$O(N^3)$ Algorithm for Bisimulation Equivalence w.r.t CTL^* without the Next-Time Operator between Kripke Structures

Mahesh Girkar Robert Moll
Department of Computer Science
University of Massachusetts
Amherst MA 01003

CMPSCI Technical Report 93-79
October 1993

Abstract

Concurrent systems are often modeled by labeled state-transition graphs called Kripke Structures [5]. To reason about such systems, one standard approach is to provide a temporal semantic for the structure. Properties of interest regarding concurrency can then be expressed using a temporal logic formula. In this paper we consider the following problem: Given a Kripke structure, determine for all pairs of states s and s' whether they are equivalent with respect to CTL^*/X (*i.e.* s and s' model the same set of formulas in CTL^*/X). CTL^* [3], a new logic, incorporates the features of both Linear Temporal Logic and Branching Temporal Logic. In Logic CTL^*/X, the nexttime operator is discarded, since if this operator is used indiscriminately, it can lead to a "misleading" specification of a concurrent property [6, 5]. In [5], an $O(N^5)$ algorithm is presented for the all state pairs equivalence problem, where N the number of states in the structure. The primary contribution of this paper is an $O(N^3)$ algorithm for the same problem.

Concurrent systems are often modeled by labeled state-transition graphs called Kripke Structures [5]. To reason about such systems, one standard approach is to provide a temporal semantic for the structure. Properties of interest regarding concurrency can then be expressed using a temporal logic formula.

Linear-Time Temporal Logic (LTL) and Branching-Time Temporal Logic (BTL) are two widely used logics for expressing properties of a Kripke structure. In [3] a new logic, CTL^*, was introduced which incorporates the features of both LTL and BTL. This logic has both branching-time and linear-time operators: a path quantifier, either **A** ("for all paths") or **E** ("for some paths") can prefix an assertion composed of arbitrary combinations of the usual linear-time operators **G** ("always"), **F** ("sometimes"), **X** ("nexttime"), and **U** ("until").

However, in reasoning about concurrent systems, the use of the nexttime operator may be dangerous, since it refers to the "global" next state as opposed to the "local" next state within a process [6, 5]. The use of this operator essentially allows one to write a formula that can count the number of transition steps in a structure. Such a powerful operator is often not needed to express most properties of interest, such as reachability of a state within a structure, absence of deadlock, etc. Indeed, if this operator is used indiscriminately, it can lead to a "misleading" specification of a concurrent property [6]. Thus, analyzing properties of Kripke structures which are expressed without the usage of this operator is an important research topic in concurrent systems [7, 4, 5].

In this paper we consider the following problem: Given a Kripke structure, determine for all pairs of states s and s' whether they are equivalent with respect to CTL^*/**X** (*i.e.* s and s' model the same set of formulas in CTL^* without the nexttime operator **X**). In [5], an $O(N^5)$ algorithm is presented, where N the number of states in the structure. The primary contribution of this paper is an $O(N^3)$ algorithm for the same problem. Both algorithms can also be used to determine the equivalence for all pairs of states s and s' in two different Kripke structures M_1 and M_2 with respect to CTL^*/**X** by applying them to a new Kripke structure obtained by taking a disjoint union of M_1 and M_2. Here we present our algorithm for the all pairs equivalence problem within a single Kripke structure.

Section 1 discusses some of the preliminary definitions needed to formally describe the bisimulation problem. In Section 2, we present the $O(N^5)$ algorithm. This is followed in Section 3 and 4 by the $O(N^3)$ algorithm.

1 Preliminary Definitions

1.1 The logic CTL^*/X

In this section we describe formulas in CTL^*/**X**. Whenever possible, we use the notation developed in [5].

Let **AP** be a set of atomic proposition names. A formula in CTL^*/**X** is either a *state* formula or a *path* formula. A state formula is any formula obtained using the following rules:

1. A is a state formula if $A \in$ **AP**;
2. if f and g are state formulas, then $\neg f$ and $f \vee g$ are state formulas;
3. if f is path formula, the **E**(f) is a state formula.

A path formula is either

1. a state formula;
2. if f and g are path formulas, then $\neg f$, $f \vee g$, and f**U**g are path formulas.

CTL^*/**X** is the set of state formulas generated by the above rules.

1.2 Kripke Structure and Validity of a Formula

A Kripke Structure is a triplet $M = < S, R, \mathcal{L} >$ where

1. S is a finite set of states;
2. $R \subseteq S \times S$ is the transition relation, which must be total; and

3. $\mathcal{L} : S \to \mathcal{P}(\mathbf{AP})$, where $\mathcal{P}(\mathbf{AP})$ represents the power set of $\mathbf{AP}$.

The labeling function $\mathcal{L}$ associates with each state a (possibly empty) subset of $\mathbf{AP}$. Intuitively, such a subset indicates exactly those propositions that "hold" in that state. If it is empty, then we say that the constant *True* holds in that state.

Let there be N states in M denoted by $s_1, s_2, \ldots s_N$. We use symbols s, s', etc. to refer to any of these states. Note that M is essentially a directed graph, where the vertices correspond to the states and edges are provided by the relation R. The labeling function $\mathcal{L}$ imposes an additional semantic to this graph. We use standard graph terminology to refer to the transition relation in the Kripke structure. Thus, $s_i \to s_j$ indicates that $(s_1, s_2) \in R$.

A *path* from a state s in M is a sequence of states, $\pi = t_0, t_1, \ldots$ such that $t_0 = s$, and for every $i \geq 0, t_i \to t_{i+1}$. π^i will denote the suffix of π starting at state t_i. $M, s \models f$ means that f holds at state s in the structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along path π in M. When the Kripke structure is clear from context, we write $s \models f$ ($\pi \models f$) instead of $M, s \models f$ ($M, \pi \models f$).

Assuming below that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas, the relation $\models$ is inductively defined as follows:

1. $s \models A \Leftrightarrow A \in \mathbf{AP}$.
2. $s \models \neg f_1 \Leftrightarrow s \not\models f_1$.
3. $s \models f_1 \vee f_2 \Leftrightarrow s \models f_1$ or $s \models f_2$.
4. $s \models \mathbf{E}(g_1) \Leftrightarrow$ there exists a path π starting with s such that $\pi \models g_1$.
5. $\pi \models f_1 \Leftrightarrow s$ is the first state of π and $s \models f_1$.
6. $\pi \models \neg g_1 \Leftrightarrow \pi \not\models g_1$.
7. $\pi \models g_1 \vee g_2 \Leftrightarrow \pi \models g_1$ or $\pi \models g_2$.
8. $\pi \models g_1 \mathbf{U} g_2 \Leftrightarrow$ there exists a $k \geq 0$ such that $\pi^k \models g_2$ and for all $0 \leq j < k, \pi^j \models g_1$.

Note that $f \wedge g \equiv \neg(\neg f \vee \neg g)$, $\mathbf{F}(f) \equiv \mathit{True}\ \mathbf{U} f$, $\mathbf{A}(f) \equiv \neg\mathbf{E}(\neg f)$, and $\mathbf{G}(f) \equiv \neg\mathbf{F}(\neg f)$.

We now describe formally when two states are equivalent. Intuitively, two states are equivalent if and only if they model the same subset of formulas. The all pairs equivalence problem is the problem of determining for all pairs of states, if they are equivalent.

Definition 1 *Given a Kripke structure M, two states s_1 and s_2 of M are* equivalent *with respect to CTL^*/X* iff *for all state formulas $f \in CTL^*/X$,* $s_1 \models f \Leftrightarrow s_1 \models f$.

2 The Existing $O(N^5)$ Algorithm

In this section we describe the algorithm presented in [5] for the all pairs equivalence problem within a single Kripke structure. This algorithm uses the following relation C defined on $S \times S$.

Definition 2 $C = \bigcap_{j \geq 0} C_j$ *where*

1. $C_0 = \{(s, s') | \mathcal{L}(s) = \mathcal{L}(s')\}$;

2. Before defining C_{j+1} let

(a) $ST_{j+1}(s) = \bigcup_{d \geq 0} ST^d_{j+1}(s)$, *where*

i. $ST^0_{j+1}(s) = \{s\}$, *and*

ii. $ST^{d+1}_{j+1}(s) = ST^d_{j+1}(s) \bigcup$
$\{s' \mid s' \notin ST^d_{j+1}(s) \ \bigwedge\ \exists s'' \in ST^d_{j+1}(s)$ *such that* $s'' \to s' \ \bigwedge\ s' C_j s\}$

(b) $NEXT_{j+1}(s) = \{s' \mid s' \notin ST_{j+1}(s) \ \bigwedge\ \exists s'' \in ST_{j+1}(s)$ *such that* $s'' \to s'\}$.

(c)

$$LOOP_{j+1}(s) = \begin{cases} 1 & \textit{if there is a cycle containing only states in } ST_{j+1}(s). \\ 0 & \textit{otherwise} \end{cases}$$

Then,

$$\begin{aligned} C_{j+1} = \{(s,s')| \quad & LOOP_{j+1}(s) = LOOP_{j+1}(s') \ \wedge \\ & sC_js' \ \wedge \\ & \forall s_1 \in NEXT_{j+1}(s)\ \exists s_1' \in NEXT_{j+1}(s') \textit{ such that } s_1C_js_1' \ \wedge \\ & \forall s_1' \in NEXT_{j+1}(s')\ \exists s_1 \in NEXT_{j+1}(s) \textit{ such that } s_1C_js_1'\} \end{aligned}$$

If $s' \in ST_{j+1}(s)$, then the following two conditions hold: (a) sC_js': and (b) there exists a path from s to s' (using the transition relation R) such that for any state s'' on this path, $s''C_js$. An alternative way of interpreting $ST_{j+1}(s)$ is as follows: Viewing the Kripke structure as a graph (states are vertices and transitions are edges), consider its subgraph consisting of states that are C_j related to s. Then $ST_{j+1}(s)$ is the set of states that can be reached from s in this subgraph.

State $s' \in NEXT_{j+1}(s)$ when the following two conditions hold: (a) s' is not C_j related to s: and (b) s' is exactly one transition (step) away from from some state in $ST_{j+1}(s)$. Thus, $NEXT_{j+1}(s)$ is a "frontier" of states s' such that there is a path from s to s' with the following property: All states on this path except s' are C_j related to each other. We write $s\overline{C_j}s'$ if and only if s and s' are not C_j related. The following result is proved in [5]:

Theorem 1 *Let M be Kripke structure with N states. Then for any two states s and s' in M.* sC_Ns' iff $\forall f \in CTL^*/X \quad s \models f \Leftrightarrow s' \models f$.

Thus, one approach to solve the all pairs equivalence problem is to compute relation C_N for all pairs of states. In [5] the algorithm that computes C_N performs exactly the steps outlined in the definition of C_N above. Essentially, computing $ST_j(s)$, and hence $NEXT_j(s)$ requires time $O(N^3)$. Computing C_{j+1} from C_j requires time $O(N^4)$ since at most N^2 pairs of states needs to be checked and each pair requires $O(N^2)$ time to check. Computing C_N requires at most N outermost iterations. Thus, the algorithm takes $O(N^5)$ time.

3 The $O(N^3)$ algorithm

The algorithm presented in this paper also performs the outermost N iterations as in the previous algorithm. However, these iterations are performed on a "reduced" graph (denoted by $\mathcal{G}_{\text{reduced}}$). $\mathcal{G}_{\text{reduced}}$ is obtained from the original Kripke graph (denoted by $\mathcal{G}_{\text{original}}$) by "collapsing" certain states in $\mathcal{G}_{\text{original}}$ into a "node" of $\mathcal{G}_{\text{reduced}}$.

We define sets $STR_j(n)$, $NEXTR_j(n)$, boolean array $LOOPR_j(n)$, and relation CR_j for nodes of $\mathcal{G}_{\text{reduced}}$ (the definitions of these sets is similar to Definition 2 which described $ST_j(s)$, $NEXT_j(s)$, $LOOP_j(s)$, and C_j for states of $\mathcal{G}_{\text{original}}$) and establish a correspondence between pairs of equivalent nodes of $\mathcal{G}_{\text{reduced}}$ and pairs of equivalent states of $\mathcal{G}_{\text{original}}$. Using certain properties of these sets, we show that updating the sets described above can be performed "incrementally" at each outermost jth iteration in $O(N^2)$ time.

We begin with a description of the steps performed initially to collapse states into nodes.

3.1 Constructing $\mathcal{G}_{\text{reduced}}$

The procedure used to construct $\mathcal{G}_{\text{reduced}}$ performs the following steps:

Procedure Construct_Reduced_Graph

```
Begin
    (1) Partition states of $\mathcal{G}_{\text{original}}$ into equivalence classes
    using equivalence relation $C_0$. Let there be a total of $h$ classes.
    Denote the $i$th equivalence class by $\mathcal{E}_i$, $1 \le i \le h$.
```

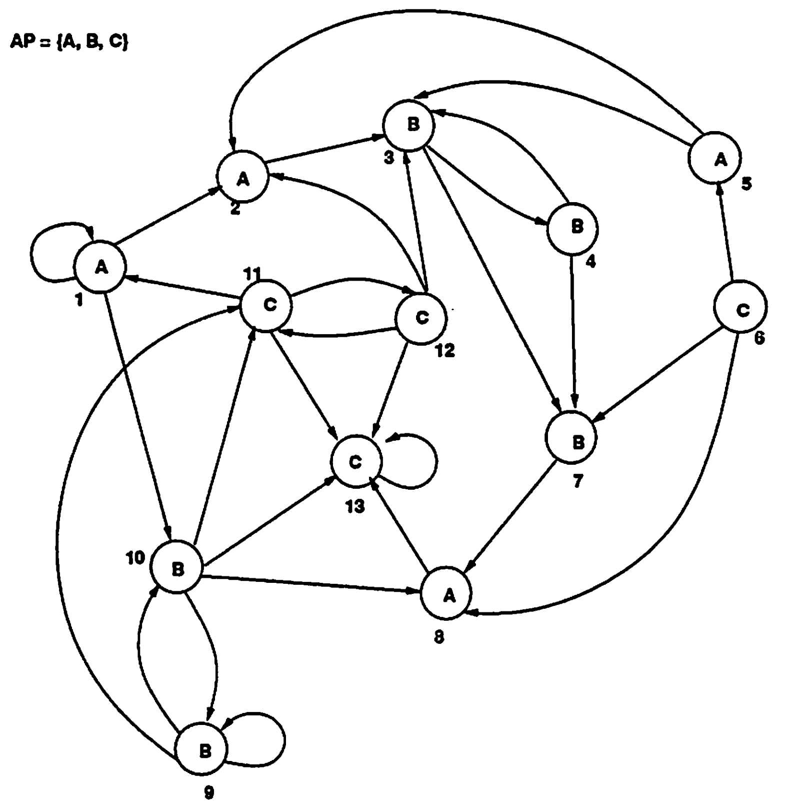

Figure 1: A Kripke Structure (13 states) with Propositional Symbols A, B, and C

```
(2) For each equivalence class $\mathcal{E}_i$:
    (2a) Obtain the subgraph of $\mathcal{G}_{\text{original}}(\mathcal{E}_i)$ induced by $\mathcal{E}_i$.
    (2b) Find the maximal strongly connected components of $\mathcal{G}_{\text{original}}(\mathcal{E}_i)$.
    Let $\mathcal{S}_{\text{original}}(\mathcal{E}_i)$ denote the set consisting of these components.
  endfor

(3) Let $\mathcal{S}\_\text{reduced} = \bigcup_{1 \leq i \leq h} \mathcal{S}_{\text{original}}(\mathcal{E}_i)$. Let there be $K$ elements in $\mathcal{S}\_\text{reduced}$ and
let us denote the $i$th element by $\mathcal{S}_i$. $\mathcal{G}_{\text{reduced}}$ has $K$ vertices.
The $i$th vertex, $n_i, 1 \leq i \leq K$ corresponds to $\mathcal{S}_i$.

(4) There is an edge from vertex $n_i$ to a vertex $n_j$ if and only if
  (a) $n_i \neq n_j$: and (b) there exists states $s \in \mathcal{S}_i$, $s' \in \mathcal{S}_j$ such that $s \rightarrow s'$.
endProcedure
```

Figure 1 shows a Kripke structure with 13 states over a propositional alphabet consisting of 3 symbols — A, B, and C. The labeling function $\mathcal{L}$ is a singleton set for each node and is shown within the circle representing the state.

Figure 2 shows $\mathcal{G}_{\text{reduced}}$ for the example in Figure 1. It has a total of 8 nodes corresponding to the 8 strongly connected components of the original graph. Notice that intra–state edges within a component have been eliminated from the original graph.

Let us assume that the adjacency list representation of $\mathcal{G}_{\text{original}}$ and a 2-d matrix representing the C_0 relation between states are provided as input.

Proposition 1 *The adjacency list/matrix representation of $\mathcal{G}_{reduced}$ can be constructed in $O(N^3)$ time.*

Proof: The equivalence classes can be obtained easily in $O(N^2)$ time. Each state can be labeled with an integer such that all states labeled with the same integer belong to the same equivalence class (in other words, these states are C_0 related to each other). Using this labeling, and, by traversing

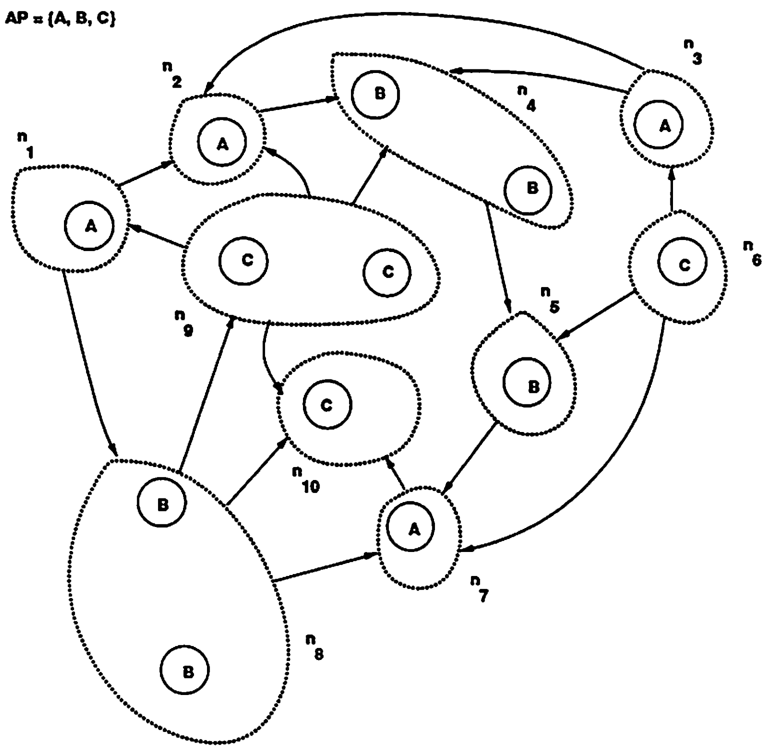

Figure 2: Reduced Graph

all the edges in the adjacency list representation of $\mathcal{G}_{\text{original}}$, it is straightforward to construct the adjacency list representation of the subgraph induced by vertices in any equivalence class in $O(N^2)$ time. Also, the standard algorithm for finding strongly connected components of a graph with N vertices takes $O(N^2)$ time [2]. Since there are a total of at most N equivalence classes, it will take $O(N^3)$ time to find all the vertices of $\mathcal{G}_{\text{reduced}}$. Each vertex of $\mathcal{G}_{\text{reduced}}$ is essentially a strongly connected component consisting of states of $\mathcal{G}_{\text{original}}$ which are C_0 related to each other. Each state can be labeled with an integer i such that all states with label i belong to the component (vertex) S_i. Now, by using this labeling, and by traversing all the edges in the adjacency list representation of $\mathcal{G}_{\text{original}}$, the adjacency matrix representation of $\mathcal{G}_{\text{reduced}}$ can be obtained in $O(N^2)$ time. Using this matrix, in $O(N^2)$ time, we can construct the adjacency list representation of $\mathcal{G}_{\text{reduced}}$. Thus, the total time is $O(N^3)$. □

The following proposition makes two observations: (1) states within a strongly connected component are equivalent; and (2) if two states s and s' are equivalent, then every pair of states s_1 and s_2 are equivalent, where s_1 and s belong to component S_i, and s_2 and s' belong to component S_j (i and j can be equal).

Proposition 2 *Let s, s' be any two distinct states in $S_i, 1 \leq i \leq K$. Then, $\forall j. sC_js'$. Further, suppose for some pair $(s, s') \in S_i \times S_j$, sC_js'. Then, for all pairs $(s, s') \in S_i \times S_j$, sC_js'.*

Proof: We prove the first part by induction. By construction of $\mathcal{G}_{\text{reduced}}$, if $s, s' \in S_i$, then sC_0s'. Suppose sC_ks' for some $k \geq 0$. Since S_i is strongly connected, the sets ST_{k+1}, $NEXT_{k+1}$, and $LOOP_{k+1}$ are identical for all states in S_i. Therefore, for all $s, s' \in S_i$, $sC_{k+1}s'$ (by Definition 2).

To prove the second part of the proposition, note that $\forall i, C_i$ is an equivalence relation [5]. Using this fact, and from the first part of this proposition, we get the required result. □.

3.2 Relation CR_j for nodes

We now define the relation CR_j for nodes of $\mathcal{G}_{\text{reduced}}$. Recall that each node n_i represents a set of states, denoted by $\mathcal{S}_i$, such that states in $\mathcal{S}_i$ are C_0 related to each other and $\mathcal{S}_i$ is a stongly connected component. Recall also that there are a total of N states in $\mathcal{G}_{\text{original}}$ and K nodes in $\mathcal{G}_{\text{reduced}}$. We use the symbols s, s', s'', etc. to refer to states, n, n', n'', etc. to refer to any one node, and $\mathcal{S}, \mathcal{S}', \mathcal{S}''$ etc. to refer to any one $\mathcal{S}_i$.

Definition 3 *Two nodes n_i and n_j are CR_0 related* iff *$\exists s \in \mathcal{S}_i$ and $\exists s' \in \mathcal{S}_j$ such that sC_0s'.*

Since all states in any $\mathcal{S}_i$ are C_0 related to each other (Proposition 2), it follows that if $n_i CR_0 n_j$ then for all pairs of states $(s, s') \in \mathcal{S}_i \times \mathcal{S}_j$, sC_0s'. In Figure 2, nodes n_1, n_2, n_3, and n_7 are CR_0 related to each other. So are nodes n_6, n_9, n_{10}, and n_4, n_5, n_8.

Reasoning about equivalence between nodes in $\mathcal{G}_{\text{reduced}}$ is closely related to the equivalence problem between states in $\mathcal{G}_{\text{original}}$. This is partially hinted at in Proposition 2. $\mathcal{G}_{\text{reduced}}$ is "nicer" than $\mathcal{G}_{\text{original}}$ because it does not have self-loops. In addition, the edges within a strongly connected component of $\mathcal{G}_{\text{original}}$ are not represented. However, since loops play an important role in the definition of C, we must keep track of loop structure. We do this by using the sizes of the strongly connected component.

Definition 4 *A strong connected component $\mathcal{S}$ has an "inherent" loop* iff *either*
(a) $|\mathcal{S}| = 1$ and the state in $\mathcal{S}$ has a self loop in $\mathcal{G}_{original}$; OR
(b) if $|\mathcal{S}| > 1$.

In Figure 2, $\mathcal{S}_1, \mathcal{S}_4, \mathcal{S}_9, \mathcal{S}_{10}, \mathcal{S}_8$ have inherent loops, where as the rest do not. We next define the relation CR between strongly connected components.

Definition 5 $CR = \bigcap_{j \geq 0} CR_j$ *where*

1. $CR_0 = \{(n, n') | \exists s, s', i, j,$ *such that* $n = n_i, n = n_j, s \in \mathcal{S}_i, s' \in \mathcal{S}_j,$ *and* $sC_0s'\}$;
2. *Before defining* CR_{j+1} *let*

   (a) $STR_{j+1}(n) = \bigcup_{d \geq 0} STR^d_{j+1}(n)$, *where*

      i. $STR^0_{j+1}(n) = \{n\}$, *and*

      ii. $STR^{d+1}_{j+1}(n) = STR^d_{j+1}(n) \bigcup$
      $\{n' \mid n' \notin STR^d_{j+1}(n) \ \bigwedge$
      $\exists n'' \in STR^d_{j+1}(n)$ *such that* $n'' \rightarrow n' \ \bigwedge$
      $n' \ CR_j \ n\}$

   (b) $NEXTR_{j+1}(n) = \{n' | n' \notin STR_{j+1}(n) \ \bigwedge$
   $\exists n'' \in STR_{j+1}(n)$ *such that* $n'' \rightarrow n'\}$.

   (c)

$$LOOPR_{j+1}(n) \begin{cases} 1 & \text{iff } \textit{either the strong component represented by } n \\ & \textit{has an inherent loop; or} \\ & \textit{for some node } n' \in STR_{j+1}(n), \\ & n' \textit{ has an inherent loop.} \\ 0 & \textit{otherwise} \end{cases}$$

Then,

$$\begin{aligned} CR_{j+1} = \{(n, n') | \ & LOOPR_{j+1}(n) = LOOPR_{j+1}(n') \ \bigwedge \\ & n \ CR_j \ n' \ \bigwedge \\ & \forall n'' \in NEXTR_{j+1}(n) \exists n''' \in NEXTR_{j+1}(n') \text{ such that } n'' \ CR_j \ n''' \ \bigwedge \\ & \forall n''' \in NEXTR_{j+1}(n') \exists n'' \in NEXTR_{j+1}(n) \text{ such that } n'' \ CR_j \ n'''\} \end{aligned}$$

We write $n\overline{CR_i}n'$ when n and n' are not CR_i related.

Essentially, $STR_{j+1}(n)$ is the set of nodes that can be reached from n in the subgraph of $\mathcal{G}_{\text{reduced}}$ induced by nodes that are CR_j related to n. Thus, if $n' \in STR_{j+1}(n)$, then there exists a path from n to n' in $\mathcal{G}_{\text{reduced}}$ such that all nodes on this path are CR_j related to each other.

If $n' \in NEXTR_{j+1}(n)$, then the following two conditions hold: (a) n' is not CR_j related to n; and (b) there exists a path from n to n' with the following property: All states on this path except n' are CR_j related to each other.

Proposition 3 *CR_j is an equivalence relation for all $j \geq 0$. Further, $CR_{j+1} \subseteq CR_j$.*

Proof: By Definition 3 CR_0 is an equivalence relation. By Definition 5, if CR_j is an equivalence relation, then so is CR_{j+1}. Using the same definition, it follows that $\mathrm{CR}_{j+1} \subseteq \mathrm{CR}_j$. □

Thus, CR_{j+1} is essentially a "refinement" of CR_j. The proposition below clarifies the correspondence between the sets in Definition 2 and those in Definition 5. The index j in this proposition corresponds to the index j used in Definitions 2 and 5. Indices i and p are used to refer to any two nodes n_i and n_p respectively.

Proposition 4 *Let state $s \in S_i$ (recall that n_i is the node that corresponds to component S_i). Then the following conditions hold:*

1. *$\forall j \geq 1$. $ST_j(s) = \bigcup_{S \in STR_j(n_i)} S$.*

2. *$\forall j \geq 1$. $LOOP_j(s) = LOOPR_j(n_i)$.*

3. *For any $j \geq 1$, let $s' \in NEXT_j(s)$. Then there exists n_p such that $n_p \in NEXTR_j(n_i)$ and $s' \in S_p$. Similarly, if $n_p \in NEXTR_j(n_i)$, then $\exists s'$ such that $s' \in NEXT_j(s)$ and $s' \in S_p$.*

4. *If $n_i CR_j n_p$, then for all pairs $(s, s') \in S_i \times S_p$, sC_js'. Further, if two states s, s' contained in distinct components S_i, S_p respectively are such that sC_js', then $n_i CR_j n_p$.*

Proof: (4) holds for the case when $j = 0$ (Definitions 2, 5, and 3). Using this fact, we will first prove that (1), (2), and (3) hold when $j = 1$.

To prove (1), suppose $s' \in ST_1(s)$. Then there is a path $t_1t_2\ldots t_v$ where $t_1 = s, t_v = s'$ and for $0 \leq x < v, t_xC_0t_{x+1}$. From the construction of $\mathcal{G}_{\text{reduced}}$, it should be clear that this path can be partitioned into $w > 0$ segments such that (i) each segment is a path consisting of states all of which are contained in a node of $\mathcal{G}_{\text{reduced}}$ and; (ii) the edge connecting two consecutive segments connects states in two different nodes. Let $r_1, r_2, \ldots r_w$ be the nodes representing the w segments. Clearly, there is a path $r_1r_2\ldots r_w$ in $\mathcal{G}_{\text{reduced}}$ where $r_1 = n_i$. Also, each of $r_1, \ldots r_w$ belong to $STR_1(n_i)$ and for $0 \leq x < w, r_x\ CR_0\ r_{x+1}$ (Definitions 5 and 3). Let S' be the set of states in node r_w. Obviously, $t_v \in S'$. Thus, $s' \in \bigcup_{S \in STR_1(n_i)} S$. In a similar fashion we can show that $\bigcup_{S \in STR_1(n_i)} S \subseteq ST_1(s)$.

To prove (2), suppose $LOOP_1(s) = 1$. Thus there is a path from state s such that contiguous states are C_0 related to each other and the path contains a loop. The states involved in the loop must belong to some node of $\mathcal{G}_{\text{reduced}}$. From arguments similar to those presented above, this node must belong to $STR_1(n_i)$ and it must have an "inherent" loop (see Definition 4). Thus $LOOPR_1(n_i) = 1$. The other direction follows in a similar fashion. (3) can also be proved using arguments similar to those used in proving (1) and (2).

Since (1), (2), and (3) hold, (4) holds for $j = 1$ (from Definitions 2, and 5). Finally, by repeating all of the above arguments for the case when $j = 2, 3, \ldots$, we can now show that (1), (2) and (3) hold for those cases. □

We occasionally use an alternate notation to indicate the containment conditions for the sets STR and $NEXTR$.

Definition 6 *We write $n \overset{j}{\rightsquigarrow} n'$ if and only if there is a path from n to n' such that all nodes on this path are CR_j related to each other (or in other words, $n' \in \mathrm{STR}_{j+1}(n)$). We write $n \overset{j}{\rightsquigarrow} n'' \rightarrow n'$ if and only if $n \overset{j}{\rightsquigarrow} n''$, $n'' \rightarrow n'$, and n' is not CR_j related to either n or n''. Note that if $n \overset{j}{\rightsquigarrow} n'' \rightarrow n'$ then $n' \in \mathrm{NEXTR}_{j+1}(n)$ and if $n' \in \mathrm{NEXTR}_{j+1}(n)$ then there exists node n'' such that $n \overset{j}{\rightsquigarrow} n'' \rightarrow n'$.*

3.3 Updating STR_j and $NEXTR_j$

The proposition below is critical, because it justifies an efficient "incremental" update procedure for the set STR_{j+1} from STR_j and relation CR_j.

Proposition 5 *For any $j \geq 1$,*

1. *if $n_p \in STR_j(n_i)$ then*
   (a) *$STR_j(n_p) \subseteq STR_j(n_i)$;*
   (b) *$NEXTR_j(n_p) \subseteq NEXTR_j(n_i)$; and*
   (c) *If $LOOPR_j(n_p) = 1$, then $LOOPR_j(n_i) = 1$.*
2. *$STR_{j+1}(n_i) \subseteq STR_j(n_i)$.*
3. *Let $r_1 r_2 \ldots r_y$ be any path in $\mathcal{G}_{reduced}$ where $r_1 = n_i$, and suppose all nodes on this path are CR_j related to each other. If r_1 and r_y are CR_{j+1} related then $r_1, r_2, r_3, \ldots, r_{y-1}$ are CR_{j+1} to each other.*
4. *$STR_{j+1}(n_i) = \{n | n \;\; CR_j \; n_i \;\; \bigwedge \;\; n \in STR_j(n_i)\}$.*

Proof: (1) is easy to prove. To prove (2), suppose $n \in STR_{j+1}(n_i)$. Then there is a path from n_i to n such that contiguous nodes are CR_j related to each other. By Definition 5, these nodes are also CR_{j-1} related to each other. Therefore, $n \in STR_j(n_i)$.

To prove (3), we need to show that for any node pair $(r_1, r_p), 2 \leq p < y$, the following observations (see Definition 5) hold: (i) : $r_1 \text{CR}_j r_p$; (ii) : $\text{LOOPR}_{j+1}(r_1) = \text{LOOPR}_{j+1}(r_p)$; (iii) : $\forall n'' \in \text{NEXTR}_{j+1}(r_1) \exists n''' \in \text{NEXTR}_{j+1}(r_p)$ such that $n'' \text{CR}_j n'''$; and (iv) : $\forall n''' \in \text{NEXTR}_{j+1}(r_p) \exists n'' \in \text{NEXTR}_{j+1}(r_1)$ such that $n'' \text{CR}_j n'''$.

Since r_1 and r_y are CR_{j+1} related (given), the above observations hold for the node pair (r_1, r_y). Observation (i) holds (follows trivially from the "antecedent" part of statement (3)). By 1(b) above, for all $p, 1 \leq p < y, NEXTR_{j+1}(r_{p+1}) \subseteq NEXTR_{j+1}(r_p)$. Since Observations (iii) and (iv) hold for node pair (r_1, r_y), and since for all $p, 1 \leq p < y, NEXTR_{j+1}(r_{p+1}) \subseteq NEXTR_{j+1}(r_p)$, it follows that for any node pair $(r_1, r_p), 2 \leq p \leq y$, Observations (iii) and (iv) hold. In a similar fashion, using 1(c) above, we can show that Observation (ii) holds.

Finally, from (2) and (3) above, (4) holds. □

The above proposition is important because it tells us that we need not recompute $STR_{j+1}(n)$ set from scratch. Instead, $STR_{j+1}(n)$ can be computed from $STR_j(n)$ by discarding from it those nodes that are not CR_j related to n. This proposition also provides intuition into the "refinement" process for the set of nodes $STR_j(n)$ in $\mathcal{G}_{\text{reduced}}$. The proposition below formalizes this intuition. (Recall that a DAG is a directed acyclic graph.)

Proposition 6 1. *The subgraph of $\mathcal{G}_{reduced}$ induced by nodes in some equivalence class obtained using the CR_j relation ($j \geq 0$) is a DAG.*

2. *For any node n and for all $j \geq 1$, the subgraph induced by nodes in the set $STR_j(n)$ is a DAG. Further, the undirected version of this subgraph is connected.*

Proof: For (1), when $j = 0$ the proof follows from the construction of $\mathcal{G}_{\text{reduced}}$. When $j \geq 1$, note that if two nodes are not CR_j related then they cannot be CR_{j+1} related (Definition 5). Finally, note that the subgraph of a DAG induced by any subset of its vertices is a DAG.

Similarly, to prove (2), note that for any node n, the subgraph induced by nodes in the set $STR_1(n)$ is a DAG (otherwise, the nodes involved in the cycle can be collapsed into a larger node, contradicting the fact that nodes are a maximal strongly connected set of states that are C_0 related to each other). Thus, the first part of statement (2) follows from statement (2) in Proposition 5. The second part above follows from statement (3) in Proposition 5. □

As an example, consider Figure 3. The set $STR_j(n_i)$ is shown as a DAG. All the nodes in this DAG are CR_{j-1} related to each other. Suppose now that node r_p is CR_j related to n_i. Then all nodes inside the dashed lines are also CR_j related to n_i. These nodes are ancestors of r_p and descendants of n_i. In this case, the set $STR_{j+1}(n_i)$ will contain all of these nodes along with r_p.

The next proposition provides insight into the set $NEXTR(n)$.

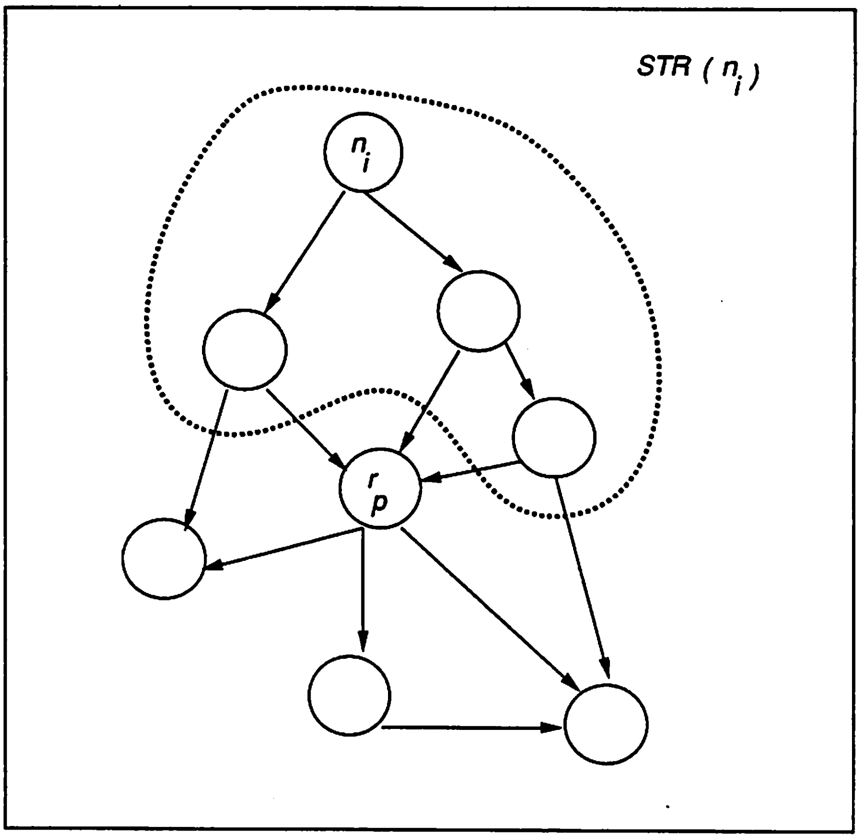

Figure 3: Updating $STR_j(n_i)$

Proposition 7 *For any $j \geq 1$, the following conditions hold:*

1. *Suppose $n_r \in NEXTR_j(n_i)$ but $n_r \notin NEXTR_{j+1}(n_i)$. Then for all $q \geq 1$, $n_r \notin NEXTR_{j+q}(n_i)$.*
2. *Suppose $n_r \in NEXTR_j(n_i)$ and $n_r \in NEXTR_{j+1}(n_i)$. Then for any node n_b, where $n_i \overset{j}{\leadsto} n_b \rightarrow n_r$, it follows that $n_i \overset{j-1}{\leadsto} n_b \rightarrow n_r$.*
3. *Suppose $n_r \notin NEXTR_j(n_i)$ and $n_r \in NEXTR_{j+1}(n_i)$. Then it must be the case that (a) $n_r \in STR_j(n_i)$; (b) $n_r \notin STR_{j+1}(n_i)$; and (c) there exists n_p such that $n_i \overset{j}{\leadsto} n_p \rightarrow n_r$. Similarly, if $n_r \in STR_j(n_i)$, $n_r \notin STR_{j+1}(n_i)$ and there exists n_p such that $n_i \overset{j}{\leadsto} n_p \rightarrow n_r$, then $n_r \notin NEXTR_j(n_i)$ but $n_r \in NEXTR_{j+1}(n_i)$.*

Proof: Statement (1) in the above proposition essentially says that once n_r fails to appear n_i's frontier list $NEXTR_{j+1}(n_i)$ after appearing in that list for the previous iteration (set $NEXTR_j(n_i)$), then it never appears in n_i's frontier list for any iteration number $j+1$ or greater. This follows from statement (2) of Proposition 5. Statement (2) follows from the fact that $CR_{j+1} \subseteq CR_j$. To prove (3), suppose $n_r \notin NEXTR_j(n_i)$ and $n_r \in NEXTR_{j+1}(n_i)$. Since $n_r \in NEXTR_{j+1}(n_i)$, there is a node n_b such that $n_i \overset{j}{\leadsto} n_b \rightarrow n_r$. Since CR_j is a refinement of CR_{j-1}, $n_i \overset{j-1}{\leadsto} n_b$. Since $n_r \notin NEXTR_j(n_i)$, it must be the case that $n_r \in STR_j(n_i)$. Since $n_r \in NEXTR_{j+1}(n_i)$, it follows that $n_r \notin STR_{j+1}(n_i)$, and that there exists n_p such that $n_i \overset{j}{\leadsto} n_p \rightarrow n_r$. The proof of (3) in the reverse direction is similar. □

We next provide an "implicit" way of computing the sets $NEXTR$.

Definition 7 *$WITNESS\_NEXTR_{j+1}(n_i, n_r)$ is the set obtained as follows: If $n_r \notin NEXTR_{j+1}(n_i)$, then $WITNESS\_NEXTR_{j+1}(n_i, n_r)$ is empty; otherwise, $n_b \in WITNESS\_NEXTR_{j+1}(n_i, n_r)$ if and only if $n_i \overset{j}{\leadsto} n_b \rightarrow n_r$.*

The following proposition provides the correspondence between the sets $WITNESS\_NEXTR$ and $NEXTR$. It also provides a method for updating $WITNESS\_NEXTR$ (statement (2) in the proposition below).

Proposition 8 *The following statements hold:*

1. *For any $j \geq 1$, $WITNESS\_NEXTR_j(n_i, n_r)$ is not empty if and only if $n_r \in NEXTR_j(n_i)$.*

2. *For any $j \geq 1$, if $WITNESS\_NEXTR_j(n_i, n_r)$ is not empty, then $WITNESS\_NEXTR_{j+1}(n_i, n_r) \subseteq WITNESS\_NEXTR_j(n_i, n_r)$. In this case, $WITNESS\_NEXTR_{j+1}(n_i, n_r) = \{n_b | n_b \in WITNESS\_NEXTR_j(n_i, n_r) \bigwedge n_b \in STR_{j+1}(n_i)\}$.*

3. *For any $j \geq 1$, if $WITNESS\_NEXTR_j(n_i, n_r)$ is empty, and $WITNESS\_NEXTR_{j+1}(n_i, n_r)$ is not empty, then $n_r \in STR_j(n_i)$. In this case, $WITNESS\_NEXTR_{j+1}(n_i, n_r) = \{n_b | n_b \in STR_{j+1}(n_i) \bigwedge n_b \rightarrow n_r\}$.*

Proof: (1) is trivial. The first part of statement (2) above follows from statement (2) of Proposition 7. To prove the second part of (2), consider first any node $n_b \in WITNESS\_NEXTR_{j+1}(n_i, n_r)$. From the first part of (2), $n_b \in WITNESS\_NEXTR_j(n_i, n_r)$. By Definition 7, $n_i \overset{j}{\rightsquigarrow} n_b \rightarrow n_r$, and $n_i \overset{j-1}{\rightsquigarrow} n_b \rightarrow n_r$. This implies that $n_b \in STR_j(n_i)$ and $n_b \in STR_{j+1}(n_i)$ (Definition 6). In the reverse direction, for the second part of statement (2), the proof is straightforward. The proof for the first part of statement (3) follows from statement (3) of Proposition 7. The proof for the second part of statement (3) is straightforward. □

From statement (1) in the above proposition we can ascertain $n_r \in NEXTR_j(n_i)$ using the cardinality of the set $WITNESS\_NEXTR_j(n_i, n_r)$. Statements (2) and (3) provide insight into the update procedure for the $WITNESS\_NEXTR_j$. The algorithm to follow keeps track of the cardinality of the set $WITNESS\_NEXTR$ at every iteration.

In summary, the relation CR_{j+1} is a "refinement" of relation CR_j. Further, the refinement process preserves certain important properties of the graph $\mathcal{G}_{\text{reduced}}$. One crucial property preserved is the following: The subgraph induced by nodes in $STR_j(n)$ for any node n is a DAG; Further, the undirected version of this DAG is connected (Proposition 6). Since $STR_{j+1}(n) \subseteq STR_j(n)$ (Proposition 5), it follows that the DAG representing the set $STR_{j+1}(n)$ is a subgraph of the DAG representing the set of nodes $STR_j(n)$; further the undirected version of the DAG representing the set $STR_{j+1}(n)$ is connected. Finally, the sets needed in computing CR_{j+1}, namely, $LOOP_{j+1}$, $NEXTR_{j+1}$, and STR_{j+1}, can be computed "incrementally" as suggested by Propositions 5 and 8.

4 Data Structures by the Algorithm

We are now ready to describe the algorithm. It performs K outermost iterations (K is the number of nodes in $\mathcal{G}_{\text{reduced}}$) in a fashion similar to the N outermost iterations in the $O(N^5)$ algorithm of Section 2. However, as we will see subsequently, the new algorithm takes at most $O(N^2)$ time for each of these iterations.

We assume that the initial input to the algorithm is the relation CR_0 between nodes (provided as a 2-d binary array) and an adjacency list as well as an adjacency matrix representation of $\mathcal{G}_{\text{reduced}}$. Throughout the description of the algorithm, the letter j is used to refer to the iteration number of the algorithm. Thus j ranges from 1 to K.

The first part (initialization steps) of the algorithm computes sets STR_1 and $NEXTR_1$, (refer to Definition 5). Then, at the $(j+1)$th iteration $(0 \leq j \leq K-1)$, it first computes $LOOPR_{j+1}$, and by using CR_j, STR_{j+1}, $NEXTR_{j+1}$, and $LOOPR_{j+1}$, it computes CR_{j+1}. Before performing the next iteration, it computes STR_{j+2} and $NEXTR_{j+2}$.

We use several arrays and lists to represent the sets described in the previous sections, and these are not calculated in the manner hinted at in Definition 5 (if we performed steps as suggested by Definition 5, then at any $(j+1)$st iteration, using CR_j we will compute STR_{j+1}, $NEXTR_{j+1}$ and $LOOPR_{j+1}$ and then CR_{j+1}). With the above description, we hope to have provided the reader with the sequence of steps that are performed by the algorithm to update the sets.

We first describe the arrays used by the algorithm and provide a brief interpretation of their entries. We also provide the names of the procedures that change or set the entries in these arrays.

In the next section, we will describe these procedures in detail. At that time we will describe certain invariant properties that hold for the entries in the arrays and establish the correspondence between these entries and the sets in Definition 5.

We use both 1-dimensional and 2-dimensional arrays. The indices of these arrays either refer to the nodes or their equivalence class number (Recall that for all $j \geq 0$, CR_j is an equivalence relation (Proposition 3)). For any CR_j, the maximum number of equivalence classes cannot exceed K, the total number of nodes. We use this fact and assign equivalence class numbers to nodes so that these do not exceed K.

For some of the arrays, we need to keep track of their values one iteration prior to the current iteration of the algorithm. We will use the convention that the letters "Prev" at the beginning of the name will be attached to the name of the array which contains the relevant values one iteration prior to the current outermost iteration. Similarly, the letters "Current" will be attached to the name of the array which contains the values for the current iteration.

- *Inherent_Loop*[1..K] : This is a binary array that realizes Definition 4. We will use this array to compute the array *Loop* (described below). The indices of this array represent nodes. The array is initialized as follows:

$$Inherent\_Loop[i] = \begin{cases} 1 & \text{if component } S_i \text{ for node } n_i \text{ has an inherent loop} \\ 0 & \text{otherwise} \end{cases}$$

This array is set once at the beginning and is never changed.

- *Algo_CR*[1..K][1..K] : *Algo_CR* is a binary 2-d array whose indices in both dimensions represent nodes. The entries are used to store the CR_j relation between nodes. Initially, the values in *Algo_CR* store the CR_0 relation.

$$Algo\_CR[i][p] = \begin{cases} 1 & \text{if } n_i\ CR_0\ n_p \\ 0 & \text{otherwise} \end{cases}$$

Algo_CR is updated by Procedure *Update_CR*.

- *Equiv_No*[1..K] : *Equiv_No* is a 1-d array whose indices represent nodes and whose entries contain equivalence class numbers for nodes. *Equiv_No* is set by procedure *Set_Equiv_No* using the array *Algo_CR*.

- *Current_STR*[1..K][1..K], *Prev_STR*[1..K][1..K] : These are binary 2-d arrays whose indices in both dimensions represent nodes. Their entries keep track of the *STR* sets (see Definition 5) for the current and the previous outermost iterations. Entries in *Current_STR* are set initially as follows:

$$Current\_STR[i][p] = \begin{cases} 1 & \text{if } n_p \in STR_1(n_i) \\ 0 & \text{otherwise} \end{cases}$$

This initialization can be performed in $O(N^3)$ time (see Proposition 9 below). *Current_STR* is updated by Procedure *Update_Current_STR*.

- *Loop*[1..K] : *Loop* is a 1-d binary array whose indices represent nodes. Its values represent the values of the set *LOOPR* in Definition 5 for the current iteration. *Loop* is set by Procedure *Set_Loop_Reduced* at every iteration using the values in arrays *Current_STR* and *Inherent_Loop*.

- *Next_Edge_Count*[1..K][1..K] : Entries in this array represent the cardinality of the set $WITNESS\_NEXTR$ for a pair of nodes (see Definition 7). *Next_Edge_Count* is initialized as follows:

$$Next\_Edge\_Count[i][r] = \begin{cases} q & \text{if there are } q \text{ nodes } n_{i_1}, n_{i_2}, \ldots, n_{i_q} \text{ such that} \\ & \text{for all } p, 1 \leq p \leq q, n_i \overset{0}{\sim} n_{i_p} \to n_r \\ 0 & \text{otherwise} \end{cases}$$

As discussed earlier, initial values for entries of *Current_STR* are a representation of the set STR_1, and those of *Algo_CR* are a representation of the relation CR_0. Thus, the initial value of the entry *Next_Edge_Count*[i][r] is non-zero if and only if $n_r \in NEXTR_1(n_i)$. In that case, this entry has a

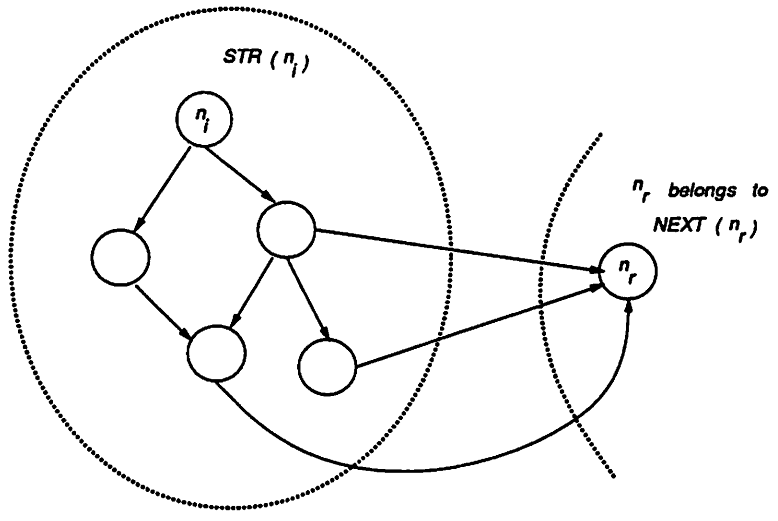

Figure 4: Illustration for entry in *Next_Edge_Count*

value equal to the number of nodes $n \in STR_1(n_i)$ with the property that $n \rightarrow n_r$. This array is updated by Procedure *Update_Next_Edge_Count*.

As an example, consider the scenario shown in Figure 4 at the first iteration. In this figure, we have shown the nodes in $STR_1(n_i)$ as a DAG (recall Proposition 6). Node $n_r \in NEXTR_1(n_i)$. The entry *Next_Edge_Count*[i][r] will be 3 since there are 3 nodes in $STR_1(n_i)$ which have direct edges to node n_r.

• *External*[1..K], *Internal*[1..K] : The indices of these two 1-d arrays correspond to nodes. Each entry, unlike the entries of the previous arrays, points to a linked list of indices. The initial linked list of indices for *Internal*[i] is as follows: index p belongs to list *Internal*[i] if and only if $n_i \rightarrow n_p$ and n_i CR_0 n_p. The initial linked list of indices for *External*[i] is as follows: index p belongs to list *External*[i] if and only if $n_i \rightarrow n_p$ and n_i $\overline{CR_0}$ n_p. These two arrays are updated by Procedure *Update_Internal_External*.

•*For_Next_CR*[1..K][2K + 1] : This binary array determines CR_{j+1} from CR_j, $LOOPR_{j+1}$, and $NEXTR_{j+1}$ (see Definition 5). It stores information from arrays CR_j, $LOOPR_{j+1}$, and $NEXTR_{j+1}$ in binary form. The rows represent the nodes. Suppose the current iteration is j, and consider row i, corresponding to node n_i. Then, the first column of *For_Next_CR* is a copy of the $LOOPR_j$ array (stored in our algorithm by the *Loop* array). The next K columns are used to represent the equivalence class numbers (array *Equiv_No* in our algorithm) assigned to the nodes. Since the number of equivalence classes for any CR_{j+1} relation will always be less than K, K columns are sufficient. If node n_i has equivalence class number p, then the $(i, p+1)$th entry is set to 1; otherwise, it is set to 0. The last K columns are used to represent the $NEXTR_j$ set for a node as follows: If $n_p \in NEXTR_j(n_i)$, then the $(i, p+1+K)$th entry is set to 1; otherwise, it is set to 0. This representation can be obtained from array *Next_Edge_Count* in our algorithm. The entries of *For_Next_CR* are set by Procedure *Set_For_Next_CR*.

Once *For_Next_CR* is constructed, it follows that if two rows in
For_Next_CR are identical, then the corresponding nodes are CR_{j+1} related (see Property 3 in the next section). To find rows that are identical, it suffices to "bucket" sort them and then compare entries in row i and row $(i+1)$ for all $i, 1 \leq i \leq K-1$. With some additional bookkeeping, it is then straightforward to find sets of nodes (equivalence classes for the next iteration) that are CR_{j+1} related to each other.

• *Next_Equiv_Node_List*[1..K] : Each entry in this array points to a list of nodes (possibly empty). If not empty, such a list contains nodes whose corresponding rows in the array *For_Next_CR* are identical. Procedure *Bucket_Sort_For_Next_CR* constructs the lists for the entries of this array. Using these lists, we can obtain *Algo_CR* (or the CR relation) for the next iteration easily (see Procedure

Set_Next_CR).

Figure 5 shows the values of those arrays whose initial values were defined (in their description above) for the reduced graph in Figure 2. The values shown in array
Equiv_No are the ones obtained by applying procedure *Set_Equiv_No* (described in the next section).

Proposition 9 *Initial values for Inherent_Loop, Current_STR, Next_Edge_Count, Algo_CR, and constructing the lists External, and Internal can be performed in $O(N^3)$ time.*

Proof: *Inherent_Loop* can be initialized in $O(N)$ time. *Algo_CR* can be initialized in $O(N^2)$ time (recall that CR_0 is provided as input in the form of a binary 2-d array – *Algo_CR* is initially identical to CR_0). *Current_STR*(i) can be computed by performing a depth first search from n_i in the reduced graph. This takes at most $O(N^2)$ time; hence for all nodes, *Current_STR* can be computed in $O(N^3)$ time. For each index i, its *External* and *Internal* list of edges can be computed in $O(N)$ time (we use the adjacency list representation of the graph, and the initial contents of array *Algo_CR*). Thus, for all nodes, this takes $O(N^2)$ time.

Next_Edge_Count can be computed from *Current_St* and *External* as follows: Initialize *Next_Edge_Count*$[g][h] := 0$, $1 \leq g, h \leq K$. Then, for each ordered pair (i, p) $(1 \leq i, p \leq K)$, check if *Current_STR*$[i][p] = 1$. If so, do the following: For each index $r \in$ *External*$[p]$, increment *Next_Edge_Count*$[i][r]$ by 1. Clearly, this algorithm takes at most $O(N^3)$ time. □

4.1 The Main Steps of the Algorithm

We now describe the steps performed by the algorithm (Procedure *Compute_CR_Relation* below) for each j'th iteration. Most steps are calls to procedures, some of which were mentioned in the previous section. For each procedure, we write the array(s) that it "uses" (or reads as input) and the array(s) that it modifies (or sets). Also, we write down a reference to an invariant property for the entries in the arrays (if one exists) after each of the steps. These properties will be stated later in the paper and they help in establishing the correspondence between entries of the arrays and the sets described in Definition 5.

```
Procedure Compute_CR_Relation:
Begin
  For $j, 1 \leq j \leq K$
    (1) Do Procedure Set_Equiv_No
    Uses: Algo_CR  Sets:Equiv_No
    Property 1 holds.

    (2) Do Procedure Set_Loop_Reduced
    Uses: Current_STR, Inherent_Loop  Sets:Loop
    Property 2 holds.

    (3) Do Procedure Set_For_Next_CR
    Uses: Loop, Equiv_No, Next_Edge_Count  Sets:For_Next_CR
    Property 3 holds.

    (4) Do Procedure Bucket_Sort_For_Next_CR
    Uses: For_Next_CR  Sets: Next_Equiv_Node_List
    Property 4.

    (5) Save Current_STR in Previous_STR

    (6) Do Procedure Set_Next_CR
    Uses: Next_Equiv_Node_List  Sets: Algo_CR
    Property 5 holds.

    (7) Do Procdeure Update_Current_STR
    Uses: Equiv_No, Previous_STR  Updates: Current_STR
```

CR [1..8] [1..8]

	1	2	3	4	5	6	7	8	9	10
1	1	1	1	0	0	0	1	0	0	0
2	1	1	1	0	0	0	1	0	0	0
3	1	1	1	0	0	0	1	0	0	0
4	0	0	0	1	1	0	0	1	0	0
5	0	0	0	1	1	0	0	1	0	0
6	0	0	0	0	0	1	0	0	1	1
7	1	1	1	0	0	0	1	0	0	0
8	0	0	0	1	1	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	1
10	0	0	0	0	0	1	0	0	1	1

Current_STR [1..8] [1..8]

	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0
3	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	1	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	1	0	0
9	0	0	0	0	0	0	0	0	1	1
10	0	0	0	0	0	0	0	0	0	1

Next_Edge_Count [1..8] [1..8]

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	1	0	0	0	1	0	0
2	0	0	0	1	0	0	0	0	0	0
3	0	0	0	2	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	1	0	0	0
6	0	0	1	0	1	0	1	0	0	0
7	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	1	0	1	1
9	1	1	0	1	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

Inherent_Loop [1..8]

1	2	3	4	5	6	7	8	9	10
1	0	0	1	0	0	0	1	1	1

Loop [1..8]

1	2	3	4	5	6	7	8	9	10
1	0	0	1	0	0	0	1	1	1

Equiv_No [1..8]

1	2	3	4	5	6	7	8	9	10
1	1	1	5	5	8	1	5	8	8

Internal [1..8]

1	2	3	4	5	6	7	8	9	10
2	E	2	5	E	E	E	E	10	E

E : indicates an empty list

External [1..8]

1	2	3	4	5	6	7	8	9	10
8	4	4	E	7	3	10	7	1	E
					5		10	2	
					7		9	4	
								10	

Figure 5: Initial Values of Arrays and Matrices

```
    Property 6 holds.

    (8) Do Procedure $Update\_Next\_Edge\_Count$
    Uses: $Next\_Edge\_Count$, $Current\_STR$, $Previous\_STR$, $External$  Updates: $Next\_Edge\_Count$
    Property 7 holds.

    (9) Do Procedure $Update\_External\_Internal$
    Uses: $Internal$, $External$, $Algo\_CR$  Updates: $Internal$, $External$.
    Property 8 holds.

  endfor
endProcedure
```

We now describe the steps for each of the procedures at steps (1)...(4) and (6)...(9) above. At the end of each description, we state invariant properties of the arrays with respect to the sets in Definition 5 that hold after the procedure is executed. Each property is stated for any $j, 1 \leq j \leq K$. In each case, the proof follows from earlier results and properties previously established.

The first procedure below sets the array *Equiv_No* using the array *Algo_CR*.

```
Procedure $Set\_Equiv\_No$
Variable $class\_no$ initialized to 1, and variable $count$
Array $Mark[i], 1 \leq i \leq K$ initialized to 0.
Begin
  For $i, 1 \leq i \leq K$
    if $Mark[i] = 0$ then
      $Mark[i] := 1$
      $Equiv\_No[i] := class\_no$
      $count := 1$
      For $p, 1 \leq p \leq K, p \neq i$
        if $Algo\_CR[i][p] = 1$ then
          $Mark[p] := 1$
          $Equiv\_No[p] := class\_no$
          $count := count + 1$
        endif
      endfor
      $class\_no := class\_no + count$
    endif
  endfor
endProcedure
```

Property 1 *Suppose that $Algo\_CR[i][p] = 1$ if and only if $n_i \ CR_{j-1} \ n_p$ prior to executing step (1) (Procedure Set_Equiv_No) of procedure Compute_CR_Relation. Then, after executing step (1) $Equiv\_No[i] = Equiv\_No[p]$ if and only if $Algo\_CR[i][p] = 1$; further, $Equiv\_No[i] \leq K$ for all $i, 1 \leq j \leq K$.*

The next procedure sets the *Loop* array.

```
Procedure $Set\_Loop\_Reduced$
Begin
  (1) For $i, 1 \leq i \leq K$
    $Loop[i] := 0$
    if $Inherent\_Loop[i] = 1$ then
      $Loop[i] := 1$
      goto next $i$ at step (1)
    endif
    for $p, 1 \leq p \leq K$
      if $Current\_STR[i][p] = 1$ and $Inherent\_Loop[p] = 1$
        $Loop[i] := 1$
        goto next $i$ at step (1)
    endif
  endfor
endProcedure
```

Property 2 *Prior to executing step (2) (Procedure Set_Loop_Reduced) of procedure Compute_CR_Relation, suppose that* $Current\_STR[i][p] = 1$ *if and only if* n_p $STR_j(n_i)$. *Then, after executing step (2) of procedure Compute_CR_Relation,* $Loop[i] = 1$ *if and only if* $LOOPR_j(n_i) = 1$.

Procedure *Set_For_Next_CR* sets entries in the array *For_Next_CR* to reflect the information in the arrays *Loop, Equiv_No,* and *Next_Edge_Count.* The first column is used to store the array *Loop.* Columns $2, 3, \ldots K + 1$ represent the equivalence class numbers assigned to nodes. If node n_i has a equivalence class number p, then we set $For\_Next\_CR[i][p+1]$ to 1; otherwise, entries in columns $2, 3, \ldots K + 1$ other than p for row i are set to 0. The last $K + 1$ columns represent (in a binary form) the equivalence class number of nodes in set $NEXTR_j(n_i)$ (see Definition 5). In our algorithm, $NEXTR_j(n_i)$ is represented implicitly by the array *Next_Edge_Count.*

```
Procedure Set_For_Next_CR
Begin
  Set all entries in For_Next_CR to 0.

  For $i, 1 \le i \le K$
    $For\_Next\_CR[i][1] := Loop[i]$
  endfor

  For $i, 1 \le i \le K$
    For $p, 1 \le p \le K$
      if $Equiv\_No[i] = p$ then
        $For\_Next\_CR[i][p+1] := 1$
      endif
    endfor
  endfor

  For $i, 1 \le i \le K$
    For $p, 1 \le p \le K$
      if $Next\_Edge\_Count[i][p] > 0$ then
        $For\_Next\_CR[i][K + 1 + Equiv\_No[p]] := 1$
      endif
    endfor
  endfor
endProcedure
```

Property 3 *Suppose that prior to executing step (3) (Procedure Set_For_Next_CR) of Compute_CR_Relation,* $Algo\_CR[i][p] = 1$ *if and only if* n_i CR_{j-1} n_p, $Loop[i] = 1$ *if and only if* $LOOPR_j(n_i) = 1$, *and* $Next\_Edge\_Count[i][p] > 0$ *if and only if* $n_p \in NEXTR_j(n_i)$. *Then, after executing step (3) of Compute_CR_Relation, for all* $m, 1 \le m \le 2K + 1$, $For\_Next\_CR[i][m] = For\_Next\_CR[t][m]$ *(i.e. rows i and t have identical entries in their corresponding columns in array For_Next_CR) if and only if* n_i CR_j n_t.

Figure 6 shows the *For_Next_CR* matrix for the first iteration in our example. The only two rows that are identical are rows 2 and 3. Thus, nodes n_2 and n_3 are CR_1 related.

Bucket_Sort_For_Next_CR performs the regular bucket sort procedure (see [2]) on the rows of the array For_Next_CR. The sorted permutation of row indices is stored in array *Sorted_Row_No.* Using this array, the lists for entries in *next_Equiv_Node_List* are set as described below. Briefly, each non-empty list of indices for any entry in *next_Equiv_Node_List* represents one equivalence class of nodes for the next iteration of the algorithm. Bucket Sort (step (1) of the procedure below) can be performed in $O(N^2)$ time. Creating the lists for entries in *Next_Equiv_Node_List* can also be performed in $O(N^2)$ time.

For_Next_CR [1..8] [1..21]

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
4	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1	0	0	0
7	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
9	1	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0

Loop

Equiv_No representation in a 1/0 form

Next_Edge_Count representation in a 1/0 form

Figure 6: Initial Values of Arrays and Matrices

```
Procedure $Bucket\_Sort\_For\_Next\_CR$
Variable $Sorted\_Row\_No[1 \ldots K]$.
Begin
   (1) Bucket sort the rows in $For\_Next\_CR$ (see [2]).
   Let $Sorted\_Row\_No[1 \ldots K]$ be the sorted permutation of row indices $1 \ldots K$.
   Thus, $Sorted\_Row\_No[p]$ represents the row index
   which is $p$th in the sorted list of rows.

   (2) For $i, 1 \leq i \leq K$
      Initialize $Next\_Equiv\_Node\_List[i]$ to the empty list.
   endfor

   (3) Using $Sorted\_Row\_No$ array, construct the lists for entries of
   $Next\_Equiv\_Node\_List$ as follows:
   • The first series of contiguous row indices in $Sorted\_Row\_No$
   representing identical rows are collected in a list.
   The pointer to be beginning of this list is stored in $Next\_Equiv\_Node\_List[1]$.
   • Similarly, the second series of contiguous row indices in $Sorted\_Row\_No$
   representing identical rows are collected in a list.
   The pointer to be beginning of this list is stored in $Next\_Equiv\_Node\_List[2]$, and so on.
   • The latter entries in $Next\_Equiv\_Node\_List$ may point to empty lists.
endProcedure
```

Property 4 *After executing step (4) (Procedure Bucket_Sort_For_Next_CR), if* Next_Equiv_Node_List[i], $1 \leq i \leq K$, *is not empty, then the nodes in that list are CR_j related to each other and they belong to one equivalence class of that relation.*

```
Procedure $Set\_Next\_CR$:
Begin
   Set all entries of $Algo\_CR$ to 0.

   For $i, 1 \leq i \leq K$
      $Algo\_CR[q][r] = 1$ for each ordered pair $(q, r)$
      where $q, r \in Next\_Equiv\_Node\_List[i]$
   endfor
endProcedure
```

Property 5 *After executing step (6) of procedure Compute_CR_Relation (Procedure Set_Next_CR), $Algo\_CR[q][r] = 1$ if and only if n_q CR_j n_r.*

```
Procedure $Update\_Current\_STR$
Begin
   For $i, 1 \leq i \leq K$
      For $p, 1 \leq p \leq K$
         if $Algo\_CR[i][p] = 1$ and $Previous\_STR[i][p] = 1$ then
            $Current\_STR[i][p] = 1$
         else
            $Current\_STR[i][p] = 0$
         endif
      endfor
   endfor
endProcedure
```

The proof of the following property follows from Proposition 5.

Property 6 *After executing step (7) (Procedure Update_Current_STR), $Current\_STR[i][p] = 1$ if and only if $n_p \in STR_{j+1}(n_i)$.*

```
Procedure Update_Next_Edge_Count
Begin
  For $p, 1 \leq p \leq K$
    Build list of indices (pointed to by) Inverse_Current_STR[p] such that
    i belongs to the list Inverse_Current_STR[p] if and only if
    Current_STR[i][p] = 1
  endfor

  For $i, 1 \leq i \leq K$
    For $p, i+1 \leq p \leq K$
      Case 1: Previous_STR[i][p] = 1 $\bigwedge$ Current_STR[i][p] = 0
        For each node t in External[p]
          Decrement Next_Neighbor_Edge_Count[i][t] by 1
        endfor
        if $n_i \rightarrow n_p$ then
          for each node t in list Inverse_Current_STR[i]
            Increment Next_Neighbor_Edge_Count[t][j] by 1.
          endfor
        endif
      Case 2: Previous_STR[p][i] = 1 $\bigwedge$ Current_STR[p][i] = 0
        Case handled as in Case 1 above with i and p interchanged
      Otherwise:
        do nothing
      endCase
    endfor
  endfor
endProcedure
```

The proof of the following property follows from the arguments presented for Proposition 8.

Property 7 *After executing step (8) (Procedure Update_Next_Edge_Count), the following condition is true:*

$$\text{Next\_Edge\_Count}[i][r] = \begin{cases} q(q>0) & \text{if there are } q \text{ nodes } n_{i_1}, n_{i_2}, \ldots, n_{i_q} \text{ such that} \\ & \text{for all } p, 1 \leq p \leq q, n_{i_p} \in \text{STR}_{j+1}(n_i), \\ & n_r \in \text{NEXTR}_{j+1}(n_i), \text{ and } n_{i_p} \rightarrow n_r \\ 0 & \text{otherwise} \end{cases}$$

```
Procedure Update_External_Internal
Begin
  For $i, 1 \leq i \leq K$
    For each p in list Internal[i]
      if Algo_CR[i][p] = 0
        Remove p from Internal[i]
        Add p to External[i]
      endif
    endfor
  endfor
endProcedure
```

Property 8 *After executing step (9) of procedure Compute_CR_Relation (Procedure Update_External_Internal) q belongs to list External[i](Internal[i]), if and only if $n_i \rightarrow n_q$ and $n_i\ \overline{CR_j}(CR_j)n_q$.*

Proposition 10 *Procedure Compute_CR_Relation is correct.*

Proof: From Properties 1 thru 8. □

4.2 Analysis of Procedure *Compute_CR_Relation*

Proposition 11 *All the procedures except Update_Next_Edge_Count take at most $O(N^2)$ time.*

It seems as if Procedure Update_Next_Neighbor_Edge_Count will take $O(N^3)$ time, since it has potentially 3 loops, one inside the other, each performing at most N iterations. However, we will show that the total time spent over the entire duration of the algorithm in performing steps outlined for Case 1 and Case 2 of that procedure does not exceed $O(N^3)$.

Proposition 12 *Procedure Update_Next_Edge_Count take at most $O(N^3)$ time over the entire duration of (or, over all the outermost iterations of) Procedure Compute_CR_Relation.*

Proof: We keep track of the total number of increments or decrements that are performed over all entries of Next_Neighbor_Edge_Count (the steps performed for Case 1 and 2 in Procedure Update_Next_Neighbor_Edge_Count). Note that using the adjacency matrix representation for $\mathcal{G}_{\text{reduced}}$, Conditions for Case 1, and 2 in Update_Next_Edge_Count, as well as the condition $n_i \rightarrow n_j$ for any two nodes can be done in constant time. Now consider a fictitious 3-d matrix *Next_Edge_Map*$[1 \ldots K][1 \ldots K][1 \ldots K]$ with boolean entries such that for any jth iteration of the outermost loop.
Next_Edge_Map$[i][p][r] = 1$ *iff* $n_i \overset{j-1}{\sim} n_p \rightarrow n_r$. Now, over the entire duration of the algorithm, one of two things can happen: (a) an entry Next_Edge_Map$[i][p][r]$ turns from 0 to 1, or it turns from 1 to 0. Each time the entry *Next_Edge_Map*$[i][p][r]$ turns from 0 to 1, entry *Next_Edge_Count*$[i][r]$ is incremented by 1. If that entry turns from 1 to 0, then entry *Next_Edge_Count*$[i][r]$ is decremented by 1. Note that when an entry changes from 1 to 0, it cannot change back to 1. This is because, when node n_p is removed from $STR_j(n_i)$'s set, it cannot become a member of STR(n_i)'s set for any of the subsequent outer iterations, by Proposition 5. This implies that the total number of increment and decrement operations that can be performed within the 2 cases in Procedure Update_Next_Neighbor_Edge_Count cannot exceed $O(N^3)$. □

Theorem 2 *Procedure Compute_CR_Relation runs in $O(N^3)$ time. It computes the relation* CR *correctly.*

Proof: From Propositions 10, 11, and 12.□

5 Conclusion

We have shown an $O(N^3)$ algorithm for determining the all pairs equivalence problem for a Kripke structure with respect to nexttime-less concurrent tree logic ($CTL^* /$ X). Clearly, this algorithm can also be used to determine the equivalence problem for pairs of states in two distinct Kripke structures by creating one Kripke structure from a disjoint union of these two.

References

[1] Antii Valmari and Matthew Clegg. Reduced Labelled Transition Systems Save Verification Effort. *CONCUR*, 1991.

[2] Corman, Liecerson, and Rivest. *Introduction to Algorithms*. 1990.

[3] E. M. Clarke, Emerson, and Sistla. Automatic Verification Of Finite-state Concurrent Systems Using Temporal Logic Specifications. *10th ACM Symposium on Principles of Programming Languages*, 1983.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing Kripke Structures in Temporal Logic. *TAPSOFT '87, LNCS 249, Volume 1*, 1987.

[5] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 1988.

[6] M. Lamport. What good is Temporal Logic ? *Proc. International Federation for Information Processing*, 1983.

[7] R. Kaivola and A. Valmari. Using Truth-Preserving Reductions to Improve the Clarity of Kripke-models. *CONCUR*, 1991.

[8] R. Kaivola and A. Valmari. The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic. *CONCUR*, 1992.