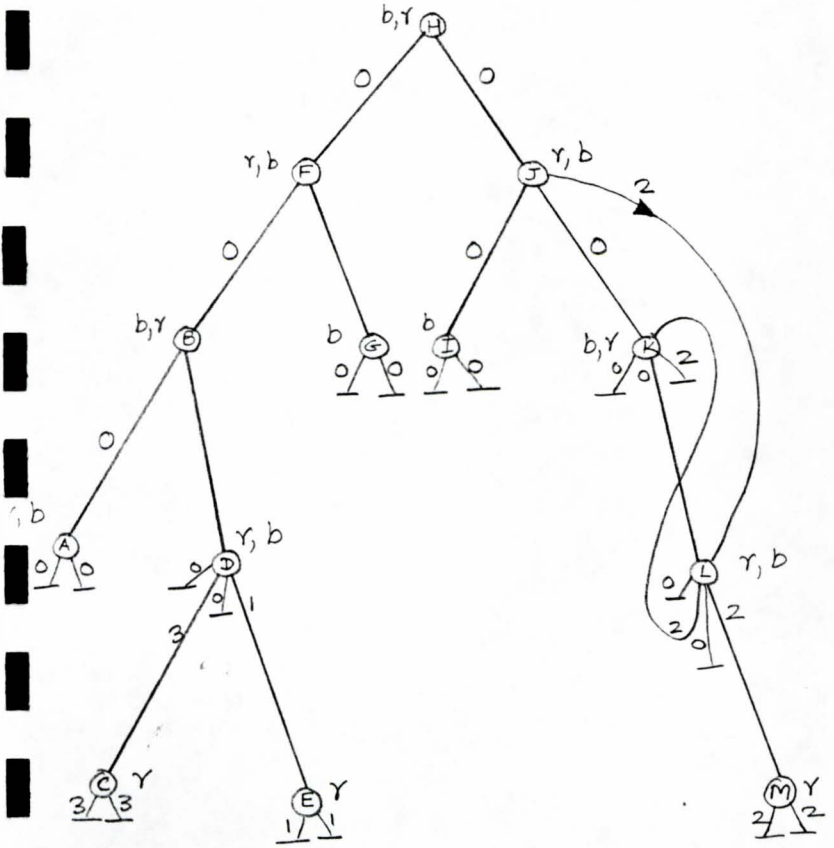


Figure 2(b) A persistent Red-Black tree with No Node Copying. The initial tree and insertions are as in fig. 2(a). Edge labeled by time of creation, nodes with their colors. Connections to horizontal lines denote null pointers.

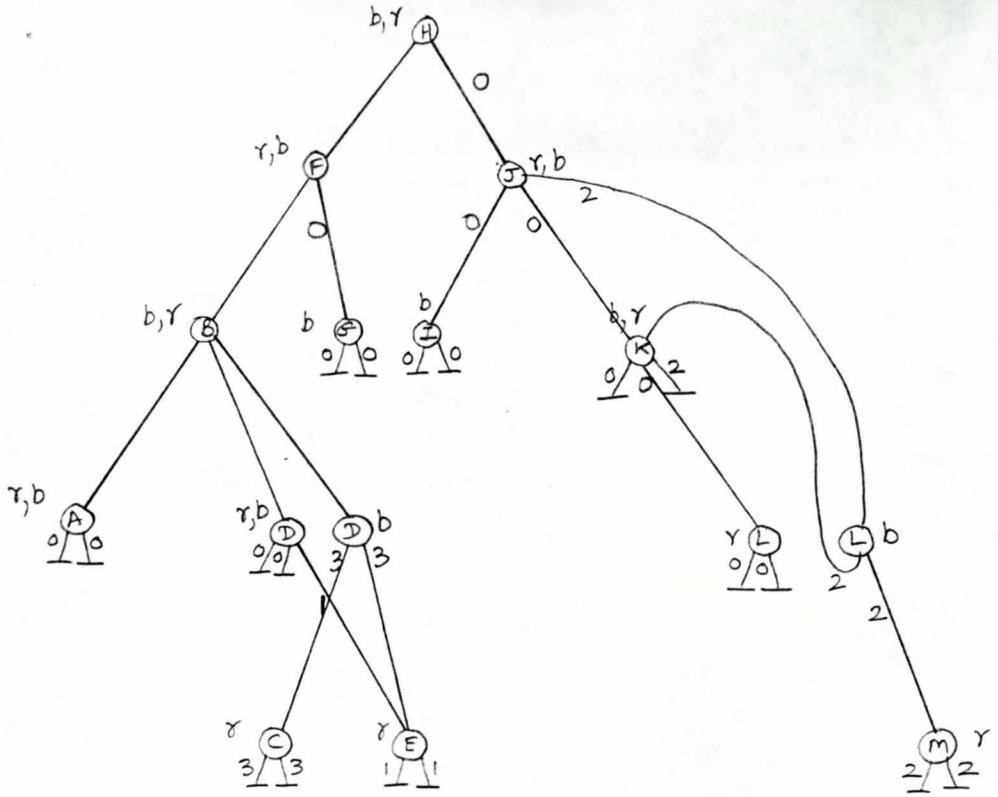
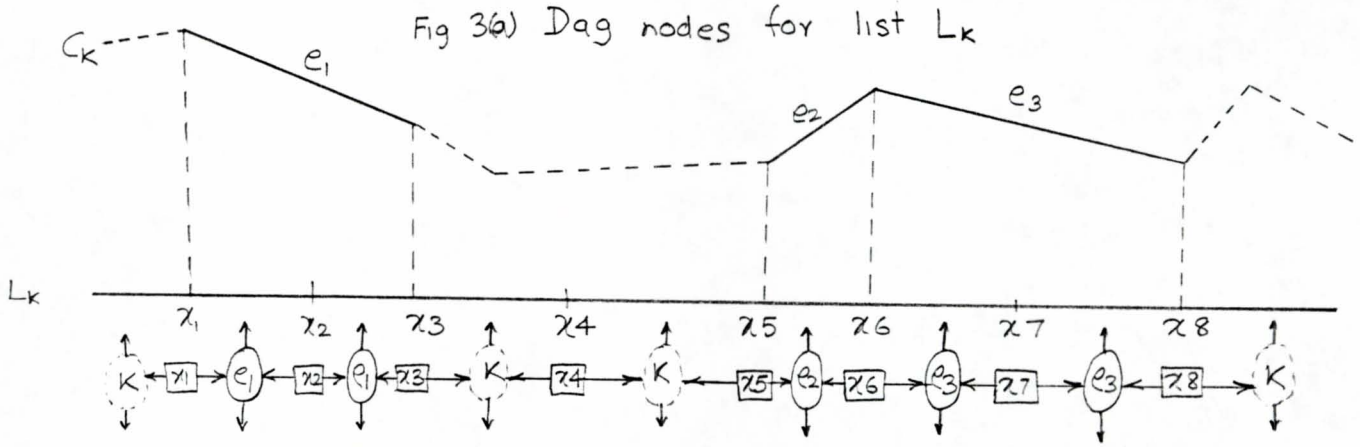


Figure 2 (c). A Persistent Red Black tree with Limited Node Copying assuming each node can hold one extra pointer.

Fig 3(a) Dag nodes for list L_K



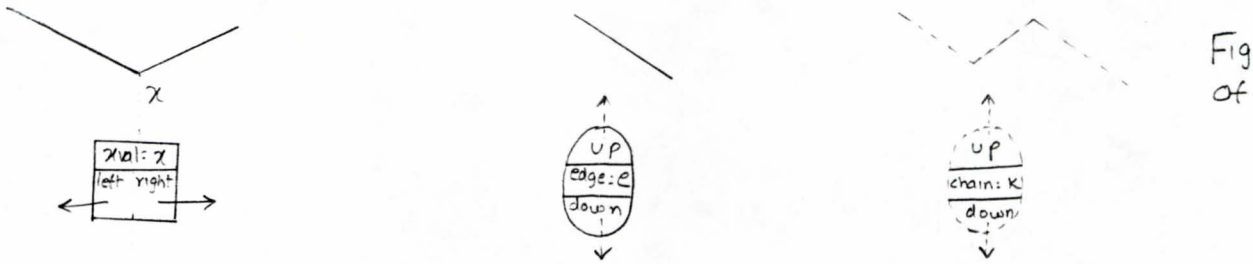


Fig 3(b). Nodes of the Dag.

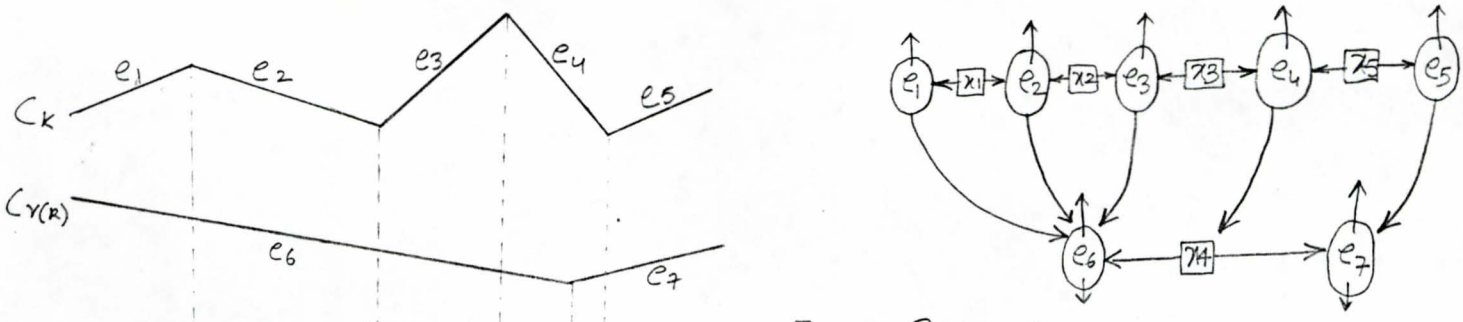


Figure 3(c). Convergence in the dag

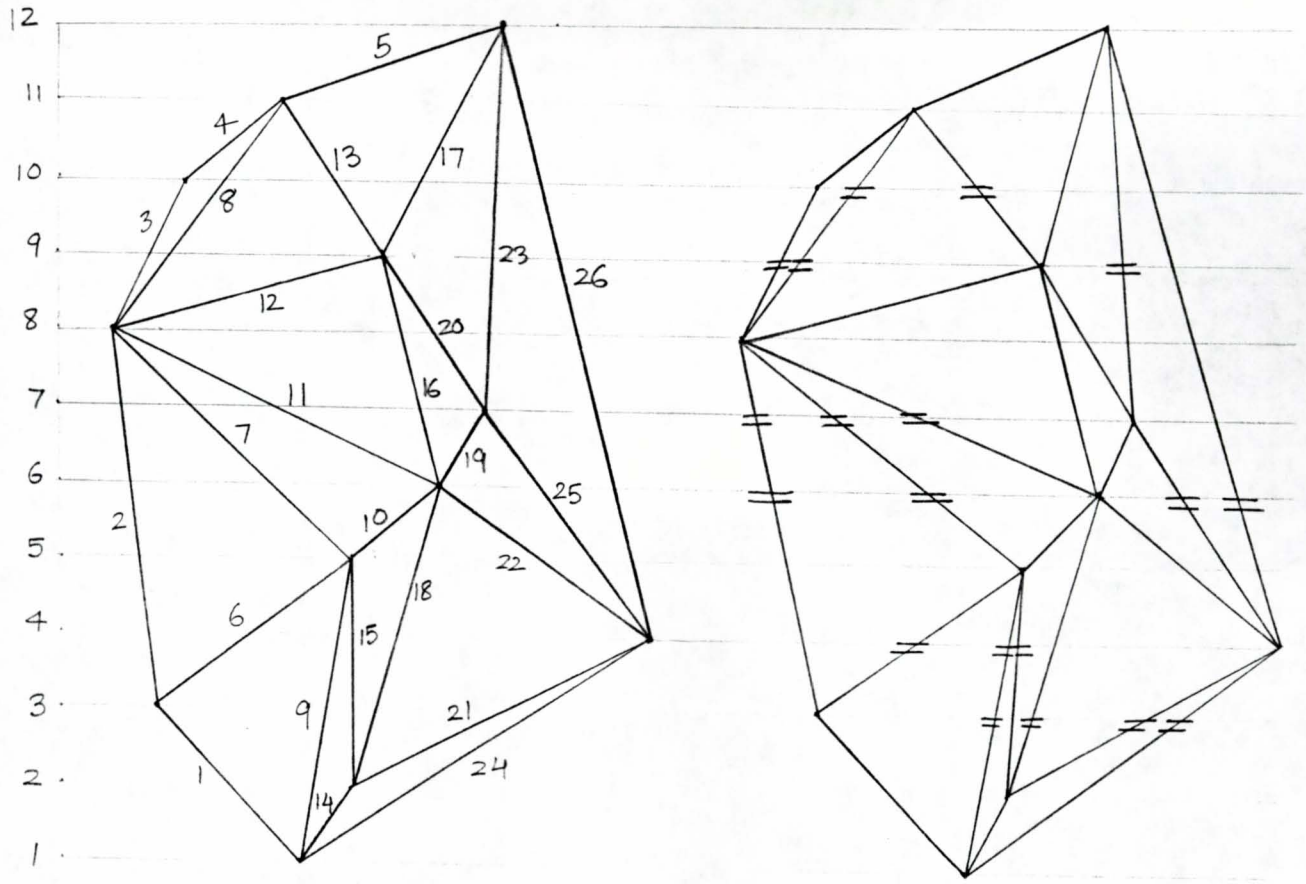


Figure 4(a) A graph G and a corresponding consistent ordering P (indices in P are shown as labels); (b) edge segmentation as induced by the segment tree T(1, 12).

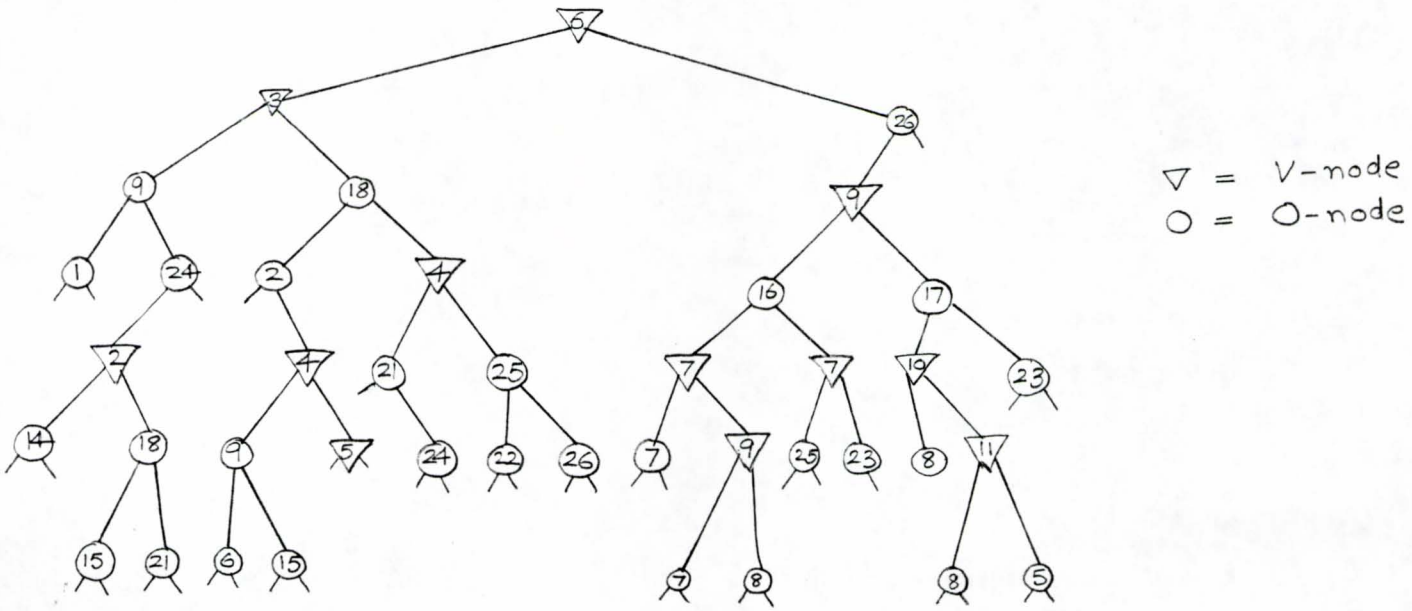


Figure 5. The search data structure K for the graph of figure 4 above.

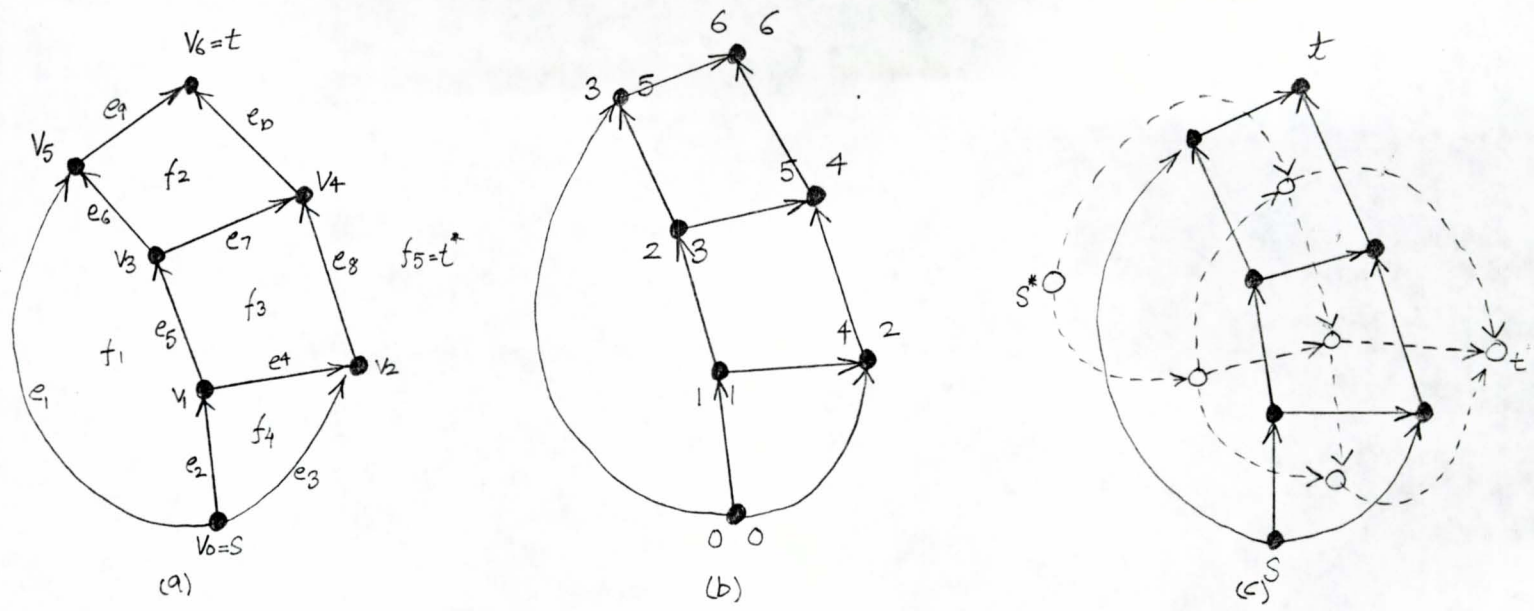


Figure 6 (a) Example of a planar st-graph G. (b) Orders $<_L$ and $<_R$ on the vertices of G; (c) G and its dual graph G'.

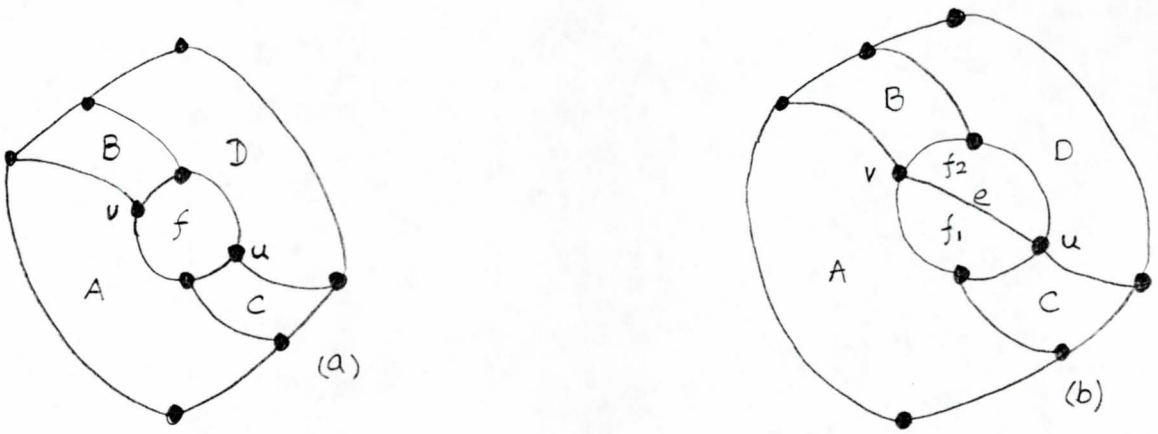


Figure 7 (a) Graph before insertion; (b) Graph after insertion of edge e across face f.

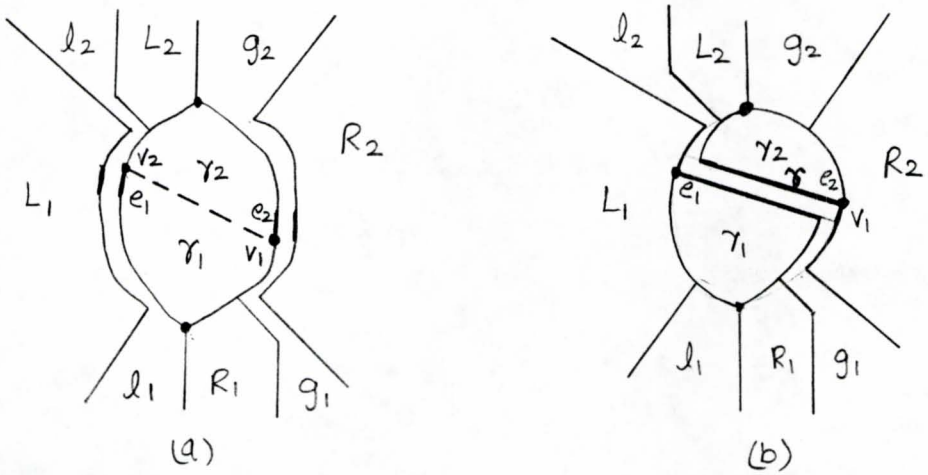


Figure 8 (a) Canonical partition of subdivision R' with reference to region r and vertices v1 & v2, and (b) the restructured subdivision after the insertion of chain γ between v1 & v2.

```
/*****************************************************************************
*                                                                            *
*                          PHYSICAL MODELING                                 *
*                                                                            *
*  This file contains an implementation of a physical modeling problem, the  *
*  "Springs and Sticks problem" in which the behavior of a series of sticks, *
*  having masses attached to their ends, is shown.  The Penalty Method is    *
*  used for calculating physical forces on the sticks which shape their      *
*  behavior.                                                                 *
*                                                                            *
*  The system consists of four modules which vary the state variables        *
*  associated with each stick, namely, its position, the forces acting on    *
*  its ends, its linear and angular accelerations.  Euler integration (using *
*  the time step which has been set up for the simulation, called dt here)   *
*  is used to calculate new velocities, positions and orientation in space   *
*  of the sticks.  The new positions are then used to calculate new          *
*  forces which have thus developed in the system and the cycle repeats.     *
*                                                                            *
*  This implementation shows the critically damped version of the model.     *
*  Other versions (underdamped, overdamped and undamped) can be easily       *
*  generated by varying the damping constant (k2).                           *
*                                                                            *
*  Written by : Sumeet Rajput                                                *
*       Date : 4/10/93                                                       *
*                                                                            *
*****************************************************************************/

/*** INCLUDES ***/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>
#include <dos.h>

/*** CONSTANTS ***/
#define MAX_Y             479.0    /* Max. Y-coordinate on the screen      */
#define RADIUS             20.0    /* Half-length of stick (its radius)    */
#define MASS              100.0    /* mass of masses attached at the ends  */

#define NUM_OF_STICKS         7    /* Number of sticks in the simulation.  */
                                   /* Note that the first and last sticks  */
                                   /* are "dummy" sticks whose ends are    */
                                   /* used only for force calculations on  */
                                   /* the rest of the "real" sticks.  So,  */
                                   /* n-2 sticks are actually involved in  */
                                   /* the simulation.                      */

#define dt                  1.0    /* Delta time between simulation loops  */
#define k1                 10.0    /* Spring constant                      */

#define k2    (2*sqrt(k1*MASS))    /* Damping constant (Critically damped) */
                                   /* k2 < 2*sqrt(k1*MASS => Underdamped   */
                                   /* k2 > 2*sqrt(k1*MASS => Overdamped    */
                                   /* k2 = 0              => Undamped      */

#define g                   0.4    /* constant for displaying gravity      */
#define HALF_PI           1.571    /* Approx. value of PI/2.0              */

/*** TYPES ***/
typedef struct                     /* Definition of a point */
{
    float x;
    float y;                       /* Its x coordinate */
}POINT;                            /* Its y coordinate */

typedef struct                     /* Stick information */
{
    /* Positional and force information */
    POINT     a;                   /* Current "a" of stick.  A stick has   */
                                   /* endpoints a and b.                   */

    POINT     b;                   /* Current "b" of stick                 */
    POINT     c;                   /* Center point of the stick            */
    POINT    ra;                   /* Radius vector towards "a"            */
    POINT    rb;                   /* Radius vector towards "b"            */
    POINT    fa;                   /* Force on "a"                         */
    POINT    fb;                   /* Force on "b"                         */
    float theta;                   /* Angle the stick makes with the       */
```

```
                                /* positive x-axis                      */

    /* Velocity information */
    POINT     adot;             /* Linear velocity of end "a"           */
    POINT     bdot;             /* Linear velocity of end "b"           */
    POINT     cdot;             /* Linear velocity of center "c"        */
    float thetadot;             /* Angular velocity of the center       */

    /* Acceleration information */
    POINT     cdotdot;          /* Linear acceleration of the center    */
    float thetadotdot;          /* Angular acceleration of the center   */
}STICK;

/*** STATIC DATA ***/
STICK stick[NUM_OF_STICKS];

/*** LOCAL PROTOTYPES ***/

/*** FUNCTIONS ***/

/*****************************************************************************
*                                                                            *
*  This function initializes the graphics system.                            *
*                                                                            *
*****************************************************************************/
void InitializeGraphics(void )
{
    /* request auto detection */
    int gdriver = DETECT, gmode, errorcode;

    /* initialize graphics mode */
    initgraph(&gdriver, &gmode, "\\borlandc\\bgi");

    /* read result of initialization */
    errorcode = graphresult();

    if (errorcode != grOk)  /* an error occurred */
    {
       printf("Graphics error: %s\n", grapherrormsg(errorcode));
       printf("Press any key to halt:");
       getch();
       exit(1);             /* return with error code */
    }
}

/*****************************************************************************
*                                                                            *
*  Set up initial conditions for the simulation.  Here the initial positions*
*  of the sticks are set to be at the bottom center of the screen.  All      *
*  other state variables are also initialized.                               *
*                                                                            *
*****************************************************************************/
void InitializeSimulation( void )
{
    stick[0].b.x =   0.0;               /* Set "b" for the first stick    */
    stick[0].b.y = 240.0;

    stick[NUM_OF_STICKS-1].a.x = 639.0; /* Set "a" for the last stick     */
    stick[NUM_OF_STICKS-1].a.y = 240.0;

    for(int i=1; i<NUM_OF_STICKS-1; i++) /* Initialize the "real" sticks  */
    {
        stick[i].c.x         = 320.0;   /* Sticks start up lying flat on  */
        stick[i].c.y         =   0.0;   /* top of each other at the       */
        stick[i].theta       =   0.0;   /* bottom of center of the screen */

        stick[i].a.x         = 300.0;
        stick[i].a.y         =   0.0;

        stick[i].b.x         = 340.0;
        stick[i].b.y         =   0.0;

        stick[i].ra.x        = -20.0;
        stick[i].ra.y        =   0.0;

        stick[i].rb.x        =  20.0;
        stick[i].rb.y        =   0.0;

        stick[i].adot.x      =   0.0;
```

```
        stick[i].adot.y      =   0.0;

        stick[i].bdot.x      =   0.0;
        stick[i].bdot.y      =   0.0;

        stick[i].cdot.x      =   0.0;
        stick[i].cdot.y      =   0.0;

        stick[i].thetadot    =   0.0;

        stick[i].cdotdot.x   =   0.0;
        stick[i].cdotdot.y   =   0.0;

        stick[i].thetadotdot =   0.0;
    }

    /* Initialize the forces on all sticks */
    for(i=1; i<NUM_OF_STICKS-1; i++)
    {
        stick[i].fb.x = k1*(stick[i+1].a.x-stick[i].b.x) - k2*stick[i].bdot.x;
        stick[i].fb.y = k1*(stick[i+1].a.y-stick[i].b.y) - k2*stick[i].bdot.y;

        stick[i].fa.x = k1*(stick[i].b.x-stick[i+1].a.x) - k2*stick[i].adot.x;
        stick[i].fa.y = k1*(stick[i].b.y-stick[i+1].a.y) - k2*stick[i].adot.y;
    }
}

/*****************************************************************************
*                                                                            *
* Draw the springs and sticks.                                               *
*                                                                            *
*****************************************************************************/
void DrawSticks( void )
{
    /*  Clear the display */
    cleardevice();

    setcolor(LIGHTMAGENTA);
    setlinestyle(SOLID_LINE, 0, THICK_WIDTH);

    /* Draw the sticks */
    for(int i=1; i < NUM_OF_STICKS-1; i++)
        line(stick[i].a.x, MAX_Y-stick[i].a.y,
             stick[i].b.x, MAX_Y-stick[i].b.y);

    setcolor(CYAN);
    setlinestyle(SOLID_LINE, 0, NORM_WIDTH);

    /* Now draw the springs */
    for(i=0; i < NUM_OF_STICKS-1; i++)
        line(stick[i].b.x,   MAX_Y-stick[i].b.y,
             stick[i+1].a.x, MAX_Y-stick[i+1].a.y);
}

/*****************************************************************************
*                                                                            *
*  Module 1 for the simulation.                                              *
*                                                                            *
*  This function calculates the end points of the sticks from their center   *
*  position and their orientation w.r.t. the x-axis.  It also computes       *
*  radius vectors for use in determining forces in another module.  Before   *
*  calculating the new endpoints of the stick, the old ones are saved in     *
*  temporary variables.  The new end points are then computed and the        *
*  difference in the new and old end points is a measure of velocity of the  *
*  end points of the stick.  The velocities are required for damping         *
*  purposes                                                                  *
*                                                                            *
*****************************************************************************/
void Module1( void )
{
    POINT old_a, old_b;

    /* Do for all real sticks */
    for(int i=1; i < NUM_OF_STICKS-1; i++)
    {
        /* First copy a's and b's into old_a's and old_b's */
        old_a.x = stick[i].a.x;
        old_a.y = stick[i].a.y;
        old_b.x = stick[i].b.x;
```

```
        old_b.y = stick[i].b.y;

        /* Determine new position */
        stick[i].a.x = stick[i].c.x - RADIUS * cos( stick[i].theta );
        stick[i].a.y = stick[i].c.y - RADIUS * sin( stick[i].theta );
        stick[i].b.x = stick[i].c.x + RADIUS * cos( stick[i].theta );
        stick[i].b.y = stick[i].c.y + RADIUS * sin( stick[i].theta );

        /* Calculate the velocities of the endpoints */
        stick[i].adot.x = ( stick[i].a.x - old_a.x ) / dt;
        stick[i].adot.y = ( stick[i].a.y - old_a.y ) / dt;
        stick[i].bdot.x = ( stick[i].b.x - old_b.x ) / dt;
        stick[i].bdot.y = ( stick[i].b.y - old_b.y ) / dt;

        /* Calculate radius vector and center */
        stick[i].ra.x = stick[i].a.x - stick[i].c.x;
        stick[i].ra.y = stick[i].a.y - stick[i].c.y;
        stick[i].rb.x = stick[i].c.x - stick[i].a.x;
        stick[i].rb.y = stick[i].c.y - stick[i].a.y;

        stick[i].c.x = ( stick[i].a.x + stick[i].b.x ) / 2.0;
        stick[i].c.y = ( stick[i].a.y + stick[i].b.y ) / 2.0;
    }
}

/*****************************************************************************
*                                                                            *
*  Module 2 for the simulation.                                              *
*                                                                            *
*  This function determines linear acceleration of the center of each stick.*
*  The acceleration is obtained from the classic force equation:             *
*                                                                            *
*    F = M * A  where F is the force acting on a body                        *
*                     M is its mass and                                      *
*                     A is its acceleration                                  *
*                                                                            *
*  For each stick F equals the sum of the forces on both its end             *
*                 M is the sum of the masses attached to its ends and        *
*                 A is the required linear acceleration of the center        *
*                                                                            *
*****************************************************************************/
void Module2( void )
{
    /* Do for all real sticks */
    for(int i=1; i<NUM_OF_STICKS-1; i++)
    {
        stick[i].cdotdot.x = (stick[i].fa.x + stick[i].fb.x) / (2.0 * MASS);
        stick[i].cdotdot.y = (stick[i].fa.y + stick[i].fb.y) / (2.0 * MASS);
    }
}

/*****************************************************************************
*                                                                            *
*  Module 3 for the simulation.                                              *
*                                                                            *
*  This function determines the angular acceleration of the center of each   *
*  stick.  This is determined as follows:                                    *
*                                                                            *
*  Let R be defined as the radius vector corresponding to an end of a stick  *
*  ie. the vector obtained by going from the center of a stick to its        *
*  endpoint                                                                  *
*                                                                            *
*  Let F be the force acting on that endpoint                                *
*                                                                            *
*  Then Torque T is defined as being the cross product of R and F ie.        *
*    T = R X F.                                                              *
*                                                                            *
*  The angular acceleration, A, for the stick is then given by:              *
*    A = (Torque on end point a + Torque on end point b) / r*r*M             *
*      where r is the radius of the stick and                                *
*            M is the sum of the masses attached to the end points.          *
*                                                                            *
*  In the present case we have:                                              *
*                                                                            *
*    A                 = thetadotdot                                         *
*    Torque on point a = ra X fa                                             *
*    Torque on point b = rb X fb  (ra, rb are radius vectors and fa,fb are   *
*                                  force vectors)                            *
*    r = RADIUS = 20.0 and                                                   *
```

```
*    m = MASS   = 10.0                                                    *
*                                                                         *
*************************************************************************/
void Module3( void )
{
    float cross1, cross2;  /* Temp variables for storing cross products */

    /* Do for all real sticks */
    for(int i=1; i<NUM_OF_STICKS-1; i++)
    {
        /* compute cross products */
        cross1 = (stick[i].ra.x * stick[i].fa.y) - (stick[i].fa.x * stick[i].ra.y);
        cross2 = (stick[i].rb.x * stick[i].fb.y) - (stick[i].fb.x * stick[i].rb.y);

        /* compute angular acceleration */
        stick[i].thetadotdot = ( cross1 + cross2 ) / (2.0*RADIUS*RADIUS*MASS);
    }
}

/*************************************************************************
*                                                                         *
*  Module 4 for the simulation.                                           *
*                                                                         *
*  This function determines penalty forces that are applied to the ends of *
*  the sticks.  The end points are damped by using the velocity at that   *
*  point.  The appearance of gravity is provided by deliberately reducing *
*  the y component of the force on each end point by a fraction of the    *
*  height of the center of the stick.                                     *
*                                                                         *
*************************************************************************/
void Module4( void )
{
    /* Do for all real sticks */
    for(int i=1; i<NUM_OF_STICKS-1; i++)
    {
        stick[i].fb.x = k1*(stick[i+1].a.x-stick[i].b.x) - k2*stick[i].bdot.x;
        stick[i].fb.y = k1*(stick[i+1].a.y-stick[i].b.y) - k2*stick[i].bdot.y
                                                         - g*stick[i].c.y;

        stick[i].fa.x = k1*(stick[i-1].b.x-stick[i].a.x) - k2*stick[i].adot.x;
        stick[i].fa.y = k1*(stick[i-1].b.y-stick[i].a.y) - k2*stick[i].adot.y
                                                         - g*stick[i].c.y;
    }
}

/*************************************************************************
*                                                                         *
*  Euler integrator for linear components.                                *
*                                                                         *
*  This function implements euler integration on the linear acceleration to *
*  get the change in linear velocity.  This is then added to the old linear *
*  velocity to get the current linear velocity.  Another euler integration *
*  on the velocity gets us the change in position.  This is then added to  *
*  the old position to get the current position.                          *
*                                                                         *
*************************************************************************/
void LinearIntegrator( void )
{
    /* Do for all real sticks */
    for(int i=1; i<NUM_OF_STICKS-1; i++)
    {
        /* Get new linear velocity */
        stick[i].cdot.x = stick[i].cdot.x + stick[i].cdotdot.x * dt;
        stick[i].cdot.y = stick[i].cdot.y + stick[i].cdotdot.y * dt;

        /* Get new position */
        stick[i].c.x = stick[i].c.x + stick[i].cdot.x * dt;
        stick[i].c.y = stick[i].c.y + stick[i].cdot.y * dt;
    }
}

/*************************************************************************
*                                                                         *
*  Euler integrator for angular components.                               *
*                                                                         *
*  This function implements euler integration on the angular acceleration to*
*  get the change in angular velocity.  This is then added to the old     *
*  angular velocity to get the current angular velocity.  Another euler   *
*  integration on the angular velocity gets us the change in orientation. *
```

```
 *  This is then added to the old orientation to get the current orientation.*
 *                                                                           *
*****************************************************************************/
void AngularIntegrator( void )
{
    /* Do for all real sticks */
    for(int i=1; i<NUM_OF_STICKS-1; i++)
    {
        /* Get new angular velocity */
        stick[i].thetadot = stick[i].thetadot + stick[i].thetadotdot * dt;

        /* Get new orientation */
        stick[i].theta    = fmod(stick[i].theta + stick[i].thetadot*dt, HALF_PI);
    }
}

/*****************************************************************************
*                                                                            *
*  The main program.                                                         *
*                                                                            *
*  Initializes the graphics and the simulation state variables.              *
*  Repeatedly calls the modules defined above redrawing the status of the    *
*  sticks in each cycle.  The simulation stops when the user presses a key.  *
*  Finally it shuts up the graphics system and exits.                        *
*                                                                            *
*****************************************************************************/
void main( void )
{
    /*  Set up for graphics */
    InitializeGraphics();

    /*  Set up the initial conditions */
    InitializeSimulation();

    /*  Start up new */
    cleardevice();

    /*  Begin the simulation.  Repeat the procedure until the user presses */
    /*  a key.                                                             */
    while( !kbhit() )
    {
        Module1();
        DrawSticks();
        Module4();
        Module2();
        Module3();
        LinearIntegrator();
        AngularIntegrator();
    }

    /* closes down the graphics system */
    closegraph();
}

/*** END OF FILE ***/
```