A LINEAR TIME ALGORITHM FOR THE GENERALIZED CONSECUTIVE RETRIEVAL PROBLEM

by

Paul Dietz
Merrick Furst
John Hopcroft

TR79-386

July, 1979

Department of Computer Science
Cornell University
Ithaca, New York 14853

A LINEAR TIME ALGORITHM FOR THE GENERALIZED CONSECUTIVE RETRIEVAL PROBLEM*

Paul Dietz
Merrick Furst
John Hopcroft

Abstract

The Generalized Consecutive Retrieval Problem (GCRP) is to find a directed tree on n records in which each of k subsets forms a directed path. The problem arises in organizing information for efficient retrieval. A linear time algorithm for the GCRP is given. Further generalization leads to problems that are complete for NP.

*This research was supported in part by the Office of Naval Research under Grant N00014-76-C-0018.

Introduction

In 1976 Booth and Lueker [2] described a linear time algorithm for solving the Consecutive Retrieval Problem originally proposed by Ghosh in [3]. This problem has received a great deal of attention in recent years. Simply stated, an instance of the CRP consists of n records and k subsets of these records. The objective of the algorithm is to produce a linear ordering of the records so that each subset forms a contiguous subsequence. The motivation for a linear ordering is that the k subsets can be completely determined by specifying their first and last records.

More recently a generalization of this problem has appeared in the literature. The generalization, called the Generalized Consecutive Retrieval Problem (GCRP), can be described as follows: Given n records and k subsets of these records, produce a directed tree on the records so that each subset lies on a contiguous path. The analogy with the CRP should be apparent. In the CRP each record has a unique NEXT pointer associated with it and every record is pointed to by at most one other. In the GCRP this latter constraint is relaxed and many records are allowed to point to the same record. Many problems not solvable in the linear setting can be solved by the generalized approach.

M. Truszczynski proved that the GCRP can be solved in polynomial time [4]. In this paper we offer an algorithm, for constructing a solution, that runs in time linear in the size of the input when the subsets are represented as linked lists.

Preliminaries

A tree is a directed acyclic graph in which each vertex has outdegree 1 except for a unique vertex called the sink that has outdegree 0. The edges in a tree are thus all directed towards the sink with sons pointing to their fathers.

A subset S of the vertices of a tree T is linear if and only if the points in S form a contiguous path in T. The tree T is said to linearize S if S in linear in T.

The Genrealized Consecutive Retrieval Problem (GCRP) can now be stated as follows:

> Given a set of points $A = \{1,\ldots,n\}$ and a collection $A_1,\ldots,$ subsets of A, whose union equals A, construct a tree T that linearizes each A_i.

Without loss of generality we will assume that the A_i are connect in the sense that the graph with vertices A_i and with edges betwe nondisjoint A_i's is connected.

General Discussion and Some Examples

This section is devoted to a number of facts and simple examples that may be useful to bear in mind while reading the rest the paper.

Let the subsets $A_1,\ldots,A_k$ and the vertices $1,\ldots,n$ be a part lar instance of the GCRP. It is easy to prove that if A_i and A_j intersect then the points in their intersection must be linearize For this reason, the GCRP can be viewed as k overlapping Consecut Retrieval Problems. Each problem consisting of the points in an and the sets $A_i \cap A_j$ for $j=1,\ldots,k$.

If two distinct points a, b both lie in some A_i, then in any solution to the GCRP, there must be a path (in an undirected sense) between a and b that lies entirely within the set A_i. It is useful to represent this fact with a picture like $a \underset{A_i}{\sim\sim\sim} b$. Since directed trees are also trees in an undirected sense, if there must be two distinct undirected paths joining points a and b then there can be no solution since trees do not have cycles. This fact is used to prove the following lemma, as well as many later results.

<u>Lemma</u> (Three Intersection): If three sets A_1, A_2, A_3 overlap as in Fig. 1, then at least one of regions X, Y, or Z must be empty.

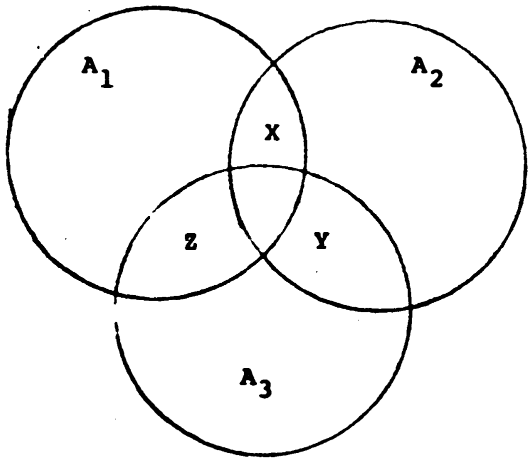


Figure 1
Three intersecting sets

Proof: Suppose none of the specified regions is empty. Let x be a point in X, y a point in Y, and z a point in Z. Then $x \underset{A_2}{\sim} y$, $y \underset{A_3}{\sim} z$, and $z \underset{A_1}{\sim} x$, which implies that there must be an undirected cycle and so there can be no solution.

Two sets X and Y that have the property that either $X \subseteq Y$ or $Y \subseteq X$ are called nested. A nest is a collection of sets that can be linearly ordered by inclusion. Using these definitions the previous lemma can be couched in a more useful form.

Cor: Let A_2 and A_3 both intersect A_1. If A_2 and A_3 also intersect each other outside of A_1 then $A_2 \cap A_1$ and $A_3 \cap A_1$ must be nested.

A collection $S_1, \ldots, S_r$ of sets forms a chain if $S_i \cap S_{i+1} \neq \phi$ for each i, and if all other intersections (except possibly $S_1 \cap S_r$) are empty, Fig. 2(a).

The sets $S_1,\ldots,S_r$ form a <u>cycle</u> if in addition to being a chain, $S_r \cap S_1 \neq \phi$, Fig. 2(b).

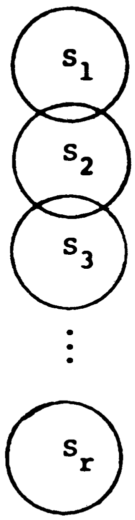


Figure 2(a)
A chain of sets

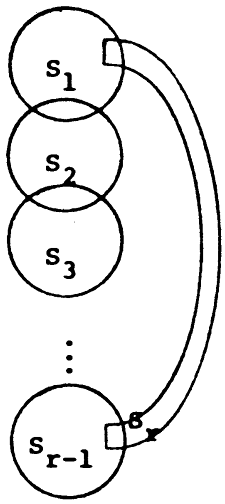


Figure 2(b)
A cycle of sets

<u>Lemma</u>(Cycle): An instance of the GCRP has no solution if the input contains a cycle of sets.

<u>Proof</u>: Since each $S_i \cap S_{i+1}$ must be a linear, we can without loss of generality, assume that each contains only a single point. Let a_i be the point in $S_i \cap S_{i+1}$ for $i=1,\ldots,r-1$. Let a_0 be the point in $S_r \cap S_1$. Then $a_0 \underset{S_1}{\sim} a_1 \underset{S_{r-1}}{\sim} a_r \underset{S_r}{\sim} a_0$ which implies there must be an undirected cycle that includes a_0. Therefore there can be no solution to the GCRP. □

This fact is used later to prove the "jump" lemma.

Top Level Sets

The algorithm for solving the GCRP can be understood as the composition of several steps. In the first step a subset of the A_i called top level sets, whose union is A, is selected. For notational convenience we assume that the A_i are numbered so that the top level sets are $A_1,\ldots,A_t$, $t \le k$. This allows us to refer to the top level sets by the names $T_1,\ldots,T_t$ where $T_i = A_i$. In addition to selecting these sets, we form a tree on them by assigning to each T_i (except the root) a T_j called his father (denoted $F(T_i)$). The top level sets and father pointers are chosen so that the T_i form a structure we shall call a nested, straightened tree without jumps.

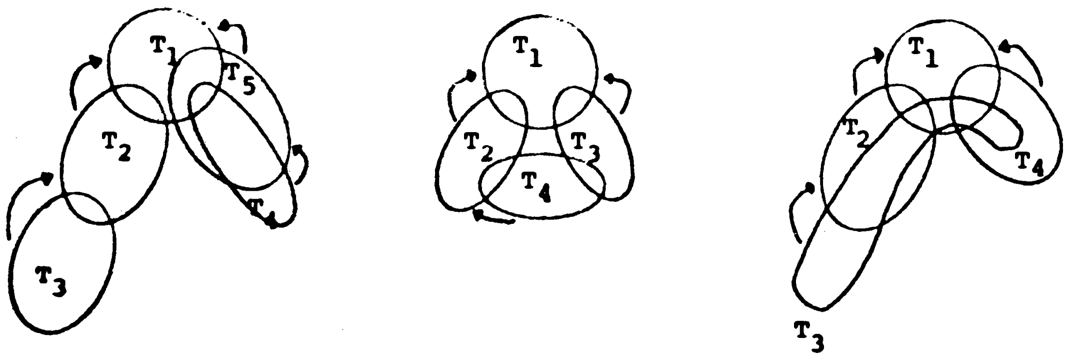
A top level tree is a collection of sets T_i, $1 \le i \le t$ that form a tree with father function F such that

(1) for each T_i except the root $T_i \cap F(T_i) \ne \phi$,

(2) the collection $\{T_i - F(T_i) \mid 1 \le i \le t\}$ is pairwise disjoint,

(3) for each T_i except the root, the size of $T_i \cap F(T_i)$ is at least as large as the intersection of $F(T_i)$ with any A_j that intersects a descendant of T_i.

Intuitively, conditions (1) and (2) force the T_i to "look" like a "tree" as in Fig. 3a. Fig. 3b is ruled out since $T_4 - T_2$ intersects $T_3 - T_1$ and Fig. 3c is ruled out since $T_3 - T_2$ intersects $T_4 - T_1$.



(a) (b) (c)

Figure 3
(a) is a tree, (b) and (c) are not

In a nested top level tree the overlap between fathers and sons is simple. The way a father and son overlap can be formally expressed by defining their juncture. Let F be a father and S his son in the top level tree. Define

$J_S^F = \{A_i \cap F \mid \text{both } A_i \cap F \text{ and } A_i \cap (S - F) \text{ are non-empty}\}.$

Thus the juncture of F and S is a collection of sets, each contained within F. In Fig. 4, $J_{T_2}^{T_1} = \{ \{2,3,6\}, \{3\} \}$ and $J_{T_3}^{T_1} = \{ \{6,7\} \}$.

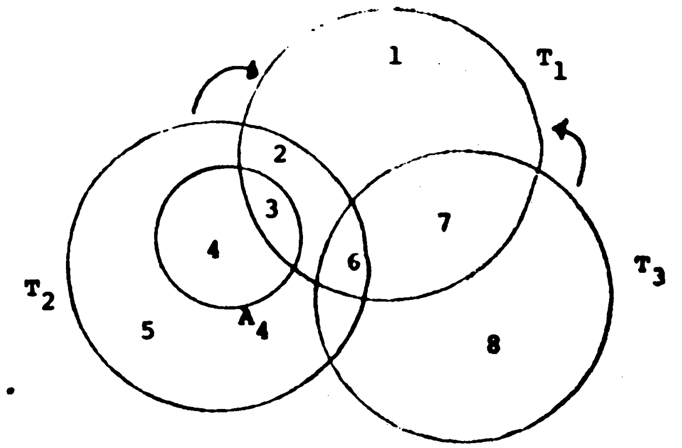
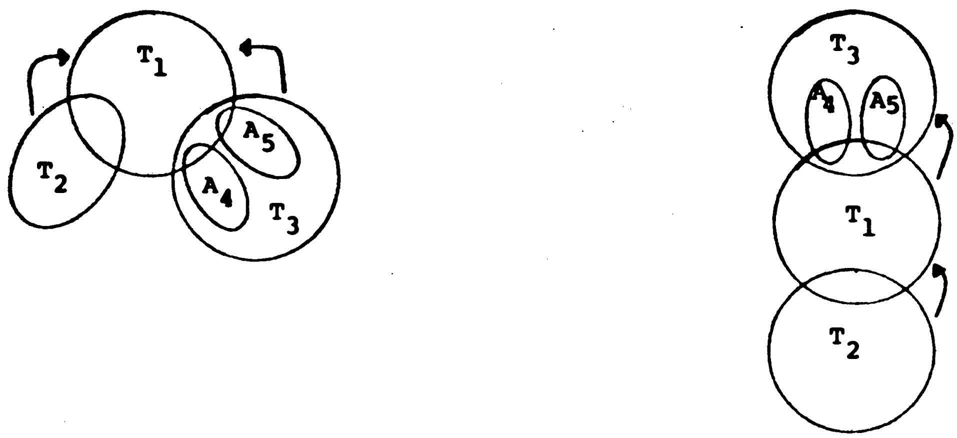


Figure 4
An example of a tree with two junctures

A top level tree is <u>nested</u> if and only if J_S^F is a nest for every father and son. [In any nested top level tree the largest A_i must be a T_j.]

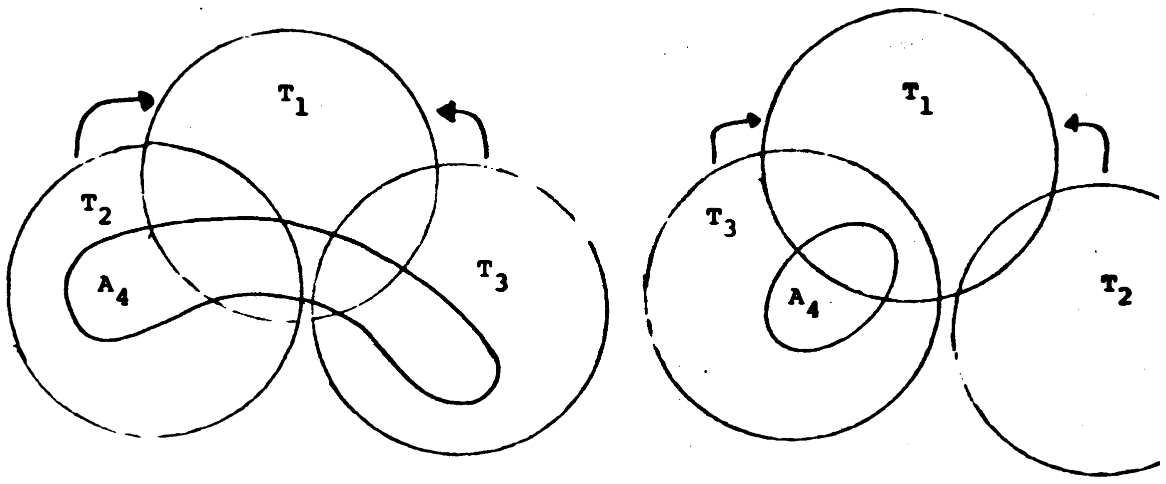
A <u>bad juncture</u> is one in which the sets do not form a nest The tree of T_i in Fig. 5a does not satisfy the nest condition since $A_4 \cap T_1$ and $A_5 \cap T_1$ are not nested. A nested tree is illustrated in Fig. 5b. Note that $J_{T_3}^{T_1}$ is just the one set $T_1 \cap T_3$ since neither A_4 nor A_5 intersect T_1 outside of T_3.



(a) (b)

Figure 5
(a) is a non-nested tree, (b) is a nested tree.

A <u>straightened</u> top level tree is one in which no A_i intersects the subtrees of two distinct sons of a father outside of the father. This rules out Fig. 6a but not Fig. 6b. Observe that Fig. 6b does not satisfy the nest condition since $J_{T_2}^{T_1} = \{T_1 \cap T_2, A_4 \cap T_1\}$ is not a nest. A more intuitive way of defining straightened is to say a tree is straightened if and only if every A_i lies along a branch.



(a) (b)

Figure 6

(a) is a non-straightened tree, (b) is a straightened tree.

A tree has a jump if there exists a set S and some ancestor A of S such that S intersects the father of A but does not intersect A outside of this father. Such a situtation is illustrated in Fig. 7.

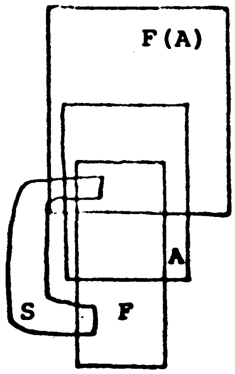


Figure 7
A tree with a jump

From now on by top level tree (TL tree) we will mean a straightened, nested and jump-free tree of top level sets. The next lemma exhibits an important property of TL trees.

<u>Lemma</u>: (One Set) In a TL tree every A_i is contained in some T_j.

Proof: Let A_i be one of the original subsets. Let T_j be the top level set farthest from the root such that $T_j - F(T_j)$ intersects A_i. We will show by induction on r that $A_i \cap F^r(T_j)$ is contained within T_j. This will imply, since the TL tree is straightened and thus A_i can only intersect top level sets on the branch from the root to T_j, that A_i is contained in T_j.

Clearly, $A_i \cap F^O(T_j) = A_i \cap T_j \subseteq T_j$. Assume $A_i \cap F^P(T_j) \subseteq T_j$. Using the corollary to the Three Intersection lemma and by condition (3) on top level sets, we can deduce that $F^{P+1}(T_j) \cap A_i$ is contained in $F^P(T_j)$. Therefore, $A_i \cap F^{P+1}(T_j)$ is contained in T_j.

□

A TL tree depicts an instance of the GCRP as a collection of overlapping CRP's, one for each T_i. Within each top level T_i, the solution to the GCRP must also be a solution to the CRP defi by the points in T_i and the sets formed by the A_j's restricted t

Unfortunately, it is not enough to simply solve the CRP's f each T_i. For one thing, a set that lies entirely within the int section of m T_i's would have to be represented m times so the to running time would not be linear. A more important problem is t each CRP may have many solutions. If T_i and T_j intersect then n all of the solutions to the CRP for T_i will necessarily "fit" al of the solutions to the CRP for T_j. In Fig. 8 the line $2 \rightarrow 3 \rightarrow 5$ is a solution to T_1 but there is no solution to T_2 in which 3 has a free outdegree.

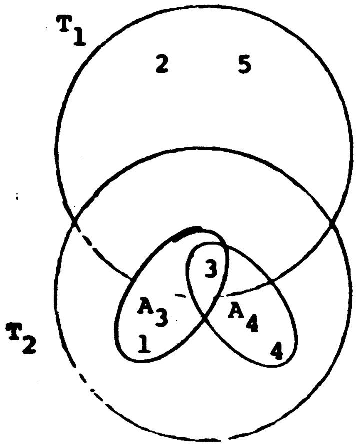


Figure 8
A top level tree in which T_1 cannot contain the sink

We explain how to solve the "fitting" problem after we first show how to construct the TL tree.

<u>Finding a Nested, Straightened Jump-Free Tree of Top Level Sets</u>

The tree of top level sets is constructed by a multi-phase algorithm. The first phase selects the top level sets in a depth first fashion and records the order in which sets are selected. The procedure Buildtree is initially called with the index of the largest A_i. At any stage in the recursion, B_i represents that fraction of A_i that does not lie in the tree already constructed. Initially then, for all i, $B_i = A_i$.

Buildtree(index)

1 <u>for each</u> B_i that intersects B_{index} <u>do</u>

2 place i on a list of neighbors

3 <u>for each</u> x in B_{index} <u>do</u> remove x from each B_i

4 sort neighbor list by size of $A_i \cap A_{index}$ (note: the A_i are the original sets, the B_i are the A_i with certa points removed)

5 <u>for each</u> i in the neighbor list (by decreasing size of interse

6 <u>if</u> $B_i \neq \phi$ <u>then</u>

<u>do</u>

7 set $F(T_i) := T_{index}$

8 Buildtree(i)

<u>od</u>

Buildtree constructs the tree depth first so the tree it produc is necessarily straightened. Jumps in the tree can be detected by a slight modification of the Buildtree procedure. Initially $F(T_i)$ is to zero for every i. Prior to setting $F(T_i)$ to index in line 7 chec to see if $F(T_i)$ is either 0 or $F(T_{index})$. If it isn't then a jump exists since A_i intersects a portion of the tree already constructed, namely $F(A_i)$, but does not intersect $F(A_{index})$. The next lemma show that if a jump occurs then the sink in any solution to the GCRP must lie in the subtree rooted at $F(T_{index})$.

<u>Lemma</u>(jump): If Buildtree encounters a jump while processing set T_r then in any solution to the GCRP, the sink must lie in the subtree rooted at $F(T_r)$.

<u>Proof</u>: Suppose that Buildtree encounters a jump at T_r, i.e. let

$$T_0 \longleftarrow T_1 \longleftarrow \cdots \longleftarrow T_r \longleftarrow C,$$

($r \geq 2$) be a branch in the tree being built by Buildtree where $C \cap T_0 \neq \phi$ but

$$C \cap T_1 = \cdots = C \cap T_{r-1} = \phi.$$

The sets $T_0, \ldots, T_r, C$ have to be arranged in a very special way which we proceed to illustrate.

We first demonstrate that each $T_i \cap T_0 \neq \phi$, for otherwise we could locate a cycle of sets as follows. Let i and j be initially 1 and let $S_0 = T_0$, $S_1 = T_1$. Assume that a chain of sets $S_0, \ldots, S_i = T_j$ has been constructed so far. Clearly $T_{j+1} \cap S_i \neq \phi$ as T_j ($= S_i$) is the father of T_{j+1}. There are three cases which can occur.

Case 1: If $T_{j+1} \cap S_{i-1} \neq \phi$ then we know from the corollary to the Three Intersection lemma that $T_{j+1} \cap S_{i-1}$ and $S_i \cap S_{i-1}$ are nested. Furthermore, $S_i (= T_j) \cap S_{i-1}$ must contain $T_{j+1} \cap S_{i-1}$ since the T_m's were chosen in order of size of intersection. Therefore, we must have a picture as in Fig. 9.

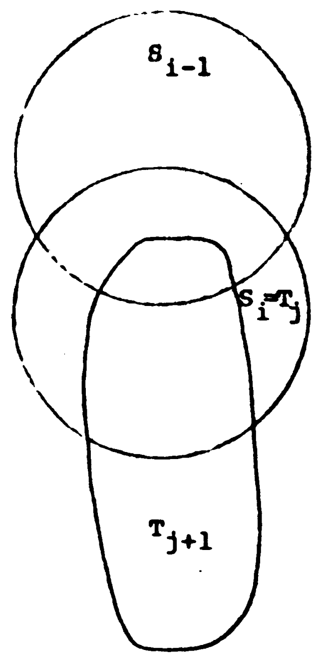


Figure 9
Case 1

In this case, we set S_i to be T_{j+1} and continue.

Case 2: (a) T_{j+1} does not intersect any S_m prior to S_i so the picture is as in Fig. 10.

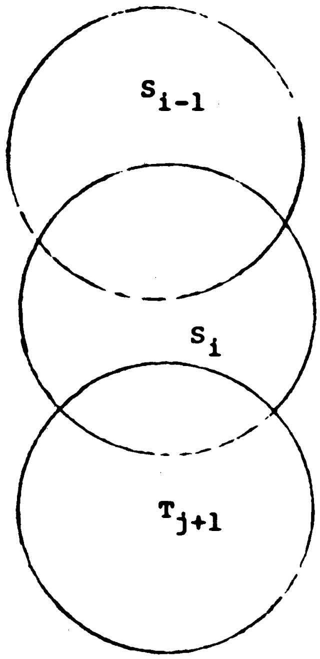


Figure 10
Case 2a

and we extend the chain by making $S_{i+1} = T_{j+1}$.

(b) If T_{j+1} intersects the chain $S_0, \ldots, S_{i-2}$, but not S_{i-1}, let S_t be the last set in $S_0, \ldots, S_{i-2}$ that intersects T_{j+1}. Then, $T_{j+1}, S_t, \ldots, S_i$ forms a cycle (Fig. 11) so there can't be a solution to the GCRP.

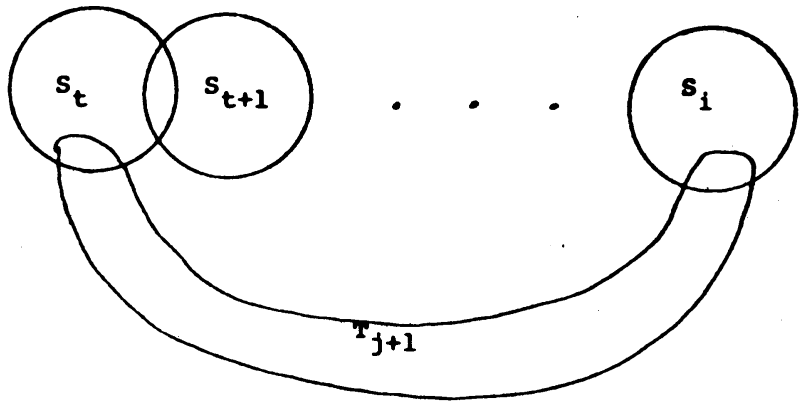


Figure 11
Case 2b

The process of extending the chain must end in a cycle because $C \cap T_0 \neq \phi$. Therefore, each T_i intersects T_0.

Now consider T_0, T_{r-1}, T_r and C. Since the sets are top leve $T_{r-1} \cap T_0$ contains $T_r \cap T_0$. The nesting lemma implies that $T_r \cap T_0$ and $C \cap T_0$ are nested. Therefore, we can select a point c_o in $C \cap T_0 \cap T_{r-1} \cap T_r$. Let c_r be a point in $C \cap T_r - (T_0 \cup \ldots \cup T_{r-1})$ and a point x in $T_r \cap [T_{r-1} - (T_0 \cup \ldots \cup T_{r-2})]$ as in Fig. 12. Since c_0 and c_r lie in C there must be an undirected path between them in any solution. Similarly, C_0 T_{r-1} x since c_0 and x are both in T_{r-1}. As c_0, c_r, and x are all in T_r they must lie along a lin As x is not in C and c_r is not in T_{r-1} either $c_r \longrightarrow c_0 \longrightarrow x$ or $x \longrightarrow c_0 \longrightarrow c_r$. In the case where $x \longrightarrow c_0 \longrightarrow c_r$ there is an arrow out of T_0, $T_1, \ldots, T_{r-1}$ into T_r and thus the sink must be in the subtree rooted at T_r. In the other case where $c_r \longrightarrow c_0 \longrightarrow x$ there is an arrow out of $T_0, \ldots, T_{r-2}$ into T_{r-1} and so the sink must be in the subtree rooted at T_{r-1}. In either case the sink must lie in the subtree rooted at $F(T_r)$. □

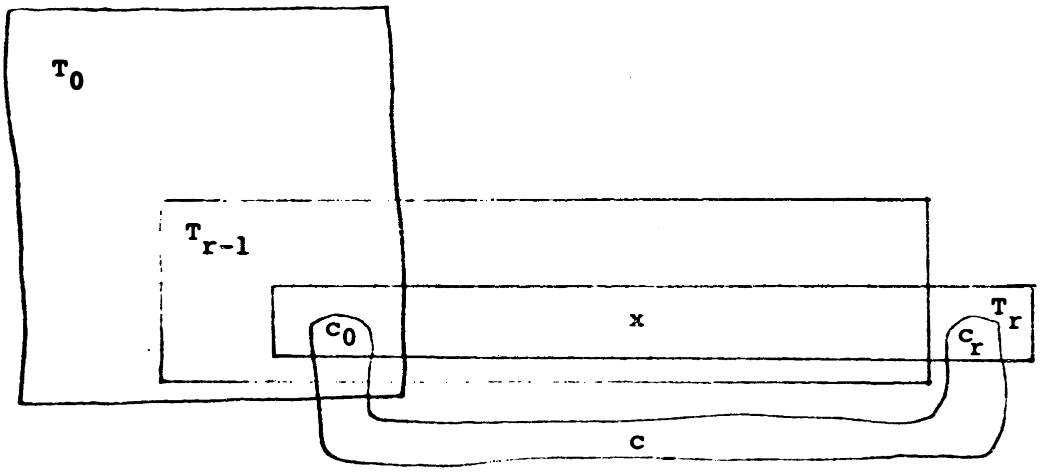


Figure 12
How T_0, T_{r-1}, T_r, and C must be arranged

The above lemma implies that all jumps must lie along a path in the tree. Let T_i be the set farthest from the root where a jump was detected. If a new tree were constructed starting at the father of T_i then the tree would have to be jump-free.

In addition to removing jumps, we must also avoid bad junctures. If $J_S^{F(S)}$ is a bad juncture we can prove that the sink has to lie in the subtree rooted at S:

Lemma (Bad nest): If the buildtree algorithm locates a non-nest at the juncture of a father F and son S, then, in any solution to the GCRP, the sink must lie in the subtree rooted at S

Proof: Since the sets in J_S^F do not form a nest, there must be two sets, A_i and A_j, which contribute to the juncture but are not nested within F. The fact that A_i and A_j contribute to the nest implies that $A_i \cap F$, $A_i \cap (S - F)$, $A_j \cap F$, and $A_j \cap (S - F)$ are all non-empty. Let i_F be a point in $(A_i - A_j) \cap F$, and let j_F be a point in $(A_j - A_i) \cap F$. Also, let i_S be a point in $(A_i - F) \cap S$, and j_S be a point in $(A_j - F) \cap S$. See Fig. 13. Notice that i_S and j_S can be chosen so that they are distinct. Since $F \cap S$ has to be linear, $i_F \underset{F\cap S}{\text{——}} j_F$. Furthermore, $i_F \underset{A_i}{\sim\sim} i_S$ and $j_F \underset{A_j}{\sim\sim} j_S$.

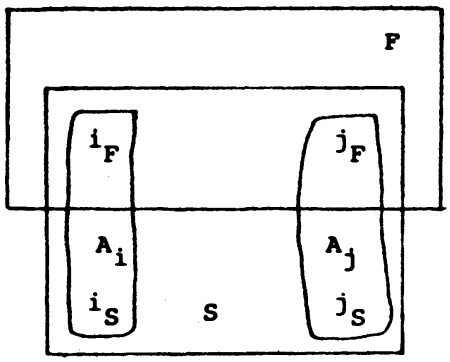


Figure 13
Example of a bad juncture

Therefore, either $i_F \rightsquigarrow i_S$, or $j_F \rightsquigarrow j_F$. In either case this implies that the sink must lie in the subtree rooted at S. □

Thus if the GCRP has a solution, all bad junctures and all jumps must lie along a single path in the tree. This means that Buildtree need only check for bad junctures in the subtree below the lowest jump. This is important, for it means that Buildtree will be searching for bad junctures in a subtree which is jump-free. The absence of jumps allows the testing for bad junctures to be done quickly.

In the timing analysis section we describe how to quickly test the junctures in a jump-free tree for non-nests. The process is straightforward but involves detailed manipulation of lists.

Finally, consider the lowest bad juncture or jump. If it is a bad juncture between $F(A_{index})$ and A_{index} then set newroot equal to index. If it is a jump which is detected at index, set newroot equal to the grandfather of index. Then restart the algorithm by calling Buildtree(newroot). It is very important that Buildtree constructs the same tree below newroot as it did in the original phase in order to insure that this portion of the tree ends up having no jumps or bad junctures. For this reason we assume that ties are always broken in a uniform way, say, lexicographically.

We can now show that the Buildtree procedure, as described above, works.

Theorem: Either Buildtree produces a tree of top level sets that is straightened, nested and jump-free or there is no solution to the original GCRP.

Proof: In the first phase Buildtree locates a vertex r such that the sink in any solution must lie in a subtree of r (Jump Lemma). Furthermore, the subtree rooted at r is straightened, nested and jump-free. The second phase of Buildtree starts with r as the root and builds the original subtree of r in the same fashion as in the first phase. Thus if the tree is faulty, there is another vertex v not in the original subtree of r such that the sink of any solution must lie in the new subtree of v, contradicting the fact that the sink had to lie in the original subtree of r. There fore, if Buildtree fails to construct a straightened, nested and jump-free tree there can be no solution to the original GCRP. □

PQ Trees

Given a set of vertices A, a PQ tree over A is a tree whose leaves are elements of A and whose internal nodes are labeled as being either P nodes or Q nodes. Each element of A is represented as a leaf exactly once.

By reading the leaves of a PQ tree from left to right, one gets a linear order of the vertices. The left-to-right order is called the tree's frontier.

Two PQ trees are equivalent if one can be obtained from the other by any sequence of the following operations:

(1) arbitrarily permute the children of a P node,

(2) reverse the sons of a Q node

In this way, a PQ tree T defines a certain collection of linear orders on the vertices, those you get by considering the frontiers of all trees equivalent to T.

Booth and Lueker [2] have shown how to construct, in linear time, a PQ tree which represents all linear orderings in which certain sets of vertices are consecutive. They thereby showed how to solve an instance of the CRP in linear time. We assume that their algorithm is used whenever we say to construct a PQ tree for an instance of the CRP.

A particular subset S of A is last (first) in a PQ tree if all of the points of S appear at the right (left) end of the frontier. Given a linear order, as perhaps described by a PQ tree, FIRST(S) denotes that point in S that appears before any other point in S. LAST(S) is the point that follows all the others.

Solving Overlapping CRP's

Having a TL tree for an instance of the GCRP enables us to construct a corresponding tree of Consecutive Retrieval Problems in which points are not represented too often and whose solution can be easily put together to form a solution to the GCRP.

Consider the situation in which there is a TL tree with jus two sets, a father F and his son S. The original A_i break into categories, those that are totally contained within F and those that intersect S outside of F. Let these sets be $F_1,\ldots,F_r$ and $S_1,\ldots,S_s$ respectively. Notice that the sets F and S both appea somewhere in these collections. How can we find a tree on the points in F U S so that each A_i is linearized?

Recall that since $F \leftarrow S$ forms a TL tree, the juncture J_S^F i a nest of sets. The beauty of the nesting constraint is that it implies a good bit about the order of the points of $F \cap S$ in any solution. Without loss of generality assume the S_i have been indexed so that

$$S_1 \cap F \subseteq S_2 \cap F \subseteq \ldots \subseteq S_s \cap F,$$

as for example in Fig. 14.

$$\begin{aligned} \text{Let}\quad S_1' &= (S_1 \cap F), \\ S_2' &= (S_2 \cap F) - S_1, \\ &\vdots \\ S_s' &= (S_s \cap F) - S_{s-1}. \end{aligned}$$

The S_i' are the sets which look like parts of rings in Fig. 14

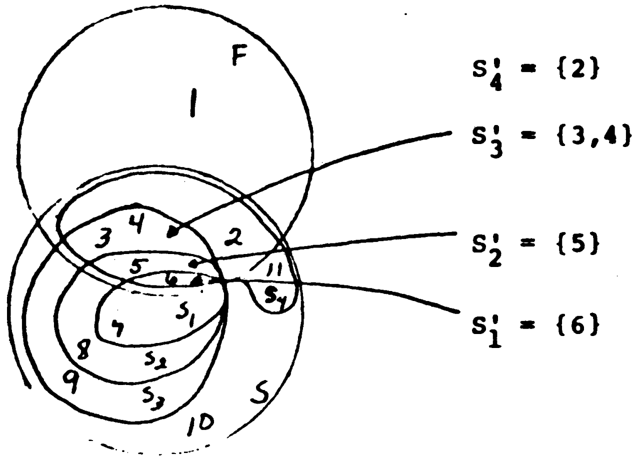


Figure 14
Example of a two set top level tree

A collection of disjoint sets $S_1', \ldots, S_s'$ is aligned in a solution to the GCRP if and only if either (for all i)

1. The points in S_i' immediately precede the points in S_{i+1}', or
2. The points in S_{i+1}' immediately precede the points in S_i

The following lemma shows how the points in $F \cap S$ must be arranged.

Lemma (Alignment): If F is the father of S in a TL tree and the S_i' are as defined above then each S_i' is linear and the sets $S_1', \ldots, S_s'$ are aligned.

<u>Proof</u>: Observe that the specified sets are aligned if and only if either

$$\mathrm{LAST}(S_1 \cap F) = \ldots = \mathrm{LAST}(S_s \cap F)$$
$$\text{or} \quad \mathrm{FIRST}(S_1 \cap F) = \ldots = \mathrm{FIRST}(S_s \cap F).$$

There are two cases to consider:

<u>Case 1</u>: There is an arrow from $(S - F)$ into S_1'. Since S_1' is the intersection of two sets, it must be linear. With respect to arrows in S, each vertex in S_1' has indegree = 1 (S must be linear). Since $S_2 \cap F$ is linear and since no arrow can go from S_2 into S_1', we conclude that S_1' precedes S_2' and thus $\mathrm{FIRST}(S_1 \cap F) = \mathrm{FIRST}(S_2 \cap F)$. Then by induction,

$\mathrm{FIRST}(S_1 \cap F) = \ldots = \mathrm{FIRST}(S_s \cap F)$, and each S_i' is linear.

<u>Case 2</u>: There is an arrow from S_1' into $(S - F)$. Arguing as above, we conclude that S_1' must follow S_2' and therefore $\mathrm{LAST}(S_1 \cap F) = \mathrm{LAST}(S_2 \cap F)$. Then by induction,

$\mathrm{LAST}(S_1 \cap F) = \ldots = \mathrm{LAST}(S_s \cap F)$, and each S_i' is linear. □

It is a simple matter to construct a CRP, call it CRP(F) for the points in F that linearizes each $A_i \cap F$ as follows:

(1) The points in CRP(F) are the points in F.

(2) The sets in CRP(F) are $F_1,\ldots,F_r$, $S_1 \cap F$, $\ldots, S_s \cap F$.

Adding the set $(S_s \cap F - S_1 \cap F)$ to CRP(S) allows only those solutions in which $S_1 \cap F$ precedes or follows $(S_s \cap F - S_1 \cap F)$. The fact that $S_2 \cap F$ has to be linear implies that S_1' and S_2' have to be aligned. A simple extension of this argument shows that adding the set $(S_s \cap F - S_1 \cap F)$ to CRP(S) actually rules out all solutions to CRP(S) in which $S_1', \ldots, S_s'$ are not aligned.

By placing all of the sets $S_i \cap F$, $i = 1,\ldots,s$ in CRP(F), the S_i impose all the constraints they possibly can on the relative order of points within $F \cap S$. Thus in constructing a CRP for S we can leave out the points of $F \cap S$ and still be able to determine all the linear orderings of S. The points of $F \cap S$ are represented by two special points. The location of these two points in valid orders for S determines where the linear segment for $F \cap S$ can fit, Fig. 15.

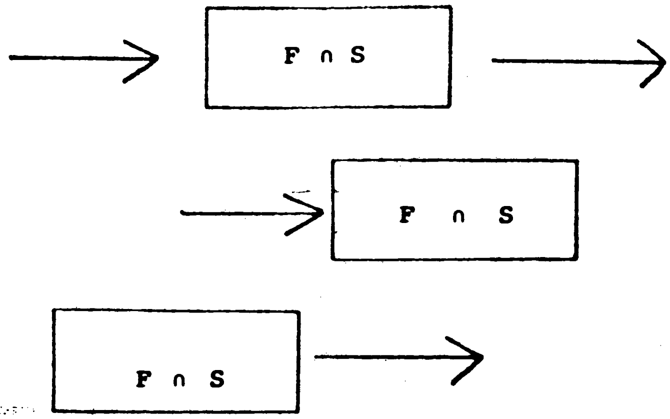


Figure 15
Possible arrangements of F∩S in solutions to S

The S_i' in F∩S have to be linear and aligned in any solution. Thus they must form a picture as in Fig. 16 where each S_i' has bee shrunk to a point.

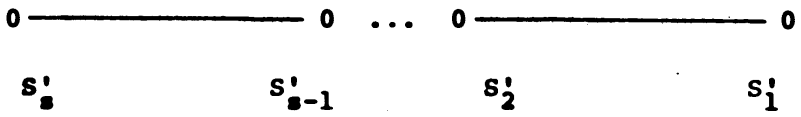


Figure 16
Aligned Sets

Within S, $F \cap S$ is represented by the two points I_S and O_S that stand for the endmost points of $S \cap F$ (I_S stands for the Innermost set, S_1' and O_S stands for the Outermost set, S_s'). Since there has to be a path connecting the endmost points of $S \cap F$, I_S and O_S are made adjacent by adding the set $\{I_S, O_S\}$. Any S_i containing $F \cap S$ must have $F \cap S$ as a linear segment in its solution. Therefore, in each S_i that contains $S \cap F$ all of $F \cap S$ is replaced by the two points I_S and O_S. If an S_j intersects F but does not contain all of $F \cap S$ then it must contain $S_1 \cap F$ since J_s^F is nested. This means that the points in $F \cap S$ must either all follow or all precede the points in $(S_j - F)$, Fig. 17. Therefore, we can replace all of the points in $S_j \cap F$ by the single point I_S.

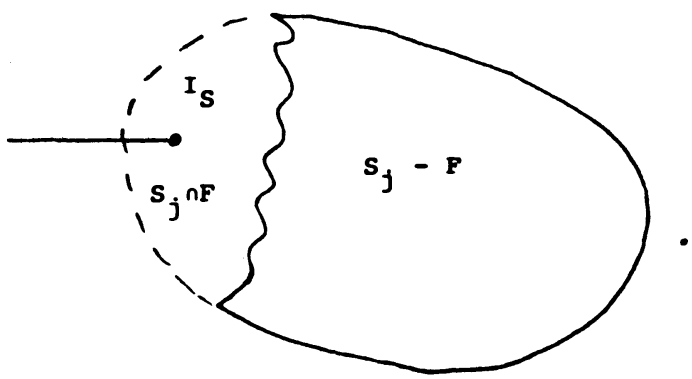


Figure 17
The points in S_j - F must precede or follow I_S

A second CRP, CRP(S) is then formed using the modified S_i. In particular, CRP(S) consists of:

(1) all the points in S - F plus the two points I_S, O_S,

(2) the set $\{I_S, O_S\}$, and

(3) the modified S_i.

Solving CRP(S) and substituting valid linear orderings of the points in F∩S for the edge between I_S and O_S yields the set of all solutions that linearize the S_i.

Armed with PQ trees which represent solutions to CRP(F) and CRP(S) we are in a position to solve the original GCRP.

Consider the PQ tree which represents all possible solutions to CRP(S). Using the following simple recursive algorithm we can determine whether O_S can be made last.

```
marklast (node)
   case
      [P node]:  for each son of node do marklast (son)
      [Q node]:  marklast(leftmost son of node);
                 marklast(rightmost son of node)
      [leaf]:    mark node
   esac
```

If O_S cannot be last then we know that the points in F∩S must lie in the middle of any solution to S. In Fig. 18a F∩S must be in the middle of any solution to S, while in Fig. 18b F∩S can be either first or last in S.

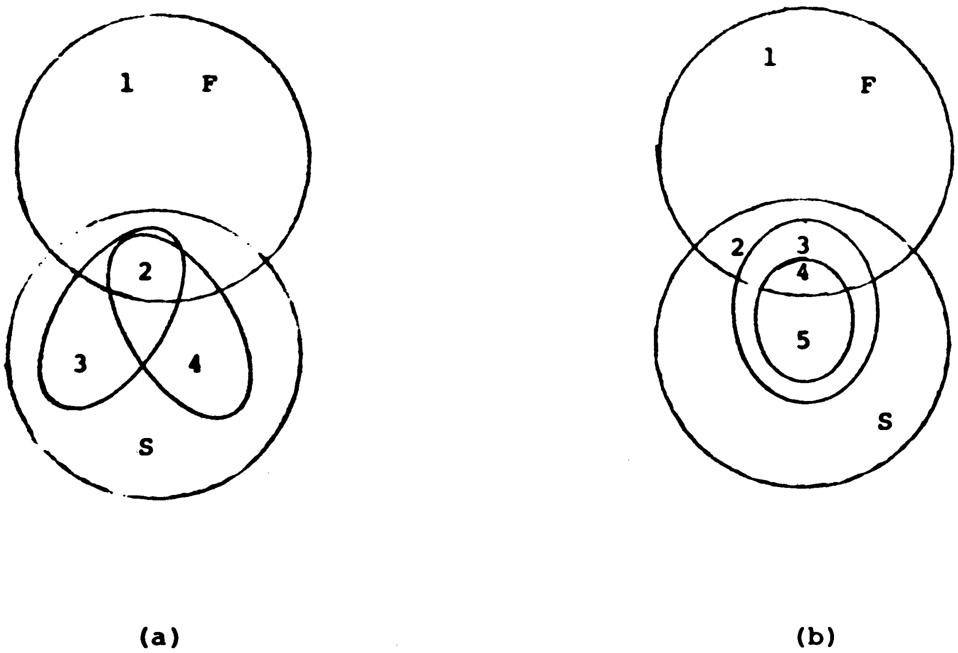


Figure 18
(a) F∩S can't be last in S (b) F∩S can be last S

If O_S cannot be last then there must be an arrow from F∩S into (S - F). Since F∩S is linear we conclude that there can be no arrow from F∩S into (F - S). Thus, F∩S must be last in F. Take the PQ tree for CRP(F) and rearrange the nodes so that the points in F∩S come last. Examine the points in F∩S. If they are ordered with S_1' last then choose a solution form CRP(S) in which O_S-->I_S otherwise choose one in which I_S-->O_S. Reading from the frontier of both trees, insert the appropriate arrows but replace I_S and O_S with the corresponding endmost points of F∩S.

If O_S can be made last then choose a solution to CRP(F) in which S_1' precedes S_s'. Reading from the frontiers, insert the

appropriate arrows and replace I_S by FIRST($F \wedge S$) and O_S by LAST($F\wedge$S
The procedure outlined will yield a solution if there is one.

The case in which the father F has more than one son is sligl more complicated since junctures may overlap in F. To solve a situation in which F has m sons $S_1,\ldots,S_m$, start by constructing the m+1 CRP's CRP(F), CRP(S_1), ...,CRP(S_m) as above.

If some son, S_i, does not admit of a solution in which I_{S_i} ca be last then we say de demands an arrow because in any solution ai arrow must go from F into S_i - F. At most, one son can demand an arrow if there is a solution.

Let $N_1,\ldots,N_m$ be the junctures $J^F_{S_1},\ldots,J^F_{S_m}$. So each N_i is a nest of sets. If a solution to the GCRP causes the innermost set of a nest to be last in the nest then the nest is said to be directed outwards. If the orientation is the other way then the nest is directed inwards. The reason for these definitions is clear when you notice that if a juncture is directed outward then in order to make the A_i from which the innermost set in the nest was obtained linear, an arrow must extend from the nest into the remainder of this A_i, i.e. there must be an arrow directed out oi this nest. Therefore, in any given father, at most one juncture can be directed outwards. Moreover, if a juncture is directed ou ward then it must be last.

In the PQ tree for CRP(F) each nest N_i is directly associate with a Q node since Q nodes are the only mechanism for aligning sets. More than one nest may be associated with a given Q node but if this happens there are only two ways the nests can be arranged. If more than one nest is oriented in one direction then there can be at most one nest oriented in the other direction. If one nest, say N_1, is oriented in one direction and more than one nest is oriented in the opposite direction, then N_1 must be directed outward and all other nests must be directed inward. Also, N_1 must be made last in F.

The only other possible arrangement is if there are exactly two oppositely oriented nests on a Q node. When this case arises we know that one or the other must be directed outward and be last.

By considering sons that demand arrows, and Q nodes in CRP(F) that have conflicting nests, it is possible to match solutions to CRP(F) and CRP(S_1),...,CRP(S_m) thereby solving the GCRP. Finally, we note, and demonstrate in the following section, that this algorithm can be extended to work recursively if F is actually the son of another set.

<u>The Solvetree Algorithm</u>

It is useful to extend Booth and Lueker's definition of PQ trees to include a new type of internal node. This new node is called an R node, where the R stands for rigid. While sons of a P node may be permuted arbitrarily, and those of a Q node reversed, the sons of an R node may not be rearranged. This new type of tree is called a <u>PQR tree</u>.

We now describe in more detail the algorithm for constructing a solution for a GCRP once a TL tree has been found. Solvetree operates in three steps. In Step 1 Solvetree traverses the TL tree and constructs, for each T_i, a Consecutive Retrieval Problem, CRP(T_i). The points in CRP(T_i) are the points of T_i - F(T_i) plus the points I_i, O_i. The sets A_j that are the constraints of CRP(T_i are computed as follows:

> <u>for</u> <u>all</u> A_j that intersect $T_i - F(T_i)$
> <u>do</u> $A_j' := A_j \cap (T_i - F(T_i))$
>   <u>if</u> $A_j \cap F(T_i) \neq \phi$ <u>then</u> $A_j' := A_j' \cup \{I_i\}$
>   <u>if</u> $A_j \supseteq (T_i \cap F(T_i))$ <u>then</u> $A_j' := A_j' \cup \{O_i\}$
> <u>od</u>

In Step 2 Solvetree again traverses the TL tree, this time in postorder:

> Step2(F):
>   <u>for</u> <u>each</u> son S of F <u>do</u> Step2(S) <u>od</u>
>   Build a PQ tree for CRP(F).
>   Examine the PQ tree and the constraints added by th sons of F.
>   Using PQR trees, determine if there are any solution to CRP(F) which "fit" with solutions to the sons F. Having the solutions that "fit", determine wh of the following three situations are possible am these solutions:

(1) $I_F \dashrightarrow O_F$, where O_F is the sink of subtree rooted at F

(2) $I_F \dashrightarrow O_F \dashrightarrow x$, for some x in the subtree rooted at F

(3) $O_F \dashrightarrow I_F \dashrightarrow x$, for some x in the subtree rooted at F

Finally, in Step 3, Solvetree traverses the TL tree in preorder and at each set F chooses an order that fits the order already chosen for the father of F and also solves CRP(F). The algorithm then replaces the edge between I_F and O_F with the appropriate linear segment of the solution so far constructed.

It is easy to see how Step 1 is implemented so we just describe how Step 2 and Step 3 can be done in linear time.

<u>Step 2</u>

Having recursively developed PQR trees for all top level sets that are descendants of F, and also having determined for the sons of F which of (1), (2), and (3) are possible, a PQ tree is constructed for F that reflects the constraints:

(i) each $A_i \cap F$ must be linear

(ii) each juncture with a son must be aligned

The solutions allowed by this PQ tree must be further restricted due to demands made by the sons. If more than one son demands an arrow out of F then clearly there is no solution. Similarly, if a Q node has two or more nests oriented in each direction or if two or more Q nodes have nests oriented in both directions then there is an unresolvable conflict and so there can be no solution, Fig. 19

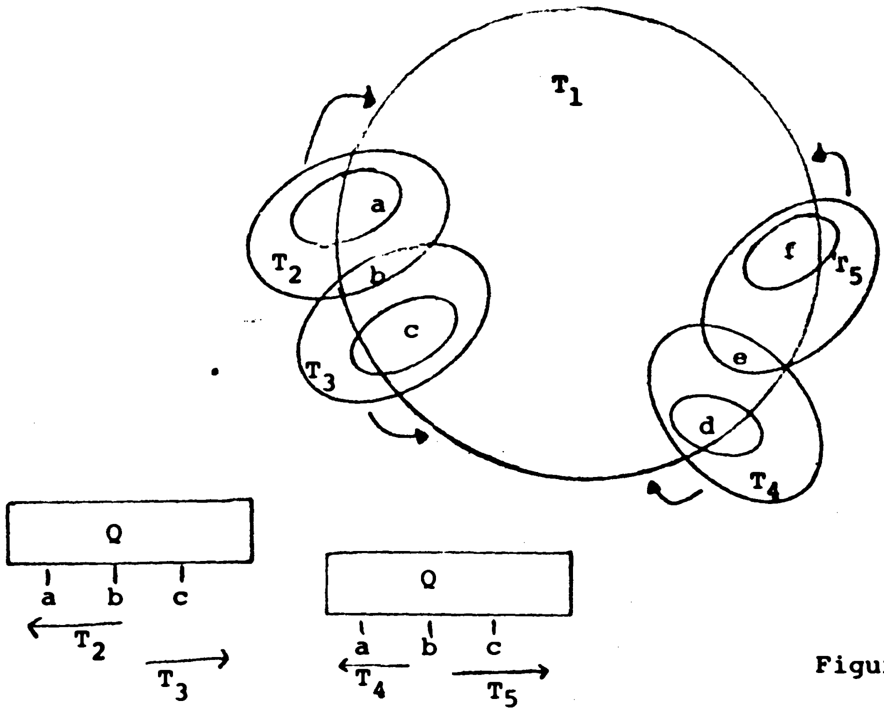


Figure 19

GRP(T_1) has two Q nodes with nests oriented oppositely

The algorithm has to handle two cases. In Case 1 there is exactly one Q node with a conflict. In Case 2 there are no Q no with conflicts. Case 1 divides into two subcases.

<u>Case 1a</u>: The Q node with a conflict has two or more nests orien one way and exactly one nest N_1 oriented the other way Let S_1 by the son whose juncture is N_1. If any son other that S_1 forces an arrow out of F, Solvetree directs N_1 outward and modifies the PQ tree to make N_1 last. In doing this, Solvetree converts certain nodes to type R as in Fig. 20. In addition, all othe

Q nodes with nests are turned so that their nests are directed in and are then made rigid. Solvetree returns that I_F may either precede or follow O_F but O_F may not be last. O_F and I_F must be adjacent sons of a Q (or R) node because of the set $\{I_F, O_F\}$ (we call a P node with exactly two sons a Q node). If it is a Q node that contains I_F and O_F then Solvetree returns that I_F may either precede or follow O_F. If it is an R node then Solvetree returns that I_F must precede O_F or that O_F must precede I_F depending on the order of the sons in the rigid node. In no solution can O_F be last in the subtree rooted at F because N_1 must be directed outward and hence an arrow must go from O_F in to the subtree rooted at S_1.

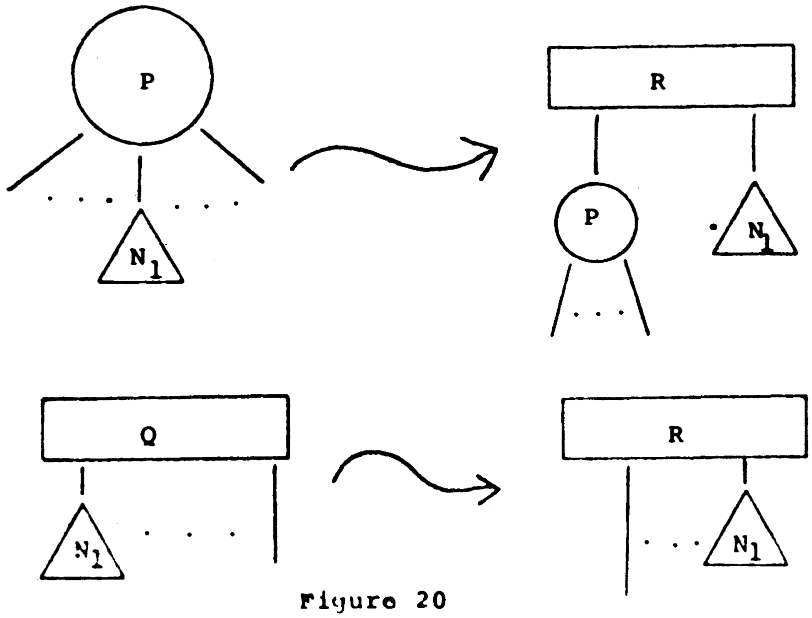


Figure 20
How to convert P and Q nodes to R nodes when moving N_1 to be last

__Case 1b__: The Q node with a conflict has exactly two nests, N_1 and N_2 , oriented oppositely. Let S_1 and S_2 be the sons with junctures N_1 and N_2 respectively. Clearly, one or the other of these nests will have to force an arrow out of F. If either S_1, or S_2 demands an arrow, say S_1, then direct N_1 outward and make N_1 last. If neither demands an arrow then Solvetree must examine two possibilities, namely, either N_1 or N_2 may be made last in F and directed outward. Again, all other Q nodes with nests are made rigid with their nests directed inward. Solvetree returns the information as to whether I_F may precede or follow O_F. In no case may O_F be last in the subtree.

__Case 2__: No Q node has nests with conflicting orientations. If some son, say S_1, with juncture N_1, demands an arrow, then we check to see whether $I_{S_1} \dashrightarrow O_{S_1}$ or $O_{S_1} \dashrightarrow I_{S_1}$ or both are allowed. In the first case, N_1 is directed in, in the second case N_1 is directed out and in the third case both possibilities must be tried. In all cases, N_i must be made last in the PQR tree.

If no son demands an arrow, Solvetree first determines if O_F may be last in the subtree rooted at F. This is done by moving O_F to be last and directing all nests inward. If no contradictions are encountered then O_F may be last and a solution exists with $I_F \dashrightarrow O_F$ where O_F is last in the subtree. Finally, Solvetree determines if I_F may precede or follow O_F. This is done by trying to direct the Q node containing I_F and O_F both ways. A failure occurs if this Q node would have more than one nest directed outward. If there is only one nest directed outward on the Q node, then it is moved to be last. All other nests are then directed in.

<u>Step 3</u>

In preorder, traverse the TL tree and at each set F select a solution from CRP(F) that is consistent with the solution chosen for the father of F. (There is always such a solution since constraints were passed up the tree in the previous step.)

Define <u>first'(T_i)</u> and <u>last'(T_i)</u> to be the first and last elements of T_i that appear in the solution constructed so far. Each time a solution is chosen for a TL set F, update first'(T_i) and last'(T_i) as follows:

```
for each son S of F
   for each T_i that contributes a T_i' to J^F_S
            (Note: T_i' is a nodified set and so may cont
                 an I_F or O_F)
   do
       let f,ℓ be the first and last elements of T_i' in the
       solution for CRP(F)
       if f is either I_F or O_F
       then do not update first'(S) as its value has not
            been changed
       else set first'(S):= f
       if ℓ is either I_F or O_F
       then do not update last'(S) as its value nas not
            been changed
       else set last'(S):= ℓ
   od
```

When the whole TL tree has been traversed, for each T_i there will be a linear order on the points in T_i-(Tree above T_i) and first'(T_i), last'(T_i) will be the first and last elements of $T_i \cap$(Tree above T_i). For each top level set S, except the root, th linear order will contain two adjacent vertices labelled I_S and (The arcs relating these veritces to the linear order are replaced as follows:

<u>if</u> (in the solution to CRP(S))
$y \to I_S \to O_S \to x$
<u>then</u> replace $y \to I_S$ by $y \to$ first'(S)
replace $O_S \to x$ by last'(S)$\to x$
<u>elsif</u> $y \to O_S \to I_S \to x$
<u>then</u> replace $y \to O_S$ by $y \to$ first'(S)
<u>elsif</u> $y \to I_S \to O_S$, O_S last in the solution to CRP(S
<u>then</u> replace $y \to I_S$ by $y \to$ first'(S)

<u>Solvetree</u> <u>Works</u>

If Solvetree fails then either

1) a father has two sons demanding arrows,

2) a solution to CRP(S) does not exist for some S,

or 3) the PQ tree for a CRP(S) has unresolvable Q node co

In any of these cases the original GCRP has no solution. Therefo we know the following:

:ain

<u>Lemma</u>: If Solvetree fails then the GCRP has no solution.

Now, if we can show that when Solvetree produces a tree then it is a solution to the GCRP we will have shown:

<u>Theorem</u>: Solvetree produces a solution to the GCRP when one exists, and fails otherwise.

<u>Proof</u>: Let T be the solution tree produced by Solvetree. We have to show that T linearizes each A_i. The proof is by induction on the TL tree. The induction hypothesis is that if $A_j \cap F(T_i)$ is linear then so is $A_j \cap T_i$.

Base case: Let T_{root} be the root of the top level tree. From the fact that $CRP(T_{root})$ includes each of $A_j \cap T_{root}$ we can conclude that each of these sets is linearized.

Inductive step: We want to show that if $A_j \cap F(T_i)$ is linear, then $A_j \cap T_i$ is linear. To prove this, let A_j be a set that intersects T_i. There are 3 cases.

<u>Case 1</u>: A_j is completely contained within $F(T_i)$. In this case $A_j \cap T_i = A_j \cap F(T_i)$ and therefore, by induction, it is linear.

<u>Case 2</u>: A_j does not intersect $F(T_i)$. In this case, the set $A_j \cap T_i$ appears as is in $CRP(T_i)$ and so $A_j \cap T_i$ is linear in the tree output by Solvetree.

<u>Case 3</u>: A_j intersects $F(T_i)$, but $A_j \not\subseteq F(T_i)$. We know that $A_j \cap F(T_i)$ is linear, by induction. A_j' in $CRP(T_i)$ must contain the point I_i and since I_i is adjacent to O_i in any solution to $CRP(T_i)$, either I_i precedes or I_i follows, all of a $A_j \cap (T_i - F(T_i))$. In both cases, since the solutions to $CRP(F(T_i))$ and $CRP(T_i)$ were chosen to fit together, $A_j \cap T_i$ is linearized.

Thus by induction we conclude that for all i and j, $A_j \cap T_i$ is linear. Since the TL tree is top level and nested, every A_j is contained in some T_i (One Set Lemma). Therefore, every A_j is linearized.

Timing Analysis

As a conceptual device the whole algorithm has been exposite as having many phases although an efficient implementation would only need three; one to generate the TL tree, another to calculat and solve the smaller CRP's, and a third to piece the smaller solutions together.

Let m be the sum of the lengths of the subsets given as link lists for the GCRP. We claim that the algorithm runs in time O(m on a random access machine. As usual we assume that indexing int arrays of size m is possible in unit time but that bit vector operations of size m cannot be done in one step.

The analysis of the subroutine Buildtree is standard. The only points of interest are the manner of finding the B_i that intersect B_{index} in step 1, the sorting done in step 4, and the checking for bad junctures in a jump-free subtree. Given linked lists for the A_i we can in time O(m), construct a list for each j of the sets A_i that contain j. Thus when removing the element j from B_{index} we can locate all sets intersecting B_{index} at j. Aft removing all elements from B_{index} we have all B_i intersecting B_{in}

Now consider the problem of sorting the neighbor list by size of $A_i \cap A_{index}$. At the start of the recursive call Build-tree(index) assume we have a sorted list of $|A_i \cap$(tree already explored)$|$ (at the first call the list is empty.) Apply the following procedure.

> <u>for</u> <u>each</u> A_i on the list in decreasing size of intersection
>   <u>if</u> A_i is not marked
>   <u>then</u> mark all the A_j on the list
>   that intersect B_i with an "i"
> partition the original list into lists of A_i having
>   the identical marks (preserving the original order)

Each sublist corresponds to a son of A_{index} and is passed as a list at the recursive call for the son. The sons are the A_i that appear at the beginning of each sublist. The other sets on the sublist correspond to the sets in the juncture of A_{index} with the son.

The sublist passed to a recursive call can be used to determine whether the junction forms a nest. Since the list is sorted by size of intersection with the father, it is easy to determine if the sets are ordered by inclusion.

<u>Intractability of a More General Problem</u>

Many instances of GCRP that have no solution may be solved if each record is allowed to have two or more NEXT pointers. This leads naturally to the following definition.

Generalized Consecutive Retrieval Problem of Order k (k-GCRP
Given a pair (A,B), A a set of records $\{1,\ldots,n\}$ and B a collection $A_1,\ldots,A_m$ of subsets of these records (constraint is it possible to construct k trees on $\{1,\ldots,n\}$ such that each A_i is linear in at least one tree?

For all positive integral values of k, k-GCRP is clearly in NP. In this section we show that for $k > 1$ the k-GCRP is NP-complete. We do this for $k > 2$ by exhibiting a polynomial time reduction of the vertex k-colorability problem [1] to the k-GCRP. For $k = 2$ we exhibit a polynomial time reduction of the NAE-SAT problem (Schaefer[4]) to the 2-GCRP. (An instance of the *Not-all Equal Satisfiability* problem consists of a set of variables $\{x_1,\ldots,x_n\}$ and a set of triples $\{(x_{i_1},x_{j_1},x_{k_1})$, ... , $(x_{i_m},x_{j_m},$ drawn from these variables. The problem is to determine if there exists an assignment of two colors to the variables such that for no r does $x_{i_r} = x_{j_r} = x_{k_r}$).

In the reductions that follow we make use of an object calle a k-sink. Informally, a k-sink is an instance of the k-GCRP that has a distinguished vertex v such that in any solution every tree contains an arrow going out of v. A k-sink is a device that is used for forcing arrows to go in a particular direction. If we take a new point x, not in the k-sink, and insist that there be an edge between x and the distinguished point v, then this edge must be directed from x to v in any solution.

Formally, we define a <u>k-sink</u> to be an instance $S = (A, \{A_1, \ldots, A_m\})$ of k-GCRP with a distinguished element v in A, such that

(1) S has a solution, and

(2) in no solution is the vertex v the root of any tree.

An example of a 2-sink is shown in Fig. 21. (Included, but not shown, are all possible three-element constraints that contain the distinguished vertex, 1.)

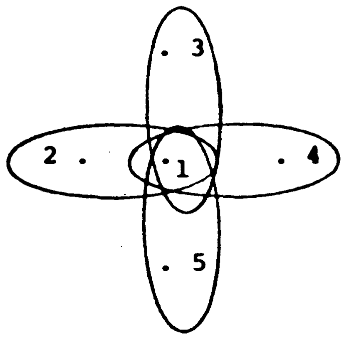


Figure 21
A 2-sink with distinguished vertex 1

<u>Lemma</u> NAE-SAT $\leq_p$ 2-GCRP

<u>Proof</u>

Let F be an instance of NAE-SAT, i.e. let F be a set of triples in the variables $x_1, \ldots, x_n$. Construct an instance of 2-GCRP as follows: Let v and w be the distinguished vertices of distince 2-sinks. For each variable x_i make the two constraints

$\{x_i,v\}$, $\{x_i,w\}$. For each triple (x_i,x_j,x_k) in F, add three new points a_i, a_j, a_k and the constraints

$\{a_i,\ x_i,\ v\}$,
$\{a_j,\ x_j,\ v\}$,
$\{a_k,\ x_k,\ v\}$.

These constraints imply that the paths $a_i \to x_i \to v$, and $a_j \to x_j \to v$, and $a_k \to x_k \to v$ must appear in any solution. Connecting a_i,a_j and a_k by the constraints $\{a_i,a_j\}$, $\{a_j,a_k\}$ and $\{a_k,a_j\}$ insures that not all three paths can lie in the same tree. Thus the resulting 2-GCRP has a solution if and only if the formula F is NAE satisfiable. □

From this we conclude:

<u>Theorem</u> 2-GCRP is NP complete

<u>Constructing k-sinks for $k \geq 1$</u>

The k-sink we construct will have r elements and k(r-1) two-element constraints (edges); the value of r is determined shortly. By a simple counting argument we observe that any arrow in a solution has to lie along one of these edges; otherwise, there would have to be a cycle.

Suppose that among the r elements and k(r-1) two-element constraints there exist elements $u_1,\ldots u_k$, $w_1,\ldots,w_k$ and v such that for all i, (u_i,v) and (w_i,v) are constraints but (u_i,w_i) is not a constraint. By adding the k three-element constraints $\{u_i,v,w_i\}$

we insure that (u_i,v) and (v,w_i) are in the same tree. Since there are exactly k trees and k values of i, v cannot be the root of any tree. Fig. 22 illustrates how a distinguished element is produced.

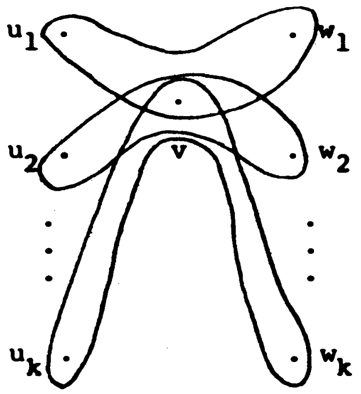


Figure 22
The distinguished element v in a k-sink

<u>Lemma</u> For all $k \geq 1$, there exists a k-sink

<u>Proof</u>

Let r be an even integer larger that $4k-2$ that is relatively prime to all odd integers less than 2k. Let $A = \{0,1,\ldots,r-1\}$. Starting at 0, make the $k(r-1)$ constraints

$$\begin{array}{llll}
\{0,1\} & \{1,2\}, & \ldots & \{r-2,r-1\}, \\
\{0,3\} & \{3,6\}, & \ldots & \{3(r-2),\ 3(r-1)\}, \\
\vdots & & & \vdots \\
\{0,2k-1\}, & \{2k-1,2(2k-1)\}, & \ldots & \{(2k-1)(r-2),(2k-1)(r-1)\}
\end{array}$$

where each integer is interpreted modulo r. One can view the r points as being arranged in a circle. Since r is relatively prime to all the odd integers less than 2k, each row of constraints forms a simple path of length r-1. In fact each path is a cycle minus one edge. Let v be the element 2k-1. Then v is in the 2k two-element constraints

$$\{v-1,v\},\ \{v,v+1\}$$
$$\{v-3,v\},\ \{v,v+3\}$$
$$\vdots$$
$$\{v-(2k-1),v\},\ \{v,v+(2k-1)\}.$$

Since the difference between v-i and v+i is even, there are no two-element constraints {v-i,v+i}. Therefore, by adding the constraints {v-i,v,v+i}, i = 1,3,...,2k-1 we make certain that v satisfies condition (2) for a k-sink.

Clearly condition (1) can be satisfied by the k trees

$$0 \to 1 \to 2 \to \ldots \to r-1$$
$$0 \to 3 \to 6 \to \ldots \to 3(r-1)$$
$$\vdots$$
$$0 \to 2k-1 \to 2(2k-1) \to \ldots \to (2k-1)(r-1). \qquad \Box$$

We now reduce graph k-colorability to k GCRP.

<u>Lemma</u> Vertex k-colorability $\leq_p$ k-GCRP

<u>Proof</u>

Let (V,E) be an undirected graph. Construct a k-GCRP as follows. Let $s_1,\ldots,s_k$ be the distinguished vertices of k k-sinks. For each v in V add the k constraints $\{s_i,v\}$. For each edge (v,w) in E add a new point x and the two constraints $\{s_1,v,x\}$, $\{s_1,w,x\}$ to the k-GCRP. This forces $x \to v \to s_1$ to be a path in some tree as well as $x \to w \to s_1$ to be a path in a tree. Since x can have out degree at most one in any tree, these two paths must lie in distinct tree. Thus this k-GCRP has a solution if and only if (V,E) is k-colorable. □

We conclude:

<u>Theorem</u> If $k \geq 2$, then k-GCRP is NP complete.

REFERENCES

1. Aho, A., Hopcroft, J., and Ullman, J., The Design and Analysi of Computer Algorithms, Addison-Wesley (1974).

2. Booth, K. S. and Lueker, G. S., "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms. JCSS 13 (1976), 335-379.

3. Ghosh, S. P., "File Organization: The Consecutive Retrieval Property," ACM 9 (1972), 802-808.

4. Schaefer, T., "The Complexity of Satisfiability Problems," Proc. 10th SIGACT (1978), 216-226.

5. Truszczynski, M., "The Theorem Characterizing the Acyclic Families of Sets,", ICS Polish Academy of Science Reports 314 (1978).