Transforming Lattices into Non-deterministic Automata with Optional Null Arcs

Abstract

The problem of transforming a lattice into a non-deterministic finite state automaton is non-trivial. We present a transformation algorithm which tracks, for each node of an automaton under construction, the larcs which it reflects and the lattice nodes at their origins and extremities. An extension of the algorithm permits the inclusion of null, or epsilon, arcs in the output automaton. The algorithm has been successfully applied to lattices derived from dictionaries, i.e. very large corpora of strings.

Introduction

Linguistic data -- grammars, speech recognition results, etc. -- are sometimes represented as lattices, and sometimes as equivalent finite state automata. While the transformation of automata into lattices is straightforward, we know of no algorithm in the current literature for transforming a lattice into a non-deterministic finite state automaton. (See e.g. Hopcroft *et al* (1979), Aho *et al* (1982).)

We describe such an algorithm here. Its main feature is the maintenance of complete records of the relationships between objects in the input lattice and their images on an automaton as these are added during transformation. An extension of the algorithm permits the inclusion of null, or epsilon, arcs in the output automaton.

The method we present is somewhat complex, but we have thus far been unable to discover a simpler one. One suggestion illustrates the difficulties: this proposal was simply to slide lattice node labels leftward onto their incoming arcs, and then, starting with the final lattice node, to merge nodes with identical outgoing arc sets. This strategy does successfully transform many lattices, but fails on lattices like this one:

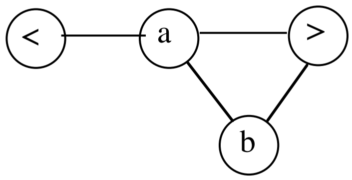


Figure 1

For this lattice, the sliding strategy fails to produce either of the following acceptable solutions. To produce the epsilon arc of 2a or the bifurcation of Figure 2b, more elaborate measures seem to be needed.

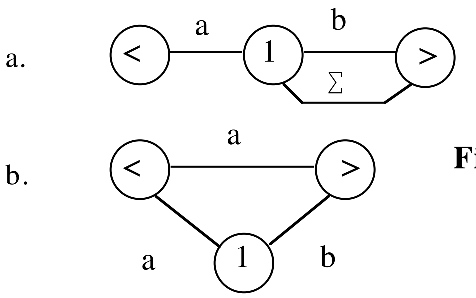


Figure 2

We present our datastructures in Section 1; our basic algorithm in Section 2; and the modifications which enable inclusion of epsilon automaton arcs in Section 3. Before concluding, we provide an extended example of the algorithm in operation in Section 4. Complete pseudocode and source code (in Common Lisp) are available from the authors.

1 Structures and terms

We begin with datastructures and terminology. A **lattice** structure contains lists of **lnodes** (lattice nodes), **larcs** (lattice arcs), and pointers to the **initial.lnode** and **final.lnode**. An **lnode** has a **label** and lists of **incoming.larcs** and **outgo-**

ing.larcs. It also has a list of **a-arcs** (automaton arcs) which *reflect* it. A **larc** has an **origin** and **extremity**. Similarly, an **automaton** structure has **anodes** (automaton nodes), **a-arcs**, and pointers to the **initial.anode** and **final.anode**. An **anode** has a **label**, a list of **larcs** which it reflects, and lists of **incoming.a-arcs** and **outgoing.a-arcs**. Finally, an **a-arc** has a pointer to its **lnode**, **origin**, **extremity**, and **label**.

We said that an anode has a pointer to the list of **larcs** which it reflects. However, as will be seen, we must also partition these **larcs** according to their shared origins and extremities in the lattice. For this purpose, we include the field **larc.origin.groups** in each **anode**. Its value is structured as follows: (((larc larc) lnode) ((larc larc) lnode) ...) Each *group* (sublist) within **larc.origin. groups** consists of (1) a list of larcs sharing an origin and (2) that origin lnode itself. Likewise, the **larc.extremity.groups** field partitions reflected larcs according to their shared extremities.

During lattice-to-automaton transformation, it is sometimes necessary to propose the *merging* of several anodes. The merged anode contains the union of the larcs reflected by the mergees. When merging, however, we must avoid the generation of strings not in the language of the input lattice, or *parasites*. An anode which would permit parasites is said to be *ill-formed*. An anode is ill-formed if any larc list in an origin group (that is, any list of reflected larcs sharing an origin) fails to intersect with the larc list of every extremity group (that is, with each list of reflected larcs sharing an extremity). Such an ill-formed anode would purport to be an image of lattice paths which do not in fact exist, thus giving rise to parasites.

2 The basic algorithm

We now describe our basic transformation procedures. Modifications permitting the creation of epsilon arcs will be discussed below.

Lattice.to.automaton, our top-level procedure, initializes two global variables and creates and initializes the new automaton. The variables are ***candidate.a-arcs*** (a-arcs created to represent the current lnode) and ***unconnectable.a-arcs*** (a-arcs which could not be connected when processing previous lnodes). During automaton initialization, an **initial.anode** is created and supplied with a full set of **larcs**: all outgoing larcs of the initial lnode are included. We then visit every lnode in the lattice in topological order, and for each lnode execute our central procedure, **handle.current.lnode**.

handle.current.lnode: This procedure creates an a-arc to represent the current lnode and connects it (and any pending a-arcs previously unconnectable) to the automaton under construction. We proceed as follows: (1) If current.lnode is the initial lattice node, do nothing and exit. (2) Otherwise, check whether any a-arcs remain on ***unconnectable.a-arcs*** from previous processing. If so, push them onto ***candidate.a-arcs***. (3) Create a candidate automaton arc, or **candidate.a-arc,** and push it onto ***candidate.a-arcs***.[1] (4) Loop until ***candidate.a-arcs*** is exhausted. On each loop, pop a **candidate.a-arc** and try to connect it to the automaton as follows: Seek potential **connecting.anodes** on the automaton. If none are found, push **candidate.a-arc** onto ***unconnectable.a-arcs***; otherwise, try to merge the set of **connecting.anodes**. (Whether or not the merge succeeds, the result will be an updated set of **connecting.anodes**.) Finally, execute **link.candidate** (below) to connect **candidate.a-arc** to **connecting.anodes**.

Two aspects of this procedure require clarification.

First, what is the criterion for seeking potential connecing.anodes for **candidate.a-arc**? These are nodes already on the automaton *whose reflected larcs intersect with those of the origin of candidate.a-arc*.

Second, what is the final criterion for the success or failure of an attempted merge among **connecting.anodes**? The resulting anode must not be *ill-formed* in the sense already outlined above. A good merge indicates that the a-arcs leading to the merged anode compose a legitimate set of common prefixes for **candidate.a-arc**.

link.candidate: The final procedure to be explained has the following purpose: Given a **candidate.a-arc** and its **connecting.anodes** (the anodes, already merged so far as possible, whose

[1] The new a-arc receives the label of the lnode which it reflects. Its origin points to all of that lnode's incoming larcs, and its extremity points to all of its outgoing larcs. **Larc.origin.groups** and **larc.extremity. groups** are computed for each new anode. None of the new automaton objects are entered on the automaton yet.

larcs *intersect* with the larcs of the a-arc origin), seek a final **connecting.anode**, an anode to which the **candidate.a-arc** can attach (see below). If there is no such anode. It will be necessary to *split* the **candidate.a-arc** using the procedure **split.a-arc**. If there is such an anode, a we connect to it, possibly after one or more application of **split.anode** to split the **connecting.anode**.

A **connecting.anode** is one whose reflected larcs are a *superset* of those of the **candidate.a-arc**'s origin. This condition assures that all of the lnodes to be reflected as incoming a-arcs of the connectable anode have outgoing larcs leading to the lnode to be reflected as **candidate.a-arc**.

Before stepping through the **link.candidate** procedure in detail, let us preview **split.a-arc** and **split.anode**, the subprocedures which split **candidate.a-arc** or **connecting.anodes**, and their significance.

split.a-arc: This subroutine is needed when (1) the origin of **candidate.a-arc** contains both initial and non-initial **larcs**, or (2) no **connecting.anode** can be found whose larcs were a superset of the larcs of the origin of **candidate.a-arc**. In either case, we must split the current **candidate.a-arc** into several new **candidate.a-arc**s, each of which can eventually connect to a **connecting.anode**. In preparation, we sort the larcs of the current **candidate.a-arc**'s origin according to the **connecting.anodes** which contain them. Each grouping of larcs then serves as the **larcs** set of the origin of a new **candidate.a-arc**, now guaranteed to (eventually) connect. We create and return these **candidate.a-arcs** in a list, to be pushed onto ***candidate.a-arcs***. The original **candidate.a-arc** is discarded.

split.anode. This subroutine splits **connecting.anode** when either (1) it contains both final and non-final **larcs** or (2) the attempted connection between the origin of **candidate.a-arc** and **connecting.anode** would give rise to an ill-formed anode. In case (1), we separate final from non-final **larcs**, and establish a new splittee anode for each partition. The splittee containing only non-final larcs becomes the **connecing.anode** for further processing. In case (2), some larc origin groups in the attempted merge do not intersect with all larc extremity groups. We separate the larcs in the non-intersecting origin groups from those in the intersecting origin groups and establish a splittee anode for each partition. The splittee with only intersecting origin groups can now be connected to **candidate.a-arc** with no further problems.

In either case, the original anode is discarded, and both splittees are (re)connected to the a-arcs of the automaton. (See available pseudocode for details.)

We now describe **link.candidate** in detail. The procedure is as follows: Test whether **connecting.anode** contains both initial and non-initial larcs; if so, using **split.a-arc**, we split **candidate.a-arc**, and push the splittees onto ***candidate.a-arcs***. Otherwise, seek a **connecting.anode** whose larcs are a superset of the **larcs** of the origin of **a-arc**. If there is none, then no connection is possible during the current procedure call. Split **candidate.a-arc**, push all splittee a-arcs onto ***candidate.a-arcs***, and exit. If there is a **connecting.anode**, then a connection can be made, possibly after one or more applications of **split.anode**. Check whether **connecting.anode** contains both final and non-final larcs. If not, no splitting will be necessary, so connect **candidate.a-arc** to **connecting.anode**. But if so, split **connecting.anode**, separating final from non-final larcs. The splitting procedure returns the splittee anode having only non-final larcs, and this anode becomes the **connecting.anode**. Now attempt to connect **candidate.a-arc** to **connecting.anode.** If the merged anode at the connection point would be ill-formed, then split **connecting.anode** (a second time, if necessary). In this case, **split.anode** returns a connectable anode as **connecting.anode**, and we connect **candidate.a-arc** to it.

A final detail in our description of **lattice.to.automaton** concerns the special handling of the final.lnode. For this last stage of the procedure, the subroutine which makes a new **candidate.a-arc** makes a *dummy* a-arc whose (real) origin is the **final.anode**. This anode is stocked with larcs reflecting all of the final larcs. The dummy **candidate.a-arc** can then be processed as usual. When its origin has been connected to the automaton, it becomes the **final.anode**, with all final a-arcs as its incoming a-arcs, and the automaton is complete.

3 Epsilon (null) transitions

The basic algorithm described thus far does not permit the creation of epsilon transitions, and thus yields automata which are not minimal. However, epsilon arcs can be enabled by varying the current procedure **split.a-arc**, which breaks an unconnectable **candidate.a-arc** into several eventually connectable a-arcs and pushes them onto ***candidate.a-arcs***.

In the splitting procedure described thus far, the a-arc is split by dividing its origin; its label and extremity are duplicated. In the variant (proposed by the third author) which enables epsilon a-arcs, however, if the *antecedence condition* (below) is verified for a given splittee a-arc, then its label is instead ∑ (epsilon); and its extremity instead contains the larcs of a sibling splittee's origin. This procedure insures that the sibling's origin will eventually connect with the epsilon a-arc's extremity. Splittee a-arcs with epsilon labels are placed at the top of the list pushed onto ***candidate.a-arcs*** to ensure that they will be connected before sibling splittees.

What is the antecedence condition? Recall that during the present tests for **split.a-arc**, we partition the a-arc's origin larcs. The antecedence condition obtains when one such larc partition is "antecedent" to another partition. Partition P1 is antecedent to P2 if every larc in P1 is antecedent to every larc in P2. And larc1 is antecedent to larc2 if, moving leftward in the lattice from larc2, one can arrive at an lnode where larc1 is an outgoing larc.

A final detail: the revised procedure can create duplicate epsilon a-arcs. We eliminate such redundancy at connection time: duplicate epsilon a-arcs are discarded, thus aborting the connection procedure.

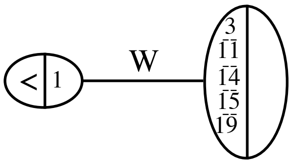
4 Extended example

We now step through an extended example showing the complete procedure in action. Several epsilon arcs will be formed.

We show anodes with their reflected **larcs**. We show **larc.origin.groups** on the left side of anodes when relevant, and **larc.extremity.groups** on the right.

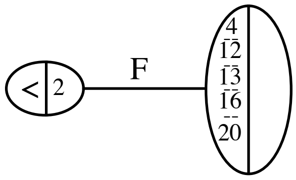
Consider the lattice of Arabic forms shown in Figure 3. After initializing a new automaton, we proceed as follows:

- Visit lnode W, constructing this **candidate.a-arc**:

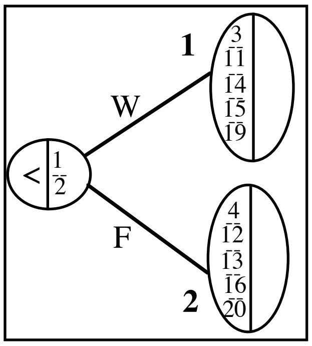


- The a-arc is connected to the initial anode.

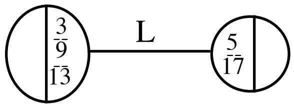
- Visit lnode F, constructing this **candidate.a-arc**:



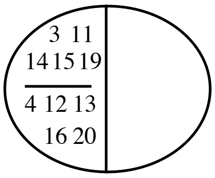
- The only **connecting.anode** is that containing the label of the initial lnode, >. After connection, we obtain:



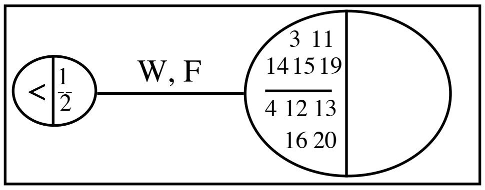
- Visit lnode L, constructing this **candidate.a-arc**:



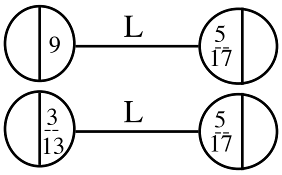
- Anodes 1 and 2 in the automaton are **connecting.anodes**. We try to merge them, and get:



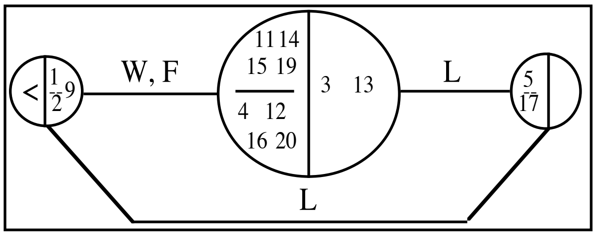
The tentative merged anode is well-formed, and the merge is completed. Thus, before connection, the automaton appears as follows. (For graphic economy, we show two a-arcs as a single a-arc with two labels.)



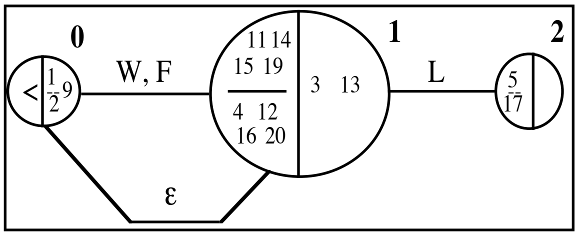
Now, in **link.candidate**, we split **candidate.a-arc** so as to separate the larcs of the **initial.anode** from other larcs. The split yields two **candidate.a-arcs**: one contains arc 9, since it arrives from the origin node; and the second contains the other arcs.



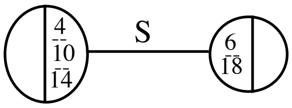
Following our *basic* procedure, the connection of these two arcs would give the following automaton:



However, the augmented procedure will instead create one epsilon and one labeled transition. Why? Our split separated larc 9 and larcs (3, 13) in the **candidate.a-arc**. But larc 9 is antecedent to larcs 3 and 13. So the splittee **candidate.a-arc** whose origin contains larc 9 becomes an epsilon a-arc, which connects to the automaton at the initial anode. The sibling splittee — the a-arc whose origin contains (3, 13) — is processed as usual. Because the epsilon a-arc's extremity was given the **larcs** of this sibling's origin, connection of the sibling will bring about a merge between that extremity and anode 1. The result is as follows:

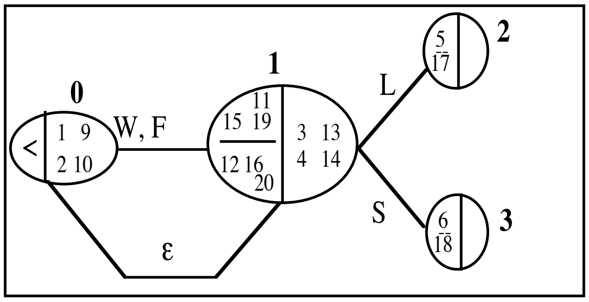


- Visit lnode S, constructing this **candidate.a-arc**:

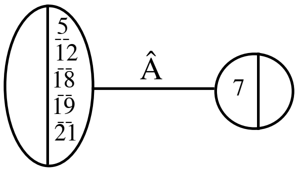


Anode 1 is the tentative connection point for the **candidate.a-arc**, since its larc set has the intersection (4, 14) with that of **candidate.a-arc**'s origin.

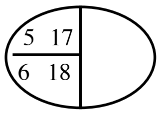
Once again, we split **candidate.a-arc**, since it contains larc 10, one of the **larcs** of the initial node. But larc 10 is an antecedent of arcs 4 and 14. We thus create an epsilon a-arc with larc 10 in its origin which would connect to the initial anode. Its extremity will contain larcs 4 and 14, and would again merge with anode 1 during the connection of the sibling splittee. However, the epsilon a-arc is recognized as redundant, and eliminated at connection time. The sibling a-arc labeled S connects, to anode 1, giving



- Visit lnode Â, constructing this **candidate.a-arc**

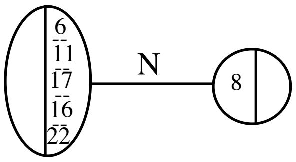


The two **connecting.anodes** for the **candidate.a-arc** are 2 and 3. Their merge succeeds, yielding:



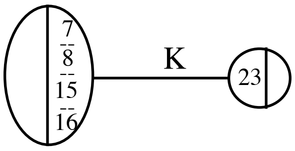
We now split the **candidate.a-arc**, since it finds no anode containing a superset of its origin's larcs: larcs (12, 19, 21) do not appear in the merged **connecting.anode**. Three splittee candidate automaton arcs are produced, with three larc sets in their origins: (5, 18), (12, 19), and (21). But larcs 12 and 19 are antecedents of larcs 5 and 18. Thus one of the splittees will become an epsilon a-arc which will, after all siblings have been connected, span from anode 1 to anode 2. And since (21) is also antecedent to (5, 18) a second sibling will become an epsilon a-arc from the initial anode to anode 2. The third sibling splittee connects to the same anode, giving Figure 4.

- Visit lnode N, constructing this **candidate.a-arc**:



The **connecting.anode** is anode 2. Once again, a split is required, since this anode does not contain arcs 11, 16, and 22. Again, three **candidate.a-arc**s are composed, with larc sets (6, 17), (11, 16) and (22). But the last two sets are antecedent to the first set. Two epsilon arcs would thus be created, but both already exist. After connection of the third sibling splittee, the automaton of Figure 4 is obtained.

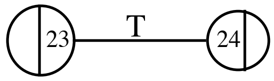
- Visit lnode K, constructing this **candidate.a-arc**:



We find and successfully merge **connecting.anodes** (3 and 4). For reasons already discussed, the **candidate.a-arc** is split into two siblings. The first, with an origin containing larcs (15, 16), will require our first application of **split.anode** to divide anode 1. The division is necessary because the connecting merge would be ill-formed, and connection would create the parasite path KTB. The split creates anode 4 (not shown) as the extremity of a new pair of a-arcs W, F -- a second a-arc pair departing the initial anode with this same label set.

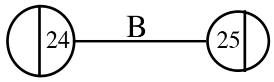
The second splittee larc contains in its origin state larcs 7 and 8. It connects to both anode 3 and anode 4, which successfully merge, giving the automaton of Figure 5.

- Visit lnode T, constructing this **candidate.a-arc**:



The arc connects to the automaton at anode 5.

- Visit lnode B, making this **candidate.a-arc**:



The arc connects to anode 6, giving the final automaton of Figure 6.

Conclusion and Plans

The algorithm for transforming lattices into non-deterministic finite state automata which we have presented here has been successfully applied to lattices derived from dictionaries, i.e. very large corpora of strings (Meddeb-Hamrouni (1996), pages 205-217).

Applications of the algorithm to the parsing of speech recognition results are also planned: lattices of phones or words produced by speech recognizers can be converted into initialized charts suitable for chart parsing.

References

Aho, A., J.E. Hopcroft, and J.D. Ullman. 1982. *Data Structures and Algorithms*. Addison-Wesley Publishing, 419 p.

Hopcroft, J.E. and J.D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing, 418 p.

Meddeb-Hamrouni, Boubaker. 1996. *Méthods et algorithmes de représentation et de compression de grands dictionnaires de formes*. Doctoral thesis, GETA, Laboratoire CLIPS, Féderation IMAG (UJF, CNRS, INPG), Université Joseph Fourier, Grenoble, France.

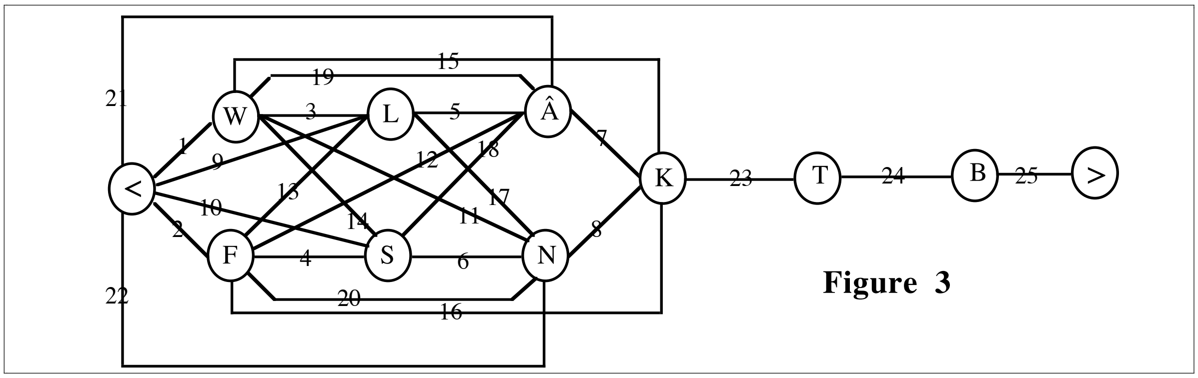


Figure 3

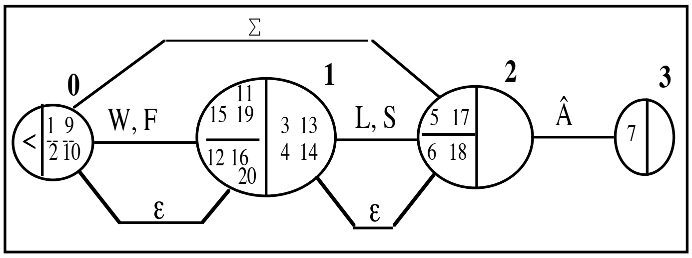


Figure 4

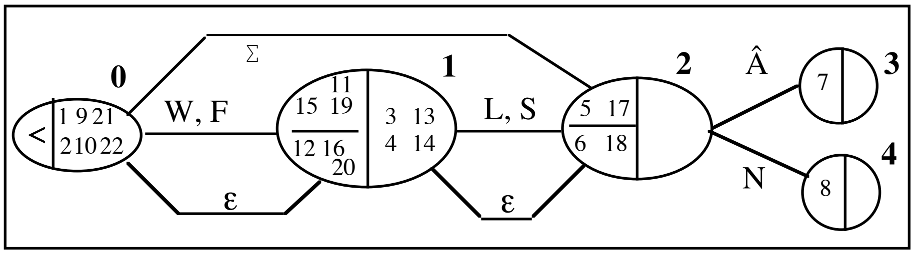


Figure 5

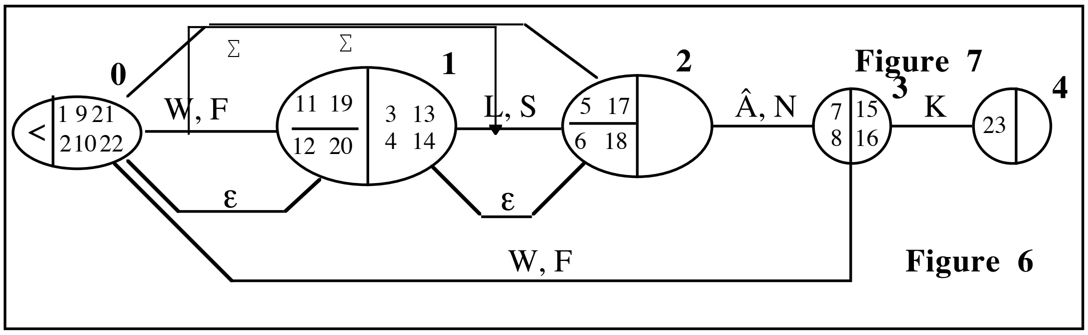


Figure 7

Figure 6