PATTERN RECOGNITION IN LAYERED TRACK CHAMBERS USING A TREE ALGORITHM

D.G. CASSEL * and H. KOWALSKI

Deutsches Elektronen-Synchrotron DESY, Hamburg, Germany

We present a new approach to the pattern recognition problem in multi-layer track chambers. Tracks are constructed from locally related hits using a tree algorithm. This approach has worked well in quite different track finding problems in the large cylindrical drift chamber in the TASSO detector at PETRA. Its success in analyzing complicated e^+e^- jet events indicates that it can be applied to other problems, including the complicated jet topologies expected at higher energy.

1. Introduction

High transverse momentum events with hadron jets have become a major preoccupation of high energy physicists. These events are difficult to analyze because of their tightly collimated jets with high multiplicities and secondary vertices. Multi-layer drift chambers have demonstrated their effectiveness in detecting the charged particles in these events. However, they aggravate the already difficult track finding problem, since the left–right ambiguity arising from the conversion of drift time to space points yields background hits that are inherently near the real tracks. Fig. 1 illustrates this problem with a high energy e^+e^- annihilation jet event in the TASSO detector [1] at PETRA, shown with and without the tracks found. Utilization of drift chambers to detect jet events obviously requires fast and efficient programs that accurately solve the pattern recognition problem. At higher energies, where jets are expected to be more collimated and have higher multiplicity, the demands on program performance will undoubtedly increase.

In this paper we describe a new approach to track finding in multi-layer track chambers which we call a link-and-tree method and which we have successfully applied in the TASSO experiment [2]. We gather hits into links (in the simplest case pairs of points in

nearby layers) and construct lists of links that are locally related to each other (e.g. two links with a hit in common and nearly the same slope would be related). We then construct complete tracks as unbroken chains of links from these lists of related links using a fast tree search algorithm. Other track recognition methods have also used links and their local relationship [3] but we have taken a different approach which appears to be particularly appropriate for the problems arising with jets in drift chambers. The versatility of our approach is indicated by its application in two very different problems in the TASSO analysis, the rapid rejection of background in the first stage, and the exhaustive search for tracks in the last stage. In both situations we find that this approach leads to fast, efficient and highly structured programs that give us a measure of control that we have found lacking in our previous experience.

In section 2 we introduce our approach by comparing it with a naive version of the more familiar road procedure. In section 3 we define the data structures we used and show how the three algorithm combines them into track candidates. In section 4 we show how to use this strategy in an exhaustive search for tracks. Section 5 describes the especially fast search for high quality tracks using Minimal Spanning Tree ideas. We conclude with a few suggestions for extensions in section 6 and a summary in section 7.

To make this paper self-contained, we include a few details of the construction of the relevant parts of the TASSO inner detector in appendix A. In appendix B, we collect a few geometric expressions and give some details of the TASSO track finding that are not directly related to the new strategy.

---

* On leave from Cornell University, Ithaca, N.Y., U.S.A.

TASSO

(a)
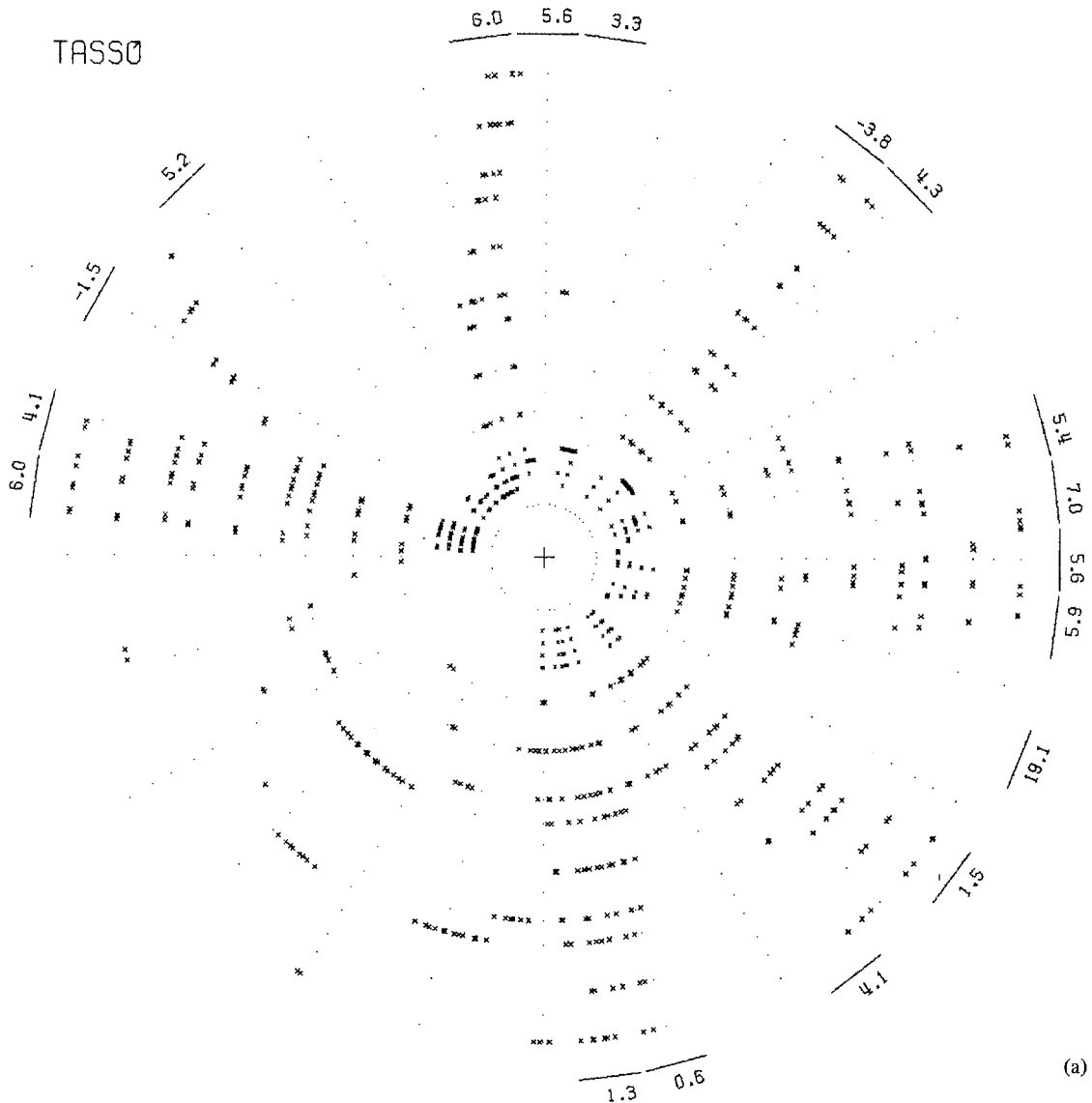
2. General

To illustrate how our approach works, we compare it with a road [4] procedure in a considerably simplified example. Fig. 2 illustrates a straight track in a drift chamber, along with the extra hits resulting from the left—right ambiguity. The road method takes any two hits from the innermost and outermost layers and gathers all the hits within a road around the track determined by these two hits. From our experience in working with this method the first difficulty is that the road generally cannot be too narrow, since the track parameters determined from two hits may be imprecise. Therefore, there is often more than one hit per layer inside the road as shown in fig. 3, and the best track has to be found by fitting all permutations of hits within the road. Fitting is slow, so fitting the large number of possible permutations requires substantial CPU time. The second difficulty results from chamber inefficiencies, which forces the program to cycle through a substantial fraction of all layer pairs to choose the hits that define roads. This means that many of the permutations will be encountered again and again as different layer-pairs are used. These complications diminish the apparent simplicity of the method and make it difficult to understand the
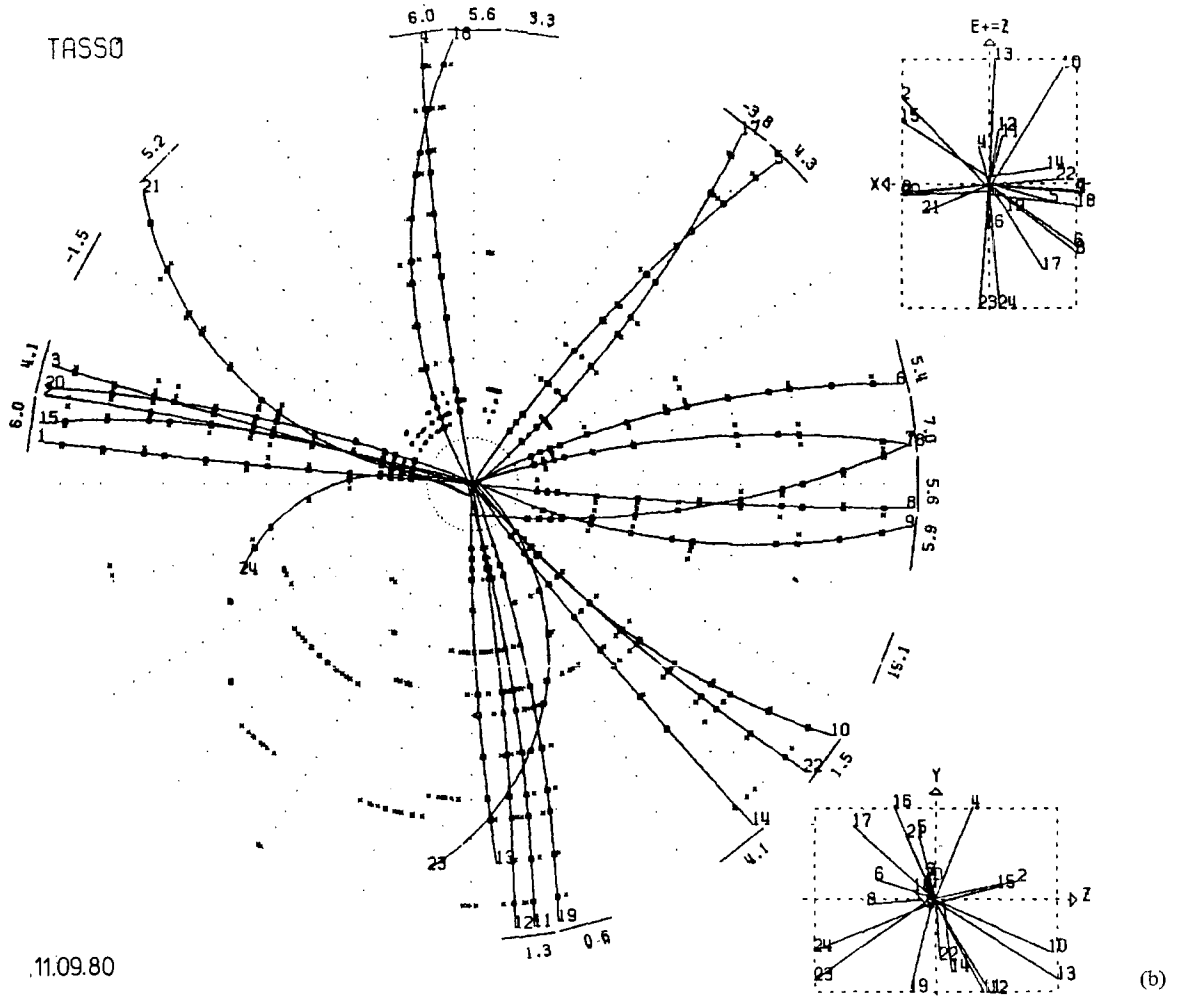
Fig. 1. A typical TASSO one photon annihilation hadronic event (a) withough track reconstruction; (b) with track reconstruction.

performance. For example, to be sure that all possible track candidates with at most two missing chambers have been found, it is necessary to carry the loops over chamber pairs nearly to completion. The question of how many road-defining pairs to use is a cut that must be studied and optimized along with other cuts. Since it is generally impossible to fit all permutations of hits in all possible roads, programs that use this approach usually include strategies for reducing the number of roads and permutations. This further obscures understanding the performance.

In our procedure, speed, clarity and control are obtained at the price of working with objects which are more complicated than single hits: in this example, the elementary objects are pairs of hits called "links" illustrated in fig. 4 and the "elementary

trees" composed of them. Since a link has two hits, a link already determines the parameters of a straight track or a circular track coming from the interaction point. The elementary tree of any given link is composed of this link, which is then called the trunk of the elementary tree, and all links having the vertex in common and with approximately the same values of track parameters. (These links are then called branches of the elementary tree.) Of course, a trunk of one elementary tree can also be a branch of another one. Examples of elementary trees are shown in fig. 5. (In this case the track parameter is the slope of the link.) The common vertex and the near equality of track parameters are the local relationship between links mentioned above.

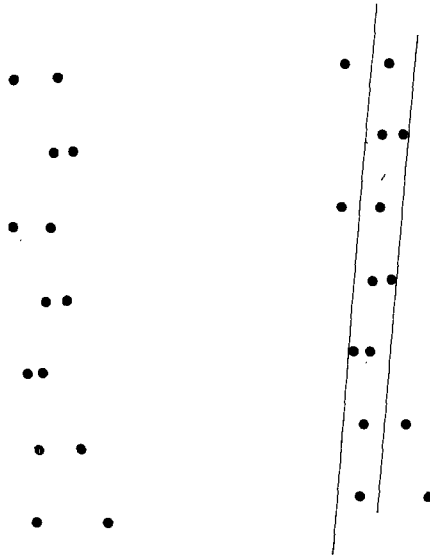The elementary trees can be combined to form a

Fig. 2. A straight track in a drift chamber along with the extra hits from the left–right ambiguity (the horizontal scale is considerably larger than the vertical one).

Fig. 3. The straight track of fig. 2 together with the road determined by a pair of hits.

full tree as shown in fig. 6. * The correct track is immediately recognized as the longest chain of links in the full tree. (The length of a chain is the number of links in it.) In the program a fast algorithm climbs the full tree from the root link, locating all chains from the root to any leaf but never finding the same chain twice. In this way tracks (or more precisely – reliable track candidates) can be recognized without fitting; the only necessary measure of global quality is the length of the chain. This is due to the fact that the local cuts that define the elementary trees tend to break chains containing bad hits. How this work is shown in fig. 7 which illustrates a track that would appear within the road in fig. 3. This track does not appear as a long chain, since the potential elementary tree consisting of links 5 and 9 does not appear. This is due to the fact that the angle η is roughly twice as large as ϵ so links 5 and 9 fail the equal slope cut if we assume that ϵ is greater than about half of the cut angle.

---

* We borrow the term tree from the graph theory [5] where our links are called edges. Our trees may technically not be trees since they may have joining branches but this does not affect our algorithm.
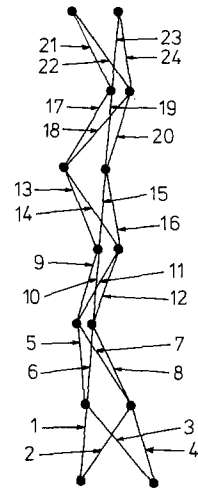

Fig. 4. A set of one-gap links corresponding to a hit pattern of the track from fig. 2. The numbers indicate the indices of the links in the link list.
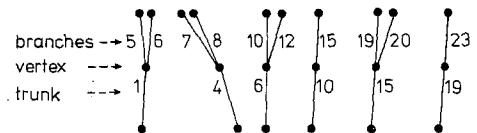

Fig. 5. List of the elementary trees composed of links of fig. 4. The selection criterion is the approximate equality of slopes.
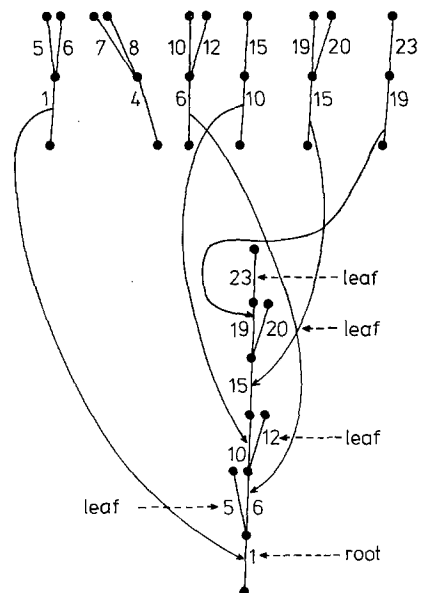

Fig. 6. A full tree composed from some elementary trees of fig. 5 and containing the straight track from fig. 2.
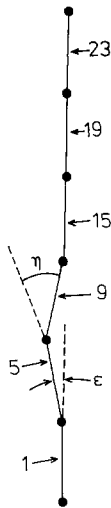
Fig. 7. Example of a hit combination which has to be considered as a track candidate in the road but not in the tree method. In the tree method this track does not appear as a long chain since the elementary tree consisting of links 5 and 9 does not exist. This is because η is roughly twice as large as ϵ so the links 5 and 9 fail the equal slope cut if we assume that ϵ is greater than about half of the cut angle.

If more than one reasonable track is found, additional global quality cuts like the χ^2 from a fit to the track can be applied. In our experiences, this fitting is not a large burden, due to the high selectively in finding the tracks and the fact that each track is found only once.

Of course this procedure becomes more complicated when inefficient layers and wild hits are taken into account. They require simple extensions described in section 4, which retain the high selectivity and the guarantee that all tracks consistent with the local cuts will be found with none being found twice.

3. Links, elementary trees and full trees

A link is a segment of a potential track with enough hits to determine the track parameters of the segment. For example, if the tracks being sought are straight lines or circles coming from the interaction point, only two hits are required to define the track parameters, z_0 and $\tan \lambda$ for straight tracks or R and ϕ_0 for circles. (For notation see appendix B). If the tracks are circles originating at arbitrary points (e.g. decay vertices) at least three hits are necessary to define a link since three points define a circle. So far

we have applied out method only to straight tracks and circles coming from the origin, so for simplicity we will always assume that two hits determine a link.

Since every pair of hits in different layers defines a link *, the number of all possible links is far too large to work with. The number of links that must be considered can usually be reduced by limiting the number of hits that must be considered in several ways. First, we do not construct all links in the whole chamber at once; we rather work in a specified region of the chamber (for details see sect. 5 and appendix B)

The second reduction is to consider only those links whose track parameters are within an interesting range. In the search for $s-z$ tracks, only links whose z_0 is within a fiducial region around the interaction point are used. In the fast search for circular tracks, we accept only those links whose radius, R, is larger than a minimum value $R_{\min}$.

The number of links remaining after this preselection is still too large. Further reduction emerges from the obvious strategy of first searching for tracks with the smallest number of missing hits. For example, an exhaustive search for tracks naturally divides into a number of successive steps. In the frist step tracks with no missing hits are sought. The next step looks for tracks with one hit missing for any reason, the third step seeks tracks with two missing hits, and so forth. This strategy reduces the number of links needed, since, for example, in the first step only links between hits in adjacent layers are required. This leads us to classify links by the number of gaps they span, where gap refers to the space between two adjacent layers. In searching for tracks with no missing hits we require only one gap links. In the next step we also require two gap links, and so forth. Therefore the best tracks are found using relatively few links.

The fourth possibility of reducing the number of links occurs after a track is found. Its hits and their left—right twins are cancelled from the hit list, which in turn diminishes the number of links which can be created while searching for further tracks. Note that this interplay between finding the best tracks early and cancelling hits leads to a considerable reduction of the number of links as tracks are found, which is essential for tracking within a jet.

The links required at a given stage of the search

* Since in TASSO the tracking is made always in projections, see appendix B, the links are created only between hits from layers of the same projection.

are combined into elementary trees. As stated in sect. 2, the elementary tree consists of one link called the trunk, together with all links having the middle point in common, and with approximately the same value of the track parameter. For example, in the search for straight lines, the slope $a = \tan \lambda$ and intercept $b = z_0$ are calculated for each link being considered. Two links 1 and 2 with a hit incommon are associated into an elementary tree if they satisfy the cuts,

$$|a_1 - a_2| < \Delta a , \quad |b_1 - b_2| < \Delta b . \tag{1}$$

The magnitudes of the cuts Δa and Δb are determined from the measurement errors for single hits propagated to a and b. It is usually sufficient to check only the slopes, since the slopes and intercepts are strongly correlated, expecially after the z_0 cut mentioned earlier. The cuts used for cirlces coming from the origin will be discussed in sect. 5. Obviously, since building elementary trees occurs in inner loops of the program, the cuts used must be made on quantities that can be rapidly calculated from the measured coordinates of the hits.

The links involved in elementary trees are stored in a list. The elementary tree associated with link ILINK is specified by two arrays, NNEXT(ILINK) and INEXT(IBRNCH, LINK). NNEXT(ILINK) is the number of branches attached to link ILINK, and IBRNCH runs from 1 to NNEXT(ILINK), if LINK has any branches. INEXT (IBRNCH, ILINK) is the index of branch IBRNCH of link ILINK. Of course, NNEXT(ILINK) equals zero means that there are no branches for that link. The word "next" really describes a relationship between a trunk and its branches; the branches come next after the trunk, along the path of the particle. INEXT points from a link to all others related to it by "next".

A full tree is a tree built on a preselected root link. How root links are selected depends on the application and is described later. The tree is built by associating branches with the root link and then iterating the process with all branches of all links in the tree. Although fig. 6 illustrates a full tree built on link number 1, there is not structure in the program that actually represents it. Instead, we use a fast algorithm which climbs up all possible paths on the tree and recognizes tracks as chains of links. This algorithm, which is of the Depth-First Search type [6], is the core of the program. In the following we will give a detailed description of its realization in our application.

We illustrate the operation of the climbing algorithm on the full tree of fig. 6. The algorithm is given the address of the starting link, in this case, ISTART = 1 and the arrays NNEXT and INEXT. It woks with its own depth counter IDEPTH, link index ILINK, and two arrays, LNKLST( ) and NBRNCH( ), both indexed with IDEPTH. LNKLST is the list of links climbed in the chain being created. NBRNCH(IDEPTH) keeps track of the number of branches of the link LNKLST(IDEPTH) that have been already climbed. If NBRNCH(IDEPTH) = NNEXT(LNKLST(IDEPTH)) then all branches from the link LNKLST(IDEPTH) have been processed.

Since climbing-up the full tree from the one link to another is obviously recursive, it would be easier to describe it in a language that supports recursive procedures. Although FORTRAN does not accommodate recursive structures, it is possible to realize the algorithm, at the cost of some clarity, with interleaved GO TO statements. This is illustrated in fig. 8. Fig. 9 shows the algorithm written in an extension of FORTRAN called FLECS. FLECS is a FORTRAN preprocessor that translates block structure control statements into normal FORTRAN statements. Recursion is not supported in the sense that a stack is kept to

```
       SUBROUTINE CLIMB (IROOT,NNEXT,INEXT )

       INTEGER LLINK(*),NBRNCH(*),NNEXT(***),INEXT(**,***)

       IDEPTH = 0
       ILINK  = IROOT

   10  CONTINUE      ! CLIMB-UP-ILINK

       IDEPTH = IDEPTH + 1

       NBRNCH(IDEPTH) = 0
       LLINK (IDEPTH) = ILINK

   20  CONTINUE      ! CLIMB-UP-ILINKS-NEXT-BRANCH

       NBRNCH(IDEPTH) = NBRNCH(IDEPTH) + 1

       IBRNCH = NBRNCH(IDEPTH)

       IF (IBRNCH .GT. NNEXT(ILINK)) GO TO 30

       ILINK  = INEXT(IBRNCH,ILINK)

       GO TO 10

   30  CONTINUE      ! CLIMBING STOPPED AT THE TOP OF A CHAIN - SAVE IT

       CALL SAVCHN(IDEPTH,LLINK )

   50  CONTINUE      ! CLIMB-DOWN-ILINK

       IDEPTH = IDEPTH - 1

       IF (IDEPTH .EQ. 0) RETURN

       ILINK  = LLINK (IDEPTH)

       IF (NBRNCH(IDEPTH).LT.NNEXT(ILINK)) GO TO 20

       GO TO 50

       END
```

Fig. 8. A FORTRAN version of the subroutine CLIMB.

```
SUBROUTINE CLIMB (IROOT,NNEXT,INEXT)

INTEGER LLINK(*),NBRNCH(*),NNEXT(***),INEXT(**,***)

IDEPTH = 0
ILINK = IROOT

CLIMB-UP-ILINK

CLIMB-ILINKS-NEXT-BRANCH

TO CLIMB-UP-ILINK
.   IDEPTH = IDEPTH + 1
.
.   NBRNCH(IDEPTH) = 0
.   LLINK (IDEPTH) = ILINK
.
...FIN

TO CLIMB-ILINKS-NEXT-BRANCH
.   NBRNCH(IDEPTH) = NBRNCH(IDEPTH) + 1
.
.   IBRNCH = NBRNCH(IDEPTH)
.
.   IF (IBRNCH .LE. NNEXT(ILINK))
.   .   ILINK = INEXT(IBRNCH,ILINK)
.   .   CLIMB-UP-ILINK
.   .   CLIMB-ILINKS-NEXT-BRANCH
.   ...FIN
.
.   IF (IBRNCH .GT. NNEXT(ILINK))
.   .   CALL SAVCHN(IDEPTH,LLINK )
.   .   CLIMB-DOWN-ILINK
.   ...FIN
.
...FIN

TO CLIMB-DOWN-ILINK
.   IDEPTH = IDEPTH - 1
.
.   IF (IDEPTH .EQ. 0) RETURN
.
.   ILINK = LLINK (IDEPTH)
.
.   IF (NBRNCH(IDEPTH).LT.NNEXT(ILINK)) CLIMB-ILINKS-NEXT-BRANCH
.
.   IF (NBRNCH(IDEPTH).GE.NNEXT(ILINK)) CLIMB-DOWN-ILINK
.
...FIN

END
```

Fig. 9. The CLIMB subroutine written in FLECS, a preprocessor for FORTRAN that accommodates block structures.

allow the program to unwind from a sequence of recursive procedures, but that is not important for our purposes. The organization of the algorithm into the blocks illustrated clarifies its logic. The program is to be read like ordinary FORTRAN, except for the FLECS control statements. The statements between an IF and the associated FIN are to be executed if the logical condition is satisfies. Likewise, the TO statements define internal procedures terminated by the associated FIN. A procedure is executed whenever the string following the TO appears elsewhere in the program. All variables in the subroutine are available to all of the procedures, and the RETURN statement

results in a normal exit from the whole subroutine.

The subroutine begins with no links climbed and the next link to climb being the starting link, ISTART, so IDEPTH is set to zero and ILINK to ISTART. This is illustrated in fig. 10a where all links are drawn as dashed lines to show that no links appear in LNKLST yet. The next step is to climb the starting link, using the procedure CLIMB-UP-ILINK. This procedure just stores the new link found in the array LNKLST and sets the branch counter to its starting value. It is done by incrementing IDEPTH, updating LNKLST with ILINK, and clearing the branch counter NBRNCH(IDEPTH). The result is represented in fig. 10b by a solid line for link 1 to indicate that it now appears in the link list.

Recursive climing is done by the procedure CLIMB-ILINKS-NEXT-BRANCH. It first checks whether there are branches to be climbed, so NBRNCH(IDEPTH) is incremented by 1 and then compared to NNEXT (ILINK). If NBRNCH is less than or equal to NNEXT, there are branches to climb, so ILINK is updated from INEXT and the new link is climbed by CLIMB-UP-ILINK. Then the whole process is iterated by having CLIM-ILINKS-NEXT-BRANCH reference itself. The procedures CLIMB-ILINKS-NEXT-BRANCH and CLIMB-UP-ILINK climb from one link to another following the chain of links, and store every new link in the array LNKLST, until a link is found which has no branches at all. (This occurs when NBRNCH is larger than NNEXT.) This link, which is a leave, is the end of a chain so the subroutine SAVCHN is called to save this chain which is so far only stored in LNKLST. This chain is one of the track candidates in the tree. In the example of fig. 10 the first completed chain (fig. 10c) consists of links 1 and 5 since link 5 has no branches. After the end of a chain is found and stored, the procedure CLIMB-DOWN-ILINK is started. In this procedure, IDEPTH is decremented by 1 which means that control is passed to the previous link in a chain. If this link has unused branches * the procedure CLIMB-DOWN-ILINK is recursively continued; it quits with a RETURN statement when it has climbed down link ISTART so there are no more links in the array LNKLST (i.e. IDEPTH equals 0).

---

* If the full tree being climbed has joining branches some branches will be climbed more than once, but these branches will always have different chains below them each time they are passed. Clearing NBRNCH in CLIMB-UP-ILINK takes care of all potential problems.
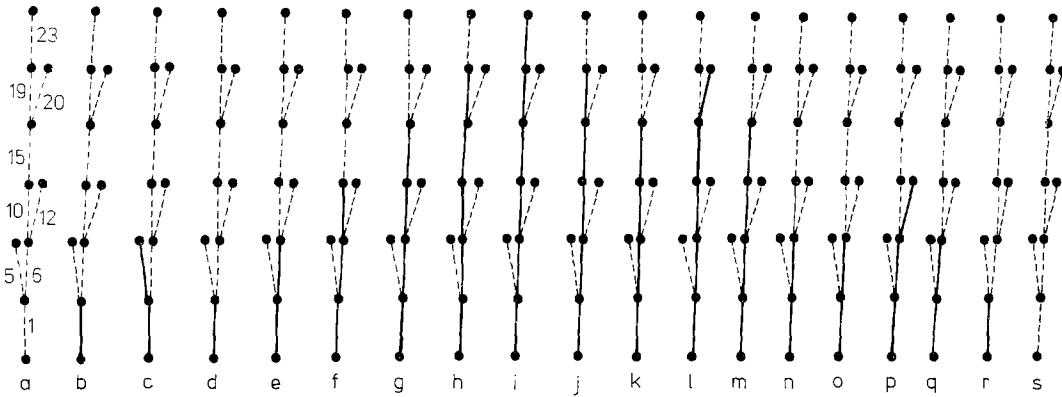
Fig. 10. The graphical representation of the action of the subroutine CLIMB on the tree of fig. 6. The full lines represent links which are actually stored in the array LNKLST at any point in CLIMB.

Figs. 10a—s illustrate the operation of the subroutine CLIMB by showing the content of the array LNKLST in all program steps. In this figure the letters a, b, c, d ... s denote all updates of the array LNKLST and the full lines represent links which are actually stored in this array. The output of the subroutine CLIMB consists of all completed chains found during its operation. In fig. 10 these are the chains c, i, l and p. SAVCHN stores the chains that satisfy its cuts in an appropriate list. The cuts used depend on the application and two examples will be mentioned in the next two sections.

In the TASSO experiment the tree algorithm was written in FORTRAN as shown in fig. 8 to maintain program compatibility. It is very fast since it accesses the precomputed information stored in the links and elementary trees by updating counters and computing indices. The execution time is linear in the number of links in the full tree, since each link is passed only once going up and once going down.

In defining elementary trees, we mentioned that the branches of an elementary tree were farther from the interaction point than the trunk. We could have defined trees with the branches closer to the interaction point than the trunk. In fact, such elementary trees are also needed in the procedures in Sections 4 and 5. We used additional arrays, IPREV and NPREV to list branches which are closer to the interaction point than the trunk, since the branches are "previous" to the trunks instead of "next" after them. No additional computations are involved in filling IPREV and no new links are added, since link i is previous to link j if link j is next to link i. From a given root link CLIMB moves towards or away from the interaction point in looking for chains, depending on whether the "previous" or "next" pointers appear in its argument list.

4. A systematic search for tracks

The highly systematic and exhaustive pattern recognition strategy we describe here, was used in finding straight tracks in the $s-z$ plane determined by combining circular tracks in the $r-\phi$ plane with the hits in the 6 stereo layers (see appendix B). Of course, the strategy is more general than straight tracks in 6 layers, but we stick to this example here, since extensions are obvious. The goal was to rapidly and efficiently find the $s-z$ track associated with a given $r-\phi$ track (this is explained in more details in appendix B), even if this track had only 3 hits, while giving preference to tracks with 6 hits over those with 5, etc. This is accomplished by organizing the search in the following stages [*]:

1) All 6-hit chains are found using the tree procedure, fitted and stored in a track candidate list.

2) A 6-hit track is accepted and the search stops if the track candidate with smallest χ^2 among those passing a z_0 cut also passes the χ^2 cut.

3) Otherwise all 5-hit track candidates are found, fitted and stored.

_____

[*] We discuss here only the organization of the program for finding the $s-z$ track itself. Other essential steps, in particular finding the necessary $r-\phi$ track and cancelling the hits and ambiguity twins belonging to the $r-\phi$ and $s-z$ tracks, are assumed to occur outside of this program.

4) A 5-hit track is accepted if one is good enough according to stage (2).

5) Otherwise all 4-hit track candidates are found, fitted and stored.

6) A 4-hit track is accepted if one is good enough according to stage (2).

7) Otherwise the z_0 and χ^2 cuts are relaxed and the lists of 6, 5, and 4-hit tracks were again searched in that order. The search stops when the best track with a given number of hits is found to be acceptable.

8) If no 6, 5 or 4-hit track is acceptable, 3-hit tracks are considered with tighter χ^2 cuts. This is trivial since all 3-hit tracks are all pairs of links related by the tan λ and z_0 cuts for links.

We achieve this organization which guarantees that all tracks with a given number of hits have been found at stages (1), (3) and (5) by controlling the links used in elementary trees. We do not avoid finding some 5 and 4-hit chains in stage (1), [and 4-hit tracks in stage (3)], but they appear only once, so we save them for later use if necessary.

We begin this search for tracks by establishing the list of hits in the $s-z$ plane as described in appendix B. Building the list of elementary trees proceeds then in organized steps. We have already mentioned that classifying links according to the number of gaps spanned is the key to controlling the number of missing hits in found tracks. Fig. 11 illustrates the complete classification for 6 layers. 6-hit tracks obviously can be built only with links of types 1–5, and a 6-hit track is a chain with one link of each type. At the other extreme, links of type 15 span layers 1–6, so they can only be used for 2-hit tracks; they did not appear in the program. Of course, links of types 6–9 are required for 5 hit tracks, to account for inefficiencies in layers 2–5. Links of types 10–12 are needed in 4-hit tracks in which two successive layers can be inefficient *. Finally, types 13 and 14 can appear in only 3-hit tracks.

For each link that occurs in an elementary tree we store NNEXT, INEXT, NPREV and IPREV. We also store the hit address in the $s-z$ hit list of the hits on the inner and outer end of the link. The elementary tree list is organized with pointers [8]. This organization enables us to find links of a given type and to update rapidly the "next" and "previous" pointers

---

* Track finding programs often limit the number of succesive missing layers in a track. This is accomplished automatically in our strategy by limiting the types of links used.
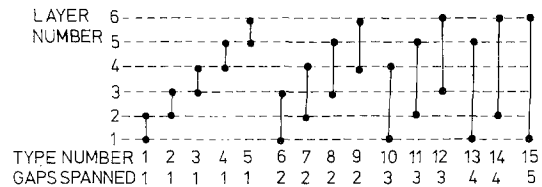

Fig. 11. Complete classification of all links for 6 layers.

when a link already in the list is associated with a new link in an elementary tree.

After preparing this elementary tree list we loop through all links of type 1 in the list and use them as starting links for the tree algorithm. Only the "next" pointers are used here since climbing can only move outwards from type 1 links. The chains saved by SAVCHN are examined and 6-hit ones are fitted and saved in a track candidate list. In this step also some chains constructed from one-gap links but with lower number of hits are found. They are also fitted and saved in the same list. This list is also organized with pointers [8], so all tracks with a given number of hits can be rapidly examined to facilitate stages (2), (4), (6) and (7). At this point stage (1) is finished–all 6-hit track candidates have been found. Step (2) is accomplished by looping through the 6-hit track candidates and rearranging the pointers to order the track candidates list according to increasing χ^2. The first track with an acceptable z_0 is then the desired track if χ^2 is small enough.

If no acceptable 6-hit track is found the search continues with stage (3). There are now 3 ways to get 5-hit track candidates not already found,

(3a) by climbing outwards from all type 2 links,

(3b) by copying 6-hit tracks and deleting the hit in layer 6,

(3c) by introducing elementary tracks with links of types 6–9 and climbing trees in *both* directions from these links.

In step (3c) track candidates are obtained by combining all top chains found by climbing outward from a given link with all bottom chains found by climbing inward from this link. Fig. 12 illustrates this with a type 7 root link, where the chains 12b–12j arise from the full tree 12a.

In order to avoid finding the same chains repeatedly, (3c) is organized in steps. In each step, a new link is introduced and elementary trees are constructed by combining these links only with links of *lower* type number. In step (3c) these are consecutively the links of types 6, 7, 8 and 9. As in step (1),
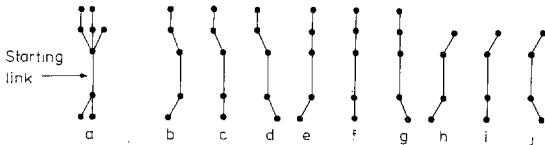
Fig. 12. Track candidates (chains) obtained by combining top chains and bottom chains starting with a type 7 link.

some chains with fewer than 5 hits can be found. For example starting from a link of type 8 it is possible to construct a 4-hit chain using a link of type 6.

Finding the remaining 4-hit track candidates in stage (5) duplicates the logic of stage (3),

(5a) by climbing outwards from type 3 links,

(5b) by copying 5-hit tracks and deleting the last hit

(5c) by introducing elementary trees with type 10–12 links and climbing trees in both directions from the new link.

The systematic search described here leads to a track finding program whose quality is equivalent to fitting all possible hit permutations and choosing the best ones according to χ^2 and number of hits attached. It has a slight bias towards increasing the number of attached hits in a track since if there are two track candidates, both fulfilling the χ^2 cut, the track with the higher number of hits is chosen. This bias is easily controlled by keeping the χ^2 cut reasonably low.

In spite of its high quality the program is reasonably fast. The average IBM 370/168 CPU time consumption in e^+e^- jet events at 15 GeV beam energy is 10–12 ms per s–z track. The high speed of the method is in our view mainly due to three reasons:

a) The time consuming operations of following the track and fitting is in our approach largely replaced by chain construction from locally related links. The determination of local relations demands only few multiplications and divisions and the construction of chains is a pure integer operation.

b) The information which was worked out once is not lost. For example the information contained in one-gap links, which is necessary for the 6-hit track search is still required for 4-hit tracks.

3) Every chain is, with only one exception, constructed only once. The exceptional double construction occurs in step (5b) where the same 4-hit track can be found by deleting two different last hits. Of course, a quick search in the track list can eliminate this duplication. Note that in the road method there are usually many ways in which the same track candidates can be found.

5. A fast search for tracks

A fast track finding program based on our tree climbing algorithm is used as the event filter in the first pass data reduction in the TASSO experiment. It must recognize a good event (i.e. one with circular tracks in the r–ϕ plane coming from near the origin) in a background of 10^4 bad triggers. Therefore, it is optimized for speed at the price of some efficiency by working mainly with one-gap links and building chains analogous to minimal spanning trees [9].

Since we are searching for tracks that are circles through the origin, the links in this program are arcs of circles joining hits in two nearby layers. Our starting point was the observation that a real track will nearly always be the chain with shortest geometric length spanning all layers of the chamber. Except for a few special cases, this is obviously true if the real track is compared with a chain made with at least some left–right twin hits.

However, it is not practical to use arc length in recognizing tracks, since it is only second order in the displacement of a bad hit from the real track, and it cannot be evaluated quickly. Instead we use the curvature of a link as the measure of distance as shown in fig. 13. Here links 1 and 2 belong to the real track, while 3 and 4 are derived from a bad hit. In this case, link 4 has a smaller radius or larger curvature than 1 and 2. In general, a chain formed with one or more bad hits will have some links with larger curvature than the real track. This implies that we should use the links of smallest curvature first in the search. The curvature or radius can be quickly evaluated in the limit of high momentum, using eq. (4) of
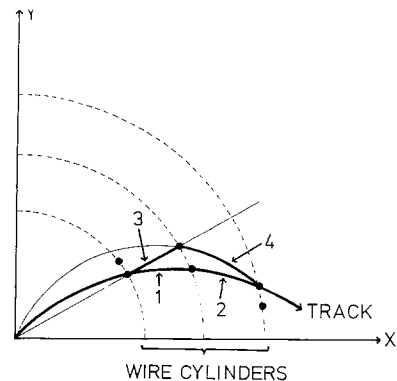

Fig. 13. Construction of curved 2-hit links with the help of the interaction point.

appendix B (with $D = 0$), which reduces to,

$$\phi = \phi_0 + \frac{Qr}{2R} .$$ (2)

Here ϕ is the angle of a hit in the layer of radius R, and ϕ_0 is the angle of the tangent to the track at the origin. For two hits i and j, our curvature variable κ is given by,

$$\kappa = \frac{1}{2R} = Q\frac{\phi_1 - \phi_j}{r_i - r_j} = Q\frac{\Delta\phi}{\Delta r} .$$ (3)

This expression is suitable, since it is first order in the displacements of the hits, has no signularities, and can be quickly evaluated *. (Note that the sign of Q is chosen to make R and κ positive.) In our program we use uncorrected drift distances to compute the ϕ angles. However, if more accuracy is necessary, the angle β to the normal required for computing drift time corrections (see appendix B) can be rapidly approximated using these r and ϕ values.

The program starts by splitting the event into fragments in which the trackfinding takes place. The fragments are crude subdivisions of event constructed by associating hit $0°$ wires in nearby layers with similar ϕ angles, starting with wires in outer layers, and working inwards. The proceedure is then iterated, working alternately outwards and inwards until it converges. A fragment contains only one track if it is well isolated, but it will generally contain all of the tracks in a narrow jet.

Next the links of the fragment are constructed. For each hit wire in the fragment, its neighbours (i.e. wires in adjacent layers with similar ϕ angles) are found. If a wire has no neighbour in an adjacent layer, a jump to the next layer is allowed. Since the left–right ambiguity gives two hits for every wire, 2 neighbouring wires give 4 links. For each link with κ less than a maximum value κ_{max}, the curvature κ and the addresses of the two hits defining the link are stored in the link list.

The link list is then ordered according to increasing κ with help of the Heapsort sorting algorithm [10]. This ordering has three desirable properties:

1) Links belonging to high momentum tracks appear early in the list.

2) Links belonging to a given real track are near each other in the list.

3) Some of the links necessary to construct a long chain involving some bad hits appear later in the list than links from the corresponding real track.

(1) obviously biases the search towards finding the high momentum tracks in a jet first. (2) and (3) allow us to optimize the speed of our search, since links for a real track are gathered together near the beginning of the list, and nearly all long chains including bad hits require links appearing later in the list. Therefore, these chains need not appear if the program is properly organized.

Now the links are pulled out of the list one after another. The previous and next elementary trees are then constructed using the new link as the trunk, but only using links *earlier* in the list as branches. This new link is then used as the starting link to construct chains by climbing in both directions from it. Due to (2) and (3) the first long chain (e.g. with 5 links in our 9 $0°$ layers) found in this way is most likely the right track or a piece of it. If not all layers are crossed a request for a small number of additional links is made (i.e. 2–5). If the found chain is not extended by these links the search is stopped. The found chain (or chains if more than one appears at the same time) is then fitted and, if necessary, followed to the other layers. If the track satisfies a χ^2 cut it is accepted. If there are several such tracks, both length and χ^2 are used in selecting the best track.

Note that this way of finding tracks is influenced by the minimal spanning tree construction. However, there are important differences. The MST of a set of hits is a tree that connects *all* hits and which has a minimal weight (in our case it is a sum of curvatures κ's). Our "MST" is in general the chain with the minimum weight than spans all or nearly all layers. We do not construct a real MST since in our sets there are always many hits, due to the left–right ambiguity, which do not belong to any track.

After the track was found and accepted all hits from wires used in the track are calcelled from the active hit list, and all links formed with these hits are cancelled from the link list, to prevent their reappearence in subsequent tracks. Then all wires in the fragment are checked again to find ones which, after the cancellation, lost all of their direct neighbours. If there are any, the program tries to construct links spanning over two or three gaps for them. The search for further tracks in the fragment continues iteratively with the shortened link list until all of the high

---

* This expression is clearly useful only for cylindrical layers. If plane layers are used, so Cartesian coordinates are appropriate, the signed square of the curvature can be obtained quickly without special functions using dot products.

quality tracks through the origin are found. Hence the program not only acts as a fast event filter, but also provides the first approximation to a complete list of tracks.

If a chain with at least 4 links is not found by some maximum number of requests for new links (between 20 and 40), the search is stopped and the fragment is abandoned. This demand for rapidly growing chains obviously greatly influences the speed, and it again is possible only because (2) and (3) are true.

This organization is particularly suitable for the rapid rejection of background in the TASSO experiment as mentioned at the beginning of this chapter. The vast majority of background triggers are due to interactions of the e^+ or e^- beams with the beam pipe, so there are no tracks coming from the interaction point. From the point of view of the program, this implies that there are no tracks in the fragment. However, this is recognized long before the whole fragment is reconstructed since the search is already abandoned after 20–40 unsuccessful requests for links. In addition, this organisation automatically finds high momentum tracks with the maximum number of hits first. These are the best tracks (they have the highest measurement quality and the most important ones, so finding them first is essential to any strategy to resolve complicated events. Here it happens automatically.

Our pattern recognition program, called FOREST, was optimised as an event filter since every TASSO trigger event is sent through it. The average analysis time is 60 ms, including 20 ms for reading, writing and decoding. The TASSO trigger requires a minimum number of tracks (chosen to be between 2 and 5, depending on background condition) with $P_T > 320$ MeV or two back to back tracks (P_T denotes the transverse momentum relative to the beam axis). The efficiency is momentum dependent since we work in this program with the uncorrected space–drift time relation. The largest deviations from linearity occur at large entrance angles β (see appendix B and ref. 1) and therefore affect mainly the low-p_T particle tracks. The efficiency (per track) of FOREST was found to be:

40%   for $p_T < 250$ MeV/c ,

88%   for $p_T \doteq 250 - 750$ MeV/c ,

95–98%   for $p_T > 750$ MeV/c .

The decrease in the efficiency with p_T is not due to

the use of the approximate form of eq. (4) described above. The influence of this approximation is seen only below $p_T = 150$ MeV.

The above efficiencies were determined with hadronic events from one photon annihilation at 30 GeV. The effieiency of reconstructing Monte Carlo events of the same type produced with linear speca–drift time relations was found to be 93% for all kinds of track from the interaction points with $p_T > 150$ MeV.

Due to its high efficiency the program is able to reduce the ratio of physics to background events from $1 : 10^4$ to $10 : 1$, i.e. it produces an almost clean event sample. These events are further analysed with a considerably slowed road type program *, called MILL, to obtain the highest possible efficiency for all kinds of tracks. A slightly modified FOREST program, working with the corrected space–drift time relation is used also within the MILL program to provide the first approximation to a complete list of tracks. The MILL efficiency is around 97% for all kinds of track with $p_T > 100$ MeV.

6. Other tracks, other links and possible extensions of the method

In spite of the fact that in TASSO we so far use the link-and-tree method only to search for circles from the neighbourhood of the interaction point and for straight lines, this method is not restricted to these cases. As we have already mentioned, the generalization to searching for circles originating from arbitrary points, (e.g. decay vertices, secondary interactions etc.) is straightforward. To find the projection of this kind of track on the $r-\phi$ plane we have to work with links built from three hits instead of two. In the construction of elementary trees there are two a priori possibilities:

a) to relate links of approximately equal track parameters which have the first and the last hit in common,

b) to relate links of approximately equal track parameters which have the first *two* and the last *two* hits in common, as shown in fig. 14.

---

* Roads are used only in finding $r-\phi$ tracks, the track finding in the $s-z$ plane is done entirely with the strategy described in sect. 4. Of course, this strategy would be appropriate for the $r-\varphi$ plane in the MILL program, but MILL was developed first and we have not yet brought it up to date.
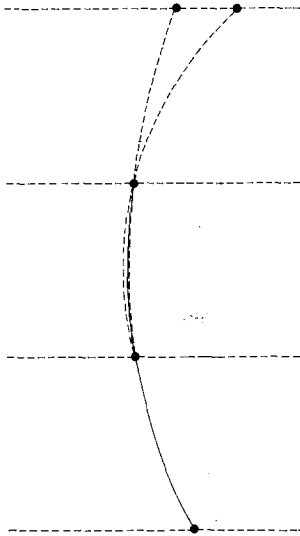
Fig. 14. An elementary tree constructed from three-hit links. The dashed circles are branches and the solid one is trunk.

We would recommend the second possibility since it introduces more information into the tree construction. Both track searching methods described in the previous two chapters can also be used with 3-hit links. The operation of the climbing algorithm is independent of the kind of links and elementary trees provided; however, the organization of link and elementary tree lists have to change since there are now more types of links.

The use of three hits instead of two enlarges the list of links considerably. To get a feel of how much the program processing would slow down, we make the following very rough estimate: The length of the link list is proportional to n^2 or n^3, for 2 or 3 hit links respectively, where n denotes the average number of hits per layer in the considered region. In our TASSO programs, n was typically 3 or 4. Since the execution time of the program is roughly proportional to the number of links used, we estimate that the 3-hit link program would also be a factor of 3—4 slower (assuming that the proportionality factor for n^3 is similar to that for n^2). This does not necessarily lead to a considerable increase in the total computing time, since the 3-hit link program would be used only for the small fraction of all triggers containing physics events.

It is also possible to generalize the link-and-tree methods to detectors that make a dense sequence of measurements on a track. In this case it makes no sense to create links from every pair or triplet of points; rather many points should be gathered into one link. However, care should be taken that also in the noisy regions of a chamber all links belonging to real tracks are created. This should happen even if jumps over large regions of a chamber are necessary since the links with big jumps correspond to multigap links in detectors with fewer layers as in the TASSO detector. In spite of the fact that in dense measurement chambers the number of hits is considerably larger than in chambers with fewer layers, the number of links created should be smaller since there is more information to eliminate wrong links. From the created set of links, it should be possible to construct every track as a chain of links without any interruption. Then all the methods described above apply.

Another straighforward generalization of the method is the case of a detector with an inhomogenious but slowly varying magnetic field. In this case links, elementary trees and full trees can be constructed in the same way as described above as long as the field is approximately constant in any elementary tree region. In our method the track parameters of the last link of a long chain can be quite different from those of the first one, since they can (slowly) dirft along the chain. The only parts of the program which have to be changed to adapt it to the effects of a nonhomogeneous magnetic field, are the track fits which, of course, being nonlocal, have to feel deviations from precise circles.

7. Conclusions

In this paper we have described a pattern recognition method based on the creation of lists of all track elements and local relations between them. The tracks are recognized as chain built from these elements. We have utilized two different applications of the method, a systematic search and a quick search. Both applications lead to efficient and fast programs. From our experience in working these programs we feel that it is posssible to write effective programs working with the link-and-tree method for reconstruction of all types of tracks.

Our method puts additional requirements on the performance of tracking chambers. The joining of links into elementary trees, which is a procedure of preselecting link combinations, requires good local precision (in contrast to a precision obtained by sampling many measurements). Also a considerable degree of homogeneity of the chamber (e.g. same

drift cell size throughout the chamber) is extremely valuable, since the tracks are recognized as chains of only locally related links. Local inhomogeneities (e.g. at the borders between different chamber types) tend to break the chains and in turn make track finding more difficult.

We developed this approach to track finding for the TASSO experiment and we gratefully acknowledge the numerous discussions with and the criticisms of our TASSO colleagues. We are particularly indebted to P. Söding and G. Wolf. We would also like to thank B. Radig and M.-J. Schachter for careful reading of the manuscript and useful comments.

Appendix A

The TASSO detector

The TASSO detector is illustrated in fig. 15. All of the pattern recognition work described here used data from the cylindrical inner drift chamber which is

described in more detail elsewhere [1]. Hits in the proportional chamber inside the drift chamber were added to tracks in later stages in the track reconstruction.

The cylindrical drift chamber is located inside the coil of a large solenoid magnet with a nearly uniform field of about 0.5 T parallel to the beam axis. The sensitive length of the chamber is 323 cm. It has 15 equally spaced layers, 9 zero-degree (0°) layers with the sense wires parallel to the axis and 6 stereo layers with the sense wires oriented approximately ±4° to the axis. Each 0° degree wire layer forms a cylinder whereas the twisted ones have a form of a hyperboloid. These layers are spaced 6.11 cm apart between an inner radius of 36.7 cm and an outer radius of 122.2 cm (measured at the ends of the chamber). There is a total of 2340 drift cells each with radial and azimuthal dimensions of 1.2 cm and 3.2 cm respectively. The cells are made of wires only; the repeating unit of a cell consists of 3 field wires and 1 sense wire.

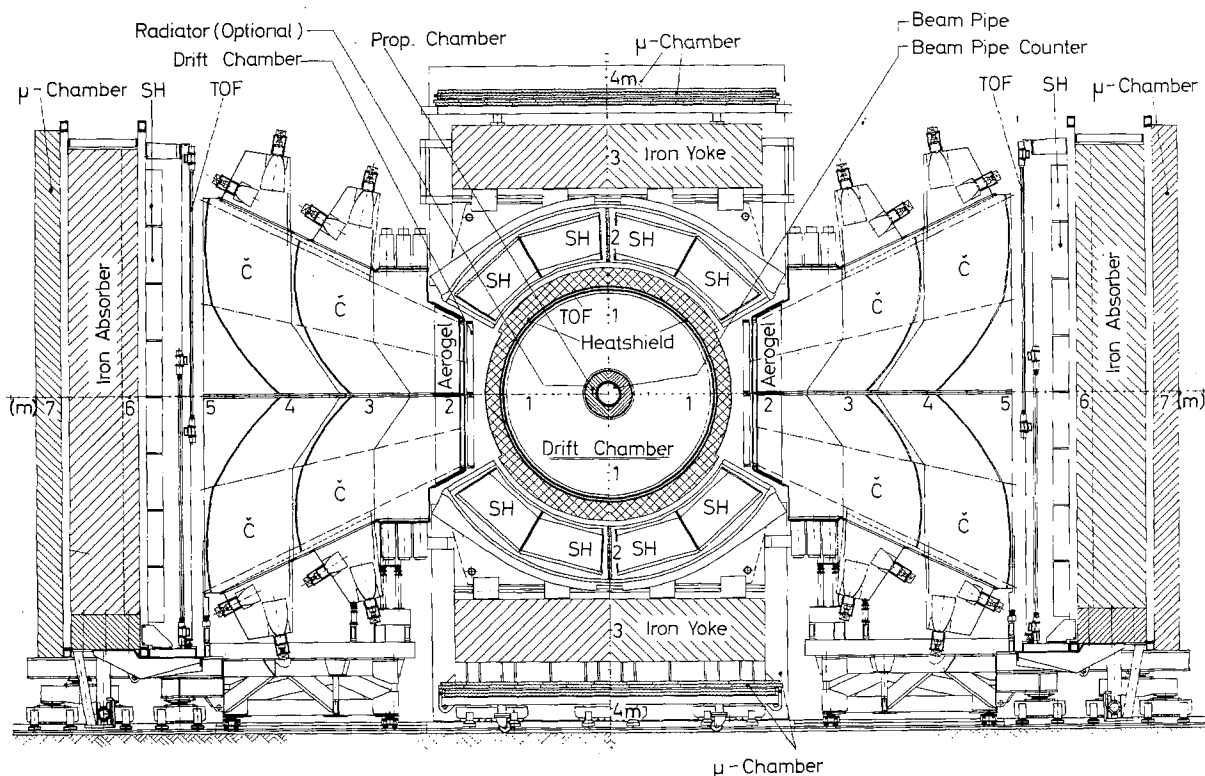The chamber was operated with a gas mixture of either 90% argon and 10% methane (or 50% A, 50%

Fig. 15. The TASSO detector viewed along the beam.

ethane). Its observed efficiency was 98% per wire including a 0.5% electronic inefficiency. The single hit electronics recorded the drift time of the track nearer to the sense wire if two tracks in an event crossed the same cell. No measurement of the longitudinal position of a hit on a single wire is made. The recorded drift time and wire address for each hit wire were passed on to the pattern recognition programs.

Appendix B

Geometry conventions and details of the track finding program

The input data for the pattern recognition program consists of the addresses and measured drift times for hit wires. Using an average drift velocity, the drift time for each wire is converted to an approximate drift distance d. This drift distance is assumed to be an arc length on the wire circle * and the angles ϕ_L and ϕ_R with respect to the x axis are calculated for the two possible hits to the left and right of the wire. These angles, the drift distances and bookkeeping information are stored in a common block for the track finding programs.

The first stage of the trackfinding uses only the $0°$ wires, yielding the projection of the track onto the $x-y$ or $r-\phi$ plane **. The magnetic field is sufficiently uniform to allow use to assume that the projected tracks are circles and the 3-dimensional track is a helix.

Fig. 16 illustrates the geometry of a circular track intersecting a cylinder of $0°$ wires. r is the radius of the wire cylinder and R is the radius of the track circle centered at (x_c, y_c). The track is at a distance D from the origin at its point (x_0, y_0) of closest approach and its angle with respect to the x-axis is ϕ_0 there. The track intersects the wire cylinder at angle ϕ with respect to the x-axis and it turned through an angle ψ in going between the point of closest approach and the wire cylinder. The track shown is a track with $Q = +1$ and $D \geqslant 0$. A track that does not enclose the origin has $D < 0$.

In terms of these quantities the angle ϕ is given by

$$\phi = \phi_0 + Q \arcsin\left[\frac{2RD - D^2 - r^2}{2r(R - D)}\right]. \tag{4}$$

The drift distance d, calculated with an average drift velocity, is only approximately correct. The actual drift distance depends not only on the drift time (ergo d) but also on the angle β between the normal to the wire cylinder and the track direction at that point. β is given by

$$\beta = \arcsin\left[\frac{r^2 + 2RD - D^2}{2rR}\right]. \tag{5}$$

After a track is found using only $0°$ wires, the drift distances are corrected using polynomials determined in an iterative procedure *. For simplicity we use different polynomials in different intervals of β to account for the dependence of the corrections upon β. A best fit track is then calculated using the corrected drift distances, always projected onto the wire circle.

After the projection of the track on the $r-\phi$ plane is found, a search for the corresponding projection in the plane including z axis can be started. If the dip angle of the 3-dimensional track is denoted by λ, the z dependence of the track is given by

$$z = z_0 = R\psi \tan \lambda = z_0 + s \tan \lambda, \tag{6}$$

where z_0 is the z coordinate of the track at (x_0, y_0). ψ is calculated from

$$\psi = 2 \arcsin\left|\frac{r^2 - D^2}{4R(R - D)}\right|^{1/2}. \tag{7}$$

The projection of the track on the $r-\phi$ plane defines a cylinder with its axis parallel to the z axis, as illustrated in fig. 17. The helix describing the 3-dimensional track must lie in this cylinder. The first step in the $s-z$ track finding is to calculate the z and $s = R\psi$ values for all stereo hits that could possibly lie on the track cylinder. The angle β for the track in the stereo layer is calculated using the radius of the layer at $z = 0$. The fully corrected drift distance for each hit wire is used to determine the ϕ angles of the left and right hits, again using the position of the wire at $z = 0$. For a single hit with angle ϕ_s this leads to the point (x_s, y_s) on the stereo wire circle of radius r_s at $z = 0$ shown in fig. 17. If the angle between the

---

* For stereo wires we used the circle at $z = 0$.

** The z axis is along the incident e$^+$ direction the y axis is vertical and the x axis completes a right-handed coordinate system.

* Since this expression is insensitive to D when D is small, the corrections can already be made earlier using the radius of curvature determined from the links as described in sect. 5.
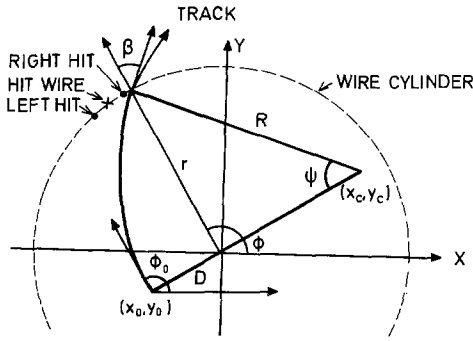
Fig. 16. Geometry of a circular track intersecting a cylinder of 0° wires.

stereo wires in this layer and the axis is α, the hit must lie on the line:

$$x = x_s + e_x z , \qquad y = y_s + e_y z , \tag{8}$$

with

$$e_x = -\tan \alpha \sin \phi_s , \qquad e_y = \tan \alpha \cos \phi_s . \tag{9}$$

If the hit belongs to the track it must lie on the hit cylinder, so the position of the hit in space must be

a)

projection in the r, ϕ plane

cross - section through the hyperboloid at Z = 0
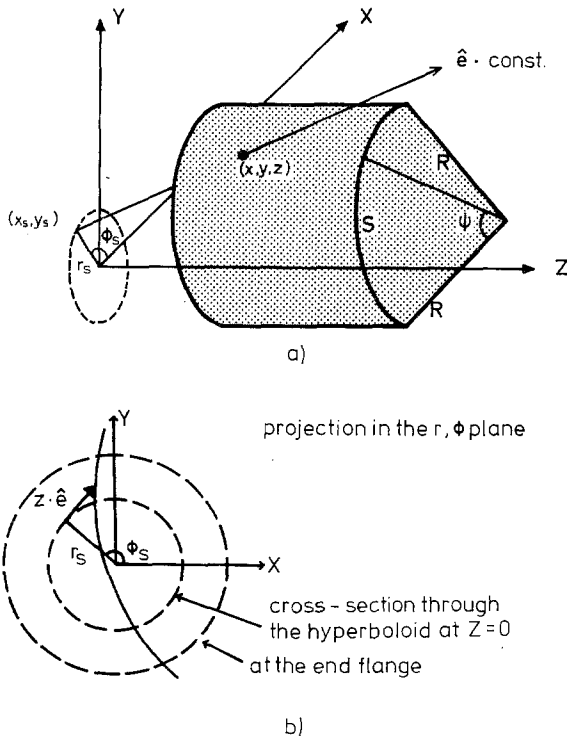
at the end flange

b)

Fig. 17. Geometry of a circular track intersecting a (not explicitly shown) hyperboloid of stereo wires. (a) Three-dimensional view; (b) Projection in the $r-\phi$ plane.

the intersection of the line given by eq. (8) with the cylinder given by

$$(x - x_c)^2 + (y - y_c)^2 - R^2 = 0 . \tag{10}$$

The z coordinate of the intersection is,

$$z = \frac{-2c}{b + (b^2 - 4ac)^{1/2}} , \tag{11}$$

where

$$a = \tan^2 \alpha , \qquad b = -2x_c e_x - 2y_c e_y , \tag{12}$$

$$c = (x_s - x_c)^2 + (y_s - y_c)^2 - R^2 .$$

The radius of the hit, r, is determined from z and eq. (8). Eq. (6) then gives ψ so the values of z and $s = R\psi$ are known for this hit if it belongs to the circular track. This calculation is repeated for all hits on all stereo wires that intersect the track cylinder. This gives a set of hits in the $s-z$ plane, each one associated with a stereo layer. According to eq. (6), z and s for a helix are linearly related, so the track finding problem in the third dimension reduces to finding straight tracks in the $s-z$ plane.

This calculation is too time consuming to actually use it on all possible stereo hits. Hits are first selected quickly using the uncorrected ϕ angle ϕ_S of the hit in the $z = 0$ plane. This is done for each stereo layer by fist calculating the angle ϕ_T of the intersection of the $r-\phi$ track with a circle of radius r_s representing the stereo layer at $z = 0$. A stereo hit in this layer is accepted if its angle ϕ_S is

$$|\phi_S - \phi_T| < \Delta\phi . \tag{13}$$

$\Delta\phi$ is the angular difference between the positions of the stereo wires in the given layer at $Z = 0$ and the end of the chamber, suitably enlarged for resolution and the missing corrections.

The s and z coordinates of each accepted hit and its address in the master hit list are stored in a $s-z$ hit list indexed with layer number and serial number or the accepted hit within the layer. The hits are ordered according to increasing z to optimize constructing the elementary trees. (With this ordering we can stop the search when the outer link has a larger slope than the inner link and fails the cut.)

References

[1] H. Boerner, H.M. Fischer, H. Hartmann, B. Löhr, M. Wollstadt, D.G. Cassel, U. Kötz, H. Kowalski, B.H. Wiik, R. Fohrmann and P. Schmüser, DESY 80/27.

[2] H. Kowalski, DESY 80/72.

[3] C.T. Zahn, Proc. Int. Computing Symp., Davos, (1973) (North-Holland , Amsterdam, 1974).

[4] H. Grote and P. Zanella, CERN — Data Handling Division, DD/80/11.

[5] F. Harary, Graph theory (Addison-Wesley, Reading, Mass.).

[6] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms (Addison-Wesley, Reading, Mass.) p. 176.

[7] T. Beyer, FLECS user's manual, Department of Computer Science, University of Oregon, Eugene, Oregon 97403 (1975).

[8] A.V. Aho, J.E. Hopcraft and J.D. Ullman, The design and analysis of computer algorithms (Addison-Wesley, Reading, Mass.) p. 44.

[9] A.V. Aho et al., op. cit., p. 173.

[10] A.V. Aho et al., op. cit., p. 87.